Nagy hálózatok előfeldolgozása gyorsabb útvonalkereséshez

Szakdolgozat

Írta: Góbor Dániel

Matematika BSc.
alkalmazott matematikus szakirány

Témavezető:

Király Zoltán, egyetemi docens

Számítógéptudományi Tanszék
Eötvös Loránd Tudományegyetem, Természettudományi Kar

Eötvös Loránd Tudományegyetem
Természettudományi Kar
2013

Tartalomjegyzék

1. Történeti áttekintés **5**

1.1. Dijkstra algoritmusa 5

1.1.1. Bináris kupac adatszerkezet 5

1.1.2. Az algoritmus inicializációja 6

1.1.3. Futás 7

1.1.4. Leállás 7

1.1.5. Kétirányú Dijkstra algoritmus 7

1.2. A* algoritmus 8

1.3. Gráfok előfeldolgozását igénylő módszerek 8

1.3.1. Autópálya hierarchiák 8

1.3.2. Transit node 9

1.3.3. Highway Node Routing 10

1.3.4. Összehúzási Hierarchiák 10

2. Összehúzási hierarchiák **11**

2.1. Definíciók 11

2.2. Motiváció 11

2.2.1. Nehézségek a feladat feldolgozása során 12

2.3. Csúcsok összehúzása 13

2.4. Csúcsok sorba rendezése 13

2.4.1. Prioritások frissítése 14

2.4.2. Prioritási tényezők: élkülönbség 14

2.4.3. Prioritási tényezők: egyenletesség 15

2.4.4. Prioritási tényezők: egyéb 15

2.4.5. Prioritási tényezők kombinációja 16

2.5. Legrövidebb út keresése a feldolgozott gráfban 16

2.5.1. Lehetséges javítások 19

2.5.2. Az utak visszakeresése 19

3. Az implementáció **20**

3.1. LEMON 20

3.1.1. Gráf adattípusok 20

3.1.2. Csúcs és él függvények 21
3.1.3. Gráf adapterek 21
3.1.4. Egyéb felhasznált LEMON funkciók 21
3.2. Az osztályok 21
3.2.1. Dijkstra algoritmus 22
3.2.2. Az összehúzáshoz használt Dijkstra 22
3.2.3. Az élkülönbséghez használt Dijkstra 23
3.2.4. A kereséshez használt Dijkstra 23
3.2.5. A keresést felügyelő osztály 23
3.2.6. A feldolgozást felügyelő osztály 24
3.3. Fontosabb metódusok 25
3.3.1. Élkülönbség 26
3.3.2. Összehúzás 26
3.3.3. Utak visszakeresése 27
3.3.4. Utak lekérdezése 27
3.4. Használat 27

4. Tesztek **29**
4.1. Mérési módszerek 29
4.1.1. A feldolgozás 29
4.1.2. A keresés ideje 29
4.1.3. Felhasznált tár 30
4.1.4. Véglegesített csúcsok száma 30
4.1.5. Távolság és időtartam 31
4.1.6. Útvonal újratervezése 31
4.2. A változatok összehasonlítása 31
4.2.1. Az alap verzió 32
4.2.2. Módosított verziók tesztjei 34
4.3. CH és transit node kombinációja 38
4.3.1. A feldolgozás 38
4.3.2. A keresés 39
4.3.3. Transit node-ok előre kiválogatása 40
4.3.4. A teszt eredménye 40

4.4. Egyéb mérési eredmények 42
4.4.1. A feldolgozás ideje 42
4.4.2. Memória használat 43
4.4.3. Behúzott élek száma 44
4.4.4. A legrövidebb utak élszáma 44
4.5. A kiugró eredmények magyarázata 44
4.5.1. Távolság és idő 44
4.5.2. Különleges gráf 45
4.6. Alkalmazási és továbbfejlesztési lehetőségek 45
4.6.1. Az előkészítés és keresés 46
4.6.2. Memória használat javítása 46
4.6.3. Távolság és gyorsaság 46
4.6.4. Alkalmazások 47

1. Történeti áttekintés

Napjainkra egyre fontosabb, hogy nagy gráfokban minél gyorsabban találjunk legrövidebb utakat. Ez különösen igaz az úthálózatokra, ahol a GPS gyors terjedésének köszönhetően a kisebb erőforrásokkal rendelkező mobil eszközökön kell ezt a feladatot hatékonyan elvégezni.

A kilencvenes évek végétől kezdték el tanulmányozni a meglévő klasszikus algoritmusok továbbfejlesztési lehetőségeit. 2005-ben elindult a 9. DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) implementáció verseny [10]. Ekkor tették mindenki számára elérhetővé az európai és amerikai úthálózatok gráfjait [2]. Ezzel kezdetét vette a leggyorsabb útkereső algoritmus kidolgozása utáni hajsza.

Azonban az algoritmusok gyorsasága erősen függ az előfeldolgozásra szánt időtől és a felhasznált tártól, így az eredményeket nem egyszerű összehasonlítani.

Ebben a fejezetben ismertetjük az eddig elért fontosabb eredményeket. Nagyobb hangsúlyt fektetünk azokra az algoritmusokra, melyek a saját implementációnkban nagyobb szerepet kaptak.

1.1. Dijkstra algoritmusa

Az algoritmust 1956-ban fejlesztette ki Edsger Dijkstra informatikus [6]. Az algoritmus egy nemnegatív élsúlyú G gráfban adott s kezdőcsúcsból találja meg a legrövidebb utat a gráf többi csúcsába.

1.1.1. Bináris kupac adatszerkezet

A bináris kupacot egy balra tömörített teljes bináris fa adatszerkezettel ábrázoljuk. A csúcsai rekordokat tartalmaznak, melyekben adott egy elem és egy kulcs. Például a Dijkstra algoritmus során az elemek a gráf csúcsai, a kulcsok pedig a becsült távolságok az s kezdőcsúcstól.

A kulcsok között teljes rendezésnek kell lennie. Minimum kupac esetén egy adott csúcs kulcsa nem kisebb, mint szülőjének a kulcsa. Ezt hívjuk kupactulajdonságnak.

A számunkra fontos kupacon végezhető műveletek a következők:

- Létrehoz: üres kupacot hoz létre.
- Minkeres: visszaadja a minimális értékű kulccsal rendelkező elemet, mely a bináris fa gyökerében található.
- Mintöröl: törli a kupac legkisebb kulcsú elemét és helyreállítja a kupacszerkezetet.
- Csökkent: csökkenti egy elem kulcsát.
- Beszúr: adott elemet adott kulccsal beszúr a kupacba.

A kupac adatszerkezet előnye, hogy a műveleteket $O(\log n)$ időben el tudjuk végezni, ahol n a kupacban lévő elemek száma. Legfeljebb ennyi lépés kell a kupactulajdonság helyreállításához.

A Dijkstra algoritmus során használjuk a kupac adatszerkezetet. Így a futási idő $O(|E| \log |V|)$ lesz. Ha nem használnánk kupacot, akkor a futásidő $O(|V|^2)$ lenne.

1.1.2. Az algoritmus inicializációja

A Dijkstra algoritmus $G = (V, E)$ irányított gráfban keresi az $s \in V$ csúcsból kiinduló legrövidebb utakat a többi csúcsba adott $c : E \to \mathbb{R}^+$ költségfüggvény esetén.

Adott három halmaz. Q lesz a már véglegesített csúcsok halmaza, L a kupacban lévő csúcsok halmaza, míg S tartalmazza a még nem látott csúcsokat. Egy csúcsot látottnak nevezünk, ha már bekerült a kupacba, vagy ha már véglegesítve lett.

Egy $v \in V$ csúcsra jelölje $dist(v)$ a csúcs becsült távolságát s-től, $pred(v)$ pedig az a véglegesített csúcs, ahonnan v csúcsba érkeztünk. A keresés során mindkét adat változhat a nem véglegesített csúcsokon.

Kezdetben $\forall v \in V \setminus \{s\}$ $dist(v) = \infty$, $pred(v) = null$, $dist(s) = 0$, $pred(s) = null$, $Q = \emptyset$, $S = V \setminus \{s\}$, $L = \{s\}$.

1.1.3. Futás

Az algoritmus futása alatt felépíti az s gyökerű, legrövidebb utakból álló fát.

Egy lépés során kiválasztjuk L halmazból azt a v csúcsot, melyre $dist(v)$ minimális.

Ezután $\forall e = (v, w) \in E$ élt, melyre $w \in S \cup L$, relaxálunk. Azaz ha $dist(v)+c(e) < dist(w)$ teljesül, akkor $dist(w) := dist(v)+c(e)$ és $pred(w) := v$. Valamint w csúcs az L halmazba kerül ha előtte S-ben volt.

Végül véglegesítjük v-t. Azaz $L := L \setminus \{v\}$, $Q := Q \cup \{v\}$. Vagyis v csúcsot többbet nem fogjuk vizsgálni, $dist(v)$ értéke pedig a v csúcs valódi távolsága s-től.

Megjegyzés. A dolgozatban sokszor használjuk a véglegesítés, relaxálás és a látott csúcsok fogalmát. Ezeken minden esetben az itt ismertetett eljárást értjük.

1.1.4. Leállás

Az algoritmus leáll, ha $L = \emptyset$, azaz nincs több csúcs a kupacban. Ha adott t célcsúcsba kerestük a legrövidebb utat s-ből, akkor leállíthatjuk az algoritmust, ha $t \in Q$.

Leállás után $\forall v \in Q$ csúcsra $dist(v)$ tárolja s és v távolságát, $pred(v)$ pedig azt a csúcsot, melyen v-be jutottunk, így az utat vissza tudjuk keresni.

1.1.5. Kétirányú Dijkstra algoritmus

Az eredeti egyirányú Dijkstra algoritmushoz képest itt egy s kezdőcsúcsból egy t célcsúcsba kereshetünk legrövidebb utat. Az algoritmus két egyirányú Dijkstrát léptet felváltva. Az előrekeresés s csúcsból indul az eredeti gráfon, a hátrakeresés t-ből indul a fordított gráfon, vagyis ahol az élek irányítását megfordítottuk.

Amikor a két keresés találkozik egy v csúcsban, vagyis v-t mindkét keresésben véglegesítettük, akkor a legrövidebb út vagy $P(s, v) \cup P(v, t)$ alakú, vagy $\exists u, w \in V$ hogy a legrövidebb út $P(s, u) \cup \{(u, w)\} \cup P(w, t)$, ahol u csúcsot az előrekeresésben, w-t a hátrakeresésben véglegesítettük.

1.2. A* algoritmus

1968-ban Nils Nilsson javasolta a Dijkstra algoritmus általánosítását heurisztika alkalmazásával [5]. A csúcsokon adott egy függvény, mely minden csúcs esetén nemnegatív, konzisztens alsó becslés a t célcsúcstól vett távolságra. Úthálózatok esetén ilyen például az euklideszi távolság.

Tehát adott egy $G = (V, E)$ gráf, $c : E \to \mathbb{R}^+$ költségfüggvény, és $h : V \times V \to \mathbb{R}^+$, melyre teljesül a következő feltétel: $\forall (u, v) \in E$ élre $h(t, v) - h(t, u) \leq c(u, v)$.

Az A* valójában egy Dijkstra algoritmus a G gráfon s-ből t-be, $c' : E \to \mathbb{R}$ költségfüggvény mellett, ahol egy (u, v) élre $c'(u, v) = c(u, v) + h(t, v) - h(t, u)$.

Ha $h \equiv 0$, akkor pont a Dijkstra algoritmust kapjuk. Ha h minden csúcsra a csúcs pontos távolsága t-től, akkor az algoritmus minden s kezdőcsúcsból csak a $P(s, t)$ legrövidebb utak csúcsait véglegesíti.

A heurisztika segítségével tehát azokat a csúcsokat részesítjük előnyben, melyek „jó” irányban helyezkednek el a kezdő és célcsúcs között, így a Dijkstra algoritmushoz képest kevesebb véglegesített csúcs lesz.

Hasonlóan a kétirányú Dijkstra algoritmushoz, létezik kétirányú A* is.

1.3. Gráfok előfeldolgozását igénylő módszerek

A továbbiakban olyan keresőalgoritmusokat ismertetünk, melyek használata előtt a bemeneti gráfon előfeldolgozást kell végeznünk.

A bemeneti gráf mostantól legyen irányítatlan. A feladat mindig a legrövidebb út megtalálása s és t csúcsok közt. Az algoritmusokat úthálózatokon futtatjuk, így feltehető, hogy a legrövidebb utak egyértelműek. Az u és v csúcsok közti legrövidebb utat jelölje $P(u, v)$.

1.3.1. Autópálya hierarchiák

Az autópálya hierarchiák (highway hierarchies, ezentúl HH) módszer a $G = (V, E)$ gráfon végzett előfeldolgozással gyorsítja a keresést [2].

Autópálya élen a következőt értjük. Minden v csúcshoz megadunk egy $r(v) \in \mathbb{R}^+$ környezetet. Ekkor (u, v) autópálya él, ha $\exists s, t \in V$, hogy

$(u, v) \in P(s, t)$, valamint $d(s, u) > r(s)$ és $d(v, t) > r(t)$, ahol $P(s, t)$ egy legrövidebb s, t út, u, v csúcsokra $d(u, v)$ pedig az u, v csúcsok távolsága. Az $r(v)$ környezetet általában úgy választják, hogy adott H számú csúcs kerüljön bele.

A gráf előfeldolgozása közben két eljárás közt alternál:

- A nem-autópálya élek törlése.
- Csúcsok összehúzás: olyan csúcsokat töröl, melyre legfeljebb két él illeszkedik, és a csúcs helyett rövidítő éleket húz be, hogy az így kapott gráfban a csúcsok közti távolságokat megőrizze.

A keresés során egy módosított kétirányú Dijkstra algoritmust használ.

1.3.2. Transit node

Egy úthálózaton, ha két távoli s és t pont között keressük a legrövidebb utat, akkor könnyen látható, hogy adott néhány csúcs s környezetében, hogy a legrövidebb út ezek valamelyikén keresztülmegy. Ilyen csúcsok lehetnek például egy városból kivezető főbb utak végpontjai. Jelöljük az ilyen csúcsokat $A(s)$-el.

Transit node-oknak hívjuk a $T = \bigcup_{v \in V} A(v)$ halmaz elemeit. Két közeli v és w csúcs esetén $A(v)$-nek és $A(w)$-nek lehetnek közös elemei, vagy meg is egyezhetnek. Például ha v és w egy kis utca két végpontja. Így várhatóan $|T|$ nem lesz túl nagy. Az Európa úthálózatát ábrázoló gráfban körülbelül 10000 ilyen tulajdonságú csúcsot találtak [3].

Hasonlóan a HH-hoz, előfeldolgozást kell végeznünk a gráfon a keresések előtt. Meg kell határoznunk minden v csúcsra $A(v)$ halmazt. Ezután minden $u \in A(v)$ csúcs esetén tároljuk $d(u, v)$ és $d(v, u)$ távolságokat. Végül egy mátrixban tároljuk $\forall u, v \in T$ csúcspárra $d(u, v)$-t.

Ekkor az s és t távoli csúcsok távolságát megkapjuk néhány lekérdezéssel. Vagyis $d(s, t) = min\{d(s, v) + d(v, w) + d(w, t) : v \in A(s), w \in A(t)\}$.

Közeli csúcsok esetén ez a módszer nem működik, a legrövidebb út nem fog átmenni transit node-on. Így a módszer hátulütője, hogy el kell dönte-

nünk, hogy használhatjuk-e a fenti módszert. Erre alkalmas lehet a földrajzi távolság [3].

1.3.3. Highway Node Routing

A Higway Node Routing (HNR) hasonlóan a HH-hoz, az útvonalhálózatok hierarchikus felépítését használja ki a keresés gyorsításához. A gráf előfeldolgozása a csúcsokat szintekbe rendezi, majd új rövidítő éleket húz be [4].

Induljunk ki a $G_0 = (V, E)$ gráfból. Válasszunk egy $S_1 \subseteq V$ csúcshalmazt. $\forall u, v \in S_1$ csúcspárra határozzuk meg $d(u, v)$-t. Legyen $G_1 = (S_1, E_1)$, ahol E_1-ben olyan (u, v) élek szerepelnek $d(u, v)$ súllyal, hogy a $P(u, v) \subset G_0$ legrövidebb út egyik belső csúcsa[1] sem szerepel az S_1 halmazban. Ezt az eljárást ismételjük a $G_1, G_2 \ldots$ gráfokra. Egy v csúcs szintje i ha $v \in G_i$ de $v \notin G_{i+1}$.

Az előfeldolgozás során a cél az S halmazokat jól megválasztani és az új éleket behúzni. Miután véget ért a fenti eljárás, legyen $G = (V, E \cup E_1 \cup E_2 \cup \ldots)$. Vagyis létrehozunk egy gráfot, melyben minden eredeti és új él szerepel.

A keresés során egy módosított kétirányú Dijkstra algoritmust használunk G gráfon. Az előrekeresés s-ből indul, a hátrakeresés a fordított gráfban pedig t-ből. Mindig csak azokat az (u, v) éleket relaxáljuk, ahol u szintje $\leq$ mint v szintje.

1.3.4. Összehúzási Hierarchiák

A HNR egy speciális esete, amikor minden szint egyetlen csúcsot tartalmaz. A dolgozat célja ennek a módszernek az ismertetése, az implementáció és teszteredmények bemutatása. Az algoritmust a következő fejezetben ismertetjük.

[1] Nem a kezdő vagy a végpontja az útnak.

2. Összehúzási hierarchiák

A gyors útvonalkereséshez a Dijkstra algoritmus már önmagában megfelelő eszköz. Azonban a gráfon végzett előfeldolgozással és a Dijkstra algoritmus módosított változatával az egyes lekérdezések sokkal gyorsabban elvégezhetőek.

Ebben a fejezetben az [1] cikk első négy fejezete kerül feldolgozásra.

2.1. Definíciók

A későbbiekben ismertetett eljárás leírásához több kifejezést használunk, ami esetleg a megszokottól eltérő jelentéssel bír.

Az algoritmus egy irányított $G = (V, E)$ gráfon fut. Az egyszerűség kedvéért azonban felhasználunk irányítatlan gráfokra vonatkozó definíciókat is.

Definíció. Egy csúcs fokszámán a kiélek és beélek számának összegét értjük. Egy csúcs szomszédai legyenek a ki és beszomszédok halmazának uniója.

Definíció. Egy csúcs feldolgozása, összehúzása, törlése, sorbarendezése mind a 2.3. fejezetben ismertetett eljárást jelenti. Hasonlóan a feldolgozott, összehúzott, törölt csúcsok azok a csúcsok, melyekre már futtattuk az említett eljárást.

Definíció. Egy $e \in E$ élre $c(e)$ jelölje az él költségét. Hasonlóan $c(u, v)$ az (u, v) él költsége.

Definíció. Egy út hossza jelentse az út költségét, azaz a benne szereplő élek súlyának összegét. Ekkor $u, v \in V$ csúcsokra $d(u, v)$ jelöli az u-ból induló és v-be érkező legrövidebb út hosszát. Ezt értjük u, v csúcsok távolságán is. Mivel irányított gráfról van szó, így általában $d(u, v) \neq d(v, u)$.

2.2. Motiváció

A feldolgozás során a csúcsokat megfelelő sorrendben összehúzzuk a gráfban, közben új rövidítő éleket húzunk be úgy, hogy az új gráfban a legrövidebb utak megmaradjanak.

A sorrend meghatározásához több szempontot is figyelembe kell vennünk. Az egyik legfontosabb az élkülönbség, mely az összehúzás utáni behúzott élek száma és az összehúzás előtti fokszám különbsége. Valamint fontos, hogy a gráfban egyenletesen helyezkedjenek el az egymás után feldolgozott csúcsok. Erre jó heurisztikát ad a törölt szomszédok száma. Ezen kívül még sok paraméter figyelembe vehető a csúcsok sorrendjének meghatározásakor. Ezzel még tovább növelhető a keresés gyorsasága, azonban a feldolgozás tovább tarthat. A csúcsok sorba rendezéséről bővebben a 2.4. fejezetben lehet olvasni.

Egy v csúcs törlésekor megvizsgáljuk minden $u, w \in V$ csúcspárra, ahol $(u, v) \in E$ és $(v, w) \in E$, a köztük lévő legrövidebb utat. Amennyiben a legrövidebb út $P = u, v, w$ alakú, úgy behúzzuk az (u, w) élt $c(u, w) = c(u, v) + c(v, w)$ súllyal. A csúcsok összehúzásával bővebben a 2.3. fejezet foglalkozik.

A módosított gráf és a csúcsok sorrendje adja az összehúzási hierarchiát (angolul contraction hierarchy, ezentúl CH). Létrehozunk két új gráfot: $G_\uparrow = (V, E_\uparrow)$ és $G_\downarrow = (V, E_\downarrow)$, ahol $E_\uparrow$-ben találhatók azok az élek, melyek alacsonyabb sorszámú csúcsból mennek magasabba, $E_\downarrow$-ben pedig azok, melyek magasabb sorszámúból mennek alacsonyabba. Ezt felhasználva egy módosított kétirányú Dijkstra algoritmust alkalmazunk. Az előre keresést $G_\uparrow$ gráfban végezzük, míg a hátrakeresést a $G_\downarrow$ gráfban. A keresésről bővebben a 2.5. fejezetben írtunk.

Részletes leírás a következő alszakaszokban olvasható. A saját implementációval a 3. fejezet foglalkozik. A tesztekről és elért eredményekről a 4. fejezetben írtunk.

2.2.1. Nehézségek a feladat feldolgozása során

Néhány esetben a cikk meglehetősen homályosan fogalmaz, és kevésbé tér ki a részletekre, vagy nem ad kellő magyarázatot. Ilyen például a 2.5.2. szakaszban található utak visszakeresése, és néhány prioritási tényező kiszámításának módja 2.4-ban.

Ilyen esetekben a leírást néhol kibővítettük magyarázatokkal, illetve a 3.

fejezetben részletesebben írtunk a megvalósításról.

2.3. Csúcsok összehúzása

Legyen G az eredeti gráfunk, G' pedig az a gráf, melyet néhány csúcs összehúzása után kapunk. A cél az, hogy $\forall u, v \in G'$ csúcspárra $d'(u, v) = d(u, v)$ teljesüljön, ahol a fenti definíciók szerint $d(u, v)$ a két csúcs közti távolság G gráfban, $d'(u, v)$ pedig G'-ben.

Most tegyük fel, hogy már k darab csúcsot megfelelően összehúztunk. Azaz adott $G' = (V', E')$ irányított gráf, ahol V' a még nem törölt csúcsok halmaza, E' a még meglévő régi, valamint az eddig behúzott új élek halmaza, és G'-re teljesül az előbbi feltétel.

A következő v csúcs összehúzásánál meg kell vizsgálnunk, mely új rövidítő élekre lesz szükségünk. Ehhez tekintsük az $u, w \in V'$ csúcspárokat, melyre $(u, v) \in E'$ és $(v, w) \in E'$. Csak akkor nem kell behúznunk az (u, w) élt, ha $d^*(u, w) \leq c(u, v) + c(v, w)$, ahol $d^*(u, w)$ az u és w csúcsok közti távolság a G' gráfban, eltávolított v csúcs esetén. Vagyis csak akkor vesszük a gráfhoz (u, w) élt, ha végpontjai közt minden legrövidebb út a v csúcson át megy.

A legegyszerűbb, ha minden ilyen u csúcsból indítunk egy Dijkstra keresést, míg el nem érjük az összes megfelelő w csúcsot. Elegendő, ha a keresést $c(u, v) + max\{c(v, w) : (v, w) \in E'\}$ távolságig futtatjuk u csúcsból, ugyanis ezután $d^*(u, v)$ csak nagyobb lehet mint $c(u, v) + c(v, w)$.

2.4. Csúcsok sorba rendezése

A legfontosabb feladat a CH elkészítése közben a csúcsok sorrendjének megfelelő meghatározása. Ez befolyásolja a behúzott élek számát, valamint nagyban hozzájárul a keresés gyorsaságához. A tesztjeink alapján a teljesen véletlen sorrend esetén a Dijkstra algoritmus is gyorsabb volt az implementációnknál.

A sorrend meghatározásához egy kupacban minden csúcshoz hozzárendelünk egy értéket, mely több tényező lineáris kombinációja. Mindig a legkisebb prioritású elemet húzzuk össze.

2.4.1. Prioritások frissítése

A csúcsok összehúzása befolyásolhatja más csúcsok prioritását. A minél jobb eredmény elérése érdekében így újra meg kell határoznunk bizonyos csúcsok prioritását. Erre három módszert és ezek kombinációit vizsgáltuk:

- Lusta frissítés: mindig újraszámoljuk az éppen összehúzni kívánt csúcs prioritását, és csak akkor húzzuk össze, ha még mindig ő a legjobb jelölt.
- Szomszédok frissítése: adott csúcs összehúzása után újraszámoljuk a csúcs szomszédainak a prioritásait.
- Újraszámoljuk az összes csúcs prioritását valamilyen feltétel teljesülése esetén, például minden századik csúcs összehúzása után.

2.4.2. Prioritási tényezők: élkülönbség

Az egyik legfontosabb összetevője a csúcsok prioritásainak az élkülönbség. Az úthálózatok ritkák, egy csúcs átlag fokszáma 4 és 5 között van [10]. Szeretnénk, ha ez nem növekedne jelentősen az összehúzás során. Erre használjuk az élkülönbséget.

A tényező értékét megkapjuk az összehúzás utáni újonnan behúzandó élek száma és az összehúzás előtti fokszám különbségeként. Ehhez előtte meg kell vizsgálnunk, hány él kerülne behúzásra, aminek a meghatározása egyenértékű a csúcs összehúzásával.

Ez meglehetősen erőforrás igényes feladat. Az élkülönbség a feldolgozás során változik, így frissítenünk kell a prioritásokat. Az éppen összehúzott csúcs szomszédaira mindenképpen újra kell számolnunk. Ezzel együtt a lusta frissítés a legtöbb esetben már elegendő az értékek karbantartására.

Azonban mivel heurisztika és egy csúcsra az értéket többször is újra kell számolnunk, így viszonylag kis pontatlanság árán sokkal gyorsabbá tehető a gráf feldolgozása hop limit használatával. Vagyis tovább korlátozzuk a Dijkstra futását, csak a k-nál kevesebb élt tartalmazó utakat vizsgáljuk. Az implementációnkban $k = 5$.

2.4.3. Prioritási tényezők: egyenletesség

Már kizárólag az élkülönbség használatával is elég jó CH-k készíthetők. Azonban ha például a gráf egy út, és a csúcsokat sorrendben húzzuk össze, akkor visszakapjuk az eredeti utat, így a keresésünk nem lesz gyorsabb. Ezért érdemes a gráf csúcsait szétszórtan összehúzni, vagyis ha már egy csúcsot összehúztunk, akkor a közelében lévő csúcsokra csak később kerül sor.

Az egyik lehetőségünk, hogy minden csúcson nyilvántartjuk, hány szomszédját dolgoztuk már fel. Az összehúzás során a csúcsoknak új szomszédai is keletkeznek a behúzott élek miatt, ezeket is figyelembe kell vennünk. Ezt könnyen megtehetjük az összehúzott csúcs szomszédainak frissítésével. Minél több szomszédot töröltünk, annál később kerül összehúzásra a csúcs.

Másik lehetőségünk Voronoi cellák használata. Egy csúcs $R(v)$ Voronoi cellája álljon azokból a csúcsokból, amiket már összehúztunk, és közelebb vannak v-hez mint más még nem összehúzott csúcsokhoz. Ekkor $\sqrt{|R(v)|}$ lehet egy prioritási tényező. Amikor összehúzunk egy csúcsot, akkor a cellája szétoszlik a még meglévő csúcsok közt. Azonban ennek az újraszámítása meglehetősen számításigényes.

Ha az úthálózatban rendelkezésre állnak a csúcsok GPS koordinátái, akkor ezeket is felhasználhatjuk az egyenletes összehúzás biztosítására. Itt a prioritási tényező adott csúcs esetén a csúcs és a hozzá legközelebbi összehúzott csúcs távolsága, ahol a távolságon most a földrajzi távolságot értjük kilométerben. Az implementációnkról a 3.2.5 és 3.2.6. fejezetekben írtunk.

2.4.4. Prioritási tényezők: egyéb

Használhatunk egyéb tényezőket is a prioritások megbecsléséhez. Ilyen lehet például adott csúcs esetén a talált utak élszámára egy felső becsülés: minden csúcsra kezdetben $Q(v) = 0$, majd egy csúcs összehúzása után minden u szomszédjára $Q(u) = max(Q(u), Q(v) + 1)$. Ez szerepel az implementációnkban.

Ezen kívül vizsgálhatjuk, hogy lokálisan hány darab legrövidebb út halad át adott csúcson, esetleg a CH építésének költsége is lehet egy tényező. Más heurisztikát is kereshetünk a csúcsok sorrendjének meghatározására, azonban

a dolgozatban csak a fent említettekkel foglalkoztunk.

2.4.5. Prioritási tényezők kombinációja

Ha kiszámítottuk adott csúcsra a fenti tényezőket, akkor meg kell határoznunk a csúcs prioritását, ami az előbbi értékek lineáris kombinációja lesz. Azonban minél több tényezőnk van, annál nehezebb őket összehangolni.

A legfontosabbak az élkülönbség és az egyenletesség, így ezek nagyobb hangsúlyt kell, hogy kapjanak. A soron következő csúcs mindig a legkisebb prioritású lesz.

Az implementációban az élkülönbség együtthatója 190, a törölt szomszédoké 180, a többi tényezőé 1.

Ha adottak a csúcsokon a GPS koordináták, akkor az élkülönbség együtthatója 190, a földrajzi távolságé pedig -5, mivel a legkisebb prioritású csúcsot húzzuk össze legközelebb és nekünk a legtávolabbi csúcsot érdemes előbb összehúznunk.

2.5. Legrövidebb út keresése a feldolgozott gráfban

Miután elkészült a CH, megkapjuk $G' = (V, E')$ gráfot, amiben szerepel minden régi és behúzott él is. Ekkor létrehozunk két új gráfot: $G_\uparrow = (V, E_\uparrow)$ és $G_\downarrow = (V, E_\downarrow)$, ahol $E_\uparrow = \{(u,v) \in E : sorsz(u) < sorsz(v)\}$, $E_\downarrow = \{(u,v) \in E : sorsz(u) > sorsz(v)\}$, ahol $sorsz(v)$ jelöli, hogy v-t hányadikként húztuk össze.

Definíció. Egy $(u,v) \in E'$ élt nevezzünk előreélnek ha $sorsz(u) < sorsz(v)$, hátraélnek, ha $sorsz(u) > sorsz(v)$.

A két gráfon egy módosított kétirányú Dijkstra algoritmust alkalmazunk. Az előre keresést $G_\uparrow$ gráfban, míg a hátrakeresést a $G_\downarrow$ gráfban végezzük. Ehhez az implementációban $G_\downarrow$ fordítottját tároljuk.

Definíció. $d_\uparrow(u,v)$ jelölje az u és v csúcsok távolságát $G_\uparrow$ gráfban. Hasonlóan $d_\downarrow(u,v)$ az u és v csúcsok távolsága $G_\downarrow$ gráfban.

Definíció. Egy $P(s,t)$ útban nevezzük két egymás utáni $e = (u_{k-1}, u_k)$ és $f = (u_k, u_{k+1})$ élt alsó alternáló élpárnak, ha $sorsz(u_k) < sorsz(u_{k-1})$ és $sorsz(u_k) < sorsz(u_{k-1})$. Vagyis u_k csúcsot előbb húztuk össze, mint az útban előtte és utána lévő két csúcsot. A fenti definíció alapján tehát e hátraél, f előreél.

Definíció. Az előző definícióhoz hasonlóan a két él legyen felső alternáló élpár, ha e előreél, f pedig hátraél.

Állítás. *Tekintsük a G' gráfban szereplő legrövidebb $P(s,t)$ utat, mely a legkevesebb élt tartalmazza. Ekkor P-ben nem lehet alsó alternáló élpár.*

Megjegyzés. Nem tehetjük fel, hogy a legrövidebb út egyértelmű. Az új élek behúzásával a legrövidebb utak hosszai megmaradnak, de új legrövidebb utak is keletkeznek.

Bizonyítás. Indirekt tegyük fel, hogy $P = (u_0, u_1, \ldots, u_p)$ egy ilyen út, azaz legrövidebb és a legkevesebb élt tartalmazza, viszont szerepel benne $e = (u_{k-1}, u_k)$ és $f = (u_k, u_{k+1})$ alsó alternáló élpár. Vagyis a definíció szerint $sorsz(u_k) < sorsz(u_{k-1})$ és $sorsz(u_k) < sorsz(u_{k-1})$, tehát u_k csúcsot előbb húztuk össze, mint az útban lévő szomszédait.

Az u_k csúcs feldolgozásakor két lehetőség volt:

1. Be kellett húznunk (u_{k-1}, u_{k+1}) élt. Ez azt jelenti, hogy ha P-ből e és f éleket elhagyjuk, helyette hozzávesszük (u_{k-1}, u_{k+1}) élt, akkor egy ugyanolyan hosszú utat kaptunk, melyben eggyel kevesebb él van.

2. Nem kellett behúznunk (u_{k-1}, u_{k+1}) élt. Vagyis u_{k-1} és u_{k+1} közt van egy rövidebb Q út, ami nem halad át u_k csúcson. Ekkor e és f élek helyett a Q út éleit véve P-nél rövidebb utat kaptunk.

Mindkét esetben ellentmondásra jutottunk, tehát P-ben nem lehetnek alsó alternáló élek. □

Állítás. *Ismét tekintsük a G' gráfban szereplő legrövidebb $P(s,t)$ utat, mely a legkevesebb élt tartalmazza. Ekkor P-ben legfeljebb egy felső alternáló élpár lehet.*

Bizonyítás. Tegyük fel indirekt, hogy egy ilyen útban mégis van legalább két felső alternáló élpár. Ekkor bármely két ilyen élpár között kellene lennie egy alsó alternáló élpárnak is, mely az előző állítás miatt nem lehetséges. □

Következmény. *Az előző két állításból következik, hogy egy $P(s,t) = (u_0, u_1, \ldots, u_p)$ útnak háromféle alakja lehet.*

1. $\forall k \in \{0, 1, \ldots, p-1\}$ $sorsz(u_k) < sorsz(u_{k+1})$, *ahol* $u_0 = s$ *és* $u_p = t$.
   Vagyis P *minden éle* $G_\uparrow$ *gráfban van.*

2. $\forall k \in \{0, 1, \ldots, p-1\}$ $sorsz(u_k) > sorsz(u_{k+1})$.
   Vagyis P *minden éle* $G_\downarrow$ *gráfban van.*

3. $\exists m, \forall k \in \{0, 1, \ldots, m-1\}$ $sorsz(u_k) < sorsz(u_{k+1})$, *és* $\forall k \in \{m, m+1, \ldots, p-1\}$ $sorsz(u_k) > sorsz(u_{k+1})$.
   Vagyis van egy csúcs, mely esetén az útban előtte lévő élek $G_\uparrow$ *gráfban vannak, míg az után következő élek* $G_\downarrow$ *gráfban.*

Állítás. $d(s,t) = min_{v \in V}\{d_\uparrow(s,v) + d_\downarrow(v,t)\}$.

Bizonyítás. Az előző következmény alapján 3 lehetőség van $P(s,t)$ út alakja szerint.

1. P út minden éle $G_\uparrow$ gráfban van. Ekkor $v = t$ lesz a megfelelő csúcs.

2. P minden éle $G_\downarrow$ gráfban van. Ekkor $v = s$.

3. A v csúcs olyan, hogy $P(s,v) \subseteq P(s,t)$ minden éle $G_\uparrow$ gráfban van és $P(v,t) \subseteq P(s,t)$ minden éle $G_\downarrow$ gráfban található. Ekkor egy v csúcs esetén $d(s,t) \leq d_\uparrow(s,v) + d_\downarrow(v,t)$.

Mivel az eddigiek alapján ez az összes lehetőség, így valamely v csúcsra $d(s,t) = d_\uparrow(s,v) + d_\downarrow(v,t)$ teljesülni fog. □

Megjegyzés. Az előző bizonyításban az első két pont a 3. pont speciális esete, így az implementációban elegendő a 3. pont alapján ellenőrizni a lehetséges utakat.

2.5.1. Lehetséges javítások

A keresés során felváltva léptetjük az előre és hátra kereséseket. Ha találtunk egy v csúcsot, melyet mindkét irányban véglegesítettünk, akkor kapunk egy új legrövidebb út jelöltet.

A keresést megállíthatjuk az egyik irányból, ha a soron következő csúcs távolsága a kezdőponttól nagyobb mint a legjobb út hossza, hiszen ekkor ebben az irányban már nem találhatunk rövidebb utat.

Ezen kívül kihasználhatjuk a másik irányban lévő gráf éleit is. Amikor véglegesítjük a következő v csúcsot $G_\uparrow$-ben, megnézzük, hogy létezik-e w, hogy $(w, v) \in E_\downarrow$ élre $d_\uparrow(s, w) + c(w, v) < d_\uparrow(s, v)$. Ha találunk ilyen csúcsot, akkor a v-be vezető út nem optimális, így ebből a csúcsból nem kell folytatnunk a keresést (az éleket nem relaxáljuk). Hasonlóan járhatunk el $G_\downarrow$ esetén is.

A gráfokat tárolhatjuk egy adatstruktúrában, ahol minden él esetén jelezzük, hogy melyik gráfhoz tartozik. Valamint az éleket elegendő a kisebb sorszámú végpontjukban tárolni.

2.5.2. Az utak visszakeresése

Várhatóan a talált legrövidebb út tartalmazni fog olyan éleket, melyeket a feldolgozás során húztunk be. Azonban szeretnénk visszakapni az út eredeti éleit.

Ehhez minden új élen tárolnunk kell, hogy mely két él helyett lett a gráfhoz adva, esetleg tárolhatjuk, hogy mely csúcs összehúzása során kellett behúznunk. Vagyis v csúcs összehúzása során ha be kell húznunk (u, w) élt, akkor ezen az élen tárolnunk kell v csúcsot, vagy (u, v) és (v, w) éleket. Ha a gráf magvalósításából gyorsan le tudjuk kérdezni két csúcs közti élt, akkor elegendő a csúcsot tárolni a behúzott élen, különben gyorsabb lesz a visszakeresés az élek tárolásával.

Ezekből az információkból rekurzívan visszakereshető az eredeti út élhalmaza. A konkrét megvalósításról a 3.3.2. és 3.3.3. fejezetekben írtunk.

3. Az implementáció

Az implementáció a LEMON (Library for Efficient Modeling and Optimization in Networks) C++ könyvtár gráfadattípusaira építve készült a 2. fejezetben tárgyaltak alapján, néhány módosítással.

A paraméterekkel és a különböző megvalósításokkal bővebben a 4. fejezet foglalkozik.

3.1. LEMON

A projektet 2003-ban indította az Egerváry Jenő Kombinatorikus Optimalizálási Kutatócsoport (EGRES) az ELTE Operációkutatás Tanszékén. A projekt nyílt forráskódú, szabadon használható a benne szereplő licensz feltételek megtartásával [7].

A könyvtár célja a hatékony algoritmusok és adatszerkezetek implementálása a hálózatok és a kombinatorikus optimalizálás területén. A készítők erős hangsúlyt fektettek a rugalmasságra is: a generikusan megírt algoritmusok az összes megfelelő adatszerkezeten jól működnek. A hatékonyság ára azonban a biztonságos kód hiánya, így a kellő ellenőrzésekről és a helyes inputról a felhasználónak kell gondoskodnia. Ezért részletes dokumentáció áll a felhasználók rendelkezésére [8].

3.1.1. Gráf adattípusok

A LEMON-ben három fő irányított gráf implementáció van.

A gráf feldolgozása során megfelelő feltételek mellett új éleket adunk a gráfhoz (2.3. fejezet). Mikor az élkülönbséget[2] szeretnénk meghatározni egy csúcsra, akkor ellenőriznünk kell, mely új éleket kellene behúznunk, ha a csúcsot összehúznánk. Ehhez ténylegesen hozzá kell vennünk ezeket az éleket a gráfhoz. A számítás után így ezeket törölnünk is kell. Erre csak a ListDigraph implementáció alkalmas.

A keresés leggyorsabb megvalósításához a StaticDigraph adatszerkezetet használjuk, mely, mint a neve is mutatja, módosíthatatlan, így tárolása

[2] Adott csúcs összehúzása esetén a behúzott új élek száma és a fokszám különbsége.

hatékony. Ebből eredően az élein az iteráció gyorsabb, mint más gráfimplementációkon.

3.1.2. Csúcs és él függvények

A csúcsokhoz illetve élekhez különböző értékeket rendelhetünk úgynevezett Map objektumok használatával. Így tárolhatjuk az éleken a költségeket (ArcMap), vagy a csúcsokon a sorrendjüket, amiben összehúzzuk őket (NodeMap). Tulajdonképpen kulcs-érték párokat tárolunk.

Az C++ Standard Library-ben lévő implementációval ellentétben itt konstans időben tudjuk lekérdezni az adott kulcshoz tartozó értéket [9].

3.1.3. Gráf adapterek

A LEMON-ben lehetőségünk van egy létező gráfon új adapter objektumot létrehozni, mely segítségével új funkcionalitások érhetők el.

Az implementációban a legfontosabb a SubDigraph, mely segítségével a gráfban éleket és csúcsokat rejthetünk el azok törlése nélkül. Ez segít a StaticDigraphok létrehozásában a feldolgozás végén.

3.1.4. Egyéb felhasznált LEMON funkciók

A teszteléshez használt gráfok beolvasásában is segít a könyvtár, mely a LEMON saját formátumán kívül egyéb formátumokat is támogat, így a használt DIMACS formátumot is.

A Dijkstra algoritmus során a csúcsok távolságát, valamint a feldolgozás során a csúcsok prioritását egy bináris kupac adatszerkezetben tároljuk.

Teszteléshez a könyvtárban lévő véletlen szám generátor segítségével választjuk ki a keresendő utak végpontjait.

A StaticDigraphok létrehozásában a DigraphCopy osztály nyújtott segítséget.

3.2. Az osztályok

Az objektumorientált szemlélet elengedhetetlen volt a megvalósításhoz.

A feladat elvégzése során három különböző Dijkstra implementációra volt szükség, melyek az eredetitől a hatékonyság növelése, a tesztelés megkönnyítése és az egyszerűbb továbbfejlesztési lehetőség érdekében térnek el.

Ezen kívül az egyik osztály magáért a CH elkészítéséért felel, valamint a keresés felügyeletét is egy külön osztály végzi.

3.2.1. Dijkstra algoritmus

A LEMON könyvtárban szereplő Dijkstra algoritmus bár hatékony, a célunknak nem volt megfelelő. A legnagyobb problémát az inicializáció során újra létrehozott Mapek okozták. A létrehozás nagy gráfokon érezhetően sokáig tartott, így a gráf feldolgozása, valamint a keresés is egy nagyságrenddel lassabb volt.

A **CHDijkstra** osztályban csak két szükséges Map maradt. Az egyik a csúcsok távolsága, a másik pedig a csúcsok állapota, melyet egyben a kupac is használ.

Amint egy csúcs bekerül a kupacba, egy verembe is betesszük. Az osztály *clear* metódusában kiürítjük a kupacot, valamint a veremben lévő csúcsok állapotát visszaállítjuk. Mivel a feldolgozás során csak lokálisan keresünk, valamint a legrövidebb út meghatározása során csak kevés csúcs érintett, így ezzel a módszerrel jelentős idő megtakarítható.

A gráfból nem szeretnénk a csúcsokat törölni, a keresőgráfok létrehozásánál szükségünk lesz rájuk. A csúcsok elrejtésével pedig bonyolultabb lenne a megvalósítás. Így a következő trükköt alkalmazzuk: ha egy csúcsot törölnünk kellene, akkor a csúcsot a keresőosztályban beállítjuk véglegesítettnek, így többet nem kerülhet be a kupacba. Így az útkeresés szempontjából olyan, mintha töröltük volna a gráfból. Ez az elrejtéshez képest nem okoz többlet futási időt.

Ez az osztály szolgál a három különböző variáció őseként.

3.2.2. Az összehúzáshoz használt Dijkstra

A csúcsok összehúzásakor használt **ContractDijkstra** osztály. Teljes egészében megegyezik a **CHDijkstrával**, azonban teszteléshez és későbbi fej-

lesztések esetén így könnyebben módosítható.

3.2.3. Az élkülönbséghez használt Dijkstra

Az **EdgeDiffDijkstra** osztály segítségével állapítjuk meg adott csúcsra az élkülönbséget. Mivel az összehúzás meglehetősen időigényes, és az élkülönbség kiszámításához meg kell vizsgálnunk mi történne, ha a csúcsot összehúznánk, így itt használjuk a hop limitet (2.3. fejezet).

Ehhez egy NodeMapben tároljuk, hogy adott csúcsot hány élen keresztül érünk el. A kezdő csúcs szintje 0. Minden v csúcsra, ami bekerül a kupacba, $szint(v) = szint(u) + 1$, ahol u csúcs v elődje. Ha a hop limit k volt, akkor ha a következő feldolgozandó csúcs szintje eléri azt, akkor véglegesítjük, de éleit nem relaxáljuk.

Készült egy változat, ahol ezt nem használtuk, illetve ahol hop limit helyett a véglegesített csúcsok számát korlátoztuk a keresésben.

3.2.4. A kereséshez használt Dijkstra

Ebben az osztályban keressük a StaticDigraphban a legrövidebb utat. Mivel szeretnénk lekérdezni az út éleit is, így a csúcson egy NodeMapben tároljuk, hogy melyik élen jutottunk el oda.

3.2.5. A keresést felügyelő osztály

A **CHSearch** osztály felel a StaticDigraphokban a keresésért a 2.5. rész alapján.

Külön tároljuk a kezdő csúcstól (s) és célcsúcstól (t) elért távolságot (ds és dt)[3]. Vagyis ds az előrekeresésben utoljára véglegesített csúcs távolsága s-től, dt pedig a hátrakeresésben utoljára véglegesített csúcs távolsága t-től. Ezen kívül tároljuk az eddigi legjobb út jelöltre a közös csúcsot ($nodemin$) és az út összköltségét ($dmin$).

Két megvalósítást vizsgáltunk, hogy melyik irányú keresést léptetjük.

- Ha $ds \leq dt$ akkor $G_\uparrow$ gráfban keresünk, különben $G_\downarrow$ gráfban.

[3] A keresés a $G_\uparrow$ gráfban s-ből és $G_\downarrow$ gráf fordítottjában t-ből fut.

- Felváltva léptetjük az előre és hátra keresést.

A keresés leállási feltétele minden estben azonos: ha mindkét irányban az adott irányhoz tartozó következő csúcs távolsága nagyobb mint a tárolt legjobb út hossza. Vagyis ha $ds > dmin$ és $dt > dmin$ akkor a keresés leáll, hiszen biztosan nem találhatunk rövidebb utat.

Minden véglegesített csúcsot ellenőriznünk kell, hogy rajta keresztül kapunk-e jobb jelöltet. Vagyis az ellenőrzés során megvizsgáljuk, hogy teljesül-e $d_{\uparrow}(s, v) + d_{\downarrow}(t, v) < dmin$. Ha igen, akkor ez lesz $dmin$ új értéke, $nodemin$ értékét pedig v-re állítjuk.

Legjobb út jelölt ellenőrzésére három módszert implementáltunk. Egy v csúcsot ellenőrzünk, ha:

- Mindkét keresésnél véglegesítettük.
- Az egyik keresésnél véglegesítettük, a másik keresés esetén már a kupacban van, vagy volt.
- Mindkét keresés esetén a kupacban van vagy volt.

Megjegyzés. A második és harmadik esetben $d_{\uparrow}$ és $d_{\downarrow}$ értékek csak felső becslések a távolságokra.

Ez az osztály készíti el az utat s-ből $nodemin$-be és onnan t-be, majd a kettőt összefűzi egy élekből álló vektorba.

Készült egy változat, mely csúcsokon adott GPS koordinátákat felhasználva ad alsó becslést az éppen vizsgált v csúcs esetén $d_{\uparrow}(s, v) + d_{\downarrow}(t, v)$ értékére. Ha ez a becslés nagyobb mint $dmin$, akkor ezt a csúcsot véglegesítjük, de éleit nem relaxáljuk, hiszen jobb utat nem kaphatunk.

3.2.6. A feldolgozást felügyelő osztály

A feldolgozás a **CH** osztályban történik. A feldolgozás elindítása után a csúcsokat egy kupacba tesszük, melyek prioritása az élkülönbség.

Egy csúcs összehúzása előtt frissítjük a prioritását, azaz újra kiszámoljuk az élkülönbségét. Ha továbbra is ő a legkisebb elem a kupacban, akkor

összehúzzuk. Majd minden szomszédjára frissítjük a prioritásukat: növeljük a törölt szomszédok számát, kiszámítjuk a keresési hely nagyságát 2.4.4 alapján, végül az élkülönbséget. A feldolgozás során minden csúcsra tároljuk, hogy hanyadikként került összehúzásra.

Az összes csúcs összehúzása után elkészítjük a $G_\uparrow$ és $G_\downarrow$ gráfokat. Ehhez két Subdigraph segítségével elrejtjük a nem kellő éleket. Azokat az éleket, melyek $G_\downarrow$ gráfba tartoznak, megfordítjuk. Majd DigraphCopy segítségével átmásoljuk ezeket StaticDigraphokba.

Végül töröljük a már nem kellő Mapeket, a kupacot és a használt Dijkstra keresőket.

Több implementáció készült a prioritások meghatározásához.

- Egy esetben csak az élkülönbséget használjuk prioritási tényezőként.
- Készült egy véletlen összetevőt is használó változat.
  Itt minden csúcs prioritása a normál prioritás ezerszerese, és egy $[0, 1000)$ intervallumról vett véletlen szám összege.
- Vizsgáltuk, hogy érdemes-e összehúzni a kis fokszámú csúcsokat az elején.
- Teszteltük, hogy ha adottak a csúcsok GPS koordinátái, akkor a törölt szomszédok helyett ennek a segítségével biztosítjuk az összehúzandó csúcsok egyenletes elhelyezkedését a gráfban.
- Készült egy verzió, mely elején töröljük azokat az éleket, mely biztosan nem szerepel egy legrövidebb úton sem.

Ez az osztály biztosítja a kommunikációt a külvilág felé. Valamint ez az osztály alakítja vissza a talált rövidített utat csak eredeti éleket[4] tartalmazó úttá.

3.3. Fontosabb metódusok

Itt azokról az eszközökről lesz szó, ami a [1] cikkben nem egészen világosan vannak leírva, illetve ami ki van egészítve a feldolgozás jobb működéséhez.

[4] Azok élek, melyek az input gráfban szerepelnek.

A függvények mindegyike a **CH** osztályban található.

3.3.1. Élkülönbség

Az edgediff függvény, egyetlen bemeneti paramétere egy csúcs (v).

Az élkülönbség kiszámításakor vesszük a v csúcs kiéleinek maximális súlyát ($maxout$). A v csúcsot beállítjuk véglegesítettnek, így a keresés figyelmen kívül fogja hagyni. Ezután minden beél tövéből (u) indítunk egy **EdgeDiffDijkstra** keresést. A keresés leáll ha elértük $c(u,v)+maxout$ távolságot, vagy nincs több csúcs a Dijkstra kupacában. Majd minden megfelelő kiélre, aminek hegye w, megvizsgáljuk, hogy teljesül-e $d(u,w) < c(u,v) + c(v,w)$. Ha nem teljesül, vagy $d(u,w)$ nem ismert, mert w nem véglegesített, akkor a gráfhoz hozzáadjuk (u,w) élt $c(u,v) + c(v,w)$ súllyal.

A végén töröljük a gráfból a hozzáadott éleket, visszaállítjuk az **EdgeDiffDijkstra** adattagjait, majd visszatérünk a fokszám és behúzott élek számának különbségével.

3.3.2. Összehúzás

A *contract* metódus, egyetlen bemeneti paramétere egy csúcs (v). Az *edgediffhez* képest itt változik néhány dolog.

Ha be kell húznunk egy élt, akkor előtte megnézzük van-e vele párhuzamos él. Ha találunk ilyet, akkor ennek a súlyát változtatjuk meg és úgy kezeljük, mintha új él lenne. Ezt megtehetjük, mivel az él a régi súlyával biztosan nem szerepel legrövidebb útban.

Az új élen (e) tároljuk egy ArcMapben (*pack*), hogy mely két él helyett lett behúzva. Ha v összehúzása során be kell húznunk (u,w) élt, akkor $pack[e].first = (u,v)$ és $pack[e].second = (v,w)$. Ennek segítségével fogjuk visszakeresni az eredeti utat.

A végén nem töröljük a behúzott éleket mint az *edgediff* esetén. A csúcs törlése helyett a 3.2.1 részben említett módszert alkalmazzuk: a kereső osztályokban a csúcsot véglegesítettnek állítjuk be, így a további keresések alkalmával nem lesznek figyelembe véve.

3.3.3. Utak visszakeresése

Az *unpack* metódus. Bemenete egy élekből álló vektor (*path*), és egy él (*e*).

Ha az él eredeti, azaz nem tároltunk rajta élpárt, akkor hozzáadjuk a vektorhoz. Ha behúzott él, akkor rekurzívan meghívjuk a metódust a vektorral és az első részéllel, majd ugyan így a második részéllel. Az algoritmus a következő:

```
unpack(path, e):
    if (pack[e] == null):
        path.push_back(e)
    else:
        unpack(path, pack[e].first)
        unpack(path, pack[e].second)
```

3.3.4. Utak lekérdezése

A *getPath* függvény. Nincs bemeneti paramétere. Az út eredeti éleinek vektorával tér vissza.

Itt lekérdezzük a **CHSearch** útját, ami tartalmazhat rövidítő éleket. Létrehozunk egy üres vektort (*path*). Majd sorban minden *e* élre meghívjuk az *unpack(path, e)* metódust. Az élek referenciáinak helyességére ügyelnünk kell, ehhez szükség van a két részút hosszára.

A hátrafelé élek kibontása itt gondot jelentene, mivel a keresőgráfok létrehozásakor ezeket megfordítottuk. Azonban ezzel a tárolt két él sorrendje nem változik meg, így a kibontás jól működik.

3.4. Használat

A használathoz létre kell hoznunk egy **CH** objektumot. A konstruktorhoz kell egy ListDigraph és egy ArcMap az élsúlyokkal. Ha a koordinátákat használó implementációt használjuk, akkor a szélességi és hosszúsági koordinátákat tartalmazó NodeMapeket is itt kell megadni.

A *run* metódus feldolgozza a gráfot. Nagy gráf esetén ez sokáig tarthat. Az eredeti gráf ezzel megváltozik.

Az *addSource* és *addTarget* metódusok a kezdő és célcsúcsot várják paraméterül.

A *runSearch* metódust kell meghívnunk a keresés futtatásához. Itt több lehetőségünk van. Ha két csúcsot adunk paraméterként, akkor a két csúcs közt keres. Ha paraméter nélkül hívjuk meg, akkor az *addSource* és *addTarget* esetén megadott csúcsok közt keres.

Ha futtattuk a keresést, akkor a *dist* függvény adja vissza a két csúcs közti legrövidebb út hosszát, a *getPath* függvény pedig a legrövidebb út éleit tartalmazó vektorral tér vissza.

Ha új keresést akarunk futtatni, akkor elegendő meghívni a *clear* metódust, a gráfot nem kell újra feldolgozni.

Lentebb található egy lehetséges kódrészlet. Feltételezzük, hogy g a gráf, *length* az élsúlyokhoz tartozó ArcMap, s és t pedig a gráf csúcsai.

```
#include "CH.h"
...
int main() {
    ...
    CH ch(g, length);
    ch.run();
    ch.runSearch(s,t);
    int distance = ch.dist();
    vector<ListDigraph::Arc> path = ch.getPath();
    ...
}
```

Lehetőségünk van kihasználni az előző keresés eredményét is. Konkrétan ha az *addSource* metódust haszáljuk, akkor csak az előrekeresés eredményei törlődnek, a hátrakeresés érintetlen marad, a tárolt távolságok megőrződnek. Így a keresés is gyorsabb lesz.

Az összehúzáshoz használt metódusok publikusak, így akár kívülről is lehetőségünk van megadni, milyen sorrendben legyenek összehúzva a csúcsok.

4. Tesztek

Több különböző változata készült az implementációnak. A legtöbb apróbb módosításokat tartalmaz a 2. fejezetben leírtakhoz képest. A 3. fejezetben írtunk az elkészült implementációkról. Ebben a részben ezeket a verziókat hasonlítjuk össze egymással, valamint egy egyszerű Dijkstra algoritmussal.

A teszteléshez a DIMACS Implementation Challenge [10] oldalán található gráfokat, valamint egy, Magyarország úthálózatát tartalmazó gráfot használtuk fel.

A teszteket egy 2 gigabyte memóriával, és egy Pentium T3200 processzorral rendelkező laptopon futtattuk. A programot 4.6.3 verziójú g++ fordítóval fordítottuk Ubuntu 12.04 operációs rendszeren, 3.2.0-41 Linux kernel alatt.

Az elkészült programok és a mérések megtalálhatóak a `http://www.cs.elte.hu/~godsaat/ch/` weboldalon.

4.1. Mérési módszerek

A keresés több tulajdonságát mértük. Ebben a szakaszban a kidolgozott tesztekről lehet bővebben olvasni.

4.1.1. A feldolgozás

A feldolgozás ideje nem túl fontos tényező. A keresések futtatásához elegendő egyszer feldolgozni a gráfot. Valamint a feldolgozott gráfot is el lehet menteni későbbi használatra.

Azonban érdekes lehet, hogy mennyi új élt kellett behúznunk a különböző módszerekkel. Valamint, hogy a csúcsok összehúzási sorrendje mennyiben befolyásolja a keresés gyorsaságát.

4.1.2. A keresés ideje

A legfontosabb tényező, egyben a dolgozat célja is a legjobb keresési idő elérése. Itt megmértük, hogy az összes tesztesetet mennyi ideig tartott lefuttatni és ebből számoltunk átlagot. Külön vizsgáltuk, hogy mennyi ideig

tart a teszt futása, ha az utakat is visszakeressük. Mindkét esetben 1000 véletlenül kiválasztott csúcspárra futtattuk a teszteket.

A mért idő tartalmazza a keresés inicializálását is, azonban ez a Dijkstra keresés esetén is fenn állt.

A Dijkstra teszt volt az összehasonlítási alap. A 3.2.4. fejezetben említett saját Dijkstra implementációt használtuk StaticDigraph adatszerkezeten, hogy gyorsabb futást biztosítsunk, és csökkentsük az inicializáció idejét az egyes tesztek között.

4.1.3. Felhasznált tár

A felhasznált tárat főleg a kevés memóriával rendelkező mobil eszközök miatt fontos vizsgálni. Azonban pontos meghatározása meglehetősen problémás.

Vizsgálhatjuk, hogy mennyi tárat foglalnak a keresőgráfok és éleiken a súlyok. Ez elegendő információ két csúcs közti távolság meghatározására. Azonban ha az utat is vissza szeretnénk keresni, esetleg térképen kiszínezni a reprezentált útszakaszokat, akkor az eredeti gráf éleire is szükségünk van.

Mivel maga a teszt és a CH létrehozása közben tárolt objektumok is sok memóriát foglalnak, így a méréshez a következő módszert alkalmaztuk. A beolvasott gráfból létrehoztunk egy StaticDigraphot, amin a Dijkstra keresést futtattuk, és megmértük a kereső helyfoglalását a létrehozott gráffal és Mapekkel. A CH elkészültével pedig megmértük a futó program memóriafoglalását, elkészítettük a keresőgráfokat és a keresőosztályt, majd ismét megmértük a memóriafoglalást. Így a két mérés különbségével megkaptuk, hogy ha csak a távolságot szeretnénk lekérdezni, akkor mennyi helyre lenne szükségünk.

4.1.4. Véglegesített csúcsok száma

A CH előnye, hogy a keresési tartomány (searchspace) sokkal kisebb, mint más módszerek esetén. Ezt szeretnénk megfigyelni a véglegesített csúcsok számának segítségével.

4.1.5. Távolság és időtartam

A felhasznált gráfok közül néhány esetén rendelkezésre áll adott élen a reprezentált út hossza, valamint az út megtételéhez szükséges idő is. Így megfigyelhetjük, hogyan viselkedik a CH, ha legrövidebb vagy leggyorsabb utat keresünk.

Ezen kívül szeretnénk megtudni, hogy mennyivel lesz gyorsabb a keresés CH segítségével, ha a keresés két közeli csúcs között fut, vagy esetleg távoli célpontokat keresünk. Megvalósításként a tesztek eredményeit rendeztük a Dijkstra algoritmus által véglegesített csúcsok számának alapján. Az első 5%-ot tekintettük közeli csúcspároknak, az utolsó 75%-át a méréseknek pedig távoli csúcspároknak.

4.1.6. Útvonal újratervezése

Ha utazás során letérünk az ajánlott útról a GPS újratervezi azt. Ez CH segítségével várhatóan gyorsabb lesz, hiszen a meglévő hátrafelé keresést teljes egészében felhasználhatjuk, míg egy Dijkstra esetén újra kell kezdenünk a keresést.

A mérés során a kezdeti útkeresés idejét és az újratervezett út idejét egyszerre kellett mérnünk, hogy fel tudjuk használni a CH-ban a hátrakeresés eredményeit.

Ezt csak abban az esetben teszteltük, amikor az utakat vissza is kerestük, hiszen gyakorlatban az útvonal újratervezésekor erre is szükségünk van.

4.2. A változatok összehasonlítása

Itt található részletes leírás az elkészült különböző verziókról, amiket már a 3. fejezetben említettünk, valamint a hozzájuk tartozó mérési eredményekről.

Az alábbi gráfokat használtuk a tesztekhez:

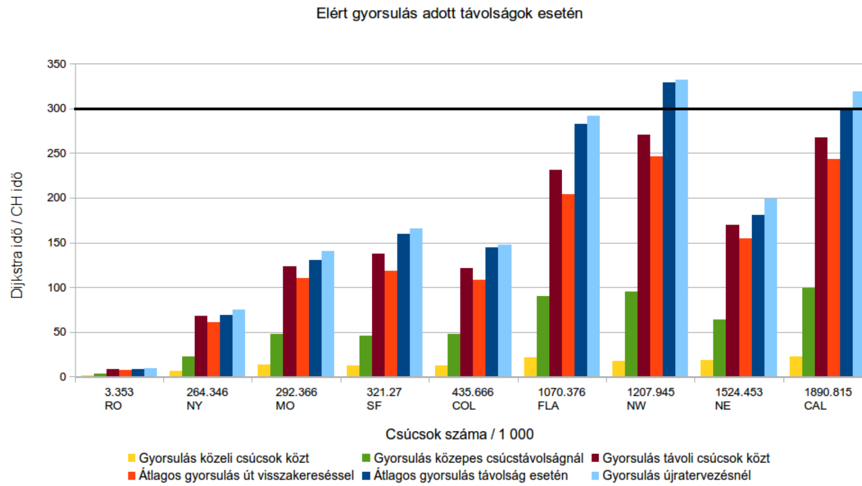
Hely	Rövidítés	Csúcsok	Élek
Róma	ROM	3353	8870
New York	NY	264346	733846
Magyarország	MO	292366	777988
San Francisco	SF	321270	800172
Colorado	COL	435666	1057066
Florida	FLA	1070376	2712798
Északnyugat USA	NW	1207945	2840208
Északkelet USA	NE	1524453	3897636
Kalifornia	CAL	1890815	4657742
Kelet USA	EUSA	3598623	8778114

1. táblázat. A felhasznált gráfok

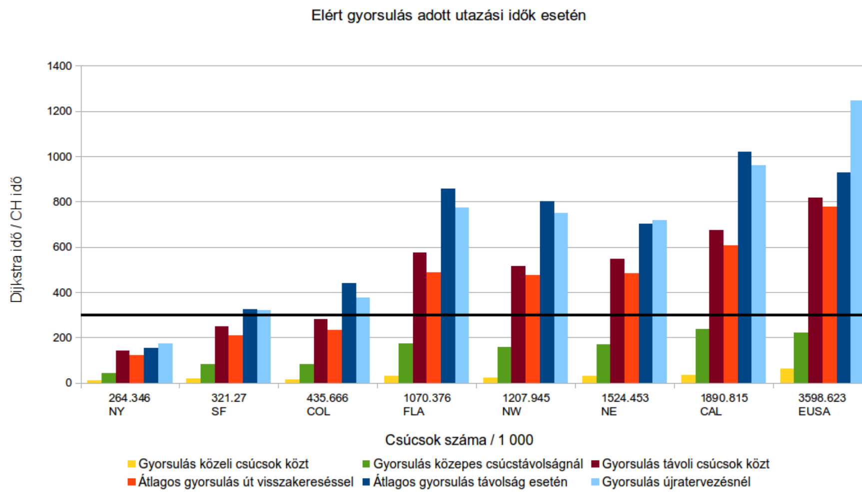
4.2.1. Az alap verzió

Ez az implementáció kihasználja a legtöbb prioritási tényezőt: élkülönbség hop limit használatával (2.4.2), törölt szomszédok (2.4.3), becslés az utak élszámára (2.4.4). A keresést a gráfokon felváltva egyesével léptetjük és a mindkét keresésben véglegesített csúcsokat ellenőrizzük. Ezt az implementációt használjuk kiindulási pontként a többi esetén.

A következő ábrákon a Dijkstrához képest elért gyorsulást ábrázoltuk a gráf mérete szerint.



1. ábra. Elért gyorsulás éleken adott távolságok esetén



2. ábra. Elért gyorsulás éleken adott utazási idők esetén

Tehát a következő eredményt kaptuk. A CH használatával jelentős, nagy

gráfokon több mint 200-szoros gyorsulást sikerült elérni, ha legrövidebb utat kerestünk. Azonban leggyorsabb utak keresése esetén sikerült elérni az 1000-szeres gyorsulást a Dijkstra algoritmushoz képest.

Az ábrák alapján jól látható, hogy közeli csúcspárok közt nem sokkal lett gyorsabb a CH, távolabbiak közt sokkal érdemesebb használni.

A másik meglepő eredmény, hogy az utak visszakeresése majdnem annyi időt vesz igénybe, mint az út megtalálása.

Várakozásainknak megfelelően az útvonal újratervezésekor is jelentős gyorsulást sikerült elérni a Dijkstra algoritmushoz képest.

Az EUSA gráf tesztjén az átlagidő, ha nem kerestük vissza az utakat, lassabb lett mint CAL esetén. Ezen a gráfon csak 100 tesztet futtattunk, így ennek oka lehet mérési hiba, vagy rosszul megválasztott csúcspárok. Azonban minden más mért adat jobb lett, így jól megfigyelhető, hogy a CH nagyobb gráfokon gyorsabban teljesít.

4.2.2. Módosított verziók tesztjei

A következő teszteken csak apróbb módosítások történtek az alap változathoz képest. A teszteket a Magyarországot ábrázoló gráfon és a New York úthálózatát ábrázoló gráfon futtattuk.

1. Távolság szerinti lépés
   Ebben az implementációban a gráf feldolgozása megegyezik az alap verzióval, azonban a keresést nem felváltva léptetjük, hanem a 3.2.5. fejezetben említett másik módszerrel: az előre irányú keresést léptetjük, ha az ott utoljára véglegesített csúcs távolsága a kezdőponttól nagyobb mint a hátrafelé keresés utoljára véglegesített csúcsának távolsága a végponttól.

2. Élkülönbség
   Ez a verzió csak az élkülönbséget használja a prioritások megállapításához. Így mutatjuk meg, hogy van alapja a 2.4.3. fejezetben lévő állításnak. Vagyis már az élkülönbség is elég jó, de fontos az egyenletesség is a csúcsok összehúzásánál.

3. Hop limit

   Ezzel a teszttel vizsgáljuk, hogy a 3.2.3. fejezetben említett módszer a CH elkészítésének gyorsítására mennyire befolyásolja a keresés gyorsaságát. Erre két változat készült.

   (a) Első esetben nem használtuk a hop limitet, vagyis az élkülönbséget pontosan határoztuk meg.

   (b) Másik esetben a véglegesített csúcsok számát korlátoztuk 50-re az utak élszámának korlátozása helyett.

4. Kis fokszámú csúcsok

   Ebben az implementációban a feldolgozást az ötnél kisebb fokszámú csúcsokkal kezdtük véletlen sorrendben.

5. Nem kellő élek

   Ebben a verzióban vizsgáltuk, hogy érdemes-e a feldolgozás előtt törölni azokat az éleket, melyeken biztosan nem megy át legrövidebb út.

6. Véletlen összetevő

   Ez a teszt a 3.2.6. fejezetben említett módon a prioritások meghatározásánál használ véletlen összetevőt is. Így az eredetileg azonos prioritású csúcsok várhatóan véletlen sorrendben lesznek összehúzva. Azonban itt nem tudjuk használni a lusta frissítést (lásd 2.4.1. fejezet).

7. Csúcs vizsgálat: két kupac

   Ebben a verzióban minden csúcsot megvizsgáltunk, ami mindkét irányban a kupacba került, hogy rajta keresztül rövidebb utat kapunk-e.

8. Csúcs vizsgálat: egy kupac

   Itt azokat a csúcsokat vizsgáltuk, mely az egyik irányban véglegesített, a másik irányban pedig vagy véglegesített, vagy még a kupacban van.

9. Adott GPS koordináták

   A Magyarország gráfon adottak a csúcsok GPS koordinátái. Ez alapján a törölt szomszédok helyett 2.4.3 alapján az egyik prioritási tényező a csúcsok sorrendjének meghatározásakor a földrajzi távolság lesz a

már összehúzott csúcsoktól. A koordinátákat továbbá a keresésben is felhasználhatjuk 3.2.5 alapján: egy véglegesítendő v csúcsra alsó becslést kapunk s és v, valamint v és t között.

(a) Csak a CH elkészítésénél használjuk a koordinátákat.

(b) A keresésnél is használjuk a koordinátákat.

10. Lokális legrövidebb utak
    Ebben a verzióban minden csúcsra meghatároztuk, hogy lokálisan hány darab legrövidebb út megy át rajtuk. Erről kicsit részletesebben a 4.3.3. fejezetben írtunk. A csúcsok prioritásában a tényező mellett használtuk az élkülönbséget és törölt szomszédokat is.

11. A 3b, 5 és 8. verziók kombinációja
    A teszteken ezek a verziók szerepeltek jobban, mint az alap verzió, így megvizsgáltuk, hogy együtt mennyivel teljesítenek jobban.

A következő mérési eredmények születtek a Magyarország úthálózatát ábrázoló gráfon. A véglegesített csúcsok és idő alatt a mérések átlagát értjük. Az időt mikroszekundumban tüntettük fel.

Változat	Behúzott élek	Vég. csúcsok	Idő	Gyorsulás
Dijkstra	0	146544	64597,3	1
Alap verzió	851322	590,215	496,792	130
1	851322	764,093	573,15	112,7
2	846672	1638,53	1287	50,19
3a	846987	585,699	486,89	132,67
3b	850906	536,738	448,271	144,1
4	877720	676,403	597,79	108
5	830647	591,052	489,374	132
6	852915	759,964	585,802	110,27
7	851322	593,648	535,393	120,65
8	851322	594,307	495,786	130,29
9a	753702	1314,26	1022,33	63,18
9b	753702	1314,26	1721,19	37,53
11	831486	529,213	445,944	144,85

2. táblázat. Változatok a Magyarország gráfon

A következő táblázat a New York utazási idő gráfon mért adatokat tartalmazza:

Változat	Új élek	Vég. csúcsok	Idő	Gyorsulás
Dijkstra	0	129299	38238,3	1
Alap verzió	823655	385,508	250,45	152,67
1	823655	459,95	270,027	141,6
2	821972	838,546	425,742	89,81
3a	821275	383,505	242,147	157,91
3b	824854	377,788	246,676	155
4	805971	368,511	236,667	161,57
5	819747	378,582	243,805	156,83
6	820951	442,767	265,767	143,87
7	823655	383,869	270,045	141,6
8	823655	384,67	248,491	153,88
10	994203	637,124	371,522	102,92
11	820578	390,035	253,317	150,95

3. táblázat. Változatok a New York gráfon

A tesztek alapján a következő tapasztalataink születtek.

A 4. és 5. verzió tesztjei alapján úgy tűnik, hogy a kis fokszámú csúcsokat, és a nem kellő éleket érdemes az összehúzás előtt kitörölni, ha leggyorsabb utat keresünk.

A 8. verzió méréseiben láthatjuk, hogy bár nem sokkal, de gyorsabb lett a keresés, ha az olyan csúcsokat is megvizsgáltuk, melyek csak az egyik keresésben lettek véglegesítve.

A 3. verziók tesztjei azt mutatják, hogy az élkülönbség meghatározásának pontatlansága nem befolyásolja nagyban a keresés gyorsaságát. Azonban hop limit helyett a véglegesített csúcsok számának korlátozásával meglepően jobb eredményt kaptunk, főként ha az éleken a távolságok voltak megadva. Ennek oka valószínűleg az lehet, hogy hop limit használatával az első néhány

összehúzandó csúcs élkülönbségét kevésbé pontosan határozzuk meg. Ekkor nem lesz 50 véglegesített csúcs. Meglepően a 3b verzióban a feldolgozás ideje is nagyjából a felére csökkent az alap verzióhoz képest. Ennek pedig az lehet az oka, hogy a feldolgozás vége felé a G' gráf[5] sokkal sűrűbb, így a hop limites megoldással 50-nél jóval több véglegesített csúcs lesz.

Az 1. verzió tesztje szerint pedig érdemes a keresést felváltva egyesével léptetni, és nem távolság alapján.

A 9. verziók tesztjei érdekes eredménnyel szolgálnak. Jól látható, hogy adott GPS koordináták segítségével a CH-ban sokkal kevesebb új él lett behúzva, és a feldolgozás ideje több mint a felére csökkent. Azonban a keresés lassabb lett. A keresésben is kihasználva a koordinátákat nem sikerült jobb eredményt kapnunk.

A 10. verzió esetén a feldolgozás a Magyarország gráfon több óráig tartott, így éleken adott távolságok esetén ez a módszer nem elég hatékony. Adott utazási idők esetén első tesztek alapján nem adott jobb eredményt és a feldolgozás ideje is a duplájára nőtt. Azonban érdemes lehet a későbbiekben jobban összehangolni más prioritási tényezőkkel.

Legrövidebb útkeresés esetén a 11. verzió volt a leggyorsabb. Ugyan ez a verzió leggyorsabb út keresésére nem sokkal volt lassabb, mint az alap verzió a tesztelt gráfon. Ez alapján látható, hogy a későbbiekben érdemes lesz megvizsgálni, hogyan viselkednek a különböző verziók kombinációi.

4.3. CH és transit node kombinációja

Készült egy implementáció, mely a 1.3.2. fejezetben említett transit node keresésre és a CH-ra épül. Ezzel mutatjuk meg, hogy a CH-t más módszerekkel is viszonylag könnyen lehet kombinálni és gyorsabb keresésre számíthatunk.

4.3.1. A feldolgozás

A gráf feldolgozása egy ideig megegyezik a 3.2.6. fejezetben ismertetett módszerrel. Azonban mikor az utolsó néhány n (az implementációban $n =$

[5] Amiben a még nem törölt csúcsok és a köztük lévő eredeti és már behúzott élek szerepelnek.

$\left\lfloor \sqrt{|V|} \right\rfloor$) csúcs kerülne feldolgozásra, másként folytatjuk az eljárást. Ezek a csúcsok lesznek a transit node-ok. Jelöljük ezt a halmazt T-vel. [1] és [2] alapján érdemes így választani a halmazt. Készült egy változat, ahol előre választjuk ki a transit node-okat, erről a 4.3.3. fejezetben írtunk.

Létrehozunk egy $n \times n$ méretű M mátrixot. Minden csúcsból[6] indítunk egy Dijkstra keresést, és a mátrixban tároljuk minden csúcspár távolságát. Ehhez a csúcsokat megindexeljük egy 0 és $n-1$ közti számmal. Így $M_{i,j}$ lesz az i-edik és j-edik transit node közti távolság. Használjuk v, w csúcsok esetén az $M_{v,w}$ jelölést a két csúcs távolságára. Mivel irányított gráfról van szó, így a mátrix nem feltétlenül szimmetrikus.

A keresőgráfokat létre tudjuk hozni a 3.2.6. szakasznak megfelelően. Ehhez az utolsó n csúcs sorrendje nem számít, kizárólag az összehúzott csúcsok sorrendjétől[7] kell nagyobbnak lenniük. Ez azért fontos, hogy a csúcshalmazba érkező és onnan kilépő élek a keresőgráfokban megfelelően szerepeljenek. Vagyis a beérkező élek $G_\uparrow$ gráfban jelenjenek meg, a kivezető élek pedig $G_\downarrow$ gráfban.

4.3.2. A keresés

A keresés is módosul a 3.2.5. fejezetben tárgyaltakhoz képest. A két irányt felváltva léptetjük, ez bizonyult a gyorsabb megoldásnak a fenti tesztek alapján.

Ha a következő véglegesítendő csúcs nem transit node, akkor, mint eredetileg, véglegesítjük, éleit relaxáljuk, majd ellenőrizzük, hogy jó jelöltet ad-e legrövidebb útra.

Ha a csúcs transit node, akkor véglegesítjük, de az éleit nem relaxáljuk. Helyette megvizsgáljuk, hogy a csúcs és a másik irányban véglegesített transit node-ok adnak-e legrövidebb útra jelöltet.

Formálisan, felhasználva 2.5. szakasz jelöléseit, ha éppen v csúcsot véglegesítjük az előre keresésben, $v \notin T$, és v-t már láttuk[8] a hátrafelé keresés során, akkor a következő lehetséges jelölt a legrövidebb út költségére

[6] A gráfban már csak a nem összehúzott csúcsok szerepelnek.
[7] Egy csúcs sorrendjén azt értjük, hogy a csúcsot hanyadikként dolgoztuk fel.
[8] A csúcs benne van vagy volt a keresés kupacában.

$d(s,t) = d_{\uparrow}(s,v) + d_{\downarrow}(v,t)$. Ha $v \in T$, akkor a lehetséges jelöltek $d(s,t) = d_{\uparrow}(s,v) + M_{v,w} + d_{\downarrow}(w,t)$ alakúak, ahol w véglegesített transit node a hátrafelé keresésben. Hasonlóan járunk el ha v-t a hátra keresésben véglegesítjük.

4.3.3. Transit node-ok előre kiválogatása

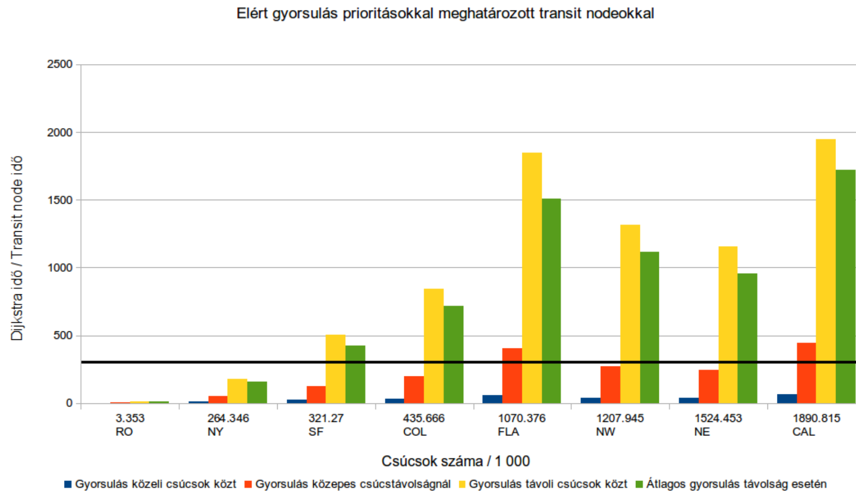
Készült egy verzió, melyben a T halmazt a csúcsok összehúzása előtt határoztuk meg. Ehhez minden csúcsra megvizsgáltuk, hogy lokálisan hány darab legrövidebb út megy át rajta.

Minden s csúcsból indítottunk egy Dijkstra keresést $n = \sqrt{|V|}$ lépésre korlátozva. Amikor véget ért egy keresés, akkor minden $v \neq s$ véglegesített csúcsra meghatároztuk, hogy hány másik csúcsba vezető legrövidebb úton szerepel az adott keresőben. Ez a szám az s gyökerű legrövidebb utak fájában a v gyökerű részfa csúcsszáma mínusz egy. Ezt jelöljük $l_s(v)$-vel.

Miután lefutott az összes Dijkstra keresés, minden v csúcsra $L(v) = \sum_{s \in V} l_s(v)$. Az az n darab csúcs kerül T halmazba, melyekre ez az érték nagy. A gráf feldolgozása és a keresés innentől az előzőek alapján megy.

4.3.4. A teszt eredménye

A következő diagramon szemléltetjük a keresés gyorsulását a Dijkstra algoritmushoz képest.



3. ábra. Elért gyorsulás transit node és CH kombinációjával

Itt a következőket tapasztalhatjuk. Kis városokban a módszer kevésbé jól működik. A legkisebb tesztelt gráfon (Róma, 3353 csúcs) a gyorsulás elhanyagolható. Nagy eltérés figyelhető meg a közeli és távoli csúcspárok közti keresés esetén. Egész államokat ábrázoló gráfokon a távoli csúcsok közti keresés sokkal gyorsabb lett, azonban közeli csúcsok esetén a javulás nem túl jelentős.

A CH-hoz képest nagyobb gráfokon nagyjából másfélszeres gyorsulást sikerült így elérni.

A csúcsok előre kiválogatásáról a következő táblázatban foglaltuk össze a mért gyorsulást. A gráf méretét csúcsok számával adtuk meg, a gyorsulást a Dijkstra algoritmushoz képest vettük.

Gráf méret	CH	Transit	Előreválogatott Transit
3353	8,77	11,1	7,47
264346	152,08	156,91	144,86
321270	325,4	421,84	331,59
435666	437,8	716,51	460,53
1070376	855,72	1505,14	625,56

4. táblázat. Előre kiválogatott transit node eredményei

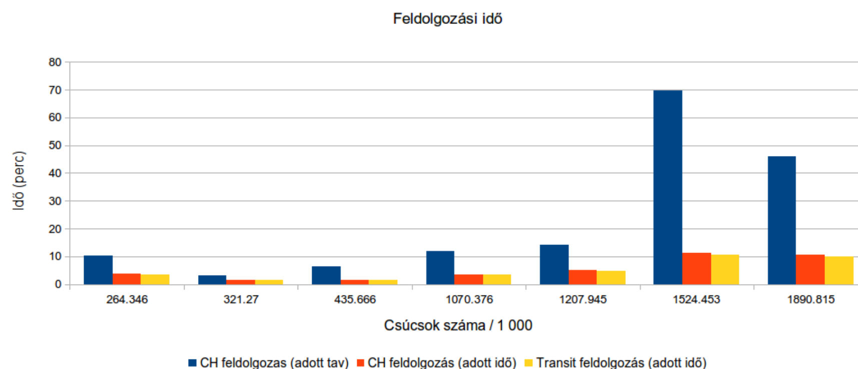
Ezek alapján úgy tűnik, hogy nem érdemes a transit node-okat előre meghatározni. Legalábbis az általunk ismertetett módszer nem adott jobb eredményt, mint ha a CH által fontosnak ítélt csúcsokat tekintenénk transit node-oknak. A későbbiekben érdemes lehet más módszereket keresni a csúcsok előre kiválogatásához.

4.4. Egyéb mérési eredmények

Ebben a fejezetben ismertetjük a feldolgozással és élszámokkal kapcsolatos tesztek eredményét.

4.4.1. A feldolgozás ideje

A feldolgozás idejével kapcsolatos méréseket a következő diagram szemlélteti:

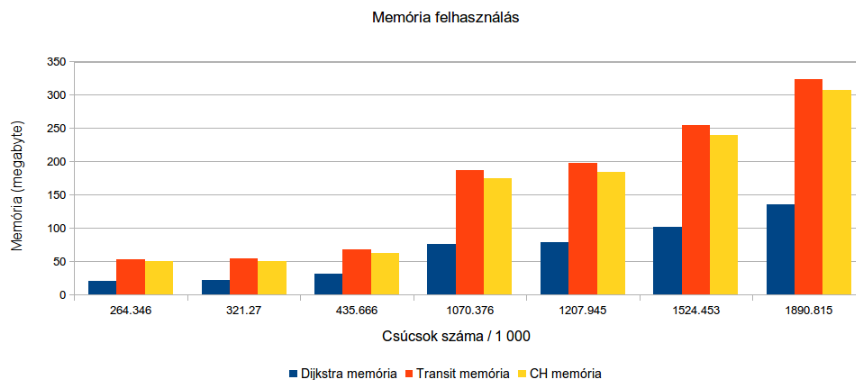


4. ábra. Feldolgozási idők

A következő meglepő eredményt kaptuk. Az éleken adott úthosszak esetén a feldolgozás ideje többszöröse, mint mikor az utazási idő gráfokat vizsgáltuk. Ekkor, mint látható, a feldolgozás viszonylag nagy gráfokon is viszonylag kevés időt vesz igénybe. A transit node és CH kombinációjának feldolgozása pedig még gyorsabb, mint az egyszerű CH feldolgozása utazási idők esetén.

4.4.2. Memória használat

A memóriahasználat mérését 4.1.3. fejezetben ismertettük. Az eredményeket az alábbi diagram szemlélteti:



5. ábra. Memória használat

A CH tehát nagyjából két és félszer annyi memóriát igényel, mint a Dijkstra algoritmus az implementációnkban. Azonban a LEMON korlátai miatt nem tudtuk kihasználni a 2.5.1. fejezetben említett memória felhasználást csökkentő technikákat. A mérések alapján a transit node-os megoldás nem igényel sokkal több memóriát, mint a CH, ha csak a távolságokat szeretnénk meghatározni.

4.4.3. Behúzott élek száma

A mérések során megfigyeltük a behúzott élek számát. Itt minden esetben azt tapasztaltuk, hogy nem húztunk be több élt, mint ahány eredetileg volt. Tehát a CH élszáma nem nőtt az eredeti élszám kétszeresénél nagyobbra.

A éleken adott távolságok esetén minden esetben több élt kellett behúznunk. A transit node-os megoldás esetén pedig az utoljára feldolgozandó csúcsok közt egyáltalán nem húztunk be új éleket.

4.4.4. A legrövidebb utak élszáma

Az NW, NE és CAL gráfokon az éleken adott utazási idők és távolságok esetén is megmértük, hogy a talált legrövidebb utak hány élt tartalmaztak. Az összes esetben esetben azt tapasztaltuk, hogy a talált utak átlagosan 15 élből álltak. Ezzel szemben a kibontott, csak eredeti éleket tartalmazó utak több mint 1000 élt tartalmaztak.

4.5. A kiugró eredmények magyarázata

Ebben a fejezetben megpróbáljuk megmagyarázni az előbb ismertetett mérési eredmények közül a meglepő és kiugró eredményeket.

4.5.1. Távolság és idő

A tesztek alapján jól látható, hogy az éleken adott távolságok esetén a CH elkészítése és a keresés is jóval lassabb, mint ha az utazási időket adták volna meg.

Néhány tesztünkben megfigyeltük, hogy hányszor kell lusta frissítést végeznünk (lásd 2.4.1. fejezet). Kiderült, hogy ilyenből az éleken adott távolságok esetén 1,2-szer több kellett. Ezen kívül összességében is több él lett ekkor behúzva.

Megvizsgáltuk a csúcsok prioritását is. Kiderült, hogy az éleken adott utazási idők esetén az utolsó $\sqrt{n}$ csúcs prioritása sokkal kisebb, mint adott távolság esetén. Vagyis ha az éleken a távolságokat adták meg, akkor a

CH sokkal több csúcsot ítélt fontosnak. Ez látható a véglegesített csúcsok számán is 4.2.2. fejezetben. Így érthető, hogy a keresés is lassabb lett.

Összességében a következő mondható el. A CH az úthálózat hierarchiáját használja ki. Ez a hierarchia azonban adott távolságok esetén nem látszik. A szántómezőn átvezető egyenes földút előnyösebb távolság szempontjából mint a szomszéd falun átvezető főút.

Ezt a gondolatot tovább vihetjük. Egy városban ha két csúcs közt a távolságot tekintjük, akkor a keresés során több véglegesített csúcs lesz, mintha leggyorsabb utat keresnénk, hiszen előbbi esetben a sebességkorlátozásokat nem vesszük figyelembe.

4.5.2. Különleges gráf

4.2.1. fejezetben látható, hogy az NE gráfon érdekes módon egyik esetben sem nőtt a gyorsulás a gráf méretének megfelelően.

Ennek magyarázata az lehet, hogy a gráf kicsivel sűrűbb, mint a többi vizsgált gráf. Az éleken adott utazási idők esetén a különbség nem volt túl jelentős. Ebben az esetben a legrövidebb utak élszámát megfigyelve azt tapasztaltuk, hogy az nagyjából ugyan annyi, mint NW és CAL gráfok esetén. Azonban a kibontott utak élszáma sokkal kevesebb volt.

A transit node és CH kombinációja esetén az előbbi különbség nagyítódik fel.

Az éleken adott távolságok esetén pedig igaz az előző fejezetben említett probléma: a CH nem ad jó eredményt legrövidebb út esetén. Az NE gráf pedig az Egyesült Államok északkeleti részét ábrázolja, ahol több nagyváros van mint a többi vizsgált gráf esetén.

Elképzelhető, hogy más prioritási tényezők segítségével javítható a CH a kiugró esetekben is. Ez további vizsgálódást igényel a későbbiekben.

4.6. Alkalmazási és továbbfejlesztési lehetőségek

A fent ismertetett mérési eredményekből kiderült, hogy a gráf előfeldolgozásával a Dijkstra keresés sebessége jelentősen javítható. Ebben a fejezetben

áttekintjük a módszer alkalmazásainak lehetőségeit, és hogy hogyan lehetne tovább fejleszteni a későbbiekben a módszert.

4.6.1. Az előkészítés és keresés

4.2.2. fejezetben megmutattuk, hogy a csúcsok sorrendjének meghatározása erősen befolyásolja a keresés gyorsaságát. Így a későbbiekben érdemes lehet más prioritási tényezőket vizsgálni, valamint a már teszteltekek jobban összehangolni.

Ugyanitt láthattuk, hogy adott GPS koordináták mellett kevesebb élt tartalmazó CH készíthető, azonban a keresés a módszerünkkel lassabb lett. Elképzelhető azonban, hogy más tényezőkkel kombinálva a földrajzi távolságokat ez javítható. A keresés esetén pedig érdekes volna megvizsgálni, hogy javítaná-e a keresés sebességét, ha Dijkstra algoritmus helyett A* algoritmust használnánk az előre és hátrakeresés során.

A transit node és CH kombinációja esetén az implementációban egyenlőre nem szerepel az utak visszakeresésének lehetősége. Ehhez valószínűleg tárolnunk kellene a transit node-ok közti legrövidebb út éleit, mely azonban sok memóriát igényelne, így a módszert csak nagyobb kapacitással rendelkező eszközökön lehetne eredményesen használni. Esetleg a talált legrövidebb út két trasit transit node-ja között használhatjuk a CH-t az útvisszakereséshez, mely várhatóan gyors lesz.

4.6.2. Memória használat javítása

4.4.2. fejezet alapján látható, hogy a memóriahasználaton lehetne javítani. Ehhez a LEMON-ben kellene a CH-nak megfelelő gráfimplementációt létrehozni.

4.6.3. Távolság és gyorsaság

4.2.1. fejezetben láthattuk, hogy közeli csúcspárok közt az eljárás nem sokat gyorsít, azonban távoli csúcspárok közt akár ezerszeres gyorsulás is elérhető.

Meglepve tapasztaltuk, hogy a módszer az éleken adott utazási idők esetén sokkal többet javít a Dijkstra sebességén, mint adott úthosszak esetén.

Sőt, 4.4.1. szakaszban megmutattuk, hogy a CH elkészítése is sokkal kevesebb időt vesz igénybe az éleken adott utazási idők esetén.

4.3.4. és 4.4.1. szakaszokban megmutattuk, hogy a transit node és CH kombinációja gyorsabb keresést tesz lehetővé kevesebb előfeldolgozási idővel, mint az egyszerű CH.

4.6.4. Alkalmazások

A tesztek alapján tehát elmondható, hogy a CH-t nem feltétlenül érdemes mindig használni.

Közeli célpontok esetén, például ha okostelefonunkon egy közeli kávézót keresünk, nem nyújt észrevehető gyorsulást.

Ha azonban éppen az autónk GPS eszközére hagyatkozunk nagyvárosi környezetben, akkor ki tudjuk használni a CH nyújtotta előnyöket, legfőképp ha letérünk a kijelölt útról és várnunk kellene az útvonal újratervezésére.

Ha pedig nemzetközi árufuvarozással foglalkozunk és nagy mennyiségű memória áll rendelkezésünkre, akkor a CH és transit node kombinációja segíthet a gyors útvonaltervezésben kontinentális méretű úthálózatokon is.

Hivatkozások

[1] Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Experimental Algorithms, LNCS 5038, (pp. 319-333). Springer Berlin Heidelberg.

[2] Delling, D., Sanders, P., Schultes, D., & Wagner, D. (2009). Engineering route planning algorithms. In Algorithmics of large and complex networks, LNCS 5515, (pp. 117-139). Springer Berlin Heidelberg.

[3] Bast, H., Funke, S., Sanders, P., & Schultes, D. (2007). Fast routing in road networks with transit nodes. Science, 316(5824), (pp. 566-566).

[4] Schultes, D., & Sanders, P. (2007). Dynamic highway-node routing. In Experimental Algorithms, LNCS 4525, (pp. 66-79). Springer Berlin Heidelberg.

[5] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. Systems Science and Cybernetics, IEEE Transactions on, 4(2), (pp. 100-107).

[6] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische mathematik, 1(1), (pp. 269-271).

[7] LEMON, https://lemon.cs.elte.hu/trac/lemon

[8] LEMON documentation, http://lemon.cs.elte.hu/pub/doc/1.2.3/index.html

[9] LEMON map,
http://lemon.cs.elte.hu/pub/tutorial/a00010.html#sec_digraph_maps

[10] 9th DIMACS Implementation Challenge (2006),
http://www.dis.uniroma1.it/challenge9/