Eötvös Loránd Tudományegyetem
Informatikai Kar
Számítógéptudományi Tanszék

Többtermékes folyam-algoritmusok

Király Zoltán
egyetemi docens

Bójás Zoltán
nappali tagozat
programtervező matematikus

Budapest, 2009.

A témabajelentő helye

Tartalomjegyzék

1. Bevezetés

Ez a fejezet a folyamalgoritmusok bemutatásával és a hozzájuk kapcsolódó feladattípusok ismertetésével foglalkozik. Az egyszerű maximális folyam problémán túl ismerteti a több termelős több fogyasztós, a minimális költségű, a többtermékes, az osztatlan folyam problémáját, valamint az ezekkel kapcsolatban felmerülő legfontosabb kérdéseket. Ezen kívül tartalmazza az ehhez szükséges fontosabb definíciókat is.

1.1. Hálózatok, folyamok

A folyamalgoritmusok, az irányított gráfokat "folyamhálózatokként" kezelik, azaz valamilyen anyag a gráf két kitüntetett pontja, a termelő és a fogyasztó, közötti áramlását modellezik. Példának vehetünk egy vízvezeték rendszert, vagy egy elektromos hálózatot. A termelő állandó ütemben állítja elő az áramoltatott anyagot (víz, áram, stb.), míg a fogyasztó állandó ütemben használja fel azt. Az irányított élek pedig a vezetékek, amin az anyag áramlik. Minden vezetéknek adott a kapacitása, aminél több anyagot nem tud elvezetni. A termelő és a fogyasztó kivételével minden csúcsban, a beáramló anyag mennyisége meg kell egyezzen a kiáramló anyag mennyiségével.

Hálózat: Legyen $G = (V, E)$ irányított gráf, $c : E \to \mathbb{R}$. Ha $\forall (u, v) \in E$ élen $c(u, v) \geq$ ≥ 0, valamint $s, t \in V$, akkor a (G, c, s, t) négyest hálózatnak nevezzük.

Folyam: A következő tulajdonságokkal rendelkező $f : E \to \mathbb{R}$ függvényt folyamnak nevezzük

- $f(u, v) \geq 0$
- **megmaradási szabály:** $\forall v \in V \backslash \{s, t\}$ csúcsra $\sum_{e=(u,v)\in E} f(e) - \sum_{e=(v,u)\in E} f(e) =$ $= 0$

Ahol s a termelő, t pedig a fogyasztó.

Az f folyam **megengedett**, ha tiszteletben tartja a kapacitásokat, azaz $f(e) \leq c(e)$.

A folyamokat vonatkoztathatjuk emberekre vagy árukra valamilyen közlekedési hálózatban, elektromos hálózatokra, de léteznek felhasználásai például az ökológia területén is,

ahol tekinthetjük a tápanyagok és az energia áramlását a táplálkozási láncban.

1.2. Maximális folyam

A **folyam értéke** az e élen $f(e)$, míg a **folyam nagysága** $\|f\| = \sum_{(s,v)\in E} f(e) - \sum_{(v,s)\in E} f(e)$ a termelőből kiinduló összes folyam.

A maximális folyam problémája az egyik legegyszerűbb folyamprobléma. Adott G hálózat, keressük az s-ből a t-be vezető maximális nagyságú, megengedett folyamot. Ennek a problémának számos polinomiális futási idejű megoldása ismert, például az Edmonds-Karp-algoritmus, vagy Goldberg és Tarjan általános előfolyam-algoritmusa, Goldberg és Tarjan [5].

1.3. Több termelő, több fogyasztó

A maximális folyam problémát általánosabban is megfogalmazhatjuk, úgy hogy van n termelő $s_1, s_2, \ldots, s_n$, ahol minden s_i termelő ugyanazt a terméket termeli, és m fogyasztó $t_1, t_2, \ldots, t_m$, és a termelőkből a fogyasztókba vezető maximális folyamot keressük.

Ez a probléma visszavezethető az egyszerű maximális folyam problémára a következőképpen. Vegyünk fel egy szupertermelőt t és egy szuperfogyasztót s majd kössük össze a fogyasztókkal és a termelőkkel végtelen kapacitású élekkel. Az s-ből t-be vezető maximális folyam nagysága pontosan az s_i termelők és t_j fogyasztók között maximálisan elvezethető folyamok nagyságának összegével lesz egyenlő.

1.4. Minimális költségű folyam

Rendeljünk a gráf éleihez költséget a következőképpen. $v : E \to \mathbb{R}$, ahol $v(e)$ annak a költsége, hogy az e élen egységnyi folyamot átáramoltatunk. Legyen $v(e) \geq 0$.

A folyam költségét ekkor a következőképpen definiáljuk $cost(f) = \sum_{e\in E} f(e)v(e)$

Így van értelme minimális költségű folyamot keresni, például minimális költségű maxi-

mális folyamot.

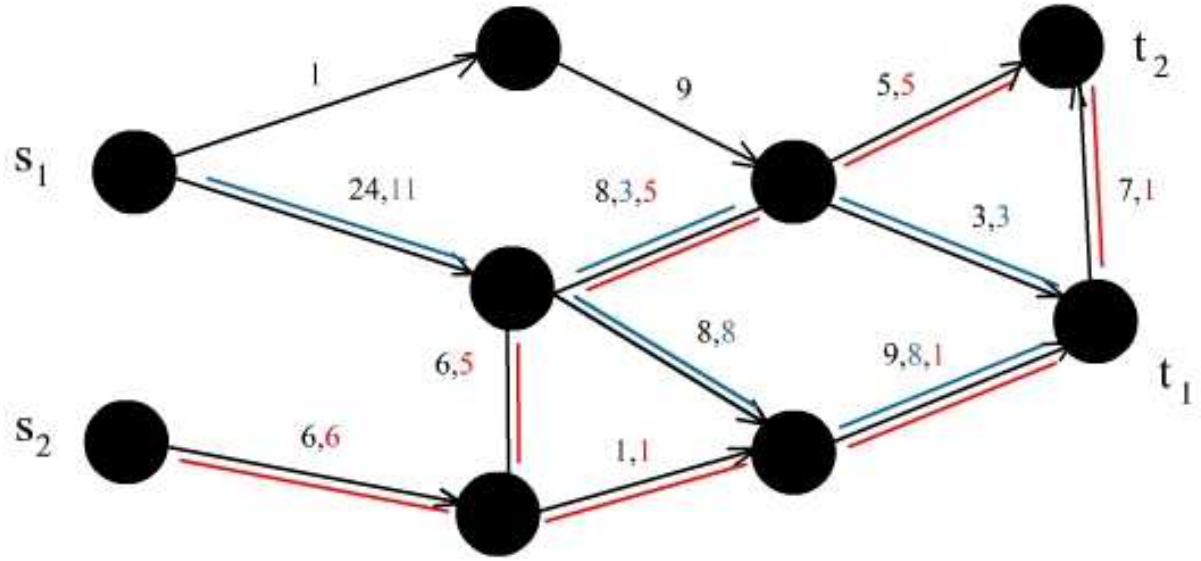
Adott (G, c, s, t) hálózat, keresünk olyan f maximális folyamot, amire $cost(f)$ minimális.

1.5. Többtermékes folyam

Egy termék egy (s, t, d) hármas, ahol s a termelő csúcs, t a fogyasztó csúcs, $0 < d \in \mathbb{R}$ pedig az igény mértéke.

A többtermékes folyam problémája a következő. Adott egy $G = (V, E)$ irányított gráf, és egy $c : E \to \mathbb{R}$ kapacitás függvény, amire $\forall (u, v) \in E$ élen $c(u, v) \geq 0$, és k darab termék. Vezessük el a termékek f_i folyamait úgy, hogy összegük nem haladja meg a kapacitásokat. Ez a következőt jelenti.

- $f_i(u, v) \geq 0$
- $\forall v \in V \setminus \{s_i, t_i\}$ csúcsra $\sum_{e=(u,v)\in E} f_i(e) - \sum_{e=(v,u)\in E} f_i(e) = 0$
- $\sum_{e=(u,v)\in E} f_i(e) - \sum_{e=(v,u)\in E} f_i(e)$ csak $v = s_i$-re negatív és $v = t_i$-re egyenlő d_i-vel
- $f(e) = \sum_{i=1}^{k} f_i(e) \leq c(e)$



A többtermékes folyamproblémával kapcsolatban többek között a következő kérdések merülnek fel.

- **Megengedett megoldás:** Lehetséges-e minden terméket elvezetni?
- Mi az a legnagyobb β, amivel minden termék kapacitásigényét megszorozva, még létezik megengedett megoldás? Azaz keressük β-t, ahol $\beta = \max\{\min \frac{c(e)}{f(e)}\}$.
- Mi az a legkisebb $\alpha \in \mathbb{R}$, amivel az élek kapacitásait megszorozva, már minden terméket el tudunk vezetni? Azaz keressük α-t, ahol $\alpha = \min\{\max_{e \in E} \frac{f(e)}{c(e)}\}$.
- Az általunk keresett f folyamok közül melyik használja a legkevesebb erőforrást? Azaz azt az f folyamot, amire $\sum_{e \in E} f(e)$ minimális.

Látható hogy a második és a harmadik kérdés ekvivalens, $\alpha = \frac{1}{\beta}$.

A többtermékes folyamok fontos szerepet kapnak a hálózattervezésben, legyen az telekommunikációs közlekedési vagy egyéb hálózat, logisztikai feladatoknál vagy például a kapcsolt optikai hálózati technikák(Optical Burst Switching, OBS) alkalmazása során felmerülő útvonal és hullámhossz hozzárendelési probléma (Route and Wavelength Assignment, RWA) megoldásában, vagy áramkör tervezési feladatok megoldásakor.

Bár az általános többtermékes folyamproblémának létezik polinomiális futási idejű egzakt megoldása, tekintettel a problémák méretére, a gyakorlatban mégis gyakran csak közelítő megoldást keresünk.

1.5.1. Egészértékű többtermékes folyam

Külön érdekes lehet az az eset, mikor csak egész folyamértékek megengedettek. Ez logisztikai feladatoknál annak felel meg, hogy szeretnénk az árukat teli teherautókkal szállítani.

Ellentétben az általános többtermékes folyamproblémával, az egészértékű többtermékes folyamprobléma NP-nehéz.

1.6. Többtermékes osztatlan folyam

Telekommunikációs hálózatokban a kommunikáció gyakran a következő modell szerint történik. Először felépítünk egy kommunikációs csatornát a hívó és a hívott között. Minden kommunikáció ezen a csatornán történik, végül pedig a csatorna lebontásra kerül. Látható, hogy az eddigi hálózati modellek nem felelnek meg teljesen ennek a modellnek. Például, ha az információ több útvonalon is közlekedhet, akkor a különböző késleltetések miatt összekeveredhet.

A többtermékes osztatlan folyam probléma a többtermékes folyam probléma speciális változata. Adott egy $G = (V, E)$ irányított gráf, $c : E \to \mathbb{R}$ kapacitások az éleken $\forall (u, v) \in E$ élen $c(u, v) \geq 0$ és k darab termék. Keresünk egy olyan f többtermékes folyamot, ahol minden termék f_i folyamát egy úton vezetjük el, azaz egy k termékes osztatlan folyam utak egy halmaza $P_1, P_2, \ldots, P_k$, ahol P_i az s_i termelő csúcsból a t_i fogyasztó csúcsba vezet. Az utak a következő f folyamot határozzák meg $f(e) = \\ = \sum_{e:i \in P_i} d_i$.

A többtermékes osztatlan folyam probléma már két termékre is NP-teljes.

Az osztatlan folyamokkal kapcsolatban több érdekes kérdés is felmerül.

- **Megengedett megoldás:** Lehetséges-e minden terméket osztatlanul elvezetni?
- Mi az a legkisebb $\alpha \in \mathbb{R}$ amivel a kapacitásokat megszorozva már minden terméket osztatlanul el tudunk vezetni?
- **Maximalizáció:** Keressük a termékek olyan T részhalmazát, melyet osztatlanul el lehet vezetni, és amire $\sum_{i \in T} d_i$ maximális.
- **Fordulók száma:** Hány forduló szükséges ahhoz, hogy minden terméket osztatlanul elvezessünk? Azaz osszuk a termékek halmazát diszjunkt részhalmazokra olyan módon, hogy az így kapott T_j halmazban szereplő igényeket osztatlanul el lehessen vezetni. Egy ilyen részhalmazt nevezünk egy fordulónak. Olyan felosztást keresünk, amire a részhalmazok száma minimális.

A többtermékes osztatlan folyamok problémája kerül elő az MPLS (Multiprotocol Label Switching) hálózatokban is. Az új protokollok használatával minden termelő-fogyasztó

párhoz lehet külön utat meghatározni. Így szükségessé válik új, többtermékes osztatlan folyamokon alapuló megoldások használata.

1.7. Approximáció

A többtermékes osztatlan folyamok problémája NP-nehéz. Ez már csak azért is így van, mert több ismert NP-nehéz problémát meg lehet fogalmazni többtermékes osztatlan folyam problémaként. Például: partícionálás, ládapakolás. Ezekkel később részletesen is foglalkozunk.

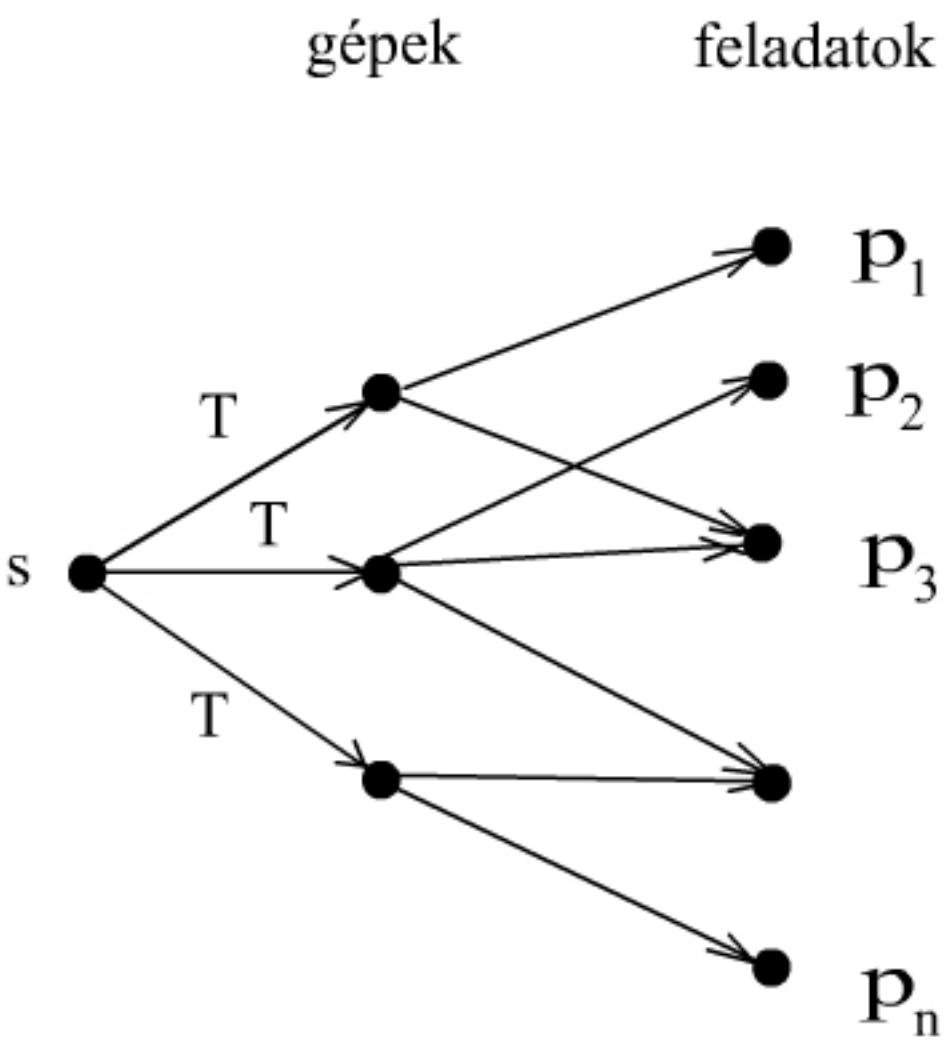
A valós problémák mérete ráadásul tipikusan igen nagy. Egy telekommunikációs hálózatban több száz csúcs (gép) és több ezer él található, a termékek (a továbbítani kívánt adatcsomagok) száma pedig akár a több tízezret is elérheti. Ezek együttesen szükségessé teszik hatékony approximációs algoritmusok illetve heurisztikák használatát.

1.8. Példák

1.8.1. Ütemezés

A következő ütemezési feladat, az osztatlan folyamok problémájának egy speciális esete. n feladatot kell ütemeznünk m párhuzamos gépen. Minden feladatot csak a gépek egy részhalmazán lehet végrehajtani, a j-edik feladatot a gépek $M(j)$ részhalmazán. a j-edik feladat végrehajtási ideje p_i bármely megengedett gépen. A feladat a végrehajtási idő minimalizálása.

A feladatot osztatlan folyam problémaként is meg lehet fogalmazni. Legyen G irányított gráf, és hozzunk létre minden gépnek és minden feladatnak egy egy csúcsot G-ben, valamint egy s termelőt. Kössük s csúcsot minden géphez T kapacitással, és kössük a gépeket azokhoz a feladatokhoz, amiket végre tudnak hajtani, végtelen kapacitással. Legyen minden csúcs, ami egy feladatot reprezentál, egy fogyasztó, a feladat végrehajtási idejével megegyező igénnyel.



Látható, hogy az osztatlan folyam akkor és csak akkor megengedett, ha létezik ütemezés legfeljebb T végrehajtási idővel. Ha létezik a problémának osztott megoldása, akkor létezik ütemezés, aminek a végrehajtási ideje $T + \max_j p_j$

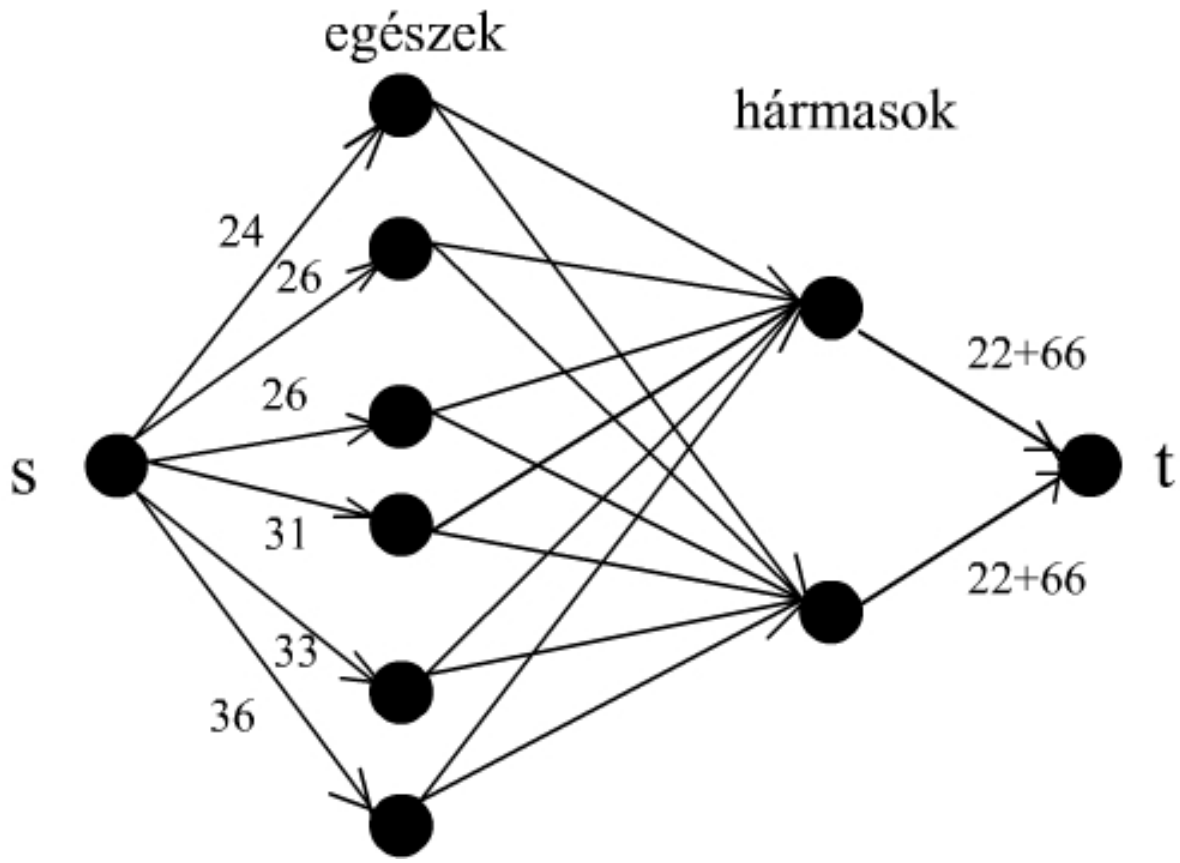
1.8.2. Partícionálás (3-partition problem)

A 3-partícionálás problémája a következő. Adott $n = 3m$ darab, nem feltétlen különböző, pozitív egész szám. Csoportosítsuk őket m darab számhármasba ($S_1, S_2, \ldots, S_m$), úgy, hogy a számhármasokban szereplő számok összege azonos.

Például: $\{2,4,4,9,11,14\} \rightarrow S_1 = (11,2,9), S_2 = (4,4,14)$

A feladatot osztatlan folyam problémaként is meg lehet fogalmazni. Vegyünk fel egy

csúcsot az n darab szám mindegyikének, majd egy csúcsot minden S_i számhármasnak. Ezután pedig egy s csúcsot a termelőnek. Kössük s-t egy egy éllel a számokat reprezentáló csúcsokhoz. Ezután kössünk össze a számokat reprezentáló csúcsokat, minden a számhármasokat reprezentáló csúccsal. Jelöljük az összes szám összegét N-el, az i-edik szám értékét pedig v_i-vel. Minden az i-edik számot reprezentáló csúcsba vezető él kapacitása egyezzen meg $v_i + N$-el. Végül pedig vegyünk fel egy t csúcsot a fogyasztónak. Kössünk minden S_i számhármasnak megfeleltetett csúcsot egy éllel t-hez, aminek a kapacitása pontosan $3N + N/m$ legyen. Legyen az n darab termék a következő $(s_i, t_i, d_i) = = (s, t, v_i + N)$.

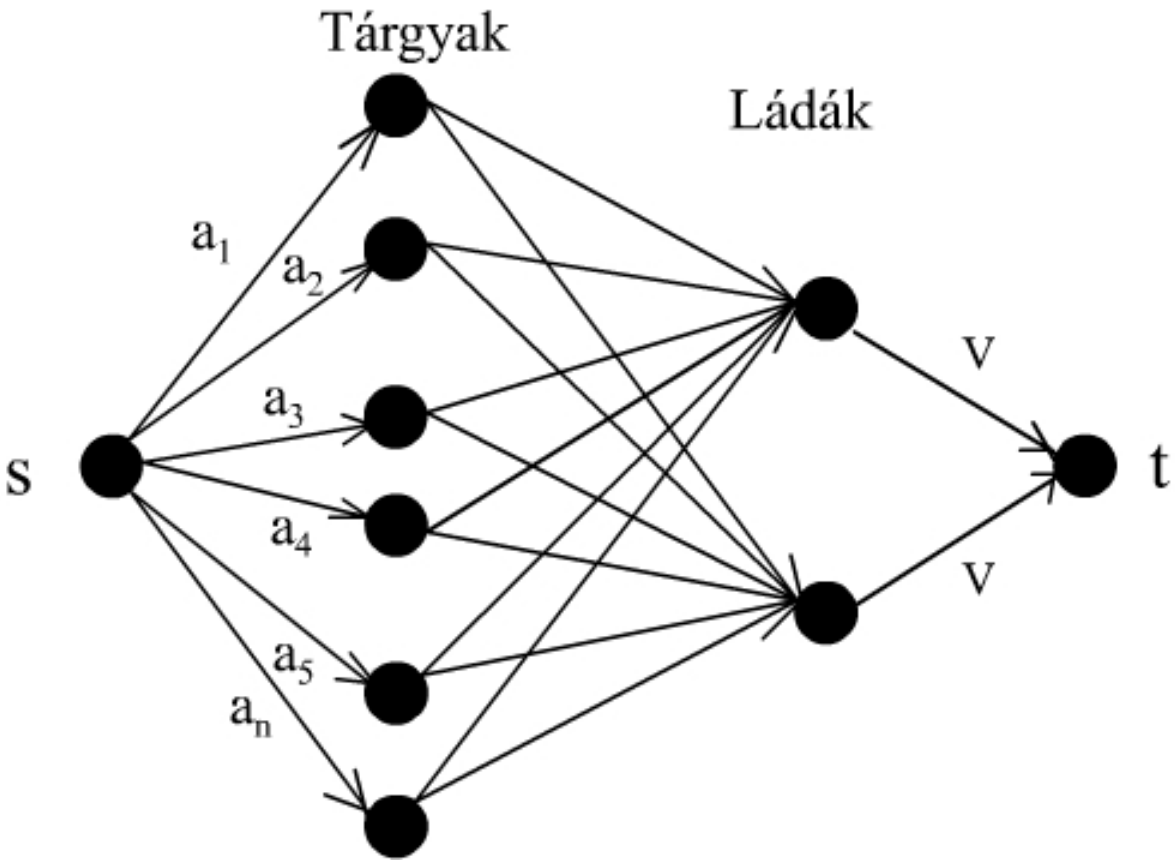


Látható, hogy az osztatlan folyam akkor és csak akkor megengedett, ha létezik $S_1, S_2, \ldots, S_m$, hogy a számhármasok számainak összege azonos. Azok a számok kerülnek az S_i számhármasba, amiket reprezentáló termékek az S_i számhármast reprezentáló csúcsot érintik.

1.8.3. Ládapakolás (Bin Packing)

Adott egy tároló V kapacitással, és $\{a_1, a_2, \ldots, a_n\}$ a tárolóban elhelyezni kívánt tárgyak méretének egy listája. Keressük azt a legkisebb B számot, valamint $\{1, \ldots, n\}$-nek azt az $S_1 \cup \ldots \cup S_B$ partícionálását, amire $\forall k = 1, \ldots, B$-re, $\sum_{i \in S_k} a_i \leq V$.

Először vegyünk fel egy s csúcsot a termelőknek és egy t csúcsot a fogyasztóknak, majd vegyünk fel egy csúcsot minden tárgynak és minden tárolónak. Összesen $n + B$ darabot. Kössük egy éllel s-t minden tárgyat reprezentáló csúcshoz. A tárgyak csúcsait kössük a tárolókhoz, a méretüknek megfelelő kapacitással. Végül kössük a tárolókat t-hez V kapacitással.



Az osztatlan folyam akkor és csak akkor megengedett, ha létezik megfelelő partícionálás. A minimális B-t úgy is kereshetjük, hogy a tárolóknak felveszünk n csúcsot és a tárolókat t-vel összekötő élek kapacitását úgy határozzuk meg, hogy az szigorúan monoton növekedjen. Például az i-edik tárolót t-hez kötő él költsége legyen i. Ezután pedig minimális

osztatlan folyamot keresünk.

A problémának több változata is létezik, például kétdimenziós pakolás, súly szerinti pakolás, költség szerinti pakolás, és mások.

2. Áttekintés

Ez a fejezet a problémakör irodalmának, az előzményeknek rövid áttekintését tartalmazza. Az összefoglaló Martin Skutella [14] írása és az Új algoritmusok könyv alapján készült.

A hálózati folyamokkal kapcsolatos problémákat először Ford és Fulkerson [10] vizsgálták. Olyan kérdésekkel foglalkoztak, mint a maximális folyam probléma és a maximális párosítás problémája páros gráfokban. Az általuk javasolt algoritmus első implementációi szélességi keresésen alapultak. Ezekről az implementációkról Edmonds és Karp [2] valamint Dinic [1] egymástól függetlenül bebizonyították, hogy futási idejük polinomiális.

Az előfolyamok módszerét Karzanov [7] dolgozta ki. Az általános előfolyam-algoritmust Goldberg és Tarjan dolgozta ki, Goldberg [3], Goldberg és Tarjan [5]. Ezután számos kutató dolgozott ki előfolyam-algoritmusokat.

A blokkoló folyamok fogalmát Dinic [1] vezette be, és többek között Goldberg és Rao [4] dolgozott ki a segítségével hálózati folyam algoritmust.

A többtermékes osztatlan folyamok problémájával Kleinberg [8] foglalkozott először, aki az egy termelős esetet vizsgálta, azaz amikor minden termék termelője azonos csúcsban található. A probléma több NP-teljes probléma általánosítása, a kombinatorikus optimalizálás különböző területeiről. Ezután Kolliopoulos és Stein [9], valamint Dinitz, Garg és Goemans [15] dolgoztak ki olyan algoritmusokat, melyek Kleinberg [8] eredeti eredményeinél jobb approximációt biztosítottak a kapacitások túllépésének mértékének a minimalizálására. Ezek a módszerek azonban nem tudják kezelni a költségeket. Martin Skutella [14] dolgozott ki elsőnek olyan algoritmust, ami a folyam költségét nem növeli. Dinitz, Garg és Goemans [15], valamint Martin Skutella [14] algoritmusa egy osztott folyamot vesz kiindulási alapul, ezt alakítja osztatlan folyammá. Az algoritmusok részletes ismertetésével a Nagyprogramom foglalkozik, ami a mellékletben megtalálható.

Az általános többtermékes osztatlan folyam problémájára több heurisztikus módszert is kidolgoztak, például Pióro "Szimulált Allokációs" módszere, Pióro és Gajowniczek [12]. Más módszerek Dijkstra legrövidebb út kereső algoritmusát használják, megfelelően megállapított él súlyokkal. Ilyen például, többek között Plotkin [13] vagy Kodialam és Lakshman [11] módszere.

Józsa Balázs Gábor, Király Zoltán, Magyar Gábor és Szentesi Áron [6] valós méretű problémákon is hatékonyan működő heurisztikus módszert adtak a többtermékes osztatlan folyamok problémájára, ami képes a keresett úthalmazzal kapcsolatosan bizonyos megkötések kezelésére is, mint például a keresett utak hosszának korlátozása.

3. Tesztadatok generálása

Egy teszteset a következőkből áll. Egy $G = (V, E)$ irányított gráf, $c : E \to \mathbb{R}$ kapacitások az éleken, és (s_i, t_i, d_i) termékek egy halmaza. Ezeket a gen_top.h fájlban található függvények segítségével állítom elő.

Egy tesztesetet két fájlban tárolok. Az első fájl tartalmazza a gráf csúcsait, éleit és a hozzájuk tartozó kapacitásokat. A második fájlban a termékek (s_i, t_i, d_i) hármasai találhatók. A fájlokat a következőképpen neveztem el. Ha a gráfot tartalmazó fájl neve teszt_1, akkor a termékeket tartalmazó fájlé teszt_d1.

3.1. Topológia

A $G = (V, E)$ gráfot a következő függvénnyel hozom létre.

```
void GenTop(Graph &g, int nodes, int edges)
```

Ez a függvény egy üres gráfhoz nekünk megfelelő módon ad hozzá csúcsokat és éleket. A csúcsok és az élek számát határozhatjuk meg.

Mivel a termékeket csak akkor lehet elvezetni, ha van út a termelőből a fogyasztóba, ezért a gráfnak összefüggőnek kell lennie. Ezt úgy érem el, hogy minden csúcsot felfűzök egy körre. Az $e = (v_i, v_{i+1}) \in E$ ahol v_i az i-edik csúcs, valamint az utolsó csúcsból is vezet egy él az első csúcsba.

A maradék éleket véletlenszerűen hozza létre, a lemon rnd függvénye segítségével.

3.2. Termékek és kapacitások

A termékeket a következő függvény segítségével állítom elő. (demv a termékek egy vektora)

```
demv GenDem(Graph &g,int num,int low,int high)
```

Megadhatjuk a termékek számát, és a kapacitás igények alsó illetve felső határát. A függvény minden termékhez véletlenszerűen kiválaszt egy termelő és egy fogyasztó csúcsot, majd meghatároz egy kapacitás igényt az alsó és felső határ között, a lemon rnd függvénye segítségével.

Az egytermelős esethez a GenDemCS függvényt használhatjuk.

```
demv GenDemCS(Graph &g,int num,int low,int high,int seed=0)
```

Az előzőhöz hasonlóan működik, de itt minden terméknek ugyanazt a csúcsot választja termelőnek.

Az élekhez tartozó kapacitásokat a következő függvény állítja elő.

```
EdgeMap GenCap(Graph &g, demv demands)
```

Szeretnénk ha a feladatnak lenne megoldása, azaz ellenőrizhető legyen az algoritmusok hatékonysága, de nem szeretnénk hogy túl egyszerű legyen a feladat, például ne a legrövidebb úton kelljen az igényeket elvezetni. Ezért a kapacitásokat a következő módon generálja a függvény. Minden (s_i, t_i, d_i) termékhez véletlenszerűen meghatároz egy P_i utat a termelőből a fogyasztóba, majd ennek az útnak a mentén a kapacitásokat megemeli d_i-vel.

Bizonyos esetekben ezeket a kapacitásértékeket később módosítottam, vagy az élekhez költségeket rendeltem. Ezt mindig az adott feladatnál fogom közölni.

4. Az egytermelős többtermékes osztatlan folyam

Dinitz, Garg és Goemans [15], és Martin Skutella [14] algoritmusának leírása, valamint egy egy konkrét implementáció ismertetése a Nagyprogramom része volt, ami megtalálható a mellékletben. Néhány kérdés azonban megválaszolatlan maradt, amik közül talán a következő kettő a legérdekesebb.

4.1. Dinamikusan változó költségfüggvény

A fent említett algoritmusok egy osztott folyamból indulnak ki, majd azt osztatlan folyammá alakítják. Amint az a Nagyprogramomban is olvasható, ha megfelelően megválasztott osztott folyamból indulunk ki, például a kiindulási folyamot minimális költségű folyamként keressük a kapacitással fordítottan arányos költségek mellett, akkor sokkal jobb megoldást kapunk, ráadásul a futási időt is kedvezően befolyásolhatja.

Felmerül a kérdés, hogy ismerve, hogy az osztatlan folyam mely éleken lépi túl a kapacitást és mennyivel, kereshetünk-e olyan kiindulási folyamot, hogy az abból előállított osztatlan folyam jobb megoldást szolgáltasson. Erre a következő iteratív módszert javaslom.

1. Keressük a kiindulási folyamot minimális költségű folyamként a kapacitással fordítottan arányos költségek mellett. Ezt alakítsuk osztatlan folyammá.
2. Vizsgáljuk meg, mely éleken lépi túl a folyamérték a kapacitást, majd a túllépés mértékének függvényében növeljük meg az adott élen a költséget. Amennyiben a túllépés mértéke elég kicsi, vagy elértünk egy előre meghatározott számú iterációt leállunk.
3. Keressünk új kiindulási folyamot az új költségekkel, majd alakítsuk osztatlan folyammá. Folytassuk a 2. lépéssel.

Nem egyértelmű, hogy mennyivel növeljük a költségeket, mint ahogy az sem, hogy egy iterációban csak egy vagy több él költségén is változtassunk-e.

4.1.1. Tesztadatok

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a normal_x és normal_dx fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

Élek száma: 2000

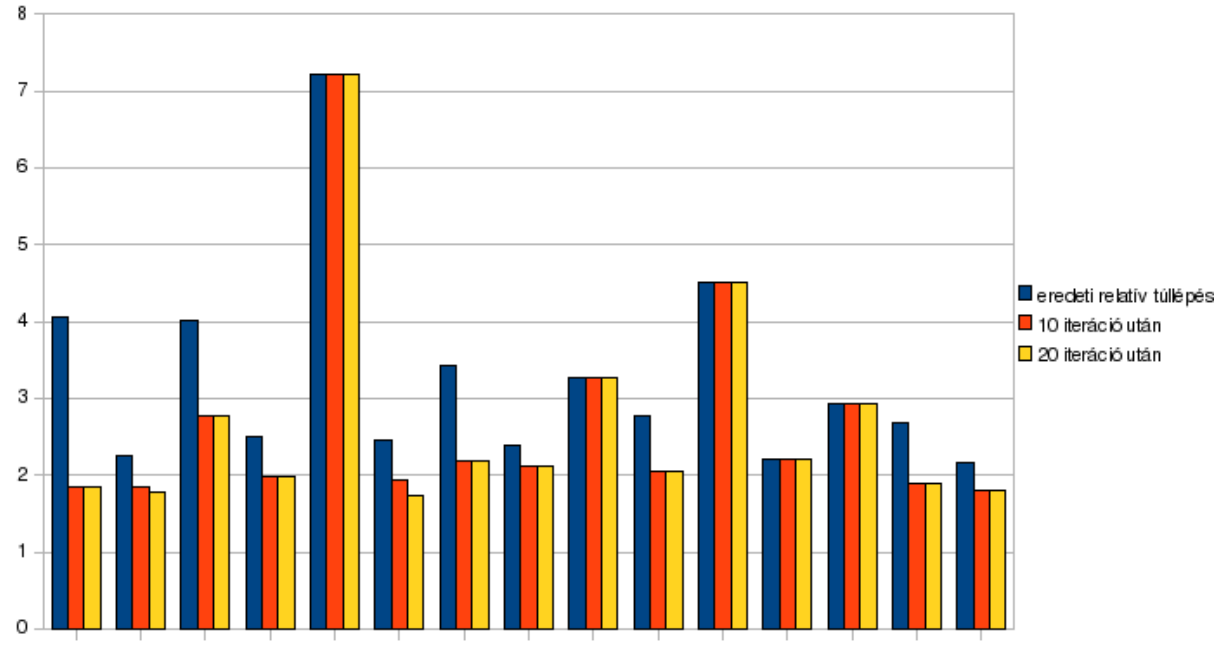
Termékek száma: 1000

Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

Az éleken található költségeket kezdetben a következőképpen határoztam meg. Legyen a költség e élen $cost(e) = \frac{10C_{max}}{c(e)}$, ahol C_{max} a kapacitások maximuma, $\forall e \in E$ élre, ha $c(e) > 0$, különben legyen C_{max}.

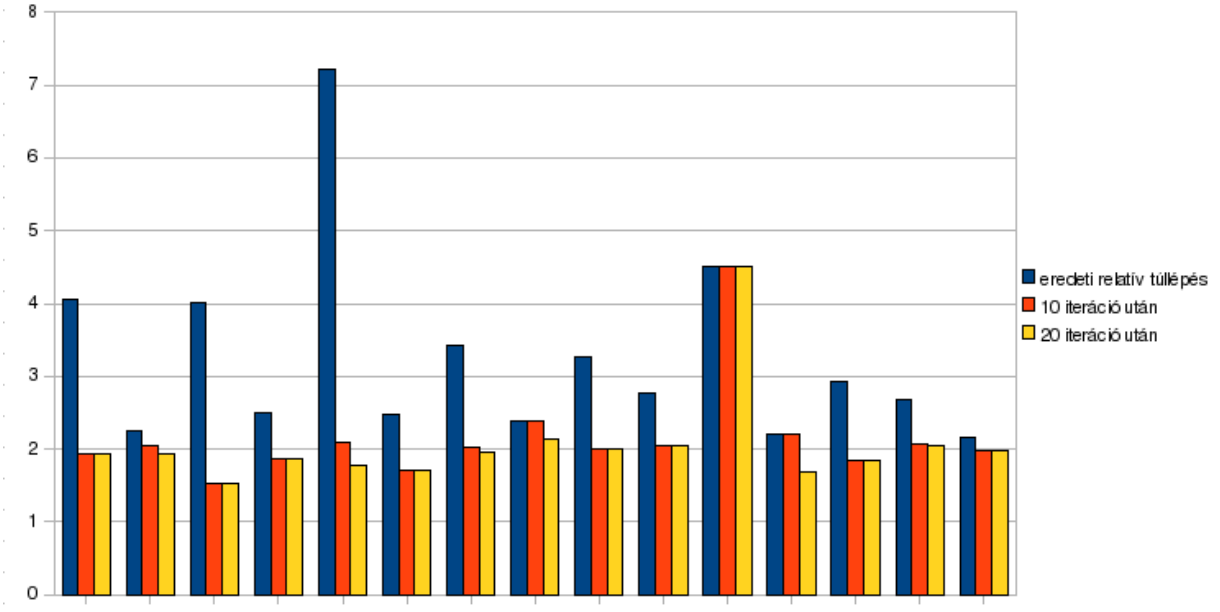
4.1.2. Tesztek eredménye

A következő ábrák Dinitz, Garg és Goemans [15] algoritmusának futási eredményeit mutatják a tesztadatokra, a fent vázolt iterációs módszerrel. A cél $\max \frac{f(e)}{c(e)}$ relatív túllépés minimalizálása. Elsőnek azt az esetet vizsgálom, mikor csak azon az élen változtatjuk a költségfüggvényt, amelyikre $\frac{f(e)}{c(e)}$ maximális. Az adott élen a költségfüggvény új értékét a $cost_{uj}(e) := cost_{regi}(e)\frac{f(e)}{c(e)}$ képlettel határozzuk meg.



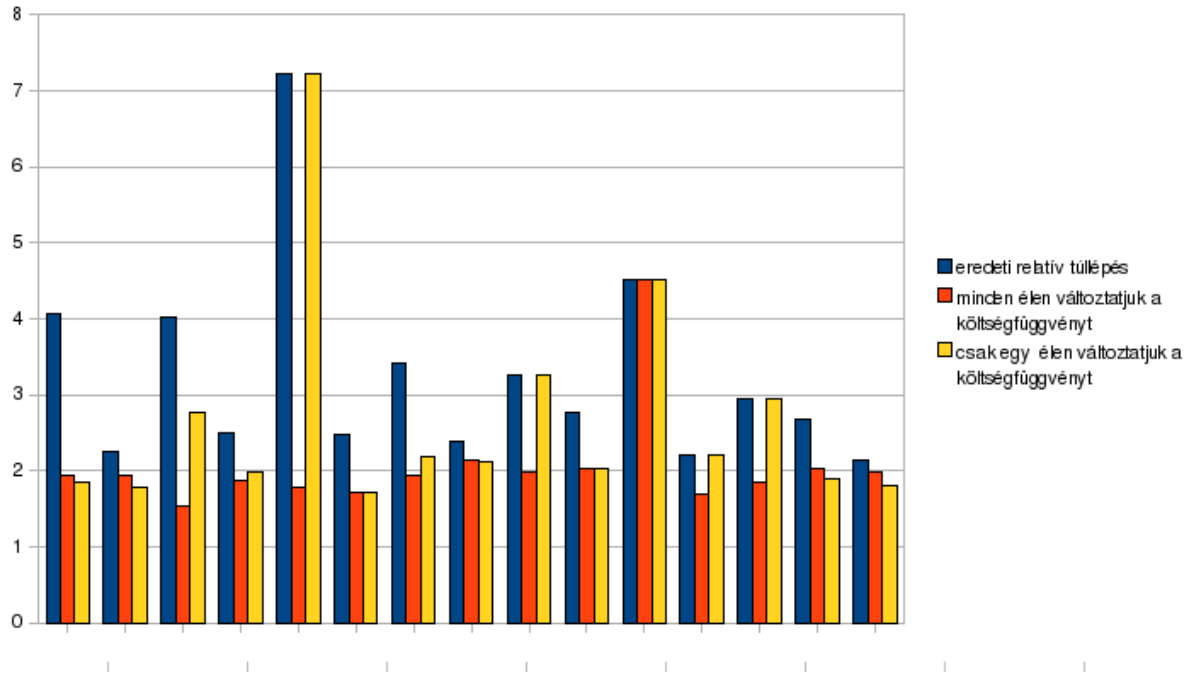
A módszer mindössze az esetek kétharmad részében hatásos. Az átlagos javulás húsz iteráció után 18%. Megfigyelhető továbbá, hogy nincs nagy különbség az eredmények között 10 iteráció valamint 20 iteráció után. Az eredmény csak két esetben javul.

Vizsgáljuk most azt az esetet, mikor minden olyan élen változtatjuk a költségfüggvényt, amelyik élre $f(e) > c(e)$. Ezekre az élekre legyen $cost_{uj}(e) := cost_{regi}(e)\frac{f(e)}{c(e)}$.



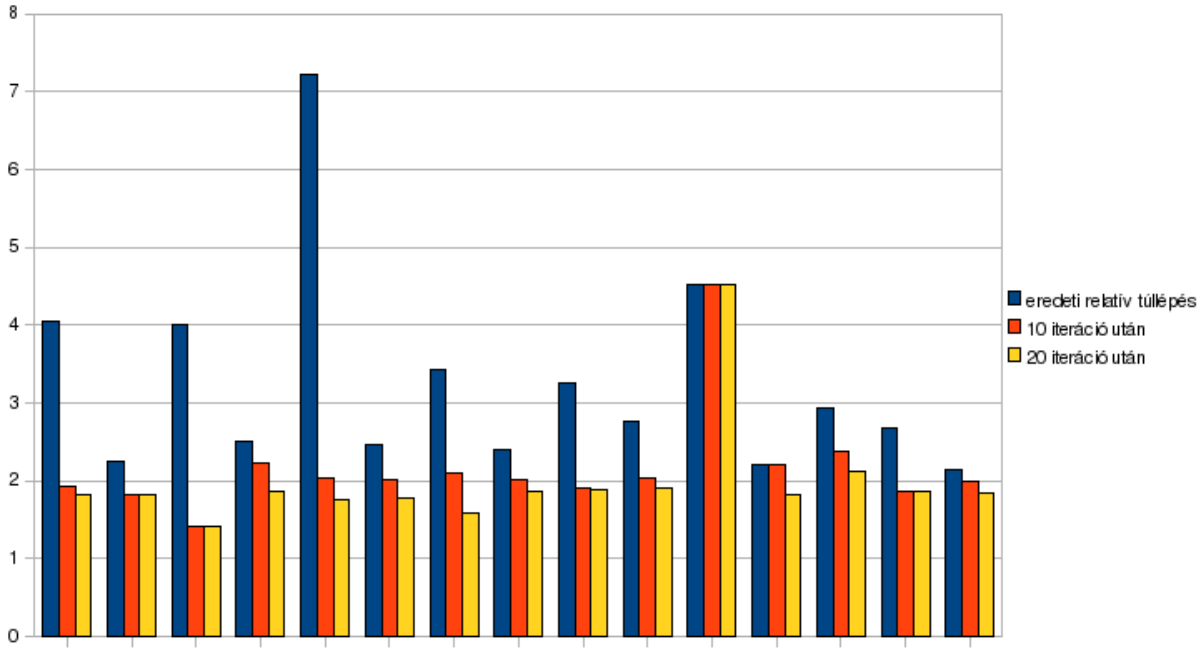
Ebben az esetben a módszer sokkal robusztusabbnak bizonyul, tíz iteráció alatt három, míg húsz iteráció alatt, csak egyetlen esetben nem tud javítani az eredeti megoldáson. Tizenöt esetből kilencben már tíz iteráció alatt megtalálja a legjobb osztatlan folyamot, pontosabban húsz iteráció alatt sem talál jobbat. A javulás tíz iteráció alatt átlagosan 28%, míg húsz iteráció alatt átlagosan 31%.

A jobb összehasonlítás érdekében a következő ábrán az előző két eredmény összevetése következik, húsz iteráció esetén.



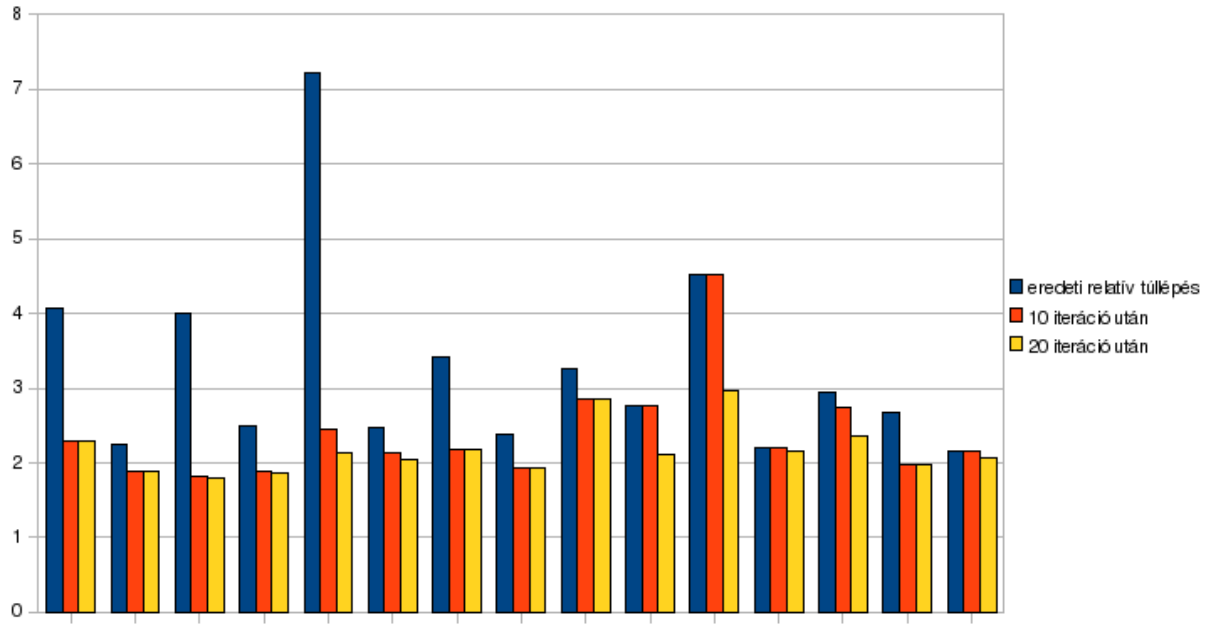
Látható, hogy bár több esetben is jobb eredményt érünk el, ha csak egy élen változtatjuk a költségfüggvényt, ezekben az esetekben a különbség nem olyan nagy. Ellenben azokban az esetekben mikor így nem tudunk jobb eredményt elérni, gyakran ha minden élen megváltoztatjuk a költségfüggvényt, akkor jelentősen csökkenthetjük a $\max \frac{f(e)}{c(e)}$ értéket.

Ezután vizsgáljuk azt az esetet, mikor minden olyan élen változtatjuk a költségfüggvényt, amelyik élre $f(e) > c(e)$. Ezekre az élekre legyen $cost_{uj}(e) := cost_{regi}(e)\sqrt{\frac{f(e)}{c(e)}}$.



Tíz iterációra az átlagos javulás 28%, de húsz iterációra minden eddiginél jobb, 34%. Tizenötből négy esetben adja ugyan azt az eredmény tíz és húsz iterációra. Egyetlen esetben nem tud javítani a kezdeti relatív túllépésen.

A következő esetben szintén minden olyan élen megváltoztatjuk a költségfüggvényt, ahol az osztatlan folyam megsérti a kapacitásokat. Az új költség legyen $cost_{uj}(e) := cost_{regi}(e)\left(\frac{f(e)}{c(e)}\right)^2$.



A javulás mértéke nem kiemelkedő. Az átlagos javulás tíz iterációra 21%, míg húsz iterációra 27%. Egyedül azt érdemes megemlíteni, hogy húsz iterációra már mind a tizenöt tesztesetre talál a kezdetinél jobb megoldást, beleértve a tizenegyedik tesztesetet, ami minden előző függvényen kifogott.

Összességében megállapítható, hogy a legjobb eredményt akkor értem el, ha a költségfüggvényt minden olyan élen növeltem, ahol a folyam megsértette a kapacitásokat. A legnagyobb javulást, tíz iteráció alatt 28, húsz iteráció alatt 34 százalékot, a következő függvény használatával értem el $cost_{uj}(e) := cost_{regi}(e)\sqrt{\frac{f(e)}{c(e)}}$ A tesztesetek majdnem harmadánál nem tudtam a tíz iteráció alatt talált megoldáson tovább javítani. Megjegyzem, hogy az eredmények kevés számú, viszonylag speciális tesztesetből származnak, ezért a következtetések nem tekinthetőek általános érvényűnek, viszont jól demonstrálják a megfelelő függvény megválasztásának szükségességét, és magának az iterációs módszernek a hatékonyságát.

4.2. Skutella algoritmusának összehasonlítása a DGG Algoritmussal speciális esetben

A probléma méretének, illetve a termelők számának növekedésével, Skutella [14] algoritmusának futási ideje gyorsan nő, míg Dinitz, Garg és Goemans [15] algoritmusára kevésbé van hatással. A részleteket a nagyprogramom tartalmazza, ami megtalálható a mellékletben.

Felmerül azonban a kérdés, hogy nem e inkább a különböző kapacitás igénnyel rendelkező termékek száma van nagyobb befolyással Skutella [14] algoritmusának futási ideje, mint a probléma mérete.

4.2.1. Tesztadatok

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a time_tests könyvtárban található test_x_y és test_x_dy fájlokban találhatók, ahol x a probléma méretére utal, míg y a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 250-től 2000-ig

Élek száma: 750-től 6000-ig

Termékek száma: 500-tól 4000-ig

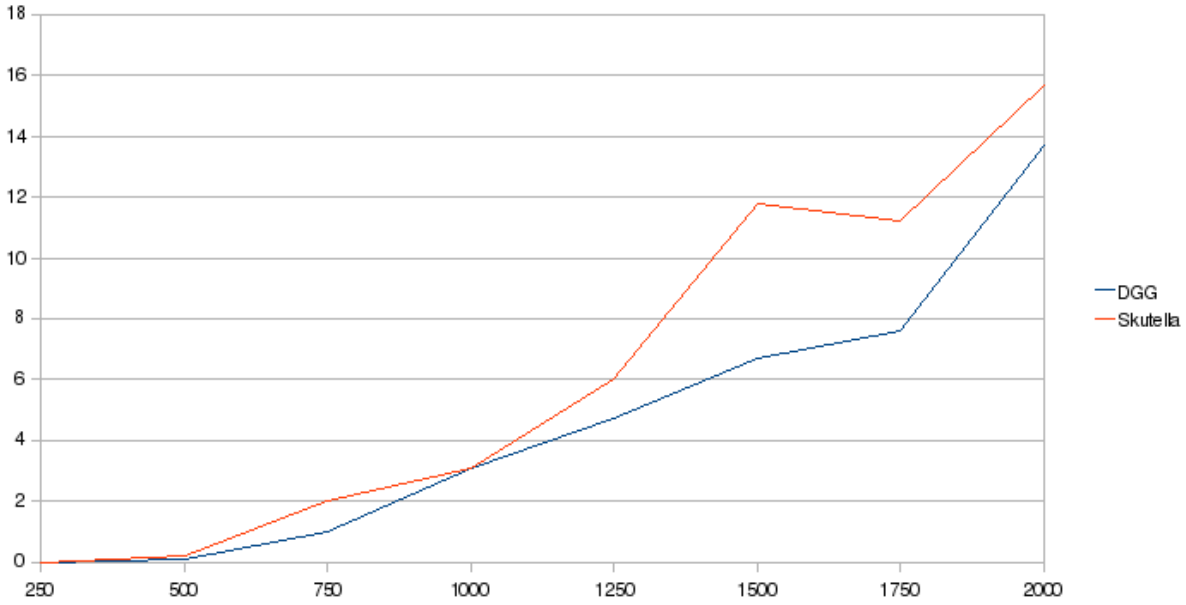
Termékek kapacitásigénye: 500 és 510 között véletlenszerűen, azaz csak 10 fajta kapacitásigény létezik

A generált gráfoknál a csúcsok és élek számának aránya, valamint a csúcsok és termékek számának aránya nem változik.

Az éleken található költségeket kezdetben a következőképpen határoztam meg. Legyen a költség e élen $cost(e) = \frac{10C_{max}}{c(e)}$, ahol C_{max} a kapacitások maximuma, $\forall e \in E$ élre, ha $c(e) > 0$, különben legyen C_{max}.

4.2.2. Tesztek eredménye

A következő grafikon a lemon realTime függvényével mért futási időket mutatja növekvő csúcsszám mellet a két algoritmusra. Minden probléma méretre tíz mérést végeztem, majd ezeknek az átlagát vettem.



A feltételezésünk helyesnek bizonyult abban az értelemben, hogy a két algoritmus futási ideje között sokkal kisebb a különbség a vizsgált feladatok esetében. Ennek ellenére Dinitz, Garg és Goemans [15] algoritmusa továbbra is jobban teljesít.

5. GPO Algoritmus

Józsa Balázs Gábor, Király Zoltán Magyar Gábor és Szentesi Áron [6] komplex heurisztikus módszert adtak a többtermékes osztatlan folyamok problémájára. Egészen pontosan azt a feladatot oldja meg, mikor a termékek olyan T részhalmazát keressük, melyet osztatlanul el lehet vezetni, és amire $\sum_{i \in T} d_i$ maximális. A módszer Dijkstra legrövidebb út algoritmusát használja dinamikusan változó súlyfüggvénnyel. Az algoritmus lényegében három dolgot tart nyilván. A feldolgozatlan termékek D halmazát, a meghatározott utak P halmazát, valamint az aktuális folyamértékeket $\forall e \in E$ élen.

Az algoritmus allokációk és deallokációk sorozatát hajtja végre. Az allokáció során az aktuális (s_i, t_i, d_i) termékhez meghatároz egy P_i utat és elhelyezi P-ben, a terméket törli a feldolgozatlan termékek közül, majd a folyamértéket megnöveli a P_i út mentén d_i-vel. Deallokációnak, vagy a P_i út felszabadításának pedig azt nevezzük, mikor egy P_i úthoz tartozó terméket visszahelyezünk a feldolgozatlan termékek halmazába, csökkentjük a folyamértéket az út mentén, majd P_i utat töröljük P-ből.

Az algoritmus képes kezelni különböző technikai megszorításokat, például korlátos úthossz és előírt útvonalak, kezelni, és alkalmas különböző célfüggvények szerinti optimalizációra. Ezen kívül meg fogom mutatni, hogy kisebb változtatásokkal képes a termékek prioritásának hatékony kezelésére.

5.1. Az algoritmus váza

Inicializáció:

Kezdetben a folyam értéke legyen $f(e) = 0 \ \ \forall e \in E$ élen, az iterációk számlálója pedig szintén $r = 0$. A termékek előfeldolgozása, a termékek rendezése nehézségük és a szerint, hogy létezik e több legrövidebb út.

Az r-edik iteráció:

1. A feldolgozatlan termékek rendezése a nehézségük szerint
2. **Allokáció:** Amíg vannak feldolgozatlan termékek ($D \neq \emptyset$) iteráljuk a következő lépéseket. Ha $D = \emptyset$ akkor a feladatnak megtaláltuk egy megoldását, következhet

az optimalizációs fázis.

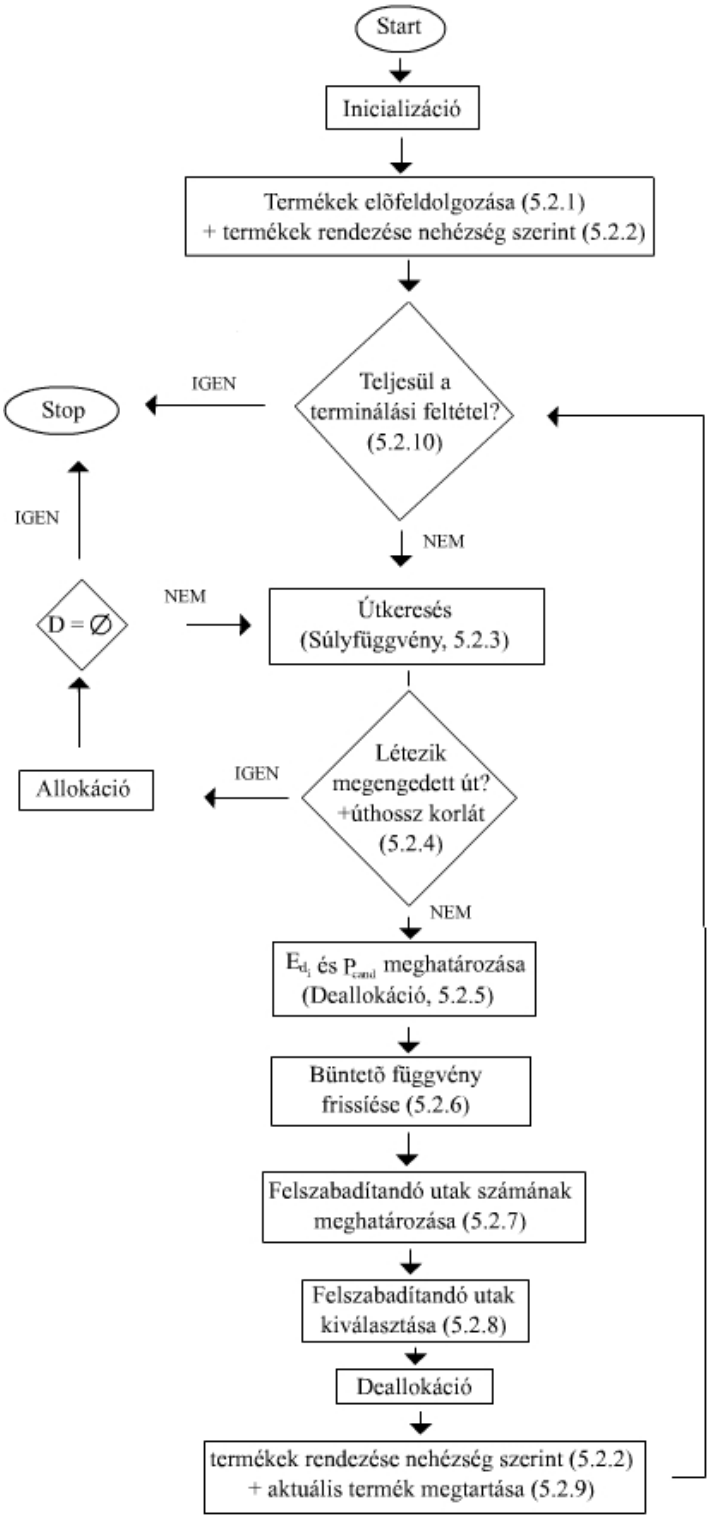
Út keresés: Vegyük az első feldolgozatlan (s_i, t_i, d_i) terméket a D rendezett listából és számítsuk ki a hozzá tartozó P_i utat Dijkstra algoritmusa segítségével, egy jól megválasztott $w(e)$ súlyfüggvénnyel.

Út hozzáadása: Ha a megoldás megengedett marad a P_i út hozzáadása után ($f(e)+ + d_i \leq c(e), \forall e \in P_i$), vegyük ki az (s_i, t_i, d_i) terméket a D listából és vegyük a P_i utat az utak P listájához. Növeljük meg az f folyam értékét az $e \in P_i$ éleken, azaz $f(e)_{uj} = f(e)_{regi} + d_i \ \ \forall e \in P_i$. Amennyiben a megoldás nem maradna megengedett a P_i út hozzáadása után, abbahagyjuk az allokációs fázist, és a 3. lépéssel folytatjuk az algoritmust.

3. **Deallokáció:** Amennyiben az f folyam nagysága egy előre meghatározott számú iteráción keresztül nem javult, az algoritmus leáll és az eddigi legjobb P útlistát adja vissza mint részleges eredményt. Ellenkező esetben azonosítja a szűk keresztmetszetnek számító éleket az utolsó el nem vezetett termék alapján, és egy előre meghatározott számú utat felszabadít a P úthalmazból, ennek megfelelően módosítja az f folyamot a felszabadított utakhoz tartozó éleken, és visszateszi a felszabadított utakat a fel nem dolgozott termékek közé. r értékét megnöveli egyel, és az 1. lépéstől folytatja.

Optimalizációs fázis: Az úthalmaz finomítása már kiszámolt utak újraszámolásával. Az újraszámolandó utakat az optimalizációs cél alapján választjuk ki. A megengedett megoldás keresésekor alkalmazott allokációs-deallokációs ciklushoz hasonlóan járunk el.

A következő ábrán egy valamivel részletesebb folyamatábra található az egyes kapcsolódó alfejezetek megjelölésével.


Start
Inicializáció
Termékek elõfeldolgozása (5.2.1)
+ termékek rendezése nehézség szerint (5.2.2)
Teljesül a terminálási feltétel? (5.2.10)
IGEN
Stop
NEM
Útkeresés (Súlyfüggvény, 5.2.3)
Létezik megengedett út? +úthossz korlát (5.2.4)
IGEN
NEM
D = ∅
IGEN
NEM
Allokáció
E_{d_i} és P_{cand} meghatározása (Deallokáció, 5.2.5)
Büntetõ függvény frissíése (5.2.6)
Felszabadítandó utak számának meghatározása (5.2.7)
Felszabadítandó utak kiválasztása (5.2.8)
Deallokáció
termékek rendezése nehézség szerint (5.2.2)
+ aktuális termék megtartása (5.2.9)

5.2. Az algoritmus részei

5.2.1. Termékek előfeldolgozása

Egyes speciális termékeket még az első iteráció előtt kiválogatunk, és elvezetünk, rögtön az inicializáció befejezése után. Azokról az (s_i, t_i, d_i) termékekről van szó, amiknél pontosan egy legrövidebb út vezet s_i csúcsból t_i csúcsba, az élek száma szerint.

Annak meghatározásához, hogy a legrövidebb út egyedi-e két legrövidebb út keresést kell Dijkstra algoritmusával végrehajtani. Első alkalommal állítsuk az élsúlyokat 1-re, így kapunk egy legrövidebb utat. Ezután a talált úton szereplő élek értékét egy elhanyagolható ϵ értékkel megnöveljük, és végrehajtunk egy újabb legrövidebb út keresést. A legrövidebb út akkor és csak akkor egyedi, ha a másodjára talált legrövidebb út az elsőre találttal megegyezik.

Ezzel a módszerrel időt spórolunk, és elkerüljük, hogy ezeket a termékeket később kivételesen hosszú utakon vezessük el, mikor a sor egyébként rájuk kerülne. Megjegyezzük, hogy a deallokációs fázisban, az ezekhez a termékekhez tartozó az utakat is fel lehet szabadítani, és a továbbiakban szűrés nem történik.

5.2.2. Termékek nehézségének mérése

Egy (s_i, t_i, d_i) termék nehézségét a következőképpen határozzuk meg. $h_i = \frac{d_i}{SPL_{s_i,t_i}}$, ahol $SPL_{s,t}$ az s csúcsból t csúcsba vezető legkevesebb élszámú út hossza. A P út éleinek számát $|P|$-vel jelöljük. A nehézségeket az előfeldolgozási fázisban rendeljük a termékekhez.

A fenti módszer nagyobb nehézséget rendel azokhoz a termékekhez, amiknek több kapacitásra van szüksége, azaz a d_i értékük nagy. Így ezeket a termékeket nagyobb eséllyel vezetjük el a legkevesebb élszámú útjaik mentén, mivel ennek a valószínűsége a termékek rendezett listájában előrébb szereplő termékekre nagyobb.

Másrészt a nehézség fordítottan arányos a termékhez tartozó legkevesebb élszámú út hosszával, hogy azokat a termékeket, ahol s_i és t_i távolsága relatív nagy, a későbbi fázisok során vezessük el. Ezen termékekhez ugyanis tipikusan több minimális élszámú út

szokott tartozni, mint ahol s_i és t_i távolsága kicsi.

5.2.3. Súlyfüggvény

A heurisztika Dijkstra legrövidebb út algoritmusát használja a termékekhez tartozó utak kiszámításához. Az $e \in E$ élhez tartozó $w(e)$ súly nem statikus, hanem az élen aktuálisan folyó $f(e)$ folyam érték, az él $c(e)$ kapacitása és az aktuális termék d_i kapacitás igényének függvényében dinamikusan változik.

$$w(e) = \lambda_e \Big(1 + \frac{1}{|V|} \frac{z}{z + \frac{c(e)-f(e)-d_i}{c(e)}}\Big)$$

Ahol (s_i, t_i, d_i) az aktuális termék, amihez az utat keressük, $\lambda_e \geq 1$ pedig egy szorzó ami minden élre értelmezve van. $|V|$ a gráf csúcsainak számát jelöli, ami felső korlát a minimális élszámú utakra bármely s és t csúcs között. z egy érzékenységi paraméter, ami a leginkább megterhelt él relatív szabad kapacitását mutatja, és a következőképpen számoljuk.

$$z = \max\Big(\min_{e \in E}(1 - \frac{f(e)}{c(e)}), z_{min}\Big)$$

Ahol $z_{min} = 0.005$.

Amennyiben $c(e) < f(e) + d_i$ a terméket nem lehet ezen az élen elvezetni (nem ér el), és az él súlyát végtelenre állítjuk $w(e) = \infty$, így a legrövidebb út algoritmus ezeket az éleket automatikusan el fogja kerülni, ha létezik megengedett út s_i és t_i csúcsok között, azaz olyan P_i út, hogy $c(e) \geq f(e) + d_i \;\; \forall e \in P_i$.

A fenti függvény használata mögött a következő megfontolások húzódnak meg. Azokat az éleket, ahol $c(e) - f(e) \geq d_i$ a függvény a relatív terhelés szerint súlyozza. $w(e)$ értékének kiszámításánál, zárójelben szereplő összeg egy egész, egységnyi súlyból és egy törtrészből áll. A törtrész nagyjából fordítottan arányos az élen aktuálisan szabad kapacitással, továbbá úgy van skálázva, hogy a törtrészek összege egyetlen úton sem lépheti túl az 1-et, bármely két csúcs között. Így minden $x = |P_i|$ hosszú P_i úton az úthoz tartozó $\sum_{e \in P_i} w(e)$ súly az $(x, x+1)$ intervallumba esik a legrövidebb út keresése során, ha $\lambda_e = 1, \forall e \in P_i$. Más szóval a fenti súlyfüggvény

- azt az utat adja, amire az élek száma minimális

- ezek közül az utak közül azt választja, ahol a törtrészek összege minimális a használt éleken, azaz a legkevesebb nagy leterheltségű élet tartalmazza

Így több optimalizációs célt egy időben vesz figyelembe. A fő cél minél kevesebb él használata, ami biztosítja, hogy ne pazaroljuk az erőforrásokat.

5.2.4. Dinamikus út hossz korlát

A Dijkstra legrövidebb út algoritmusa által talált utakat nem fogadjuk el rögtön. Egy az $SPL_{s,t}$-hez relatív δ korlátot alkalmazunk. Az s és t közötti P utat akkor és csak akkor fogadjuk el, ha $|P| \leq SPL_{s,t} + \delta$, különben a deallokációs fázissal folytatjuk. A δ korlát dinamikusan változik, és attól függ, hány iteráción keresztül nem nőtt az eddig talált legnagyobb f folyam nagysága.

Kezdetben $\delta = 0$, azaz csak a minimális élszámú utakat fogadjuk el. A későbbi iterációk során ez az érték fokozatosan nő, ha az allokációk és deallokációk során a talált folyam nagysága adott ideig nem nő. Egészen pontosan $\delta = \lfloor \frac{r_u}{3} \rfloor$, ahol r_u jelöli azon iterációk számát mikor az f folyam nagysága nem nőtt. Ez $r_u = 100$ sikertelen iteráció után $\delta > > 33$, ami a gyakorlatban nem jelent megszorítást.

5.2.5. Deallokáció

A deallokációs fázis a módszer szempontjából kiemelkedően fontos. Az utak véletlenszerű eltávolítása helyett a következő módszert használjuk. Először azonosítjuk a szűk keresztmetszetet jelentő élek E_{d_i} halmazát. Emlékezzünk, hogy az allokációs fázis akkor ér véget, ha az aktuális (s_i, t_i, d_i) terméket nem tudjuk megengedetten elvezetni. Ebben az esetben E_{d_i} egyszerűen számolható egy $s_i - t_i$ vágásból a gráfban. Egy $e = (u, v)$ él akkor és csak akkor lesz eleme E_{d_i} halmaznak, ha

- u elérhető s_i csúcsból megengedett éleken $(c(e) - f(e) \geq d_i)$
- v nem elérhető s_i csúcsból megengedett éleken

Ha az utat a δ út hossz korlát miatt nem fogadjuk el, azaz létezik megengedett út s_i és t_i között, de az túl hosszú, akkor az E_{d_i} halmazt a következőképpen határozzuk meg. Egy

$e = (u, v)$ él akkor és csak akkor eleme E_{d_i} halmaznak, ha

- $(c(e) - f(e) < d_i)$, azaz e lehet hogy szűk keresztmetszet
- $r_{s_i,u} + 1 + SPL_{v,t_i} \leq SPL_{s_i,t_i} + \delta$, ahol $r_{s_i,u}$ az s_i csúcsból az u csúcsba vezető minimális élszámú megengedett út $(c(e) - f(e) \geq d_i)$ hossza

A fenti egyenlőtlenség jobb oldalán az út hossz korlát áll, míg a bal oldalon a jelenlegi távolság s_i és u között $(d_{s_i,u})$, az (u, v) él, és a minimális élszámú út v csúcsból t_i csúcsba (SPL_{v,t_i}). Így, az egyenlőtlenség teljesülése esetén, ha kapacitást szabadítunk fel e élen, lehet hogy az (s_i, t_i, d_i) terméket a következő iterációban már el tudjuk vezetni. Ez a módszer különösen fontos, ha adott mértéknél hosszabb utakat nem kívánunk elfogadni.

Miután az E_b halmazt kiszámítottuk, meghatározzuk az utak azon P_{cand} halmazát, amiből a felszabadítani kívánt utakat fogjuk kiválasztani. Csak azok az utak jönnek szóba, melyek tartalmaznak szűk keresztmetszetnek számító éleket. $P_{cand} = \{P_i \in P | P_i \cap E_b \neq \emptyset\}$

Ezután eltávolítunk néhány utat a szűk keresztmetszetet jelentő élekről, és a hozzájuk tartozó termékeket később más utakon újra elvezetjük.

5.2.6. Büntető függvény

A $\lambda : E \rightarrow \mathbb{R}$ büntetőfüggvény a módszer fontos része, aminek segítségével megpróbáljuk elkerülni a szűk keresztmetszetet jelentő éleket $(\lambda_e = \lambda(e))$. Kezdetben $\lambda_e = = 1.0 \ \ \forall e \in E$ élre. Minden iterációban az E_b halmazban található élekre a következőképpen változtatjuk meg $\lambda'_e := \lambda_e + \epsilon_\lambda$. A mi implementációnkban $\epsilon_\lambda = 0.1$.

A büntető függvény minden iterációban dinamikusan változik, így a Dijkstra algoritmushoz használt súlyfüggvény is. Ha $\lambda_e > 1$, akkor lehetségessé válik nem minimális élszámú utak választása. Ha minden minimális élszámú út szűk keresztmetszetet jelentő éleken keresztül vezet, valamivel hosszabb utakkal próbálkozunk, hogy a szűk keresztmetszetet jelentő éleket elkerüljük.

5.2.7. A felszabadított utak száma

Miután meghatároztuk E_{d_i} segítségével az utak azon P_{cand} halmazát, amiből a felszabadítani kívánt utakat fogjuk kiválasztani, néhány utat fel kell szabadítani, azaz el kell távolítani P és P_{cand} halmazokból, csökkenteni kell az út mentén $f(e)$ értékét d_i-vel, és vissza kell helyezni az (s_i, t_i, d_i) terméket a fel nem dolgozott termékek D halmazába.

A felszabadítandó utak száma q iterációnként változó. Kezdetben a $q := q_{min} = 5$ értéket használjuk. Később, ha f folyam nagysága nem nőtt adott számú allokációs-deallokációs fázison keresztül q értékét növeljük meg. $q := q + q_\delta$, ahol $q_\delta = \max\{\frac{|V|}{10}, 5\}$. Hasonlóan, ha adott számú iteráción keresztül f értéke nőtt, csökkentsük a felszabadítani kívánt utak számát. $q := \max(q_{min}, q - q_\delta)$

Ablak méretnyi $r_{ws} = 5$ iterációt várjunk, mielőtt q értékét bármely irányba módosítjuk.

5.2.8. A felszabadítandó utak kiválasztása

Ez a fázis azután következik, hogy kijelöltük az utak P_{cand} halmazát, amiből a felszabadítani kívánt utakat fogjuk kiválasztani, és meghatároztuk a felszabadítani kívánt utak számát (q).

A felszabadítandó utak 20%-át véletlenszerűen választjuk ki P_{cand} halmazból, majd felszabadítjuk. A maradék 80%-ot egyesével választjuk ki, a következő módon. Veszünk K különböző utat, véletlenszerűen, P_{cand} halmazból. Azt a (s_i, t_i, d_i) termékhez tartozó utat szabadítjuk fel, amire $\frac{|P| - SPL_{s_i,t_i}}{d_i}$ maximális. Azaz lehetőleg olyan termékeket próbálunk újra elvezetni, amiket nem a minimális élszámú úton vezettünk el, és a kapacitás igényük nem túl nagy.

5.2.9. Az aktuális termék megtartása

Mivel szeretnénk megakadályozni, hogy minden eltávolított terméket ugyan úgy vezessünk el mint korábban, azt a terméket, amelyiket nem tudtunk elvezetni, a feldolgozatlan termékek D listájának elejére helyezzük. Az adott terméket a nehézség szerinti rendezés során is a lista elején tartjuk, és a következő allokációs fázisban ezt a terméket próbáljuk elvezetni. Ha így sem sikerül a terméket elvezetni, újabb deallokációs fázis következik,

amíg végül sikerül.

5.2.10. Terminálási feltétel

Az algoritmus kétféleképpen állhat le. Minden terméket elvezettünk, $D = \emptyset$, vagy ha az f folyam nagysága adott számú iteráción keresztül nem nőtt. Ezt az értéket $q_{term} = 1000$ iterációra állítottuk be.

5.2.11. Nagyobb deallokáció

A korábban már ismertetett deallokációs fázison túl, ha a szűk keresztmetszetnek számító élek felhasználásával hosszú ideig, $\frac{q_{term}}{2}$ iteráción keresztül, nem tudjuk f nagyságát növelni, a következőképpen járunk el.

Az eddig elvezetett termékek 12.5%-át felszabadítjuk. Minden lépésben véletlenszerűen választunk egy P utat, majd $P_{drop}(P, s_i, t_i) = 1 - \frac{1}{2}(\frac{SPL_{s_i,t_i}}{|P|})^2$ valószínűséggel felszabadítjuk.

Azaz ha nagyobb az eltérés P és az azonos termékhez tartozó minimális élszámú út között, akkor nagyobb eséllyel szabadítjuk fel az utat, de a minimális élszámú utakat is felszabadítjuk $\frac{1}{2}$ valószínűséggel. Ez addig folytatódik, amíg az eddig meghatározott utak 12.5%-át fel nem szabadítjuk.

5.2.12. Optimalizációs fázis

A megengedett megoldás megtalálása után a megoldást tovább finomítjuk, valamilyen optimalizációs cél szerint, a korábban ismertetett allokációs-deallokációs módszerhez hasonlóan. Az optimalizációs cél szempontjából rossznak számító utakat felszabadítjuk, a felszabadított utak e élein, a λ_e értéket növelve, megpróbáljuk új úton elvezetni az igényeket. Ha az optimalizáció célfüggvénye szerint jobb megoldást találtunk, azt eltároljuk. Ilyen optimalizációs cél lehet például $\max_{e \in E} \frac{\sum_{i|e \in P_i} d_i}{c(e)}$ minimalizálása, ekkor a leginkább terhelt élekről szabadítunk fel utakat, a Nagyobb deallokáció című fejezetben leírtak szerint. A rossz élek λ értékeit frissítjük és újraszámoljuk az utakat.

5.3. Kiegészítés a technológiai követelmények teljesítéséhez

Az algoritmus MPLS(Multiprotocol Label Switching) hálózatok kezelésére lett kitalálva, amiben az adatforgalom előre meghatározott útvonalakon LSP-ken (Label Switching Paths) történik. Először ismertetem az ezzel kapcsolatos speciális technikai követelményeket, úgy mint

- Tartalék útvonalak
- Előírt útvonalak
- Maximális úthossz

Majd ezen problémák kezeléséhez az algoritmuson szükséges változtatások leírása következik.

5.3.1. Tartalék útvonalak

Abban az esetben ha az LSP fontos, és a kapcsolat nem szakadhat meg, szükség lehet tartalék LSP-re. A tartalék LSP lefoglalhat némi erőforrást, aztán készenlétben várakozhat, hogy szükség esetén az adatforgalmat gyorsan át tudja venni. Például ha az egyik él a gráfból kiesik. Ezért a tartalék útvonalnak éldiszjunktnak kell lennie az eredeti útvonaltól. A tartalék és az eredeti LSP tulajdonságai különbözőek lehetnek. Például a sávszélesség vagy az út maximális hossza, de lehet olyan megkötés is, hogy bizonyos éleken mindenképp keresztül kell vezetni. Tipikusan azonos vagy kisebb sávszélesség és nagyobb maximális úthossz jellemző a tartalék útvonalakra.

5.3.2. Előírt útvonalak

Néha szükség lehet élek vagy csúcsok meghatározására, amiken az LSP-knek át kell haladnia. Ilyen megkötésekkel, annak eldöntése, hogy létezik-e ilyen útvonal, egyetlen termelő-fogyasztó párra is NP-teljes probléma.

5.3.3. Maximális úthossz

Mivel jelentősen terhelt hálózatokban a szűk keresztmetszetnek számító éleket néha csak a minimálisnál jóval hosszabb élszámú útvonalon lehet elkerülni, szükség lehet az utak hosszának korlátozására. Az úton szereplő élek száma ugyanis hatással van a késleltetésre.

5.3.4. Termékek bonyolultsági mértéke

A termékek bonyolultságát, azaz elvezetésük nehézségét, az előírt és a tartalék útvonalak és a maximális úthossz is befolyásolják. Megjegyezzük, hogy a tartalék útvonallal rendelkező termékeknél a tartalék és az eredeti útvonalat közösen kezeljük a terméken belül. A közös maximális úthossz a két maximális úthossz közül a kisebb, a közös igény pedig az igények összege. A termékek feldolgozásának sorrendjét lexikografikus rendezéssel határozzuk meg. Fontossági sorrendben

- Előírt útvonalak létezése
- Tartalék útvonal szükségessége
- A termék eredeti nehézségi mértéke

5.3.5. Út számítás

Tartalék útvonal szükségessége esetén a két útkeresést külön hajtjuk végre. A tartalék útvonal keresésekor az eredeti útvonal élei tiltottak. A fenti módszer bizonyos esetekben nem ad megengedett megoldást. Ebben az esetben megpróbáljuk az eredeti útvonalat megváltoztatni és új tartalék útvonalat keresni. Ha nem találunk tartalék útvonalat az eredeti útvonalak egyikéhez sem, megpróbálkozunk a sorrend felcserélésével, és a tartalék útvonalat keressük meg először.

Előírt útvonal létezése esetén az útkeresést részekre bontjuk, és egy rész kiszámításához az eredeti útkereső algoritmust használjuk, de megkötéseket teszünk a csúcsokra és az élekre, hogy elkerüljük a köröket, és hogy jó sorrendet találjunk az előírt élekre és csúcsokra. Végül ellenőrizzük a maximális úthossz megkötést. Ha ez sérül, az útkeresést megismételjük a λ_e értékek figyelembe vétele nélkül.

Ha nincsenek se előírt, se tartalék útvonalak, de a maximális úthossz követelmény nem teljesül, akkor a szűk keresztmetszetet jelentő E_{d_i} éleket a korábban ismertetett módon határozzuk meg.

6. Az általános többtermékes osztatlan folyam

Ebben a fejezetben Józsa Balázs Gábor, Király Zoltán, Magyar Gábor és Szentesi Áron [6] többtermékes osztatlan folyamok problémájára adott módszerét alapul véve fogok a következő kérdésekre választ keresni.

1. Milyen hatékonysággal használható a módszer prioritások kezelésére?
2. Lehet-e a talált megoldást javítani, a maradék kapacitások felhasználásával? (az algoritmus azt a feladatot oldja meg, mikor a termékek olyan T részhalmazát keressük, melyet osztatlanul el lehet vezetni, és amire $\sum_{i \in T} d_i$ maximális)
3. Hogyan lehet az algoritmust annak a feladatnak a megoldására használni, mikor arra keresünk választ, mi az a legkisebb $\alpha \in \mathbb{R}$ amivel a kapacitásokat megszorozva már minden terméket osztatlanul el tudunk vezetni? A cél egy jó felső korlát α értékére.
4. Hogyan lehet a 3. pontban talált felső korlátot tovább javítani?

6.1. Prioritások

A GPO algoritmus apró módosításával lehetségessé válik a termékek prioritásának kezelése. Egyedül a feldolgozatlan termékek nehézség szerinti rendezését kell úgy módosítani, hogy a kisebb prioritású termékeket a lista elejére tegye. (minél kisebb prioritású a termék annál fontosabb) Mint látni fogjuk, ennek a módszernek a segítségével könnyen kezelni tudjuk a prioritásokat, és nem fordul elő, hogy fontos termékeket nem tudunk elvezetni kevésbé fontos termékek miatt.

6.1.1. Tesztadatok

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő, azzal a különbséggel, hogy minden termékhez véletlenszerűen hozzárendeltem egy prioritási értéket is. A tesztadatok a priority$x$ és priorityd$x$ fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

Élek száma: 1000

Termékek száma: 1000

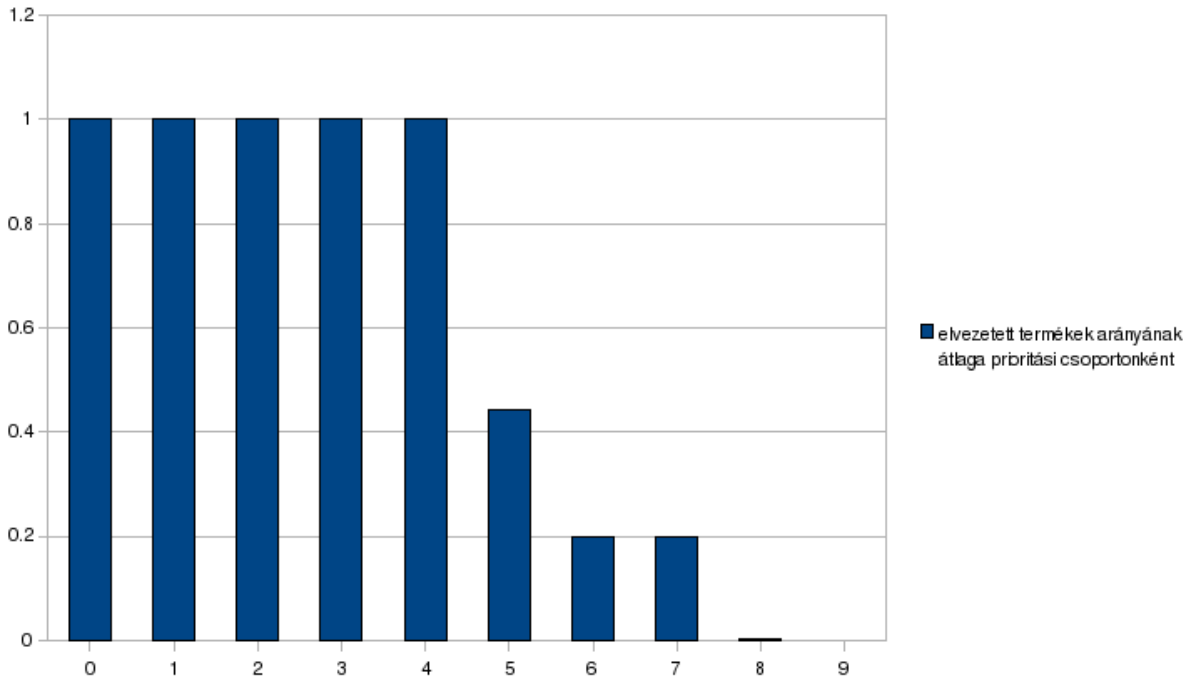
Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

Termékek prioritása: 0 és 9 között véletlenszerűen

A prioritásokat úgy kell értelmezni, hogy a kisebb prioritású a fontosabb.

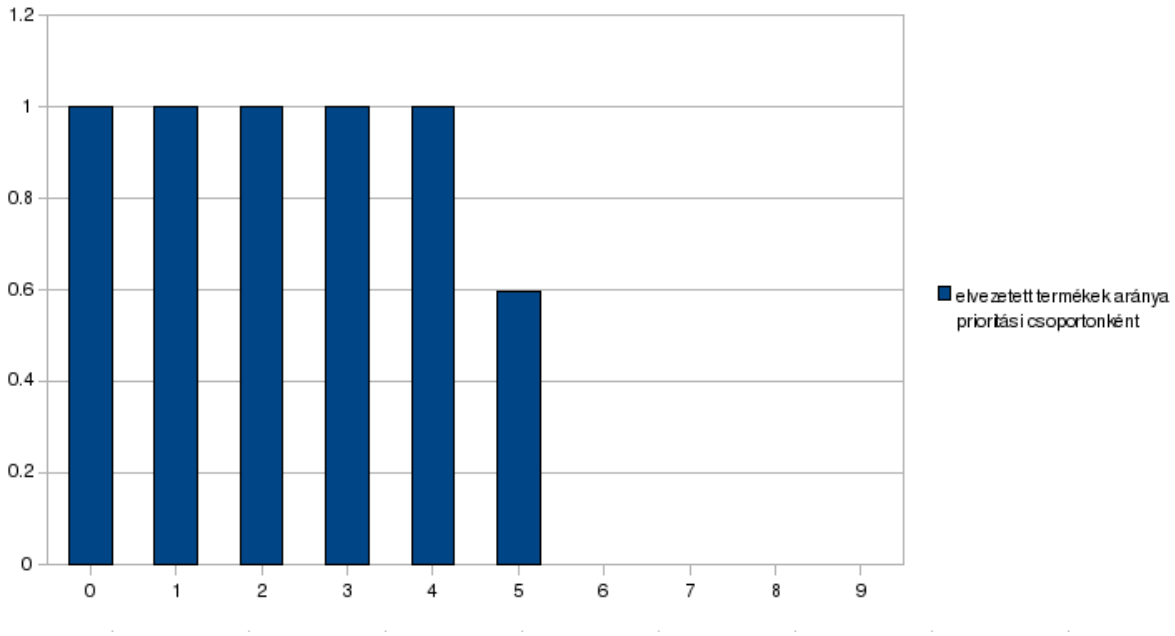
6.1.2. Tesztek eredménye

Az alábbi ábrán az elvezetett termékek arányának átlaga látható prioritási csoportonként. Azaz az általam vizsgált tizenöt teszteset mindegyikére egyesével megnézem, hogy például, a 0 prioritású termékek mekkora részét vezette el az algoritmus, majd veszem ezeknek az értékeknek az átlagát.



Látható, hogy az első öt kategóriába tartozó termékeket az algoritmus minden esetben el tudta vezetni, de ezután a teljesítménye gyorsan romlott. A legkevésbé fontos termékeket egyetlen esetben sem vezette el.

Vizsgáljuk meg külön a tizenhármas sorszámú tesztesetet.



Az algoritmus, az első öt prioritási kategóriába tartozó termékeket mind elvezette(0-4), az ötös prioritású termékeket részben elvezette, a többit nem tudta elvezetni. Ez a fajta felbontása az elvezetett és nem elvezetett termékeknek az összes többi tesztesetre is jellemző volt.

Meg kell még jegyezni, hogy bár a prioritások ilyen kezelése a fontos termékek elvezetése szempontjából igen hatásos, minél több prioritási kategóriát vezetünk be, annál inkább akadályozzuk az eredeti heurisztikát, ami a termékek nehézsége szerint rendezte a fel nem dolgozott elemeket. Legjobban akkor működik a módszer, ha csekély számú fontos terméket mindenképpen el akarunk vezetni, a termékek nagy része azonban nem kiemelt fontosságú, így csak két kategória van.

6.2. Maradék kapacitások

Mikor az algoritmus egy terméket nem tud osztatlanul elvezetni, elkezd korábbi utakat felszabadítani a szűk keresztmetszetnek számító élekről, és addig próbálkozik, amíg sikerrel nem jár, vagy terminál. Néha azonban, jobb lenne, ha a terméket egyszerűen átugorná. Mivel a feldolgozatlan elemek listáját nehézség szerint rendeztük, előfordulhat, hogy a

hátralévő termékeket könnyen el lehetne vezetni, de soha nem jutunk el hozzájuk, a nehezebben elvezethető termékek miatt. Ennek a problémának a megoldására a következő módszert lehet használni. Miután az algoritmus terminált, végezzünk utófeldolgozást a feldolgozatlan termékek listáján és egy mohó algoritmussal vezessük el a megmaradt termékeket a maradék kapacitásokon. Ezt kétféleképpen is tehetjük. Ha a cél minél több termék teljes elvezetése, járjunk el a következőképpen.

Induljunk ki a legjobb talált megoldásból

Vegyük sorra a fel nem dolgozott termékek listájának elemeit

1. Vegyünk egy s_i, t_i, di elemet a fel nem dolgozott termékek listájából
2. Keressük meg egy módosított Dijkstra algoritmussal a legszélesebb utat, ennek a szélessége legyen K
3. Ha $d_i \leq K$ vezessük el a terméket a talál úton
4. Ha nem értünk a lista végére, ugorjunk az 1. lépésre és vegyük a következő elemet

Ha van értelme a termékeket részben elvezetni, azaz $K < d_i$ kapacitású utat lefoglalni egy terméknek, akkor járjunk el a következőképpen.

Induljunk ki a legjobb talált megoldásból

Vegyük sorra a fel nem dolgozott termékek listájának elemeit

1. Vegyünk egy s_i, t_i, di elemet a fel nem dolgozott termékek listájából
2. Keressük meg egy módosított Dijkstra algoritmussal a legszélesebb utat, ennek a szélessége legyen K
3. Vezessük el a terméket, vagy a termék akkora részét amekkorát lehetséges, a talált úton. A lefoglalt kapacitás legyen $\min(d_i, K)$
4. Ha nem értünk a lista végére, ugorjunk az 1. lépésre és vegyük a következő elemet

Ennek például egy számítógép hálózat esetében lehet értelme. Ha a kevésbé fontos üzeneteknek nem is tudjuk a kívánt sávszélességet a rendelkezésére bocsátani, lehetőség szerint biztosítunk valamennyi sávszélességet.

6.2.1. Tesztadatok A

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a testa$x$ és testad$x$ fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

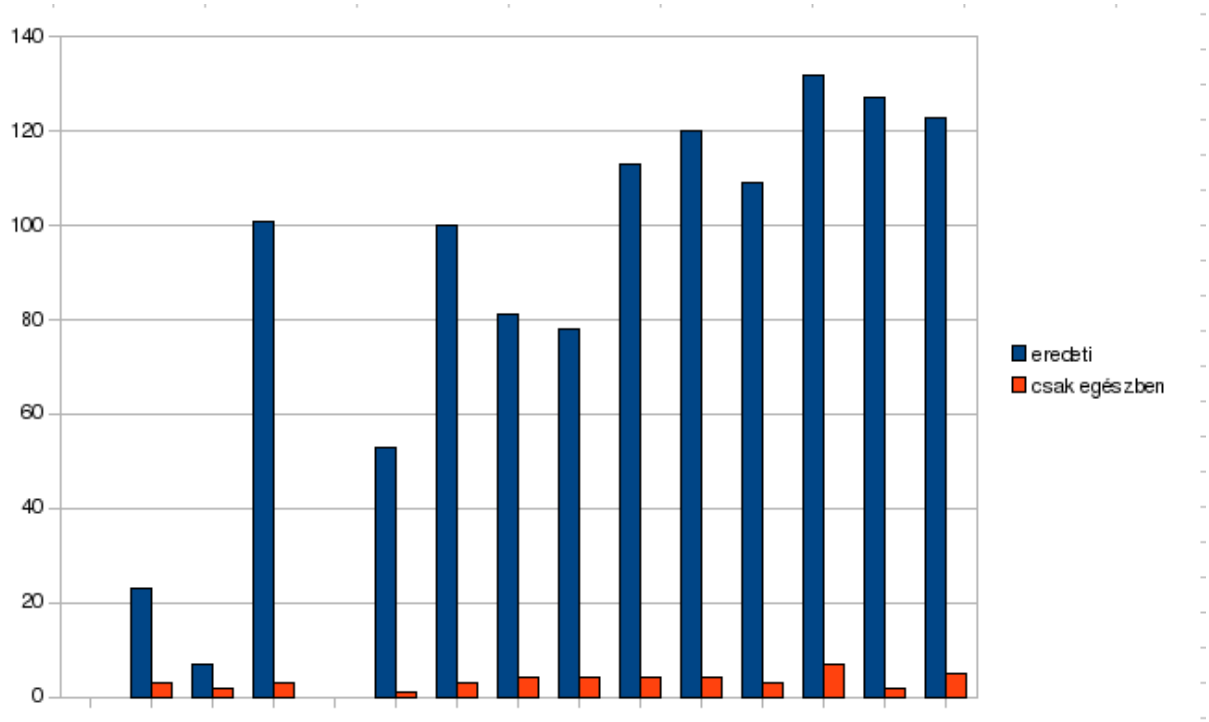
Élek száma: 1000

Termékek száma: 5000

Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

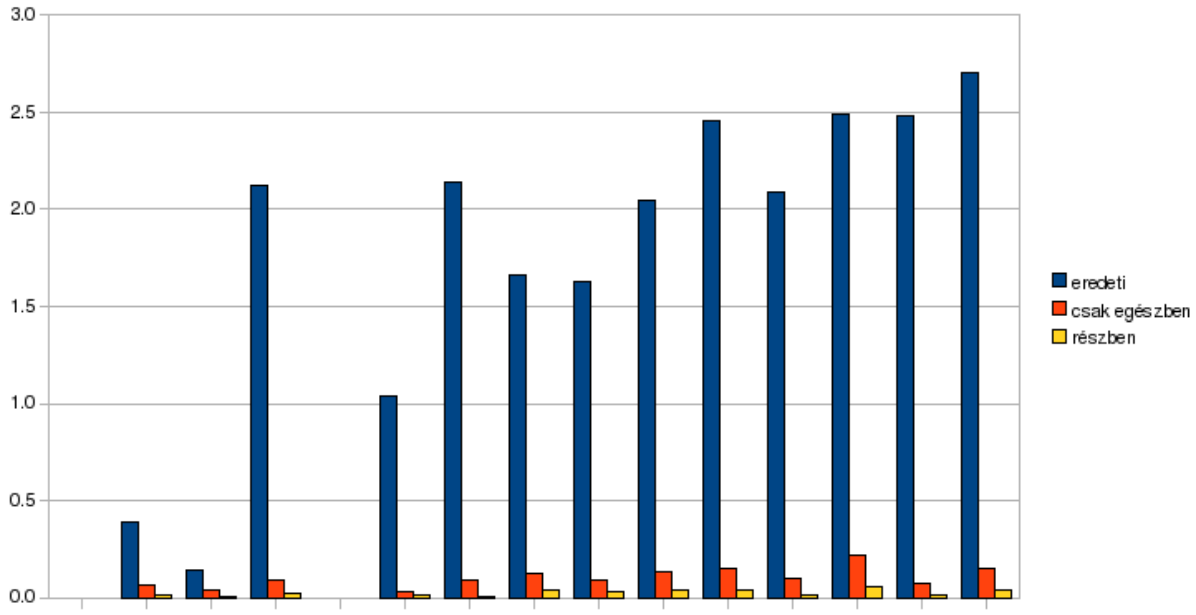
6.2.2. Tesztek eredménye A

A következő ábráról az olvasható le, hány terméket nem tudtunk elvezetni az ötezerből, a különböző módszerekkel, úgy mint az eredeti GPO algoritmussal, és azzal a verzióval amikor utólag megpróbáljuk elvezetni azokat a termékeket amelyeket egyben el lehet. Arra a verzióra, mikor a termékek részeinek is megengedjük az elvezetését, minden tesztesetre sikerült, legalább részben, mind az ötezer terméket elvezetni, így azt nem szerepeltettem az ábrán.



Látható hogy a módszer hatásosnak bizonyult, jó pár további terméket el lehetett egyben vezetni. Abban az esetben pedig, mikor a termékek részeinek elvezetését is megengedjük, minden alkalommal sikerült az összes terméket, legalább részben, elvezetni. Megjegyzem hogy két esetben már az eredeti algoritmus is megtalálta a megoldást, így nem voltak maradék termékek.

Következőnek vizsgáljuk meg azt, hogy az egyes módszerekkel a termékek kapacitásigényének hány százalékát nem tudtuk elvezetni.



Már az eredeti algoritmus is kevesebb mint három százalékát nem tudta az igényeknek elvezetni, de az utófeldolgozás használatával minden esetben jelentősen csökkenteni tudtuk az el nem vezetett igények arányát.

6.2.3. Tesztadatok B

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a normaltest$x$ és normaltestd$x$ fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

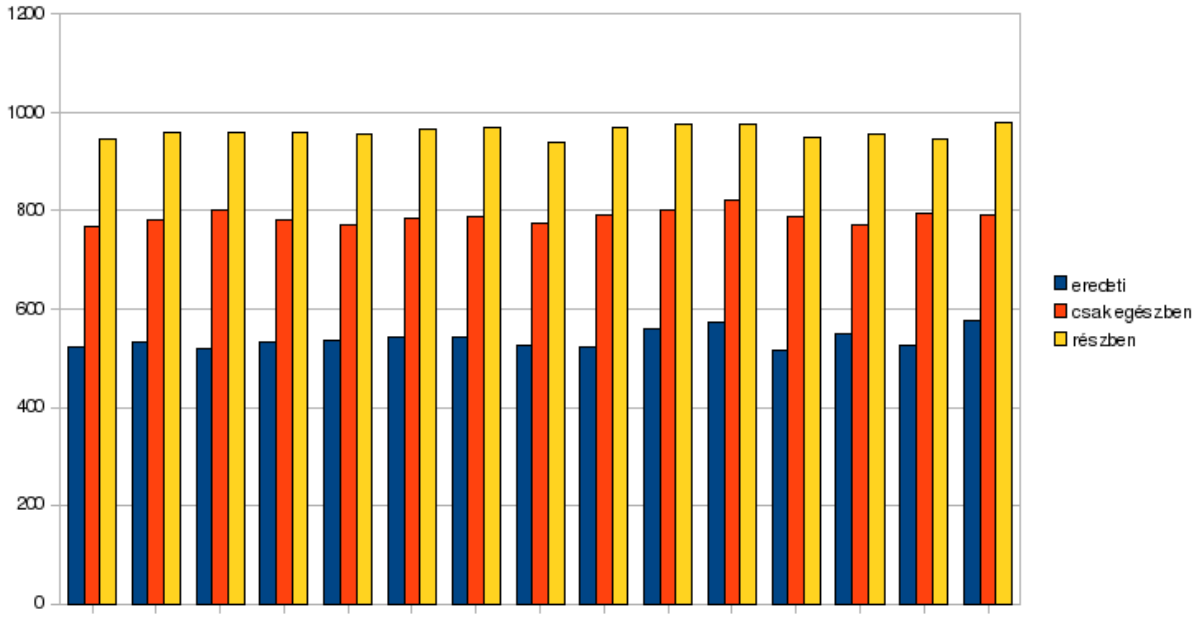
Élek száma: 1000

Termékek száma: 1000

Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

6.2.4. Tesztek eredménye B

A következő ábráról leolvasható, hogy az egyes algoritmus változatok, az eredeti GPO algoritmus utófeldolgozás nélkül,GPO algoritmus utófeldolgozással, amikor csak akkor vezetünk el egy terméket mikor azt egészben el lehet és az a verzió, mikor megengedjük, hogy egyes termékeket részben vezessünk el, hány terméket tudott 1000 termékből egészben, vagy részben elvezetni.

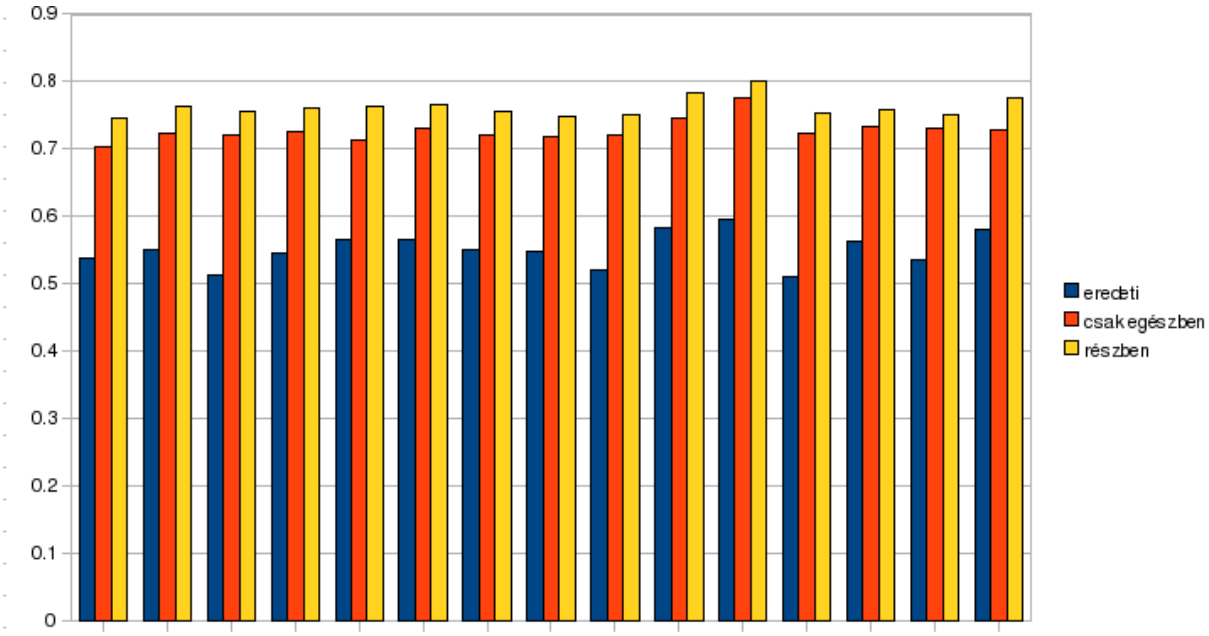


Az algoritmusok szinte minden teszteseten ugyanúgy teljesítettek. Az eredeti algoritmus öt hatszáz terméket tudott elvezetni, a csak egész termékeket engedélyező verzió hétszázötven és nyolcszázötven terméket, míg a töredék elvezetéseket is megengedő módszer kilencszáz terméknél is többet tudott egészben, vagy részben elvezetni, minden esetben.

Mivel az ábráról nehezen leolvashatók a különbségek, a pontosság kedvéért táblázatos formában is megadom az értékeket.

	Teszt 1	Teszt 2	Teszt 3	Teszt 4	Teszt 5	Teszt 6	Teszt 7	Teszt 8	Teszt 9	Teszt 10	Teszt 11	Teszt 12	Teszt 13	Teszt 14	Teszt 15
eredeti	523	534	518	534	536	544	543	525	523	559	573	516	550	525	578
csak egészben	769	780	801	782	772	784	788	775	791	801	820	787	772	794	792
részben	947	960	960	961	957	967	968	940	971	977	977	950	957	947	980

Az alábbi ábrán az egyes algoritmus változatok által elvezetett igények aránya található, az összes igényhez képest.



A helyzet az elvezetett termékek számához hasonló, de itt nincs akkora különbség az egyes módszerek között, mivel a legnagyobb igényeket az eredeti algoritmus már megpróbálta elvezetni.

6.3. Felső korlát

A GPO algoritmus eredetileg azt a feladatot oldja meg, mikor a termékek olyan T részhalmazát keressük, melyet osztatlanul el lehet vezetni, és amire $\sum_{i \in T} d_i$ maximális. Szeretnénk ezt a komplex és hatékony heurisztikus módszert arra a feladat változatra is használni, mikor keressük azt a legkisebb $\alpha \in \mathbb{R}$ számot, amivel a kapacitásokat megszorozva már minden terméket osztatlanul el tudunk vezetni.

A következő módszert javaslom, ami a GPO algoritmus felhasználásával felső korlátot ad α-ra. Egy speciálisan módosított Dijkstra algoritmus és dinamikusan változó költségfüggvény segítségével végezzünk utófeldolgozást a fel nem dolgozott termékek listáján.

Induljunk ki a legjobb talált megoldásból

Vegyük sorra a fel nem dolgozott termékek listájának elemeit

1. Vegyünk egy s_i, t_i, di elemet a fel nem dolgozott termékek listájából
1. Legyen a költségfüggvény $\forall e \in E$ élre $cost(e) = \frac{d_i + f(e)}{c(e)}$
3. Keressük meg egy módosított Dijkstra algoritmussal azt az utat, amelyiken a terméket elvezetve a legkevésbé nő $\max \frac{f(e)}{c(e)}$
4. vezessük el a terméket a talál úton, a kapacitások figyelembe vétele nélkül
5. Ha nem értünk a lista végére ugorjunk az 1. lépésre és vegyük a következő elemet

A Dijkstra algoritmust úgy kell módosítani, hogy összeadás helyett a két érték maximumát adja vissza. Ezt a widest_path_traits.h fájlban, a GPOOperationTraits struktúra segítségével valósítom meg.

Megjegyzendő, hogy bár most a célom $\max \frac{f(e)}{c(e)}$ minimalizálása, megfelelően meghatározott költségfüggvénnyel és Dijkstra algoritmussal, a módszert más feladatok megoldásához is használhatjuk, például kereshetünk felső becslést $\max f(e) - c(e)$ minimumára is.

6.3.1. Tesztadatok

Először, hogy összehasonlíthassuk a módszer hatékonyságát a DGG algoritmussal, valamint a korábban javasolt, DGG algoritmust használó iteratív módszerrel, egytermelős feladatokon futtattam.

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a normal_x és normal_dx fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

Élek száma: 2000

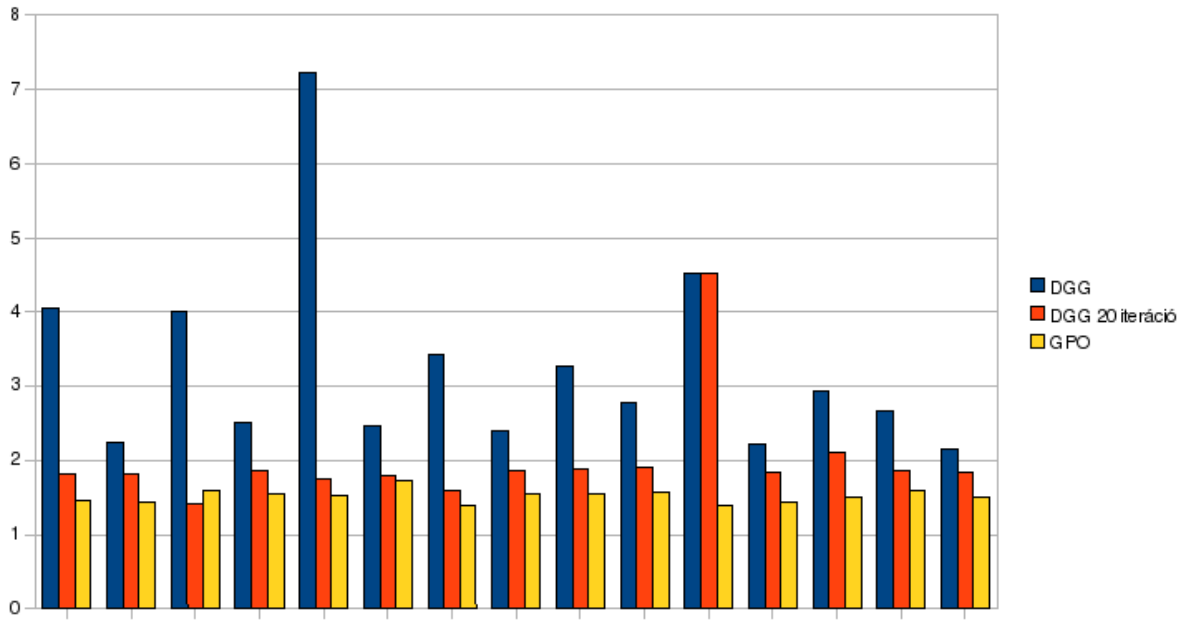
Termékek száma: 1000

Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

Az éleken található költségeket kezdetben a következőképpen határoztam meg. Legyen a költség e élen $cost(e) = \frac{10C_{max}}{c(e)}$, ahol C_{max} a kapacitások maximuma, $\forall e \in E$ élre, ha $c(e) > 0$, különben legyen C_{max}.

6.3.2. Tesztek eredménye

A következő ábrán a DGG algoritmussal, a DGG algoritmust használó iteratív módszerrel 20 iteráció után, és a GPO algoritmussal a fent leírt módszerrel talált $\max \frac{f(e)}{c(e)}$ értékek láthatóak.



Látható, hogy a módszer a DGG algoritmusnál messze jobban teljesít, míg a DGG algoritmust használó iteratív módszer 20 iteráció után csak egy esetben jobb nála.

6.4. Felső korlát javítása

A fent megadott módszerrel keresett felső korlát $\alpha \in \mathbb{R}$ értékre tovább javítható a következő módon. Keressük a korlátot logaritmikus kereséssel, a GPO algoritmus segítségével. A következő képlet segítségével könnyen adhatunk egy alsó korlátot $\alpha \in \mathbb{R}$ értékre n termék esetén $\alpha \geq \frac{\sum_{i=1}^{n} SPL_i d_i}{\sum_{e \in E} c(e)}$, ahol SPL_i az s_i és t_i csúcsok közötti legkevesebb élszámú

út hossza. A felső korlát pedig legyen kezdetben a GPO algoritmus által a korábban ismertetett módszerrel talált korlát.

Induljunk ki a legjobb talált megoldásból

$L :=$alsó korlát

$R :=$felső korlát

1. Futtassuk a GPO algoritmust úgy, hogy a kapacitásokat az eredeti kapacitások $\frac{L+R}{2}$-szeresére állítjuk
2. Ha az algoritmus talál osztatlan megoldást, azaz $D = \emptyset$ miatt terminál $R := \frac{L+R}{2}$, ellenkező esetben $L := \frac{L+R}{2}$
3. Ha elértünk egy előre megadott számú iterációt (ezt a mérésekhez ötre állítottam be) leállunk, ellenkező esetben folytatjuk az 1. lépéstől

Mivel a gyakorlatban újra és újra lefuttatni a GPO algoritmust roppant időigényes lenne, a következőképpen járunk el. Az 1. lépésben a GPO algoritmus egy olyan változatát hívjuk meg, ami nem üres P úthalmazzal indul, hanem a legjobb talált megoldásból származó feldolgozatlan termékek D listájával és P úthalmazával. Valamint nem akkor terminál, ha 1000 iteráción keresztül nem javul a megoldás, hanem már 100 iteráció után.

6.4.1. Tesztadatok A

Először, hogy a módszer hatékonysága a DGG algoritmust használó iteratív módszerrel összehasonlítható legyen, az algoritmust egytermelős feladatokon futtattam.

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a normal_x és normal_dx fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

Élek száma: 2000

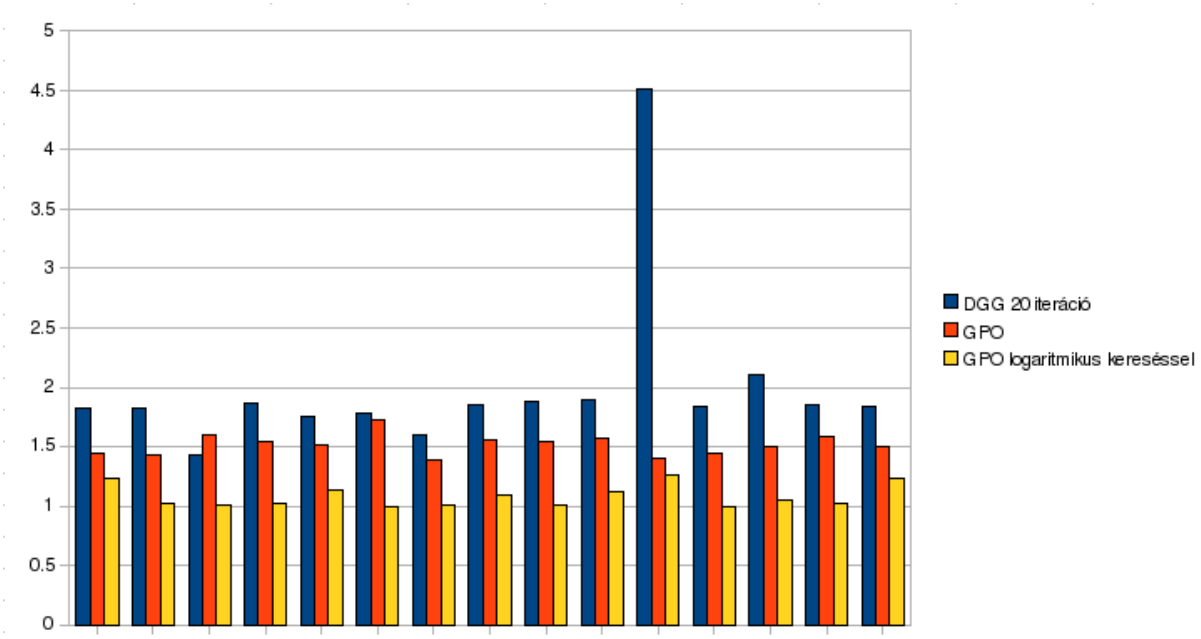
Termékek száma: 1000

Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

Az éleken található költségeket kezdetben a következőképpen határoztam meg. Legyen a költség e élen $cost(e) = \frac{10C_{max}}{c(e)}$, ahol C_{max} a kapacitások maximuma, $\forall e \in E$ élre, ha $c(e) > 0$, különben legyen C_{max}.

6.4.2. Tesztek eredménye A

A következő ábrán a DGG algoritmust használó iteratív módszerrel 20 iteráció után, és a GPO algoritmussal, valamint a GPO algoritmust használó logaritmikus keresésre épülő fent leírt módszerrel talált $\max \frac{f(e)}{c(e)}$ értékek láthatóak.



Az esetek kétharmadában a logaritmikus keresésre épülő módszerrel talált korlát 1.1 alatt van, és több mint az esetek harmadában kevesebb mint 1.01. A DGG algoritmust használó iteratív módszernél messze jobban teljesít, de a GPO algoritmussal talált eredményen is mindig tudott javítani. Bár a feladatot úgy generáltuk, hogy kapacitás növelés nélkül is legyen megoldás, annak figyelembevételével, hogy a többtermékes osztatlan folyamok problémája NP-teljes az eredmény nagyon jónak mondható. Különösen, hogy a külön egytermelős többtermékes osztatlan folyamok problémája kidolgozott DGG algoritmust messze felülmúlja a vizsgált feladatokon.

6.4.3. Tesztadatok B

Másodszor, az általánosabb, többtermelős feladatokat vizsgáltam, aminek megoldására a GPO algoritmust megalkották.

A teszteléshez használt gráfokat és termékeket a Tesztadatok generálása fejezetben leírt módon állítottam elő. A tesztadatok a normaltest$x$ és normaltestd$x$ fájlokban találhatók, ahol x a teszt sorszáma. A tesztadatok paraméterei a következők.

Csúcsok száma: 400

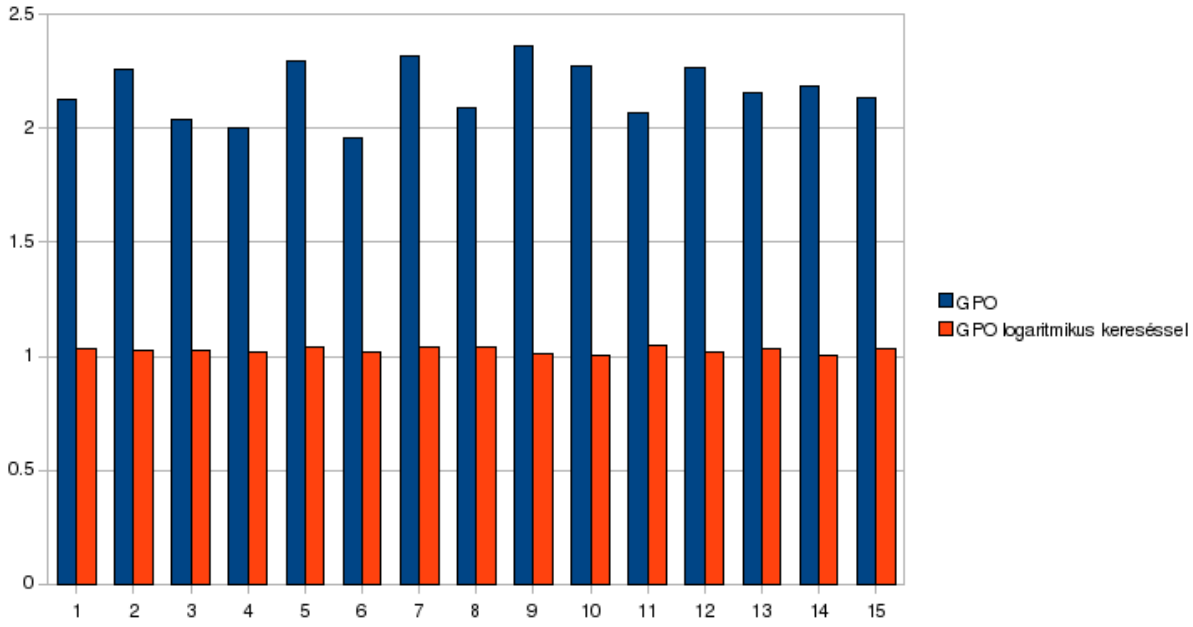
Élek száma: 1000

Termékek száma: 1000

Termékek kapacitásigénye: 10 és 1000 között véletlenszerűen

6.4.4. Tesztek eredménye B

A következő ábrán a GPO algoritmus által a korábban ismertetett módszerrel $\alpha = \max \frac{f(e)}{c(e)}$ értékre talált korlátok és a GPO algoritmust használó logaritmikus keresésre épülő módszer által talált korlátok láthatók.



A GPO algoritmust használó logaritmikus keresésre épülő módszer jól szerepel. Minden esetben képes jelentősen javítani a korláton, ami alátámasztja a módszer robusztusságát. Az $\alpha = \max \frac{f(e)}{c(e)}$ értékre talált korlátok minden tesztesetre 1.05 alatt vannak. Elég az élek kapacitását 5 százalékkal megemelni, hogy a termékeket osztatlanul el tudjuk vezetni.

7. Összegzés

Mind az egytermelős, mind az általános osztatlan folyamok problémája NP-nehéz. Hatékony approximációs algoritmusokra illetve heurisztikákra van szükség.

Az egytermelős osztatlan folyamok problémájára Dinitz, Garg és Goemans [15] algoritmusára épülő iterációs módszert adtam, ami a futási idő rovására jelentősen, a mért feladatokon átlagban 34 százalékkal javította a DGG algoritmus teljesítményét. A módszer lényege, hogy az osztatlan folyammá alakítandó osztott folyamot minimális folyam formájában keresi. Az élek költsége kezdetben a kapacitásukkal fordítottan arányos, majd a költségeket minden iterációban a kapacitás túllépésének függvényében növeljük.

Az általános osztatlan folyamok problémájára, Józsa Balázs Gábor, Király Zoltán, Magyar Gábor és Szentesi Áron [6] által adott heurisztikus módszert továbbfejlesztettem. Az ezzel kapcsolatos legfontosabb eredmények:

1. Módszert adtam a termékek prioritásának hatékony kezelésére, használatakor a tesztek során a fontosabb termékek mindegyikét el tudta vezetni az algoritmus.

2. Megvizsgáltam a fel nem dolgozott termékek utófeldolgozásának lehetőségét, az utófeldolgozásra két mohó algoritmust adtam, amik meglepően hatékonynak bizonyultak. A tesztek során mind az elvezetett termékek számát mind az elvezetett kapacitások összegét jelentősen javítani tudták. Az algoritmusok a maradék kapacitások kihasználásán alapulnak, és a legszélesebb utakon vezetik el a termékeket, egy módosított Dijkstra algoritmus segítségével.

3. Kidolgoztam az algoritmus egy olyan változatát, ami nem az eredeti feladatot oldja meg, mikor a termékek olyan T részhalmazát keressük, melyet osztatlanul el lehet vezetni, és amire $\sum_{i \in T} d_i$ maximális, hanem felső korlátot ad $\alpha = \max \frac{f(e)}{c(e)}$ minimumára.

4. Kidolgoztam egy a GPO algoritmuson és logaritmikus keresésen alapuló módszert, ami a fenti korlátot jelentősen javítja. A kapott korlátot egytermelős feladatokon a DGG algoritmus eredményével összehasonlítva, a módszerem nagyságrendekkel jobban teljesített.

A fenti módszerekhez szükséges algoritmusokat c++ nyelven, a LEMON programkönyv-

tár segítségével implementáltam. Az egyes feladatok megoldásához a következő algoritmus változatokat használtam.

egytermelős: flow_to_unsplittable_flow.h, power_algorithm.h, filter_map.h, integral_filter_map.h

általános: gpo_full.h, sortstruct.h, demand.h

általános, prioritásos: gpo_full_bp.h, sortstructp.h, demandp.h, widest_path_traits.h

általános, utófeldolgozás: gpo_full_b.h, sortstruct.h, demand.h, widest_path_traits.h

általános, felső korlát α-ra: gpo_full_b.h, sortstruct.h, demand.h, widest_path_traits.h

általános, logaritmikus kereséses: gpo_full_c.h, sortstruct.h, demand.h, widest_path_traits.h

8. Melléklet

Többtermékes folyam-algoritmusok implementálása LEMON-ban

A. Bevezetés

Az egytermelős, többtermékes osztatlan folyamok problémája a következő. Több terméket kell párhuzamosan, egy közös termelő csúcsból, meghatározott fogyasztó csúcsokba eljuttatni, adott gráfban. Az egyes fogyasztókhoz tartozó igényeket egy út mentén kell elvezetni úgy, hogy a gráf élein adott kapacitásokat az összegük nem lépheti túl. Ez a problémakör bárhol előkerülhet, ahol hálózatokkal kell dolgozni, legyen szó akár telekommunikációs, akár úthálózatokról, de szerepe lehet áramkör tervezési feladatok megoldásakor is. A feladat egzakt megoldása NP nehéz, ezért egy, valós méretű problémák megoldására is használható algoritmus csak közelítő megoldást adhat. A feladatnak kétféle megközelítése lehetséges. Az első, hogy közvetlenül keressük az igényeinknek megfelelő folyamot, míg a másik esetben először egy osztott folyamot keresünk, majd azt alakítjuk át osztatlan folyammá. Az általam implementált algoritmusok, Martin Skutella, valamint Yefim Dinitz, Naveen Garg és Michel X. Goemans algoritmusa a második megközelítést választották.

B. Felhasználói Dokumentáció

B.1. A feladat rövid megfogalmazása

Célom olyan algoritmusok implementálása volt, amik megoldást nyújtanak a többtermékes osztatlan folyam problémára, és a következő feltételeknek tesznek eleget:

- a Lemon könyvtáron és szabvány C++on kívül nem használ mást
- képes a Lemon bármely gráf reprezentációját használni
- képes különböző típusú folyamértékeket kezelni
- a felhasználó számára kényelmes, a Lemonban megszokott felületet biztosít a programozó számára
- nagy gráfokon is viszonylag hamar lefut

Célom volt továbbá az implementált algoritmusok viselkedésének vizsgálata különböző tulajdonságú gráfokon, valamint az algoritmusok összehasonlítása egymással.

B.2. Az algoritmusok rövid ismertetése

Definíció: Legyen egy hálózat egy irányított gráf $G = (V, E)$, nem negatív $c : E \to \mathbb{R}$ kapacitásokkal, és $w : E \to \mathbb{R}$ költségekkel az éleken, egy $s \in V$ forrás, k termék a hozzá tartozó t_i fogyasztókkal és $d_i \in \mathbb{R}$ igényekkel. Egy csúcs akárhány fogyasztót tartalmazhat.

Definíció: Egy hálózat $G = (V, E)$ gráfján értelmezett $f : E \to \mathbb{R}$ függvényt (osztott) folyamnak nevezünk, ha $f_{be}(v) - f_{ki}(v)$ kizárólag s ben negatív, és minden más csúcsban egyenlő az ott található igények összegével.

Definíció: Egy osztatlan folyam utak egy halmaza $P_1, \ldots, P_k$, ahol P_i s csúcsból a t_i fogyasztót tartalmazó csúcsba vezet. Az utak a következő f folyamot határozzák meg $f(e) = \sum_{i:e \in P_i} d_i \forall e \in E$-re.

Az algoritmusok célja: Adott egy hálózat, valamint egy a kapacitásokat nem sértő, azaz $\forall e \in E f(e) \leq c(e)$, f (osztott) többtermékes folyam. Szeretnénk találni olyan osztatlan többtermékes folyamot, ami a kapacitásokat minél kisebb mértékben, lehetőség szerint egyáltalán nem lépi túl.

B.2.1. A DGGAlgoritmus

Yefim Dinitz, Naveen Garg és Michel X. Goemans algoritmusa

Bemenet: Egy hálózat, az azon értelmezett f osztott körmentes folyammal.

Eredmény: Egy osztatlan többtermékes folyam.

Az algoritmus úgy oldja meg a feladatot, hogy a fogyasztókat a megfelelő éleken keresztül a forrásba mozgatja. Az élek, amiken a fogyasztók a forrásba jutnak alkotják az utakat, melyek az osztatlan folyamot meghatározzák. Az algoritmus figyelmen kívül hagyja azokat az éleket, amin a folyam értéke nulla.

Mozgatás: Egy v csúcsban lévő t_i fogyasztó $e = (u, v)$ élen u csúcsba mozgatása öt lépésből áll:

1. t_i fogyasztót u csúcsba helyezzük
2. csökkentjük a folyam értékét e élen d_i-vel
3. hozzáadjuk e élt P_i elejéhez
4. töröljük e élt a gráfból ha $f(e) = 0$, azaz a folyam eltűnik e élen
5. töröljük a terméket, ha a fogyasztója eléri s-t, azaz $u = s$

Megjegyzés: f osztott folyam marad.

Definíció: A t_i fogyasztó v csúcsban reguláris, ha a hozzá tartozó d_i igény nagyobb mint a folyam értéke bármely v-be vezető élen, különben irreguláris.

Definíció: Szinguláris azok az $e = (u, v)$ élek, amiknél v és minden v csúcsból elérhető csúcs kifoka legfeljebb 1.

Bevezető fázis:
Legyenek a P_i utak üresek.
Vegyük sorra a fogyasztókat.
Legyen v a t_i fogyasztót tartalmazó csúcs.
Ellenőrizzük, hogy van-e olyan bejövő él $e = (u, v)$, hogy $f(e) \geq di$.
Amennyiben igen, t_i fogyasztót u csúcsba mozgatjuk.
Ezt addig ismételjük, amíg lehetséges.

Megjegyzés: A megtartott fogyasztók regulárisak.

Ezután az algoritmus iterációkon keresztül közelít a megoldáshoz. Egy iteráció négy részből áll. Minden iteráció elején a szinguláris éleket megjelöljük. Az iteráció során szinguláris válló éleket ebben az iterációban nem fogjuk annak tekinteni. Keresünk egy alternáló kört.

A keresést egy tetszőleges v csúcsból indítjuk, majd addig követjük a kimenő éleket amíg lehetséges. Így egy előre utat kapunk. Mivel a gráf körmentes ez a keresés megáll. A feltételek miatt, a csúcsban ahol megáll van legalább egy reguláris t_i fogyasztó. Ezután

veszünk egy a t_i fogyasztót tartalmazó csúcsba mutató élt, ami különbözik az eddig bejárt élektől, és követjük visszafelé a bejövő szinguláris éleket amíg lehet, így eljutunk egy v' csúcsba, aminek több kimenő éle is van. Ezután egy előre út alkotásával folytatjuk v' csúcsból. Idővel eljutunk egy w csúcsba ahol már jártunk. A talált utak egy alternáló kört tartalmaznak.

A folyamot ennek az alternáló körnek a mentén módosítjuk, úgy, hogy az előre utakon csökkentjük a folyamot, a vissza utakon pedig növeljük ugyanazzal a mennyiséggel. A kérdéses mennyiség két érték minimuma. Az első, $\epsilon_1 > 0$ az előre éleken folyó folyam minimuma a körön. A második ϵ_2 egyenlő $min(d_j - f(e))$, ahol a minimum az összes $e = (u, v)$ élen értendő, ahol v tartalmaz legalább egy t_i fogyasztót, és a kör egy vissza útján szerepel, valamint minden t_j fogyasztóra v csúcsban, amire $d_j > f(e)$. Ezért a minimumot nem az üres halmazon vesszük, tehát $\epsilon_2 > 0$.

A program futása során az, hogy f folyam, tehát kielégíti az igényeket, invariáns tulajdonság.

Ezután a következő szabályok szerint mozgatjuk a fogyasztókat. t_i fogyasztót akkor mozgatjuk v csúcsból u csúcsba $e = (u, v)$ élen, amikor
1. e él szinguláris és $f(e) = d_i$, vagy
2. e él nem szinguláris és $f(e) \geq d_i$
Több ilyen t_i ,illetve e esetén, válasszuk azokat, melyekre $f(e) = d_i$. Az algoritmus akkor áll le, mikor minden fogyasztó eljut a termelőbe. Az algoritmusra igaz a következő:

Tétel: Az algoritmus olyan osztatlan folyamot talál, amiben $\forall e \in E$ élen az osztatlan folyam értéke az eredeti folyam értékét kevesebbel lépi túl, mint a maximális igény.

B.2.2. SA

Skutella általános algoritmusa

A speciális eset

Bemenet: Egy hálózat, amiben $d_1|d_2|\ldots|d_k$, azaz az igények osztói egymásnak, és egy (osztott) f_0 folyam a hálózaton.

Eredmény: Egy osztatlan folyam.

Definíció: Legyen $w(f) = \sum_{e \in E} w(e) * f(e)$, az f folyam költsége.

Tétel: Az algoritmus minden, a kapacitásokat nem sértő folyamhoz talál olyan osztatlan folyamot, ami a kapacitásokat bármely élen kevesebb, mint $d_{max} - d_{min}$-nel lépi túl, és melynek költsége nem nagyobb az eredeti folyam költségénél.

Definíció: Egy folyamot a-egésznek nevezünk, ha $\forall e \in Ea|f(e)$.

$i := 1; j := 0;$
while $i \leq k$ do
$j := j + 1; \delta_j := d_j;$
$\forall e \in E$-nek állítsuk a kapacitását $c_j(e) := f_{j-1}(e)$-re,
majd kerekítsük fel δ_j legközelebbi többszörösére;
keressünk egy a kapacitásokat nem sértő,
δ_j-egész f_j folyamot, amire $w(f_j) \leq w(f_{j-1})$;
töröljünk minden e élt a gráfból, amire $f_j(e) = 0$;
while $i \leq k$ és $d_i = \delta_j$ do
határozzunk meg egy tetszőleges P_i utat s-ből t_i-be G-ben;
csökkentsük f_j-t P_i mentén d_i-vel;
töröljünk minden $e \in Ef_j(e) = 0$ élt G-ből;
$i := i + 1;$
return $P_1, \ldots, P_k$;

A megfelelő δ_j-egész folyamot a következőképpen lehet előállítani az aktuális folyamból:

Vegyük az aktuális gráf azon részgráfját, ami azokból az élekből áll, amiken a folyam nem δ_j-egész.
Tetszőleges csúcsból indulva, az irányítás figyelmen kívül hagyásával, mohón határozzunk meg egy C kört ebben a részgráfban.
Növeljük az előre mutató éleken a folyamot α-val, és csökkentsük ugyanannyival a visszafelé mutató éleken a kör mentén, amíg valamely élen a folyam nem válik δ_j-egésszé.
α előjelét válasszuk meg úgy, hogy a folyam költsége ne nőjön.
Töröljük azokat az éleket, amiken a folyam δ_j-egész, és iteratívan folytatjuk amíg lehetséges.

Az általános eset

Bemenet: Egy hálózat, és egy (osztott) f_0 folyam a hálózaton.

Eredmény: Egy osztatlan folyam.

Első lépésként minden d_i igényt $d_i = d_{min} * 2^n$-re kerekítünk, ahol n a $log(\frac{d_i}{d_{min}})$ alsó egész része. Majd második lépésként úgy módosítjuk a folyamot, hogy az csak a kerekített igényeket elégítse ki. Ennek során mindig a legköltségesebb utak mentén csökkentjük a folyamot. Ezután erre a folyamra alkalmazzuk az előző algoritmust, de a kapott Pi utakon az eredeti d i igényeket vezetjük el. Ekkor igaz a következő tétel.

Tétel: Az algoritmus talál egy osztatlan folyamot, aminek a költségének felső korlátja az eredeti folyam költsége, valamint a folyam értéke bármely e élen kevesebb mint $2*f_0(e)+$ $+ d_{max}$. Pontosabban bármely e élen át irányított igények összege, egy igény kivételével, kevesebb mint az e élen folyó eredeti folyam értékének kétszerese.

C. A program használatához szükséges információk

C.1. Hardware és Software követelmények

Az algoritmusok a Lemon könyvtár segítségével lettek implementálva, így használatukhoz szükség van a Lemon 0.7es verziójára, valamint egy azzal kompatibilis fordítóra, például a GNU C++ fordítóra (g+ +) méghozzá legalább 3.3as verziószámúra.

Különösebb hardware követelmények nincsenek.
A tesztelés a következő feltételek mellett történt:
Processzor: Intel(R) Core(TM)2 CPU T5500 1.66GHz
Memória: 980MHz 0,99 GB RAM
Operációs rendszer: Ubuntu Linux (release 8.04, hardy)

C.2. A publikus függvények

C.2.1. DGGA (flow_to_unsplittable_flow.h)

template <typename GR=ListGraph, typename Value=int>
class lemon::FlowToUnsplittableFlow

Paraméterek:

GR: Az irányított gráf típusa, amin az algoritmus fut
Value: a folyamértékek és kapacitások típusa

Publikus függvények:

FlowToUnsplittableFlow(const Graph & graph, FlowMap flow, Node s, std::vector<Node> t, std::vector<Value> d)
Ez az osztály konstruktora. Meg kel adni egy irányított gráfot, egy az igényeket kielégítő folyamot, egy s termelő csúcsot, a termelővel nem azonos fogyasztó csúcsokat egy vektorban, a fogyasztók nullánál nagyobb igényeit azonos sorrendben

FlowToUnsplittableFlow()
Az osztály destruktora

void run()
Ez a függvény futtatja az algoritmust.

Lekérdező függvények:

Ezeket a függvényeket a run() meghívása után lehet használni.

FlowMap flowMap()
Az eredmény folyamot adja vissza.

Value flow(Edge edge)
Az edge élen adja vissza az eredmény folyam értékét.

Path uPath(int i)
Az iedik fogyasztóhoz tartozó utat adja vissza.

C.2.2. SA (power_algorithm.h)

template <typename GR=ListGraph, typename Value=int> class lemon::PowerAlgorithm

Paraméterek:

GR: Az irányított gráf típusa, amin az algoritmus fut
Value: a folyamértékek és kapacitások típusa

Publikus függvények:

PowerAlgorithm(Graph &graph, FlowMap flow, CostMap cost, Node s, std::vector<Node> t, std::vector<Value> d)
Ez az osztály konstruktora. Meg kel adni egy irányított gráfot, egy az igényeket kielégítő folyamot, az élek költségeit, egy s termelő csúcsot, a termelővel nem azonos fogyasztó csúcsokat egy vektorban, a fogyasztók nullánál nagyobb igényeit azonos sorrendben

PowerAlgorithm()
Az osztály destruktora

void run()
Ez a függvény futtatja az algoritmust.

Lekérdező függvények:

Ezeket a függvényeket a run() meghívása után lehet használni.

Node terminal(int i)
Az algoritmus megváltoztatja a fogyasztók sorrendjét. Ezzel a függvénnyel ezt a belső sorrendet lehet lekérdezni. Az uPath() függvény helyes használatához szükséges.

Value demand(int i)
Az igények belső sorrendjének lekérdezésére való függvény. A terminal(k) fogyasztóhoz tartozó igény a demand(k)

FlowMap flowMap()
Az eredmény folyamot adja vissza.

Value flow(Edge edge)
Az edge élen adja vissza az eredmény folyam értékét.

Path uPath(int i)
Az iedik fogyasztóhoz tartozó utat adja vissza. Ebben az esetben ez a terminal(i).

C.2.3. Példa:

```
#include <iostream>
#include <lemon/list_graph.h>
#include "Unsplittable/flow_to_unsplittable_flow.h"
#include <vector>

int main()
{
  typedef lemon::ListGraph Graph;
  typedef Graph::EdgeMap<int> EdgeMap;
  typedef Graph::Node Node;
  typedef Graph::Edge Edge;
  using lemon::INVALID;

//létrehozzuk a gráfot
  Graph g;
  Node v1=g.addNode();
  Node v2=g.addNode();
  Node v3=g.addNode();
  Node v4=g.addNode();
  Edge v1_v2=g.addEdge(v1, v2);
  Edge v1_v3=g.addEdge(v1, v3);
  Edge v2_v3=g.addEdge(v2, v3);
  Edge v2_v4=g.addEdge(v2, v4);
  Edge v3_v4=g.addEdge(v3, v4);
//beállítjuk a termelőt, a fogyasztókat és az igényeket
  Node s=v1;
  Node t1=v3;
```

```
  Node t2=v4;
  std::vector<Node> t;
  std::vector<int> d;
  t.push_back(t1);
  t.push_back(t2);
  d.push_back(7);
  d.push_back(4);
//megadunk egy folyamot, ami kielégíti az igényeket
  EdgeMap flow(g);
  flow[v1_v2]=8;
  flow[v1_v3]=3;
  flow[v2_v3]=6;
  flow[v2_v4]=2;
  flow[v3_v4]=2;
//futtatjuk az algoritmust
  lemon::FlowToUnsplittableFlow<Graph,int> ftu(g,flow,s,t,d);
  ftu.run();
//lekérdezzük az eredményt
  std::cout << "A v1_v3 elen a folyam erteke:"<<ftu.flow(v1_v3);
}
```

D. Fejlesztői Dokumentáció

D.1. Implementáció

D.1.1. DGGAlgoritmus

A DGGAlgoritmus implementációja meglehetősen egyértelmű. Az implementáció a [15] cikk alapján készült és az egyes részek jól megfeleltethetők a cikkben leírt lépéseknek. Az algoritmus a cikkben leírtak szerint figyelmen kívül hagyja azokat az éleket, amin a folyam értéke 0.

```
void run()
```

```
{
  preliminaryPhase(); //bevezető fázis
  int i=0;
  while(i<t.size()){
    if(t[i]==s) {
      ++i;
     } else {
        updateSingularity(); //szinguláris csúcsok megjelölése
        Path path=findCycle(); //alternáló kör keresése
        augumentFlow(path); //alternáló kör módosítása
        moveTerminals(); //fogyasztók mozgatása
        i=0;
      }
   }
}
```

D.1.2. SA

Skutella algoritmusának implementációja az [14] cikk alapján történt, de egy apróságban eltér attól.

Az algoritmus miután elvégezte az igények kerekítését nem a legköltségesebb utak mentén csökkenti a folyam értékét, hanem egyszerűen egy minimális költségű folyamot keres az új igényekkel, olyan módon, hogy a költségeket változatlanul hagyja, a kapacitásokat pedig az eredeti folyamértékekkel teszi egyenlővé.

Ez a verzió nagyon hasonló a Skutella által a cikk futási idővel foglalkozó fejezetében leírt változathoz. Az algoritmus figyelmen kívül hagyja azokat az éleket, melyeken a folyam értéke 0. Az algoritmus egy olyan részgráfon dolgozik, aminek csak azok az élek képezik részét, melyekre igaz, hogy a rajtuk folyó folyam 0-nál nagyobb.

Ezt a Lemon EdgeSubGraphAdaptor osztálya és a filter_map.h-ban található FilterMap osztály segítségével valósítja meg. Mikor az algoritmus a folyamot δ_j- egésszé alakítja, szintén egy hasonló részgráfon dolgozik. Ehhez a Lemon EdgeSubGraphAdaptor osztályát és az integral_filter_map.h-ban található IntegralFilterMap osztályt használja.

```
template<typename Graph,typename Value>
class IntegralFilterMap: public MapBase<typename Graph::Edge, bool>{

  typedef typename Graph::template EdgeMap<Value> FlowMap;
  typedef typename Graph::Edge Edge;
  const FlowMap& _flow;
  Value _dj;

public:

  IntegralFilterMap(const FlowMap& fm, Value dj) :
  _flow(fm), _dj(dj){}
  bool operator[](const Edge& key){
    return _flow[key]%(int)_dj != 0;
  }
};
```

D.2. Tesztelés

D.2.1. A tesztadatok előállítása

A tesztadatokat több lépcsőben, több kisebb program segítségével állítjuk elő. Első lépésként létrehozunk egy generálási listát. Majd egy netgen nevű programmal ennek a listának a segítségével létrehozzuk a kívánt gráfokat. Egy fájlban egy gráfot tárolunk. A netgen által generált gráfok adott számú csúccsal és éllel rendelkeznek. Megadhatjuk továbbá a fogyasztók számát, az igények összegét, a minimális és maximális kapacitásokat illetve költségeket az éleken.

A netgen ezeket az értékeket garantáltan úgy generálja, hogy létezzen osztott folyam, ami az igényeket kielégíti és a kapacitásokat nem sérti.

A generált fájlok elnevezésekor a következőképpen jártam el:
A fájl elnevezése utal a gráf paramétereire, csúcsokszáma_élekszáma_fogyasztókszáma_sorszám
Például: 1000_2000_300_2

Amennyiben további módosításokra lenne szükség, például a kapacitások vagy a költségek megváltoztatására, azt újabb programok segítségével elvégezzük.

D.3. A tesztek eredményei

A tesztek során azt vizsgáltam, hogy az algoritmusok által talált osztatlan folyamok mennyivel lépik túl a kapacitásokat. Ehhez két különböző képletet használtam:

1. $max\frac{f(e)}{c(e)}$
2. $max\frac{f(e)-c(e)}{d_{max}}$

Valamint néhány esetben vizsgáltam a futási időt is. Az algoritmusok bemenete minden esetben egy minimális költségű osztott folyam volt az adott gráfon. Ezeket a folyamokat a teszteléshez használt függvények állítják elő, amik a flow_tests.h fájlban találhatók.

D.3.1. Általános eset

A generáláshoz használt fájlok:

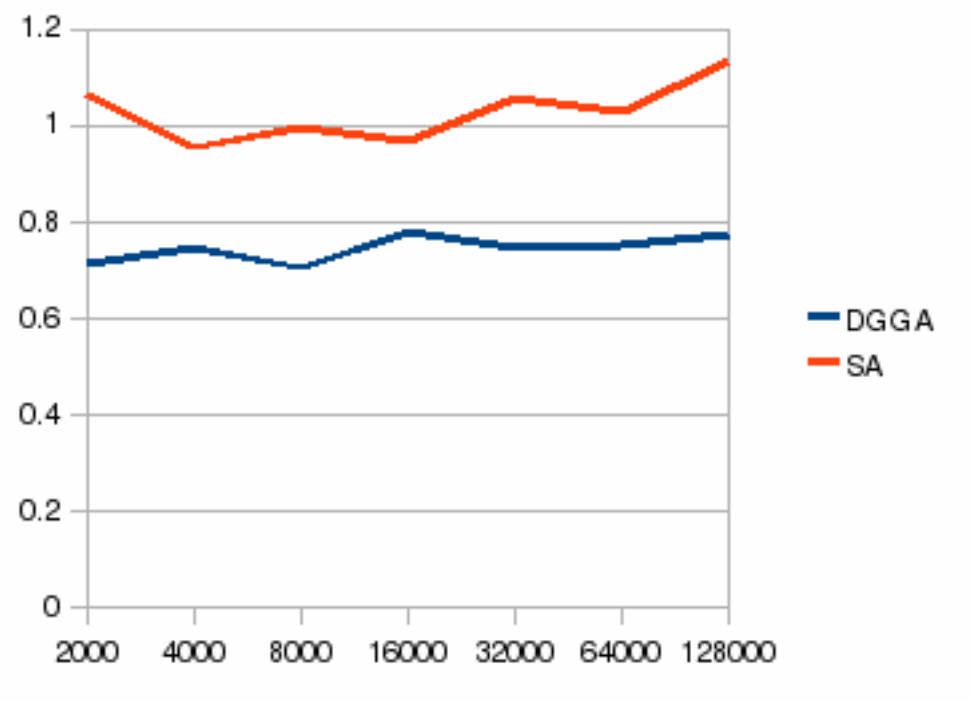
- testgen3 (testgen3.cc)
- netgen
- gengraphs_simple (gengraphs_simple.cc, generate_gr.h)

A generált gráfok tulajdonságai:

csúcsok száma:1000
élek száma: 2000-től 128000-ig
fogyasztók száma: 300

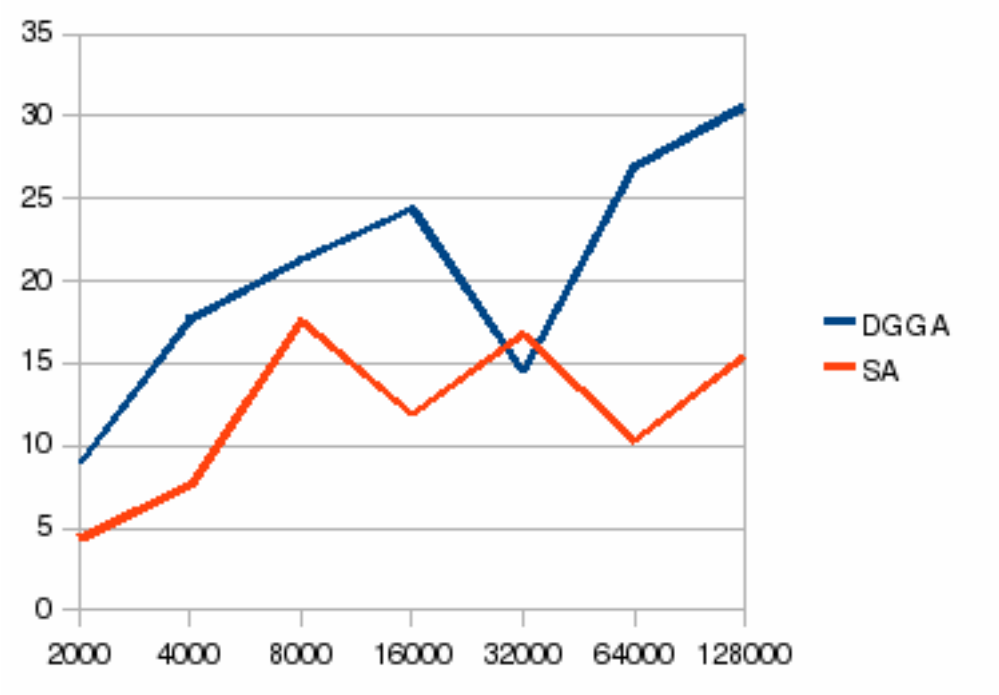
A netgen által generált költségeket újra generáltuk, valamint a kapacitásokat úgy módosítottuk, hogy a kapacitásokat véletlenszerűen kiválasztott utakon megemeltük, olyan

módon, hogy minden fogyasztóhoz vezessen legalább egy olyan $Q_i$út, amin a fogyasztóhoz tartozó igényt el lehet vezetni. Azaz, $\forall e \in Q_i : c_j(e) := max(d_i, c(e))$. A következő grafikon $max\frac{f(e)-c(e)}{d_{max}}$ értékét mutatja növekvő élszámú, a fenti tulajdonságokkal rendelkező gráfokon.



Nem lehet semmilyen egyértelmű összefüggést felfedezni az élek és a csúcsok arányával kapcsolatosan. Megfigyelhető viszont, hogy a DGGAlgoritmus kevésbé lépi túl a kapacitásokat, ami várható is az algoritmusokra vonatkozó tételek figyelembe vételével. Ettől lényegesen eltérő adatot a későbbi mérések során sem találtam , bár a különbségek mértéke feladattípusonként változó.

Az azonos gráfokon mért $max\frac{f(e)}{c(e)}$ értékek épp ellenkezőek, Skutella algoritmusa 32000 élszámú gráfokra valamivel rosszabbul teljesít, de a többi esetben jelentősen jobban. Az értékek nagyon érzékenyek a kapacitások és az igények arányára, amit a következő teszt adatai is alátámasztanak.



D.3.2. Garantáltan létező megoldás

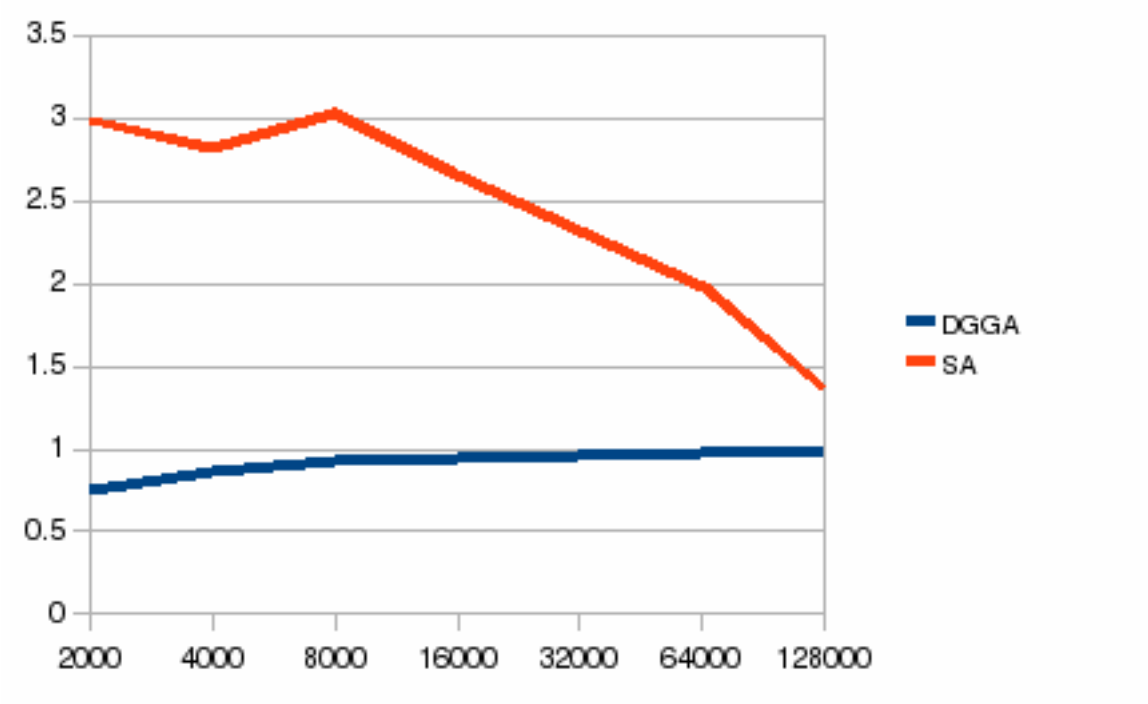
A generáláshoz használt fájlok:

- testgen3 (testgen3.cc)
- netgen
- gengraphs_simple (gengraphs_simple.cc, generate_gr.h)

A generált gráfok tulajdonságai:
csúcsok száma:1000
élek száma: 2000-től 128000-ig
fogyasztók száma: 300

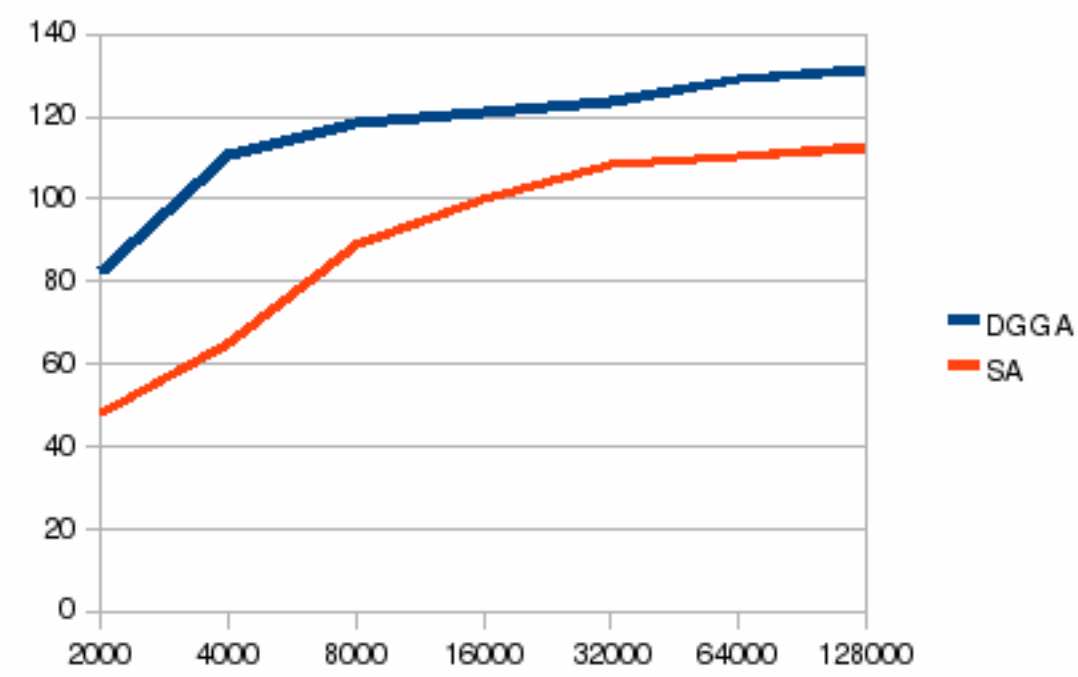
A netgen által generált költségeket újra generáltuk, valamint a kapacitásokat úgy módosítottuk, hogy kezdetben az élekhez véletlenszerű, alacsony kapacitásokat rendeltünk, majd a kapacitásokat véletlenszerűen kiválasztott utakon megemeltük, olyan módon, hogy a P_i út mentén a kapacitásokhoz d_i-t adtunk. Így biztosan van olyan osztatlan folyam, ami nem sérti a kapacitásokat.

A következő grafikon $max\frac{f(e)-c(e)}{d_{max}}$ értékét mutatja növekvő élszámú, a fenti tulajdonságokkal rendelkező gráfokon.



Megfigyelhető, hogy míg a DGGAlgoritmus élszámtól függetlenül az általános esethez hasonlóan teljesít, addig Skutella algoritmusa kis élszámú gráfokra jóval rosszabb teljesítményt mutat és csak nagy élszámokra éri el a közel azonos szintet.

A $max\frac{f(e)}{c(e)}$ értékek esetében viszont pont ellenkezőleg, Skutella algoritmusa végig jobban teljesít. Itt még inkább látható, hogy a $max\frac{f(e)}{c(e)}$ értékek milyen érzékenyek a kapacitások és igények arányára. Mivel az igényeket azonos módon generáltuk, mint az általános esetben, míg a kapacitásokat csak azokon az éleken emeltük meg ahol ez a megoldás létezéséhez szükséges volt, sok alacsony kapacitású él van a gráfokban. A $max\frac{f(e)}{c(e)}$ értékek ezért sokkal nagyobbak, mint az általános esetben.



D.3.3. A maximális és a minimális igény aránya

A generáláshoz használt fájlok:

- testgen3 (testgen3.cc)
- netgen
- gengraphs_dmax (gengraphs_dmax.cc, generate_gr.h)

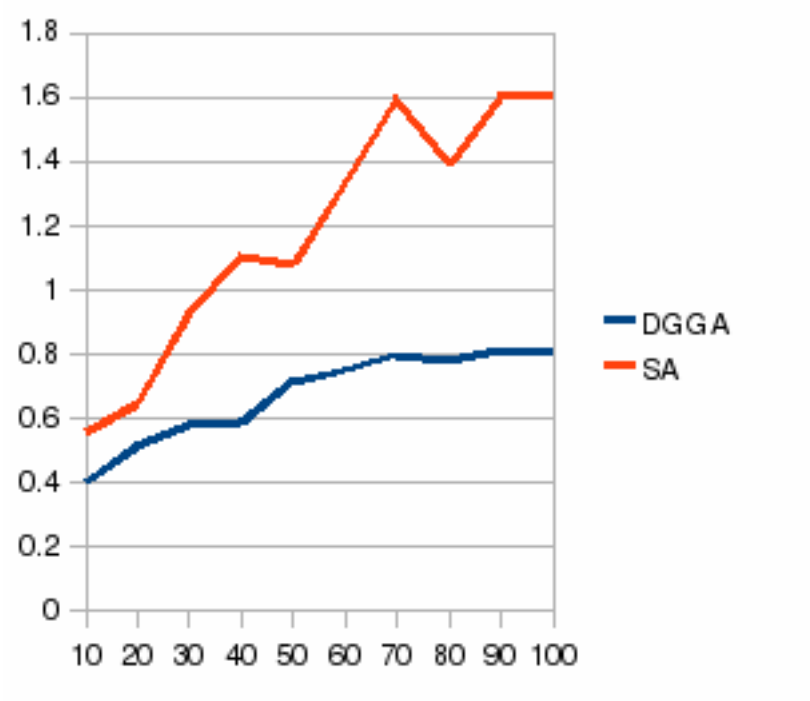
A generált gráfok tulajdonságai:
csúcsok száma:1000
élek száma: 4000
fogyasztók száma: 300

A netgen által generált költségeket újra generáltuk, valamint a kapacitásokat úgy módosítottuk, hogy kezdetben az élekhez véletlenszerű, alacsony kapacitásokat rendeltünk. Az igényeket úgy generáltuk újra, hogy a minimum igény 1 legyen a maximum igény pedig egy adott érték. A többi igényt véletlenszerűen határoztuk meg e két érték közöttire. Ezután a kapacitásokat véletlenszerűen kiválasztott utakon megemeltük, olyan módon,

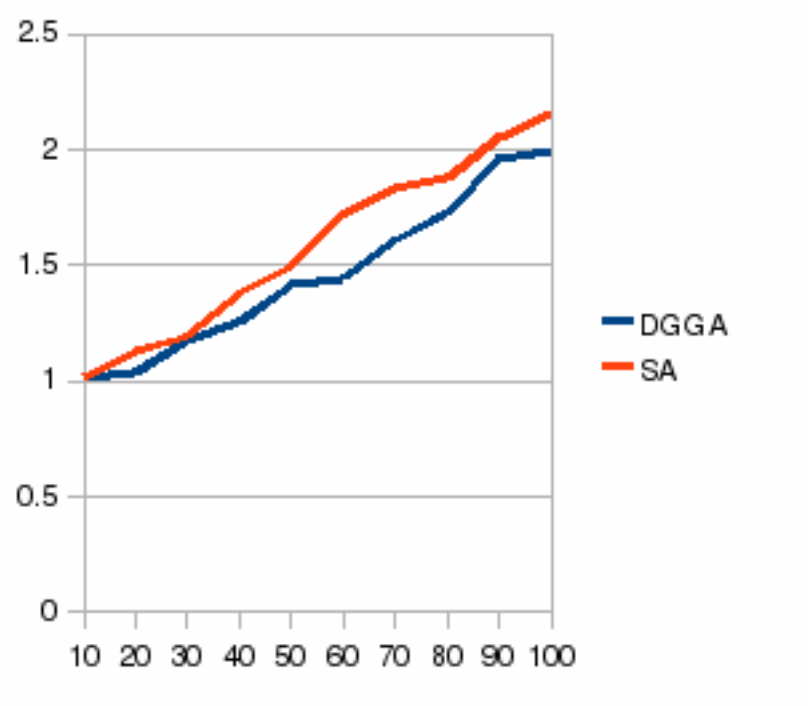
hogy minden fogyasztóhoz vezessen legalább egy olyan út, amin a fogyasztóhoz tartozó igényt el lehet vezetni. Azaz, $\forall e \in Q_i : c_j(e) := max(d_i, c(e))$.
Végül létrehoztunk egy t_s szuper fogyasztót (hozzáadtunk egy csúcsot a gráfhoz) és összekötöttük a fogyasztókkal úgy, hogy $t_i$fogyasztóból t_s-be egy él vezessen pontosan d_i kapacitással.

Ezután meghatároztuk a maximális folyamot s-ből t_s be, és amíg ez nem érte el $\sum d_i$-t addig iteratívan növeltük a kapacitásokat oly módon, hogy minden kapacitást megszoroztunk$\sum \frac{d_i}{f_{val}}$-al, ahol f_{val} a maximális folyam értéke.

A következő grafikon $max\frac{f(e)-c(e)}{d_{max}}$ értékét mutatja a fenti tulajdonságokkal generált gráfokon, növekvő $\frac{d_{max}}{d_{min}}$ arány mellett.



Az algoritmus láthatóan rosszabbul teljesít, ha a $\frac{d_{max}}{d_{min}}$ arány nő, bár a romlás nem egyenletes. Hasonló változás figyelhető meg a $max\frac{f(e)}{c(e)}$ értékeket figyelve. Látható, hogy a DGGAlgoritmus itt valamivel jobban teljesít, ami valószínűleg annak köszönhető, hogy a generálás iterációi során minden élen növeltük a kapacitásokat.



Általában megállapítható, hogy mindkét algoritmus rosszabbul teljesít olyan feladatokon, melyekben $\frac{d_{max}}{d_{min}}$ értéke nagyobb.

D.3.4. A költségek meghatározása

A generáláshoz használt fájlok:

- testgen3 (testgen3.cc)
- netgen
- gengraphs_simple (gengraphs_simple.cc, generate_gr.h)
- generate_costs (generate_costs.cc, generate_gr.h)

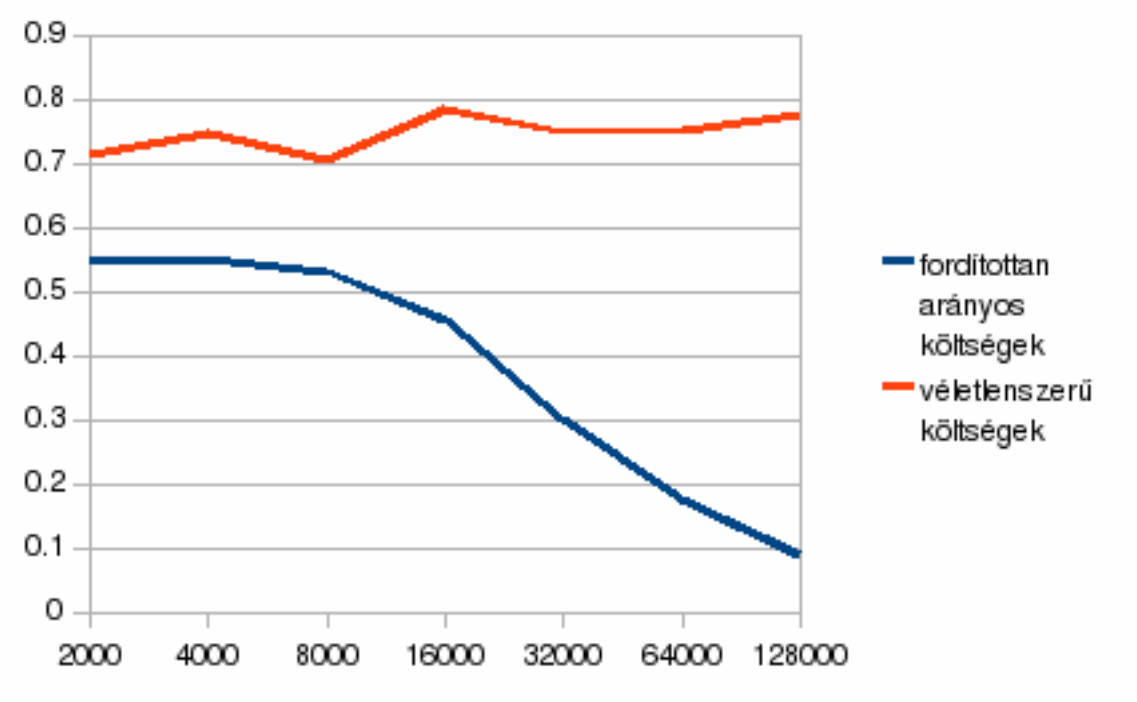
A generált gráfok tulajdonságai:
csúcsok száma:1000
élek száma: 2000-től 128000-ig
fogyasztók száma: 300

A gráfok megegyeznek az általános esetben generált gráfokkal, azzal a különbséggel, hogy a költségeket nem a kapacitásoktól függetlenül állapítjuk meg, hanem azzal egyenesen, illetve fordítottan arányosan.

A következő grafikon $max\frac{f(e)-c(e)}{d_{max}}$ értékét mutatja növekvő élszámú, a fenti tulajdonságokkal rendelkező gráfokon, valamint az általános esetben.

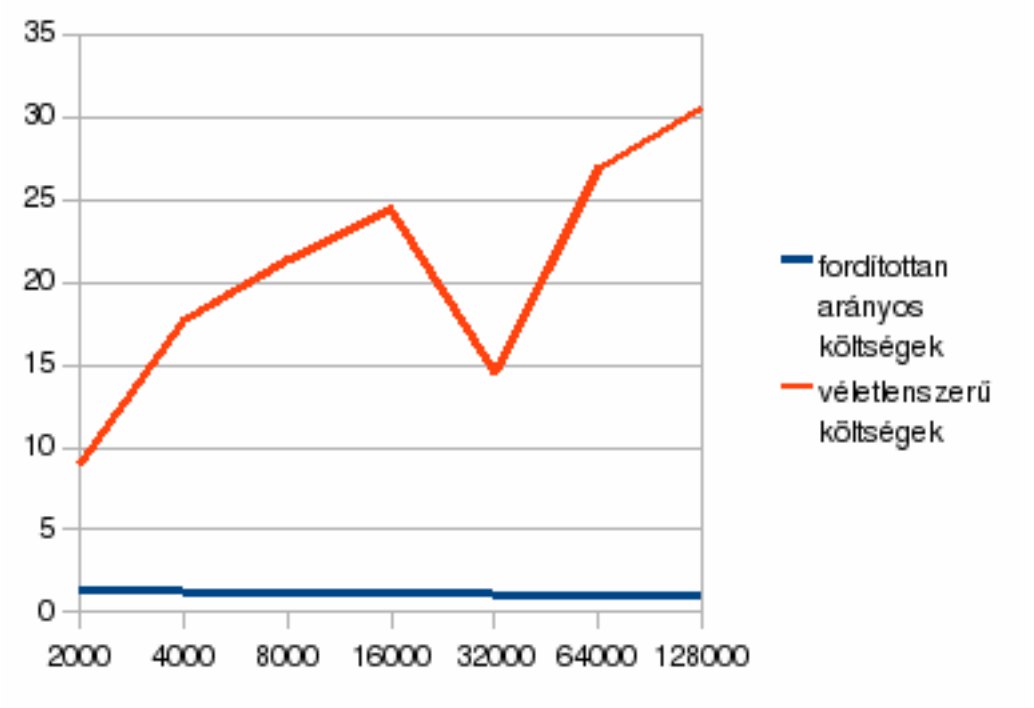
A fordítottan arányos költségeket a következőképpen határoztuk meg: költség(e)=$\frac{30000}{c(e)}$.

A DGGAlgoritmus:



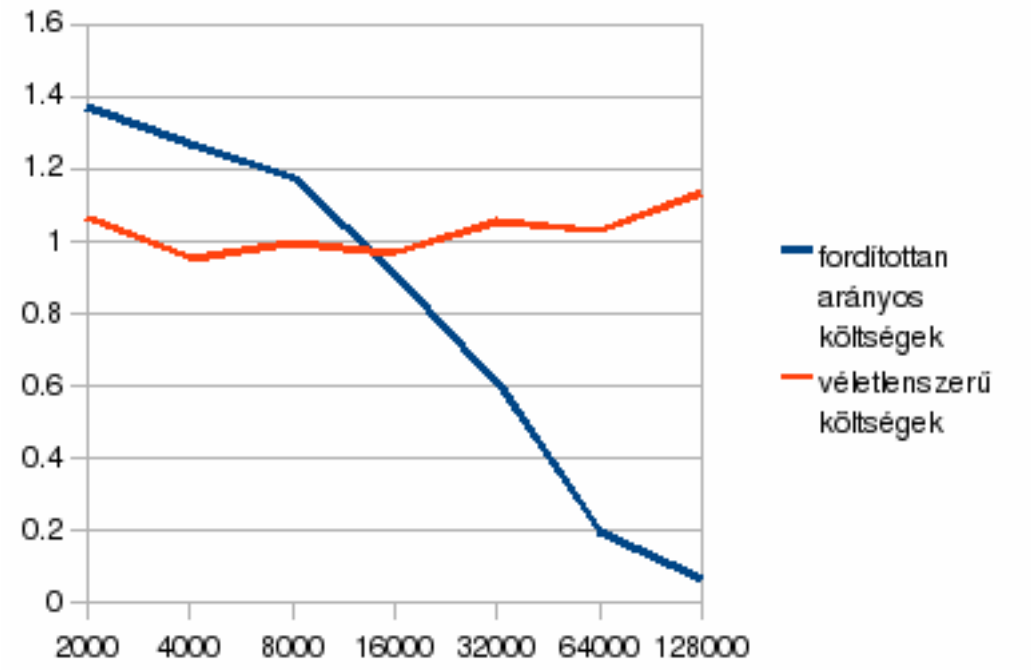
Látható, hogy A DGGAlgoritmus sokkal jobban teljesít, mint az általános esetben, pedig az algoritmus maga nem számol a költségekkel, így azok csak a kezdeti folyam megválasztásában játszanak szerepet.

DGGAlgoritmus, $max\frac{f(e)}{c(e)}$ értéke növekvő élszámú, a fenti tulajdonságokkal rendelkező gráfokon, valamint az általános esetben.:



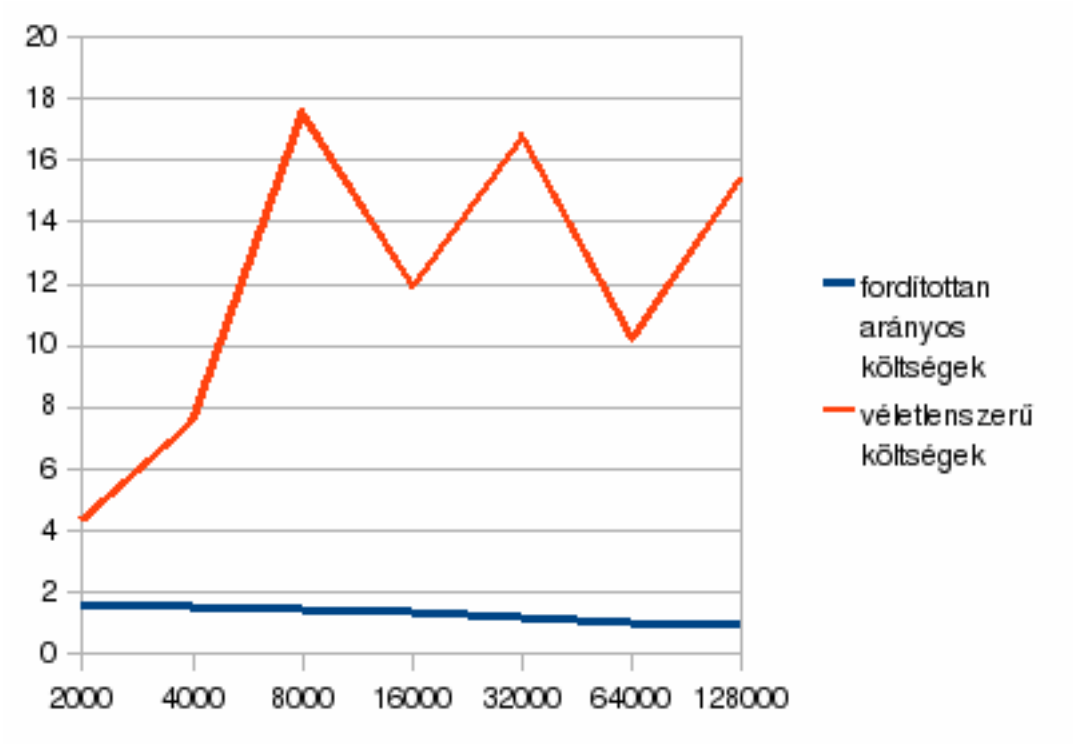
Itt is látszik, hogy az algoritmus jobban teljesít, ráadásul mindkét esetben az élek számának növelése esetén az algoritmus teljesítménye nem hogy romlana, hanem épp ellenkezőleg egyre javul.

Skutella algoritmusa, $max\frac{f(e)-c(e)}{d_{max}}$ értéke növekvő élszámú gráfokon:



Bár azt várnánk, hogy mivel az algoritmus használja a költségeket az osztatlan folyam kiszámításához, sokkal jobb teljesítményt nyújtson mint általános esetben, ez nem így van. Kis élszámú gráfokra a teljesítménye rosszabb mint az általános esetben, bár gyorsan javul, és nagy élszámokra közel azonos a DGGalgoritmuséval. (128000 élű gráfokra valamivel jobb is)

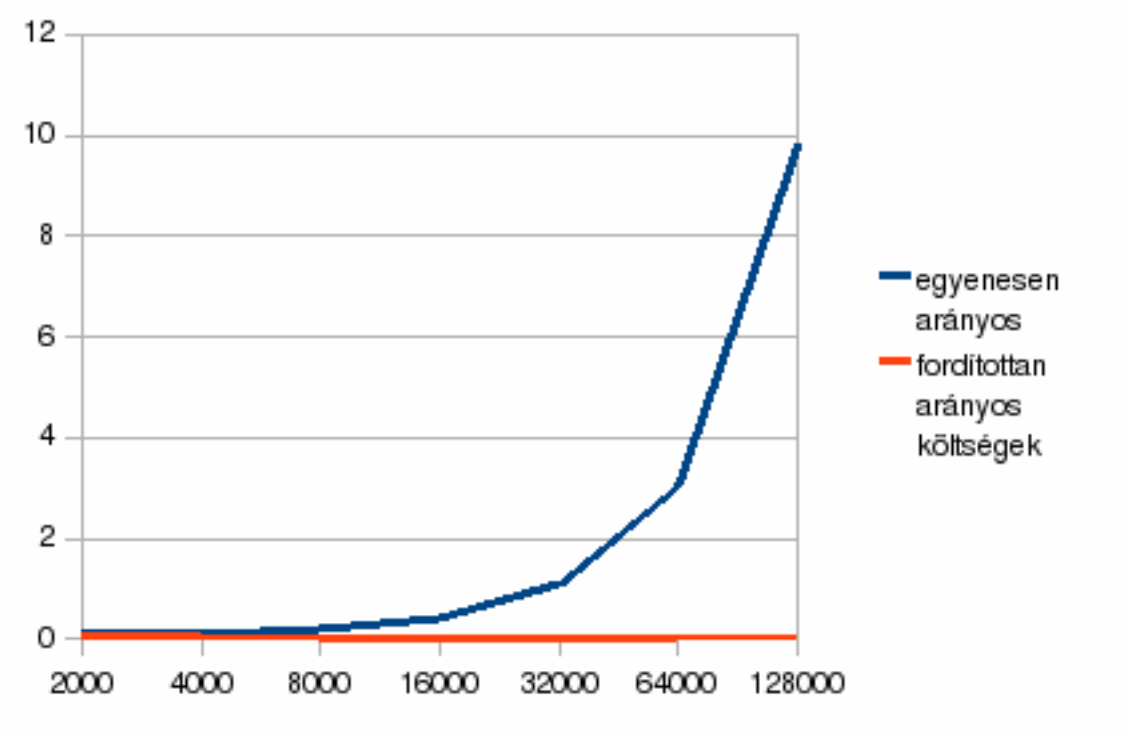
Skutella algoritmusa, $max\frac{f(e)}{c(e)}$ értéke növekvő élszámú gráfokon:



Az algoritmus e szerint a mérték szerint jobban teljesít mint az általános esetben. Nagy élszámokra ez a teljesítmény megközelíti a DGGAlgoritmusét, de nem éri el azt.

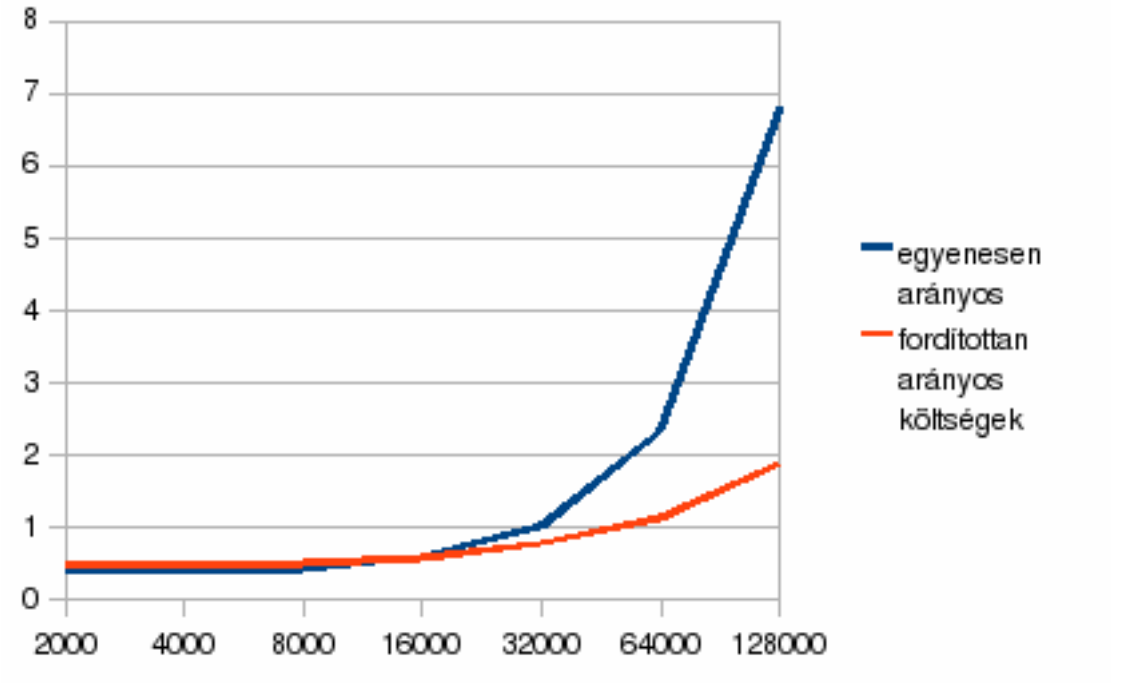
A költségek meghatározásának várhatóan a futási időre is hatással kell lenniük. A következő grafikonokon az egyes algoritmusok a Lemon Timer osztályának realTime() függvényével mért futási idejét hasonlítjuk össze, úgy, hogy a költségeket egyszer a kapacitásokkal azonosnak másszor pedig az előzőleg is használt költség(e)=$\frac{30000}{c(e)}$ képlettel határozzuk meg.

DGGAlgoritmus futási ideje növekvő élszámú gráfokon:



Látható, hogy a futási idő a vártnak megfelelően alakul. Míg egyenes arányos esetben nagy élszámokra a futási idő gyorsan nő, fordítottan arányos esetben végig alacsony marad.

Skutella algoritmusának futási ideje növekvő élszámú gráfokon:



A hatás nem olyan egyértelmű ennél az algoritmusnál. Magas élszámokra a fordítottan arányos esetben az algoritmus láthatóan gyorsabb, míg alacsonyabb élszámokra kicsivel lassabb.

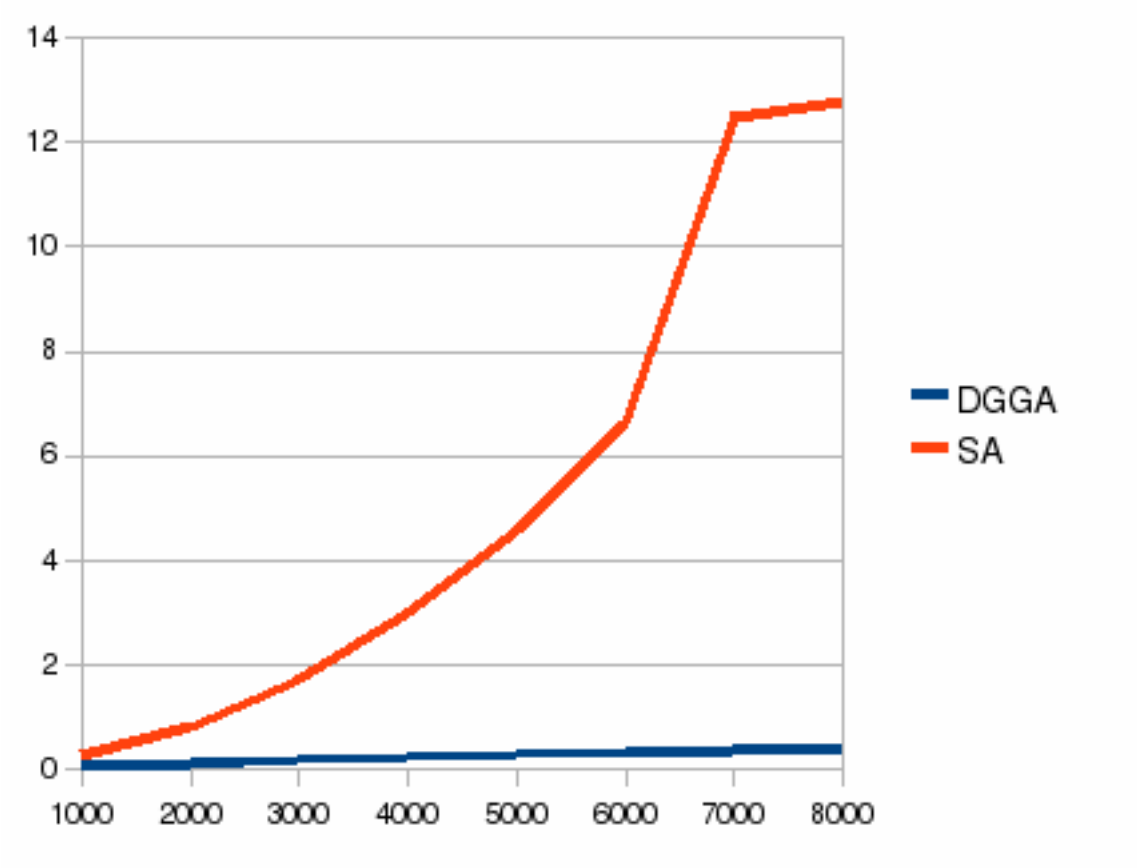
D.3.5. A futási idők összehasonlítása

A generáláshoz használt fájlok:

- testgen4 (testgen4.cc)
- netgen

A generált gráfok tulajdonságai:
csúcsok száma:1000-től 8000-ig
élek száma: 4000-től 32000-ig
fogyasztók száma: 200-tól 1600-ig

A generált gráfoknál a csúcsok és élek számának aránya, valamint a csúcsok és fogyasztók számának aránya nem változik. A következő grafikonokon a két algoritmus futási idejét hasonlíthatjuk össze, a fenti tulajdonságú, növekvő csúcsszámú gráfokon.



Látható, hogy a DGGAlgoritmus sokkal jobban teljesít, a különbség a probléma méretének növekedésével gyorsan nő. Ennek oka valószínűleg az egyre több különböző méretű igény.

Míg a DGGAlgoritmus futási idejét nem befolyásolja az egymástól különböző igények száma, pontosabban a futási idő nem változik, még akkor sem ha minden igény egyforma, addig Skutella algoritmusában az iterációk számát leginkább ez határozza meg. Például abban az esetben, ha minden igény egyforma, elég egyetlen iteráció.

D.3.6. Fogyasztók száma

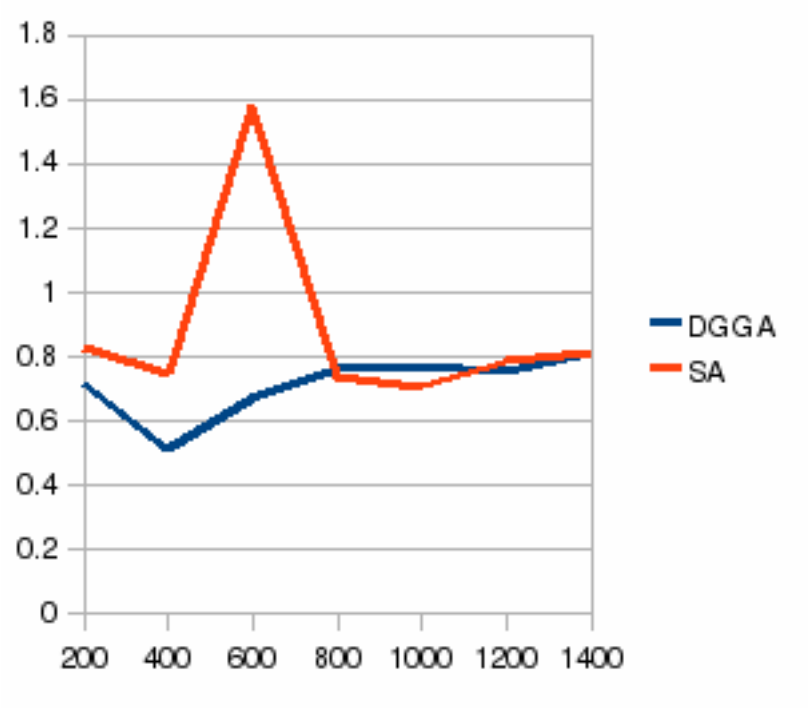
A generáláshoz használt fájlok:

- testgen5 (testgen5.cc)
- netgen
- gengraphs_terminals (gengraphs_terminals.cc, generate_gr.h)

A generált gráfok tulajdonságai:
csúcsok száma:1000
élek száma: 4000
fogyasztók száma: 200-tól 1400-ig

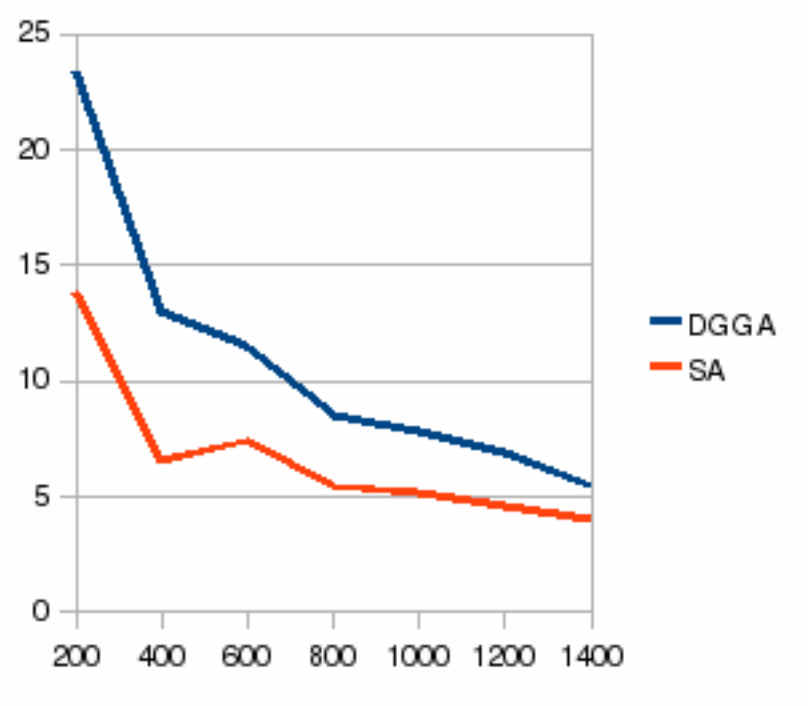
Mivel a netgen nem tud egy csúcsba több fogyasztót generálni, ezért szükség volt a netgen által generált fogyasztók szétválasztására, több fogyasztóvá.

A következő grafikon 1000 csúcsú, 4000 élű gráfokon mutatja $max\frac{f(e)-c(e)}{d_{max}}$ értékét, növekvő fogyasztó szám mellett.



Néhány kiugró esettől eltekintve, a fogyasztók száma nem befolyásolja lényegesen a $max\frac{f(e)-c(e)}{d_{max}}$ értéket.

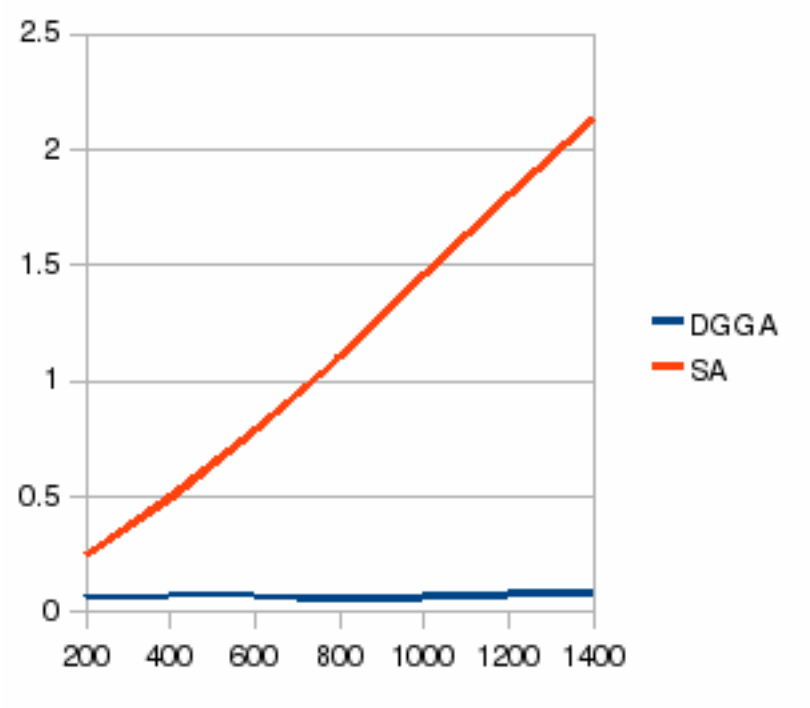
Azonos gráfokon $max\frac{f(e)}{c(e)}$ értéke a fogyasztók számának növekedésével csökken.



Ez egyezik a várakozásainkkal, mivel az igények összege nem változott, így a fogyasztók számának növekedésével d_{max} csökken, ahogy $max\frac{f(e)}{c(e)}$ is.

Megfigyelhető, hogy Skutella algoritmusa valamivel jobban teljesít, de a különbség a fogyasztók számának növekedésével csökken.

A futási időket tekintve sem ér minket meglepetés, Skutella algoritmusa növekvő fogyasztó szám mellett gyorsan lassul, míg a DGGAlgoritmus futási idejét a fogyasztók számának növekedése sokkal kevésbé befolyásolja.



E. Konklúzió

Az algoritmusokra vonatkozó tételek állításaival összhangban, a DGGAloritmus kevésbé sérti a kapacitásokat mint Skutella algoritmusa, ha a $max\frac{f(e)-c(e)}{d_{max}}$ értéket vizsgáljuk. $max\frac{f(e)}{c(e)}$ szerint viszont Skutella algoritmusa gyakran jobban teljesít, tehát olyan esetekben, mikor a kapacitások d_{max}- hoz képest kicsik, érdemesebb lehet ezt használni.

Mindkét algoritmus nehezebben kezeli azokat a feladatokat, mikor $\frac{d_{max}}{d_{min}}$ nagyobb.

Megfelelő költségek megállapításával jelentően javítani lehet mind a teljesítményt, mid a futási időt. Ez a DGGAlgoritmus esetében a kiindulási folyam jobb megválasztását jelenti, és a nagyobb kapacitású élek preferálását. Az ilyen gráfokon a DGGAlgoritmus bizonyult hatékonyabbnak.

A továbbiakban érdekes lehet megvizsgálni, hogy más, bonyolultabb költségfüggvénnyel, esetleg dinamikusan változó költségekkel nem-e lehet jobb teljesítményt elérni.

Külön érdekes lehet, hogy megfelelő költségek meghatározásával elérhető-e, hogy Skutella algoritmusa a DGGAlgoritmusnál jobban teljesítsen.

A futási időt vizsgálva elmondható, hogy általános esetben a DGGAlgoritmus hatékonyabb, de érdekes lehet megvizsgálni azokat a speciális eseteket, amikor minden igény egyforma, vagy csak néhány különböző méretű igény van.

Hivatkozások

[1] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Sov. Math. Dokl.*, 11:1277–1280, 1970.

[2] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[3] A. V. Goldberg. Efficient graph algorithms for sequential and parallel computers. Technical report, Cambridge, MA, USA, 1987.

[4] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[5] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[6] Király Zoltán Magyar Gábor és Szentesi Áron Józsa Balázs Gábor. On the single-source unsplittable flow problem. *Optimization and Engineering*, 2:321–347, 2001.

[7] A. Karzanov. Determining the maximal flow in a network by the method of preflows. *Sov. Math. Dokl.*, 15:434–437, 1974.

[8] J.M. Kleinberg. Single-source unsplittable flow. *Proceedings of the 41th Annual IEEE Symposium on Foundations of Computer Science*, 45:68–77, 1996.

[9] Stavros G. Kolliopoulos and C. Stein. Approximation algorithms for single-source unsplittable flow. *SIAM Journal on Computing*, 2002.

[10] D. R. Fulkerson L. R. Ford. Flows in networks. *Princeton University Press*, 1962.

[11] T.V. Lakshman M. Kodialam. Minimum interference routing with applications to mpls trafficengineering. *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2:884–893, 2000.

[12] P. Gajowniczek M. Pioro. Simulated allocation: a suboptimal solution to the multi-commodityflow problem. *11th UK Teletraffic Symposium*, 1994.

[13] S. Plotkin. Competitive routing of virtual circuits in atm networks. *IEEE Journal on Selected Areas in Communications*, 13:1128–1136, 1995.

[14] Martin Skutella. Approximating the singlesource unsplittable mincost flow problem. *Proceedings of the 41th Annual IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2000.

[15] Michel X. Goemans Yefim Dinitz, Naveen Garg. On the singlesource unsplittable flow problem. *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 290–299, November 1998.