

TDK dolgozat

Készítették:

Harangozó Péter

Programtervező Informatikus, BSc

Szabó István

Programtervező Informatikus, BSc

Egy nehéz gráfelméleti probléma megoldása szuperszámítógép segítségével



Pécsi Tudományegyetem

Természettudományi Kar

Informatika Tanszék

Témavezetők:

Dr. Szabó Sándor

Egyetemi tanár

Zaválnij Bogdán, PhD

Egyetemi adjunktus

Készítették:

Harangozó Péter

Programtervező Informatikus, BSc

III. évfolyam

Szabó István

Programtervező Informatikus, BSc

III. évfolyam

Tartalomjegyzék

Bevezetés	2
1. Matematikai alapfogalmak	3
1.1. Jelölések	3
1.2. Gráfelméleti alapfogalmak	3
1.3. Gráfok színezése és klikkek	6
1.4. Speciális színezések	8
2. Az informatikai háttér és a használt eszközök bemutatása	10
2.1. A szuperszámítógép	10
2.2. Párhuzamos programozás	13
2.3. Felhasznált könyvtárak	14
2.4. Kiterjesztések	17
3. Az algoritmus implementációja	24
3.1. s -klikk mentes színező	26
3.2. Színezés ellenőrző program	28
3.3. Színosztályok újraosztályozását végző program	28
3.4. Párhuzamos program	32
3.5. Ellenőrző program	34
3.6. A program futtatása	35
4. Mérési eredmények	36
4.1. Első mérési eredmények	37
4.2. Szisztematikus mérések	49
4.3. További tesztek	70
4.4. Mérések összefoglalása	72
5. Továbbfejlesztési lehetőségek	74
Összefoglalás	76

Bevezetés

A gráfelmélet a diszkrét matematika egy olyan ága, mely napjainkban egyre népszerűbb és talán a leggyorsabban fejlődők közé is sorolhatjuk.

A gráf egy diszkrét modell, melynek definíciója megengedi, hogy tetszőleges szándékot, jelentést tulajdonítsunk a csúcsoknak és éleknek. Rengeteg érdekes és szép gráfelméleti tétel és algoritmus született egy-egy valós életbeli problémára adott válaszként. Általában absztrakt gráfokkal foglalkozunk, azaz megfeledekezünk a hozzá társított jelentésről, ezt tükrözi a szóhasználat is. Az absztrakció magával vonja azt is, hogy egyetlen gráf a valós élet számos problémájának reprezentációja. Ebből az is következik, hogy ha ezzel a gráffal kapcsolatosan megválaszolatlan kérdéseink vannak, azok megválaszolásával egyszerre rengeteg problémára adunk megoldást.

A dolgozatunkban egy olyan gráfelméleti kérdésre keresünk választ, ami napjainkban egyre több matematikust és informatikust foglalkoztat.

A kérdés megválaszolásához a legnagyobb segítségünkre a párhuzamos programozás lesz. A programunkat C++ nyelven írtuk, a program párhuzamosításához pedig MPI-t (Message Passing Interface) használtunk. Ezen kívül pedig az egyetemi tananyagot túlmutató tudásra is szert tettünk – a szuperszámítógépek programozására. Sikerült alkotnunk egy olyan algoritmust, amely a szuperszámítógép erőforrásait hatékonyan kihasználja. Úgy véljük, hogy a szupergép kihasználásával (annak számítási kapacitásának köszönhetően) egy személyi számítógéphez képest áttörést értünk el a megoldható problémák méretét illetően.

Dolgozatunkban elsőként bemutatjuk a program matematikai és informatikai hátterét, majd szót ejtünk a pécsi szuperszámítógépről és az MPI-ről. Végül pedig bemutatjuk a programot, és a mérési eredményeket.

1. fejezet

Matematikai alapfogalmak

1.1. Jelölések

A továbbiakban a következő jelöléseket fogjuk használni:

V – csúcsok (vertex/vertices)

E – élek (edges)

G – gráf (graph)

C – színosztály (color class)

r – színosztályok száma

q – kvázi színosztályok száma

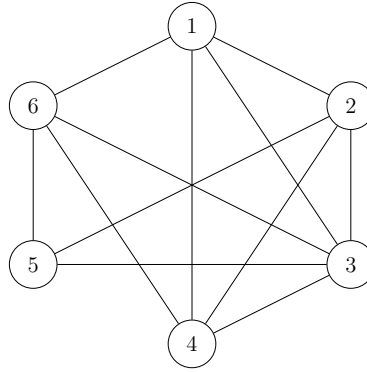
k – a keresett klikk mérete

1.2. Gráfelméleti alapfogalmak

A következő alfejezetben a gráfelmélet alapvető fogalmait ismertetjük, mint a gráf, élek gráfokban, két csúcs szomszédsága, irányított és irányítatlan gráfok, részgráf, teljes gráf, foksám, független ponthalmaz, szomszédsági mátrix, klikk, színezés.

Gráf

A gráf a matematikai gráfelmélet és a számítógép-tudomány egyik alapvető fogalma. A gráf dolgok (csomópontok, azaz csúcsok) és rajtuk értelmezett összeköttetések (élek) halmaza. Egy G gráf tehát csúcsok V és rajtuk értelmezett élek E halmazából áll. [Har69, p. 9] Egy gráfot megadhatunk csúcsainak és éleinek felsorolásával, vagy szemléletesebben egy diagram formájában, ahol a pontok felelnek meg a gráf csúcsainak, az őket összekötő ívek pedig az éleknek. A két megadási mód ekvivalens, azaz a gráf pusztán egy struktúra, semmilyen megjelenítési információt nem tartalmaz, így különböző diagramok is tartozhatnak ugyanahhoz a gráfhoz. Ilyen ábrázolási mód az 1.1 ábrán látható.



1.1. ábra. Egy 6 csúcsú 12 élű gráf.

Élek

A gráf egy éle két csúcsot köt össze. Azt az élt, amely ugyanabba a csúcsba fut, amelyikből kiindult, hurokélnek nevezzük. Két pont között futhat két él is, ilyenkor dupla-élről (adott esetben dupla hurokélről) beszélünk.

Szomszédság

Egy gráfban két csúcs akkor szomszédos, ha van köztük futó él. [Har69, p. 9]

Írányított és irányítatlan gráfok

Alapértelmezésben a gráf irányítatlan, azaz nem teszünk különbséget A -ból B -be, illetve B -ből A -ba menő élek között. [Har69, p. 9] Ezzel szemben az irányított gráfokban (angolosan: digráf) a két iránynak irányított élek felelnek meg. [Har69, p. 10]

Részgráf

R gráf egy G gráf részgráfja, ha R megkapható G -ből csúcsok és élek elhagyásával. A fenti elhagyások halmaza lehet üres is. Azaz G a G gráf részgráfja. Ha ezt ki akarjuk zárni, akkor valódi részgráfról beszélünk.

Másképpen, a G gráf részgráfja egy olyan gráf, melynek csúcs- és élhalmazai részhalmazai G csúcs- és/vagy élhalmazainak. Azt mondjuk, hogy G tartalmazza H -t, ha G -nek van olyan részgráfja, mely megegyezik vagy izomorf H -val (az adott helyzettől függ). [Har69, p. 11] Egy H részgráf feszített részgráfja G -nek, ha H bármely két csúcsára igaz, hogy szomszédosak H -ban akkor és csak akkor, ha szomszédosak G -ben. Másként fogalmazva H feszített részgráfja G -nek, ha úgy adódik, hogy vesszük G bizonyos csúcsait és minden köztük futó élt. Egy gráfot, amelyik nem tartalmazza H -t feszített részgráfként H -mentesnek hívunk.

Teljes gráf

A teljes gráf olyan egyszerű gráf, amelynek minden csúcsa össze van kötve minden más csúccsal, vagy másként fogalmazva: csúcsai páronként szomszédosak. Az n csúcsú teljes gráfot K_n -el jelöljük. [Har69, p. 16]

Fokszám

Egy gráfban egy csúcs fokszáma azoknak az éleknek a száma, amik illeszkednek a csúcsra (azaz valamelyik végpontjuk a szóban forgó csúcs). Egy hurokél kettővel járul hozzá a csúcs fokszámához. [Har69, p. 14]

Egy v csúcs fokszámának jele $\deg(v)$.

Független ponthalmaz

A G gráf egy olyan részgráfja, amelyben semelyik két csúcs közt nem fut él. [Har69, pp. 40-41]

Szomszédsági mátrix

Egy véges irányított vagy irányítatlan n csúcsú G gráf szomszédsági mátrixa az az $n \times n$ -es mátrix, amelynek a nem a főátlóban szereplő a_{ij} eleme az i csúcsból a j csúcsba vezető élek száma, míg a főátlóban található a_{ii} , vagy az i csúcsnál lévő hurkok számának kétszerese vagy csak a hurkok száma. Minden egyes gráfnak létezik egy egyedi szomszédsági mátrixa, mely nem szomszédsági mátrixa egyetlen más gráfnak sem, így a szomszédsági mátrix az adott gráf egy reprezentációjának tekinthető. A véges egyszerű gráfok speciális esetében a szomszédsági mátrix egy nullákból és egyesekből álló mátrix. Ahol a főátlóban csak nullák szerepelnek. Ha a gráf irányítatlan, akkor a szomszédsági mátrixa szimmetrikus. [CLRS01, p. 446] A szemléletesebb ábrázolhatóság kedvéért mi a szomszédsági mátrixokban egyesek helyett pontokat ($\bullet$) nullák helyett pedig üres helyeket használunk. Vegyük az 1.1 ábrán látható G gráfot. Ekkor G szomszédsági mátrixát az 1.1 táblázat adja meg.

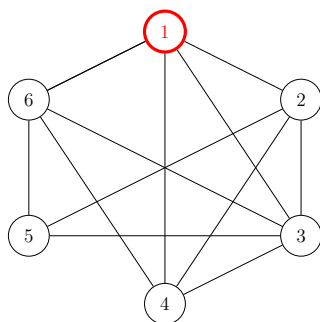
	1	2	3	4	5	6
1		•	•	•		•
2	•		•	•	•	
3	•	•		•	•	•
4	•	•	•			•
5		•	•			•
6	•		•	•	•	

1.1. táblázat. $A(z)$ 1.1 ábrán látható gráf szomszédsági mátrixa.

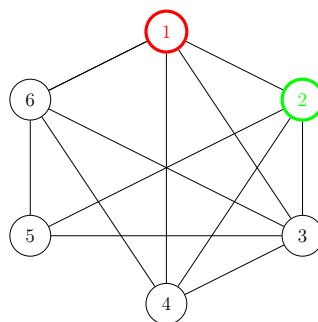
1.3. Gráfok színezése és klikkek

Gráfok színezése

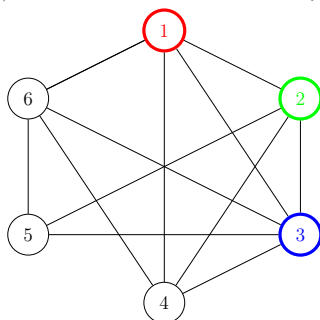
Gráfok színezésének nevezzük, amikor színeket rendelünk egy gráf csúcsaihoz, esetleg éleihez, ezeket a színeket szokás szerint számokkal jelöljük. Egy gráf legális színezése csúcsainak olyan színezése, ahol összekötött csúcsok eltérő színeket kapnak. Az 1.2f ábrán egy legális színezés folyamata látható. [KRS06]



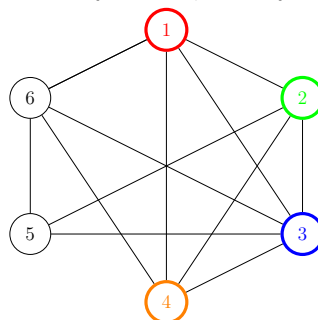
(a) Az 1-es csúcs bekerül az első színosztályba.



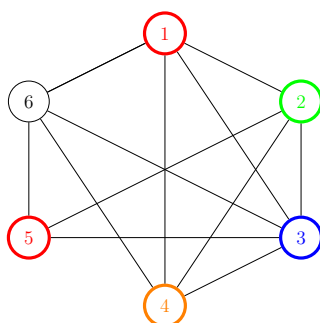
(b) A 2-es csúcs szomszédja az 1-esnek, ezért az új színosztályba kerül



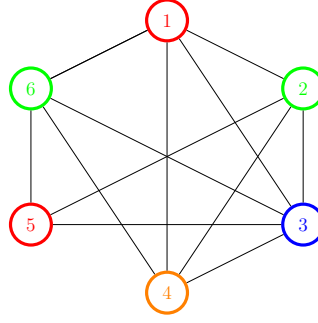
(c) A 3-as csúcs szomszédja az 1-esnek és a 2-esnek is, ezért ez is egy új színosztályba kerül.



(d) A 4-es csúcs szomszédja az előző három csúcsnak, ezért ez a 4. színosztályba kerül.



(e) Az 5-ös csúcs az 1-esnek nem szomszédja, ezért azonos színosztályba kerülhet vele.



(f) A 6-os csúcs pedig szomszédja az 1-esnek, de a 2-esnek nem, ezért a 2-essel azonos színosztályba kerülhet.

1.2. ábra. Vegyük a(z) 1.1 ábrán látható G gráfot. A(z) 1.2a-tól a(z) 1.2e ábráig egy színezési folyamatot láthatunk, ezt a módszert használjuk a programunk során is, később a(z) 3.1 fejezetben lesz róla szó. Ez a folyamat G gráfnak egy legális színezését állítja elő. A színezés eredménye a(z) 1.2f ábrán látható.

Kromatikus szám

Egy G hurokmentes gráf r színnel színezhető, ha minden csúcsot ki lehet színezni r szín felhasználásával úgy, hogy bármely két szomszédos csúcs színe különböző legyen. G kromatikus száma $\chi(G) = r$, ha G r színnel kiszínezhető, de $r - 1$ színnel nem. Egy ilyen színezésnél az azonos színt kapott csúcsok halmazát színosztálynak nevezzük.

Ha hurokél csatlakozna egy csúcshoz, akkor azt a csúcsot nem lehetne kiszínezni. Ha két csúcs között dupla él fut, az nem befolyásolja a csúcsok színezhetőségét, tehát a dupla élek egyikét akár ki is törölhetjük. Ekkor dupla-él mentes és hurokélmentes gráfokra fókuszálhatjuk a figyelmünket, amikor színezésről beszélünk. [KRS06]

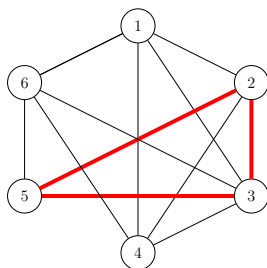
Klikk

A klikk olyan részgráf, aminek bármely két csúcsa között van él. Más szóval, egy olyan részgráf, ami teljes gráf. A klikk csúcsainak számát a klikk méretének vagy rendjének is mondják. A k csúcsú klikket röviden k -klikknek is nevezik.

Sok esetben a gráf legnagyobb klikkje (a maximális klikk) az érdekes. Egyes szerzők klikk alatt automatikusan maximális klikket értenek. A G gráf klikkszám a legnagyobb klikkjének a mérete; a jele $\omega(G)$. [And97]

Háromszög

Olyan klikk, amelynek három csúcsa van. Az 1.3 ábrán látható egy háromszög.



1.3. ábra. Egy háromszög (3 méretű klikk) a(z) 1.1 ábrán látható gráfban.

Klikkprobléma

A G gráf klikkszám a legnagyobb klikkjének a mérete; a jele $\omega(G)$. Egy gráfnak több legnagyobb klikkje is lehet, helyesebb lenne egyik legnagyobb klikkjéről beszélnünk. A klikkszám kiszámítása (a klikk probléma) NP-nehéz. Mi a dolgozatunk során a klikk-problémának egy alternatív feladatával foglalkozunk. Ez pedig a k -klikk probléma. Ekkor nem a gráf legnagyobb klikkjének megállapítása a cél, hanem az, hogy megállapítsuk van-e a gráfban k méretű klikk. Egy k -klikk megtalálása NP-teljes feladat. [CLRS01, pp. 1086-1089]

Hasznos lehet a klikk méretet felülről becsülni, erre egy lehetséges módszer (az i -edik csúcs fokszámát d_i -vel jelölve) [And97]

$$\omega(G) \geq \sum_{i=1}^n \frac{1}{n-d_i}$$

Kromatikus szám kapcsolata a klikkszámmal

Egy gráf kromatikus számának alsó becslését a gráf klikkszám adja (tehát a gráf kromatikus száma mindig nagyobb, vagy egyenlő, mint a gráf klikkszám), mert ha egy adott G gráfnak a legnagyobb klikkje n méretű, akkor a szóban forgó n csúcs mindegyike össze van kötve az összes többivel, így kiszínezésükhöz n színre van szükségünk. Ebből adódik, hogy minimum n szín szükséges a gráf színezéséhez.

Másként, a színezés egy felső becslést ad a klikkszámra, ezt a dolgozatunk során is használtuk.

1.4. Speciális színezések

Kvázi színezés

Legyen $G = (V, E)$ egy véges, egyszerű gráf. Legyenek $Q_1, \dots, Q_k$ a V partíciói, és legyen a t_i az a szám, ami egy Q_i partíció csúcsait összekötő élek számosságát adja meg. Más szavakkal a Q_i által G -ben kifeszített részgráfról beszélünk, és t_i -vel jelöljük ennek a részgráfnak az élszámát. Legyen $t = t_1 + \dots + t_j$. Ha $t = 0$, akkor mindegyik Q_i független halmaz G -ben, és G egy legális színezésének színosztályai lesznek. A $C_1, \dots, Q_j$ particionálást a G egy (j, t) -kvázi színezésének fogjuk nevezni. Azt is mondhatjuk, hogy ha t közeli nullához, akkor a (j, t) -kvázi színezés közel van egy j -színezéshez. A t mennyiségét úgy is nevezhetjük, hogy a zavaró élek száma. [Sza14, p. 128]

Zavaró él

A kvázi színezés eredménye, hogy az eddig élmentes színosztályokban előfordulhatnak élek, tehát a színosztályról már nem mondható el, hogy független pontthalmaz. Ezeket az éleket hívjuk zavaró éleknek, hiszen zavarunk minket abban, hogy ez a kvázi színezés egy legális színezés legyen. Ezek az élek lesznek a későbbiek során az eliminálandó élek. Ha lenne egy olyan módszerünk aminek a segítségével törölhetjük ezeket, akkor egy legális színezést kapunk. A dolgozat egy ilyen módszert mutat be.

Háromszögmentes színezés

Legyen C részhalmaza V -nek, ha G gráf C által feszített részgráfja nem tartalmaz háromszöget, akkor C -t háromszögmentes halmaznak hívjuk. V -nek egy $C_1, \dots, C_r$ felosztását

háromszögmentes felosztásnak hívjuk, ha C_i háromszögmentes részhalmaza V -nek, minden i -re, $1 \leq i \leq r$. Tehát, G -nek r színnel való színezését háromszögmentesnek hívjuk, ha $C_1, \dots, C_r$ színosztályok háromszögmentes részhalmazai V -nek. [SZ12]

Következtetés: Ha G -nek van egy háromszögmentes színezése r színnel, akkor $\omega(G) \leq r \times (3 - 1)$.

Bizonyítás:

Legyen D egy k -klikk G -ben, és legyen G -nek egy háromszögmentes színezése r színnel. D -nek legfeljebb $3 - 1$ csúcsa lehet minden $C_1, \dots, C_r$, színosztályban. Ebből következik, hogy $k \leq r \times (3 - 1)$, azaz $\omega(G) \leq r \times (3 - 1)$.

s -klikk mentes színezés

s -klikk: egy s csúcsból álló klikk. A háromszögmentes színezést tovább fokozva eljuthatunk az s -klikk mentes színezésig. [SS68]

Legyen C részhalmaza V -nek, ha G gráf C által feszített részgráfja nem tartalmaz s -klikket, akkor C -t s -klikk mentes halmaznak hívjuk. V -nek egy $C_1, \dots, C_r$ felosztását s -klikk mentes felosztásnak hívjuk, ha C_i s -klikk mentes részhalmaza V -nek, minden i -re $1 \leq i \leq r$. Tehát, G -nek r színnel való színezését s -klikk mentes színezésnek hívjuk, ha $C_1, \dots, C_r$ színosztályok s -klikk mentes részhalmazai V -nek.

Következtetés: Ha G -nek van egy s -klikk mentes színezése r színnel, akkor $\omega(G) \leq r \times (s - 1)$.

Bizonyítás:

Legyen D egy k -klikk G -ben, és legyen G -nek egy s -klikk mentes színezése r színnel. D -nek legfeljebb $s - 1$ csúcsa lehet minden $C_1, \dots, C_r$, színosztályban. Ebből következik, hogy $k \leq r \times (s - 1)$, azaz $\omega(G) \leq r \times (s - 1)$. [SZ12]

Az s -klikk mentes színezés általánosításával eljutunk a conditional coloring-hoz. Azaz a legális színezés feltételét, miszerint azonos színosztályba nem kerülhet két szomszédos csúcs, egy másik feltételre cseréljük. Ilyen például ha ez a feltétel az, hogy a színosztályok nem tartalmazhatnak köröket. Így az s -klikk mentes színezés is beleillik a conditional coloring elméletébe. [Har85]

2. fejezet

Az informatikai háttér és a használt eszközök bemutatása

A munkánk egy fontos része a szuperszámítógép(ek) használatának elsajátítása. Ebben a fejezetben ennek a menetét mutatjuk be. Továbbá bemutatásra kerül a program elkészítése során felhasznált könyvtárak, valamint a párhuzamosítás egy lehetséges formája, a postafiók elv.

2.1. A szuperszámítógép

Általános információk

A NIIF Intézet a 2011-ben telepített 4 új szuperszámítógépe közül az egyiket a Pécsi Tudományegyetemen helyezte el. A szuperszámítógéphez való hozzáférés kulcsos azonosítást használó SSH parancssori, valamint grid köztes réteg segítségével lehetséges. Általános hozzáférési szabály az, hogy a számítást végző csomópontokhoz (compute node) a felhasználók nem férhetnek közvetlenül hozzá. A felhasználók minden helyszínen egy úgynevezett előtét szerverre(frontend) lépnek be, azon fordítanak, és arról adják fel a feladataikat a számítási csomópont(ok) felé. Az eredményeket is e szerveren kapják meg, illetve ezt a szerveret használhatják a számítások eredményéül előállt adathalmaz grafikus megjelenítésére is. A számítógépeket kötegetelt módban lehet használni, amely azt jelenti, hogy egy számítási feladatot egy erre szolgáló leírónyelv segítségével fogalmazzunk meg: többek között meg kell mondanunk, hogy mi az a bináris állomány, amelyet futtatni kívánunk, hány processzoron, milyen futtatói környezetben, melyek az input paraméterei, állományai, és melyik állományba irányítjuk át az eredményeket. Az így megfogalmazott számítási feladatot várakozósorba helyezzük, majd az egyes helyszíneken telepített ütemezők azokat elindítják, és futtatják, erről a Programok fordítása és futtatása a szuperszámítógépen bekezdésben írunk. [NIIa]

A Pécsi szuperszámítógép

A számítógép fő paraméterei:

A berendezés egy SGI UltraViolet 1000 rendszer, amely NUMA architektúrával rendelkezik: a rendszerben lévő modulok olyan speciális kapcsolattal vannak összekötve, amely gondoskodik arról, hogy a teljes rendelkezésre álló memóriát minden egyes processzor egy egységes címtérben ki tudja címezni. A speciális kapcsolat hardveresen biztosítja a cache koherenciát, valamint gondoskodik arról, hogy a memóriaelérés sebessége és latenciája lehetőség szerint független legyen a processzor és a memória elhelyezkedésétől a rendszeren belül.

A felhasználó szempontjából tehát a rendszer egyetlen számítógépnek látszik, amelyben

- 192 darab Intel X7542 Xeon Processzor, 6 core/2.66Ghz/18M/5.86GT/s, összesen 1152 db processzor mag
- 1536 x 4GB DDR3 rendszer memória, vagyis 6144 GB teljes megosztott memória áll rendelkezésre.

A rendszer fő fizikai paraméterei:

A rendszer 3 darab 24 inch-es szekrényből és 2 darab 19 inch-es léghűtéses rack szekrényből áll. Előbbiekben a számítási egységek találhatók, tömegük egyenként 1134 kg. A másik két szekrény a tároló és vizualizációs egységeket tartalmazza, tömegük 780 kg illetve. 475 kg. A rendszer maximális villamos fogyasztása 59.9 kW, a valós várt fogyasztás 44-49kW körüli érték.

Programok fordítása és futtatása a szuperszámítógépen

Belépés SSH segítségével: A jelszavas felhasználó-azonosítás nem engedélyezett, csak az RSA vagy DSA kulcsok segítségével történő. A publikus kulcsokat a regisztrációs folyamat során kell megadnunk.

ssh -l usernev login.pecs.hpc.niif.hu

A programok, fordítása az előtétmodulon zajlik, a hagyományos számítógépeken megszokott módon. Soros programok futtatására szintén az előtét modulon van lehetőségünk, a hagyományos módon. Párhuzamos programok esetén már nem ilyen egyszerű a helyzet. A szuper gépeken CPU óra (gépidő) alapú ütemezés működik. Ez azt jelenti, hogy a fogyasztási időtartamra (hónap) az elérhető maximális gépidő fel van osztva a felhasználók között. Minden UNIX felhasználó egy vagy több számlához (Account) tartozhat. Az ütemező számla egy adott HPC-s projekthez tartozik, hasonló a UNIX csoportéhoz. Szupergépes feladatokat csak az egyes számlák terhére lehet indítani. A CPU óra az elhasznált idő (Walltime) és az igényelt processzorok számának szorzata. Például a budapesti gépen 2 db node (48 processzor mag) foglalása 30 percre $48 * 30 = 1440$ CPU

perc = 24 CPU óra. A CPU órát az ütemező a futás kezdetétől a befejezésig méri.

A SLURM ütemező működése, használata: [NIIb] Az ütemezőbe batch scripteket kell beküldeni, melynek felépítése:

```
#!/bin/bash
#SBATCH -A account
#SBATCH -job-name=jname
#SBATCH -n 18
#SBATCH -time=01:30:00
/opt/nce/bin/numainfo
mpirun $SLURM_NTASKS ./parallel p_hat700-3.clq p_hat700-3_3_63.chk 3 62
```

A script értelmezése:

- **-A account:**
Azt a számlát kell feltüntetni, amelynek a terhére szeretnénk indítani a feladatot.
- **-job-name=jname:**
a feladat nevét jelzi, fontos szerepe nincs, a feladat azonosítására szolgál.
- **-n 18**
a magok számát jelzi, a példában a feladat 18 magon indul el.
- **-time=01:30:00**
az időkorlátot adja meg, ha ennyi idő alatt nem futott le a program akkor leállításra kerül.
- **mpirun**
A 2.3 fejezetben leírt mpirun parancs, annyi különbséggel, hogy a folyamatok számát a script-ben adtuk meg.

Hasznos utasítások a szupergépen:

- **sbatch**
Ezzel a paranccsal tudjuk beküldeni az előzőleg megírt scriptet: `sbatch pelda.sh`. Sikeres beküldést követően, ha sorra került a programunk, akkor a script mellé létrejön egy kimeneti fájl, ami tartalmazza a program által konzolra kiírt tartalmat, illetve, hogy melyik processzorok végezték el a feladatot.
- **squeue**
Ezzel tudjuk lekérdezni a sorban levő feladatokat, a kimenet megmutatja a már futó és még várakozó feladatokat, azok azonosítóját, nevét, beküldőjét, futásidejét is.
- **scontrol show job ID**
Ezzel tudunk bővebb információt kérni az ID azonosítójú feladatról, megmutatja többek között azt is, hogy mennyi az időkorlát.

- **sbalance**

A felhasználónkhoz tartozó számlát listázza ki, hogy az adott hónapban mennyi felhasználó és mennyi felhasználható gépidő van még.

A pécsi szupergépen a pécsiek előnyben vannak részesítve a többi felhasználóval szemben, azonban bárki használhatja ezt a gépet is, aki a NIIF-nél regisztrál.

2.2. Párhuzamos programozás

Az algoritmusunk alapelvéről, a postafiók-elvről és a párhuzamosításra használt eszközről, az MPI-ről lesz szó a következőkben.

Postafiók-elv

A postafiók-elvet akkor szokás használni, amikor szétszjtjuk a részproblémákat a gépek között. A postafiók tárolja a még feldolgozásra váró feladatokat. Kétféle folyamat segítségével kezeljük, az egyik a termelő, a másik a fogyasztó. A termelő feladata, hogy betegye a postafiókba a leveleket, amik itt a megoldandó feladatokat jelentik. A fogyasztó figyel, hogy van-e a postafiókban újabb levél. Ha van, kiveszi, elvégzi a feladathoz kapcsolódó utasításokat, megoldja a feladatot. Ez után ismét megnézi, hogy van-e következő feladat a fiókban, és kezdődik az egész procedura előlről. Általában több fogyasztó van, és egy termelő. De a feladat eredménye szempontjából termelő is válhat a fogyasztóból. Ez akkor fordulhat elő, amikor visszaadja a kiszámolt eredményt, és azt egy másik fogyasztó feldolgozza.

Mivel nincs előre meghatározva az egyes feladatok nehézsége, a terheléelosztás várhatóan automatikusan megvalósul. Mivel sok fogyasztó van, előfordulhat az is, hogy amíg az egyik folyamat hosszabb ideig dolgozik egy feladattal, azalatt a másik megold több problémát is. A másik nagy előnye ennek a rendszernek a hibatűrő képessége. Ha az egyik folyamat valamilyen okból elromlik, és a megoldandó feladatnak csak a másolatával rendelkezett, ilyen esetben észreveszi a többi fogyasztó, hogy nem lett megoldva az adott feladat, sokáig nem jött eredmény, ekkor egy másik folyamat fogja magához venni. Mindebből a felhasználó csak annyit lát (ha jól van megírva a program), hogy tovább tart az egyik feladat megoldása, mint ahogy azt eredetileg feltételezte. Tehát egy processzor meghibásodása még nem jelent akkora problémát, de ha csak egy termelő van és pont az romlik el, vagy a postafiók és a folyamatok közötti kommunikációs csatorna (a rendszersín) , ezeket a hibákat már nem tudja kezelni.

A feladatok betöltését és kivételét a postafiókból a szemaforok segítségével is végezhetjük. Azért lehet rájuk szükség, mert az osztott memóriás rendszerekben előfordulhatnak olyan esetek, amikor a postafiókot egyszerre több folyamat is szeretné írni. A szemafor egy versenyhelyzetmentes atomi változó, amely értékének növelésével, illetve csökkentésével blokkolhatjuk a folyamatokat. A programozónak nem minden esetben kell használnia ezt

a változót, hiszen a párhuzamos programozáskor használt függvényeknek köszönhetően nem lép fel az egyszerre olvasás-írás problémája. Ilyen függvényeket például a későbbiekben bemutatásra kerülő MPI tartalmaz. [VZ14, o. 45]

2.3. Felhasznált könyvtárak

MPI – Message Passing Interface

Ebben a bekezdésben említett példák és definíciók a [VZ14] könyvben találhatóak. Tudományos feladatok megoldása során sokszor előfordulnak hosszabb ideig tartó számítások. Ezek között a számítások között lehetnek olyan (rész)feladatok amelyeket, ha egy időben végzünk, akkor a számítás ideje nagyságrendileg lerövidül. Az ilyen feladatokat párhuzamos programozással tudjuk megoldani. Az üzenetküldéses párhuzamos programozási paradigma alkalmazására a PVM (Parallel Virtual Machine) és az MPI (Message Passing Interface) használatos, azonban míg az előbbi lassan visszaszorult, addig az utóbbi elterjedt napjainkban. Az MPI által megvalósított üzenetküldéseket használhatjuk adatátvitelre (két vagy több folyamat között) illetve kommunikációra is. [VZ14, p. 3]

Egy C/C++ nyelven megírt MPI program formájában nem sokban különbözik egy soros C/C++ programtól, a legfőbb eltérések az MPI előtaggal ellátott függvényhívások. Ugyanúgy használjuk a program elején az include-okat és a programot továbbra is le kell fordítani, ehhez egy külön felparaméterezett fordítót kell használni, ami a megfelelő paraméterekkel lefordítja és linkeli a szükséges könyvtárakat. A C++ programokhoz az MPIC++ vagy az MPICxx használható, ez a telepített MPI programtól függ, ami meghívja a rendszerben telepített C++ fordítót (g++, icc stb.). MPI program fordítása során ugyanazokat a kapcsolókat használhatjuk, mint egy soros program fordítása során. Pl.: a -o kapcsolóval megadhatjuk a kimeneti fájl nevét, illetve a -O3 kapcsolóval erőteljesebb optimalizálást érhetünk el. [VZ14, p. 15]

Példa:

```
mpic++ -O3 -o hello hello.cpp
```

Sikeres fordítás után egy újabb programot kell meghívunk a futtatáshoz, ez az mpirun. Az -np kapcsoló határozza meg azt, hogy hány szálon szeretnénk futtatni a lefordított programot. Tehát az mpirun feladata csak annyi, hogy több szálon elindítja ugyanazt a programot.

Példa az MPI program futtatására:

A hello programot 4 példányban elindítja:

```
mpirun -np 4 ./hello
```

Egy alap MPI program felépítése:

```

#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char** argv){
    int id, nproc;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    cout<<"Process "<<id<<" say Hello to World!"<<endl;
    MPI_Finalize();
}

```

A program párhuzamos része az MPI_Init() függvénnyel indul és az MPI_Finalize() függvénnyel fejeződik be. Az MPI_Comm_rank() az adott folyamat azonosítóját, míg az MPI_Comm_size() az összes folyamat számát adja meg. [VZ14, p. 16]

Üzenetküldés

```

MPI_Send(
    void *buffer //A kuldo puffer cime. (A kuldott adat)
    int count //A kuldendo elemek szama.
    MPI_Datatype datatype //A kuldendo adat tipusa.
    int destination //A cimzett rangja.
    int tag //Az uzenet cimkeje.
    MPI_Comm comm //MPI kommunikator.
)

```

[VZ14, p. 20]

Üzenetfogadás

```

MPI_Recv(
    void *buffer //A fogado puffer cime. (A fogadott
    adat)
    int count //A fogadott elemek szama.
    MPI_Datatype datatype //A fogadott adat tipusa.
    int source //A kuldo rangja.
    int tag //Az uzenet cimkeje.
    MPI_Comm comm //MPI kommunikator.
    MPI_Status status //Statusz objektum.
)

```

[VZ14, p. 20]

Üzenetszórás

Ez egy speciális üzenetküldési eszköz az MPI-ban, ez arra az esetre jó, ha azt szeretnénk, hogy minden folyamat ugyanazon az adathalmazon dolgozzon, és ne kelljen minden folyamatnak egyesével elküldeni ezt az adathalmazt.

```
MPI_BCast(  
    void *buffer                //A kuldo/fogado puffer cime.  
    int count                   //A kuldendo elemek szama.  
    MPI_Datatype datatype //A kuldendo adat tipusa.  
    int root                     //A mester rangja.  
    MPI_Comm comm               //MPI kommunikator.  
)
```

[VZ14, p. 23]

Mentési pont – Checkpoint

Hosszú ideig elhúzódó feladatoknál érdemes úgynevezett mentési pontokat használni, melynek célja, hogy a program váratlan leállásakor, vagy félbeszakításkor ne vesszen el az addig megoldott feladat. Erre egyszerű megoldás lehet a checkpoint fájl használata, mi is használtunk ilyen fájlt a program írása során, ez később bemutatásra kerül a 2.4 fejezetben.

Boost library

A boost ingyenes, szakmailag elfogadott hordozható C++ könyvtárakat biztosít, melyek többek között lehetővé teszik a dinamikus bitsetek vagy a lexical cast használatát. Ezek a könyvtárak jórészt csak headerekből állnak fordítható forráskód nélkül, emiatt könnyen hordozható. [boo]

Dynamic bitset A bitset egy olyan tömbnek fogható fel, amely csak egyesek és nullák tárolására képes, és értelmezhetőek rajta a bitműveletek(bitenkénti &, bitenkénti |).

A boost library pedig lehetővé teszi ezeknek a bitseteknek a dinamikus használatát, tehát a program futása során változtatható a méretük, ez pedig azért fontos számunkra, mert a program fordításának idejében nem tudjuk és nem is áll szándékunkban előre megadni, hogy mekkora gráfot fogunk később beolvasni.

A program során általánosan használt, a dynamic bitsethez köthető metódusok:

count(): a bitsetben tárolt egyesek számát adja vissza.

any(): visszatérési értéke false, vagy true lehet. Ha nincs a bitsetben egyetlen 1-es bit sem, akkor false, egyébként true.

none(): az any ellentéte, ha egyetlen 1-es bit sincs a bitsetben akkor true, egyébként

false.

`size()`: a bitset hosszát/méretét adja vissza.

2.4. Kiterjesztések

DIMACS gráf formátum – clq

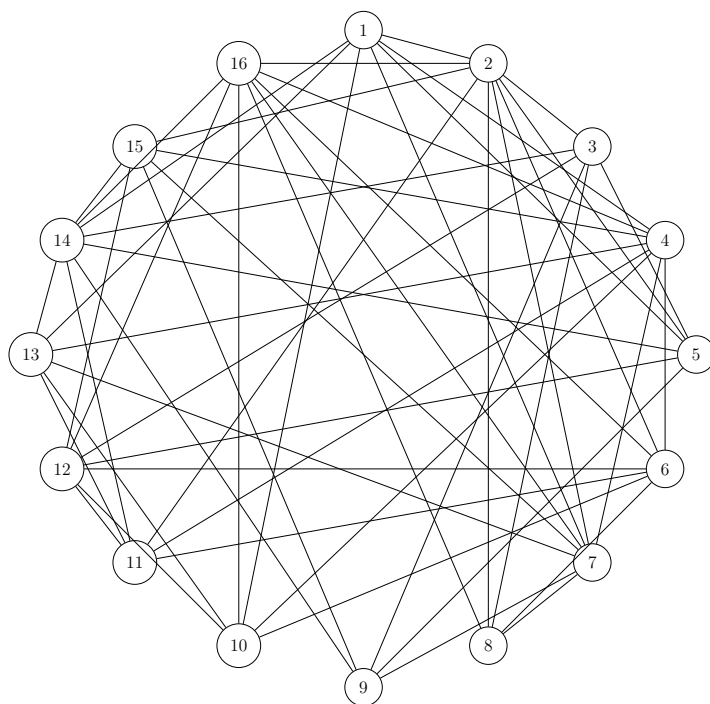
A DIMACS a Center for Discrete Mathematics and Theoretical Computer Science rövidítése. A DIMACS fő kutatási területe a diszkrét matematika, azonban jelentős informatikai fejlesztések és algoritmusok kidolgozásában vesznek részt. [DIM]

A clq fájl egy gráfot ír le, pontosabban a gráfnak azon csúcsait, amelyek között fut él. A fájlban lehetőség van megadni kommenteket is, ide általában a gráf neve szokott kerülni, de előfordulhat a legjobb klikk becslés is a kommentben. Ezek a sorok 'c' karakterrel kezdődnek. A komment sorok után jönnek a gráf tulajdonságai (csúcsok száma, élek száma). Ez a sor 'p' karakterrel kezdődik, egy szó következik utána, ez általában az 'edge' (él) szó, majd a csúcsok száma és az élek száma. Végül pedig jön maga a gráf leírása, ezek a sorok 'e' karakterrel kezdődnek, mint edge, hiszen két csúcs közti élt ad meg, az 'e' karakter után pedig egy-egy csúcsot jelölő számpár. Irányítatlan gráfokról beszélünk, ezért beolvasáskor az mind a két élt meg kell adni (`graph[ i ][ j ]` és `graph[ j ][ i ]`). A következő clq fájl a 2.1 gráf leírása, amelynek a 2.1 táblázat tartalmazza a szomszédsági mátrixát.

Egy példa a clq fájlra.

```
c FILE: graf.clq
c
p edge 16 54
e 1 2
e 1 4
e 1 5
e 1 7
e 1 10
e 1 13
e 1 14
e 2 3
e 2 5
e 2 6
e 2 7
e 2 8
e 2 11
e 2 15
```

e 2 16
e 3 5
e 3 8
e 3 9
e 3 12
e 3 14
e 4 6
e 4 7
e 4 10
e 4 11
e 4 13
e 4 15
e 4 16
e 5 9
e 5 12
e 5 14
e 6 8
e 6 10
e 6 11
e 6 12
e 6 16
e 7 8
e 7 9
e 7 13
e 7 15
e 7 16
e 8 16
e 9 14
e 9 15
e 10 12
e 10 13
e 10 16
e 11 12
e 11 13
e 11 14
e 12 15
e 12 16
e 13 14
e 14 15
e 14 16



2.1. ábra. 16 csúcsú, 54 élű gráf.

DIMACS színezés leíró fájl – col

Szintén a DIMACS által használt fájlformátum, hasonló a clq-hoz. Itt is a 'c' karakter jelzi a komment sort. Azonban ebben a fájlban egy színezés eredményét tároljuk. A kommentsorokban célszerű feltüntetni, hogy melyik gráfnak a színezése, milyen színezés (élmentes, s -klikk mentes stb...). A kommentsorok után következik a színezés végeredménye, a sor az 's col' előtaggal (solution of coloring - színezés eredménye) kezdődik és a színosztályok száma található a sor végén. A többi sorban pedig az, hogy melyik csúcs, melyik színt kapta, ezek a sorok 'l' karakterrel kezdődnek.

A fájl: Ez az előző graf.clq (2.4) gráfnak a színezését írja le. Az általunk használt col fájlok annyiban térnek el a szabvány dimacs col fájlától, hogy mi az s -klikk mentes színezés eredményét írjuk le, tehát a solution of coloring sorban, ez esetben a 3-as szám azt jelenti, hogy 3 s -klikk mentes színosztályt kaptunk, a klikkszám felül becslésére a 6-ot kapjuk, az s -klikk mentes színezésből adódóan, a képlet alapján (1.4). A színezés végeredménye a 3.1f ábrán látható.

```
c 3-free coloring of
c cols/graf_3
c nodes 16
c upperbound of omega: 6
s col 3
l 1 1
l 2 1
l 3 1
```

											1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	
1		•		•	•		•			•			•	•			
2	•		•		•	•	•	•			•				•	•	
3		•			•			•	•			•		•			
4	•					•	•			•	•		•		•	•	
5	•	•	•						•			•		•			
6		•		•				•		•	•	•					•
7	•	•		•				•	•				•		•	•	
8		•	•			•	•										•
9			•		•		•							•	•		
10	•			•		•						•	•				•
11		•		•		•						•	•	•			
12			•		•	•				•	•				•	•	
13	•			•			•			•	•			•			
14	•		•		•				•		•		•		•	•	
15		•		•			•		•			•		•			
16		•		•		•	•	•		•		•		•			

2.1. táblázat. $A(z)$ 2.1 ábrán látható gráf szomszédsági mátrixa

1 4 1
 1 5 2
 1 6 1
 1 7 2
 1 8 2
 1 9 1
 1 10 2
 1 11 2
 1 12 1
 1 13 2
 1 14 3
 1 15 1
 1 16 3

Portable BitMap – pbm

Ez egy olyan képformátum, aminek a legegyszerűbb változata egyesekből és nullákból rajzol képet, az egyes fekete, a nulla pedig fehér pixelt jelent, ezeket egy sima szöveges fájlba kell leírni és pbm formátumba kell elmenteni.

Neve a Portable BitMap-ből származik.

Felépítése:

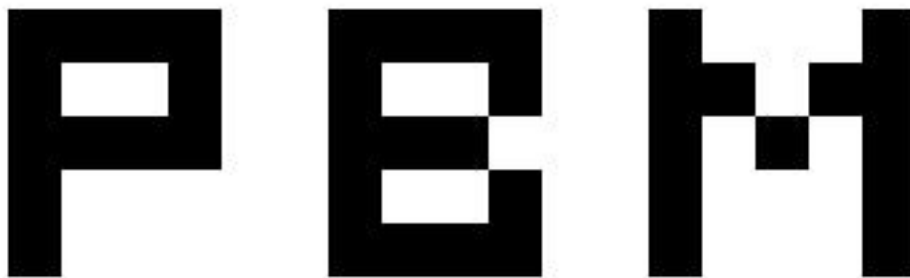
A fájl első sorába a pbm fájl típusát kell feltüntetni, a mi esetünkben ez: P1.

Lehetőség van kommentek elhelyezésére is, ezt a # jel után írhatjuk, az egész sorra kiterjed.

Majd mielőtt nekiállnánk írni az adatokat, meg kell mondani, hogy mekkora képet szeretnénk rajzolni.

```
P1
# feep.pbm
19 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 0 1 0
0 1 0 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1 0
0 1 1 1 1 0 0 1 1 1 0 0 0 1 0 1 0 1 0
0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 1 1 1 1 0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(a) PBM fájl



(b) PBM kép

2.2. ábra. Portable BitMap

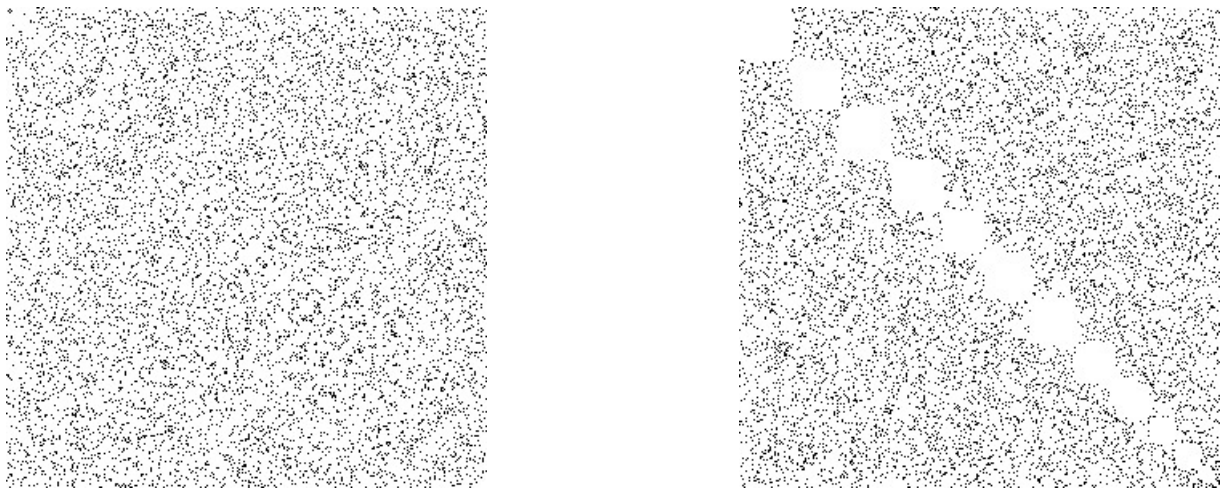
Egy vizualizációs eszköz, melynek segítségével jól látható a színezés eredménye: használatához szükségünk van egy olyan eljárásra, ami átrendezi a gráfot, a színosztályai alapján, ezt az eljárást mi *sorting* néven valósítottuk meg.

Az átrendezett gráf képén a színosztályoknak a főátló mentén elrendezett üres négyzetes blokkoknak felelnek meg. Számunka lényeges információ, hogy a színosztályok viszonylag jól felismerhetők.

Ez a rajz egy nagyon gyors ellenőrzési lehetőséget is biztosít, emiatt javasoljuk a használatát. Ilyen PBM képek a 2.3 ábrán láthatóak.

Mentési pont – chk

Ez egy saját formátum, a 2.4 és a 2.4 formátumok alapján készült el, checkpoint fájlként használatos. Feladata leírni az adott gráfhoz tartozó kvázi színezést, illetve felsorolni azokat az éleket a gráfból, amelyekkel a párhuzamos program számolni fog. A komment-



2.3. ábra. Egy véletlenszerűen generált gráf PBM képei. Bal oldalon az eredeti állapot. Jobb oldalon a színosztályok szerint átrendezett kép.

sorokat itt is a 'c' jelzi, a színezés eredményét az 's col' (ez itt már a kvázi színosztályokat írja le). A különbség az előző két fájlhoz képest, hogy ebben le van írva egy színezés és két élhalmaz, az első élhalmaz az 'edges to delete', azaz azok az élek, melyeket törölni szeretnénk a gráfból, illetve a 'deleted edges', tehát azok az élek amelyeket már sikerült törölni a gráfból. A program ismertetése során még lesz szó erről a checkpoint fájlról (3.3).

```
c 3-free coloring of
c checkpoints/peldagraf_3_4
c nodes 16
s col 1
1 1 1
1 2 1
1 3 1
1 4 1
1 5 1
1 6 1
1 7 1
1 8 1
1 9 1
1 10 1
1 11 1
1 12 1
1 13 1
1 14 1
1 15 1
```

```
l 16 1
c edges_to_delete
e 1 2
e 1 5
e 2 5
e 10 12
e 10 16
e 11 13
e 11 14
e 12 16
e 13 14
e 1 4
e 1 7
e 2 6
e 2 16
e 4 6
e 4 16
e 5 14
e 6 10
e 6 11
e 6 12
e 7 15
e 7 16
e 8 16
e 9 14
e 1 13
e 2 7
e 2 8
e 3 5
e 4 7
e 4 10
e 4 13
e 6 16
c Deleted edges e node1 node2
e 1 2
e 1 5
e 2 5
```

3. fejezet

Az algoritmus implementációja

Előzmények, az algoritmus bemutatása

Szabó Sándor párhuzamos algoritmusának implementációja

Célja a k -klikk keresés párhuzamosítása.

Az algoritmust Szabó Sándor írta le. [Sza11] Alapszintű megvalósítása két szakdolgozatban is megvalósult, ahol szimulált hűtés segítségével minimalizálták az eliminálandó élek számát. Ehhez képest a mi dolgozatunk szisztematizálja és ellenőrzi az eddigi eredményeket, továbbá különböző heurisztikákat vizsgál. Ugyancsak fontos, hogy a mi megoldásunk moduláris, illetve a checkpoint rendszer miatt újraindítható.

Ezt az algoritmust Kiss Annamária és Stráhl István már megvalósították szakdolgozatukban. [Kis13, Str14] Az átlaluk megvalósított algoritmusok a következő módszereket használják

- Kvázi színezés szimulált hűtés módszerrel [Kis13, Str14]
- Súlyozás színezéssel [Str14]
- Checkpoint kezdetleges változata [Str14]

Az algoritmus bemutatása

Programunk a következő kérdésre ad majd választ:

„Van-e k méretű klikk az adott gráfban?”

A fenti heurisztikákat Kiss Annamária és Stráhl István már elkészítette, ezért mi további módszereket, heurisztikákat implementáltunk:

- s -klikk mentes színezés:
segítségével már önmagában is egy jó felső becslést kapunk a gráf maximum klikk-számára.

- Újraosztályozás:
az s -klick mentes színosztályokból k darab kvázi s -klick mentes színosztályt képezzünk. Ezt úgy érjük el, hogy az első k darab legtöbb csúcsot tartalmazó s -klick mentes színosztályba egyesével átrakjuk a csúcsokat a maradék $(r - k)$ darab s -klick mentes színosztályból. Az elosztás során minden egyes csúcsot belepróbálunk minden színosztályba. Végül abba rakjuk bele amelyikben a legkevesebb zavaró él alkotja.
- Élek súlyozása:
Az újraosztályozás során kapott kvázi s -klick mentes színosztályokból feljegyezzük, hogy mely él kitörlésével lehetne ténylegesen s -klick mentes az adott színosztály. Stráhl István által használt módszer kvázi színezés volt. Az élék súlyozását pedig aszerint oldotta meg, hogy a zavaró él által kifizetett gráfokat egyesével kiszínezte. Az így kapott színek száma lett az élék súlya, tehát minél több színnel lehetett kiszínezni egy adott részgráfot annál nagyobb volt az él súlya.
Ezzel szemben mi egy olyan súlyozást használunk, amely megadja, hogy a zavaró él mennyi s -klikket alkot a színosztályban. Ebben az esetben amennyi s -klikket alkot egy él annyi a súlya. Ezeknek az éléknek a kitörlésével a k darab kvázi s -klick mentes színosztályból k darab ténylegesen s -klick mentes színosztály lesz.
- Feladatok sorrendjének módosítása, azok nehézségének megfelelően.
- Továbbfejlesztett checkpoint rendszer.
- Modularitás, ennek köszönhetően a prekondicionálási fázis elkészíthető nem szuperszámítógépen, a szuperszámítógép erőforrásainak okosabb kihasználása végett.

A párhuzamos program feladata ezeknek az éléknek a törlése. Egy él egy feladatot jelent, amit egy szolga fog elvégezni. A mester kiosztja a feladatokat, a szolgák pedig Lilla segítségével ellenőrzik, hogy a kapott feladat(él) alkot-e $k - 2$ klikket a gráfban. Ha nem alkot k méretű klikket akkor törölhetjük a gráfból, ha alkot akkor megkaptuk a választ, hogy van benne k méretű klikk.

Ha találtunk egy k -klikket, akkor esetleg érdekes lehet az is, hogy mely csúcsok tagjai ennek a k méretű klikknek. Annak az élnek az ismeretében amelyikre azt kaptuk, hogy alkot k -klikket, az előző kérdés megválaszolása már egyszerű, hiszen ismerjük 2 csúcsát. Ha befejeződött a program anélkül, hogy talált volna k -klikket, akkor is megkaptuk a választ, nincsen a gráfban k méretű klikk.

A program részletes leírása

A programunkban két klikk keresőt használunk:

- Az s -klick mentes színezésnél a saját s -klick keresőnket, ez kis feladatok esetén egy elég gyors algoritmus, azonban a feladatok növekedésével rohamosan csökken a hatékonysága.
- A párhuzamosítás során pedig Szabó Sándor, Lilla névre keresztelt klick keresőjét használjuk.

A program írásakor 5 algoritmust implementáltunk, az előzőeknek megfelelően:

1. s -klick mentes színező(3.1)
2. Színezés ellenőrző (3.2)
3. Újraosztályozást és az eliminálandó élek feljegyzését végző program (3.3)
4. A párhuzamos program, ami elvégzi az élek kitörlését Lilla segítségével.(3.4)
5. Ellenőrző program (3.5)

3.1. s -klick mentes színező

Ez a program színezi ki a gráfot s -klick mentesre. A 3.1, 1.1 ábrákon a színezés folyamata látható.

Maga az eljárás egy egyszerű, szekvenciális, mohó színező, egy alap backtrack Carraghan-Pardalos klick keresővel. [CP90]

A színezést végző függvény az `s_clique_coloring(...)`, melynek működése:

Egy csúcs akkor tehető bele egy színosztályba, ha nem alkot benne s -klicket. Ebből adódóan a program egy mohó színező, mivel amint talált a csúcsnak egy olyan színosztályt, amelyben nem alkot s -klicket, beleteszi és ugrik a következő csúcsra. Ha nem talált olyan színosztályt amibe bele tudta volna tenni akkor nyit egy új, üres színosztályt neki, majd ugrik a következő csúcsra. A színezés segédfüggvénye a `clq_search()` függvény, ennek segítségével állapítja meg, hogy a színosztályba betenni kívánt csúcs alkot-e s -klicket.

Az említett `clq_search()` függvény feladata, és működése:

s méretű klicket keres a kapott gráfban, egy adott részgráfban, ami esetünkben egy színosztály lesz, hiszen a cél egy s -klick mentes színezés előállítása lesz.

Létrehozunk egy stack-et, aminek az első szintjén elhelyezzük a színosztályt.

Ezt követően megkezdjük a munkát a stack felső sorával (ami jelen esetben az egyetlen): ahol találunk benne csúcsot (i), ott a stack következő szintjére betesszük az előző szint és az (i). csúcs szomszédainak (`graph[i]`) metszetét (ezzel megkapjuk, hogy mely csúcsok szomszédosak egymással). Ezzel tehát növeltük a stack méretét, és a munkát ugyanezzel a módszerrel folytatjuk a stack-en felfelé haladva. Mielőtt azonban feljebb ugranánk a stacken, az (i). csúcsot töröljük a stackből, mert erre a továbbiakban már nem lesz

szükség.

Ha a stack elérte az s szintet, akkor az azt jelenti, hogy van s darab olyan csúcs, amelyek mindegyike szomszédos egymással. Innen tovább nem érdemes menni, találtunk egy s méretű klikket, tehát az s -klikk mentesség nem áll fent. A függvény visszatért `true` értékkel.

Ha a stack legfelső sora csupa nulla, tehát nincs az (i) . csúcsnak további szomszédja a részgráfban akkor eldobjuk ezt a felső sort. Azzal, hogy eldobtuk a felső sorát a stack-nek, visszaléptünk egyet, ahol jöhet a következő csúcs, ha még van a sorban, ha nincs akkor ezt a sort is eldobjuk.

Ha elértünk a függvény végére, az azt jelenti, hogy nincs a színosztályban s -klikk. A függvény ekkor `false` értékkel tér vissza.

Ha a stack első sorába nem egy színosztályt teszünk, hanem egy csupa egyesekből álló bitsetet akkor megmondja, hogy a kapott gráfban van-e s -klikk, azonban ez csak kis s -ekre jó, a gráf méretétől és telítettségétől függően 10-15 a maximum s , amire még belátható időn belül ad eredményt.

Ez egy Carraghan-Pardalos algoritmus változat. [CP90]

```
bool clq_search(vector<dynamic_bitset<> > graph, dynamic_bitset<>
    colorclass, int s)
{
    vector<dynamic_bitset<> > stack;
    int top = 0; // a stack top-ja
    int size = graph.size(); // a gráf mérete
    stack.push_back(colorclass);

    do {
        for(int i = 0; i < size; ++i) {
            if(stack[top][i]) {
                stack[top][i] = 0;
                stack.push_back( stack[top++] & graph[i] );
                break;
            }
        }
        if(top >= s) {
            return true;
        }
        if(stack[top].none()) {
            --top;
            stack.pop_back();
        }
    } while(top >= 0);
}
```

```
    return false;
}
```

3.2. Színezés ellenőrző program

Ez egy ön-ellenőrző program, mely arra szolgál, hogy leellenőrizzük a színezéseinket, hogy valóban s -klikk mentesek-e. A programban két függvényt használunk: az egyik a saját, előzőleg bemutatott klikk kereső, a másik pedig `Lilla`. Azért használunk erre a célra két klikk keresőt is, mert nem kellőképpen megbízható, ha a saját programunk helyességét ugyanazon programmal ellenőrizzük, hiszen ha egyszer rosszul számolt, akkor ellenőrzéskor is rosszul fog.

3.3. Színosztályok újraosztályozását végző program

Ez a program végzi el a k szerinti szétosztást és az eliminálandó élek feljegyzését.

A színosztályok szétosztását a `reclassify()` függvény végzi, melynek feladata és működése:

Először sorba rendezi a színosztályokat a méretük alapján. Ez a heurisztika is a zavaró élek számát hivatott csökkenteni. Ehhez egy buborékrendezeit használ, mert egyszerű és kevés adatra kell alkalmazni, így kellőképpen gyors is.

Majd minden egyes, a kisebb színosztályban levő csúcsra megnézi, hogy melyik nagyobb színosztályban alkot kevesebb zavaró élt. Ezt a `min_edges()` függvény segítségével teszi. Amelyikben a legkevesebb élt alkotja, abba a színosztályba teszi bele.

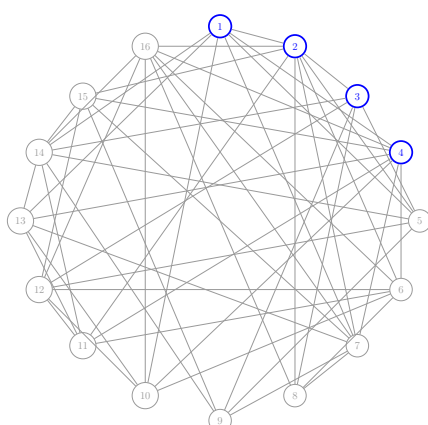
Az eljárásban használt függvények részletezése:

`bubble_sort()` feladata, működése:

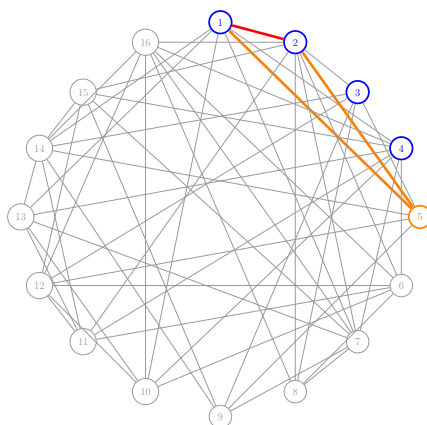
A kapott színosztályokat csökkenő sorba rendezi a bennük levő csúcsok alapján. Segítségével egyszerűen le tudjuk vágni az első q legnagyobb színosztályt. Egyszerű buborékrendezeit, akkor „nagyobb” az egyik színosztály a másiknál, ha több csúcs található benne, ezt pedig a `bitset count()` metódusával állapítja meg. Mivel kis méretű halmazokat kell sorba rendeznünk, így nincs szükség bonyolultabb és hatékonyabb rendezésre.

`min_edges()` feladata, működése:

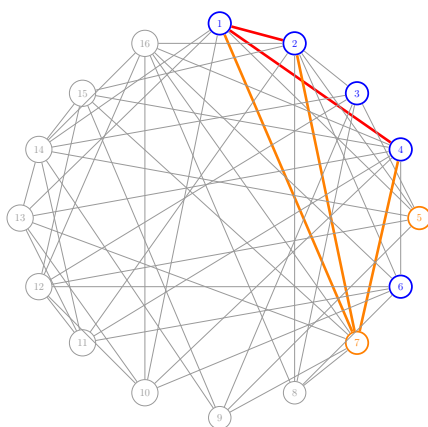
Összeszámolja, hogy a kapott csúcsnak melyik színosztályban mennyi szomszédja van, majd megkeresi ezek közül a legkevesebbet. Segítségével jól el tudjuk osztani a kisebb színosztályok csúcsait a nagyobb színosztályok között. Eredménye annak a színosztálynak az indexe, amelyben a legkevesebb szomszédja van a csúcsnak. Összességében a szétosztás is egy primitív, de gyors algoritmus, hiszen végigmegy a kisebb színosztályok csúcsain és belerakja abba a színosztályba, amelyikben a legkevesebb szomszédja van.



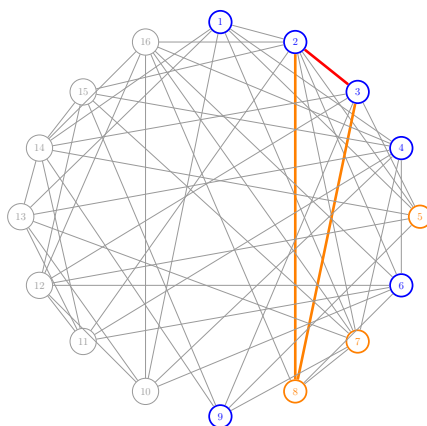
(a) Az első 4 csúcsot beleraktuk az első(kék) színosztályba, mert ezek egyike sem alkot háromszöget.



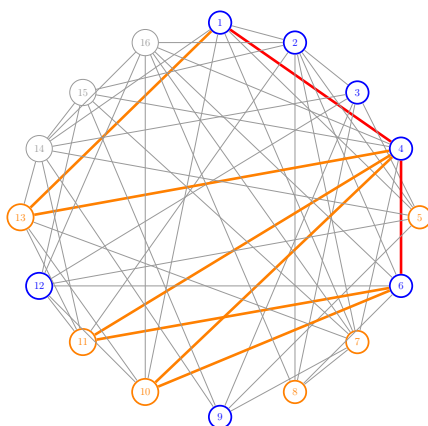
(b) Az 5. csúcs első színosztályba próbálásakor azt tapasztaljuk, hogy itt már alkot három méretű klikket, ezért új színosztályba kerül, a klikk tagjai: 1, 2, 5



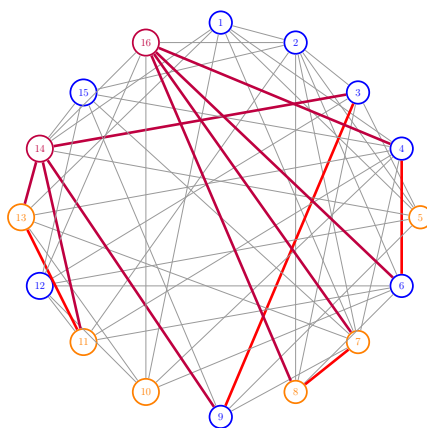
(c) A 6. csúcsot beletudjuk tenni az első színosztályba, mert nem alkot háromszöget, azonban a 7. csúcs már nem kerülhet az első színosztályba, ezért megnézzük, hogy a másodikba beletudjuk-e tenni, oda még belefér, a 7. csúcs által alkotott három méretű klikkek az első színosztályban: 1, 2, 7 és 1, 4, 7



(d) A 8. csúcs az első színosztályban alkot három méretű klikket, ezért a 8-ast a másodikba rakjuk, a 8. csúcs által alkotott klikkek az első színosztályban: 2, 3, 8, a 9. csúcs pedig nem alkot klikket sem az első sem a második színosztályban, belerakjuk az elsőbe



(e) A 10. és a 11. csúcs mindegyike alkot háromszöget az első színosztályban, ezért a másodikba tesszük őket, ezek a klikkek: 4, 6, 10 és 4, 6, 11, a 12. csúcs nem alkot háromszöget az első színosztályban ezért oda tesszük, a 13. csúcs alkot háromszöget az első színosztályban: 1, 4, 13, ezért a másodikba kerül,



(f) A 14. csúcs szintén alkot háromszöget az első színosztályban: 3, 9, 14, és a második színosztályban is: 11, 13, 14, ezért nyitnunk kell egy új színosztályt neki, a 15. csúcs bele kerülhet az első színosztályba, a 16. azonban az első és második színosztályban is alkot háromszöget: 4, 6, 16 és 7, 8, 16, ezért a harmadikba kerül

3.1. ábra. A 2.1 gráf s -klikk mentes színezése, ebben az esetben $s = 3$, tehát háromszögmentes színezésről beszélünk

A program további feladata ezeknek a zavaró éleknek a törlése és feljegyzése, a `delete_edges()` függvénnyel. A függvény működése:

Ez a függvény nem az eredeti gráffal, hanem egy másolatával dolgozik, mert éleket fog törölni belőle, ami megváltoztatja a gráfot.

A `note_edges()` függvény segítségével sorra veszi a színosztályokat és minden színosztályban megszámolja, hogy hány zavaró él van. Ezt az eljárást két lehetséges esetre írtuk meg. Az egyik az az eset, amikor az $s=2$, azaz legális színezésből előállított kvázi színosztályokban kell megkeresnünk a zavaró éleket. Ez annyiból áll, hogy megkeressük az összes, a színosztályban levő élt és feljegyezzük, hiszen az összes színosztályon belül levő él zavaró él. Egy ilyen él megtalálása annyiból áll, hogy megnézzük, van-e két azonos színosztályban levő csúcs között él. A másik eset $s > 2$, azaz s -klikk mentes színezés. Ekkor viszont nem elég megmondanunk, hogy van-e él két azonos színosztályban levő csúcs között, hanem azt is meg kell mondanunk, hogy ez alkot-e s -klikket, ami nehezebbé teszi a feladatot. Ezt követően sorba rendezi ezeket az éleket aszerint, hogy mennyi s -klikket alkotnak. Amennyiben az első helyen több él is áll, tehát több él is alkot ugyanannyi s -klikket, akkor mindet kitöröljük a gráfból, továbbá ha nincs több ugyanolyan súlyú él, akkor az első 10 élt töröljük, gyorsítva ezzel a folyamatot. Ezt a folyamatot annyszor végzi el egy színosztályra, amíg van a színosztályban olyan él, ami s -klikket alkot. Ha elfogytak ezek az élek, akkor veszi a következő színosztályt.

A `note_edges()` függvény feladata, működése:

Feljegyzi a színosztályban található zavaró éleket és megszámolja, hogy melyik él mennyi s -klikket alkot, ezt a `clq_search_count()` függvénnyel teszi. Ha a zavaró él egyetlen s -klikket sem tartalmaz, akkor törli azt az élt a gráf másolatából, de nem jegyzi fel, hiszen a feladat szempontjából ez lényegtelen él, nem befolyásolja a program végkimenetelét. A számolást azonban gyorsíthatja oly módon, hogy a következő s -klikk vizsgálatoknál már nem kell törődni ezzel az éllel, tehát egy-egy csúcsnak kevesebb szomszédját kell vizsgálni. Erre a feladatra több megoldás is született a program írása során:

- Az első elképzelés az volt, hogy nem foglalkozunk azzal, hogy melyik zavaró él mennyi s -klikket alkot, hanem csak megnézzük, hogy alkot-e s -klikket. Ha alkot, akkor kitöröljük, és megyünk tovább, ezt egészen addig ismételve, amíg van a színosztályban s -klikk.

Ezt az ötletet optimalitási problémák miatt vetettük el, mert előfordulhat, hogy elegendő lenne 20 él helyett 4-et kitörölni, és ugyanoda jutnánk, mintha 20-at töröltünk volna, tehát ez feleslegesen sok munkát csinált volna a párhuzamos programnak.

- A második elképzelés, hogy melyik él mennyi s -klikket alkot, de ekkor csak azt az élt töröltük ki a gráfból, amelyik a legtöbbet alkotta, nem törődve azzal, hogy esetleg több él is állhat az első helyen. Egy él törlése után megkerestük megint az

összes s -klikket alkotó élt, ezek közül megint töröltük a legtöbb s -klikket alkotót és így tovább. Ez már közel optimális megoldás, mert csak azokat az éleket törli, amelyeket mindenképpen szükséges. Arról azonban nem szabad megfeledkeznünk, hogy a programot úgy írtuk meg, hogy a prekondicionálás megoldható legyen egy kis teljesítményű számítógépen is. Ez a művelet pedig rendkívül erőforrás igényes, már akkor is, ha csak arra gondolunk, hogy egy s -klikket megtalálni mennyire nehéz, egy él pedig több ezer s -klikket is alkothat, tehát értelmetlenül sokáig futna egy hétköznapi számítógépen.

- Ezért jutottunk arra a megoldásra, hogy felgyorsítjuk az algoritmust, ezáltal romlott az optimalitása, de még mindig sokkal jobb, mint az első megoldásnak, ráadásul sokkal gyorsabb a másodiknál. Ez pedig az előző megoldásból született, miszerint, ha több ugyanannyi s -klikket alkotó él van az első helyen, akkor mindet kitöröljük, ha nem lenne több ilyen él akkor a megtalált élek közül az első tízet töröljük.

Az utóbbi megoldást egy feltételezés miatt két különböző checkpoint fájlba írjuk. A feltételezés az, hogy minél több s -klikkbe tartozik bele egy él az adott színosztályon belül, annál nehezebb kitörölni a gráfból, annál több erőforrást igényel a vizsgálata, hogy kitöröljük. Ellenben azokkal az élekkel, amelyek kevesebb s -klikkbe tartoznak bele, azt feltételezhetjük róluk, hogy ezek könnyebben kitörölhetők, kevesebb számolást igényelnek. Itt még csak a színosztályon belüli s -klikkeket vesszük figyelembe, de a párhuzamos program már a teljes gráffal dolgozik, tehát ha egy él sok s -klikkbe tartozik bele egyetlen színosztályon belül, akkor jó esély van rá, hogy a teljes gráfban is sok nagy méretű klikket alkot, ami arra utalhat, hogy ez egy nehéz feladat. Ezért ezeket az eliminálandó éleket egyszer abban a formában írjuk ki, ahogy megtaláltuk őket és feljegyeztük, majd növekvő sorba rendezzük őket a súlyaik(melyik mennyi s -klikket alkot) alapján. Ettől azt várjuk, hogy a könnyen törölhető élek kerülnek előre, amiket a párhuzamos program gyorsabban ki tud törölni, és esetlegesen ezeknek az éleknek a törlésével könnyíthetünk egy nehezebb feladaton is.

A függvény eredménye egy élhalmaz, amely élstruktúrákat tartalmaz:

```
struct edge{
    int node_a; //Az él egyik végpontja
    int node_b; //Az él másik végpontja
    int c; //Mennyi s-klikket alkot
}
```

A `clq_search_count()` függvény szinte teljesen ugyanaz, mint a `clq_search()` függvény, csupán annyi különbséggel, hogy amikor talált s -klikket, akkor nem tér vissza a hívó függvényhez, hanem visszalép egyet a stack-en és folytatja tovább a keresést, közben

viszont nőtt egyel az élhez tartozó s -klikkek száma.

Legyen v_1 és v_2 egy él két végpontja. Legyen H egy halmaz, amely v_1 és v_2 közös szomszédait tartalmazza. Képezzük H egy olyan részhalmazát, ami kizárólag azokat a közös szomszédokat tartalmazza, amelyek a színosztályban is megtalálhatóak. Ez a függvény ezen részhalmazra hívódik meg.

A függvény eredménye egy szám, hogy mennyi s -klikket alkot az él a színosztályban.

A `delete_edges()` függvény eredménye egy olyan élhalmaz, amelyet ha sikerül kitörölni a gráfból, akkor az előzőleg előállított kvázi s -klikk mentes színezésből s -klikk mentes színezés válik.

Mentési pont – Checkpoint

Az imént említett élhalmazt a már többször is említett checkpoint fájlba írjuk ki. Ennek segítségével válik a programunk még rugalmasabbá, hordozhatóbbá. Ha találunk egy jobb módszert az eliminálandó élek feljegyzésére, akkor használhatjuk azt is a párhuzamos programhoz, csak ennek a checkpoint fájl formátumnak kell megfelelnie: 2.4.

3.4. Párhuzamos program

Ezt a programot a 2.3 fejezetben bemutatott párhuzamosítási módszerrel írtuk meg, ez fog a szuperszámítógépen futni. Feladata az előző program által felsorolt zavaró, eliminálandó élek megvizsgálása, törlése a gráfból. Ezeket a feladatokat a checkpoint fájlból olvassa be. Annyi feladata lesz, amennyi élt felsoroltunk a checkpoint fájlban.

A program elején minden folyamat beolvassa a teljes gráfot, továbbá a megoldandó és a már megoldott feladatokat is, ezzel elkerülve a feladatmegosztás nehézségeit, azonban ezzel a döntéssel egy olyan korlátot állítottunk fel a programunknak, amely megakadályozza, hogy egy olyan rendszeren futtassuk, amely nem rendelkezik osztott tárhellyel. A mi esetünkben ez nem jelent gondot, mert a szuperszámítógépen futtattuk a programot, amely képes megosztani a háttértárat a folyamatok között.

Működése

3.4.1. A mester menetrendje

Kiveszi a feladatok közül a már megoldott feladatokat, majd kéréseket vár a szolgáltól. Az `MPI_Recv`-ben a `source` paraméter helyére az `mpi_any_source` jelzést illesztettük, így bármelyik szolgáltól kaphat kéréseket, ellenkező esetben a szolgáknak várakozniuk kellene addig, amíg az ő sorszámuk sorra nem kerül. Ez a kérés a szolga azonosítóját tartalmazza, innen tudja a mester, hogy kinek kell visszaküldeni a feladatot. Ha az azonosító

negatív szám, akkor tudja a mester, hogy az a szolga, amelyik a kérést küldte, talált egy $k - 2$ klikket. Ekkor leállítja a többi szolgát is, a kérdés pedig, miszerint van-e k -klikk a gráfban, meg lett válaszolva: van.

Ha a mester már nem tud több feladatot kiosztani, mert azok elfogytak, akkor le kell állítania minden szolgát. Ezt úgy teszi meg, hogy még egy utolsó körben mindenkitől fogad egy kérést, amit ilyenkor is megvizsgál, mert az utolsó vizsgált él is lehet egy k klikk tagja. Ha a kérés helyes volt, akkor már nem az elvégzendő feladat sorszámát küldi, hanem a leállító feltételt, ez pedig a -1.

3.4.2. Egy szolga menetrendje

Küld egy kérést a mesternek az azonosítójával, erre válaszul a mestertől kap egy indexet. Ez az index a megoldandó feladat sorszáma lesz. Ha az index egy negatív szám, akkor az azt jelenti, hogy elfogytak a feladatok, és leállhatnak a szolgák. A szolga egy saját lokális gráffal dolgozik, amelyből töröl minden olyan élt, amit a mester már kiosztott egy másik szolgának. Ezek a szolga aktuális feladatában érdektelenek, mert nem azt az élt kell vizsgálnia. Ha ezek közül az élek közül valamelyik alkot $k - 2$ klikket, tehát van k -klikk a gráfban, akkor arról majd az a szolga fogja értesíteni a mestert, amelyik éppen azon az élen dolgozik.

Legyenek v_1 és v_2 egy G gráf élének két végpontja. A G gráf b_1 és b_2 közös szomszédai által feszített részgráfja legyen G' .

A szolga erre a G' feszített gráfra hívja meg Lilla-t, $k - 2$ klikket keresve. Ha a program azt mondja, hogy nincs $k - 2$ klikk a feszített gráfban, akkor a szolga leírja ezt az élt a checkpoint fájlba, a már megvizsgált és törölt élek közé, majd egy újabb kérést küld a mesternek, és kezdi előről a folyamatot, az új élre. Azonban ha a klikk kereső talált egy $k - 2$ klikket, akkor a szolga kiírja, hogy melyik feladatnál találta a $k - 2$ klikket, negálja az azonosítóját és így küld kérést a mesternek. A mester ebből tudja, hogy van k klikk a gráfban és leállítja az összes szolgát.

A `stretch_graph()` működése:

A bitsetek használatával lényegesen le tudjuk egyszerűsíteni a műveletet.

Bemenő paramétere egy bitset, amelyben azt adjuk meg, hogy mely csúcs(ok) által megfeszített gráfra vagyunk kíváncsiak. A függvény végighalad a gráf csúcsain és amelyek csúcs szerepel a kapott bitsetben, annak a csúcsnak a szomszédait metszi a kapott bitsettel (a sorban a csúcsnak azon szomszédai maradnak, amelyek szerepelnek a bitsetben is). Amelyik csúcs nem szerepel benne, annak a sorát (szomszédait) pedig törli. Tehát azok az élek maradnak a gráfban, amelyeknek a végpontjai szerepelnek a kapott bitsetben. A függvény eredménye a feszített gráf.

MPI üzenetküldések összefoglalása

Mester	Szolgák
Feladat kérést vár.	Feladatot kér.
Megvizsgálja a kérést, ha helyes volt, újabb feladatot küld. Ha a kérés leállító üzenet, akkor nem küld több feladatot egyik szolgának sem, hanem <code>MPI_Abort()</code> -al leállítja a programot.	Ha a kapott válasz helyes volt az azt jelenti, hogy van még feladat, elvégzi, majd új feladatot kér.
Ha elfogytak a feladatok, akkor minden szolgáltól még fogad egy kérést, de a válasz már a leállító üzenet lesz.	Fogadja a leállító üzenetet.

3.5. Ellenőrző program

Ez a program csak abban az esetben használatos, ha a párhuzamos program talált k -klikket a gráfban, ekkor megmondta azt is, hogy melyik él tagja ennek a k -klikknek. Ezzel a programmal le tudjuk ellenőrizni, hogy az él valóban tagja-e egy k -klikknek, itt szintén Lilla végzi el a munkát.

Eltérések

A megoldásunk tehát abban különbözik a többi hasonló kérdéssel foglalkozó párhuzamos programtól, hogy azok a programok tisztán párhuzamos programok, kizárólag párhuzamos környezetben futtathatók, jelen program viszont több részre osztható. A prekondícionálások soros programok, nincs szükségük nagy erőforrásra, futtathatók akár egy kis teljesítményű gépen is. Ezeknek az eredményeit a párhuzamos programunk fogja használni, amit már egy másik gépen, akár szuperszámítógépen is futtathatunk. Ahhoz, hogy mindezt meg tudjuk oldani, szükségünk volt ellenőrző fájlokra, ezek a `.col` és `.chk` fájlok voltak.

A több részre osztott, moduláris program további előnye, hogy a részek külön-külön tovább fejleszthetők, optimalizálhatók, csupán az adott `.col` és `.chk` fájlok formátumainak kell megfelelniük.

3.6. A program futtatása

A programot a 2.1 fejezetben leírt pécsi szupergépen futtattuk. A program futásának során szisztematikus méréseket végeztünk. A mérések és azok kiértékelése a következő fejezetben található. A programokat terminál segítségével lehet futtatni. Az első három program a prekondicionáláshoz tartozik, ezeket lehet használni személyi számítógépen is, ahogy az 5.-et, az ellenőrző programot is. A 4. program pedig a szuperszámítógépen használatos.

A program futtatásának lépései:

1. A színező program.

A program bemeneti paraméterei: a gráf, és az s . Legális színezéshez $s=2$ -t kell megadnunk. A program elkészíti a megadott gráf egy s -klikk mentes színezését. A színezést a `cols/` almappába menti, megegyező néven a gráffal, `.col` kiterjesztéssel.

2. A színezés ellenőrző program.

Bemeneti paraméterek: a gráf, az előző program által elkészített `col` fájl, és az s . A program megmondja, hogy az adott színezés helyes-e.

3. Újraosztályozást és a zavaró élek feljegyzését végző program.

Bemeneti paraméterek: a gráf, a `col` fájl, az s és a k . Ahol a k a keresett klikk mérete. A program eredménye egy élhalmaz, amit a `checkpoints/` mappába ment, ezek a feljegyzett eliminálandó élek. A checkpoint fájl felépítése a 2.4 fejezetben található.

4. Párhuzamos program.

Bemeneti paraméterei: a gráf, a checkpoint fájl és a k . Ahol a k a keresett klikk mérete. A program a megadott checkpoint fájlba folyamatosan írja a megvizsgált éleket. Megszakítás esetén ezzel később onnan folytatható a program ahol abba-maradt. Ha talált k méretű klikket akkor kiírja a konzolra a k -klikket alkotó él két végpontját.

5. Ellenőrző program.

Bemeneti paraméterei: a gráf, a checkpoint fájl és a megvizsgálni kívánt él két végpontja.

4. fejezet

Mérési eredmények

A párhuzamos programot és a prekondicionálásokat is egyaránt a(z) 2.1 fejezetben bemutatott szuperszámítógépen futtattuk, ezen készítettük el a méréseket. A fejezetben bemutatjuk azokat a gráfokat amelyeken a programot teszteltük, ezek a(z) 4.1 táblázatban találhatók.

A táblázatban leírjuk a gráfok nevét, a csúcsok és az élek számát, a sűrűségüket, illetve, hogy mekkora a legnagyobb klikk ezekben a gráfokban.

Több fajta mérést is végeztünk a tesztelés során. Az első az általános használat eredményeit tükrözi, más-más színezéseket használtunk. A másik pedig szisztematikus mérések voltak. Ekkor ugyanazt a gráfot több színezéssel is lefuttattuk, azért, hogy kiderüljön, mennyire befolyásolja a színezés a programunk futását, és a feladatok nehézségét.

A postafiók-elv megvalósításának eredménye, hogy a processzorok közül egy mindig a mester folyamatot fogja jelenteni, tehát 300 processzoron futtatva a programot 299 szolga fog dolgozni a feladatokon, a mester pedig ezeket a feladatokat osztja ki. A mérések során mindenhol a mester folyamat futásidejét vettük figyelembe, mert ez az egyetlen olyan folyamat, amely a program indulásától a program leállásáig végig jelen van. **A program futásideje minden esetben másodpercben van megadva.**

A program tesztelésekor használt gráfok

- DIMACS weboldaláról azok a gráfok amik megoldhatónak bizonyultak, de nem túl könnyűek, hogy megérje őket párhuzamosítani [DIM]
- **rand** előtaggal kezdődő gráfok (véletlenszerűen generált gráfok)
Ezeknek a gráfoknak a neve tartalmazza a gráf fő adatait. Egyrészt, hogy mennyi csúcsa van, másrészt, hogy mekkora az élek sűrűsége. Például a **rand-n300-p06** gráfnál az n300 azt jelenti, hogy 300 csúcsa van, a p06 pedig azt, hogy 0.6-os a gráf telítettsége.
- **monoton** [Wei]
- **matr-del** [Slo]

4.1. Első mérési eredmények

A 4.1 táblázatban található gráfokon végzett prekondicionálások eredményei a 4.2 táblázatban találhatóak.

Ebben a táblázatban leírjuk, hogy az *s*-klick mentes színezés során mekkora *s*-t használtunk, mennyi ideig tartott az *s*-klick mentes színezés, és mennyi ideig tartott a színosztályok újraosztályozása.

A program által mért eredmények pedig a 4.3, illetve a 4.4 táblázatokban találhatóak. Ezeknek a táblázatoknak az értelmezése:

- Gráf - a gráf neve, amellyel teszteltük a programot.
- Feladatok száma - az újraosztályozás során feljegyzett eliminálandó, zavaró élek száma, ennyi feladata volt a párhuzamos programnak.
- A számok pedig a processzorok számát jelentik:
 - 1 - A soros klick kereső program (Lilla), 1 processzoron futtatva mért eredménye.
 - 18, 60, 120, 300 - a párhuzamos program sorra 18, 60, 120 és 300 processzoron futtatva mért eredménye.
 - rendezett - Az újraosztályozás során megemlített elmélet szerinti rendezett checkpoint fájl (3.3 segítségével mért idő, a felette megadott processzor-számra.
 - rendezetlen - Hasonlóan a rendezetthez, ez a rendezetlen checkpoint fájl segítségével mért idő.

Ahova nem írtunk értéket, ott nem futtattuk a programot annyi processzorra, mivel a feladatok száma alapján nem érte volna meg. Ha túl kevés a feladat, akkor sok processzor teljesítménye kihasználatlan marad, a program pedig nem lesz tőle gyorsabb. Ha túl sok a feladat, és a gráf is a nagyobbak közül való, akkor pedig kevés processzoron sokáig tartott volna a feladatok megoldása.

4.1.1. Prekondicionálások

A prekondicionálás folyamata három részre osztható:

- A gráfok *s*-klick mentes színezése (1.4, 3.1)
- Ezeknek a színosztályoknak az újraosztályozása (3.3)
- Az újraosztályozás után az eliminálandó élek feljegyzése.

Gráf neve	Csúcsok száma	Élek száma	Sűrűség	Maximum klikk
gen200_p0.9_44	200	17910	0.9	44
rand-n300-p06	300	26885	0.6	15
p_hat300-1	300	10933	0.25	8
p_hat300-2	300	21928	0.5	25
p_hat300-3	300	33390	0.75	36
MANN_a27	378	70551	0.99	126
DSJC500_5	500	125248	0.5	13
rand-n500-p06	500	129834	0.6	16
p_hat700-1	700	60999	0.25	11
p_hat700-2	700	121728	0.5	44
keller5	776	225990	0.37	27
rand-n800-p05	800	319342	0.5	30
p_hat1500-1	1500	284923	0.25	12
matr-del-7	128	6657	0.82	16
matr-del-8	256	28801	0.88	30
matr-del-9	512	121089	0.92	52
monoton-7	343	46305	0.79	19
monoton-8	512	106624	0.8	23
monoton-9	729	221616	0.84	28
p_hat1000-1	1000	122253	0.25	10
p_hat1000-2	1000	244799	0.5	46
p_hat1000-3	1000	371746	0.75	68

4.1. táblázat. A program tesztelésekor használt gráfok

Gráf	s	s -klick mentes színezés	színosztályok újraosztályozása
gen200_p0.9_44	5	0.15	0.99
matr-del-8	4	0.21	1.27
rand-n300-p0.6	6	0.47	1657.77
p_hat300-1	5	0.18	23.05
p_hat300-2	6	0.3	56.05
p_hat300-3	4	0.42	1.67
MANN_a27	5	1.1	0.37
DSJC500_5	3	1.43	19
rand-n500-p0.6	3	1.2	25.14
monoton-8	5	1.4	1047.87
p_hat700-1	3	1.47	22.97
p_hat700-2	5	1.9	187.45
keller5	4	2.78	447.35
rand-n800-p0.5	4	3.66	127.33
p_hat1000-1	3	4.29	209.6
p_hat1500-1	3	14.6	955.44

4.2. táblázat. Az előkészületek ideje

4.1.2. A program futásideje

Gráf	Feladatok száma	1	18		60		120		300	
			rendezett	rendezetlen	rendezett	rendezetlen	rendezett	rendezetlen	rendezett	rendezetlen
gen200_p0.9_44	340	2.91	1.14574	2.62062	0.89	1.632	0.91	1.462	0.88	1.5423
rand-n300-p0.6	1703	6.67	0.96548	2.21521	0.9328	1.83214	0.91335	0.92431	0.8314	0.9145
p_hat300-1	520	4.71	2.49382	0.83599	2.01	0.821	0.87	0.823	0.79	0.81
p_hat300-2	517	5.69	2.12571	1.04831	2.1123	0.924	1.872	0.94312	2.1257	1.0483
p_hat300-3	803	41.26	6.93518	17.6423	2.012	5.23	1.13	1.22	0.92	1.0311
MANN_a27	112	4.081	1.28368	1.27967	1.3123	1.32613	1.31659	1.30286	-	-
DSJC500_5	3847	31.4	4.36036	3.91693	3.9886	3.91693	3.84924	3.8231	3.8029	3.8239
rand-n500-p0.6	4098	46.43	4.26496	4.2356	4.2514	4.3465	4.8764	4.4536	4.234	4.1245
p_hat700-1	3548	59.54	286.865	234.075	21.182	19.2314	10.185	10.192	10.188	10.22
p_hat700-2	1889	395.45	19.613	58.2481	19.234	42.2145	10.6321	13.2015	9.9541	9.959
keller5	2922	96.18	19.9596	19.416	19.121	19.521	18.912	19.4131	18.923	19.412
rand-n800-p0.5	4275	130.35	48.214	44.3258	31.375	16.2956	14.965	14.9875	14.932	14.915
p_hat1500-1	23630	-	-	-	2430.6	2234.6	1202.1	1123.4	101.41	101.42

4.3. táblázat. A program futásideje maximum klikk kereséskor.

A maximum klikk keresés eredményeinek értékekése

A 4.3 táblázatból kiolvasható, hogy a párhuzamosított Lilla szinte minden esetben előbb megtalálta a maximum klikket. Ez annak köszönhető, hogy nagyságrendekkel kisebb feladatokkal foglalkozik, mint a soros Lilla, így előbb végez egy-egy él megvizsgálásával. Ráadásul a párhuzamos program egyszerre több él által megfeszített gráfot képes megvizsgálni. A tesztek során, a program indulásakor a processzoroknak megfelelően nagyjából egy időben 17, 59, 119, 299 (ezek a szolga folyamatok) élt kezdett el megvizsgálni.

Ha ezek között a megvizsgált élek között volt akár egyetlen olyan él, ami $k - 2$ klikket alkotott, akkor első körben megtaláltuk a maximum klikket. A feladatok kiosztása között természetesen eltelik némi idő, de ez elhanyagolható az egész feladat megoldásához képest. Azonban a `p_hat_700_1` nevű gráfnál látszik, hogy kevés processzoron futtatva sokkal több időbe telt megtalálnia a maximum klikket, mint a soros programnak. Ugyanis előfordulhat az is, hogy az említett gráfnál a 3548 feljegyzett eliminálandó, zavaró él közül az első akár 1000 él egyike sem tartozik bele a keresett k -klikkbe, ettől függetlenül ezt az 1000 élt is meg kell vizsgálni.

A processzorok számának növelésével megfigyelhető, hogy a `p_hat_700_1` gráfnál is a soros program futásideje alá csökkent a párhuzamos program futásideje.

A rendezett checkpoint fájlok a táblázat eredményei alapján nem hozták meg a várt eredményt. Sőt, volt, ahol hosszú másodpercekkel a rendezetlen checkpoint fájl után találták meg a maximum klikket.

Azonban ilyen esetekben ahol ismerjük a maximum klikkjét egy gráfnak, a rendezett checkpoint fájl, még ha a feltételezésünk helyes is, nem válna hasznunkra, hiszen elsőként olyan éleket vizsgálnánk meg amelyek potenciálisan nem tartoznak k -klikkbe. Ezáltal azoknak az éleknek a vizsgálásával menne az idő, amelyek nem fontosak a feladat szempontjából.

Gráf	Feladatok száma	1	18		60		120		300	
			rendezett	rendezetlen	rendezett	rendezetlen	rendezett	rendezetlen	rendezett	rendezetlen
gen200_p0.9_44	340	2.92	7.62643	7.70273	2.3017	2.2844	1.87	1.932	0.845603	0.810191
rand-n300-p06	1703	9.081	94.9629	95.3462	27.6459	28.0111				
p_hat300-1	520	4.75	25.821	25.7685	7.49997	7.53382	4.18936	4.18868	1.68442	1.69008
p_hat300-2	517	5.091	29.3592	29.6207	8.84942	8.82647	4.8778	4.88068	2.1607	2.1387
p_hat300-3	803	541.51	62.2183	63.7853	18.5292	18.91	9.83811	14.2638	6.30452	14.1888
MANN_a27	112	23.1	9.95371	9.93347	3.27402	3.26585	2.62953	3.03456	-	-
DSJC500_5	3847	37.66	878.2	882.07	257.296	257.415	129.221	130.322	51.15	51.114
rand-n500-p06	4098	144.27	1048.82	1069.5	1169.39	1173.25				
p_hat700-1	3548	59.51	2133.73	2133.85	621.051	621.07	306.387	306.151	122.589	122.573
p_hat700-2	1889	2064.84	1385.84	1411.43	404.845	411.006	203.784	208.47	86.9041	127.521
keller5	2922	8685.19	5685.78	5521.71	1683.5	1600.07	815.012	814.699	382.875	356.299
rand-n800-p05	4275	253.72	4047.76	4046.92	1170.76	1168.79				
p_hat1500-1	23630	584.9	-	-	41805.1	41905.3	20844.9	20767.1	8301.67	8299.44

4.4. táblázat. A program futásideje, a maximum klikknél egyel nagyobb klikk keresésekor.

A maximum klikk + 1 keresés eredményeinek értékelése

A maximum klikknél egyel nagyobb klikk keresésekor már teljesen más eredményeket kapunk, mint a maximum klikk keresésekor. Ez azért van, mert maximum klikk keresésekor a soros és a párhuzamos program is leáll, amint talált egy k méretű klikket. Ebben az esetben amikor már a maximumnál egyel nagyobb klikket keresünk meg kell vizsgálni a teljes gráfot, illetve a párhuzamos programnak meg kell vizsgálnia minden feljegyzett élt. Ez pedig általában tovább tart, mint megtalálni az első legnagyobb klikket a gráfban. Ebben a táblázatban már nem minden esetben tapasztalhatók a soros programhoz képest a gyorsulás jelei.

Amikor a soros program gyorsabb

Az olyan gráfokkal, amelyeknek kicsi a sűrűsége a soros program könnyebben megbirkózik, mint a párhuzamos. Ez annak köszönhető, hogy a kisebb sűrűségű gráfok mind a soros, mind a párhuzamos program számára könnyű feladatot jelentenek. Amíg a soros programnak csak egy „könnyű” feladattal kell dolgoznia, addig a párhuzamos program akár több tízezer valóban könnyű feladat megoldására hivatott.

A `p_hat1500-1` nevű gráf sűrűsége elég kicsi, tehát ebből a szempontból egy könnyű feladatnak is felfogható. A soros program végzett is a maximum klikknél egyel nagyobb klikk keresésével alig 10 perc alatt. A párhuzamos program viszont 18 processzoron futtatva 15 óra alatt sem volt képes elvégezni a 23630 „könnyű” feladatot. 60 processzoron is több, mint 11 órába telt neki mire végzett minden feladattal. A gyorsulás azonban rendkívüli, a 60 processzoron mért 11 óra, 120 processzor esetében már fele annyi volt, nagyjából 5,5 óra. További 180 processzor bevonásával, összesen 300 processzoron futtatva már „alig” 2,5 óra volt, de még ez is rengeteg, a soros program 10 percéhez képest. Még több processzor segítségével ez az idő egészen 153 másodpercre, azaz körülbelül 2,5 - 3 percre lenne csökkenthető, ugyanis az említett gráfban ez volt a legnehezebb feladat megoldásának ideje. (Ebben a gráfban a feladatok átlagos megoldási ideje 100 másodperc körül van.)

Ahhoz, hogy ezt a gyorsulást elérjük, több ezer processzorra lenne szükségünk, legalább annyira, amennyi feladat van. Sőt, ha csak azt szeretnénk elérni, hogy a soros programnál gyorsabb legyen, akkor is legalább tízezer processzorra lenne szükség, ekkor 6 perc körül mozogna a párhuzamos program futásideje.

Ugyanez a helyzet a `p_hat700-1` gráfnál is. A soros program mindössze 1 perc alatt megvizsgálta a teljes gráfot. A párhuzamos programnak 18 processzoron futtatva 35 perc, 60 processzoron futtatva kicsivel több, mint 10 perc, 120 processzoron nagyjából 5 perc, és még 300 processzoron is több, mint 2 perc.

Hasonlóan az előző gráfhoz, ennél a gráfnál is jobb eredményt érhetünk el további processzorok bevonásával. Akár 13 másodperc alatt (ez a legnehezebb feladat megoldásának ideje, az átlag pedig 10 másodperc körül mozog) képesek lennénk megoldani az összes

feladatot, megfelelő mennyiségű processzor segítségével.

Azonban ez esetben is kérdéses, hogy megéri-e annyi processzor munkába állítása, minimális gyorsulás érdekében.

Tehát ennél a gráfnál sem sikerült jobban teljesítenie a párhuzamos programnak a sorosnál.

Ebből azt a következtetést lehet levonni, hogy a kisebb sűrűségű gráfokat nem érdemes a párhuzamos programmal használni, mert valószínűleg a soros program előbb végezni fog a feladattal, és kevesebb erőforrásigénye is van a párhuzamos programhoz képest.

Nagy sűrűségű gráfok, nehezebb feladatok

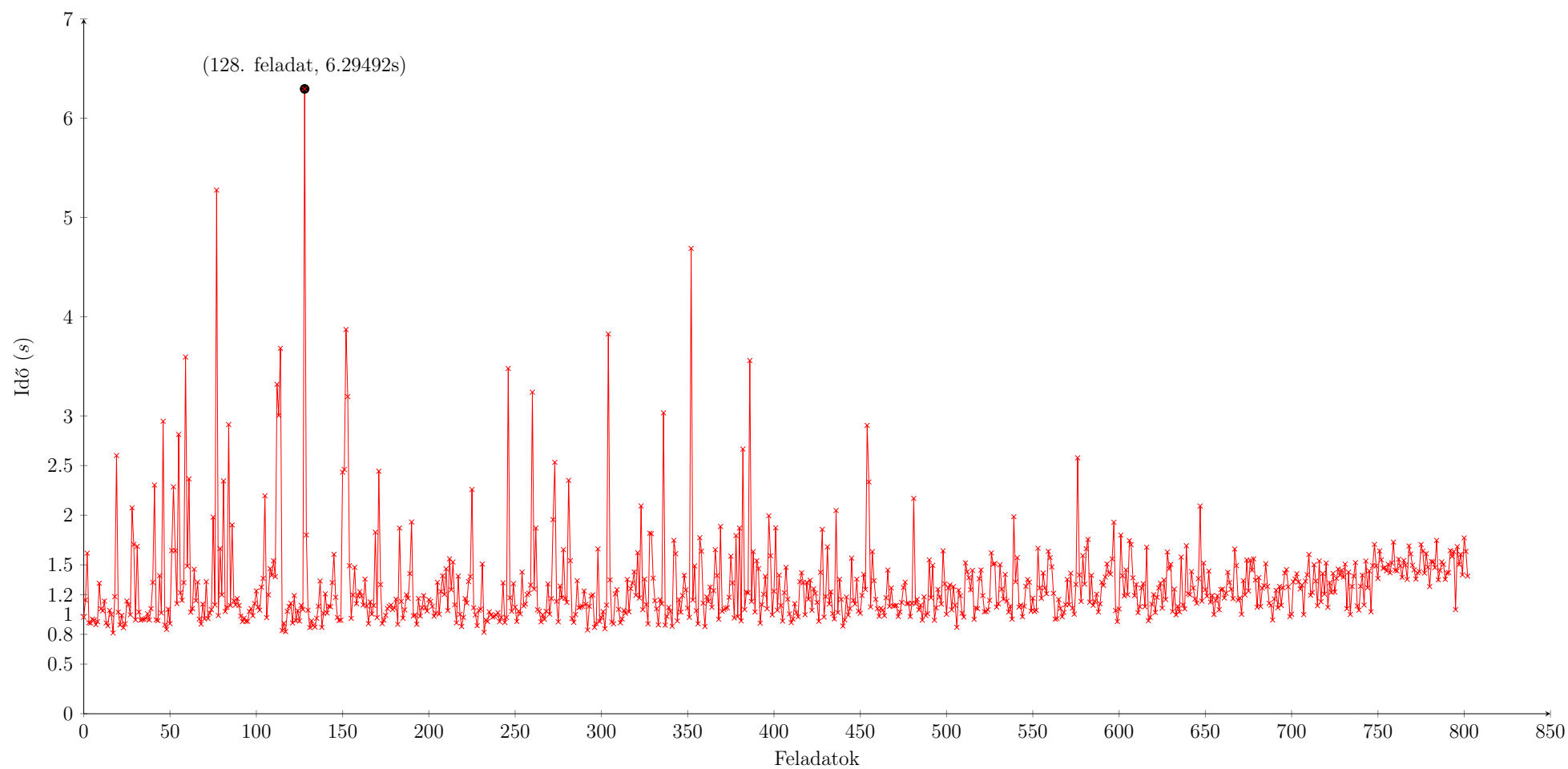
Megfigyelhető, hogy a nagyobb sűrűségű gráfoknál már kevés processzor is határozottan nagy gyorsulást hozott a soros programhoz képest. Ilyenek például a `p_hat_300-3`, `MANN_a27`, `monoton-8`, `p_hat700-2`, `keller5`.

Ahogy az a 4.1 ábrán is látszik, a `p_hat300-3` gráf feladatai átlagosan 1 - 2 másodperc alatt megoldhatóak, tehát hasonló nehézségűek, természetesen vannak kiugró feladatok, melyek megoldása tovább tart. Ezek a feladatok nehezebbek, mint a többi. Ezeket nevezzük domináns feladatoknak (éleknek), ilyen a 77. és 128. feladat a 4.1 ábrán.

Amint az előző részből kiderült, a kisebb sűrűségű gráfokat nem érdemes a párhuzamos programmal tesztelni. A nagy sűrűségű gráfoknál viszont meglehetősen ajánlott a párhuzamos program használata, hiszen már kevés processzorral is óriási gyorsulást lehet elérni a soros programhoz képest.

A `p_hat_300-3` nevű gráfnak nagyobb a sűrűsége, ezért a soros `Lilla` számára ez viszonylag egy nehéz feladat, azonban a párhuzamos program itt is kisebb, ezáltal „átláthatóbb” feladatokkal foglalkozik, amelyeket könnyebb megoldani is. Ennél a gráfnál nincs is túl sok megoldandó feladat (803), így gyorsabban is tud haladni.

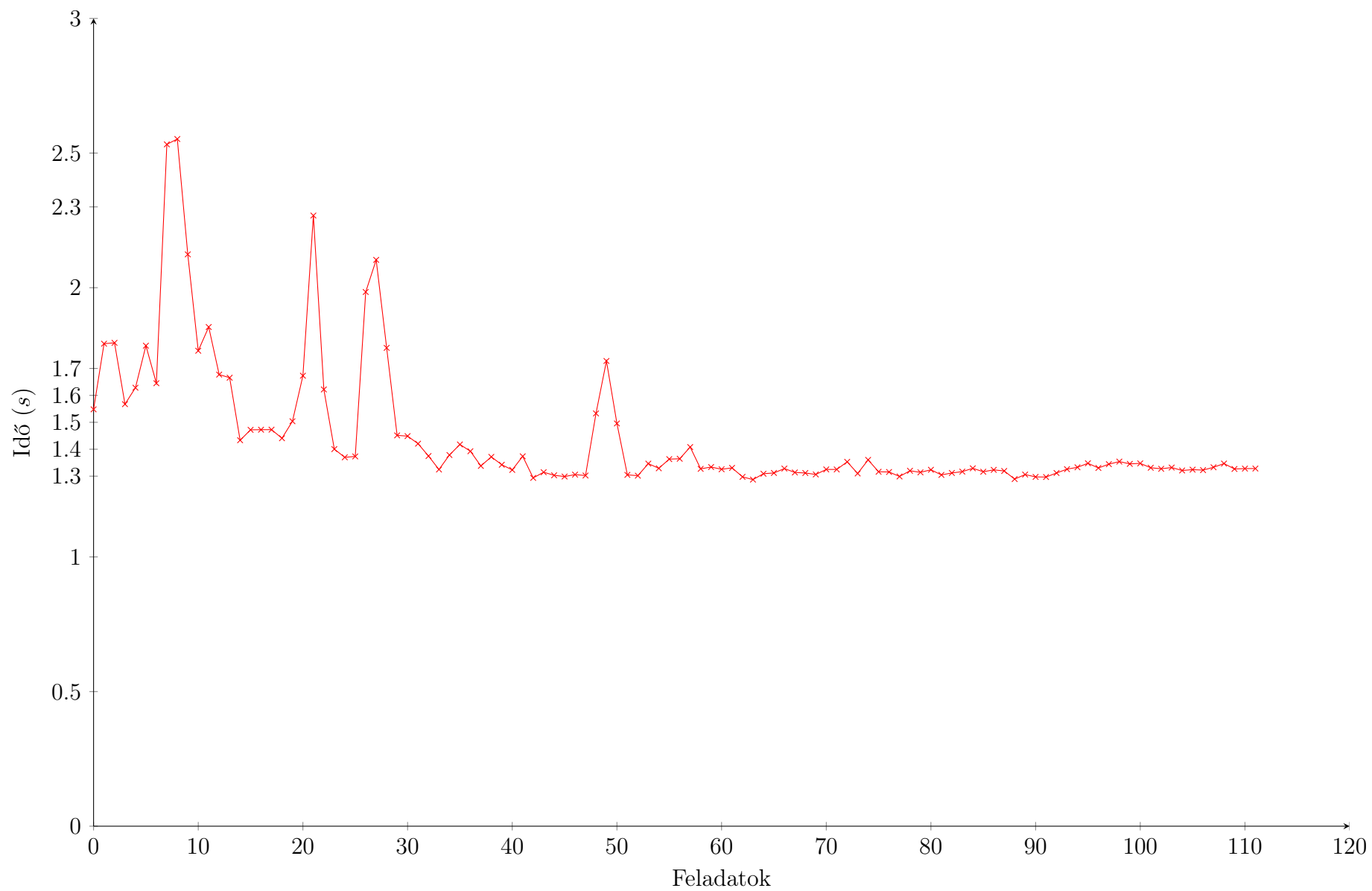
Az említett gráfnál a soros program szűk 10 perc alatt végzett. A párhuzamos program viszont csak 18 processzor segítségével közel 1 perc alatt képes volt megoldani az összes feladatot. További processzorok bevonásával egyre gyorsabb lett a párhuzamos program. 300 processzornál viszont elértük a program korlátait, ahogy az látszik a 4.1 ábrán is, a legnehezebb feladat megoldása 6.29 másodpercbe kerül. Ennél jobb eredményt több processzoron sem lennénk képesek elérni.



4.1. ábra. A p_hat_300-3 nevű gráf feladatainak megoldási ideje.

A MANN_a27 nevű gráf bár kicsinek tűnhet a maga 378 csúcsával, a sűrűsége miatt ez egy elég nehéz feladatnak mondható. A feladatok eloszlása viszonylag egyenletes, csak néhány domináns él van. A soros program 23 másodperc alatt vizsgálta meg ezt a gráfot. Ez a p_hat_300-1 gráfnál mért 4.75 másodperchez képest soknak tűnhet. Valójában nem az, míg a p_hat300-1 gráfnak 0.25-ös a sűrűsége, addig a MANN_a27 gráfé 0.99.

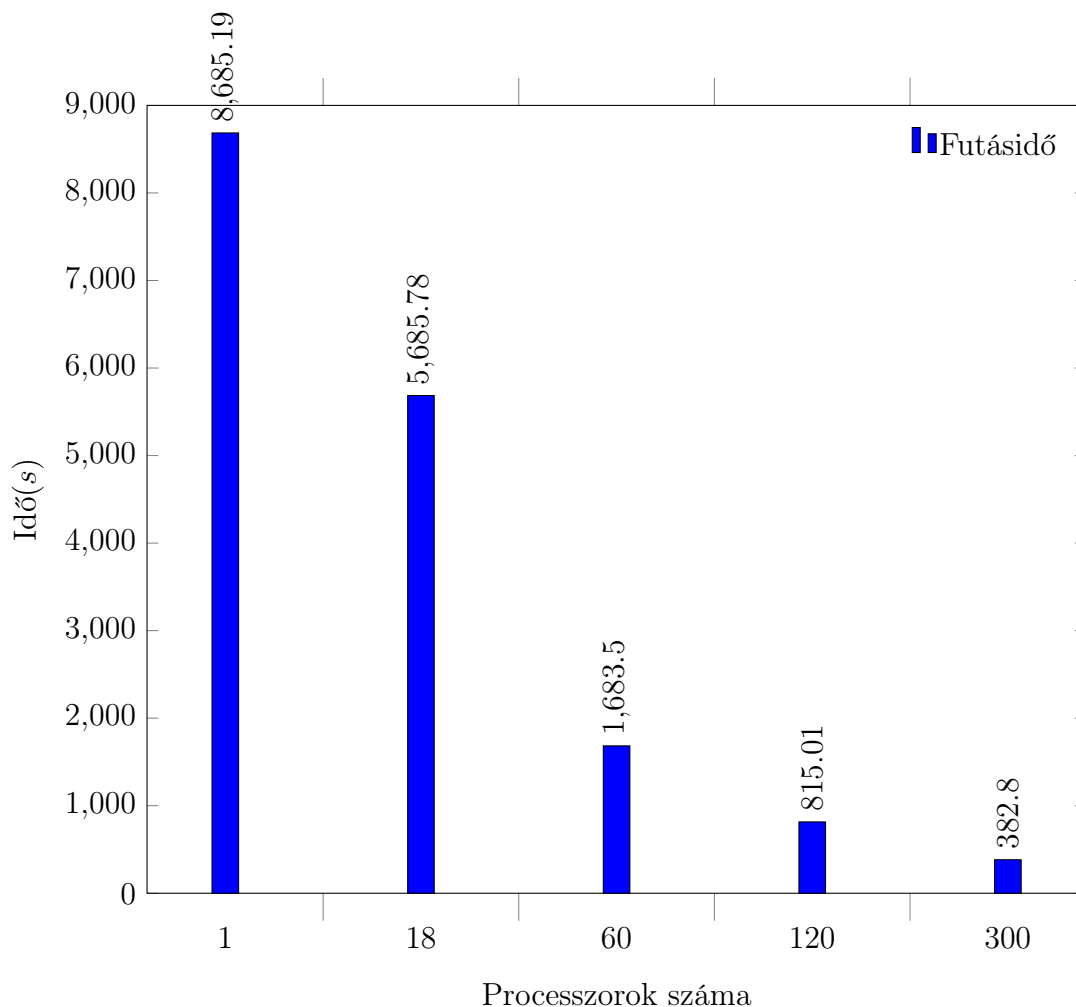
A párhuzamos program 18 processzoron futtatva ez esetben is több, mint kétszeres gyorsulást hozott, 9.9 másodperc alatt sikerült megvizsgálnia mind a 112 feljegyzett élt. További processzorok igénybevételével tovább növelhető a gyorsulás. 60 processzoron már 3.2 másodperc, 120 processzoron pedig 2.6 másodperc a program futásideje.



4.2. ábra. A MANN_a27 nevű gráf feladatainak megoldási ideje.

Meddig tart a gyorsulás?

Úgy tűnhet, hogy a processzorok számának növelésével folyamatos, akár végtelenségig tartó gyorsulás érhető el. Ha megnézzük a 4.3 ábrát, akkor is arra következtethetünk, hogy egyre gyorsabb lehet a program.



4.3. ábra. A program gyorsulása a `keller5` nevű gráf esetében, a processzorok számának növelésével

Ezen következtetésünk sajnos helytelen.

Vannak olyan domináns feladatok, amelyek nehézsége miatt már nem tudunk további gyorsulást elérni több processzoron sem (4.1 ábrán a 77. és 128. feladat). Ekkor hiába növeljük tovább a processzorok számát, csak azt érzük el, hogy egyre több és több szolgá várná tétlenül azt az egy szolgát, amelyik még nem tudta befejezni a neki kiosztott feladatot. Tehát a program futásideje nem csökkenne.

Vegyünk például a `MANN_a27` gráfot. A 4.2 ábrán jól látszik, hogy a legnehezebb feladatot nagyjából 2.6 másodpercig tart megoldania. Hiába vonnánk be több processzort, akkor sem tudnánk ezt az időt lerövidíteni.

Ennél a gráfnál 112 eliminálandó, zavaró élt jegyeztünk fel. Tehát 112 feladata van

a párhuzamos programnak. Ha 300 processzoron futtatnánk a programunkat ezzel a gráffal, akkor csak annyit érnénk el vele, hogy az első 112 folyamat dolgozna, a maradék 188 pedig tétlenül állna a program elejétől a végéig, arra várva, hogy a 112 dolgozó él befejezze a munkáját.

Checkpoint fájlok

A feltételezésünk, hogy azokat az éleket, amelyek a saját színosztályukon belül több *s*-klikket alkotnak, nehezebb kitörölni, mint amelyek kevesebbet, nem hozta a várt eredményt.

Az élek súlyának megfelelően sorba rendezett checkpoint fájl is nagyon hasonló eredményeket hozott, mint a rendezetlen checkpoint fájl. A 4.4 táblázatban legtöbb esetben maximum néhány másodperc eltérés van a két eredmény között.

A `p_hat700-2` gráfnál 300 processzoron futtatva viszont elég nagy az eltérés a rendezett és a rendezetlen checkpoint között. Ez sajnos nem a feltételezésünk miatt van, egyszerűen valamelyik szolga rendre megkapta a nehéz feladatok egyikét. Előfordulhat olyan, hogy egy kivételével az összes szolga már végzett a feladatokkal, és várják azt az egyet, aki az utolsó nehéz feladatot megkapta.

Tehát ez csak a feladatok kiosztásának véletlenszerűsége miatt fordulhatott elő, a checkpoint fájlban az élek rendezettsége aszerint, hogy mennyi *s*-klikket alkotnak nem sokat javíthatott vagy ronthatott az eredményen.

4.2. Szisztematikus mérések

Ebben a részben azt mértük, hogy az *s*-klikk mentes színezések mennyire befolyásolják a program futásidejét. A 4.1 táblázat utolsó 9 gráfját mértük le. Ezek a gráfok nehézségük szerint három kategóriába sorolhatók. Könnyű, közepes és nehéz. Könnyűnek felelnek meg a `matr-del-7`, `monoton-7` és a `p_hat1000-1` gráfok. Közepesnek a `matr-del-8`, `monoton-8` és a `p_hat1000-2`. Nehéznek pedig a `matr-del-9`, `monoton-9` és a `p_hat1000-3`. Mindegyik gráfnak elkészítettük egy legális színezését, egy háromszög mentes színezését, egy 4-klikk mentes színezését és egy 5-klikk mentes színezését. Ezekből a színezésekből újraosztályozással előállítottuk a kvázi színosztályokat és feljegyeztük az eliminálandó zavaró éleket. Ez a folyamat a legális színezés esetén más volt, mint az *s* – *klikk* mentes színezéseknél, hiszen itt minden, a színosztályon belül levő él eliminálandó zavaró élnek felel meg, tehát minden ilyen élt fel kell jegyeznünk törlésre. A prekondicionálás futásidejét a 4.5 táblázat mutatja.

Ha *s*-klikk mentes színezéskor $s=2$ -t adunk meg a programnak, akkor egy „2-klikk mentes” színezését, azaz egy legális színezését készíti el az adott gráfnak. Ez azért érdekes, mert Kiss Annamária és Stráhl István is legális színezésekkel dolgoztak a szakdolgozatuk elkészítése során.

Ezek a mérések már csak a maximum klikknél egyel nagyobb klikk keresésével foglalkoznak, mivel a maximum klikk megtalálása esetleges. Az igazán nehéz feladat viszont az, hogy megmondjuk egy gráfról, hogy a maximumnál egyel nagyobb klikk már nem létezik benne. Ekkor ugyanis nem elég csak egyetlen olyan élt találni ami szerepel a maximum klikkben, hanem az összes feljegyzett élről meg kell mondanunk, hogy nem szerepel egyetlen a maximumnál egyel nagyobb klikkben sem, erről a 4.1.1 fejezet elején írtunk. Valamint már nem használtunk kétféle checkpoint fájlt, hiszen azokkal kapcsolatban az előzőekben nem voltak jó tapasztalataink, itt mindig a sorba rendezett fájlokat használtuk.

A 4.4, 4.6, 4.9, 4.10, 4.13 valamint a 4.15 diagramok esetében sorba rendeztük a feladatokat, azok megoldási ideje szerint, ezáltal láthatóvá vált a feladatok nehézségének megoszlása. A jó láthatóság érdekében ezeknél a diagramoknál logaritmikus skálázást használtunk.

A 4.6, 4.7 valamint a 4.8 táblázatok értelmezése:

- Az első sorban a vizsgált gráf neve található.
- A második sorban a gráf maximum klikkje szerepel.
- A harmadik sorban a soros Lilla futásideje.
- A további sorok pedig színezésekre (legális, háromszögmentes, 4-klikk mentes, 5-klikk mentes) lebontva a következők:
 - Feladatok száma - az eliminálandó élek száma
 - 18, 60, 120, 300 - a 18, 60, 120 és 300 processzoron mért futásidők, másodpercben.

Ahol '-' szerepel ott nem készítettünk mérést, a feladatok mennyisége miatt, vagy azok nehézsége miatt. Ahol * szerepel ott pedig nem futott le a program a megadott időn belül.

4.2.1. Prekondicionálások

Gráf neve	$s=2$ (legális színezés)		$s=3$ (háromszögmentes színezés)		$s=4$ (4-klikk mentes színezés)		$s=5$ (5-klikk mentes színezés)	
	színezés	újraosztályozás	színezés	újraosztályozás	színezés	újraosztályozás	színezés	újraosztályozás
matr-del-7	0.07	0.0001	0.05	0.04	0.04	0.22	0.04	1.32
matr-del-8	0.52	0.01	0.28	0.21	0.22	1.261	0.21	11.62
matr-del-9	3.95	0.05	2.091	1.011	1.39	10.169	1.32	116.06
monoton-7	0.8	0.03	0.38	1.44	0.311	28.14	0.53	186.76
monoton-8	2.091	0.04	1.021	6.212	0.79	193.57	1.41	419.801
monoton-9	6.201	0.09	2.87	15.36	2.249	329.27	4.281	13789.7
p_hat1000-1	9.76	0.29	4.261	209.98	2.81	5810.86	3.32	85843.3
p_hat1000-2	19.24	0.22	9.271	17.2	6.159	221.02	5.211	3162.64
p_hat1000-3	36.94	0.32	18.77	16.36	12.83	180.25	10.32	2495.83

4.5. táblázat. Az oszlopok azt mutatják, hogy melyik színezés mennyi ideig tartott, illetve, hogy ebből a színezésből mennyi ideig tartott előállítani az új színosztályokat és feljegyezni az eliminálandó éleket.

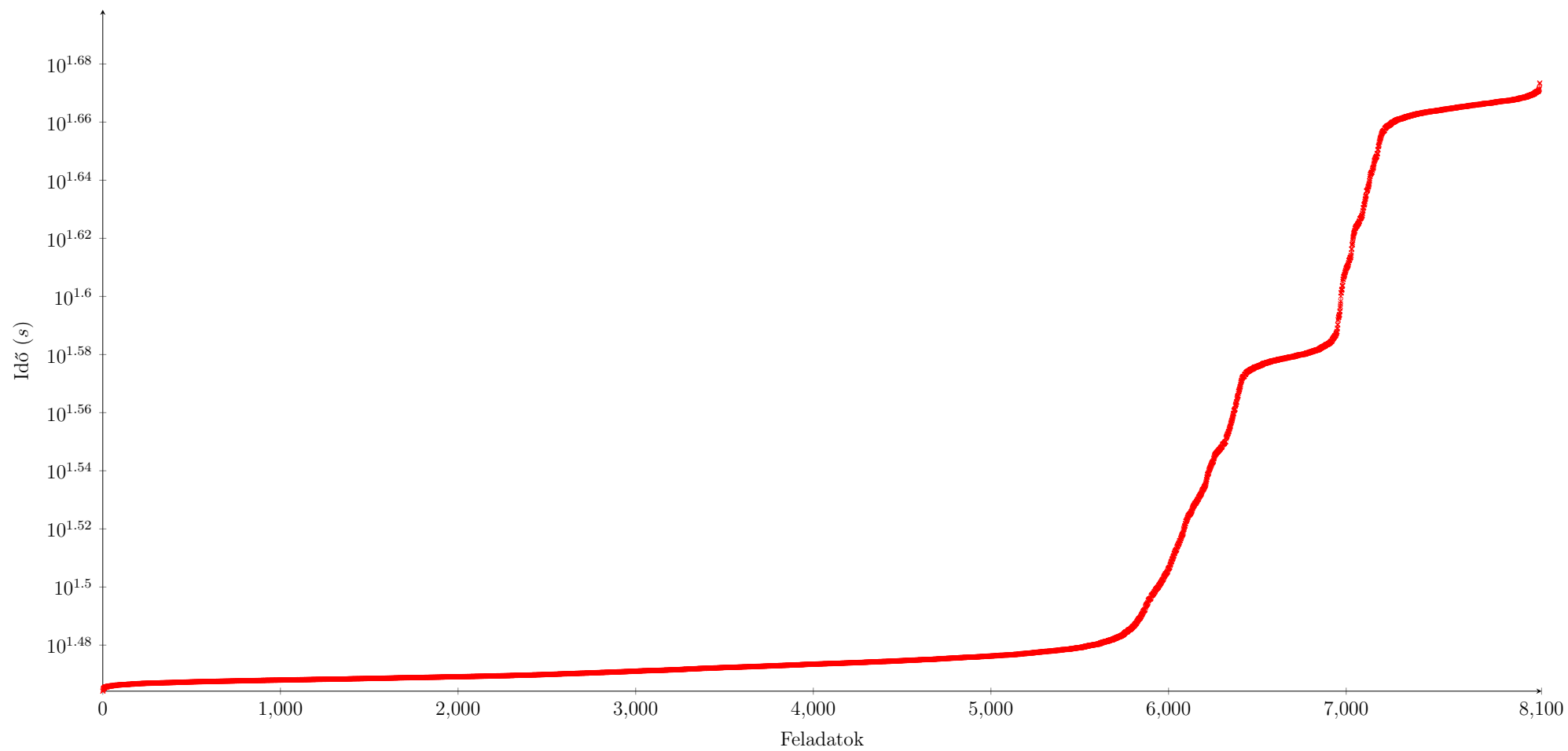
4.2.2. Könnyű feladatok

Gráf neve		matr-del-7	monoton-7	p_hat1000-1
Maximum klikk		16	19	10
Soros program futásideje		0.430	15.760	173.860
$s=2$	Feladatok száma	60	582	8091
	18	0.469	75.748	15947.500
	60	0.225	20.379	4689.950
	120	-	5.034	2354.910
	300	-	5.033	914.396
$s=3$	Feladatok száma	186	684	11304
	18	1.340	85.099	21900.600
	60	0.451	24.950	6389.410
	120	0.301	12.886	3154.124
	300	0.199	6.564	1272.430
$s=4$	Feladatok száma	215	734	13585
	18	1.529	91.863	26719.500
	60	0.476	26.985	7799.800
	120	-	14.206	3851.580
	300	0.169	6.795	1547.130
$s=5$	Feladatok száma	201	560	14014
	18	1.461	71.612	27934.500
	60	0.469	21.285	8145.380
	120	0.323	11.221	4013.920
	300	0.277	5.188	1615.450

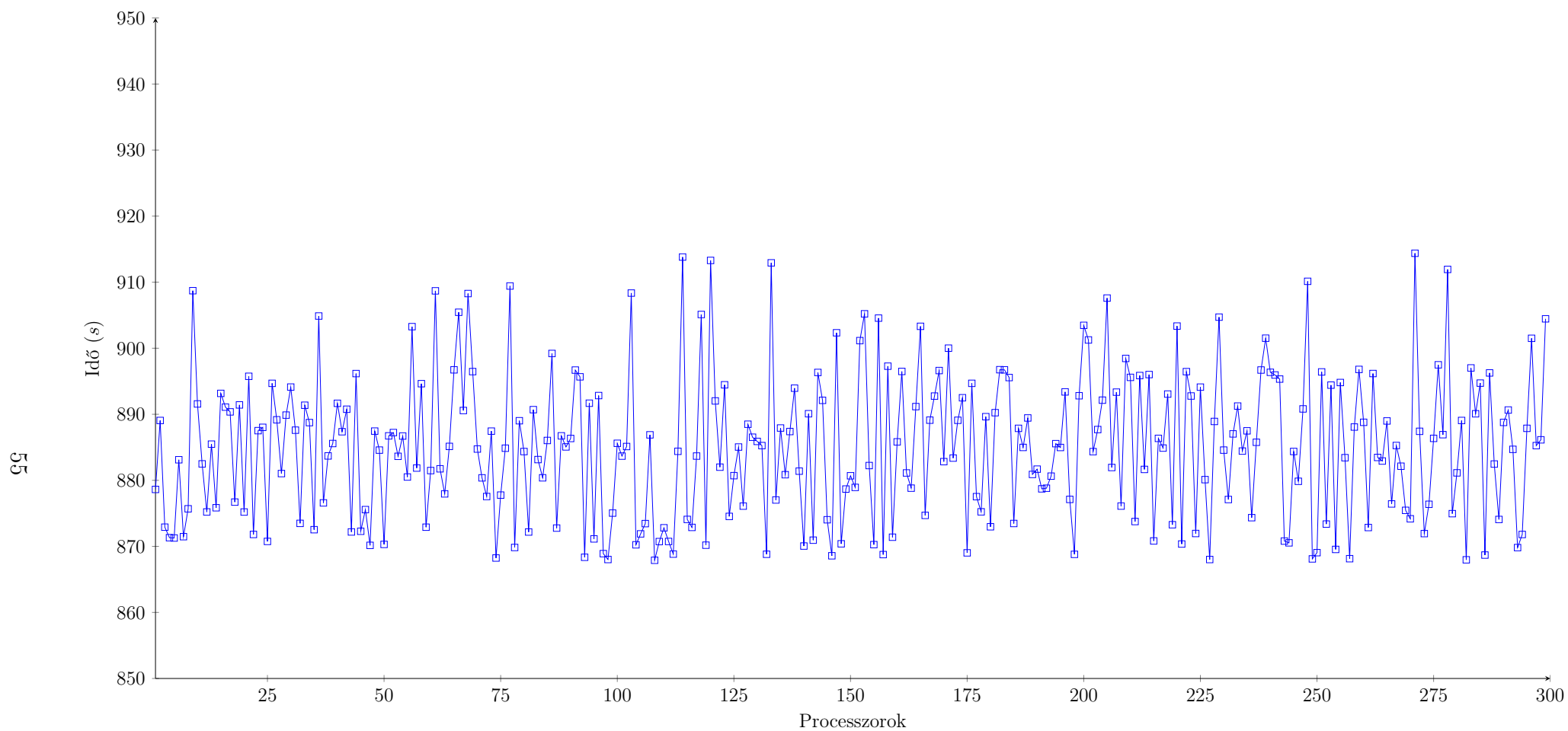
4.6. táblázat. A könnyű feladatok megoldási ideje.

A legszembetűnőbb eredményt ebben a kategóriában a `p_hat1000-1` nevű gráf hozta. Az előző részben már szó esett róla, hogy a könnyű gráfokat nem érdemes párhuzamos programmal futtatni, ez itt is bebizonyosodott. A 4.5 táblázatból is látszik, hogy $s=3$, azaz háromszögmentes színezés esetén már az újraosztályozás és az eliminálandó élek feljegyzése tovább tartott, mint a soros programnak megoldani az egész feladatot (4.6). Az említett gráfnak két színezésével készült eredményeit vizsgáljuk meg. A 4.4 ábra a legális ($s=2$) színezés után feljegyzett eliminálandó éleket (feladatokat) illetve azok megoldási idejét mutatja. A 4.6 ábra a háromszögmentes ($s=3$) színezés után feljegyzett eliminálandó éleket és azok megoldási idejét mutatja. Az ábrákból látszik, hogy hasonló nehézségűek a feladatok és ezek nehézség szerint hasonlóan is oszlanak el egymáshoz képest. A feladatok mennyiségében viszont nagy különbség van. Legális színezés esetén 8091 eliminálandó él volt, háromszögmentes színezés esetén pedig 11304. Mivel a feladatok nehézsége hasonló, ezért csak a számuk befolyásolta a program teljes futásidejét. Ahol kevesebb eliminálandó él volt, ott a program közel 300 másodperccel gyorsabban képes volt megvizsgálni az összes feladatot. Ebből arra lehet következtetni, hogy minél kevesebb élt sikerül feljegyezni törlésre, annál gyorsabb lehet a programunk. Ez sok helyen igaz lesz, ahol hasonló a feladatok nehézsége. Ott ahol akad legalább egy, esetleg néhány olyan feladat, aminek a megoldása jóval tovább tart, mint a többi, már sokat lassulna a programunk, egy olyan esethez képest, ahol több, de könnyebb feladat van. A 4.5 és 4.7 ábrák pedig azt mutatják, hogy a folyamatok mennyi időt töltöttek a feladatok megoldásával. Ebben az esetben mindegyik színezés megfelelő alapot adott, nem volt a feladatok között kiugróan nehéz. Ebből adódóan a feladatok eloszlása is arányos volt. Nem fordult elő olyan, hogy valamelyik szolga kapott egy nehéz feladatot, ami lefoglalta volna a program futása során. A legnagyobb eltérés mindkét esetben legfeljebb 50-60 másodperc ami ekkora nagyságrendnél elhanyagolható.

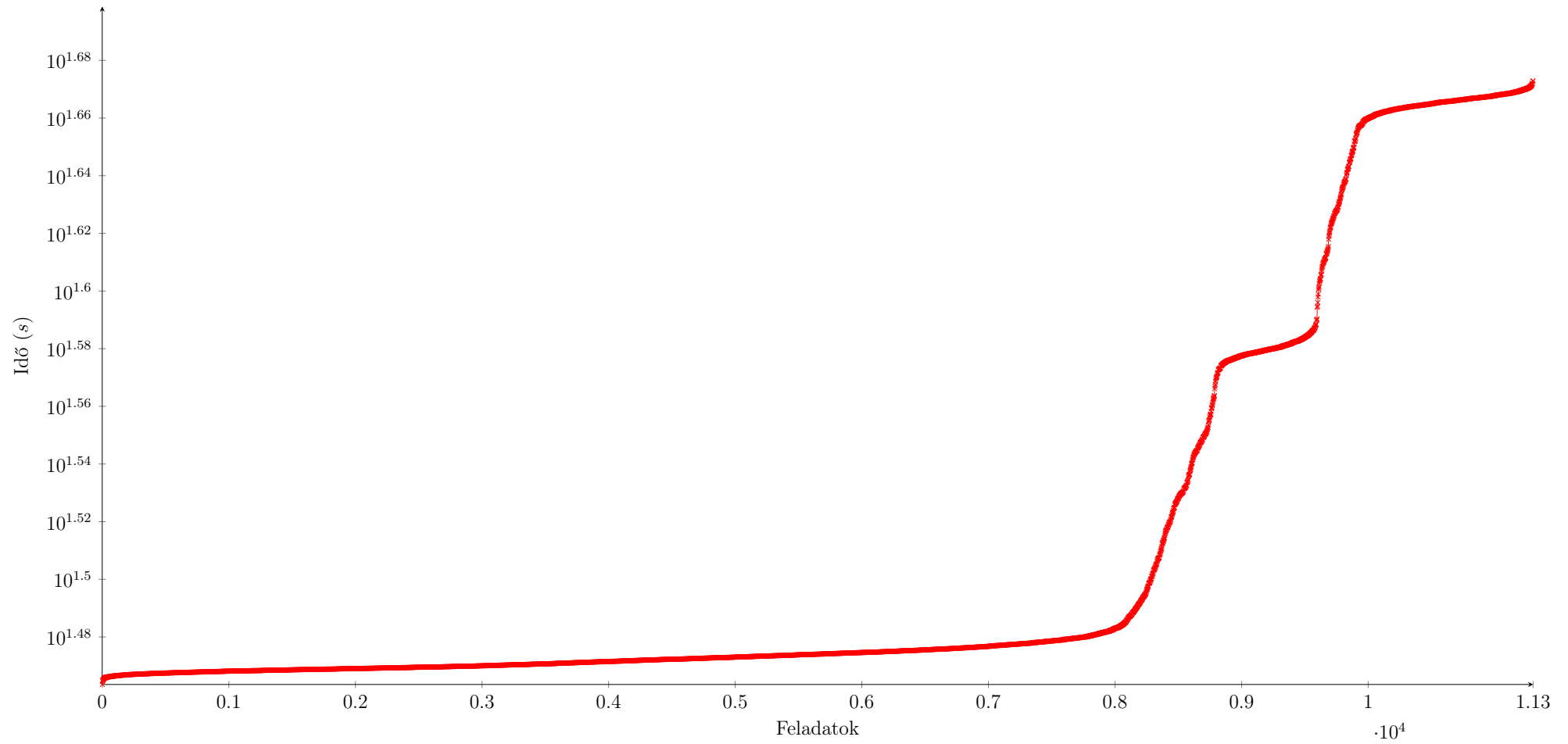
Összességében a könnyű feladatokat továbbra sem érdemes a párhuzamos programra bízni, sok esetben már a prekondicionálás jóval tovább tart, mint a soros programnak megoldani a teljes feladatot. Ennek ellenére több helyen is megfigyelhető gyorsulás a soros programhoz képest. Például a `matr-del-7` és `monoton-7` gráfok esetében, 120 és 300 processzornál egyértelmű a gyorsulás, de ezek néhány másodpercek, feleslegesen sok erőforrást igényelnek.



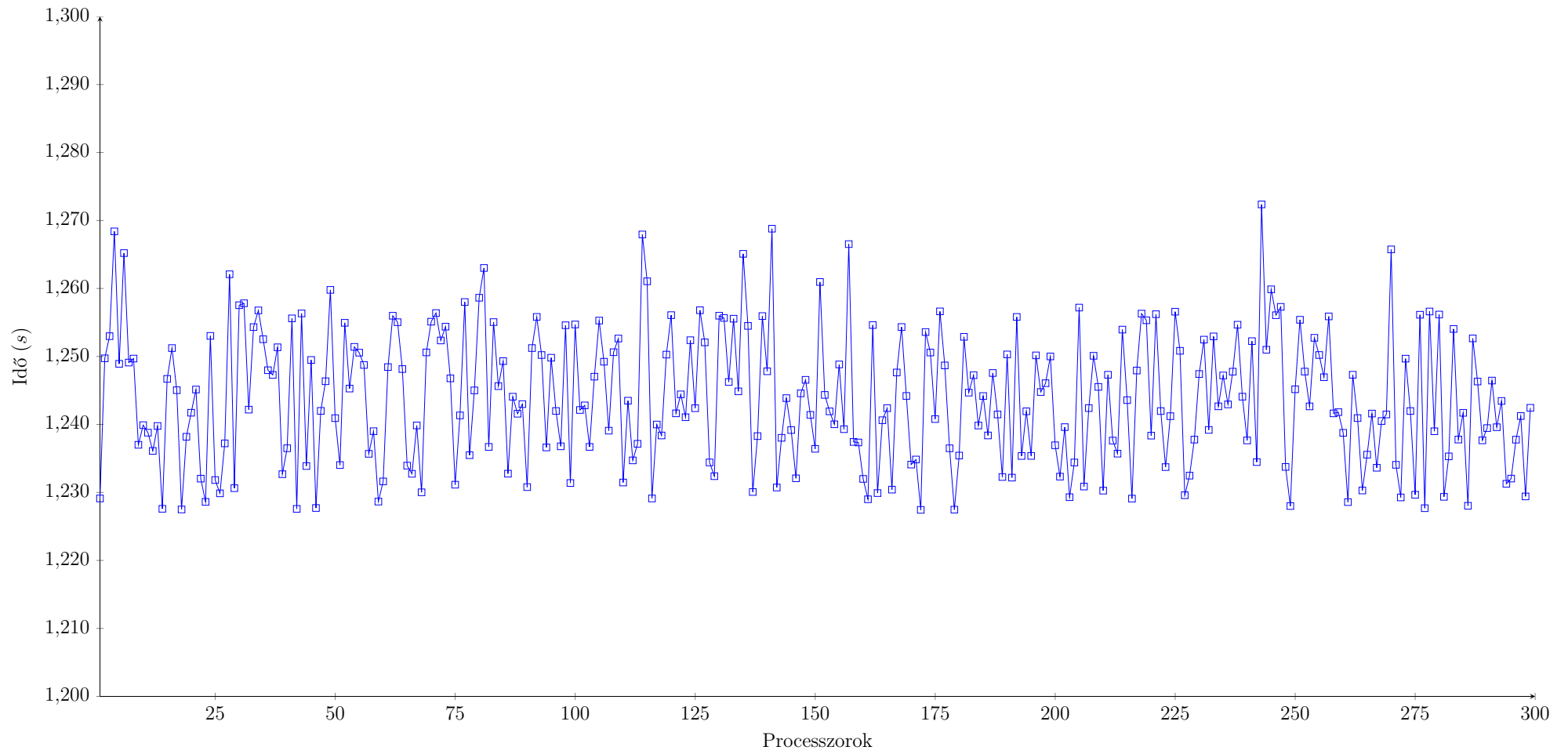
4.4. ábra. A p_hat_1000-1 nevű gráf feladatainak megoldási ideje, $s = 2$ esetén, 300 processzorral mérve.



4.5. ábra. A p_hat_1000-1 nevű gráf feladatait megoldó folyamatok futásiideje, $s=2$ esetén, 300 processzorral mérve.



4.6. ábra. A p_hat_1000-1 nevű gráf feladatainak megoldási ideje, $s = 3$ esetén, 300 processzorral mérve.



4.7. ábra. A p_hat_1000-1 nevű gráf feladatait megoldó folyamatok futásideje, $s=3$ esetén, 300 processzorral mérve.

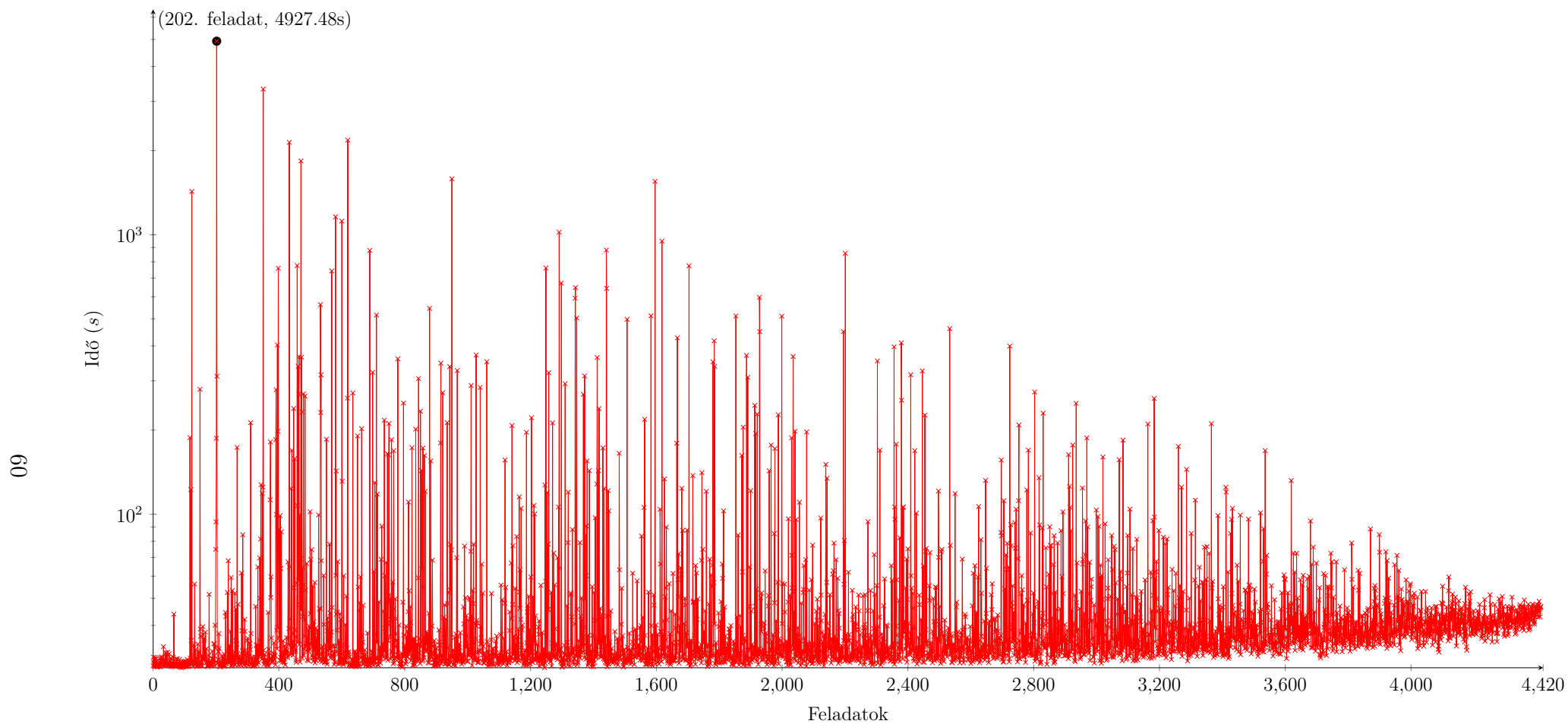
4.2.3. Közepes feladatok

Gráf neve		matr-del-8	monoton-8	p_hat1000-2
Maximum klikk		30	23	46
Soros program futásideje		9.410	4965.140	249801.000
$s=2$	Feladatok száma	168	1138	2246
	18	9.392	999.157	9061.820
	60	2.933	330.460	6622.470
	120	1.851	207.236	6605.350
	300	-	127.138	6548.470
$s=3$	Feladatok száma	373	1325	3422
	18	20.073	909.037	11677.400
	60	6.082	530.512	5191.480
	120	3.414	501.024	5083.270
	300	1.800	489.372	5015.290
$s=4$	Feladatok száma	520	1572	4250
	18	28.074	1015.090	13360.300
	60	8.291	299.376	6195.690
	120	4.546	182.456	6050.180
	300	2.691	153.587	5981.840
$s=5$	Feladatok száma	551	1887	4414
	18	29.211	1226.890	13706.900
	60	8.742	361.552	-
	120	4.554	205.239	4945.120
	300	2.005	146.327	4927.490

4.7. táblázat. A Közepes feladatok megoldási ideje.

A 4.7 táblázatból látható, hogy szinte mindenhol gyorsabb volt a párhuzamos program. Ezek a gyorsulások már kevés processzoron is jól megfigyelhetők. A soros program a `p-hat1000-2` gráfról közel 3 nap alatt tudta megmondani, hogy nem tartalmaz 47 méretű klikket. A párhuzamos programnak 18 processzoron viszont csak 2,5 óra volt, ami majdnem 30-szoros gyorsulást hozott. Viszont ha megnézzük a további eredményeket akkor azt látjuk, hogy a 1,5 órás határt már nem nagyon tudjuk átlépni, akármennyi processzort is vonunk be a folyamatba és akármilyen színezést is veszünk alapul. Ez itt is a feladatok nehézsége miatt fordulhat elő. A 4.8 ábrán a 202. feladat megoldási ideje megegyezik a program teljes futásidejével, tehát további processzorok bevonása itt sem segítené.

A másik érdekesség ebben a kategóriában, hogy míg máshol már 18 processzor is legalább 3-4-szeres gyorsulást mutatott, addig a `matr-del-8 s`-klikk mentes színezéseknél (háromszögmentes, 4-klikk mentes, 5-klikk mentes) jóval lassabb volt, mint a soros program. Legális színezésnél azonban közel azonos időt futott a soros és a párhuzamos is. A feladatok nehézsége mindegyik színezést használva hasonló volt, 0.8 másodperctől 2 másodpercig tartott megoldani őket. A feladatok száma viszont egyre nőtt az s növelésével együtt. Ezért hiába tartott mindegyik feladat megoldása közel ugyanannyi ideig, a feladatok mennyisége miatt egyre lassabb lett a program.



4.8. ábra. A $p_{\hat{1000}-2}$ nevű gráf feladatainak megoldási ideje, $s = 5$ esetén, 300 processzorral mérve.

4.2.4. Nehéz feladatok

Gráf neve		matr-del-9	monoton-9	p_hat1000-3
Maximum klikk		52	28	68
Soros program futásideje		*	*	*
$s=2$	Feladatok száma	607	2306	2527
	18	-	146460.000	-
	60	-	45446.600	-
	120	-	30870.100	-
	300	*	22804.900	*
$s=3$	Feladatok száma	1046	2040	4311
	18	-	107653.100	-
	60	-	55992.500	-
	120	604800!	55657.600	-
	300	*	55547.700	*
$s=4$	Feladatok száma	1233	1869	5066
	18	-	77938.500	-
	60	-	28383.400	-
	120	-	19531.500	-
	300	*	14725.120	*
$s=5$	Feladatok száma	1177	1940	5839
	18	-	134325.100	-
	60	-	134400.000	-
	120	-	131254.900	-
	300	*	131587.100	*

4.8. táblázat. A nehéz feladatok megoldási ideje.

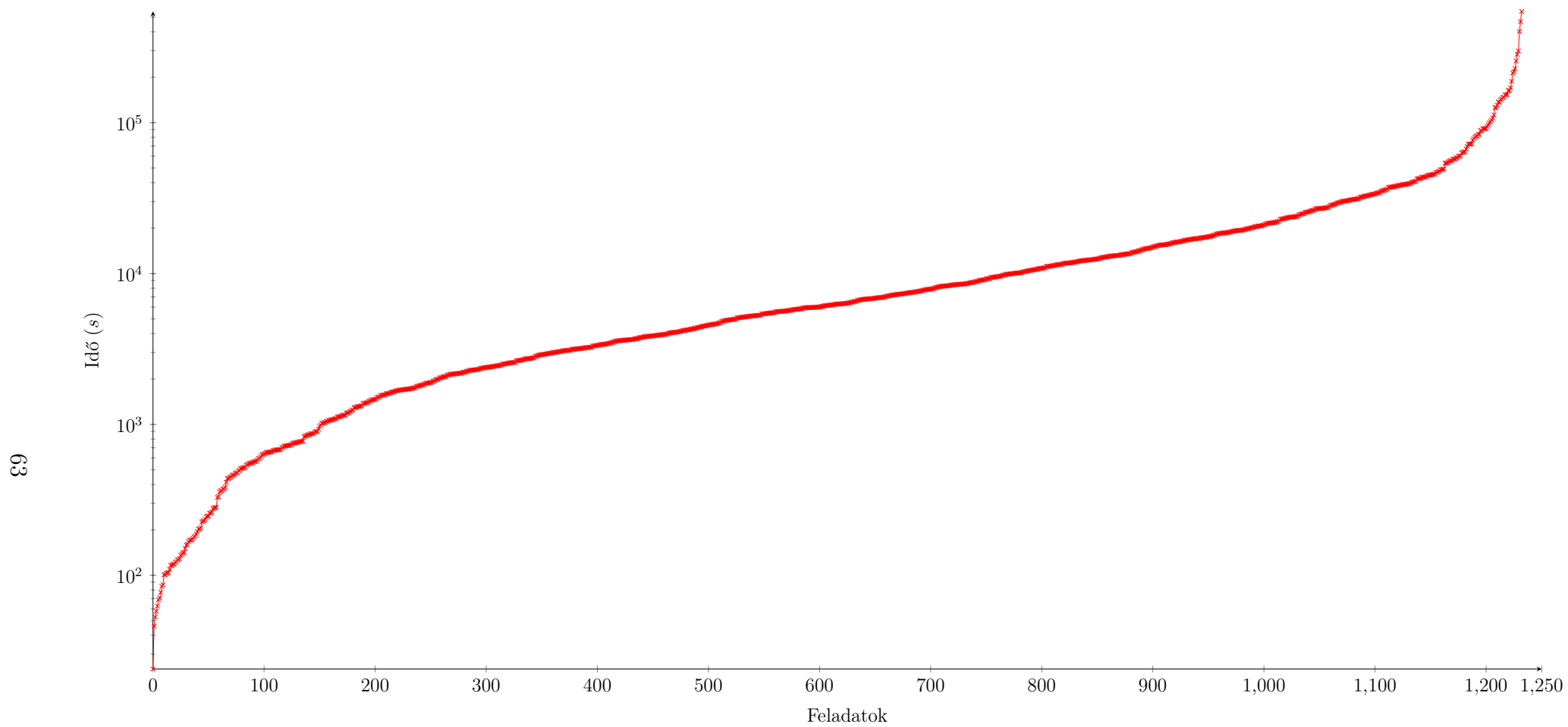
Ahol a 300 processzor is néha kevés

Ezeknek a feladatoknak a megoldásával nem birkózott meg a soros program belátható időn belül. Azonban még a párhuzamos programnak is olyan sokáig lett volna szüksége nagy erőforrásokra, hogy sok beküldött job sorra sem került, ilyenek a `matr-del-9` és a `p_hat_1000-3` gráfok.

Fontos megjegyezni, hogy ezeknél a gráfoknál már valóban nehéz feladatot jelent annak bizonyítása, hogy adott méretű klikk nem létezik bennük. A `monoton-9` gráfot Szabó Sándor szimmetriák segítségével tudta redukálni, hogy megoldható legyen. [Sza13] Ezen kívül Zaválnij Bogdán tudta párhuzamosan megoldani. [Zav14] Így ezek az eredmények a 4.8 táblázatban található eredmények megerősítésére szolgálnak.

A 4.8 táblázatban, a `matr-del-9` gráf egy hétig futott 120 processzorral. Ez alatt az egy hét alatt az első 4 napon végzett a feladatok 90%-ával. A további 3 napon végzett a többi feladattal is, egy kivételével. Az ütemező a 7. napon megszakította a job-ot, mert lejárt az időlimitnek megadott 7 nap. Ezután újra elindítottuk a feladatot további 10 napra, ekkor már csak 6 processzoron (ez a legkevesebb megadható processzor szám a pécsi szupergépen). A checkpoint fájlunk köszönhetően csak azt az egy feladatot kellett megoldania a programnak amit előzőleg nem sikerült. Ezeknek a feladatoknak a megoldási ideje nagyjából 6 és fél nap volt (564055 másodperc), tehát 6,5 napig tartott megmondani a programunknak azt, hogy a `matr-del-9` gráfban nincsen a maximumnál egyel nagyobb klikk. Elsőre pedig azért nem sikerült megoldani 7 nap alatt az összes feladatot, mert ez a nehéz feladat nem az első 120 feladat között volt, ezért később került sorra. Újrakezdekor pedig az egész él vizsgálatát előlről kellett kezdeni. Tehát erről a gráfról körülbelül 7-8 napig tartott megmondania a programunknak, hogy nem tartalmaz 53 méretű klikket: 4.9 ábra.

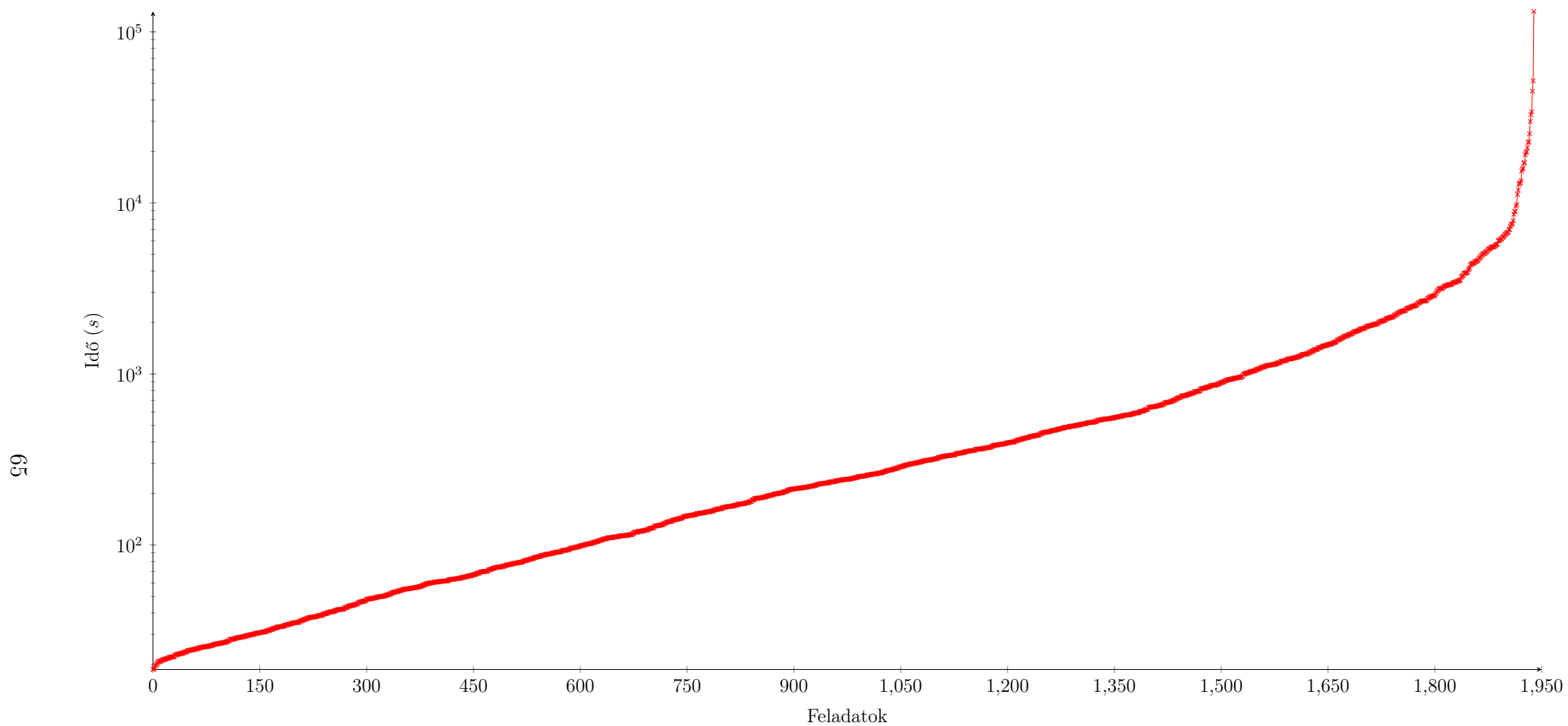
Továbbá fontos az is, hogy eddig a `matr-del-9` mátrixot csak IP és SDP módszerekkel tudták megoldani, így a mi megoldásunk az első kombinatorikus megoldása a gráfnak. [Slo]



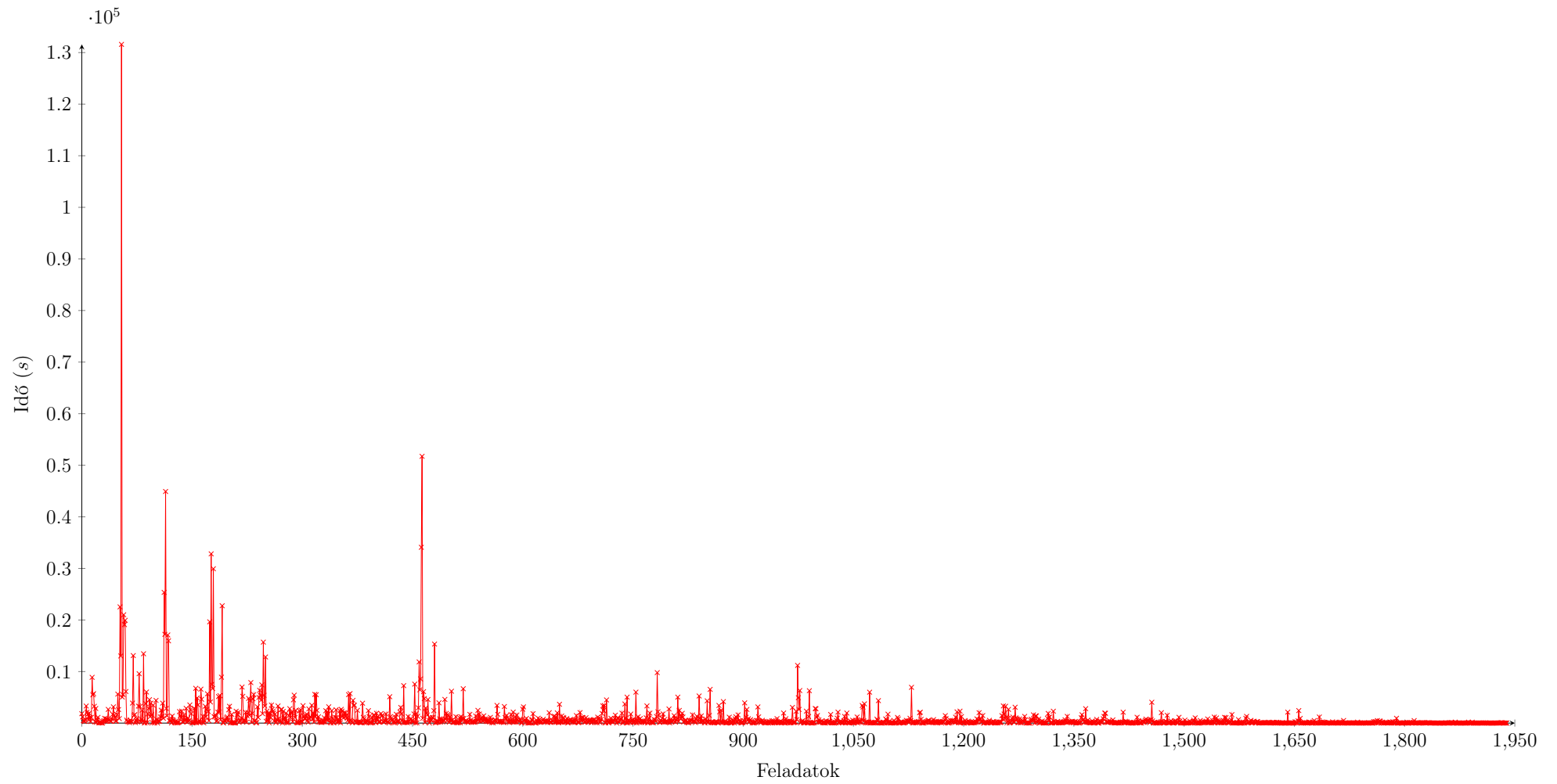
4.9. ábra. A matr-del-9 nevű gráf feladatainak megoldási ideje, $s = 3$ esetén.

Feladatok nehézsége

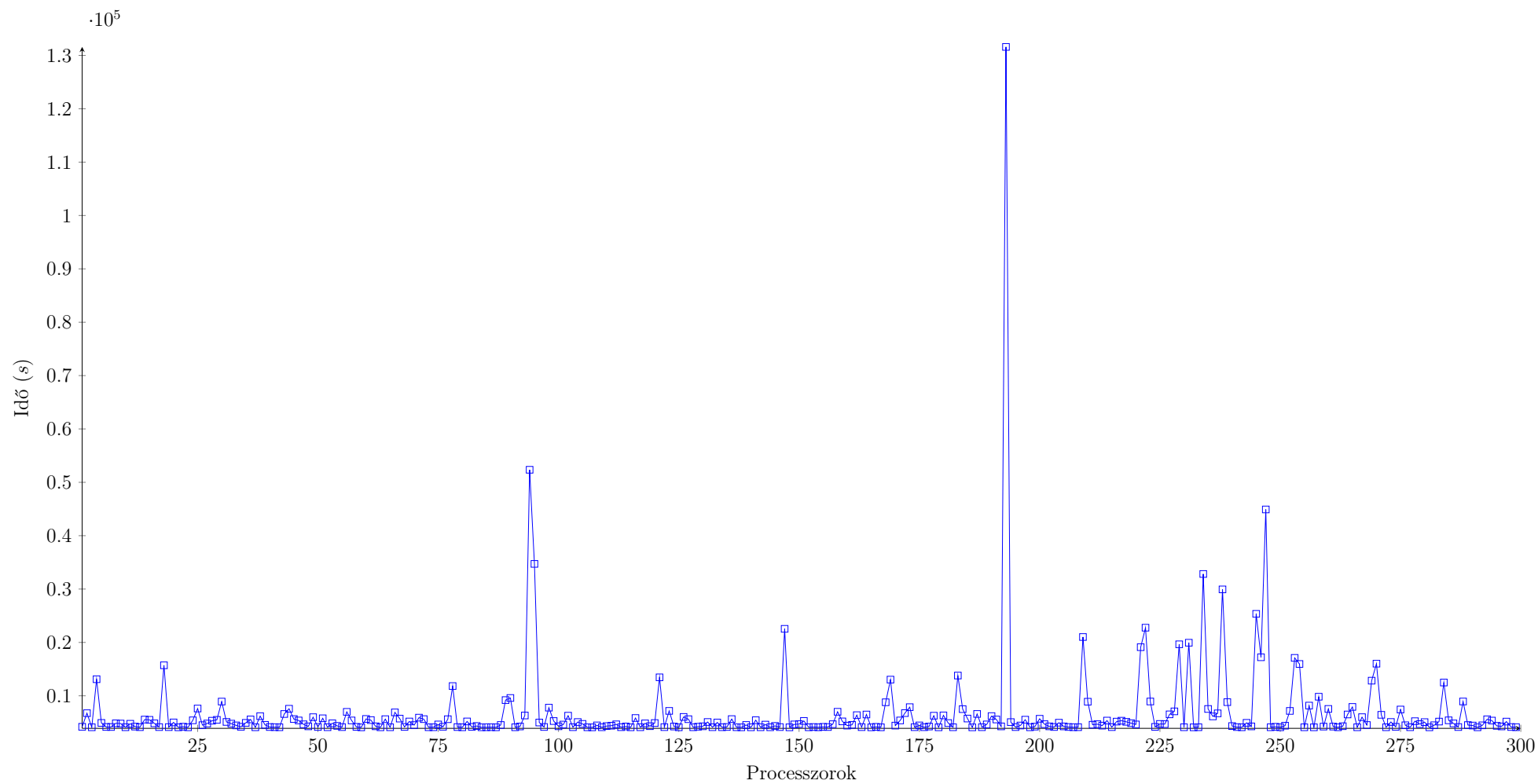
A *monoton-9* gráf más-más színezésekkel elég vegyes eredményeket hozott. 5-klikk mentes színezéskor 18 processzoron is körülbelül ugyanannyi időbe telt megoldani a feladatokat, mint 300 processzoron (4.8 táblázat). A 4.10 ábrán látszik, hogy a legnehezebb feladatot (131587 másodperc - 36.5 óra) több, mint kétszer annyi idő alatt tudta megoldani a program, mint az azt követő legnehezebb feladatot (51729 másodperc - 14.3 óra). A szolgák körülbelül 14-15 óra alatt végeztek a feladatokkal (4.12 ábra), csak egy szolga dolgozott ezután. Tehát az 5-klikk mentes színezésnél nem sikerült közel azonos nehézségű feladatokat találnunk, ezért az az egy feladat elhúzta az egész program futásidejét. Más a helyzet 4-klikk mentes színezéskor. Az átlag feladatok nehézsége hasonló (4.13 ábra), mint az 5-klikk mentes színezésnél, ellenben itt nincsenek annyira kiugróan nehéz feladatok, így sikerült a teljes feladatot megoldani alig 4 óra alatt. Nem volt a folyamatok között olyan, amelyik kirívóan tovább dolgozott volna mint a többi (4.14 ábra).



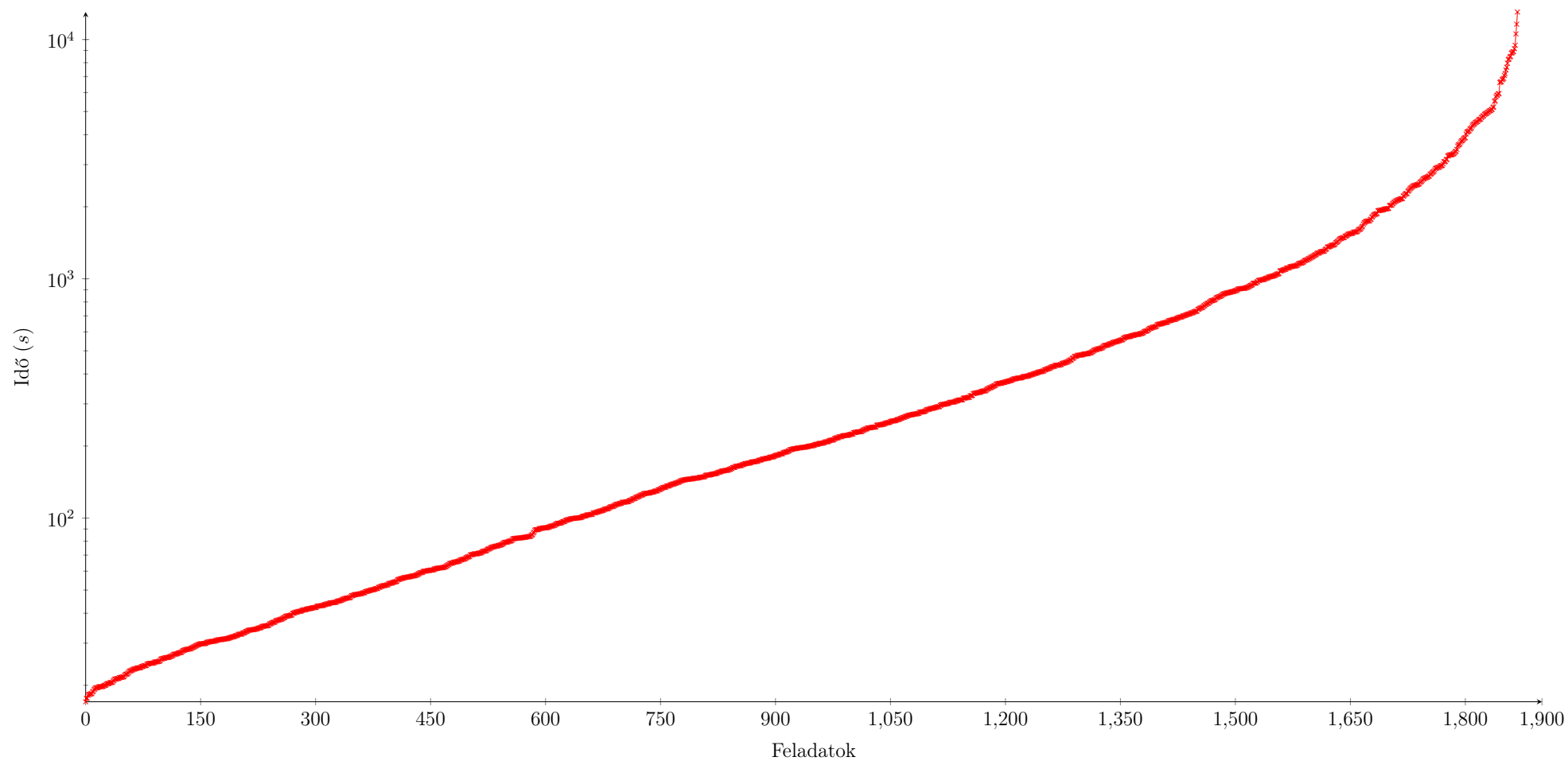
4.10. ábra. A `monoton-9` nevű gráf feladatainak megoldási ideje, $s = 5$ esetén, 300 processzorral mérve.



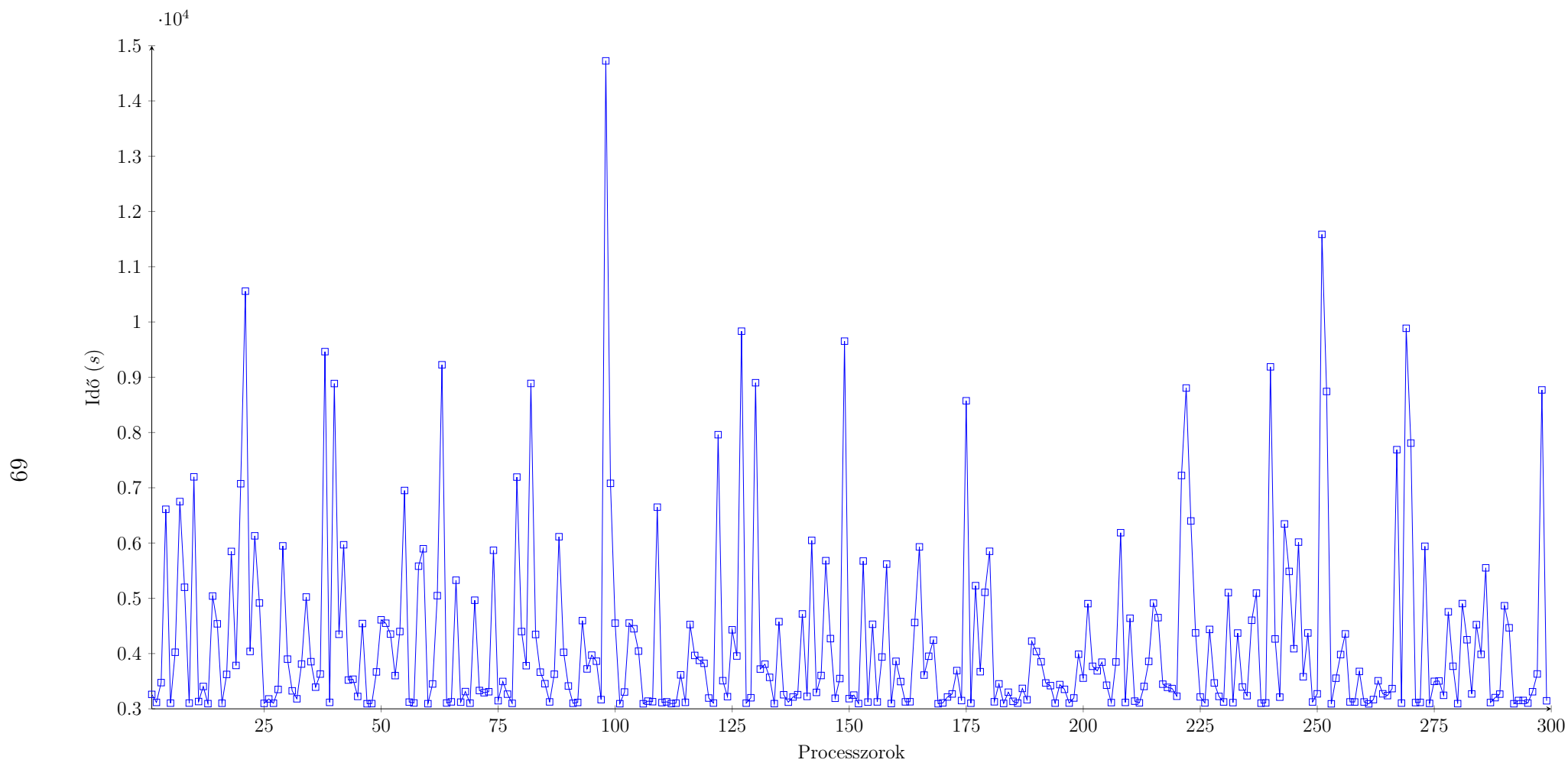
4.11. ábra. A `monoton-9` nevű gráf feladatainak megoldási ideje, $s = 5$ esetén, 300 processzorral mérve. A feladatok a feljegyzés sorrendjében szerepelnek.



4.12. ábra. A *monoton-9* nevű gráf feladatait megoldó folyamatok futásideje, $s=5$ esetén, 300 processzorral mérve.



4.13. ábra. A `monoton-9` nevű gráf feladatainak megoldási ideje, $s = 4$ esetén, 300 processzorral mérve.



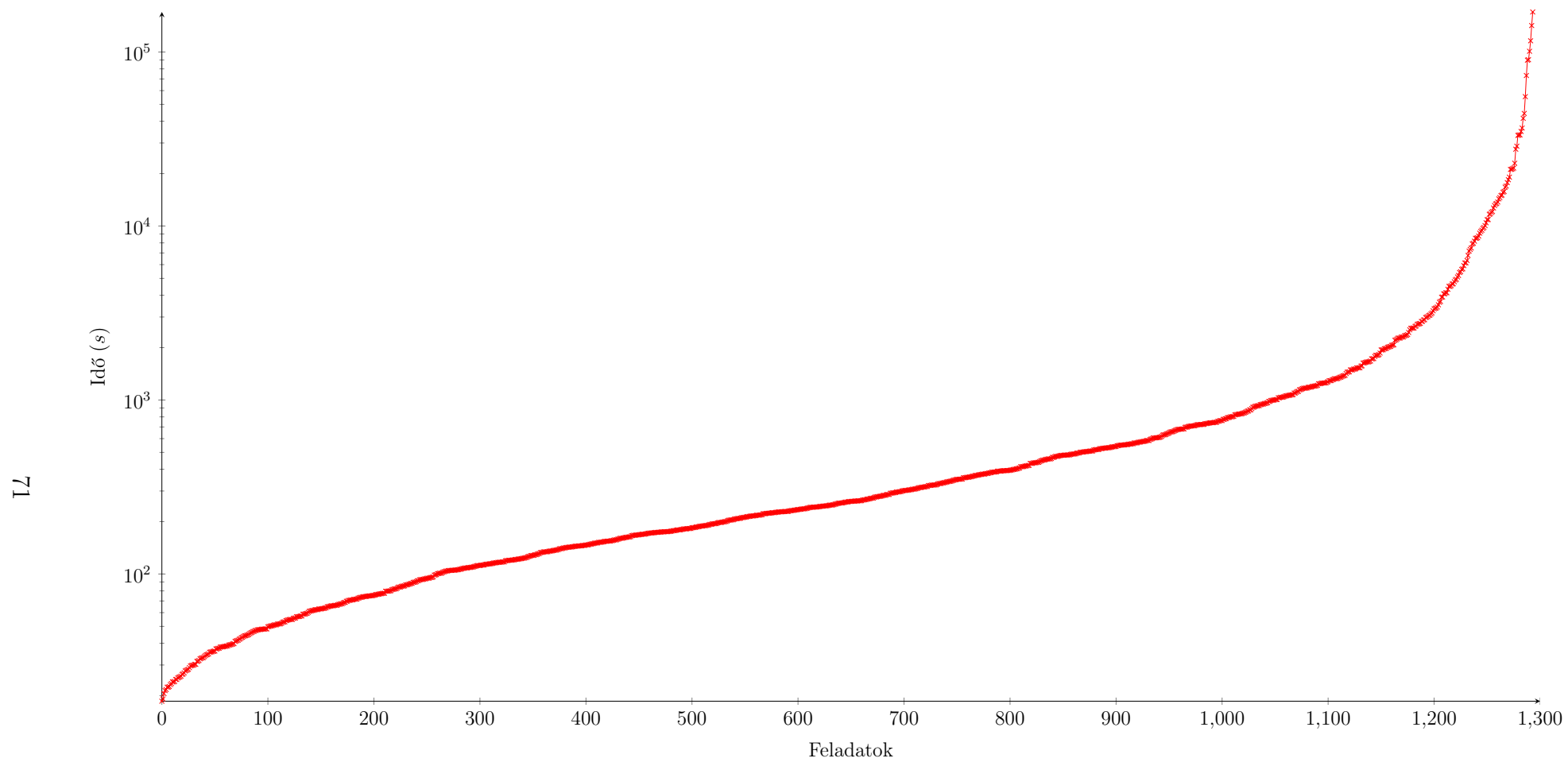
4.14. ábra. A *monoton-9* nevű gráf feladatait megoldó folyamatok futásideje, $s=4$ esetén, 300 processzorral mérve.

4.3. További tesztek

4.3.1. Modularitás

A programunkat kipróbáltuk más színezők által elkészített col fájlokkal is. Az eredmények hasonlóak voltak az általunk mért eredményekkel, azonban ez jól tükrözi, hogy a programunk képes együtt működni más programokkal, algoritmusokkal. Nem csak a színezőt lehet kicserélni, hanem ha valakinek van egy jobb ötlete az eliminálandó élek feljegyzésére, az nyugodtan használhatja a saját megoldását, mindössze csak a checkpoint fájl formátumának kell megfelelnie.

Az említett színezést egy, a témavezetőnk által írt **DSATUR** színező készítette és a **monoton-9** gráf egy legális színezése volt. [Bre79] Az újraosztályozás és az eliminálandó élek feljegyzése után 1294 feladat keletkezett. Az összes feladat megoldási ideje 169859 másodperc volt. Ezek közül a legnehezebb feladat megoldásának ideje megegyezett a program futás-idejével, azaz 169859 másodperc. A feladatok számát tekintve ugyan jobbnak ígérkezik, az általunk készített legális színezésnél (4.8), azonban a feladatok nehezebbek lettek, ezért elhúzódott a program futás ideje is (4.15).



4.15. ábra. A `monoton-9` nevű gráf feladatainak megoldási ideje, A `DSATUR` színező által elkészített legális színezést használva, 300 processzorral mérve.

4.3.2. Feladatok megoldásának sorrendje

Azt is megvizsgáltuk, hogy a feladatok megoldásának sorrendje milyen hatással van a többi feladat megoldásának futásidejére. Azaz megnéztük, hogy egy nehéz feladatot mennyiben könnyíthet meg egy jól megválasztott feladat elosztási sorrend. Eredetileg ezt az elosztást mi a 3.3 fejezetben leírt sorba rendezett checkpoint fájljal szeretnénk volna megtenni, azonban az a módszer nem volt túl hatékony. Ez a 4.11 ábrán is látszik, rögtön az elején van a legnehezebb feladat. Miután végzett a program ennek a gráfnak a feladatainak vizsgálásával, ezt a legnehezebb feladatot kitöröltük a checkpoint fájl azon részéből, amelyben a már megoldott feladatok vannak és a megoldandó feladatok végére tettük. Ekkor a program a checkpoint fájlunk köszönhetően az összes feladatot megoldottnak tekintette, kivéve ezt az utolsót, ami a legnehezebb feladat volt. Mivel átraktuk ezt a feladatot a megoldandó feladatok végére, ezért az összes többi, előtte levő feladatot törölhetette a szolgálja a saját gráf másolatából, hiszen azok már meglettek oldva más szolgálak által. Az így kapott eredmény meglepő volt. Ami eddig a legnehezebb feladat volt és elsőre több, mint 36 órába telt a megoldása, most **73 másodpercig** tartott a programnak.

Megnéztük a `matr-del-9` gráfot is (4.9). Ott a feladatot elsőre nem sikerült megoldania a programunknak 7 nap alatt, ezért újraindítottuk, hogy az utolsó feladatot is megoldja. Ennek a feladatnak a megoldási ideje 564055 másodperc volt. Majd átrendeztük a checkpoint fájlt, az előbbieik szerint, tehát ezt a nagyon nehéz feladatot tettük a megoldandó feladatok végére és kitöröltük a megoldott feladatok közül. Az eredmény ebben az esetben is óriási gyorsulás volt, ugyanis így már csak **2923.45 másodpercbe** került megoldani a feladatot.

Tehát a feltételezésünk, miszerint először a könnyű feladatokat kell megoldani, majd csak utána a nehezeket, helyes volt. A módszerünk viszont nem a legjobb, nem találtuk meg a módját a nehéz feladatok megállapításának. Erről még az 5. fejezetben lesz szó.

4.4. Mérések összefoglalása

A programunk tesztelése, illetve a teszteredmények kiértékelése során arra jutottunk, hogy a kisebb sűrűségű gráfok esetében a párhuzamos program használata nem célszerű. Ezek könnyebb feladatoknak számítanak, amit a soros program is gond nélkül, belátható időn belül képes elvégezni. A párhuzamos program viszont hajlamos lemaradni, elveszni a sok kis feladat között. Sok olyan feladat keletkezik, ami alig könnyebb a teljes feladatnál, így a párhuzamos program sokkal több kevésbé könnyű feladattal foglalkozik, mint a soros.

A nagyobb sűrűségű gráfok esetén már annál inkább ajánlott a párhuzamos program használata.

Ekkor ugyanis a nagy sűrűségű gráf egy nehéz feladatot jelent Lilla számára. A prekondicionálás során feljegyzett élek által keletkezett feladatok sokkal kisebbek és egyszerűbbek, mint a teljes gráf. Ezért a párhuzamos program gyorsabban képes végezni az ilyen kis feladatokkal, mint a soros a teljes gráffal.

Továbbá azt vettük észre, hogy bár nem hozott mindenhol szemmel látható eredményt a színezés változtatása (s növelése, illetve csökkentése) érdemes több fajta színezést is kipróbálni, hiszen akadtak olyan feladatok, amelyeknek a futásideje egy jól megválasztott színezéssel majdnem a tizedére csökkent (4.8 táblázat, **monoton-9** $s=4$ és $s=5$ esetben).

5. fejezet

Továbbfejlesztési lehetőségek

TDK munkánk során végzett szisztematikus mérések, és azok eredményei rámutattak azokra a lehetőségekre amelyekkel továbbfejleszthető a programunk. A mérések és azok által készített diagramok segítségével sikerült jobban megértenünk a probléma mibenlétét. A probléma szerkezetéből és megértéséből adódóan a következő továbbfejlesztési lehetőségeket tártuk fel:

- Az általunk írt s -klick mentes színező egy elég egyszerű, gyors, kevésbé hatékony színező. Vannak ennél az algoritmusnál sokkal jobb, hatékonyabb, és gyorsabb színező algoritmusok. Ezek könnyedén beágyazhatóak a programunkba, amely egy javítási lehetőség. Ha jobb színezést használunk a prekondicionálás második lépésében, elképzelhető, hogy kevesebb eliminálandó, zavaró él keletkezik. Ezáltal kevesebb feladata lesz a párhuzamos programnak.
- Az újraosztályozás folyamatán is lehet még javítani. Jelenleg úgy osztjuk szét a q darab kvázi színosztályba az $r - q$ fennmaradó színosztály csúcsait, hogy egyesével megnézzük a csúcsokat, hogy melyik színosztályban alkot kevesebb zavaró élt. Ehelyett itt is megszámlolhatnánk, hogy a csúcsok melyik színosztályban alkotnak kevesebb s -klikket, ahogy azt az élek feljegyzése során tesszük, de ezzel jelentősen lassulna a prekondicionálás folyamata.
- Feladatok elosztása nehézségük szerint. Az általunk választott módszer nem bizonyult túl hatásosnak (3.3). Ha nem az élek feljegyzése során próbáljuk meg megállapítani a feladatok nehézségét, hanem a párhuzamos programmal, akkor akár jobb eredményeket is elérhetünk. Egy lehetséges módszer erre, hogy veszünk a processzorok számának megfelelő mennyiségű mintát a megoldandó feladatok közül. Ezeknek a minta feladatoknak megoldjuk egy részét, ez legyen a feladatok 80%-a, közben mérjük az időt, a többi feladatot – amit nem sikerült megoldani ez alatt az idő alatt – pedig félretesszük. Ezzel kaptunk egy átlagidőt amit alapul vehetünk a további feladatok megoldása során. Ezután a szolgák nekiállnak megoldani az összes feladatot. Amelyiket sikerül megoldani az imént kapott átlagidőn belül, azt

megoldják, amelyiket nem azt félre teszik. Ezzel a folyamattal a nehéz feladatok mindig hátrébb szorulnak. Ez már önmagában is hatékony lehet a 4.3.2 fejezetben kapott eredményt alapul véve. Azonban ha tovább szeretnénk könnyíteni a nehéz feladatokon, akkor azokból is előállíthatunk egy újabb élhalmazt, ahogyan azt az egész gráffal tettük.

A lista még közel sem teljes, de már ez is mutatja, hogy milyen sok munkára lesz még szükség, amit MSc tanulmányaink során szeretnénk elvégezni.

Összefoglalás

Dolgozatunk írása során elsőként a kombinatorikus optimalizálás területén a maximum klikk problémát jártuk körül, amelyhez elengedhetetlen volt utánajárnunk a kapcsolódó szakirodalomnak. Ezek közül a dolgozathoz szorosan kötődik a Szabó Sándor által javasolt párhuzamos algoritmus, amely a programunk alapját képezi. [Sza11]

Két szakdolgozat alapját jelentő párhuzamos implementáció készült el ezen algoritmus alapján [Kis13] [Str14], melyek értelmezése után újabb heurisztikák alkalmazásával implementáltuk a fent említett párhuzamos algoritmust. Az elkészült programot C++ nyelven írtuk, a párhuzamosításra pedig az MPI nyújtotta lehetőségeket választottuk. TDK munkánk során így komoly tudásra tettünk szert a szuperszámítógépek használatában. Szisztematikus méréseket végeztünk a pécsi szuperszámítógépen. Ezekkel a mérésekkel alátámasztottuk a már meglévő eredményeket. Ezek részben Kiss Annamária és Stráhl István, részben pedig témavezetőink eredményei [Sza13, Zav14]. A mérések kiértékelésekor és a probléma megértésében nagy hasznunkra váltak az általunk elkészített diagramok. Segítségükkel jóval szemléletesebb lett mind a program működése, mind a feladatok nehézségének megoszlása.

Különböző heurisztikákat alkalmaztunk, melyeket modulárisan építettünk bele programunkba, így külön-külön tudtuk ellenőrizni azok helyes működését. A modularitás miatt a program robosztus és átlátható, valamint bővíthető, illetve továbbfejleszthető lett. Utóbbi két tulajdonság különösen fontos, hiszen így ezen heurisztikák bármelyike kicserélhető a programunkban, ami nagyobb megbízhatóságot, és széles tesztelési lehetőségeket nyújt. A szisztematikus mérési eredmények rávilágítanak arra, hogy milyen heurisztikák helyesek, és utat mutat, hogy milyen további fejlesztési lehetőségek vannak, melyekről az 5. fejezetben írtunk.

Irodalomjegyzék

- [And97] Béla Andrásfai. *Gráfelmélet*. Polygon Kiadó, 1997.
- [boo] boost. ...one of the most highly regarded and expertly designed c++ library projects in the world. <http://www.boost.org/>.
- [Bre79] Daniel Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 1979.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.
- [CP90] Randy Carraghan and Panos M. Pardalos. An exact parallel algorithm for the maximum clique problem. *Operations Research Letters*, 1990.
- [DIM] DIMACS. Center for discrete mathematics and theoretical computer science. <http://dimacs.rutgers.edu>.
- [Har69] Frank Harary. *Graph Theory*. Addison-Wesley Publishing Company, 1969.
- [Har85] Frank Harary. Conditional colorability in graphs. *First Colorado Symp. Graph Theory, John Wiley & Sons*, 1985.
- [Kis13] Annamária Kiss. Párhuzamos program maximum klikk keresésre kvázi színezéssel. Szakdolgozat, PTE, 2013.
- [KRS06] Gyula Katona, András Recski, and Csaba Szabó. *A Számítástudomány alapjai*. TypoTEX, 2006.
- [NIIa] NIIF. Nemzeti információs infrastruktúra fejlesztési intézet. <http://www.niif.hu/>.
- [NIIb] NIIF. Niif szuperszámítógépek használata. https://wiki.niif.hu/index.php?title=NIIF_szupersz%C3%A1m%C3%ADt%C3%B3g%C3%A9pek_haszn%C3%A1llata.
- [Slo] Neil Sloane. Challenge problems: Independent sets in graphs. <http://neilsloane.com/doc/graphs.html>.

- [SS68] Horst Sachs and Martin Schauble. Konstruktion von graphen mit gewissen farbungseigenschaften. *Betrage zur Graphentheorie*, 1968.
- [Str14] Istvan Strahl. Parhuzamos programok maximum klikk keresesre. Szakdolgozat, PTE, 2014.
- [SZ12] Sandor Szabo and Bogdan Zavalnij. Greedy algorithms for triangle free coloring. *AKCE International Journal of Graphs and Combinatorics*, pages 169–186, 2012.
- [Sza11] Sandor Szabo. Parallel algorithms for finding cliques in a graph. *Journal of Physics: Conference Series* *Email*, 2011.
- [Sza13] Sandor Szabo. Monotonic matrices and clique search in graphs. *Annales Univ. Sci. Budapest., Sect. Comp.*, pages 307–322, 2013.
- [Sza14] Sandor Szabo. *Temakorok a kombinatorikus optimalizalas teruleterol*. TypoTeX, 2014.
- [VZ14] Geza Varady and Bogdan Zavalnij. *Bevezetes az MPI programozasba*. TypoTeX, 2014.
- [Wei] Eric W. Weisstein. Monotonic matrix. <http://mathworld.wolfram.com/MonotonicMatrix.html>.
- [Zav14] Bogdan Zavalnij. The Las Vegas method of parallelization. *Information Society 2014 - IS, A; Intelligent Systems*:105–108, 2014.