Gyors párosítási algoritmusok

Szakdolgozat

Írta: Lócsiné Vándor Zsófia

Témavezető:

Király Tamás

Operációkutatási Tanszék

Természettudományi Kar

Eötvös Loránd Tudományegyetem

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozat szerkezete: Egy rövid bevezető után egy-két szó elhangzik arról, miért is van szükség gyors párosítási algoritmusokra, milyen eszközökkel dolgozunk majd, illetve hogy a programozás során hogyan sikerült a véletlent kezelni. Az ezt követő fejezetekben a gyors párosítási algoritmusok közül kettő részletes elemzése és implementációjának taglalása következik – 2. és 3. fejezet. A szakdolgozatban az önálló eredményt ezek megvalósítása jelenti. Végezetül egyéb gyors párosítással szolgáló algoritmusok rövid áttekintését tűztem ki célul a 4. fejezetet szánva erre. Ezek már egy fokkal általánosabbak, maximális súlyú párosításokat adnak.

1.1. Motiváció

A számítástechnika fejlődése, a rendszeroptimalizálások igénye számos algoritmus és annak részét képző szubrutin fejlesztését és gyorsítását hívta életre. Ezen optimalizáló faktorok közé sorolhatóak az ütemezéselméleti kérdések, és azok részeként, a párosítási problémák. Egy példa:

Képzeljük el, hogy céget vezetünk és úgy döntünk, hogy a cégünkön belül elvégzendő rutinmunka egy részét kiosztjuk diákmunkásoknak. A HR-esünk tanácsára *2* hallgatót be is hívunk az irodába, és hamar ki is derül róluk, hogy mindketten *pontosan ugyanolyan* jóképességű gyerekek, akik alkalmasak lesznek a feladatra. Órabérben fognak nekünk dolgozni, és addig tartjuk meg őket, amíg el nem végzik a kiszabott munkát. Tehát szeretnénk, ha *minél hamarabb* teljesítenék, amit kell. (Természetesen biztosítjuk a feltételeit annak, hogy nekik is érdekük legyen jól dolgozni. Egy ötlet: Hogy motiváltak legyenek minél előbb végezni, kijelölhetünk egy magas összegről induló prémiumot, ami az idő előrehaladtával $f(x) = 1/x$-hez hasonló függvény szerint csökken.)

Mivel igyekszünk minél kisebb befektetéssel minél nagyobb hasznot hajtani, aján-

lott megterveznünk a foglalkoztatásukat. Célunk hogy amíg nálunk vannak, *minél optimálisabban osszuk ki a rájuk szabott feladatokat* – lehetőleg ne legyen olyan, hogy csak az egyikük tud ténykedni, mert meg kell várja a másikat. Készítünk egy folyamatábrát. Az elvégzendő munkákat *óránként* teljesíthető szakaszokra bontjuk és azokat az elemeket, amik *egymásutánisága* megfellebbezhetetlen, nyíllal kötjük össze. A különböző munkák utolsó és első elemei között is nyíllal jelöljük a függőségi viszonyt. Nyilván azokon a munkákon tudunk majd időt nyerni, amik nincsenek összekapcsolva, vagyis *párhuzamosan* végezhetőek. Innentől jó lenne egy program, ami az így nyert információkat betáplálva megmutatja, mit mikor kell kiosztani. És van ilyen.

Talán az olvasó még nem is sejti, de el is jutottunk egy algoritmussal megoldható ütemezési problémához, konkrétan a $P2|prec, p_j = 1|C_{max}$ feladathoz. (Az absztrakció mögött csupán ennyi rejlik: 2 ugyanolyan gépen, bizonyos precedenciafeltételeket betartva, időegységben mérhető munkákat szeretnénk elvégezni úgy, hogy az a lehető legkevesebb időbe kerüljön.) A megoldásban ez a feladat felhasználja a párosítás fogalmát, mégpedig a függetlenül végezhető munkák pontjain. Ezzel tisztáztuk, mire lehetnek jók a párosítások.

A gyorsaság szükségességét pusztán egy erősen torzított példával tudnám illusztrálni, lévén egy valóságközelibb példa túl bonyolult lenne: Tegyük fel, hogy van egy borzasztóan nagy, borzasztóan bonyolult képletünk, amit $n + 1$ darab processzoron számolhatunk, mert a képletnek vannak olyan részei, amiket egymástól függetlenül is elvégezhetünk. Órajelenként d műveletet tud egy processzor feldolgozni, ez a d művelet legyen az, amiknek az elérése a lehető legkönnyebb az adott processzor számára. Tegyük fel, hogy memóriában csak n művelet fér el, ezt tudjuk csak fixen biztosítani, mert a többi hely kell a már meghívott függvényeknek, eljárásoknak, mert tárhely kell a számítások elvégzéséhez... Tehát a +1. processzort fogjuk felhasználni arra, hogy minden időpillanatban eldöntsük, a részfeladatok közül melyiket melyik processzor végezze el. Cél: az órajelenként előálló páros gráfból minél több párosítást megtalálni egy órajel alatt. Ha olyan programra vágyunk, ami ezt az n processzort maximálisan kihasználja, szükségünk lesz gyors párosító algoritmusra. A példán az is látszik, hogy a „gyors párosítási algoritmusok” problémakör páros gráfokra specializált változata sem érdektelen.

Arra, amikor a kiinduló páros gráf már d-reguláris is, inkább matematikai felhasználást mutatnék: ez a duplán sztochasztikus mátrixokra mond valamit. A *Birkhoff–von Neumann-tétel* mondja ki azt, hogy az ilyenek (ti. amik sor, illetve oszlopösszege 1, elemei nemnegatívak) előállnak egységmátrixok konvex burkaként. Egy ilyen előállítást tudunk megadni d-reguláris páros gráf d darab párosításaként.

1.2. Előzmények

1.2.1. Alapfogalmak

Legyen adott egy páros gráf $G = (X \cup Y; E)$, melyben $X = x_1, x_2, ..., x_n$ és $Y = y_1, y_2, ..., y_n$ jelölés mellett elmondható, hogy $(x_i, x_j) \notin E$ és $(y_i, y_j) \notin E$ semmilyen $i, j \in 0, 1, ..., n$. Erről végig feltesszük, hogy $|X| = |Y| = n$. Amikor azt mondjuk, teljes párosítást keresünk, akkor a gráf E élhalmazának olyan M részhalmazára vagyunk kíváncsiak, amely minden pontot pontosan egyszer fed, elemszáma a pontszám fele.

1.2.2. Jelölések

A dolgozatban bárhol is forduljon elő a G, V, E, M jelölés, ezek sorban mindig adott gráfot, pontok és élek halmazát, illetőleg párosítást fognak jelölni. Az n és m jelöléseket megtartom pontszámnak és élszámnak. Egy élet, ha annak pontjai egy pontosztályból valók (u, v)-vel, ha különböző pontosztályból valók (x, y)-nal fogom jelölni, ahogy végig megtartom az $V = X \cup Y$ felbontást is, de ha nem akarom hangsúlyozni egy él milyen végpontokból áll össze, akkor egyszerűen e-nek nevezem majd.

Egy A mátrix i. sorának j. elemére $A_{i,j}$ formában utalok, a determinánst és rangfügvényt $det(A)$, illetőleg $rang(A)$ szimbólumok jelzik. Mivel mindig egyértelmű lesz hogy melyik mátrixról van szó, ezért a sorokra illetve oszlopokra csak szövegesen („i. sora”), külön jelölés nélkül fogok hivatkozni.

A futási idők összehasonlítására fogjuk használni a következő jelöléseket: O, Θ, ezért ezek definícióját is frissítsük fel: f,g függvények esetén $f = O(g)$, ha létezik olyan $c > 0$ konstans és $n_0 \in \mathbb{Z}, n_0 > 0$ küszöbérték, hogy minden $n > n_0$ esetén $|f(n)| \leq c|g(n)|$; $f = \Theta(g)$, ha léteznek olyan $c_1, c_2 > 0$ konstansok és $n_0 \in \mathbb{Z}$, ahol $n_0 > 0$ küszöbérték, hogy minden $n > n_0$ esetén $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$.

1.2.3. Az implementációról röviden

A kódokat C++-ban írtam, néhány függvényt felhasználva a C++ Libraryből. Console-ban fordítottam a C++11-es szabványnak megfelelően. Ez a szabvány már képes volt a véletlenszámok generálásának azt a formáját értelmezni, amit választottam:

```
unsigned seed = chrono::system_clock::now().time_since_epoch().count();
default_random_engine generator(seed);
uniform_real_distribution<double> distribution(0.0,1.0);
```

Itt az első sor a generátor magját adja, a második a generátort inicializálja a maggal, míg az utolsó definiálja az eloszlást. Eloszlást definiálni csak egyszer lehet, ugyanakkor nekem arra volt szükségem, hogy újra megadhassam azt a halmazt, amelyből véletlen elemet választok. Ezért élve egy klasszikus programozási módszerrel, ahelyett, hogy az $\{1, ..., n\}$ elemekből választanék diszkréten – gondolva arra, hogy n értéke változni fog – inkább az $x \in [0, 1)$ valós eloszlásból kiválasztott véletlen számra vettem az $\lfloor xn \rfloor$ értéket. Azaz a következő kódsort alkalmaztam:

```
veletlenszam = int(distribution(generator)*n);
```

Még egy trükk kellett, hogy egy folyamatosan fogyó halmaz elemeit rendre ki lehessen választani. Mint láthatjuk a fentiekben, a halmaz összefüggőségét meg kell tartsuk. Erre kétféle megvalósítást is alkalmaztam, a körülményektől függött, mikor melyiket. Az egyik során egy pointertömböt állítottam az $\{1, ..., n\}$ elemekre, és a pointertömb indexe volt az, amit valójában generáltam. Így, ha az $\{1, ..., n\}$ elemek valamelyikére nem volt többé szükség, akkor a rámutató pointert az utolsó elemre állítottam, és csökkentettem n-et $n - 1$-re. A másik módszernél nem volt pointertömb, csak az utolsó elem átmásolása a már szükségtelenné vált helyére, valamint az elemszám csökkentése.

1.2.4. Elérhetőség

A programok és a futtatásukhoz tartozó rövid leírás a az alábbi két linkről érhető el:

`http://www.cs.elte.hu/~vazoadt/szakdolgozat/ford-fulk`
`http://www.cs.elte.hu/~vazoadt/szakdolgozat/gauss-elimin`

2. fejezet

Gyors párosítás Ford–Fulkerson algoritmussal

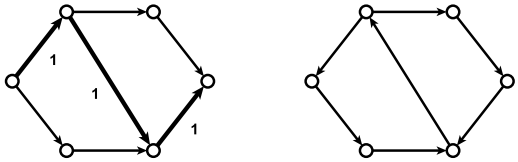
Ez az [1] cikkben szereplő algoritmus egy nagyon egyszerű elgondolás, de csak nagyon speciális gráfokra használható. Csak a reguláris páros gráfokra fut le tényleg gyorsan – azaz, ha a két pontosztályban minden egyes pontnak ugyanannyi szomszédja van. Ilyen gráf esetén, ahogy azt látni fogjuk, mindig van teljes párosítás, úgyhogy a futás is mindig eredményes lesz.

Az algoritmus maga tekinthető úgy, mint az $O(nm)$-es Ford–Fulkerson egy randomizált változata. Amitől ez gyors tud lenni, az az, hogy nem függ a várható lépésszám az élek számától. Vagyis n pont és m él esetén is $O(n \log n)$-es. Ehhez az kell, hogy a gráfban a pontok szomszédjai közül könnyen lehessen random választani. A gráfábrázolást ezért szomszédlistával célszerű megoldani.

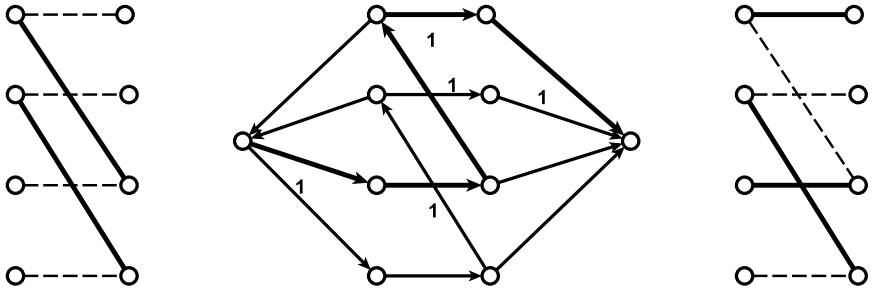
Először is idézzük fel azokat az elméleti eredményeket, amelyekre épít az algoritmus...

2.1. A Ford–Fulkerson algoritmus

Legyen $G = (X \cup Y; E)$, $|X| = |Y| = n$ páros gráf, amiben maximális párosítást keresünk (ha ennek az elemszáma n, akkor ez teljes párosítást fog adni). A problémát maximális folyam keresésére vezetjük vissza: irányítsuk az éleket X-ből Y-ba, és adjunk az irányított gráfhoz még egy s pontot, amiből vezessen él minden $x \in X$ pontba és egy t pontot, amibe pedig vezessen él minden $y \in Y$ pontból. A kapott hálózat élein definiáljuk a kapacitást egységesen 1-nek. Az M párosítás egy éle ebben a reprezentációban 1 folyamértéknek felel meg, amely áthalad a hálózaton s-ből t-be, és az adott élet tartalmazza. Ez az útfolyam kijelöl egy utat a gráfban, ami a folyamhoz tartozó reziduális gráfban fordított irányú útként jelenik meg.



2.1. ábra. A gráfon átmenő folyam hatása a reziduálisban.



2.2. ábra. A párosítás változása a folyam alapján.

Emlékezzünk vissza: a reziduális gráfban mindig azt tartjuk számon, hogy a folyam aktuális állapota mellett merre mennyi folyamértéket küldhetünk még. Amennyiben egy él alsó kapacitása f, a felső g, az élen átáramló mennyiség pedig h, amelyre $f \leq h \leq g$, az a reziduálisban ún. előre élen $g - h$ felső és 0 alsó, illetve az ún. vissza élen $h - f$ felső és 0 alsó kapacitásként jelenik meg. (Itt az előre csak annyit jelent, hogy az eredeti gráf irányításával megegyező irányú, a vissza pedig azt, hogy ezzel ellentétes.) Abban a speciális esetben, ahol a felső kapacitás 1, az alsó pedig 0 és egész mennyiséget kívánunk szállítani (tehát minden élen 0-át, vagy 1-et), elég egy darab él irányításával reprezentálni a reziduálisban, hogy hova mennyit küldhetünk még: amerre mutat, arra lehet egyet, az ellentétes irányba meg nem lehet semmit, ezért azt ki is hagyhatjuk, ahogy azt az 2.1 ábrán látjuk.

A Ford–Fulkerson növelő utakat keres ebben a reziduálisban, valamely útépítő algoritmus felhasználásával, tipikusan BFS-sel. Ennek mentén küld további folyamértéket az eredeti gráfban, majd módosítja a reziduálisbeli kapacitásokat az átküldött folyamérték függvényében – a mi esetünkben tehát megfordítja az éleket. Egy ilyen növelő út a párosításban is növelő utat ad, melyben a vissza élek felelnek meg majd a párosítás éleknek, és az előre élek lesznek azok, amiket törlünk a párosításunkból, lásd 2.2 ábra.

A cikk még egy észrevételt ajánl: a növelő út sose használ olyan élet, ami s és egy a párosított $x \in X$ között fut, mivel az ilyen él s-be mutat, hasonló igaz $y \in Y$ és t

esetén. Ezért az s-be befelé, illetve a t-ből kifelé mutató éleket akár el is hagyhatjuk.

Először is nem árt látni, miért is van egyáltalán egy ilyen gráfban teljes párosítás. Ahhoz, hogy ezt belássuk, használni fogjuk a **Hall-feltételt**. Jelölje $A \subseteq X$-re $N(A)$ az A szomszédjainak halmazát Y-ban. A Hall-feltétel azt mondja ki, hogy $G = (X \cup Y; E)$, $|X| = |Y| = n$ esetén a teljes párosítás létezéséhez szükséges és elegendő feltétel, ha $|N(A)| \geq |A|$ minden $A \subseteq X$ részhalmazra.

1. Tétel. *Ha $G = (X \cup Y; E)$ d-reguláris páros gráf, melyre $|X| = |Y| = n$. akkor $|N(A)| \geq |A|$ minden $A \subseteq X$ esetén.*

Bizonyítás. Mivel A minden pontjának d szomszédja van, $d|A|$ él lép ki A-ból. Nem túl meglepő módon hasonlóan $d|N(A)|$ él lép ki $N(A)$-ból is. A szomszédság definíciójának köszönhetően kapjuk, hogy $d|A| \leq d|N(A)|$. Az egyenlőtlenség két oldalát leosztva d-vel nyerjük az állítást. □

Ez nem csak azt bizonyítja, hogy létezik teljes párosítás, de részét képezi a következő tétel bizonyításának is.

2. Tétel. *Legyen $G = (X \cup Y; E)$ összefüggő d-reguláris gráf és legyen M egy n nagyságúnál kisebb párosítás G-ben. Legyen H a fenti módon származtatott folyam probléma, f az M-et leíró folyam, valamint H_f a reziduális gráf. Ekkor H_f minden pontjából van út t-be.*

Bizonyítás. Jelölje X_B az X és Y_B az Y azon részhalmazát, melyből nem lehet eljutni t-be irányított úton. Mivel minden párosítatlan $y \in Y$-ból van irányított él t-be, ezért az Y_B elemei szükségképpen párosítottak.

Minden $v \in V$-re jelöljük az M párosítás adta párját $M(v)$-vel. Ezzel a jelöléssel élve elmondható, hogy minden $y \in Y$-nak pontosan egy kilépő éle van H_f-ben, mégpedig az, amelyik $M(y)$-ba megy. (Kezdetben ez a kilépő él t-be mutatott, de egy y-on áthaladó, t-be érkező út miatt módosult, amikor az út élei megfordultak és az $(M(y), y)$ él a párosítás részévé vált.) Ebből az adódik, hogy $y \in Y_B$ akkor és csak akkor, ha $M(y) \in X_B$, hiszen ha y-ból nem lehetett eljutni t-be, akkor az általa mutatott pontból sem lehet, és a másik irány: ha az $M(y)$-ból nem lehet eljutni t-be, az egyetlen pontból, ahova y-ból el lehet jutni, akkor az y-ból sem lehet. Következésképpen: $|Y_B| \leq |X_B|$.

Másrészt minden $x \in X_B$-ből kilépő él $y \in Y_B$-ba kell lépjen. (Az észrevétel miatt azokat az éleket, amik $x \in X$-ből s-be mutatnának, töröltük, a maradék csak $y \in Y$-ba mutathat. Ha $x \in X_B$, akkor viszont a belőle kiépő élek olyan $y \in Y$-okba mutatnak, ahonnan nem lehet eljutni t-be, azaz ezekre az is igaz, hogy

$y \in Y_B$.) Ezért: $N(X_B) \subseteq Y_B$, vagyis az 1. tétel felhasználásával az következik, hogy $|X_B| \geq |Y_B|$.

A két fenti bekezdést összerakva, azt kaptuk, hogy $|X_B| = |Y_B|$, ugyanakkor hogy minden X_B-ből kilépő él Y_B-be érkezik, egyszersmind minden Y_B-ből induló él X_B-ben ér véget. Mivel a gráf összefüggőségét feltettük, ez azt implikálja, hogy Y_B vagy üreshalmaz, vagy maga az Y. A másik feltételünk, ami az volt, hogy a párosítás elemszáma kisebb, mint n, kizárja az utóbbit – lásd 2. bekezdés. Azonban az $Y_B = \emptyset$ magával vonja, hogy $X_B = \emptyset$, mivel a két halmaz elemszáma azonos.

Ezzel beláttuk, hogy nincs olyan pont se az X-ben, se az Y-ban, melyből ne lehetne elérni t-t, ha van még párosítatlan pontunk. □

2.2. A gyorsítás

A klasszikus Ford–Fulkerson algoritmus útnövelését BFS-sel szokták megoldani, ami minden ponthoz épít egy legrövidebb utat s-ből. A gyorsítás nagyon egyszerű. A szélességi keresés helyett csak induljunk el s-ből és sétáljunk az irányított reziduális gráfon véletlen választva mindig a következő pontot a szomszédok közül, amíg elérünk t-be. Ez a fenti 2. tétel következtében előbb-utóbb bekövetkezik. Hamarosan azt is belátjuk, hogy inkább előbb, mint utóbb. Ha pedig séta közben esetleg kört képeztünk, azt utólag vágjuk le. A módosítás eredménye egy $O(n \log n)$ idejű algoritmus.

1. Észrevétel. *Az már most is látszik, hogy az n darab útkeresés során az egyes utak hossza végül nem lehet több, mint kétszer a vissza élek száma a gráfban (minden vissza élre jut egy előre él) plusz még három (egy él s-ből az első $x \in X$-be, az első előre él $x \in X$-ből $y \in Y$-ba és az utolsó él $y \in Y$-ból t-be).*

3. Tétel. *Legyen $G = (X \cup Y; E)$ egy reguláris páros gráf, legyen M k élből álló párosítás G-ben, valamint G_M a kapcsolódó folyam probléma reziduális gráfja. Az s–t-randomsétában előforduló vissza élek várható száma ekkor:*

$$\frac{n}{n-k} - 1.$$

Bizonyítás. Jelölje X_M és Y_M rendre az X-beli, illetőleg az Y-beli párosított pontokat, X_U és Y_U azokat, amik kimaradtak a párosításból. Emellett adott $y \in Y$ esetén legyen még $M(y)$ az az X-beli pont, amely a párosításban y szomszédja. Továbbá minden v esetén vezessük be azt a $b(v)$ számot, ami egy v-ből induló séta esetén a t-be jutásig felmerülő vissza élek számának várható értéke.

A célunk felső becslést adni $b(s)$-re. Úgy fogjuk ezt elérni, hogy különböző v-k esetén megbecsüljük a $b(v)$-k közötti relációkat, majd kombináljuk ezeket. Ahhoz, hogy a kombinációk eredményei értelmesek legyenek, ellenőriznünk kellene, hogy a $b(v)$ véges minden v-re. Szerencsére erről biztosít minket a 2. tétel már, ami azt mondta ki, hogy minden pontból el lehet jutni t-be.

Az első lépés s-ből annak valamely random szomszédjába történik X_U-ban:

$$b(s) = \frac{1}{n-k} \sum_{x \in X_U} b(x). \tag{2.1}$$

Tudjuk, hogy az $y \in Y_U$ pontokra az egyetlen él, ami elhagyja őket, t-be lép be. Ezeknél $b(y) = 0$. Ugyanakkor, amennyiben $y \in Y_M$, a róla lelógó él feje $M(y)$-beli. Magyarán:

$$b(y) = \begin{cases} 0 & y \in Y_U, \\ 1 + b(M(y)) & y \in Y_M. \end{cases} \tag{2.2}$$

Egy $x \in X_U$-nak d darab belőle induló éle van még, melyek különböző $y \in Y$ elemekbe mutatnak. Azaz az x-ből induló séta várható értékét a szomszédokból induló séták várható értékének átlagából is kiszámíthatjuk. Tehát $x \in X_U$ esetén:

$$b(x) = \frac{1}{d} \sum_{(x,y) \in E} b(y) \Longrightarrow db(x) = \sum_{(x,y) \in E} b(y).$$

Ettől nem sokban tér el az az x, amelyre $x \in X_M$ áll, csak itt $d-1$ a kilépő élek száma (1 már belépő párosítás él lett), nem pedig d:

$$b(x) = \frac{1}{d-1} \sum_{(x,y) \in E-M} b(y) \Longrightarrow (d-1)b(x) = \sum_{(x,y) \in E-M} b(y).$$

Mivel $(x, y) \in M$ párosítás élek tövére $b(y) = 1 + b(x)$, azaz ezt átrendezve a fejére $b(x) = 1 - b(y)$ egyenlet áll fent. Ha a legutóbbi egyenletünk két oldalához hozzáadjuk ezt az egyenletet kapjuk:

$$db(x) = -1 + \sum_{(x,y) \in E} b(y).$$

Minden $x \in X$-re összegezve a fenti egyenletet azt nyerjük, hogy:

$$d \sum_{x \in X} b(x) = -k + \sum_{x \in X} \sum_{(x,y) \in E} b(y). \tag{2.3}$$

Amiatt, hogy minden $y \in Y$ d darab ponttal alkot élet X-ből, ez tovább egyenlő a:

$$-k + d \sum_{y \in Y} b(y) \tag{2.4}$$

értékkel. Továbbá ha a $b(y)$-ról szóló (2.2) formulát bevetjük az összes y-ra, az eredmény:

$$\sum_{y \in Y} b(y) = k + \sum_{x \in X_M} b(x). \tag{2.5}$$

Így (2.3), (2.4), valamint (2.5) összerakásával erre jutunk:

$$d \sum_{x \in X} b(x) = -k + dk + d \sum_{x \in X_M} b(x).$$

De ez azt is jelenti, a jobboldali d-vel szorzott szummás tagot kivonva mindkét oldalból, hogy:

$$d \sum_{x \in X_U} b(x) = (d-1)k.$$

Visszatérve a kiinduláshoz ez alapján végre behelyettesítünk s kezdeti (2.1) egyenletébe:

$$d(s) = \frac{1}{n-k} \sum_{x \in X_U} b(x) = \frac{(d-1)k}{d(n-k)} \leq \frac{k}{n-k} = \frac{n}{n-k} - 1.$$

Az állítást ezzel beláttuk, tényleg várhatóan legfeljebb $\frac{n}{n-k} - 1$ visszaél fordulhat elő az s–t-sétában, ha már k éle van a párosításnak. □

Ezek szerint az algoritmus futásának legelején csak néhány vissza élet kell bevegyünk. Konkrétan, ha az élek száma még kisebb, mint $n/2$, várhatóan kevesebb, mint 1-et. Azt is látjuk, hogy a k elemű M $k+1$ eleművé bővítéséhez sem kell több, legfeljebb

$$2 + 2\frac{n}{n-k}$$

élen áthaladnunk (ezt már nem csak a vissza élekre számoljuk, ez már valóban a lépésszám a 1. észrevétel alapján). Hogyha M méretét 0-ról n-re akarjuk bővíteni, az legfeljebb

$$\sum_{k=0}^{n-1} \left(2 + 2\frac{n}{n-k}\right) = 2n + 2n \sum_{i=1}^{n} \frac{1}{i}$$

lépést jelent az összes útban együtt véve. Itt az utolsó szumma az n. harmonikus számot jelöli. Közelítése $\gamma + \ln n$ formában ismert, ahol a $\gamma = 0.577215\ldots$ az Euler-konstans. Ebből a közelítésből most nekünk elég lesz annyi, hogy

$$\sum_{i=1}^{n} \frac{1}{i} \leq 1 + \ln n.$$

Visszatérve a várható lépésszámra kapott végső felső becslés valóban:

$$2n + 2n(1 + \ln n) = O(n \log n).$$

2.3. Input

Inputként a *regpsgraf1.cpp*-vel generáltam $2n$ pontú d-reguláris véletlen páros gráfot, futás időben bekért n-re és d-re. A gráfnak a szomszédsági listáját adtam meg, a Ford–Fulkersonos algoritmushoz ugyanis a cikkben ez a reprezentáció volt javasolt. Az eredményben a szomszédok sorszáma 2-től $n + 1$-ig tartott (az első pontosztály pontjait is így képzeltem az indexet eltolva), fenntartva az 1-es értéket s-nek (az $n + 2$ pedig t-t jelölte). Egy tétel sem mondta, hogy a d-reguláris gráfnak nem szabad párhuzamos élt tartalmaznia, pusztán az összefüggőség volt elvárás, mégis törekedtem rá, hogy a párhuzamos éleket elkerüljem. Szerencsétlen esetben a gráf összefüggőségét is elronthatták, amit nem akartam még egy rutinnal ellenőrizni. Mindezek függvényében először:

Haladva a mátrix elemein, soronként egy 1-től n-ig számokkal feltöltött *sor1* tömböt és egy *sor2* pointer tömböt definiáltam, az utóbbit a *sor1* elemeire mutatva. Ők jelentik hamarosan, hogy milyen elemek közül lehet véletlenszerűen választani az adott sorban. A pointer átállítós módszerrel, kiszedtem közülük azt, ami korábban már bekerült a *nemkell* listába. (Ide azon elemek voltak beszúrva, akik a *számláló* tömb alapján korábban már d-szer előfordultak a mátrixban.) Ezek után a véletlen elem, amit generáltam a *sor1* tömb indexeként nem mutathatott olyan elemre, amiből már elég van. A generálás után ismét pointerátállítással töröltem egyet a választható elemek közül, valamint, ha a *számláló* megkívánta, be is szúrtam a *nemkell* listába. Ezt az eljárást ismételtem az $n - 1$. sorig, ahova pedig első lépésben beszórtam a megmaradt éleket nagyság szerint növekvő sorrendben.

Az $n - 1$. sorig ez még jó eséllyel sikerült a párhuzamos élektől mentesnek maradni (a futtatások során néha láttam ismétlődést itt az utolsóban már az utolsó két elemre). Az volt még a célom, hogy az n. sorban ismétlődő elemeket becseréljem a mátrix más elemeire. Kihasználva a nagyságsorrendet, az elemeken lépkedve ellenőriztem, hogy az i. és $i + 1$. azonos-e. Amennyiben így volt, elindultam a páros gráf sorai között keresni egy olyat, amiben nem szerepelt az i. elemnek megfelelő érték – az első ilyet választottam. Ebben a sorban pedig kerestem egy olyan elemet, ami viszont az n. sorban nem szerepelt. Ilyen biztos volt, mivel a mátrix ezen korábbi sorában d különböző elem van – hiszen az első jó sort választottam – úgy, hogy egyik sem maga az i. elemnek megfelelő érték. Elvégeztem a cserét.

Amit kaptam gráfot, abban már csak egy-két párhuzamos él akadhatott nagy ritkán (említettem az utolsó előtti sorbeli bugot), úgyhogy nem is foglalkoztam tovább az előállítás bonyolításával. Kiírtam egy *regpsgraf.txt* fájlba.

Ez az eljárás sajnos nagy n-kre és d-kre nem működött jól, valószínűleg azért, mert már az n. előtti sorra sem feltétlenül jutott d különböző érték, és nem mindig csak az utolsó két elem volt a hibás. Azért volt ez probléma, mert a végső konstrukcióban szerettem volna automatizálva, sok különböző inputot kapni, amiket szintén automatizálva beadhatok a futó programnak. Ezért elhagytam a párhuzamos élekkel szemben támasztott elvárásomat és írtam egy egyszerűbb, de stabilabb algoritmust is. Az automatizálásnál amíg lehetett az elsőt, utána a második programot alkalmaztam. (Az összefüggőséget erre sem ellenőriztem.)

A *regpsgraf2.cpp* a mátrix egy pozíciójára pusztán egy n elemű, 2-től $n+1$-ig feltöltött tömbből választott véletlenül, minden elemet legfeljebb d-szer. Amint egy elemhez tartozó számláló elérte a d-t, az adott elem kiesett a generálhatóak közül.

Akárhogyan is, de a *parosgraf.txt* előállt, rá is térhetünk az algoritmusra...

2.4. A megvalósítás

Felhasznált függvények:

*double** foglalas(int n, int k)* : Mátrixot foglal.

*void felszabaditas(double** tomb, int sorszam)* : Mátrix foglalást szüntet meg.

int vanbennekor(list<int> l) : Eldönti listáról, hogy van-e benne ismétlődő elem.

void kormentesit(list<int>& l1, list<int>& l2, int ke, int eltole) : Törli a kört a sétából.

A *parosgraf.txt*-ben szereplő mátrixot behívtam egy `X` változóba, és gyártottam hozzá egy n elemű `Xmeret` tömböt. Ez kezdetben X minden sorához d elemet rendelt, majd iterációról iterációra egyesével lecsökkent $d-1$-re. Erre a tömbre azért volt szükség, mert azon sorok, amikbe egy útkeresésnél elsőként kerültünk `X`-ben az élek megfordításával veszítettek 1-et az általuk mutatott szomszédok közül (s-be ugyanis nem mutathat él). Feltöltöttem még egy `Y` tömböt azonosan $n+2$-vel, jelezve, hogy az Y pontosztály elemei mind t-be mutatnak. Külön létrehoztam `STX`, `STY`, `MX`, és `MY` listákat, hogy az út és párosítás X-beli és Y-beli részét számontartsam. `X` sorai közül a sétában elsőt minden iterációban egy csökkenő méretű `s` tömb segítségével választottam egyenletes valószínűséggel.

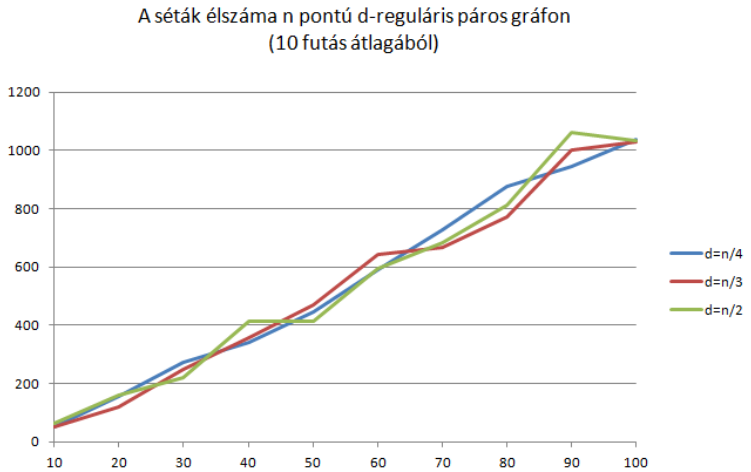
Az n darab útkeresés egyike így nézett ki: Először `STY`-ba betettem 1-et. Majd a következőket váltogatva attól függően, hova tartoznak, lejegyeztem `STX` és `STY`

valamelyikébe. Elindulásként három élet megjegyeztem a-t, b-t és c-t. Az a az X s-sel generált véletlen sora (az s-ből $x \in X$-be mutató él) volt, a b az X a sorának egy véletlen választott eleme ($x \in X$-ből $y \in Y$-ba mutató él) és c az Y b sorában talált elem ($y \in Y$-ból vagy t-be, vagy $x \in X$-be mutató él). Amennyiben c nem t-be mutatott, akkor addig, amíg ez nem történt meg, X és Y elemein lépdeltünk c-vel és b-vel.

A keletkezett sétában a `vanbennekor(STX)` és a `vanbennekor(STY)` segítségével megkerestem az esetlegesen ismétlődő pontokat `korelem1`, `korelem2`. Majd attól függően, melyikben találtam, meghívtam a `kormentesit(STX,STY,korelem1,1)`, illetve a `kormentesit(STY,STX,korelem2,0)` függvényt, hogy a rá illeszkedő kört töröljem. Ezt követően két-két iterátort léptetve az utak listáin és a párosítások listáin, frissítettem a párosítást.

Legvégül került sorra a gráf frissítése. Először az `s` tömb éleit rendeztem, hogy az elhasznált kezdő (s, x) $x \in X$ élet ne lehessen újra generálni. Lecsökkentettem az `Xmeret` változóját `X` megfelelő során. Majd az `STX` és `STY` listákon iterátorral haladva felülírtam az `X`-et és `Y`-t. (X sorában meg kellett keresnem azt az elemet, amire léptem az úton, majd ide kellett beszúrnom azt a pontot, ami a visszairányítás után lett szomszéd.)

Az algoritmus vizsgálatának tárgyául a lépésszámok alakulását választottam, mert a cikkekből az derült ki, hogy ez várható értékben és nagy valószínűséggel is $O(n \log n)$-es lesz. A kapott *fordfulk.cpp*-t futtattam $n = 10, 20, ..., 100$, $d = \lfloor n/4 \rfloor, \lfloor n/3 \rfloor, \lfloor n/2 \rfloor$ esetekre, minden esetre 10-szer, és vettem a 10 lépésszám átlagát. A sétákban előforduló élek számára így az alábbi grafikont kaptam, amely alátámasztja a korábbi eredményeket.



3. fejezet

Gyors párosítás Gauss-eliminációval

Az algoritmus, amelyet a [2] cikk ír le, lényegében csak egy ponton használ véletlent. Egy kezdeti mátrix előállításánál. Ott sem lesz komoly szerepe maguknak a véletlen számoknak, csupán annyi, hogy egy adjacencia mátrix-ot tesznek invertálhatóvá. Ami ennél sokkal inkább magját képezi a gyorsításnak az a gyors mátrixszorzó algoritmusok jelenléte. Ezeknek köszönhetően a futási idő $O(n^\omega)$, ahol ω a legjobb ilyen exponense, ez kisebb, mint 2.38. Ez jobb, mint a korábbi $O(n^{2.5})$-es eredmény, amit sűrű gráfok esetén az $O(m\sqrt{n})$-es futásidejű algoritmusok adtak (Micali és Vazirani: [3], Blum: [4] , Gabow és Tarjan: [5]). Az előző elgondoláshoz képest az a nagy előrelépés, hogy ez az algoritmus – pontosabban ennek egy változata – már általános gráfokon is futtatható. Épp ezért a kezdetben kimondott tételek nem csupán páros gráfokra lesznek megfogalmazva.

A cikk megválaszol még egy Lovász által annak idején feltett, sokáig nyitott kérdést is: Lehet-e a teljes párosítás tartalmazásának $O(n^\omega)$ idejű tesztelését olyan technikává alakítani, amely meg is ad egyet ugyanennyi idő alatt.

3.1. Gyors mátrixszorzó eljárások

Az alfejezetnek nem célja, hogy mély ismeretekkel szolgáljon a gyors mátrixszorzó eljárásokkal kapcsolatban. Egy benyomást azonban a téma érintésével mégiscsak kelteni kíván, lévén a hamarosan sorra kerülő algoritmus hatékonysága múlik a mátrixok szorzásának mikéntjén. Felhasznált irodalom: Wikipedia / Matrix multiplication / Algorithms for efficient matrix multiplication.

Sokáig, egészen 1969-ig nem tudtunk jobbat a naiv mátrix szorzásnál. Ez az eredménymátrixnak mindegyik (n^2 db) eleméhez n darab szorzás árán jut el. Volker

Strassennek köszönhető az az észrevétel, hogy egy 2×2-es mátrix, ami eredetileg 8 szorzást igénylene, megvalósítható 7-tel annak árán, hogy az összeadások és kivonások száma kicsivel megnő. Ez már $O(n^{\log_2 7})$-es, azaz kb. $O(n^{2.807})$ futásidőt jelent. Tegyük fel, hogy n 2-hatvány, A és B $n \times n$-es mátrixok. Ekkor $C = AB$ négy darab $n/2 \times n/2$-es mátrixra bontható:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Ekvivalensen:

$$\begin{aligned} r &= ae + bg, \\ s &= af + bh, \\ t &= ce + dg, \\ u &= cf + dh. \end{aligned}$$

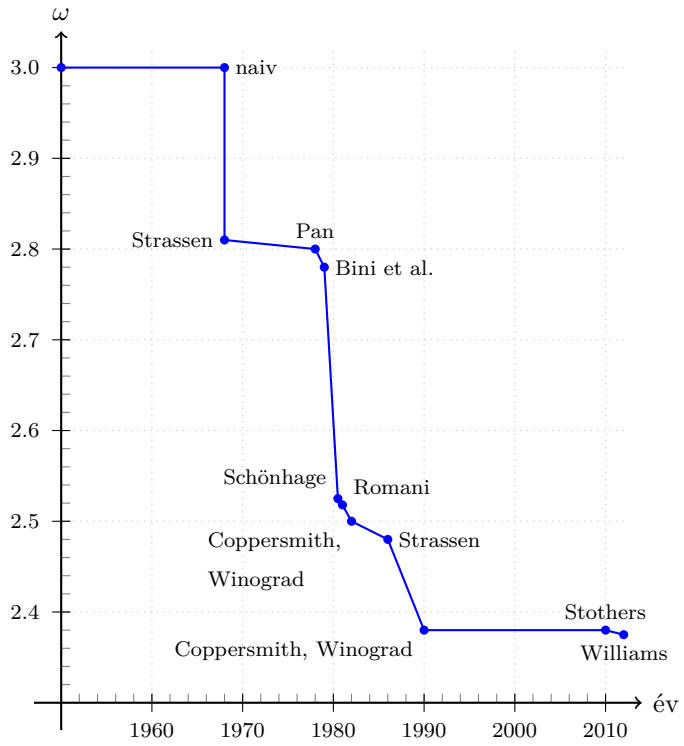
A módszer négy lépésből áll:

1. A, B mátrixokat részmátrixokra bontjuk.
2. Ezekből $\Theta(n^2)$ nagyságrendű skalár összeadás és kivonás segítségével 14 darab $n/2 \times n/2$ méretű mátrixot készítünk: $A_1, B_1, A_2, B_2, ..., A_7, B_7$.
3. Kiszámítjuk a $C_i = A_i B_i$ szorzatokat.
4. $\Theta(n^2)$ nagyságrendű skalár összeadás és kivonás segítségével, a C_i-k kombinációjaként megadjuk a C-t.

Ennyiből már látszik talán, milyen módon lehet időt nyerni, és nem fog meglepetésként érni bennünket, ha meglátjuk, hogy a most következő gyors párosítási algoritmus 2-hatványokkal operál. Aki részletesebben szeretne olvasni erről, az megtalálja Strassen algoritmusát a [15] kiadványban.

Strassen munkássága beindított egy folyamatot, minek következtében a ma ismert legjobb gyors mátrixszorzó algoritmus $O(n^{2.3727})$ idő alatt teljesít.

A fejezet magját képező cikk ajánlotta gyors mátrixszorzó eljárás Coppersmith és Winograd munkája [6]. Ez ugyan nem a legjobb, amit egyébként Vassilevska Williamsnek köszönhetünk, de gyökerét képezi annak. Mi ezek helyett az implementáció során megmaradunk az egyszerű $O(n^3)$-ös eljárásnál.



3.1. ábra. A gyors mátrixszorzó algoritmusok fejlődése, forrás: [16].

3.2. Előzmények

Legyen $G = (V; E)$ gráf, melynek $|V| = n$ pontja $v_1, v_2, ..., v_n$. Definiáljuk az alábbi $n \times n$-es $\widetilde{A}(G)$, ferdén szimmetrikus mátrixot hozzá:

$$\widetilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{ha } (v_i, v_j) \in E \text{ és } i < j, \\ -x_{i,j} & \text{ha } (v_i, v_j) \in E \text{ és } j < i, \\ 0 & \text{különben.} \end{cases}$$

Ahol $x_{i,j}$-k egyedi változókat jelölnek.

Tutte bizonyította a következőt [7]:

4. Tétel. *A $det(\widetilde{A}(G))$ szimbolikus determináns nem 0, ha G-ben van teljes párosítás.*

Amit Lovász általánosított [8]:

5. Tétel. *Ha M maximális párosítás G-ben, akkor $rang(\widetilde{A}(G)) = 2|M|$.*

Az $x_{i,j}$ elemeket helyettesítsük véletlen számokkal az $\{1, ..., R\}$ halmaz halmazból választva, ahol $R = n^{O(1)}$, és a keletkező mátrixot jelöljük $A(G)$-vel. Ezentúl erre

hivatkozunk majd *random adjacencia mátrixként*. Lovász még azt is bebizonyította, hogy:

6. Tétel. *rang*$(A(G)) \leq 2|M|$. *És az egyenlőség valószínűsége legalább* $1 - (n/R)$.

Ez az ismeret nyújtotta azt a randomizált algoritmust, ami eldönti, hogy egy gráf tartalmaz-e teljes párosítást. $A(G)$ determinánsa nagy valószínűséggel nem nulla, ha G-ben van teljes párosítás – az eljárás pedig, ami ezt a determinánst megadja $O(n^\omega)$ idejű.

Rátérve páros gráfokra a szokásos ajdacencia mátrixszal, ahol a mátrix sorai az egyik pontosztály n pontját, oszlopai pedig a másik pontosztályét jelentik, hasonló tételeket kapunk. Legyen $G = (X \cup Y; E)$ páros gráf, melyben $X = \{x_1, x_2, ...x_n\}$, $Y = \{y_1, y_2, ...y_n\}$ és definiáljuk hozzá a *random adjacencia mátrixot* a következőképpen:

$$\widetilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{ha } (x_i, y_j) \in E \\ 0 & \text{különben.} \end{cases}$$

Erre vonatkozóan a 4. tételben nincs változás, a 5.-6. tételekben pedig csak lejön a 2-es szorzó, vagyis „rang$(\widetilde{A}(G)) = |M|$”, illetve „rang$(A(G)) \leq |M|$” mellett igazak az állítások.

Lépjünk tovább: Tegyük fel ismét egy tetszőleges G-ről (vagy páros gráfról), hogy van benne teljes párosítás. Jelölje a belőle képzett random adjacencia mátrixot $A(G)$ – ugye ez az $A(G)$ nagy valószínűséggel invertálható. És most jön, amire majd elsősorban épít az algoritmus:

7. Tétel. *Ha* $G - \{v_i, v_j\}$*-ben van teljes párosítás, akkor nagy valószínűséggel az is fennáll, hogy* $A(G)^{-1}_{j,i} \neq 0$.

Tehát az inverz vizsgálatával határozzuk meg egy él *megengedettségét*, ti. azt, hogy a kiválasztása során nem rontjuk-e el annak a lehetőségét, hogy a teljes párosítás építését siker koronázza. Ennek természetes feltétele, hogy az adott élt tartalmazza teljes párosítás. A tétel maga és a 6. tétel Zippel [9] és Schwartz [10] munkáin alapuló alábbi lemmából következik:

1. Lemma. *Ha* $p(x_1, x_2, ...x_m)$ *d-fokú nemnulla polinom, melynek együtthatói egy testből valók,* S *egy test részhalmaza, akkor annak a valószínűsége, hogy a polinomba behelyettesítve* $\{s_1, s_2, ...s_m\} \in S^m$ *elemeket, melyek közül egyenletes eloszlás szerint választunk, az eredmény* 0*, legfeljebb* $d/|S|$.

A 6. tételhez elég $\widetilde{A}(G)$ determinánsát venni n-fokú polinomnak az $x_{i,j} \in \widetilde{A}(G)$ változókkal. A 7. tételhez még fel kell használni, hogy $A^{-1}_{i,j} = adj(A)_{i,j}/det(A)$ azzal

a jelöléssel, hogy $adj(A)_{i,j} = det(A')(-1)^{i+j}$, ahol A' maradt miután A-ból töröltük a j. sort és i. oszlopot. Megmutatható [11], hogy ezek a polinomok nemnullák a véges $\mathbb{Z}_p$ test felett, ezért mind a két tételre alkalmazható a lemma.

Valójában a lemmából az is következik, hogy a test rendjét adó p-nek elég n-ben polinomiálisnak lennie. Ami meg azért jó, mert a véges test műveletek konstans időben végrehajthatóak – kivéve az osztást, de azt a többi művelet dominálni fogja.

Már kezd kirajzolódni, hogyan is fog működni az algoritmus. Most még ennyit látunk belőle:

```
1. lépés: Generálunk A(G) random adjacencia mátrixot.
2. lépés: Kiszámoljuk az inverzét.
3. lépés: Megengedett élet keresünk.
4. lépés: A megengedett élet betesszük a párosításba.
5. lépés: A megengedett élet eliminájuk a gráfból.
6. lépés: Frissítjük A(G)-t.
```

Azt azért sejtjük, hogy nem lenne az algoritmus túl gyors, ha minden lépésben az újabb mátrixnak újabb véletlen $x_{i,j}$ elemekre lenne szüksége. Szerencsére Rabin és Vazirani megmutatták, hogy erre nincs szükség:

8. Tétel. *Ha $A(G)$ nemszinguláris, akkor minden v_i elemhez létezik olyan v_j, hogy $A(G)_{i,j} \neq 0$ és $A(G)^{-1}_{j,i} \neq 0$, azaz (v_i, v_j) megengedett él. Továbbá az $A(G)$-ből az i. és j. sorokat-oszlopokat törölve továbbra is nemszinguláris mátrixot kapunk.*

A későbbi inverzek számítását egy egyszerű képlettel fogjuk megtenni a Schur komplementer tulajdonságait kihasználva:

9. Tétel. *Legyen*

$$A = \begin{pmatrix} a_{1,1} & v^T \\ u & B \end{pmatrix},\ A^{-1} = \begin{pmatrix} \hat{a}_{1,1} & \hat{v}^T \\ \hat{u} & \hat{B} \end{pmatrix},$$

ahol $\hat{a}_{1,1} \neq 0$. Ekkor igaz a következő:

$$B^{-1} = \hat{B} - \frac{uv^T}{\hat{a}_{1,1}}.$$

Bizonyítás. Mivel $AA^{-1} = I$:

$$\begin{pmatrix} a_{1,1}\hat{a}_{1,1} + v^T\hat{u} & a_{1,1}\hat{v}^T + v^T\hat{B} \\ u\hat{a}_{1,1} + B\hat{u} & u\hat{v}^T + B\hat{B} \end{pmatrix} = \begin{pmatrix} I_1 & 0 \\ 0 & I_{n-1} \end{pmatrix}$$

Elindulva $B(\hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1})$ -ből a fenti egyenlőségeknek köszönhetően ki fogjuk hozni az $(n-1)\times(n-1)$-es egységmátrixot. Ha ez sikerül, csak az egyenlet két oldalát kell megszoroznunk B^{-1}-zel balról, és megvan a megoldás. Először beszorzunk a B-vel és azt használjuk ki, hogy $u\hat{v}^T + B\hat{B} = I_{n-1}$:

$$B(\hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}) = I_{n-1} - u\hat{v}^T - B\hat{u}\hat{v}^T/\hat{a}_{1,1} =$$

majd azt, hogy $u\hat{a}_{1,1} + B\hat{u} = 0$:

$$= I_{n-1} - u\hat{v}^T + u\hat{a}_{1,1}\hat{v}^T/\hat{a}_{1,1} =$$

ezután egyszerűsítünk és elvégezzük a kivonást:

$$= I_{n-1} - u\hat{v}^T + u\hat{v}^T = I_{n-1}.$$

□

Megjegyzés: Az a módosítás, amit a $\hat{B}$ leír igazából nem más, mint a Gauss-elimináció egy lépése. Ugyan az egyszerűség kedvéért az 1. sor, 1. elemét eliminálták, de $A(G)^{-1}$ tetszőleges elemét lehet így kezelni $\hat{u}$, $\hat{v}^T$ és $\hat{B}$ megfelelő meghatározásával.

3.3. Az algoritmus ismertetése több lépcsőben

1. Változat:

1. algoritmus. Az egyszerű algoritmus

$B = A(G)^{-1}$;
for *c=1* **to** n **do**

1. Keressünk egy olyan r sort, ami nincs eliminálva és amire sem $A(G)_{c,r} \neq 0$, sem $B_{r,c} \neq 0$, vagyis (x_c, y_r) megengedett él.
2. Eliminájuk az r. sort és a c. oszlopot B-ből.
3. Adjuk (x_c, y_r) élet M-hez.

end

Ha az első $i-1$ sor és oszlop eliminálása után $A(G)_{i,i} \neq 0$, szerencsénk van, a Gauss-eliminációt megúsztuk pivotálás nélkül az i. lépésben. Tegyük fel most, hogy

végig ilyen szerencsénk lesz, és a főátló elemeit tudjuk eliminálni. Hasonlóan, mint korábban az inverz újraszámításánál, könnyebb így leírni azt a fajta gyorsítást, amit a 2. változatban fogunk eszközölni. Az implementációnál majd látni fogjuk, hogy nem kell sok ahhoz, hogy tetszőleges nemnulla oszlopelemet vegyünk.

Bevezetünk egy spóroló eljárást, amire a továbbiakban a *lusta eliminációt* néven hivatkozunk majd. Az alapgondolat: Ne számoljuk ki az inverzeket minden elemre, amire aktuálisan lehet, gyűjtögessük az információkat egy esemény bekövetkeztéig. Információ alatt most a megfelelő uv^T/c értékeket értjük, az esemény egy kicsit bonyolultabb, magyarázatát elhalasztjuk kicsit későbbre. A végrehajtást, ha már a sorok R halmazának és az oszlopok C halmazának „ütött az órája" – akármit is jelentsen ez most –, az UPDATE(R,C)-vel végezzük el az $A(G)_{R,C}$ részmátrixon. Ezzel az alábbi észrevétel miatt nyerhetünk időt:

Legyenek az $A(G)_{R,C}$ részmátrix $|C|$ darab oszlopára váró frissítések rendre az $u_1v_1^T/c_1, u_2v_2^T/c_2, ..., u_kv_k^T/c_k$. Ekkor a felhalmozódott frisítéseket egyetlen mátrix szorzással és az így kapott szorzatmátrixot a részmátrix elemeiből való kivonással elintézhetjük, mivel elég kiszámolni

$$u_1v_1^T/c_1 + u_2v_2^T/c_2 + ... + u_kv_k^T/c_k = U\overline{V}$$

értékét, ahol U egy $|R| \times k$ mátrix az $u_1, u_2, ..., u_k$ oszlopokból és $\overline{V}$ egy $k \times |C|$-as mátrix $v_1^T/c_1, v_2^T/c_2, ..., v_k^T/c_k$ sorokból. $U\overline{V}$ magától értetődően gyors mátrixszorzással kiszámítható.

Az eseménnyel kapcsolatban: egyszerűsítsük a kérdést még, tegyük fel, hogy az $A_{k,k}$ $(1 < k \leq n)$ elemre vagyunk csak kíváncsiak, pontosabban arra, hogy mikor kerüljön sor a frissítésére. Emlékezve arra, hogy a gyors mátrixszorzó algoritmusok a 2-hatvány oldalú mátrixokat szeretik, így fogjuk ezt meghatározni: frissítjük $A_{k,k}$-t az $i < k$. iterációban, ha az i. oszlopban állva, a $k < i + 2^j$, ahol 2^j az i-nek a legnagyobb 2-hatvány osztója. Ahhoz, hogy megértsük, mi történik a többi elemre vonatkozóan, tekintsük az algoritmus leírást. Itt az UPDATE(A, B)-ben az A sorok és a B oszlopok halmazát jelöli:

2. Változat:

2. algoritmus. Pivot nélküli Gauss-elimináció

$B = A(G)^{-1}$;

for *i=1* **to** n **do**

1. Lustán eliminájuk az i. sort és az i. oszlopot B-ből.
2. Keressük meg a legnagyobb j egészt: $2^j \mid i$.
3. UPDATE($\{i+1, ..., i+2^j\}$,$\{i+1, ..., n\}$).
4. UPDATE($\{i+2^j+1, ..., n\}$,$\{i+1, ..., i+2^j\}$).

end

Ez az algoritmus egyébiránt a standard rekurzív faktorizációs algoritmus iteratív leírása, és őrzi az $O(n^\omega)$ komplexitást, mint azt a lemmánk mindjárt alátámasztja:

2. Lemma. *Az updatekben előforduló változtatások száma legfeljebb* 2^j.

Bizonyítás. Az i. iterációban az $i+1, ...i+2^j$ sorok, valamint ugyanezen indexű oszlopok frissítése zajlik. Mivel $2^j|i$, és j volt a legnagyobb ilyen szám, amire ez teljesül, $2^{j+1} \not| i$. Tehát $i = 2^j(2k+1)$ valamely $k \geq 0$ egészre. Átrendezve adódik, hogy $i - 2^j = 2^j(2k)$, ami magában hordozza azt az információt, hogy $2^{j+1}|i-2^j$. Ezek szerint soraink és oszlopaink már voltak frissítve az $i-2^j$. lépésben. Így a változtatások száma ebben a körben tényleg legfeljebb 2^j lehet. □

A lemma arra is rámutat, hogy az i. iterációban elkövetett update költsége arányos egy $2^j \times 2^j$-es mátrix, meg egy $2^j \times n$-es mátrix szorzásával. Ha a második mátrixot is $2^j \times 2^j$-es részmátrixokra szedjük, a művelet megtehető akár $n/2^j(2^j)^\omega = n(2^j)^{\omega-1}$ időben is. Felhasználva, hogy minden j $n/2^j$-szer fordul elő, a teljes komplexitás így alakul:

$$\sum_{j=0}^{\lceil \log n \rceil} \frac{n}{2^j} n(2^j)^{\omega-1} = n^2 \sum_{j=0}^{\lceil \log n \rceil} (2^{\omega-2})^j = O(n^2(n^{\omega-2})^{\lceil \log n \rceil}) = O(n^\omega). \qquad (3.1)$$

Tehát:

10. Tétel. *A pivotálás nélküli Gauss-eliminációs algoritmus a lustán frissítő sémát követve* $O(n^\omega)$ *futásidejű.*

Még egy csekély módosítást alkalmazunk felismerve, hogy nincs szükségünk a teljes sorok update-jére sem, ezeket is elintézhetjük később.

3. Változat:

3. algoritmus. A gyors párosítási algoritmus

$B = A(G)^{-1}$;
$M = \emptyset$;
for *c=1* **to** n **do**

1. Keressünk egy olyan r sort, ami nincs eliminálva és amire sem $A(G)_{c,r} \neq 0$, sem $B_{r,c} \neq 0$, vagyis (x_c, y_r) megengedett él.
2. Lustán eliminálju az r. sort és a c. oszlopot B-ből.
3. Adjuk (x_c, y_r) élet M-hez.
4. Legyen j a legnagyobb egész, melyre $2^j | c$.
5. UPDATE$(c+1, c+2, ...c+2^j)$.

end

Tegyük fel, hogy a futás során éppen a c oszlopnál tartunk és most amit biztosan frissíteni akarunk az a $c+1., c+2., ..., c+2^j$. oszlop. Ezek korábban már a $(c-2^j)$-edik iterációnál frissítve voltak, vagyis a $c-2^j+1, c-2^j+2, ..., c$ oszlopokhoz tartozó frissítéseket kell még elvégezni rajtuk. Maradjunk még mindig annál a verziónál, hogy a sor és oszlopindexe a kiválasztott elemeknek ugyanaz. Ekkor az 1. elvégzendő updateben az $\hat{a}_{c-2^j+1,c-2^j+1}$ elem és a hozzá tartozó még nem eliminált sorelemekből álló $\hat{v}^T$, és oszlopelemekbő álló $\hat{u}$ játszik szerepet. Ezzel nem lesz gondunk. Egészen hasonló a helyzet a 2. elvégzendő frissítésnél is, de innen már hiányoznak $\hat{v}^T$ azon elemei, amik az aktuális oszlopaink felett vannak: $B(c-2^j+2, c+2), B(c-2^j+2, c+2), ..., B(c-2^j+2, c+2^j)$. Sőt ezek előállítása előtt azzal is szembesülünk, hogy felettük is még vannak hiányosságok. De nem baj, a lusta eliminációs ötlet nem használódott el, ismét bevethető egy „kisupdate"-en belül. Az i. lépésben itt is keressük meg azt a legnagyobb l számot, melyre $2^l | i$, és egyszerre mindig ennyi sorra végezzük el a változtatást.

Hogy egy kis szemlélettel támogassuk a megértést egy kis példán megmutatjuk az algoritmus futását. Az ábrákon az látszik majd, hogy az elemek hány frissítésen vannak már túl. Kezdetben mindegyik a 0 állapotban van. A bekeretezett számok az eddig kiválasztott elemek, amik a párosításba kerültek, a vastagon sze-

dett számok pedig azok, amik az U, illetve $\overline{V}$ mátrixokat alkotják, szerepük van az update/kisupdate során.

1. lépés

	1	2	3	4	5	6	7	8
1	[0]	**0**	0	0	0	0	0	0
2	**0**	1	0	0	0	0	0	0
3	**0**	1	0	0	0	0	0	0
4	**0**	1	0	0	0	0	0	0
5	**0**	1	0	0	0	0	0	0
6	**0**	1	0	0	0	0	0	0
7	**0**	1	0	0	0	0	0	0
8	**0**	1	0	0	0	0	0	0

2. lépés (kis update)

	1	2	3	4	5	6	7	8
1	[0]	0	**0**	**0**	0	0	0	0
2	**0**	[1]	1	1	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0

2. lépés (update)

	1	2	3	4	5	6	7	8
1	[0]	0	**0**	**0**	0	0	0	0
2	0	[1]	**1**	**1**	0	0	0	0
3	**0**	**1**	2	2	0	0	0	0
4	**0**	**1**	2	2	0	0	0	0
5	**0**	**1**	2	2	0	0	0	0
6	**0**	**1**	2	2	0	0	0	0
7	**0**	**1**	2	2	0	0	0	0
8	**0**	**1**	2	2	0	0	0	0

3. lépés

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	0	0	0	0
2	0	[1]	1	1	0	0	0	0
3	0	1	[2]	**2**	0	0	0	0
4	0	1	**2**	3	0	0	0	0
5	0	1	**2**	3	0	0	0	0
6	0	1	**2**	3	0	0	0	0
7	0	1	**2**	3	0	0	0	0
8	0	1	**2**	3	0	0	0	0

4. lépés (1. kis update)

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	**0**	**0**	**0**	**0**
2	**0**	[1]	1	1	1	1	1	1
3	0	1	[2]	2	0	0	0	0
4	0	1	2	[3]	0	0	0	0
5	0	1	2	3	0	0	0	0
6	0	1	2	3	0	0	0	0
7	0	1	2	3	0	0	0	0
8	0	1	2	3	0	0	0	0

4. lépés (2. kis update)

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	**0**	**0**	**0**	**0**
2	0	[1]	1	1	**1**	**1**	**1**	**1**
3	**0**	**1**	[2]	2	2	2	2	2
4	**0**	**1**	2	[3]	2	2	2	2
5	0	1	2	3	0	0	0	0
6	0	1	2	3	0	0	0	0
7	0	1	2	3	0	0	0	0
8	0	1	2	3	0	0	0	0

4. lépés (3. kis update)

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	0	0	0	0
2	0	[1]	1	1	1	1	1	1
3	0	1	[2]	2	**2**	**2**	**2**	**2**
4	0	1	**2**	[3]	3	3	3	3
5	0	1	2	3	0	0	0	0
6	0	1	2	3	0	0	0	0
7	0	1	2	3	0	0	0	0
8	0	1	2	3	0	0	0	0

4. lépés (update)

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	**0**	**0**	**0**	**0**
2	0	[1]	1	1	**1**	**1**	**1**	**1**
3	0	1	[2]	2	**2**	**2**	**2**	**2**
4	0	1	2	[3]	**3**	**3**	**3**	**3**
5	**0**	**1**	**2**	**3**	4	4	4	4
6	**0**	**1**	**2**	**3**	4	4	4	4
7	**0**	**1**	**2**	**3**	4	4	4	4
8	**0**	**1**	**2**	**3**	4	4	4	4

5. lépés

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	0	0	0	0
2	0	[1]	1	1	1	1	1	1
3	0	1	[2]	2	2	2	2	2
4	0	1	2	[3]	3	3	3	3
5	0	1	2	3	[4]	**4**	4	4
6	0	1	2	3	**4**	5	4	4
7	0	1	2	3	**4**	5	4	4
8	0	1	2	3	**4**	5	4	4

6. lépés (kisupdate)

	1	2	3	4	5	6	7	8
1	[0]	0	0	0	0	0	0	0
2	0	[1]	1	1	1	1	1	1
3	0	1	[2]	2	2	2	2	2
4	0	1	2	[3]	3	3	3	3
5	0	1	2	3	[4]	4	**4**	**4**
6	0	1	2	3	**4**	[5]	5	5
7	0	1	2	3	4	5	4	4
8	0	1	2	3	4	5	4	4

6. lépés (update)

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	1
3	0	1	2	2	2	2	2	2
4	0	1	2	3	3	3	3	3
5	0	1	2	3	4	4	**4**	**4**
6	0	1	2	3	4	5	**5**	**5**
7	0	1	2	3	**4**	**5**	6	6
8	0	1	2	3	**4**	**5**	6	6

7. lépés

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	1
3	0	1	2	2	2	2	2	2
4	0	1	2	3	3	3	3	3
5	0	1	2	3	4	4	4	4
6	0	1	2	3	4	5	5	5
7	0	1	2	3	4	5	6	**6**
8	0	1	2	3	4	5	**6**	7

Az utolsó, 8. lépésben már a kiválasztás is egyértelmű.

Felmerül a kérdés, hogy hány lépésből valósítható meg a $c+1, c+2..., c+2^j$ oszlopok frissítése? Először is mindenképp kell a közvetlenül felettük levő 2^j darab elemet frissíteni. Ezekhez az i. frissítésnél egy $2^l \times 2^l$-es és egy $2^l \times 2^j$-es mátrixot kell összeszorozni valamely gyorsmátrixszorzó eljárással. Ezt hívtuk korábban „kisupdate"-nek. A komplexitás tehát $O(2^{j\omega})$, a 3.1 képlet szerint (n helyébe 2^j-t írva).

Miután a megfelelő elemek készen állnak, jöhet az update, mégpedig egy $(n-c)\times 2^j$-es és egy $2^j \times 2^j$-es mátrix összeszorzása formájában. A szorzás műveletek összesen $O((n/2^j)2^{j\omega}) = O(n(2^j)^{\omega-1})$ időben megtehetőek. Összegezve, amit kaptunk minden j-re, megkapjuk a futásidőt: $O(n^\omega)$.

3.4. Input

Az input előállítása során építettem a korábbi eredményekre. Egy egyszerű programmal az éllistás leírást, adjacenciamátriszá transzformáltam. A *psgraf.cpp* bekérte a *psgraf.txt*-ben tartalmazott információt. Egy megfelelő méretű azonosan 0 mátrix elemeit írta felül az éllistás információk alapján: az éllista (i, j) pozíciójában k-t olvasva az $n \times n$-es mátrix $(i, k-2)$ pozíciójába beírt egy 1-est. Végül az 1-eseket felülírta egy nagy véletlen halmazból generálva elemet. Az outputja egy *psgraf2.txt* volt. Itt elvesztek a párhuzamos élek, ha voltak.

3.5. A megvalósítás

A felhasznált függvények:

*double** foglalas(int n, int k)* : Mátrixot foglal.

*void felszabaditas(double** tomb, int sorszam)* : Mátrix foglalást szüntet meg.

*int maxabselem(double** M, int oszlopind, int elemsz)* : Invertálás során megkeresi a főelemet.

*double** foelemcsere(double** M, int oszlopind, int elemsz)* : Elvégzi a főelemcserét.

*double** inverz(double** M, int elemsz)* : Kiszámolja a mátrix inverzét a fentiek használatával.

string mod2tomb(int elemsz)* : Megad egy indexhalmazt modulo 2, hogy könnyű legyen eldönteni az adott indexben melyik a legmagasabb hatvány, ami még osztó.

int meddigupdate(int oszlopind, string oszlopindmod2) : Megmondja a legmagasabb 2 hatvány osztó értékét.

A programba nem tettem gyorsmátrixszorzó algoritmust, mivel önmagában is elég komoly feladat lenne megvalósítani egy olyat. A naiv szorzást alkalmaztam, de az algoritmus leírta struktúra szerint.

Miután a *psgraf.cpp*-vel előkészítettem a véletlen elemekkel feltöltött A „adjacenciamátrixot", ebből már a programban nyertem invertálás során B-t. Ezután elkészítettem a `mod2tomb(n)` hívással az indexek értékét modulo 2. Egy V n elemű tömben készültem gyűjteni azokat a sorindexeket, melyek a már szerepelt oszlopokban található, kiválasztásra került elemekhez tartoztak. Kezdetben üresnek tekintettem, és az iteráció számától tettem függővé az aktuális méretét. Ez a V kettős funkciójú volt, egyrészt mert egy $(i, V[i])$ párosítás élt jelölt, másrészt mert a kisupdatehez szükség volt arra, hogy mely sorindexek szerepeltek korábban, és azok milyen sorrendben voltak.

Ezenkívül egy S, n elemű tömben tartottam számon azokat a sorokat (kezdetben ez 1-től n-ig volt feltöltve), amikhez tartozó pontok még nem voltak párosításban. Attól függően, hogy hanyadik oszlop következett, mindig tudtam S és V aktuális méretét. Az elemek sorindexére ezekkel a tömbökkel hivatkozva mostantól úgy tekintjük, mintha minden lépésben a főátlóból választanánk a párosítás elemet. Még egyszer a csel: ami a kiválasztott $B_{i,i}$ elem fölött van az eddigi ábráinkon, ahhoz a program szerint $B[V[k]][i]$-ként $k < i$, ami alatta, ahhoz $B[S[k]][i]$-ként $k < i$ férünk hozzá. S-et úgy csökkentjük, hogy az indexek nagyságsorrendjét megtartjuk. V-be pedig a kiválasztás sorrendjében kerülnek a sorindexek.

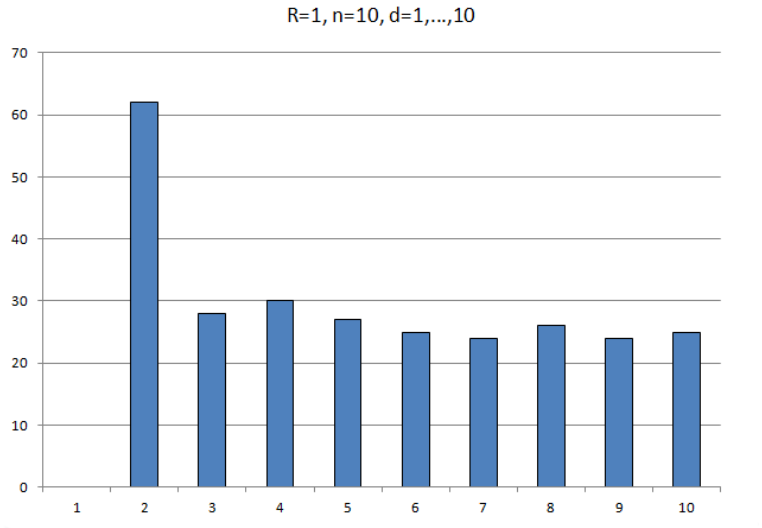
A futás során lépdelünk az oszlopokon. Adott j oszlopon állva megnézzük, hogy mely sorokban van nemnulla A-ban, és 10^{-10}-nél nagyobb érték B-ben. Ezen sorok közül véletlen választunk egyet. Kiszámoljuk a `meddig=meddigupdate(j,indexekmod2[j])` révén, hogy hány oszlopot kell frissíteni a j utániak közül. (Ebben segít a j modulo 2 ábrázolása, amit az `indexekmod2[j]`-ben tároltunk.) A kijelölt oszlopok j. sorai feletti kismátrixsszal, a kisupdatekkel kezdünk. Itt ismét a `meddig2=meddigupdate(i,indexekmod2[i])` segít abban, hogy az i. sor alatt hány sor frissítése következik (még kisebb mátrix). Akár a kisupdate-et, akár az update-et csináltam,

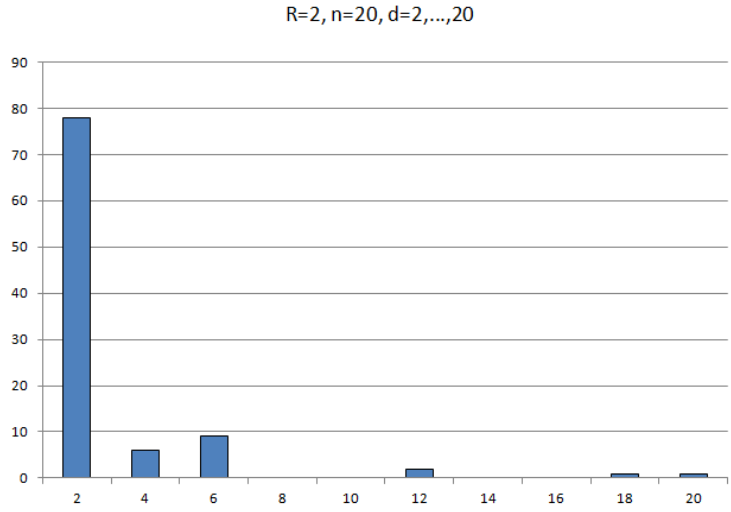
elég volt a `meddig2` és `meddig` ismerete ahhoz, hogy tudjam, melyik elemekre hivatkozzam. A kisupdate-ben mindig a V tömb soraiból vettem a két szorzó mátrix elemeit, a nagyupdate-nél az egyik mátrix S-ben volt, a másik V-ben. Így meg tudtam valósítani az indexek játékával a frissítéseket.

```
//kisupdate
for(int k=j+1-meddig+1; k<j+1; k++){
 int meddig2=meddigupdate(k-1,indexekmod2[k-1]);
 for(int a=k; a<k+meddig2; a++){
  for(int b=j+1; b<j+updoszlop+1; b++){
   for(int h=k-meddig2; h<k; h++){
    B[V[a]][b]=B[V[a]][b]-(B[V[h]][b]*B[V[a]][h])/B[V[h]][h];
   }
  }
 }
}

//update
for(int k=j+1; k<j+updoszlop+1; k++){
 for(int l=0; l<n-j-1; l++){
  for(int h=j+1-meddig; h<j+1; h++){
   B[S[l]][k]=B[S[l]][k]-(B[S[l]][h]*B[V[h]][k])/B[V[h]][h];
  }
 }
}
```

A keletkezett *gausselim.cpp* azt mutatta, hogy a kimondott hibatételnél sokkal erősebb is igaz, az algoritmus $1 - n/R$-nél sokkal nagyobb valószínűséggel ad teljes párosítást vissza. $R = i$, $n = 10i$, $d = 1i, 2i, ..., 10i$-re futtattam az algoritmust $i = 1, 2, 3$ esetén, minden esetben 100-szor és az alábbi eredmény született:





Az $i = 3$ esetet viszont már kirajzolni sem érdemes, mert az összesen 1000 futásból egy se volt hibás, azaz minden esetben találtunk teljes párosítást.

4. fejezet

Súlyozott párosítás páros gráfban gyorsan

Ebben a fejezetben a fő eredmények ismertetésére összpontosítunk majd, a bizonyításoktól eltekintünk.

1. Definíció. *Maximális súlyú teljes párosítás: Legyen $G = (V; E; w)$ súlyozott páros gráf, ahol $V = (X \cup Y)$ a pontok, $E \subseteq V \times V$ az élek halmaza, $w : E \longrightarrow \{0, 1, ..., W\}$ valamely $W \in \mathbb{N}$-nel nemnegatív egész súlyfüggvény. Keressük azt a párosítást, amelyre a $w(M) = \sum_{e \in M} w(e)$ érték maximális.*

Az egyszerűség kedvéért mostantól rövidítsük a maximális elemszámú párosítást előállító algoritmus *MM(G)*-nek, az optimális élszám értéket *optMM(G)*-nek, a maximális súlyú párosítást előállítót *MWM(G)*-nek, *optMWM(G)* optimum értékkel. Legyen $|V| = n$, $|E| = m$, ω továbbra is a gyorsmátrixszorzó algoritmus exponense. Ezekkel a jelölsekkel élve, a korábbi, páros gráfokra specializált MM(G) algoritmusok gyorsasága a [39] cikk alapján:

$O(\sqrt{n}m)$	Hopcroft, Karp (1971) [23]; Karzanov (1973) [24]
$O(\sqrt{n}m \log_n(n^2/m))$	Feder, Motwani (1991) [25]; Goldberg és Kennedy (1997) [26]
$\mathbf{O(n^\omega)}$	**Mucha és Sankowski (2004)** [27]; Harvey (2006) [28]
várható értékben $\mathbf{O(n \log n)}$	**Spielman (2012)** [1]

További jelölést is eszközlünk: legyen $W \in \mathbb{N}$ a maximális súlyú él súlya, az arra szükséges idő, hogy n ponton, m él és W maximális súly mellett megtaláljuk a legrövidebb utat irányított gráfban pedig $SP_+(n, m, W)$. Az eddigi legjobb eredmények

$MWM(G)$-re páros gráfban a [39], [35] cikkek alapján:

$O(n^4)$	Khun (1955) [17]; Munkers (1957) [18]
$O(n^2m)$	Iri (1960) [19]
$O(n^3)$	Dinic, Kronrod (1969) [20]
$O(nm + n^2 \log n)$	Edmonds, Karp (1970) [21]
$O(nSP_+(n, m, W))$	Edmonds, Karp (1970) [36]; Tomizawa (1971) [37]
$O(n^{3/4}m \log W)$	Gabow (1983) [22]
$O(W\sqrt{n}m)$	Gabow (1983) [29]; Kao, Lam, Sung és Ting (1999) [30]
$O(\sqrt{n}m \log(nW))$	Gabow, Tarjan (1988, 1989) [31], [32]; Goldberg (1993) [33]
$O(n^{2.5} \log(nW)(\frac{\log\log n}{\log n})^{1/4})$	Cheriyan, Mehlhorn (1996) [38]
$\mathbf{O(\widetilde{W}\sqrt{n}\frac{\log(n^2/(\widetilde{W}/W))}{\log n})}$	**Kao, Lam, Sung és Ting (2001)** [30]
$\mathbf{O}(Wn^\omega)$	**Sankowski (2006) [35]**
$\mathbf{(1-\epsilon)}$**-approximáció** $\mathbf{O(m\epsilon^{-2}\log^3 n)}$	**Duan és Pettie (2010)** [12]
$\widetilde{O}(Wn^\omega)$	Cygan, Gabow és Sankowski (2012) [14]
$O(\sqrt{n}m \log W)$	Duan, Su (2012) [34]

Egy jelölést még nem tisztáztunk korábban. Azt írjuk, hogy $f(n) = \widetilde{O}(g(n))$, ha $f(n) = O(g(n) \log^k n)$ konstans k-ra.

A három kiemelt, ezen utóbbiak sorát gazdagító algoritmust vesszük sorra, melyeket egészen eltérő alapgondolatuk miatt tartotta érdekesnek a szakdolgozat írója. Lényegében az ezeket leíró cikkek fő eredményeinek magyar nyelvű leírását eszközöltük.

4.1. Maximális súlyú párosítás a gyors mátrixszorzás idejében

Amire ez az algoritmus elsősorban alapoz, az az alábbi tétel:

11. Tétel. *Legyen $A \in K[x]^{n\times n}$-es egy d-fokú polinommátrix $b \in K[x]^{n\times n}$ pedig egy ugyanilyen fokú polinomvektor. Ekkor $det(A)$ és az $A^{-1}b$ racionális rendszer megoldása $\widetilde{O}(n^\omega d)$ időben programozható.*

Hasonlóan tehát a súlyozatlan esetben működő Gauss-eliminációs megközelítéshez, itt is az adjacenciamátrix inverz mátrixának segítségével keresünk majd páro-

sításéleket. Az algoritmus maga használni is fogja a korábbit (lásd 3. fejezet).

Mostantól úgy tekintjük, hogy a kiindulási gráfban van teljes párosítás. Emlékezzünk vissza: teljes párosítás esetén az kell, hogy a kiválasztás után fennmaradó gráfban továbbra is legyen teljes párosítás. Ez azzal ekvivalens, hogy a megmaradt mátrix nemszinguláris, vagyis determinánsa nemnulla. Ugyan a determinánsban exponenciálisan sok tag lehet, de ezen segít a Zippel–Schwartz lemma, lásd: 1. lemma.

Ebből adódik az alábbi következmény:

12. Tétel. *Legyen p egy $(1+c)\log n$ hosszú prímszám ($c>0$). Ha egy n-fokú nemnulla polinomot kiértékelünk a $\mathbb{Z}_p$-ből véletlen kiválasztott elemeken, akkor annak az esélye, hogy nullát kapunk, holott az elemek és polinom nemnullák voltak (hamis nulla), legfeljebb $1/n^c$.*

A standard RAM modellnek megfelelően feltesszük, hogy a szavak hossza $O(\log(n))$-es. Ezzel a feltevéssel elértük, hogy a modulo p véges testműveletek (kivéve az osztást) konstansidejűek legyenek. Az osztás megvalósítása viszont $O(\log(n))$ alatt megvalósítható, és nem is lesz belőle túl sok.

Az algoritmus első lépése előtt még bevezetjük a *pontfedés* fogalmát, valamint felelevenítjük a dualitás tételt: Legyen $C : X \cup Y \longrightarrow \mathbb{R}$ olyan függvény, melyre teljesül $C(x)+C(y) \geq w(x,y)$. Egy ilyen pontfedés *értéke*: $\sum_{u\in X\cup Y} C(u) = \widetilde{C}$. A minimális értékű pontfedés problémája duális kapcsolatban áll a maximális súlyú párosítás problémájával:

13. Tétel. *Az alábbi állítások ekvivalensek:*

1. *C minimális értékű fedés és M maximális súlyú teljes párosítás*
2. *$\widetilde{W}(M) = \sum_{u,v\in M} w(u,v) = \sum_{u\in X\cup Y} C(u) = \widetilde{C}$*
3. *$C(u)+C(v) = w(u,v)$ áll $\forall u,v \in M$.*

Nevezzük *egyenlőségi részgráfnak* egy $G=(X\cup Y; E)$ w-súlyozott gráf, és egy $C : X\cup Y \longrightarrow \{0,1,...\}$ pontfedés esetén azt a $G_c = (X\cup Y; E')$ gráfot, melynek az éleire $E' = \{(x,y) : (x,y)\in E,\ c(x)+c(y) = w(x,y)\}$. Erre igaz az Egerváry tételéből közvetlenül levezethető lemma, miszerint:

3. Lemma. *M teljes párosítás a G_c egyenlőségi részgráfban, ha maximális súlyú párosítás a G gráfban.*

Vagyis az az idea, hogy ha tudunk $\mathcal{O}(Wn^\omega)$ időben minimális súlyú pontfedést adni G-re, akkor elég lesz G_c-ben keresnünk teljes párosítást a korábbról ismert $O(n^\omega)$ idejű algoritmusunkkal, máris megvan a maximális súlyú párosításunk G-ben. De a redukció ezzel koránsem ért véget. Az eljárás többlépcsős visszavezetést alkalmaz:
Egyenlőségi részgráf→Pontfedés→Legrövidebb utak→Párosítás súlyok→Mátrix polinomok.

Iri ötlete nyomán haladunk tovább. A pontfedést felírjuk legrövidebb útkeresésként. Egyelőre kicsit furcsának tűnhet, hogy úgy adjuk meg a következő definíciót, mintha tudnunk kéne már a maximális súlyú teljes párosítást. Nemsokára fény derül rá, hogy annak ismeretére miért is nem lesz szükség.

2. Definíció. *Legyen M maximális súlyú teljes párosítás. Definiáljunk egy $D = (X \cup Y \cup \{s\}; A)$ súlyozott digráfot a megfelelő irányított élekkel úgy, hogy ha $(x, y) \in E$, akkor $(x, y) \in A$ és ha $(x, y) \in M$, $(y, x) \in A$, húzzuk be az összes (s, y) típusú élt is, a súlyok alakulása pedig:*

$$w_D(x, y) = \begin{cases} 0 & \text{minden } (s, y) \\ w(x, y) & \text{ha } (x, y) \in M \\ -w(x, y) & \text{különben.} \end{cases}$$

Jelölje $dist_D(x, y)$ az $x \in X$ és $y \in Y$ pontok távolságát D-ben. Ekkor igaz:

4. Lemma. *Az a c, amire fennállnak: $c(x) = dist_D(s, x)$, ha $x \in X$ és $c(y) = -dist_D(s, y)$, ha $y \in Y$, minimális súlyú pontfedést ad G-ben.*

Tehát a visszavezetés megtörtént és jogos. A következő lépcsőfok a minimális utak megadása maximális súlyú párosítások súlyaként. A maximális súlyokat G variánsaiból nyerjük.

Teikntsük megint a w-súlyozott G páros gráfunkat. Bővítsük két további ponttal: $s = x_{n+1} \in X$, $t = y_{n+1} \in Y$, kössük össze s-t minden $y \in Y$ ponttal 0-súlyúan, és a t-t szintén egy 0 súlyú éllel kössük hozzá valamelyik $x \in X$ elemhez. Amit kaptunk azt az elkövetkezőkben $G(x)$-nek fogjuk hívni, a belőle nyerhető maximális súlyú teljes párosítást pedig $M(x)$-nek (ilyen van, mert G-ben volt és az s és t is párbaállítható). A későbbiekre való tekintettel: $G(*)$ jelölje azt a gráfot, ahol az összes $(t, x),\ x \in X$ típusú él be van húzva.

5. Lemma. *Ha M maximális súlyú teljes párosítás G-ben, akkor: $dist_D(s, x) = w(M) - w(M(x))$ minden $x \in X$.*

Azzal a külön észrevétellel, hogy ha X-en meg tudjuk mondani a minimális pontfedést, akkor Y-on az $c(y) := \max_{x \in X}\{w(x,y) - c(x)\}$ képlettel számolható a többi, elkészült még egy redukció. Vegyük észre, hogy a lemma nem használja fel magát az M-et és $M(x)$-et, csupán a súlyukat.

Az utolsó körben az $M(x)$ súlyát nyerjük ki mátrix polinomokból. Általában egy maximális súlyú teljes párosítás súlya páros gráfban a következőképpen áll elő, definiálva egy $z_{i,j}$ változókból álló mátrixot:

$$\widetilde{B}(G,r)_{i,j} = \begin{cases} z_{i,j} r^{w(x,y)} & \text{ha } (x,y) \in E \\ 0 & \text{különben.} \end{cases}$$

6. Lemma. *r foka a $det(\widetilde{B}(G,r))$-ben a G-beli maximális súlyú párosítás súlyának felel meg. Továbbá minden konstans együttható nemnulla $det(\widetilde{B}(G,r))$-ben $\mathbb{Z}_p$ véges test felett $p \geq 2$ esetén.*

Még némi utómunkálat után látjuk, hogy a fentiek következménye: A maximális súlyú teljes párosítás súlya nagy valószínűséggel előáll $\widetilde{O}(Wn^\omega)$ időben. Tehát $w(M)$-mel már nincs gondunk, kéne még $w(M(x))$ minden $x \in X$-re. Ezekről a következő lemma gondoskodik:

7. Lemma. *Adott egy w-súlyozott $G = (X \cup Y; E)$ páros gráf. Legyen $z = det(\widetilde{B}(G(*),r))\widetilde{B}(G(*),r)^{-1}e_{n+1}$. Ekkor $w(M(x_i) = deg_r(z_i))$.*

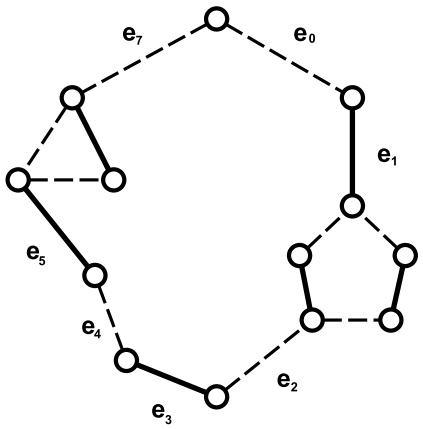
Összegezve az eddigieket: azt a sémát követjük, hogy meghatározzuk a $w(M)$-et és a $w(M(x))$ értékeket az adjacenciamátrixból nyert véletlen mátrixból. Általuk meghatározzuk a $dist_D(s,x)$ távolságokat. Innen már a $c(x)$-ok és $c(y)$-ok számítása elemi, megadva a G minimális súlyú pontfedését. És a c-vel és eredti w-vel definiált egyenlőségi részgráfban keresünk teljes párosítást.

4.2. Maximális súlyú párosítás közelítése majdnem lineáris időben

A megjelölt cikkben tárgyalt algoritmusnak azzal a változatával foglalkozunk csak, amely pozitív egész súlyokkal dolgozik, ahogy azt a fejezet elején leszögeztük. Ennek futásideje nem a táblázatban feltüntetett érték, hanem $O(Wm/\epsilon)$. Ezen felül általános gráfokon működik, nem kritériuma a párosság.

Az algoritmus fenntart egy Ω dinamikus halmazt, az *egymásba ágyazott kelyhek* halmazát. A kelyheket induktív módon definiáljuk:

- Ha $v \in V$, akkor $\{v\}$ kehely, de nem eleme Ω-nak.
- Egy páratlan hosszú $(A_0, A_1, ...A_l)$ esetén $B = \bigcup_i A_i$ kehely, ha
  - az $\{A_i\}_i$ halmazok kelyhek, ezenfelül
  - létezik olyan $e_0, e_1, ...e_l$ élsorozat, melyre $e_i \in A_i \times A_{i+1} \pmod{l+1}$ és $e_i \in M$, ha i páratlan.



4.1. ábra. Példa kehelyre.

Az Ω reprezentálására alkalmas lehet egy fákból álló erdő, amiben a pontok gyermekei a kelyheket alkotó kelyhek. Az ilyen fák gyökereire gondolunk, ha a továbbiakban a *gyökérkehely* kifejezést találnánk használni. Gráf *összehúzottja* alatt annak az eredményét értjük, amit a gyökérkelyhek 1 ponttá zsugorítása von maga után. Egy B *gyökérkehely eltüntetése* pontjainak az erdőből való törlésével, és az összehúzott gráfban a B $A_0, A_1, ..., A_l$ eredeti pontjaira történő cseréjével jár.

2. Észrevétel. *Ha M párosítás G-ben, akkor az összehúzott gráfban az, ami marad belőle, az is párosítás. Az összehúzott gráf egy alternáló útja kiterjeszthető egy alternáló úttá G-ben.*

Az Ω-n kívül fenntartunk még két potenciálfüggvényt: $q : V \to \mathbb{R}$, $z : \Omega \to \mathbb{R}$, amik az M-re nézve folyamatosan ki fognak elégíteni bizonyos követelményeket. Belőlük származtatunk egy további függvényt is: $(x, y) \in E$-re legyen $qz(x, y) = q(x) + q(y) + \sum_{B \in \Omega, (u,v) \in E(B)} z(B)$. Legyen k fix pozitív egész...

Fenntartandó tulajdonságok (relaxált komplementaritási feltételek):

1. $z(B) \geq 0$ minden $B \in \Omega$ és $z(B) > 0$ a gyökérkelyhen.

2. $qz(e) \geq w(e) - 1/k$ minden $e \in E$.

3. $qz(e) \leq w(e)$, ha $e \in M$, vagy $e \in E(B)$ valamely $B \in \Omega$.

4. A párosítás által szabadon hagyott pontokra az q-értékek egyenlők és nagyobbak, mint a párosított pontokra.

A 8. lemma azt mutatja meg, hogy ha a szabad pontokon az q-értékeket 0-ra redukáljuk, akkor a fenntartandó tulajdonságok garantálják, hogy elég jó párosításhoz jussunk.

8. Lemma. *Tegyük fel, hogy a fenntartandó tulajdonságok teljesülnek M párosításra, ahol az q-értékek a szabad pontokon 0-ák. Legyen M' egy másik párosítás, talán maga az $optMWM$. Ekkor:*

1. $w(M) \geq w(M') - |M'|/k$.

2. *M egy $(1 - 1/k) - optMWM$*

3. *Legyen $P \subseteq M \oplus M'$ alternáló kör, avagy alternáló út, aminek a végét M nem fedi. Ekkor $w(M \cap P) \geq w(M' \cap P) - |M' \cap P|/k$.*

Az algoritmus nagy vonalakban

Kezdetben: $M = \emptyset$, $\Omega = \emptyset$, $q(v) = 0$ minden $v \in V$, így teljesülnek a fenntartandó tulajdonságok.

A futás során: az algoritmus újra és újra növelő utak egy halmazát találja (a választható élekből), kelyheket épít és töröl, majd finomításokat végez q-n és z-n úgy, hogy ne rontsa el a fenntartandó tulajdonságokat és növelje a választható élek számát. De mitől is lesz egy él választható?

3. Definíció. *Az e él választható, ha a következők valamelyike teljesül rá:*

- *$e \in M$ és $qz(e) = w(e)$,*
- *$e \notin M$ és $qz(e) = w(e) - 1/k$,*
- *$e \in E(B), B \in \Omega$.*

Legyen G' a választható élek gráfja, H pedig a G'-ből az Ω-beli kelyhek összehúzása után keletkező gráf. G' és H kezdetben V és $\emptyset$. Egy iterációban háromféle lépésünk lesz: növelés, virágzás és összehúzás, a duál beállítása.

Leállás: Akkor még nem, ha az M teljes, csak, ha a q-érték a szabad pontokon eléri a 0-t.

1. lépés: Keressük a növelő utak $\max \Psi$ halmazát H-ban. $M \leftarrow M \oplus (\cup_{P \in \Psi} P)$. Frissítjük G'-t és H-t.

2. lépés: Legyenek $V_{out} \subseteq V(H)$ azon gyökérkehely pontok, amik elérhetőek a szabad pontokból páros hosszú alternáló úton. Legyen Ω' az egymásba ágyazott kelyhek maximális elemszámú halmaza V_{out}-on (Ha $(u, v) \in E(H)$, akkor u és v ugyanabba a kehelybe esik). Legyen $V_{in} \subseteq V(H)/V_{out}$ azok, amik páratlan hosszú alternáló utakon érhetőek el szabad pontokból. Állítsuk be a $z(B) \leftarrow 0$ értéket minden $B \in \Omega'$ esetén, legyen még $\Omega \leftarrow \Omega \cup \Omega'$. Frissítsük G'-t és H-t.

3. lépés: Legyen $\hat{V}_{in}, \hat{V}_{out} \subseteq V$ azon eredeti pontok, amik V_{in}-nek és V_{out}-nak is részei. q-t és z-t állítsuk át:

   - $q(v) \leftarrow q(v) - 1/2k$, ha $v \in \hat{V}_{out}$,
   - $q(v) \leftarrow q(v) + 1/2k$, ha $v \in \hat{V}_{in}$,
   - $z(B) \leftarrow z(B + 1/k)$, ha $B \in \Omega$ gyökérkehely és a pontjai $\hat{V}_{out}$-beliek,
   - $z(B) \leftarrow z(B - 1/k)$, ha $B \in \Omega$ gyökérkehely és a pontjai $\hat{V}_{in}$-beliek.

4.3. A dekompozíciós tétel maximális súlyú párosítás keresésére

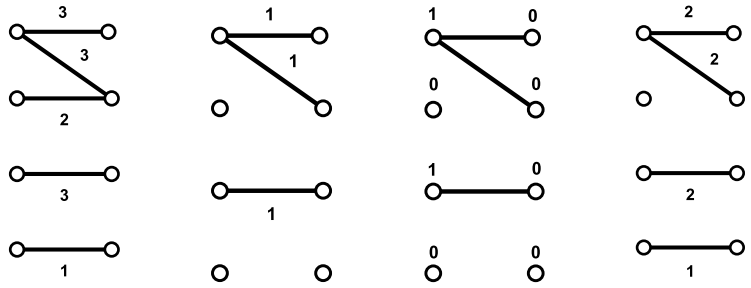
A cikk írói olyan algoritmust mutatnak be, amely G, izolált pontoktól mentes párosgráfra, $\widetilde{W}$ összélsúlyra $O(\sqrt{n}\widetilde{W}/k(n, \widetilde{W}/W))$ futásidejű, a $k(x, y) = \frac{\log x}{\log \frac{x^2}{y}}$ jelöléssel. Ami plusz eredményként kipotyog, hogy ha adott G-re már megvan annak maximális súlyú párosítása, akkor ebből $G - \{u\}$-ban maximális súlyú párosítás keresésére már csak $O(\widetilde{W})$ időt kell fordítani. A folyamat során a dualitás tételnek itt is lesz szerepe, de most a fedésről feltesszük, hogy csak pozitív értékeket vehet fel C:$X \cup Y \to \{0, 1, ...\}$.

Legyen $h \in [1, W]$. A W darab iteráció közül a h. során két páros gráfot képezünk, az egyiket jelölje G_h, a másikat $G_h^{\triangle}$: G_h-ba kerüljön az (u, v) él, ha annak súlya G-ben, $w(u, v) \in [W - h + 1, W]$, a súlya legyen ekkor $w(u, v) - (W - h)$. Legyen C_h a G_h gráf minimális súlyú pontfedése. $G_h^{\triangle}$-be akkor kerüljön be az $(u, v) \in E(G)$ él, ha arra $w(u, v) - C_h(u) - C_h(v) > 0$, az új súly pedig $w(u, v) - C_h(u) - C_h(v)$ legyen.

Például $W > 1$ esetén G_1 a maximális W súlyú élekből fog állni, amik súlya G_1-ben 1-re redukálódik és a hozzá tartozó pontfedésben 0 vagy 1 C-értékek lesznek minden ponton. $G_h^{\triangle}$-be úgy kerülnek be már az élek, hogy a maximális súlyúak súlya csökkentve lett 1-gyel.

14. Tétel. *Az alábbi állítások ekvivalensek:*

1. *C minimális értékű fedés és M maximális súlyú párosítás*
2. *$\widetilde{W}(M) = \sum_{u,v \in M} w(u,v) = \sum_{u \in X \cup Y} C(u) = \widetilde{C}$*
3. *$\forall u : C(u) > 0$ u párosított M-ben és $C(u) + C(v) = w(u,v)$ áll $\forall u, v \in M$.*



4.2. ábra. G, G_h, C_h, $C_h^{\triangle}$.

15. Tétel. *$optMWM(G) = optMWM(G_h) + optMWM(G_h^{\triangle})$, sőt kiemelve:*

$$optMWM(G) = optMM(G_1) + optMWM(G_1^{\triangle}).$$

Az algoritmus rekurzív szerkezetű és a minimális súlyú pontfedés, valamint a minimális elemszámú párosítás szubrutinját használja fel. Az első algoritmus még nem magát a maximális párosítást adja meg, hanem a hozzá tartozó súlyt:

4. algoritmus. DekompMWM(G)

1. G_1 létrehozása G-ből.
2. $optMM(G_1)$ megadása.
3. C_1 legyen G_1 minimális fedése.
4. $G_1^{\triangle}$ megadása G-ből és C_1-ből.

if $G_1^{\triangle} = \emptyset$ **then**
  return $optMM(G_1)$;
else
  return $optMM(G_1)$+*DekompMWM(*$G_1^{\triangle}$*)*;
end

A szubrutinok közül tekintsünk el az MM(G) algoritmus ismertetésétől, nézzük csak a minimális értékű pontfedést meg. Ezt $O(\sqrt{n}\widetilde{W}/k(n,\widetilde{W}/W))$ időben meg tudjuk valósítani, és az eredmény lefordítása a maximális súlyú párosítás nyelvére $O(\sqrt{n}m/k(n,m))$ időt igényel. Egy lemmát használunk fel csupán:

9. Lemma. *Tegyük fel, hogy G_h, C_h és $G_h^\triangle$ definiáltak a fenti módon. Legyen $C_h^\triangle$ tetszőleges minimális értékű fedése $G_h^\triangle$-nek. Ekkor, ha D olyan függvény $V(G)$-n, melyre $D(u) = C_h(u) + C_h^\triangle(u)$ minden $u \in V(G)$ esetben, akkor D minimális súlyú fedése G-nek.*

---

5. algoritmus. MinFedés(G)

---

1. G_1 létrehozása G-ből.
2. C_1 megadása G_1 minimális fedéseként (súlyozatlan eset).
3. $G_1^\triangle$ megadása G-ből és C_1-ből.

if $G_1^\triangle = \emptyset$ **then**
  return C_1;
else
  $C_1^\triangle :=$ *MinFedés*$(G_1^\triangle)$;
  D-t válasszuk úgy, hogy $D(u) = C_1(u) + C_1^\triangle(u) \quad \forall u \in V(G)$;
  return D;
end

A maximális súlyú párosítás visszanyerése D fedésből pedig:

---

6. algoritmus. FedésbőlMaxps(G,D)

---

1. Legyen H azon részgráfja G-nek, amire
   $E(H) = \{(u,v)|\ (u,v) \in G,\ w(u,v) = D(u) + D(v)\}$.
2. Duplikáljuk H-t kétszer: H minden u pontjának feleltessünk meg egy H^a-beli pontot, ezt jelöljük u^a-val és egy H^b-belit, jelölve u^b-vel (az éleket is megtartjuk).
3. Képezzük a H^{ab} uniót: $V(H^{ab}) := V(H^a) \cup V(H^b)$ és
   $E(H^{ab}) = E(H^a) \cup E(H^b) \cup \{(u^a u^b)|\ u \in V(H),\ D(u) = 0\}$.
4. Keressünk maximális elemszámú K párosítást H^{ab}-ben.

return $K^a = \{(u,v)|(u^a, v^a) \in K\}$;

Irodalomjegyzék

[1] Ashish Goel, Michael Kapralov, Sanjeev Khanna: Perfect Matchings in $O(n \log n)$ Time in Regular Bipartite Graphs STOC, 2010.

Daniel A. Spielman: Randomized Perfect Bipartite Matching, Design and Analysis of Algorithms, Lecture Note, March 22, 2012.

[2] Marcin Mucha, Piotr Sankowski: Maximum Matchings via Gaussian Elimination [Extended Abstract], Proceedings of the 45th annual IEEE Symposium on Foundations of Computer Science, 0272-5428/04, 2004.

[3] S. Micali, V. V. Vazirani: An $O(\sqrt{|V||E|})$ algorithm for finding maximum matching in general graphs, Proceedings of the 21st annual IEEE Symposium on Foundations of Computer Science, Pages 17-27, 1980.

[4] N. Blum: A new approach to maximum matching in general graphs, In Proc. 17th ICALP, Volume 443 of Lecture Notes in Computer Science, Pages 586-597, Springer-Verlag, 1990.

[5] H. N. Gabow, R. E. Tarjan: Faster scaling algorithms for general graph matching problems, J. ACM, 38(4):815-853, 1991.

[6] D. Coppersmith, S. Winograd: Matrix multiplication via arithmetic progressions, In Proc. of the 19th ACM conference on Theory of computing, Pages 1-6, ACM Press, 1987.

[7] W. T. Tutte: The factorization of linear graphs, J. London Math. Soc., 22:107-111, 1947.

[8] L. Lovász: On determinants, matchings and random algorithms, In L. Budach, editor, Fundamentals of Computation Theory, Pages 565-574, Akademie-Verlag, 1979.

[9] R. Zippel: Probabilistic algorithms for sparse polynomials, In International Symposium on Symbolic and Algebraic Computation, Volume 72 of Lecture Notes in Computer Science, Pages 216-226, Springer Verlag, Berlin, 1979.

[10] J. Schwartz: Fast probabilistic algorithms for verification of polynomial identities, Journal of the ACM, 27:701-717, 1980.

[11] M. O. Rabin, V. V. Vazirani: Maximum matchings in general graphs through randomization, Journal of Algorithms, 10:557-567, 1989.

[12] Ran Duan, Seth Pettie: Approximating Maximum Weight Matching is Near-linear Time, FOCS: 673-682, 2010.

[13] Chien-Chung Huang, Telikepalli KAvitha: Efficient Algorithms for Maximum Weight Matchings in General Graphs with Small Edge Weights, SODA: 1400-1412, 2012.

[14] Marek Cygan, Harold N. Gabow, Piotr Sankowski: Algorithmic Applications of Baur-Strassen's Theorem: Shortest Cycles, Diameter snd Matchings 2012.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Új algoritmusok, Scolar Kiadó

[16] `http://en.wikipedia.org/wiki/Matrix_multiplication`

[17] H. W. Kuhn: The hungarian method for the assignment problem, Naval Research Logistics Quarterly 2(1955)83-97.

[18] J. Munkers: Algorithms for the assignment and transportation problems, Journal of SIAM 5(1)(1957)32-38.

[19] M. Iri: A new method for solving transportation-network problems, Journal of the Operations Research Society of Japan 3(1960)27-87.

[20] E. A. Dinic, M. A. Kronrod: An algorithm for the solution of the assignment problem, Sovie Mathematics Doklady 10(1969)1324-1326.

[21] J. Edmonds, R. M. Karp: Theoretical improvements in algorithmic efficiency for network flow problems, Journal of ACM 19(2)(1972)248-264.

[22] H. N. Gabow: Scaling algorithms for network problems, Journal of Computer and System Sciences 31(2)(1985)148-168.

[23] J. Hopcroft, R. Karp: An $n^{5/2}$ algorithm for matching and matroid problems, SIAM J. Comput. 2:225-231, 1973. Conference Version in 12th Symposium on Switching and Automata Theory.

[24] A. Karzanov: O nakhozhedenii maksimal'nogo vida i nekotorykh prilozheniyakh, Matematicheskie Voprosy Upravleniya Proizvodstvom 31(1): 81-94, 1973.

[25] T. Feder, R. Motwani: Clique partitions, graph compression and seeding-up algorithms, J. of Computer and System Sciences 51(2): 261-272, 1995. Conference Version in STOC 1991.

[26] A. Goldberg, R. Kennedy: Global price updates help, SIAM Jounal on Discrete Mathematics 10: 551-572, 1997.

[27] M. Mucha, P. Sankowski: Maximum matchings via Gaussian elimination, 45st FOCS: 248-255, 2004.

[28] N. Harvey: Algebraic algorithms for matching and matroid problems, SIAM J. Comput., Conference Version in FOCS 2006.

[29] H. Gabow: Scaling algorithms for network problems, 24th FOCS: 248-257, 1983.

[30] M.-Y. Kao, T. W. Lam, W.-K. Sung, H.-F. Ting: A decomposition theorem for maximum weight bipartite matchings, SIAM J. Comput. 31(1): 18-26, 2001.

[31] H. Gabow, R. Tarjan: Almost-optimum speedups of algorithms for bipartite matching and related problems, 20th STOC, 514-527, 1988.

[32] H. Gabow, R. Tarjan: Faster scaling algorithms for network problems, SIAM J. Comput. 18: 1013-1036, 1989.

[33] A. Goldberg: Scaling Algorithms for the Shortest Paths Problem, SIAM J. Comput. 24(3): 494-504, 1995. Conference Version in SODA 1993.

[34] R. Duan, H.-H. Su: A Scaling Algorithm for Maximum Weight Matchings in Bipartite Graphs, 23rd SODA, 2012.

[35] P. Sankowski: Maximum weight bipartite matching in matrix multiplication time, Theoretical Computer Science 410: 4480-4488, 2009. Conference Version in ICALP 2006.

[36] J. Edmonds, R. Karp: Theoretical improvements in algorithmic efficiency for network flow problems, Combinatorial Structures and their Applications: 93-96, 1970.

[37] N. Tomizawa: On some techniques useful for solution of transportation network problems, Networks 1: 173-194, 1971.

[38] J. Cheriyan, K. Mehlhorn: Algorithms for Dense Graphs and Networks on the Random Access Computer, Algorithmica 15: 521549, 1996.

[39] Chien-Chung Huang, Telikepalli Kavitha: Efficient Algorithms for Maximum Weight Matchings in General Graphs with Small Edge Weights, IMPECS