Eötvös Loránd Tudományegyetem
Természettudományi Kar

BSc Szakdolgozat

Stabil házasítás: Közelítő algoritmusok implmentálása és tesztelése

írta: Varga Brigitta

Alkalmazott matematikus szakirány

Témavezető: Király Zoltán egyetemi docens

Számítógéptudományi Tanszék

Budapest
2010

Tartalomjegyzék

1. Bevezetés **4**
1.1. Definíciók és alapfogalmak 4
1.1.1. Optimalizálási feladatok 5

2. Stabil házasítás **7**
2.1. A probléma matematikai modellje 8
2.2. Gale-Shapley algoritmus (GS) 8
2.3. Férfiak listája szigorú 9
2.4. Általános eset 10

3. Kórház/Rezidens probléma **12**
3.1. A matematikai modell 12
3.2. Szigorú listával a rezidensek oldalán 12
3.3. Irving-Manlove algoritmusai 13

4. Implementálás **14**
4.1. Programozási megoldások 14
4.1.1. A gráfok elkészítése és fájlbamentése 15
4.1.1.1. create.cc 16
4.1.1.2. create_k.cc 17
4.1.1.3. create_pri.cc 17
4.1.1.4. create_hr.cc 17
4.1.1.5. create_hr_k.cc 17
4.1.1.6. create_hr_pri.cc 18
4.1.2. Az algoritmusok közös részei 18
4.2. Alternatívák az GS algoritmusra 19
4.2.1. A GS algoritmus, és változatainak megvalósítása 19
4.2.2. GSA1 algoritmus 22
4.2.2.1. rmGS 22
4.2.3. GSA2 algoritmus 23
4.2.3.1. rwGS 23
4.2.4. A GSA1 és GSA2 különböző változatai 23

5. Tesztelés és eredmények **24**
5.1. Stabil házasítás tesztek 24
5.1.1. GSA1 algoritmusok tesztelése 26
5.1.2. GSA2 algoritmusok tesztelése 26
5.2. hrGSA1 27

6. Utószó **29**

1. fejezet

Bevezetés

A szakdolgozatban a Stabil Házasításról (más néven Stabil Párosítás) lesz szó, mely a klasszikus gráfelméleti párosítási probléma kiterjesztése. Ez a probléma sok gyakorlati alkalmazásban előkerül, jelentős szerepet játszik a felsőoktatási felvételi rendszerben, vagy olyan problémákban ahol munkavállalókat kell különböző munkahelyekhez rendelni (pl az egészségügyben illetve a hadseregben). A probléma egyes variánsai különböző egészségügyi transzplantációs protokollokban is hangsúlyt kapnak.

Ebben a fejezetben a definíciókat, és az alapfogalmakat fogjuk tárgyalni.

A második fejezetben a Stabil Házasítás definíciója és speciális esetei szerepelnek, illetve kitérünk a Kórház/Rezidens problémára is, mely a Stabil Házasítás általánosítása. Szó lesz a Gale-Shapley algoritmusról, illetve ennek Király Zoltán által módosított változatairól is. A szakdolgozat célja ezen algoritmusok különböző verzióinak összehasonlítása tesztelések által, hogy átfogó képet kaphassunk arról, hogy milyen inputokra melyik algoritmusok szolgáltatják az optimálishoz legközelebbi megoldást.

A harmadik fejezet az implementálás folyamatát részletezi a programozási megoldásokkal együtt. A programok mind a C++ programnyelven íródtak.

A negyedik fejezet pedig a tesztelés eredményeit tartalmazza. Elsőként a megfelelő input gráfok generálása a feladat, melyeken az algoritmusok közötti különbségek jól láthatóak. Így tehát többféle gráf készítő algoritmus segítségével mutatjuk be a Stabil Házasításra, és a Kórház/Rezidens problémára adott algoritmusok hatékonyságát, különböző táblázatokon át szemléltetve.

Végül pedig egy rövid összefoglalással zárul a szakdolgozat.

1.1. Definíciók és alapfogalmak

Algoritmikus problémák megoldása során gyakran szolgáltatnak jó modellt a gráfok bizonyos objektumok és a köztük lévő kapcsolatok leírásához. A probléma jellegéből adódóan használhatunk irányított vagy irányítatlan, súlyozott vagy súlyozatlan élű, illetve adott esetben páros gráfokat is.

A továbbiakban az irányítatlan páros gráfokkal fogunk foglalkozni, és ezekben keresünk párosítást. Nézzük ezek pontos definícióit:

Definíció (páros gráf): A $G = (L; E)$ gráf egy páros gráf, ha az L csúcshalmaz felbontható két nem üres U és V részre úgy, hogy $U \cap V = \emptyset$, $U \cup V = L$, és ha $(v_1, v_2) \in E$, akkor a v_1 és v_2 csúcsok egyike U-ban, másika pedig V-ben van.

Definíció (párosítás): Legyen $G = (L; E)$ egy tetszőleges gráf. Az E élhalmaz $E' \subseteq E$ részhalmaza G egy párosítása, ha a $G' = (L, E')$ gráfban minden pont foka legfeljebb 1.

Definíció (maximális párosítás): A G gráf egy E' párosítsa maximális, ha G minden E'' párosítására $|E''| \leq |E'|$.

1.1.1. Optimalizálási feladatok

Az A feladat egy optimalizálási probléma, ha minden x példányához tartozik egy $F(x)$ halmaz, a lehetséges megoldások halmaza, és minden $s \in F(x)$ lehetséges megoldásnak van egy pozitív egész $c(s)$ költsége (maximalizálási probléma esetén is költségnek nevezzük, és a $c(s)$ jelölést használjuk). Az optimális költség definíciója $OPT(x) = min_{s \in F(x)} c(s)$ (vagy ha A egy maximalizálási probléma, akkor $OPT(x) = max_{s \in F(x)} c(s)$). Legyen M egy olyan algoritmus, amely minden x példányra egy $M(x) \in F(x)$ lehetséges megoldást szolgáltat. Azt mondjuk, hogy M egy ε relatív hibájú közelítés, ahol $\varepsilon \geq 0$, ha minden x-re

$$\frac{|c(M(x)) - OPT(x)|}{max\{OPT(x), c(M(x))\}} \leq \varepsilon.$$

Mivel a költségek pozitívak, ez az arány mindig értelmes. A nevezőben az $OPT(x)$ helyett azért használjuk a $max\{OPT(x), c(M(x))\}$ kifejezést, hogy a definíció a maximalizálási és minimalizálási esetre szimmetrikus legyen. Ezáltal ε mindkét esetben 0 és 1 közötti értéket vesz fel. Maximalizálási problémánál, egy ε relatív hibájú közelítés olyan megoldást ad, amelyik sohasem kisebb az optimum $(1 - \varepsilon)$-szeresénél. Minimalizálási problémánál a kapott megoldás sohasem nagyobb, mint az optimum $(\frac{1}{1-\varepsilon})$-szerese. Amennyiben létezik $\varepsilon < 1$ konstans relatív hibájú közelítés, akkor c-közelítésről beszélünk, ahol $c = \frac{1}{1-\varepsilon}$.

Minden NP-teljes A optimalizálási probléma esetében a legkisebb olyan ε meghatározása érdekel minket, amelyre még A-nak van polinom idejű ε relatív hibájú közelítése. Olykor nem létezik ilyen legkisebb ε, tetszőlegesen kis hibát el lehet érni a közelítő algoritmusokkal.

Definíció: Az A közelítési küszöbje (approximációs hányadosa) azon $\varepsilon > 0$ számok legnagyobb alsó korlátja, melyekre A-nak létezik polinom idejű ε relatív hibájú közelítő algoritmusa.

Egy optimalizálási probléma esetén a közelítési küszöb 0 (tetszőlegesen jó közelítés lehetséges) és 1 (lényegében nem lehet közelíteni) között lehet.

Definíció: Az A optimalizálási problémához egy polinom idejű közelítő séma (PTAS - polynomial-time approximation scheme) egy algoritmuscsalád, amely minden $1 > \varepsilon > 0$ esetén A minden x példányára egy legfeljebb ϵ relatív hibájú

megoldást ad vissza, és a futási idő az $|x|$ egy (ε-tól függő) polinomjával korlátos. Amennyiben a polinom $\frac{1}{\varepsilon}$-tól is polinomiálisan függ, teljesen polinomiális sémáról beszélünk (FPTAS).

Definíció (APX): Egy probléma APX-beli, ha létezik olyan c konstans, melyre van olyan polinom idejű algoritmus, mely c-közelítő.

Definíció (L-visszavezetés): Legyenek A és B optimalizálási problémák. Az A-nak B-re való L-visszavezetése egy R és S függvényekből álló pár, melyek mindegyike kiszámítható logaritmikus tárral, és rendelkeznek a következő két tulajdonsággal. Elsőként, ha x az A egy példánya, akkor $R(x)$ a B egy olyan példánya, amelynek optimális költségére teljesül, hogy $OPT(R(x)) \leq \alpha \cdot OPT(x)$, ahol α egy pozitív konstans. Másodszor, ha s az $R(x)$ egy tetszőleges megengedett megoldása, akkor $S(s)$ az x egy olyan megengedett megoldása, hogy

$$|OPT(x) - c(S(s))| \leq \beta \cdot |OPT(R(x)) - c(s)|,$$

ahol β egy másik, a visszavezetésre jellemző pozitív konstans (c mindkét esetben a költséget jelöli). Vagyis S garantáltan egy megengedett megoldását adja x-nek, amely ráadásul nincs sokkal messzebb az optimálistól, mint az $R(x)$ adott megoldása.

Állítás: Ha (R, S) egy L-visszavezetés A-ról B-re, az α, β kostansokkal, és a B-re létezik polinom idejű ε relatív hibájú közelítő algoritmus, akkor az A-ra van polinom idejű $\frac{\alpha\beta\epsilon}{1-\epsilon}$ relatív hibájú közelítő algoritmus.

Következmény: Ha van L-visszavezetés A-ról B-re és B-re létezik polinom idejű közelítési séma, akkor A-ra is létezik ilyen.

Definíció (APX-nehéz): Egy A optimalizálási probléma APX-nehéz, ha minden B APX-beli problémára létezik B-nek A-ra való L-visszavezetése.

APX-teljes, ha APX-beli és APX-nehéz.

2. fejezet

Stabil házasítás

Adott N_U férfi egy U halmazban és N_V nő egy V halmazban, továbbá minden személyhez egy preferencia lista, mely adott sorrendben tartalmaz bizonyos személyeket a másik halmazból. Ez a lista határozza meg, hogy kiket fogadnának el párjuknak, és milyen sorrenben. A feladat, hogy keressünk egy olyan M párosítást a férfiak és nők között, mely stabil, azaz csak elfogadható párokat tartalmaz, és nincs benne blokkoló pár. Egy $(m \in U, w \in V)$ pár akkor elfogadható, ha m listáján szerepel w, és w listáján is m. Blokkoló párnak pedig akkor nevezünk egy (m,w) elfogadható párt az M párosításra nézve, ha m listájában is szigorúan előbb van w, mint a párosításban lévő párja (vagy nincs párja M-ben), és ugyanígy w listájában is m (a pontos definíciót lásd később). Stabil párosítás mindig létezik, és lineáris időben meg is található.

A stabil házasítás általánosítása a Kórház/Rezidens probléma, ahol a férfiakat a rezidensek, a nőket a kórházak helyettesítik, továbbá minden w kórházhoz adott egy c(w) pozitív egész szám, a kapacitás, azaz az üres pozíciók száma. A kórházaknak és a rezidenseknek is adott a preferencia listájuk, és a rezidensek kórházakba való beosztása a célunk adott feltételek mellett (a pontos definíciót lásd később).

Ha a férfiak és nők listájában is szigorúan meg van határozva a sorrend, akkor minden stabil házasításban ugyanazok a férfiak és nők lesznek szabadok, illetve házasok. Így minden stabil házasítás elemszáma ugyanannyi lesz. A Gale-Shapley algoritmus (GS) mindig megad egy ilyen párosítást.

Ha azonban lehetnek akár csak az egyik oldalon is egyenlőségek, akkor ennek megtalálása NP-nehéz problémává válik (Iwama et al. 1999), eltekintve néhány nagyon speciális esettől. Sőt, APX-nehéz, melyet 2003-ban láttak be [6], miszerint 21/19-approximációs algoritmusnál jobbat nem lehet megadni, még akkor sem, ha csak az egyik nemnél engedjük meg a döntetleneket [4]. Ezt tovább finomítva Yanagisawa [5] bebizonyította, hogy ha találunk egy $(\frac{4}{3} - \varepsilon)$-közelítő algoritmust a Stabil Házasításra, akkor abból könnyen készíthető egy $(2 - \varepsilon)$-közelítő algoritmus a Minimális Lefogó Csúcshalmaz problémára, és ez arra is vonatkozik, ha a döntetlenek hossza csak 2 lehet. Ha pedig döntetlenek csak az egyik nem esetében lehetnek, akkor a $(\frac{5}{4} - \varepsilon)$-közelítő algoritmusból elkészíthető

egy $(2-\varepsilon)$-közelítő algoritmus a Minimális Lefogó Csúcshalmaz problémára. Érdekes módon a minimális elemszámú stabil párosítás keresése is APX-nehéz.

Könnyű belátni, hogy 2-közelítő algoritmust egyszerű adni, a döntetlenek véletlenszerű feltörése után a GS algoritmus stabil párosítást ad, és a mérete triviálisan legalább annyi, mint az optimális fele. Az első nem triviális approximációs algoritmust Halldórsson et al. [4] adták, mely $2/(1+L^{-2})$-közelítő, ha csak a férfiaknál vannak döntetlenek és ezek hossza legfeljebb L; amennyiben pedig mind a nőknél, mind a férfiaknál lehetnek döntetlenek, de csak 2 hosszúak maximum, akkor egy 13/7-approximációs algoritmust adtak.

Iwama, Miyazaki és Yamauchi [7] 2007-ben egy 15/8-approximációs algoritmust adott közzé. Ennek egy speciális esetét –amikor döntetlenek csak az egyik nemnél vannak megengedve, és ott is csak a lista végén– fejlesztette tovább Irving és Manlove [3] egy 5/3-approximációt adva rá.

Abban az esetben, ha a férfiak listája szigorú, Király Zoltán [1] egy 3/2-approximációs algoritmust adott, mely a Kórház/Rezidens problémára is kiterjeszthető (abban az esetben, amikor a rezidensek listája szigorú) ugyanúgy 3/2-approximációt adva rá. Továbbá az általános esetre is szolgáltatott egy algoritmust, mely pedig 5/3-approximációs és szintén lineáris időben fut.

Nézzük tehát először a probléma matematikai modelljét.

2.1. A probléma matematikai modellje

A problémát modellezzük egy $G=(U,V,E)$ irányítatlan páros gráffal, ahol U és V a fent említett halmazok, illetve E az elfogadható párok halmaza, ezek lesznek a gráf élei. A preferencia listákat az élekre írt értékekkel fogjuk eltárolni, úgy hogy minden elfogadható *(m,w)* párhoz a *pri(m,w)* jelöli az m szerinti prioritást egy *1* és N közötti egész számmal, és azt mondjuk, hogy $m \in U$ szigorúan preferálja $w \in V$-t $w' \in V$-vel szemben, ha $pri(m,w) > pri(m,w')$. A preferencia listában lévő egyenlőségeket azonos számokkal jelöljük (pl. ha m azonosan viszonyul $w_1, w_2 ... w_k$-hoz, akkor $pri(m,w_1) = pri(m,w_2) = ... = pri(m,w_k)$). Így minden él m felőli végére a *pri(m,w)*-t, míg w felőlire pedig *pri(w,m)*-et írva könnyen ábrázolhatjuk a gráfot.

Ebben a gráfban keresünk egy M párosítást. Ha egy $m \in V$ párosítva van M-ben, akkor a párját jelölje *M(m)*. Ugyanígy a w párját *M(w)*.

Definíció (blokkoló pár): Egy (m,w) pár blokkoló, ha $(m,w) \in E \setminus M$ és m egyedülálló vagy $pri(m,w) > pri(m,M(m))$ és w egyedülálló, vagy $pri(w,m) > pri(w,M(w))$.

2.2. Gale-Shapley algoritmus (GS)

Stabil párosítás keresésére már van egy jól ismert algoritmus, melyet D. Gale és L. S. Shapley [8] tett közzé 1962-ben. Az algoritmus azon alapul, hogy a férfiak kérik meg a nőket a listájuk szerinti sorrendben. A nők pedig az első kérőt ideiglenesen elfogadják, majd minden további kérőnél mérlegelik, hogy az

jobb választás-e, mint az eddigi. Eszerint két eset lehetséges: vagy megtartják az eddigit, és elutasítják az új kérőt, vagy a régit utasítják el, és az új kérő lesz az aktuális párjuk. Az algoritmus akkor ér véget, ha minden férfi vagy párban áll, vagy a listája végére ért.

Kezdetben minden férfi szabad és aktív továbbá minden nő szabad. Minden m aktív férfi megkéri a listáján szereplő első nőt, legyen ez *w,* ha w elfogadja m-t akkor m inaktív lesz, ha pedig nem fogadja el, akkor m továbbra is aktív marad. Amikor egy férfi a listája végére ér, akkor inaktívvá válik. Minden w nőre az első férfi aki megkéri az lesz *M(w)*, azaz az aktuális párja, w foglalt lesz, és ettől a ponttól kezdve már nem is válik szabaddá. Később, ha egy *m'* férfi megkéri, akkor abban az esetben utasítja vissza, ha $pri(w, m') \leq pri(w, M(w))$, ellenkező esetben pedig *M(w)*-t utasítja vissza, *M(w)* aktív lesz, és w új párja *m'* lesz $(M(w) := m')$. Az algoritmus akkor ér véget, ha minden férfi inaktívvá vált.

Az algoritmus $O(|E|)$ idő alatt fut le, ha a gráfot éllistával adjuk meg, és rendezett listát kapunk (feltehető, hogy senkinek nincs üres preferencia listája).

2.3. Férfiak listája szigorú

Abban a speciális esetben, ha minden férfi preferencia listája szigorú, Király Zoltán [1] adott egy egyszerű algoritmust, ami $O(|E|)$ időben fut le, ha optimálisan implementáljuk és 3/2-approximációs. A módszer azon alapul, hogy futtat egy GS algoritmust, majd a szabad férfiaknak extra pontokat ad. Ezek után újra aktiválva őket, a listájuk elejéről indítják a kéréseket.

Az extra pontokat $\pi(m)$-el fogjuk jelölni minden $m \in V$ férfira. Kezdetben ez $\forall m \in U$-ra $\pi(m) := 0$, és minden időpillanatban fennáll, hogy $0 \leq \pi(m) < 1$ minden férfira. Továbbá új prioritást vezetünk be $pri'(m, w) = pri(m, w)$, és $pri'(w, m) = pri(w, m) + \Pi(m)$ minden elfogadható *(m,w)* párra. Belátható, hogy ha egy M párosítás stabil a *pri'*-vel jelölt prioritásokra nézve, akkor az eredetire nézve is az lesz. Az így módosított algoritmust rmGS-nek nevezte el (reduced men-proposal GS). Kiindul egy stabil párosításból és az aktív férfiak listájából, majd megindul a GS, ahol a nők a pri'-vel jelölt értékek alapján hozzák meg a döntéseiket. Ha van olyan férfi, aki *0* extra ponttal rendelkezik, és a listája végére ért, akkor növeljük a pontját ε-al, és újra aktiváljuk, majd a listája elejéről kezdi a kérést. Legyen SM a szabad férfiak halmaza, és $\Pi_0 := \{m \in U : \pi(m) = 0\}$. Az algoritmus tehát a következő:

```
GSA1 algoritmus:
run GS
FOR m ∈ U π(m) := 0
    WHILE SM ∩ Π_0 ≠ ∅
        FOR m ∈ SM ∪ Π_0
            π(m) := ε
            m újra aktív
        run rmGS
```

Állítás: Ez az algoritmus $3/2$-es approximációt ad a feladatra.

Bizonyítás: Legyen M az algoritmus által adott párosítás, M_{OPT} pedig az optimális stabil párosítás. Vegyük az M és M_{OPT} unióját. A közös éleket egy kettő hosszú körként vegyük. Így minden komponense az $M \cup M_{OPT}$ egy alternáló kör, vagy alternáló út. Egy alternáló utat, melynek mindkét vége M_{OPT}-ban van, növelő útnak hívják. A növelő út rövid, ha 3 élt tartalmaz. Így elég belátni, hogy minden komponensben maximum $3/2$-edszer annyi él van M_{OPT}-ban mint M-ben. Ez nyilván igaz minden komponensben, kivéve a rövid növelő utat. Belátjuk tehát, hogy rövid növelő út nem létezhet. Legyen $M(m) = w$, $M_{OPT}(m) = w' \neq w$, $M_{OPT}(w) = m' \neq m$ és m'-nek illetve w'-nek nincs párja M-ben. Ha w'-nek nincs párja, az azt jelenti, hogy sosem kérték meg az algoritmus folyamán. Ebből következik, hogy $\pi(m) = 0$ az algoritmus végén, különben végigkérte volna a listáját (beleérte w'-t is). Továbbá $pri(m,w) > pri(m,w')$, mert a férfiak listáján nincsenek egyenlőségek. Amikor az algoritmus a végére ér $\pi(m') = \epsilon$ és m' végigkérte a listáján lévő összes nőt ezzel az extra ponttal, mégis visszautasította többek között w is. Ez azt jelenti, hogy $pri(w,m) = pri'(w,m) \geq pri'(w,m') = pri(w,m') + \epsilon$, tehát $pri(w,m) > pri(w,m')$. Azonban ebben az esetben az mw blokkoló pár M_{OPT}-ra nézve ami ellentmondás.

2.4. Általános eset

A stabil házasítás általános esetére Király Zoltán egy $5/3$-aprocimációs algoritmust adott.

A módszer mente, hogy elsőként, futtatunk egy GSA1-et, majd felcseréljük a férfiak és nők szerepeit, és a nők kapnak extra pontokat. Kezdetben tehát minden w nőre $\pi(w) = 0$ és mindenkor $0 \leq \pi(w) < 1$. Továbbá a prioritásokat is újradefiniáljuk: $pri'(m,w) = pri(m,w) + \pi(w)$ és $pri'(w,m) = pri(w,m) + \pi(m)$ minden elfogadható (m,w) párra. Belátható, hogy ha egy M párosítás stabil pri'-re, akkor stabil pri-re nézve is.

Az rwGS algoritmust (reduced woman-proposal GS) az rmGS algoritmushoz hasonlóan definiáljuk. Kiindul egy stabil párosításból. A szabad nők extra pontokat kapnak, és lefut egy GS algoritmus, ahol a nők kérik meg a férfiakat, és a férfiak a pri' prioritás szerint döntenek. Ám van egy lényeges különbség az rmGS-hez képest, ugyanis ha egy w nő szabaddá válik az algoritmus folyamán (elveszti a párját), akkor extra pontot kap ($\pi(w) = \varepsilon/2$), újraaktiválódik, és a listája legelejéről kezdi el a kéréseit.

Ha valamelyik nő ε-nál kisebb π értékkel marad egyedül, akkor ε-ra állítjuk a pontját, és újraaktiváljuk, és a listája legelejéről kezdi a kérést. SW legyen az egyedülálló nők listája és $\Pi := \{w \in V : \pi(w) \leq \varepsilon/2\}$, és $\varepsilon = 1/2$.

Az algoritmus a következő:

GSA2 algoritmus:
run GSA1
FOR $w \in V$ $\pi(w) := 0$

WHILE $SW \cap \Pi \neq \emptyset$
  FOR $w \in SW \cup \Pi$
    $\pi(w) := \epsilon$
    w újra aktív
  run rwGS

Állítás: Ez az algoritmus $\frac{5}{3}$-approximációt ad a Stabil Házasítás általános esetére.

3. fejezet

Kórház/Rezidens probléma

3.1. A matematikai modell

Adott egy $G = (U, V; E)$ páros gráf, ahol $|U| = N_U$, $|V| = N_V$. U elemei a rezidenseket, míg V elemei pedig kórházakat szimbolizálják, továbbá minden w kórházhoz adott egy pozitív egész szám c(w), a kapacitás (az üres pozíciók száma). A feladat egy olyan $F \subseteq G$ részgráf keresése (beosztás), melyben minden m rezidens foka maximum 1, és minden w kórház foka maximum c(w). Egy m rezidensre, mely F-ben be van osztva egy kórházhoz, F(m) jelöli a megfelelő kórházat. Egy w kórházra pedig F(w) jelenti a hozzá beosztot rezidensek halmazát. Egy w kórház telített, ha $|F(w)| = c(w)$. Egy (m,w) pár blokkoló az F-re nézve, ha $(m, w) \in E \setminus F$, és m vagy nincs beosztva sehova, vagy pri(m,w)$>$ pri(m,F(m)), ugyanakkor w nem telített, vagy telített, de pri(w,m) $>$ pri(w,m'), legalább egy $m' \in F(w)$ rezidensre. Egy F beosztás stabil, ha nincs benne blokkoló pár.

A GS algoritmus pedig könnyen módosítható úgy HRGS algoritmussá, hogy stabil beosztást adjon a kórház/rezidens problémára.

3.2. Szigorú listával a rezidensek oldalán

Abban az esetben, ha a rezidensek listája szigorú a GSA1 algoritmus egy módosításával 3/2-approximációs eredményhez juthatunk. A GS helyett HRGS-t alkalmazunk, az rmGS helyett pedig megadunk egy rmHRGS algoritmust, mely a HRGS egy módosítása lesz. Az SM halmaz tartalmazza azokat a rezidenseket, akik nincsenek még beosztva, és a $\Pi_0 := \{m \in U : \pi(m) = 0\}$.

HRGSA1 algoritmus:
run HRGS
FOR $m \in U$ $\pi(m) := 0$
    WHILE $SM \cap \Pi_0 \neq \emptyset$
        FOR $m \in SM \cup \Pi_0$

$\pi(m) := \epsilon$
m újra aktív
run rmHRGS

A HRGSA1 algoritmus úgyszintén $O(|E|)$ idő alatt fut (amennyiben rendezetten kapjuk a listákat), és egy F stabil beosztást ad.

3.3. Irving-Manlove algoritmusai

Robert W. Irving és David F. Manlove [2] 2009-es cikkükben öt algoritmust hasonlítottak össze a Kórház/Rezidens problémára. Két általuk elkészített algoritmust (Heur-H és Heur-R), a fent említett Király-féle hrGSA1 algoritmust, illetve hétféle véletlen algoritmust, melyek a döntetleneket törik fel valamilyen séma szerint. Az egyik a Heur-I, melyben minden döntetlent egymástól függetlenül, és véletlen permutációval törünk fel, míg a másik a Heur-C, ahol pedig egy fő sorrend szerint törünk meg minden döntetlent. Itt a fő sorrend a férfiak egy véletlen permutációja.

4. fejezet

Implementálás

Manapság jópár programozási nyelv rendelkezésünkre áll, ha programozásról van szó. Többek között a C, C++, C#, Java, és még sorolhatnánk. A választásunkat azonban megkönnyítette, hogy az ELTE Operációkutatás Tanszékén fejlesztik a LEMON-t [9] (Library for Efficient Modeling and Optimization in Networks), mely egy 2003-ban indított projekt. Ez egy C++ template könyvtár, ami a gyakoribb adatsruktúrák, illetve algoritmusok hatékony implementálását tartalmazza, főleg a kombinatorikus optimalizálásra fókuszálva, gráfokkal, hálózatokkal modellezve.

A programokat az ELTE hajos.cs.elte.hu gépén futtattuk le, így a két duál magos processzoron (Intel Pentium 4, 2.80 GHz-es) és 16GB memóriás gépen teszteltük az algoritmusokat.

4.1. Programozási megoldások

Gráfok tárolására két fő módszer ismeretes, az egyik az adjacencia-mátrix, a másik az éllistás tárolás. A LEMON-ban ezek alapján alkottak többféle struktúrát, melyek közül választhatunk amikor gráfokkal szeretnénk dolgozni. A gráfok tárolására így kétféle adatstruktúra is szóba jöhetett: a *ListGraph* és a *SmartGraph*. A *ListGraph* egy viszonylag gyors és rugalmas, linkelt listákkal való megvalósítása a gráfoknak, míg a *SmartGraph* sokkal egyszerűbb, és memória kímélő megoldás, azonban ennek következtében nem támogatja az élek törlését. Mivel az élek tényleges törlésére nincs szükség az alábbi algoritmusoknál, így a hatékonyság érdekében a *SmartGraph*-ot (a továbbiakban: SG) választottuk. Ennek függvénye a `void addNode()`, `void addEdge(SG :: Node, SG :: Node)`, mely csúcsok és élek hozzáadását teszi lehetővé, továbbá az `SG :: Node u(SG :: Edge)` és `SG :: Node v(SG :: Edge)`, mely egy él egyik, illetve másik végét adja vissza. A G gráfot tehát egy *SG g* változóval jelöljük a továbbiakban.

Az U és V csúcshalmaz különválasztása a legegyszerűbben úgy lehetséges, hogy az első N_U csúcs tartozik az U-ba, a következő N_V pedig a V-be. A `g.addNode()` függvénnyel így létrehozunk $N_U + N_V$ csúcsot a gráfban, melyekhez

rendre a 0,1,2...(N_U+N_V-1) természetes számokat fogja hozzárendelni a továbbiakban, tehát az `SG :: Node nodeFromId(intn)` függvénnyel az n sorszámú csúcsot kaphatjuk vissza.

A *pri(m,w)* értékek tárolására egy `SG :: EdgeMap < int > pri_m(g)`-t használunk, ami a g élein értelmezett *Map* struktúra, mely *int*-eket tartalmaz (azaz minden élhez hozzá lehet rendelni egy egész számot, a *pri(m,w)*-t, ahol az él két végpontja m és w). Ugyanígy a *pri(w,m)* tárolására létrehozzuk az `SG :: EdgeMap < int > pri_w(g)`-t.

Így az értékek tárolása megtörtént, azonban minden férfihoz és nőhöz a preferencia listát is el kell tárolnunk. Ehhez egy `vector < vector < Pair >>`-t hozunk létre, ahol a `Pair` a `pair < SG :: Edge, int >`. Így minden csúcshoz, egy `vector < Pair >` fog tartozni (a csúcs id-jével indexelve) mely a preferencia listáját tartalmazza oly módon, hogy a vector minden eleme az élből és a hozzá tartozó pri értékből álló pár.

Még egy *NodeMap*-et használunk a továbbiakban, *engw* névvel jelölve ($SG :: NodeMap < int > engw(g)$), mely megadja, hogy melyik embernek ki az aktuális párja. Ennek a default értéke -1. Később pedig a párost összekötő él sorszámával írjuk felül. Ezekkel tehát a program három nagyobb részre bontható, a gráf beolvasása, a kezdeti értékek megadása (init) és az algoritmus maga. Mivel a gráf beolvasása fájlból történik, így külön program szolgál a tesztgráfok elkészítésére és annak egy fájlba mentésére bizonyos formátumbeli megkötésekkel, hogy utána a beolvasás minél könnyebben történhessen. Így a következő két alfejezetben külön taglalom a gráf elkészítését és fájlbamentését. Továbbá az algoritmusok közös részeit, a gráf beolvasását és a kezdeti értékek megadását.

4.1.1. A gráfok elkészítése és fájlbamentése

Tesztgráfok elkészítésére többféle algoritmust is használtunk. Néhány lépés azonban közös mindegyikben.

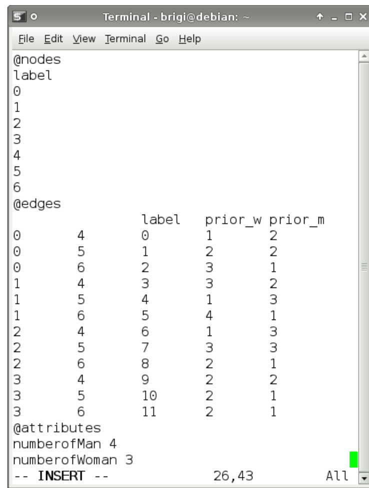
Az első input paraméter mindig az output fájl neve. Ezt azért fontos megadni, hogy többszöri meghívás esetén minden output fájl más nevet viseljen, így nem írják felül egymást. Ezt tehát kötelezően meg kell adni.

A következő két paraméter minden esetben az N_U és N_V értékei (azaz a férfiak és nők száma). Nem végez ellenőrzést a program, hogy pozitív egész számot adunk-e meg, minimum egyet-e és esetleg egy reális maximum értéknél kisebbet-e, mert feltételezhető, hogy a felhasználó ezekkel tisztában van. A megadott számok szerint az *addNode()* függvény $N_U + N_V$-szeri meghívásával létrehozza a megfelelő számú csúcsot a gráfban.

Ezután a program különböző változataiban más és más séma szerint húz be éleket, és ad meg hozzájuk prioritásokat, melyek alapján a preferencia listák elkészülnek.

Ezek után minden program meghívja a `graphWriter` függvényt, mely egy megadott formátumú output fájlt készít. Paraméterként át kell adni a gráfot, a fájlnevet, továbbá lehetőség van *NodeMap*-ek (pl. pri_w és pri_m), és int-ek átadására (N_U és N_V). Így a 2.1-es ábrán látható fájlhoz hasonlóan

menti el a gráfot. A fájlba mentésnek az a lényeges előnye, hogy így külön el lehet készíteni, akár 10, 100 vagy 1000 tesztfájlt is, különböző paraméterekkel, majd mindegyikre több algoritmust is lefuttatni, és az eredményeket táblázatba menteni.



4.1. ábra. példa a GraphWriter függvény által készített fájl szerkezetére

4.1.1.1. create.cc

A create.cc a legegyszerűbb változata a generálásnak. A fő paraméterek után még 3-at kér, egyet az élet behúzásához, kettőt pedig a pri értékek előállításához.

Az első paraméter (vagy a program által szövegesen bekért érték) egy százalékszám, ami megadja, hogy a lehetséges élek megközelítőleg hány százalékát húzza be (azaz milyen sűrű legyen a gráf). Pontos jelentése, hogy minden élre dob egy véletlenszerű számot *1* és *100* között, és ha ez a megadott értéknél kisebb, akkor behúzza az élt. Itt és a továbbiakban is, véletlenszám generálására a `drand48`() függvényt és különböző változatait fogjuk használni. A megfelelő modulust használva egyenletes eloszlásban kapunk értékeket a megfelelő tartományból. Jelen esetben a *lrand48()%100* (% jelentése: modulo) értéke 0 és 99 közé esik, és egyenletes eloszlású. Ez a módszer jól működik kevés csúcsú gráfokra, azonban 1000 férfi és 1000 nő esetén a gráf maximális élszáma 1000000, így ha az éleknek csak 3-5 százalékát húzzuk be, az esetek nagy részében akkor is könnyen található 1000 élszámú párosítás.

Az ezutáni két paraméter (x,y) nem kötelező. Ha nem adunk meg ilyen értékeket, akkor az alapértelmezett az N_U illetve N_V. Az x és y azt adja meg, hogy minden (m,w) élre a program generál egy véletlen számot 1 és x között (ez lesz pri(w,m)), majd egy másik véletlen számot 1 és y között (ez pedig pri(m,w)). Tehát x-nek és y-nak 1-nél nagyobb természetes számoknak kell

lenniük. Ezáltal, ha sok döntetlent szeretnénk a preferencia listákban, akkor kellőképpen kis x és y értékekkel ez könnyen elérhető. Viszont nem adható meg a döntetlenek minimális illetve maximális hossza.

4.1.1.2. create_k.cc

A create_k.cc esetén arra törekszünk, hogy a maximális párosítás értéke k körül mozogjon. Erre van egy módszer, miszerint az U és V halmazokat is felbonjtuk 2 részre, jelöltekre és jelöletlenekre. Egy $m \in U$ jelöletlen csúcsból p valószínűséggel húzunk be élt egy $w \in V$ jelöletlenbe, és q valószínűséggel egy $w' \in V$ jelöltbe, továbbá egy $m' \in U$ jelölt csúcsból tilos élt húzni $w \in V$ jelöletlenbe, és p valószínűséggel húzunk be élt egy $w' \in V$ jelöltbe. Ha az U halmazból n-k/2 csúcs jelölt (ahol $n = |N_U| = |N_V|$), a V halmazból pedig k/2, akkor a maximális párosítás mérete nem lehet nagyobb k-nál (k<n esetén).

Ez a generálás az előző általánosítása, ugyanis $k = 2n$ esetén az előzővel azonos módon működik.

4.1.1.3. create_pri.cc

Ez a változat arra a feladattípusra készült, ahol max L hosszúságú döntetlenek lehetnek a preferencia listákon. Erre azt a módszert alkalmazzuk, hogy minden csúcsra megvizsgáljuk a fokszámot (legyen ez d a továbbiakban), és generálunk véletlen számokat 1 és L között úgy, hogy az összegük d legyen ($d_1 + d_2 + ...d_l = d \; \forall d_i \; 1 \leq d_i \leq L$). Majd létrehozunk egy tömböt, ahova beírunk $d_1$db 1-es, d_2 db 2-est... d_l db l-est, majd egy véletlen permutációt készítünk. Ezután pedig a tömbön végighaladva minden kimenő élre kiosztjuk a tömb sorra következő elemét.

Tehát a fő paraméterek után sorban az L, k, p és q értékeket kell megadni, ahol k, p és q a create_k.cc részben leírt paramétereket jelöli.

4.1.1.4. create_hr.cc

A create_hr.cc a Kórház/Rezidens (HR) problémához készít gráfot. Itt az első N_U csúcs a rezidenseket szimbolizálja, míg a következő N_V a kórházakat. A fő paraméterek itt is ugyanazok, majd sorban a p, x, y, mint a create.cc esetében. Majd végül 1 vagy 0 aszerint, hogy a kórházak kapacitásai véletlen számok legyenek-e, vagy azonosak. Ha 0, akkor minden kórház kapacitása N_U/N_V egészrésze, ha pedig 1 akkor N_U db pozíciót osztunk ki véletlenszerűen a kórházak között, majd egy kórház kapacitása a rá osztott pozíciók összege lesz.

4.1.1.5. create_hr_k.cc

Úgyszintén a kórház/rezidens problémához készít gráfot. A fő paraméterek ugyanazok, majd sorban k,p,q,x,y és a végén a 0,1 érték valamelyike. Működése hasonló a create_k.cc-hez, kijelöl n-k/2 rezidenst, és néhány kórházat úgy, hogy megközelítőleg k/2 legyen a kapacitásuk összege. Ezután a jelöletlen

rezidensből jelöletlen kórházba p valószínűséggel megy él, míg a kijelöltbe q valószínűséggel, és jelölt rezidensből jelöletlen kórházba nem mehet él, jelöltbe pedig p valószínűséggel. Az x,y paraméterek a prioritások maximumát adják, mint az előző esetben is.

4.1.1.6. create_hr_pri.cc

A create_hr_k.cc változata, melyben a preferencia listákon maximálisan L hosszú döntetlenek lehetnek csak. A főparaméterek után így sorban az L, k, p, q értékeket kell megadni, ahol a k, p és q a fent említett paraméterek.

4.1.2. Az algoritmusok közös részei

Ezek után a program betölti a gráfot, mellyel dogozni kívánunk. Meghívja a *graphReader* függvényt a gráf nevével, ahova menteni kívánjuk, illetve az output fájl nevével, ahonnan beolvassuk. Betölthetőek továbbá a *pri(w,m)*, *pri(m,w)* értékek, és az N_U, N_V számok is.

Utána a gráf és a pri értékek szerint beállíthatóak a változók kezdeti értékei. A férfiakat betesszük egy *enableMen* adatstruktúrába, mely azokat a szabad férfiakat fogja mindenkor tartalmazni, melyeknek a listája még nem üres. Az algoritmus folyamán ebből fogunk választani férfiakat, akik megkérik a listájukon soron következő nőt, így ennek a struktúrának a megválasztása a legkérdésesebb, hiszen az elemek tárolásának és kivételének sorrendje nagyban hatással van az algoritmus menetére. Erre a későbbiekben még vissza fogunk térni, sőt, többféle struktúrát is megvizsgálunk.

Létrehozunk egy NodeMap-et (`SG :: NodeMap engw(g)`), melyben tároljuk minden csúcshoz, az aktuális párját, méghozzá az őket összekötő él sorszámával. Így a -1 érték jelenti, ha nincs párja, ha pedig van, akkor a `g.u` (az élhez tartozó férfit adja meg) vagy `g.v` (az élhez tartozó nőt adja meg) függvényekkel megkaphatjuk a párjukat.

A preferencia listákat minden személyre egy `vector < Pair >` struktúrában tároljunk, ahol a Pair a `pair < SG :: Edge, int >` rövidítése. Így tehát a C++ `sort` algoritmusával, és egy jól meghatározott rendező függvénnyel minden preferencia lista előállítható az adott pri értékekből. Mivel minden személyre van egy ilyen lista, ezért egy vector-ban tárolhatjuk ezeket a vectorokat (`vector < vector < Pair >>`) a csúcs sorszámával indexelve.

Az algoritmus folyamán ezekben a listákban fogunk végigmenni (későbbieken van hogy többször is), így tárolnunk kell ,hogy éppen melyik lista hányadik eleménél tartunk. Erre szolgál a `SG :: NodeMap priq_num(g)`, mely kezdetben 0, mutatva, hogy minden listának a legelején állunk. A továbbiakban pedig az i sorszámú férfira a `priq[i][priq_num]` adja meg a lista aktuális elemét. Amennyiben pedig a lista végére értünk, a priq_num értéke a lista hossza lesz, így a lista hosszából kivonva ezt az értéket 0-t kapunk ha a listát végigkérte már a férfi.

4.2. Alternatívák az GS algoritmusra

A GS algoritmusban nincs meghatározva, hogy a szabad férfiak közül milyen módszerrel válasszuk ki a következőt. Ezzel a kérdéssel foglalkozva négyféle lehetséges esetet vizsgálunk meg, melyekben ezen férfiakat különböző adatszerkezetekben tároljuk (halmaz, sor, verem, elsőbbségi sor). Illetve az ötödik, amikor véletlenszerűen választjuk ki közülük a következőt. Az aktív férfiakat tehát az alábbi struktúrákban tároljuk:

A(halmaz) Ebben az esetben egy halmaz, a C++ Set nevű struktúrája. A halmaz elemein egy iterátorral haladunk végig, mely a begin() mutatótól halad, amíg az end()-et el nem éri.

B(sor) Itt egy sor, a C++ Queue struktúrája. Ebben a front() függvény megadja a sor legelső elemét, a pop() törli ezt, a push()-al pedig betehetünk egy új elemet, mely így a sor végére kerül.

C(verem) Itt egy vermet használunk, a C++ Stack struktúráját. Itt a top() függvénnyel érhető el a legfelső elem, mely a pop()-al törölhető a veremből, és a push()-al helyezhetünk új elemet be. Érdekessége, hogy ha egy férfit visszautasítanak, és folytatja a kérést a listájáról, akkor rögtön ő kérhet megint, egészen addig, amíg nem talál egy nőt aki elfogadja, vagy a listája végére ér. Ekkor töröljük a veremből, és kerül sorra a következő férfi.

D(elsőbbségi-sor) Egy elsőbbségi sor az *enableMen*, a C++ Priority Queue struktúrája, és minden férfit azon értékek szerint tárolunk, hogy a listájukon még hány nő szerepel, akit megkérhetnek. Ez azért lehet hasznos, mert a nők akkor váltanak az új kérőre, ha az szigorúan jobb mint az előző, ezáltal az azonosra értékeltek közül az van jobb helyzetben, aki elsőként kéri meg a nőt.

E:(véletlen) Végül pedig egy vektort használunk és a soron következő férfit egy véletlen algoritmus segítségével választjuk ki.

4.2.1. A GS algoritmus, és változatainak megvalósítása

Minden algoritmus egy *while* ciklussal indít, hiszen addig tart az algoritmus, amíg minden férfi inaktívvá válik. Azaz az enableMen üres lesz. A méretét a *size()* függvénnyel lehet lekérni, így amíg ez nem egyenlő 0, addig fut az algoritmus.

Ezután kiválasztunk egy aktív férfit. Majd két részre válik az algoritmus egy *if-else* ciklussal. Amennyiben a választot férfinak már nincs a listáján nő (a priq_num aktuális értéke pontosan a freferencia listájának a hossza), abban az esetben m inaktívvá válik és töröljük m-et az enableMen-ből.

Ha még van a listáján nő, akkor az else ágon megyünk tovább. Ebben az esetben megnézzük a listáján lévő soron lévő nőt. A `priq[g.id(m)][priq_num].first`

jelenti az élt, mely m-et w-vel köti össze, ezt e-vel jelölve `w` = `g.v(e)` lesz a meglelelő nő.

Ezután válik ismét két részre aszerint, hogy w foglalt-e vagy szabad. Amennyiben szabad (`engw[w] == −1`, azaz a kezdeti érték), akkor m lesz az új pár, és beállítjuk a megfelelő változók értékeit: m és w párok lesznek (`engw[m]` = `g.id(e)`,`engw[w]` = `g.id(e)`). Továbbá töröljük az enableMen-ből az m férfit, illetve m listájában előre lépünk egyet (`priq_num[m]++`).

Ha nem volt szabad w, akkor $m_$ jelöli a aktuális párját, és $e_$ (ami a `g.edgeFromId(engw[w])` függvénnyel kapható) a pároshoz tartozó él.

Ezután ismét egy *if-else* ciklus bontja két részre az algoritmust. Ha az m-et w jobban kedveli, mint $m_$-t (azaz `pri_w(e)` > `pri_w(e_)`), akkor az m lesz az új párja w-nek, és beállítjuk a változók értékeit*:* m és w párok lesznek (`engw[m]` = `g.id(e)`, `engw[w]` = `g.id(e)`), míg $m_$-nek nincs párja (`engw[m_]` = `−1`, ez a kezdeti értéke az `engw`-nek), továbbá m listájában lépünk egyet (`priq_num[m]++`), majd töröljük m-et az enableMen-ből.

Abban az esetben, ha w nem kedveli jobban m-et mint $m_$-t, akkor m listájában lépünk egyet. Ezzel az algoritmus véget is ér.

Nézzük tehát kérdéses műveleteket az 5 féle megvalósításnál.

Elem kivétele az enableMenből:

A: Az enableMen egy `set < SG :: Node >`, így az ezen értelmezett iterátorral kell végigmennünk az elemeken egy for ciklussal. A program elején a `typedef set < SG :: Node >:: iterator SetIt` programsorral az iterátorra ezentúl `SetIt`-ként lehet hivatkozni a rövidítés kedvéért. Ezzel tehát a for ciklus a következő lesz:
`for(SetIt enableMen.begin(); m! = enableMen.begin(); m++)`. A továbbiakban tehát m helyett *m-el hivatkozhatunk az adott férfira.

B: A Queue-nál a `front()` függvény visszaadja az első elemet, így az `m = enableMen.front()` után m-el hivatkozhatunk a férfira.

C: Stack-nél az előzőhöz hasonlóan az `m = enableMen.top()` sor szükséges.

D: Priority_Queue esetében az `m = enableMen.top().first` szükséges, hiszen az elsőbbségi sor minden eleme egy pár, melynek első tagja a férfi, a második pedig egy számérték, és a `first`-el hivatkozhatunk az első tagra.

E: A véletlenszerű esetben az enableMen aktuális elemszáma (`enableMen.size()`) alapján generálunk egy számot, majd az enableMen ezen indexű eleme lesz az aktuális férfi. Tehát egy `int` tipusú `num` változót hozunk létre, ez lesz a véletlen szám modulo a méret (`rand()%enableMen.size()`). Majd `m = enableMen[num]`.

Elem törlése az enableMen-ből:

A: Itt mivel az m egy iterátor volt, ezért az `enableMen.erase(m)` segítségével egyszerűen törölhetjük az m-et, akárhol foglal is helyet a struktúrában.

B: Ebben az esetben csak a legelső elem törlése lehetséges az `enableMen.pop()` függvénnyel.

C: Stack esetében is az `enableMen.pop()` függvény törli a legfelső elemet, azonban figyelni kell rá, hogy amit törölni szeretnénk biztos, hogy a legfelső helyen legyen. Ez akkor fontos, amikor az aktuális férfi olyan nőt kér meg, akinek van párja, de jobbnak bizonyul, mint a régi férfi. Ekkor ugyanis a régi férfi bekerül az enableMen-be, és előbb kell törölni az aktuálisat, majd betenni a régit, különben ahogy betesszük a régi férfit az lesz a legfelső elem, és őt törölnénk a `pop()` függvénnyel.

D: Az elsőbbségi sor esetében is a `pop()` függvénnyel törlünk, azonban a sorrendre itt is figyelni kell. A C-nél említett eset itt is előfordulhat, minden oylan esetben, amikor a régi férfinak kevesebb nő van a listáján, mint az aktuálisnak. Továbbá abban az esetben, amikor az aktuális férfit visszautasítja a nő, és újból visszakerül az enableMen-be, akkor előbb töröljük, majd csökkentjük a listáját egyel, és újra betesszük, így az egyel kisebb értékkel kerül az elsőbbségi sorba.

E: Itt egy vectorban vannak tárolva a férfiak, és a vectornak csak az utolsó elemét lehet törölni a `pop_back()` művelettel. Így amennyiben a num változóval jelzett elemet kell törölni, a vector utolsó elemét (mivel 0-tól számoz, ezért a (méret-1).elem az utolsó) bemásoljuk a num indexű helyre, majd töröljük az utolsó elemet. Azaz `enableMen[num] = enableMen[enableMen.size() − 1];` `enableMen.pop_back()`.

Új elem behelyezése az enableMen-be:

A: m behelyezése az `enableMen.insert(m)`-el történik.

B: Itt az `enableMen.push(m)`-el történik, és értelemszerűen a lista leg vé gé re kerül.

C: Itt is az `push(m)`-et használjuk, azonban a verem legtetejére kerül.

D: Itt az `enableMen.push()` függvényt használjuk, a `pair < SG :: Node, int > (m, priq[g.id(m)].size())` paraméterrel, ami azt jelenti, hogy készít egy párt, az m csúcsból, és a hozzá tartozó `priq` méretéből. Majd a méret szerint teszi az elsőbbségi sor megfelelő helyére.

E: Itt a `push_back(m)` függvénnyel lehet elemet beilleszteni a vector végére. Azonban mivel az algoritmus közben csak akkor teszünk be új elemet, amikor az aktuális férfi (m) egy másik ($m_$) helyére

kerül, így az enableMen-ben az aktuális (*num* sorszámú) férfi helyére kerülhet a párját vesztett férfi, ezáltal nincs szükség elem törlésére és a vector mérete változatlan lesz. Ez pedig az `enableMen[num] = m_` művelettel könnyedén megoldható.

4.2.2. GSA1 algoritmus

Az előzőektől eltérve itt két algoritmust futtatunk, egy GS-t és egy rmGS-t a GSA1 algortimus részeként. Így lesz egy header file (GS_.h), ahol a két algoritmus meg van írva, és ezt hívja meg a GSA1.cc, mely maga a GSA1 algoritmust tartalmazza.

A GSA1.cc egyetlen input paramétere egy fájl név lesz, mely tartalmazza a gráfot, amivel dolgozni akarunk. Egy ellenőrzéssel indul a program (adtunk-e meg paramétert, ha igen, létezik-e ilyen névvel fájl). Majd egy `vector < int > matching` fogja tartalmazni a párosításban szereplő élek sorszámát, ezt a `vector`-t adjuk át minden algoritmusnak, és minden algoritmus outputja egy ilyen vektor lesz. Így a GS és rmGS algoritmusok meghívása után a `matching` mérete adja a párosítás élszámát.

A GS_.h header fájl tehát tartalmazza a GS algoritmust, mint függvényt, melynek outputja egy $vector < int >$ a párosításbeli élek sorszámával, inputja pedig `char*` a fájlnév. Ezután következik a rmGS algoritmus, melynek outputja úgyszintén egy $vector < int >$, inputja viszont két paraméterből áll, a fájlnév és az eddigi megoldás (`vector < int >`).

4.2.2.1. rmGS

Az rmGS a GS algoritmus egy módosítása, így nagyrészt a program is egyezni fog. A különbségek között az első, hogy itt bevezetünk egy újabb NodeMap-et, mely float tipusú értékeket tartalmaz, ez lesz a `SG :: NodeMap < float > pii(g)`, mely a π értékeket fogja tárolni minden csúcsra. Az inicializálásnál is lesz egy plusz lépés, miszerint a kiinduló párosítás szerint rögzítjük az engw és priq_num értékeket. Ennek módja, hogy egy ciklussal végigmegyünk a vektoron, melyben az párosításbeli élek sorszámai vannak, majd a két végpontját lekérjük, így az engw értékek könnyen beállíthatóak, illetve a preferencia listákon végigmegyünk és megkeressük az aktuális párt, és rögzítjuk ennek helyét a listában, hogy a későbbiekben innen folytathassuk tovább a kéréseket amennyiben az adott férfi szabaddá válna. Az inicializálás végén pedig minden pár nélküli férfi aktív lesz, és indulhat az algoritmus.

Az aktuális aktív férfi kiválasztása után két részre bomlik az algoritmus, mint ahogy a GS esetében is. Ha a férfi preferencia listája végére ért, és már nincs kit megkérnie, akkor ellenőrizzük, hogy a pii hozzá tartozó értéke 0-e. Ha igen, akkor epszilonra változtatjuk, és a priq_num 0-ra állításával a listája elejérőlkezdheti a kéréseket a módosított prioritásokkal. Ha pedig nem volt a pii 0, akkor csak inaktívvá válik.

Abban az esetben viszont, ha még nem volt a preferencia listája végén, akkor a GS-hez hasonlóan folytatjuk, egyetlen változtatás, hogy ha volt a kiszemelt

nőnek párja, akkor az összehasonlítás nem a pri_w[e] > pri_w[e_] függvény kiértékelésével, hanem a pri_w[e]+pii[m] > pri_w[e_]+pii[m_] kiértékelésével, ahol (az e él köti össze m és w-t, e_ pedig m_ és w-t).

4.2.3. GSA2 algoritmus

A GSA2 algoritmus pszeudo kódjából látható, hogy a GS, rmGS, illetve rwGS algoritmusokat hívja meg. Tehát ebben az esetben a header fájl 3 algoritmust fog tartalmazni.

4.2.3.1. rwGS

Az rwGS egy lényeges dologban eltér a GS és rmGS algoritmusoktól, miszerint itt felcserélődnek a szerepek, és a nők kérik meg a férfiakat, továbbá a nők kapnak plusz pontokat. Itt tehát az inicializálás során fontos, hogy a priq vectort a nők sorszámával indexeljük, és az e élhez a pri_w[e] értékek kerülnek be. Ebben az esetben is egy párosításból indulunk ki, tehát az engw értékeket be kell állítanunk aszerint, azonban a priq_num értékeket nem, mert ha egy nő az algoritmus folyamán szabaddá válik, akkor a listája legelejéről kezdi a kéréseket, nem pedig a volt párjától kezdve. Az inicializálás végeztével pedig minden pár nélküli nőt aktívvá teszünk.

Kezdődhet az algoritmus, melyben most az aktív nők közül választunk, így a struktúra neve is enableMen helyett enableWomen lesz. Működése teljesen hasonló az rmGS-hez, csak a szerepek megcserélésével.

4.2.4. A GSA1 és GSA2 különböző változatai

A GS algoritmusnak a már említett 5 verziója alapján elkészítettük az rmGS és rwGS 5-5 verzióját is, melyeket kombinálva 25 féle GSA1 algoritmus, illetve 125 féle GSA2 algoritmushoz jutottunk.

A GSA1_AA-EE.cc a GSA1 algoritmus 25 verziójának futtatására készült, mivel mindegyik GS és rmGS a header fájlban egy-egy függvényként szerepel, így egy függvény pointerrel a legcélszerűbb végigmenni a változatokon. Ez a pointer a GS esetében tehát egy vector < int > *(p_GS[5])(char*) tipusú pointer, míg az rmGS esetében egy vector < int > *(p_rmGS[5])(char*, vector < int >, float). Ezek után a p_GS[0] = GS_A, p_GS[1] = GS_B ... p_GS[4] = GS_E és ugyanígy a p_rmGS[0] = rmGS_A ... p_rmGS[4] = rmGS_E sorokkal megadhatjuk, hogy a pointerek mely függvényekre mutassanak, majd két egy más ba ágyazott ciklussal könnyen végigmehetünk mind a 25 algoritmuson. p_GS[i] majd p_rmGS[j] $\forall$i, j.

A GSA2-AAA-EEE.cc nagyon hasonlóan működik az előzőhöz, csak itt még a vector < int > *(p_rwGS)(char*, vector < int >, float) pointert is be kell vezetni, és ezek lesznek az rwGS_A,B,C,D és E változataira készített függvények. Majd három egymásba ágyazott for ciklussal mind a 125 verzió futása megoldott, így könnyen lehet vizsgálni, hogy melyik adja a legnagyobb outputot, a legkisebbet, mennyi az átlaguk, és így tovább.

5. fejezet

Tesztelés és eredmények

5.1. Stabil házasítás tesztek

Elsőként a GS_break és a GS_A-E működését vizsgáltuk, ahol a GS_break azt az algoritmust jelenti, ahol feltörünk minden döntetlent, majd sima GS-t futtatunk. A férfiak és nő száma a továbbiakban végig 1000-1000 lesz.

Elsőként a gráfokat a create.cc programmal készítettük el. Az első paraméter (azaz az élek behúzásának valószínűsége) elég magas értékeire annyi él van a gráfban, hogy könnyedén találunk maximális párosítást (1000 éllel), amennyiben pedig a következő két paramétert állítjuk túl nagyra, akkor a preferencia listákban minimális lesz a döntetlenek száma és hossza, így az algoritmusok nem adnak lényegesen eltérő elemszámú párosításokat. Még abban az esetben is, ha a p=3, azaz az élek 3%-át húzzuk csak be, és a prioritások 1-10-ig mennek mind a nők, mind a férfiak szemszögéből, az algoritmusok 990 alatti eredményeket csak extrém esetekben adnak, így az eredmények túl közeliek egymáshoz ahhoz hogy következtetéseket tudjunk levonni belőlük (lásd 5.1 táblázat), így ezt a gráf készítő algoritmust a továbbiakban nem használtuk. Mivel minden gráfra az algoritmusok által talált párosítások méretei függnek az adott gráftól, így csak egymáshoz képest vizsgáltuk az algoritmusokat, pontosabban mindegyik algoritmust a GS_break-hez mértük, hogy mennyivel nagyobb élszámú párosítást talál (pozitít szám), vagy mennyivel kisebbet (negatív érték), ezeket az értékeket mutatják a táblázatok.

Következő lépésként a create_k.cc programmal készítettünk gráfokat, ennek

10 gráf	GS_A	GS_B	GS_C	GS_D	GS_E
min	-2	-4	-2	-3	-2
max	2	4	2	4	2
átlag	-0.4	-0.8	-0.1	-0.8	-0.5
szórás	1.65	2.3	1.1	2.15	1.84

5.1. táblázat. create.cc, 10 gráf, p=3 (prioritások 1-10-ig)

1-3 prior	GS_A	GS_B	GS_C	GS_D	GS_E
min	-33	-39	-8	30	-32
max	2	-12	11	51	-7
átlag	-17.7	-22	1.5	39.5	-15.5
szórás	9.12	8.81	6.07	6.27	7.53

5.2. táblázat. create_k.cc, 10 gráf, k=800, p=q=5 (a prioritások 1-3-ig mennek)

1-5 prior	GS_A	GS_B	GS_C	GS_D	GS_E
min	-22	-26	-6	22	-22
max	-2	-1	11	35	-2
átlag	-15.1	-16	-1.4	27.7	-15
szórás	6.99	7.37	4.85	3.56	6.84

5.3. táblázat. create_k.cc, 10 gráf, k=800, p=q=5 (a prioritások 1-5-ig mennek)

előnye, hogy a k értékkel lehet beállítani, hogy a maximális párosítás mérete k körül mozogjon, illetve kétféle paraméterrel is szabályozhatjuk az élsűrűséget (p és q). A továbbiakban elég a k=800 esettel foglalkozni, mert ez elég nagy érték ahhoz, hogy az algoritmusok akár 50 körül értékekkel is eltérhetnek egymástól. Amennyiben a prior_w és prior_m értékeket lerögzítjük, és a p,q értékeket pedig csökkentjük (pl. 50-40-30-20-10-5-3-2) megfigyelhetjük, hogy a p=q=5 illetve ennél kisebb értékek esetén talál a GS_D algoritmus 800-nál kisebb élszámú párosítást. Így a továbbiakban csak az 5-nél kisebb p és q értékekkel fogunk foglalkozni. Ha pedig állandó p és q értékek mellett a prioritásokat változtatjuk, akkor megfigyelhetjük, hogy kis értékre (pl. 3) a GS_D mindig 800-at talál, míg növelve a prior_w, prior_m értékeket, egyre kevesebb élszámú párosítást talál. Az alábbi táblázatokban láthatjuk az eredményeket (5.2, 5.3 illetve 5.4-es táblázat).

Felmerülhet a kérdés, hogy elengendő-e 10 gráfra futtatni az algoritmusokat, vagy több inputra van-e szükség. Az következő két ábra közül az elsőn 10 gráfra lefuttatva láthatjuk az algoritmusok közti különbségeket, a második ábrán pedig még további futtatások után (100 gráf) adott értékek alapján ugyanez a táblázat (5.5 és 5.6-os táblázat). Látható tehát, hogy az eredmények nem változnak lényegesen a további futtatások alatt sem, így 10 gráfra történő futtatással

1-10 prior	GS_A	GS_B	GS_C	GS_D	GS_E
min	0	-22	-12	13	-19
max	19	-1	7	24	-1
átlag	8.3	-9.7	-1	18.2	-8
szórás	6.14	6.79	5.37	3.96	6.48

5.4. táblázat. create_k.cc, 10 gráf, k=800, p=q=5 (a prioritások 1-10-ig mennek)

10 gráf	GS_A	GS_B	GS_C	GS_D	GS_E
min	-22	-27	-12	9	-25
max	-2	-3	7	20	0
átlag	-9.9	-9.7	-1.6	15.8	-10.2
szórás	5.95	6.67	5.15	3.08	6.68

5.5. táblázat. create_k.cc, 10 gráf, 1000 férfi, 1000 nő, k=800, p=10, q=10 (prioritások 1-10-ig mennek)

100 gráf	GS_A	GS_B	GS_C	GS_D	GS_E
min	-22	-27	-12	6	-25
max	3	1	10	26	2
átlag	-10.21	-10.17	-0.39	16.97	-9.9
szórás	5.28	5.41	4.78	3.91	5.31

5.6. táblázat. create_k.cc, 100 gráf, 1000 férfi, 1000 nő, k=800, p=q=10 (prioritások 1-10-ig mennek)

reprezentatív mintát kapunk.

5.1.1. GSA1 algoritmusok tesztelése

Ahhoz, hogy a GSA1 25 féle verziója közötti különbséget meglássuk, olyan gráfokon kell tesztelni, ahol a GS még feltehetőleg nem az optimálisat adja. Így tehát azokon az inputokon, melyekre a GS_D adott egy 1000 vagy k=800-as tesztek esetében 800 élszámú párosítást, nem végeztünk további teszteket GSA1-re.

A create_k általi tesztek közül tehát érdemes a k=800, p=q=5 (vagy kevesebb) és prior_m=prior_w=10 értékekre vizsgálni a GSA1 algoritmusokat. Ezt lát hat juk az 5.7-es táblázatban. Átlagosan az GS_D és valamely rmGS algoritmust használó verziók adják a legjobb megoldást, azonban ez nem minden esetre igaz. Ebben a speciális tesztfájlban, melyből a táblázat készült, a 7. és 10. gráfokra például a BD találta a legnagyobb elemszámú párosítást, míg a 9. gráfra a BA, BB, és BE egyaránt legjobbat adott.

Amennyiben pedig a create_pri.cc algoritmussal készítjük a gráfokat azt vehetjük észre, hogy ha a döntetlenek maximális hossza kicsi (pl. L=2), akkor az algoritmusok által adott megoldások csoportosíthatóak aszerint, hogy melyik GS algoritmust használtuk, és ezeken a csoportokon belül az rmGS különböző változatai nem adnak lényegesen különböző megoldást. Az L paraméter növelésével a csoportokon belül nagyobb különbségek is megfigyelhetőek, L=10 esetén már 4-5 érték különbségek is megjelennek.

5.1.2. GSA2 algoritmusok tesztelése

A GSA2 125 féle verziója miatt az előző táblázatokhoz hasonlót készíteni nem érdemes, mert nem lehet átlátni. Azonban itt is megfigyelhető néhány jelen-

GS_AA-EE	AA	AB	AC	AD	AE	BA	BB	BC	BD	BE
min	14	15	13	17	14	13	14	15	14	14
max	25	25	25	26	25	25	25	25	26	25
átlag	19.4	19.5	19.7	20.7	19.4	19.9	20	19.6	20.6	19.9
szórás	3.89	3.80	4.42	3.68	4.06	3.63	3.43	3.62	3.86	3.63

GS_AA-EE	CA	CB	CC	CD	CE	DA	DB	DC	DD	DE
min	15	15	14	15	15	17	18	16	17	18
max	25	25	25	25	25	27	27	26	27	27
átlag	19.3	19.4	20.1	19.8	19.3	21.9	21.9	21.2	21.8	21.7
szórás	3.80	3.53	3.69	3.35	3.62	3.21	2.96	3.25	3.19	2.98

GS_AA-EE	EA	EB	EC	ED	EE
min	16	16	15	16	15
max	24	25	23	24	24
átlag	19	20.3	19.5	19.7	19.7
szórás	3.33	2.79	3.02	2.83	3.36

5.7. táblázat. create_k.cc, 10 gráf, 1000 férfi, 1000 nő, k=800, p=q=5 (prioritások 1-10-ig mennek)

ség. Először is, itt is 5-ös csoportokra lehet bontani az algoritmusokat, aszerint csoportosítva, hogy a GS és rmGS melyik verzióját használjuk, mert ezeken a csoportokon belül a kapott megoldások nagyon közel állnak egymáshoz. Így az 5.8-as táblázaton megfigyelhetjük, hogy ezen ötös csoportok átlagolva milyen megoldásokat adnak.

5.2. hrGSA1

A hrGSA1 algoritmus 25 féle verziója és az Irving-Manlove féle Heur-I és Heur-C algoritmusok összehasonlítása alapján a hrGSA1 nagyon jó eredményeket mutatott, ami elvárható is két véletlen algoritmussal szemben.

GS_AAA-EEE	AAx	ABx	ACx	ADx	AEx	BAx	BBx	BCx	BDx	BEx
min	21	21	22	22	22	22	22	23	24	22
max	35	35	34	35	35	36	36	37	37	36
átlag	28.7	28.7	28.6	29.7	298.8	30.7	30.7	29.9	31.4	30.5
szórás	5.2	5.39	4.64	4.76	5.14	4.27	4.32	4.17	4.11	4.31

CAx	CBx	CCx	CDx	CEx	DAx	DBx	DCx	DDx	DEx
20	20	22	22	20	23	23	25	24	23.8
34	34	33	34	33.6	38	38	38	38	38
28.9	28.8	28.5	29.6	28.8	31	31	31.1	31.5	31
4.88	5.02	4.17	4.24	4.81	4.76	4.73	4.38	4.6	4.68

EAx	EBx	ECx	EDx	EEx
22	21.6	21.8	21.6	23
35.2	35.2	36.2	37.4	35.4
28.9	28	28.8	30.4	29.46
4.58	4.71	4.70	4.57	4.08

5.8. táblázat. create_k.cc, 10 gráf, 1000 férfi, 1000 nő, k=800, p=q=5 (prioritások 1-10-ig mennek)

6. fejezet

Utószó

A vizsgált GS, rmGS és rwGS verziói közül végeredményben a GS_D, rmGS_D és rwGS_D bizonyult a legjobbnak, azonban nem minden egyes inputra. Viszont mivel a futási idő elég rövid, így megtehető, hogy több változatot is lefuttassunk egy adott problémára, és a kapott eredmények közül a legjobbat kiválasztjuk.

Azonban az, hogy ilyen lényeges különbséget mutat, ha az aktív férfiak közül más sorrendben válasszuk ki a következőt, felveti a kérdést, hogy még milyen módszereket lehetne tesztelni, milyen pontozási módszerekkel állíthatunk fel elsőbbségi sort, melyből az aktív férfiakat kiválasztjuk.

Újabb kérdést vet fel, hogy a GSA1 illetve GSA2 esetében az rmGS illetve rwGS algoritmusok többszöri futtatásával jobb eredményeket érhetünk-e el. Itt az rmGS és rwGS algoritmusok többszöri futtatása között a lényeges különbség, hogy minden kevesebb pontos m szabad férfi $\pi(m)$ extra pontját $1-(\frac{1}{2})^i$ értékre állítjuk vagy pedig a másik lehetőség, hogy minden olyan m szabad férfinál, akinél $2^i * \pi(m)$ páros, $\pi(m) := \pi(m) + (\frac{1}{2})^i$.

Illetve a kettő kombinálásaképpen az újabb futások alatt az aktív férfiakat tároló struktúráját is váltogathatjuk.

A téma szinte kimeríthetetlen, hiszen a világ több országában is használnak Stabil Házasításra és Kórház/Rezidens problémára épülő algoritmusokat.

Irodalomjegyzék

[1] Király Zoltán: Better and simpler approximation algorithms for the stable marriage problem. *Lecture Notes in Computer Science*, 2008.

[2] Robert W. Irving, David F. Manlove: Finding Large Stable Matchings. *Journal of Experimental Algorithmics*, 2009.

[3] Robert W. Irving, David F. Manlove: Approximation algorithms for hard variants of the stable marriage and hospitals/residents problem. *Journal of Combinatorial Optimization*, 2007.

[4] M. M. Halldórsson, K. Iwama, S. Miyazaki, H. Yanagisawa: Improved approximation results for the stable marriage problem. *ACM Transactions on Algorithms*, 2007.

[5] H. Yanagisawa: Approximation Algorithms for Stable Marriage Problems. *PhD Thesis (www.lab2.kuis.kyoto-u.ac.jp/yanagis/thesis_yanagis.pdf)*, 2007

[6] M. M. Halldórsson, R. W. Irving, K. Iwama, D. F. Manlove, S. Miyazaki, Y. Morita, S. Scott: Approximability results for stable marriage problems with ties. *Theoretical Computer Science*, 2003.

[7] K. Iwama, S. Miyazaki, N. Yamauchi: A 1.875-approximation algorithm for the stable marriage problem SODA '07. *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[8] D. Gale, L. S. Shapley: College admissions and the stability of marriage. *The American Mathematical Monthly*, 1969.

[9] LEMON (Library for Efficient Modeling and Optimization in Networks) Graph Lybrary: http://lemon.cs.elte.hu

[10] Christos H. Papadimitriou: Számítási bonyolultság, 1994.