

TDK-dolgozat

Vancsics Béla

**Függvény-teszteset mátrixok klaszterező
eljárásainak összehasonlítása,
tesztpriorizálás**

Készítette:

Vancsics Béla

gazdaságinformatikus MSc II. évfolyam

Témavezető:

Dr. Gergely Tamás

egyetemi adjunktus

Szegedi Tudományegyetem

Informatikai Tanszékcsoport

Szoftverfejlesztés Tanszék

Tartalomjegyzék

BEVEZETÉS	4
1. Néhány gondolat a szoftvertesztelésről.....	5
1.1 Alapelvek.....	5
1.2 Tesztelési technikák.....	6
1.3: A tesztelés folyamata.....	6
1.4: Teszthalmazok létrehozása.....	7
2. A coverage mátrixról és a SoDA library-ről.....	9
2.1. Mátrixokról.....	9
2.2 Mátrixok típusai.....	10
3 A klaszterezésről.....	13
3.1 Hamming távolságon alapuló eljárás.....	15
3.1.1 Struktúrával történő klaszterezés	15
3.1.2 Hamming távolságon alapuló particionáló eljárás.....	16
3.2 Halmazok közötti kapcsolaton alapuló eljárások	17
3.2.1 Jaccard hasonlóságon alapuló eljárások	18
3.2.2 Ochiai együtthatón alapuló klaszterezés.....	19
3.2.3 Dice hasonlóságra épülő klaszterezés.....	19
3.3 Hash függvényen alapuló eljárások.....	20
4. Eredmények és azok megjelenítései.....	22
4.1 Vizualizáció	22
4.2. Metrikák.....	23
4.2.1 Metrika 1.....	24
4.2.2 Metrika 2.....	26
4.2.3 Metrika 3.....	27
5 Mérések és az eredmények értékelése.....	29
Irodalomjegyzék.....	35

BEVEZETÉS

Dolgozatomban szoftverfejlesztés egy igen fontos részét, a szoftvertesztelést megkönnyítő, hatékonyságát növelő problémakörörről írok. Ez pedig a tesztesetek klaszterezése a függvény-teszteset mátrixok segítségével.

Noha már sok és hatékony algoritmus ismert a klaszterezésre, ez egy még viszonylag kevésbé elterjedt és bevett gyakorlat a tesztelés és a hibakeresés felgyorsítására. Pedig a benne rejlő lehetőségeket kihasználva, a feladatot és a problémát átgondolva és „újraértelmezve” a teszteset-szelekció és a teszteset-priorizálás hatékony eszköz lehet mind a fejlesztők, mind a tesztelők kezében. Ezek segítségével kiszűrhetővé válnak a redundáns tesztesetek, valamint a meghatározott fontossági sorrend segítségével kiválaszthatóak a hiba szempontjából kritikus tesztek, azok ahol a fejlesztés folyamatát követve a legnagyobb valószínűséggel fognak a hibák előfordulni. Munkámban az utóbbi szempontra fogom helyezni a hangsúlyt. Bemutatok több, általam implementált, különböző megközelítést használó algoritmust, valamint értékelem az ezek segítségével kapott eredményeket, emellett mérőszámokat adok magának a mátrixnak a klaszterezhetőségére.

A munkám alapvetően gondolatébresztő céllal készült, amely később vizsgálatok, ötletek és alkalmazások alapjául szolgálhat.

1. Néhány gondolat a szoftvertesztelésről

A szoftverfejlesztés folyamatának egy fontos része a tesztelés. Ennek gyors és hatékony elvégzése nagyban megkönnyíti a fejlesztést és segíti a fejlesztők munkáját. Sokan elbagatellizálják a tesztelés megtervezését és kivitelezését, de komoly szakmai felkészültséget igényel. Nem triviális a hatékony és a lehető legteljesebb körű tesztelés kivitelezése. Ezért is alakult ki az informatikán belül egy direkt erre szakosodott terület, amely a teszteléssel, annak minőségével ill. buktatóival foglalkozik. Az utóbbi időben egyre több szakirodalom foglalkozik a témával, egyre több elméletet dolgoznak ki erre a problémakörre. Az első részben erre térnek ki pár gondolat erejéig, mintegy felvezetve a dolgozatom témáját és annak lényegét.

Ha definiálni szeretnénk mi is a szoftvertesztelés akkor talán úgy foglalthatnánk össze, hogy a rendszer vagy a program minőségének az egyes munkatermékek ellenőrzésével és vizsgálatával történő felmérése. A hagyományos megközelítés szerint a tesztelés célja az, hogy a fejlesztés során létrejövő hibákat minél korábban felfedezze, és ezzel csökkentse azok kijavításának költségeit. A tesztelők hasznos információkkal látják el a fejlesztőgárdát, rávilágítanak a szoftverek gyenge pontjaira, hibáira és ezáltal javítják a szoftverminőséget és növelik a megbízhatóságot.

1.1 Alapelvek

A szakemberek több alapelvet megfogalmaztak a hatékony tesztelés érdekében. Nincs általánosan elfogadott tézis ezekkel kapcsolatban, létezik több általános tesztelési alapelv is. Ilyenek például [1] :

1. A teszteléssel a hibákat derítjük fel, de ha nem mutatunk hibát a (szoftver)termékben az nem jelenti azt, hogy ne is lenne benne...
2. Nincs olyan, hogy minden lehetséges esetre leteszteljük a terméket. A lehetséges bemeneteknek annyi kombinációja lehet (főleg egy komplex rendszernél), hogy nincs esély a (teljesen) kimerítő tesztelésre.

3. Érdemes minél korábban elkezdni a tesztelés folyamatát, ezáltal ugyanis olcsóbbá válik az esetleges hibák javítása.
4. Érdemes a tesztek valamely alapján csoportosítani és fontossági sorrendet felállítani közöttük, ezáltal növelni a hatékonyságot. A dolgozatomban is erre, vagyis a csoportok létrehozására tesztek kísérleteket és értékelem azok eredményeit.
5. Ügyeljünk a tesztesetek „túltanulásának” elkerülésére, vagyis rotáljuk ill. bővítsük a tesztthalmazunkat.

1.2 Tesztelési technikák

A tesztelési technikákat többféleképpen csoportosíthatjuk, aszerint például, hogy a teszteléshez szükséges-e a program futtatása; ez esetben dinamikus technikákról beszélünk.

Alapvetően kétféle nagy technika létezik a dinamikus tesztelésre. Ezek más és más megközelítés alapján vizsgálják meg a szoftvert, más típusú hibák kiszűrésére szolgálnak. Ezek a black-box, ill. a white-box technikák.

Black-box (feketedobozos) tesztelés specifikáció alapú, vagyis a lényege, hogy a kész programunkra fekete dobozként tekintünk, nem látunk bele a belső működésébe, abba, hogy mit és hogyan csinál. Pusztán csak az eredményt kapjuk meg és az alapján kell értékítéletet mondanunk a programról.

A white-box (fehérdobozos) tesztelés forráskód alapú megközelítést alkalmaz. Ilyenkor a tesztelő tisztában van a rendszer belső logikájával, a komponensek összefüggéseivel és azok kapcsolataival. Ebben az esetben nem csak a hiba tényét és annak egy tag helyét tudjuk megállapítani, hanem egész konkrétan lokalizálni tudjuk azt a forráskódban.

Lényeges fogalom a kód lefedettsége, ami megadja, hogy a mekkora részét tudjuk tesztelni a szoftvernek a meglévő tesztesetekkel. Minél nagyobb ez a rész annál teljesebb tesztelést tudunk végezni.

1.3: A tesztelés folyamata

Fontos, hogy jól átgondolva, tervszerűen hajtsuk végre a tesztelést, azáltal is növelve a

konzisztenciát a folyamatban. Ehhez az alábbi lépések betartása elengedhetetlen [2]:

1. tesztterv készítése
2. tesztesetek tervezése
3. felkészülés a végrehajtásra
4. tesztek végrehajtása
5. kilépési feltételek vizsgálata
6. eredmények értékelése
7. jelentéskészítés

Ezáltal az eredmények ellenőrizhetővé, reprodukálhatóvá és következetessé válnak, ami elengedhetetlen a tesztelés hatékonysága és a fejlesztőknek nyújtott minél teljesebb és hasznosabb információ szempontjából.

1.4: Teszthalmazok létrehozása

Az alapelveknél már említettem, hogy ajánlatos valami alapján csoportosítani a teszteseteket. Erre is több lehetséges elgondolás van. Ilyen például amikor a hasonló funkciókat rakjuk egy halmazba vagy ha valami szempont – pl.: biztonság- szerint csoportosítjuk őket.

Tegyük fel, hogy van egy termék, amiben valamilyen hibajavítás történt. Könnyen elképzelhető, hogy az újonnan be- vagy kikerülő kód miatt nem lesz konzisztens az adott komponens a programunk többi részével. Ha a teszteseteket „véletlen” sorrend szerint futtatnánk le, akkor könnyen lehet, hogy sokáig kell várnunk arra, hogy kiderüljön, működik-e a javított kód vagy sem. De ha a prioritási sorba állítanánk a teszteseteket és a megfelelő elemek kerülnének az elejére akkor sokkal hamarabb megkapnánk az eredményt. Ennek egy lehetséges „kivitelezése” ha azokat a teszteseteket vesszük előre, amik a módosított részt ellenőrzik. Így az elsők között derül ki, hogy jó volt-e a javítás vagy sem. Fontos megjegyezni, hogy ezzel nem lehet megkerülni az összes teszt végrehajtását -mivel lehet, hogy a módosítással egy új bug (hiba) keletkezett, de az természeténél fogva továbbgyűrűzik és csak egy másik szinten fog megjelenni. Tehát ez nem elsősorban a tesztesetek kihagyására, hanem a hibakeresés felgyorsítására és a redundáns tesztesetek kiszűrésére való.

Dolgozatom lényegében az ilyen csoportok létrehozásával foglalkozik, amelynek alapja a kód lefedettsége alapján előálló coverage (függvény-teszteset) mátrix. Az általam implementált algoritmusok és metrikák segítségével próbálom megadni ezeket a függvény-, valamint teszteset-halmazokat.

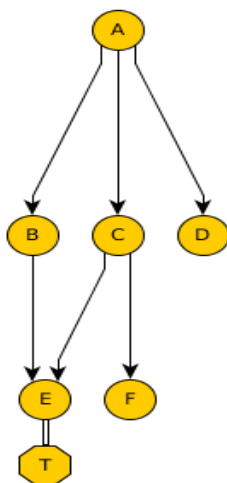
2. A coverage mátrixról és a SoDA library -ról

Az SZTE Szoftverfejlesztés Tanszék által fejlesztett SoDA library által előállított bináris fájlok a coverage mátrixok. A library megfelelő környezet mellett előállítja az adott program lefedettségi mátrixát, vagyis azt, hogy melyik függvény követett vagy közvetlen hívásával tesztelhető egy adott teszteset. A fájl alapvetően három, számomra fontos adatot tartalmaz.

1. függvények nevei
2. tesztesetek nevei
3. maga a mátrix

2.1. Mátrixokról

Az első kettő nem szorul különösebb magyarázatra. A mátrix megadja, hogy egy adott függvény mely teszteseteket fed le -közvetve vagy közvetlenül. A horizontális dimenzióban vannak a tesztesetek, a vertikálisban pedig a függvényeket találjuk. A mátrix értékei vagy 0-t vagy 1-t vehetnek fel.



(2.1. ábra: hívási fa)

A fenti kis példa esetében -ahol az *A*, a *B*, a *C*, a *D* az *E* és az *F* függvényeket jelölnék, a *T* pedig tesztesetet- a *T* tesztesethez tartozó sora a lefedettségi mátrixnak az alábbi módon fog kinézni:

	A	B	C	D	E	F
T	1	1	1	0	1	0

(2.2. ábra: példa a coverage mátrix egy sorára)

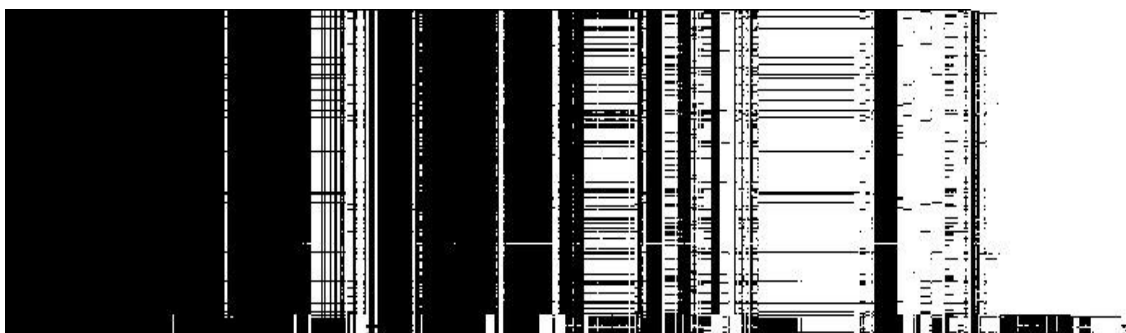
A library a mátrix előállításához szükséges „programokon” kívül több egyéb függvényt is tartalmaz, amelyeket én is felhasználtam a programom megírása közben. Ilyen például a bináris fájl beolvasására szolgáló metódus, az adott sorhoz tartozó tesztesetek neveinek lekérdezéséhez használatos függvény stb... Így nekem csak az algoritmusok implementálásával kellett foglalkoznom, a többi „eszköz” már a rendelkezésemre állt.

A programban meg vannak valósítva olyan függvények is amelyek segítségével elő lehet állítani a kívánt mátrixot, és abban a megfelelő értékeket beállítani. Ezt a méréseknél, pontosabban a bementi fájlok előállításánál ki is fogom használni.

2.2 Mátrixok típusai

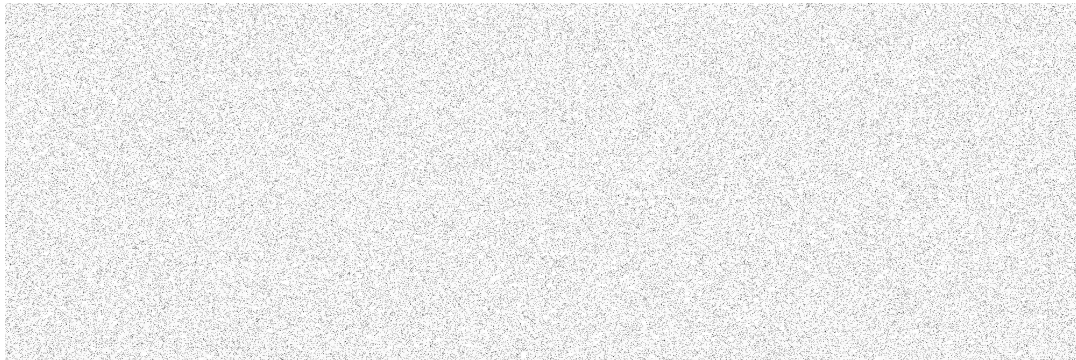
Az algoritmusok bemeneteként alapvetően háromféle mátrixot különböztetek meg, ezáltal is biztosítva a szélesebb spektrumú méréseket és átfogóbb eredmények előállítását.

1. valós, opensource rendszeren mért mátrix (2.3. ábra) : a library segítségével előállított, a szoftverhez tartozó függvényeket és teszteseteket tartalmazó mátrix keletkezik



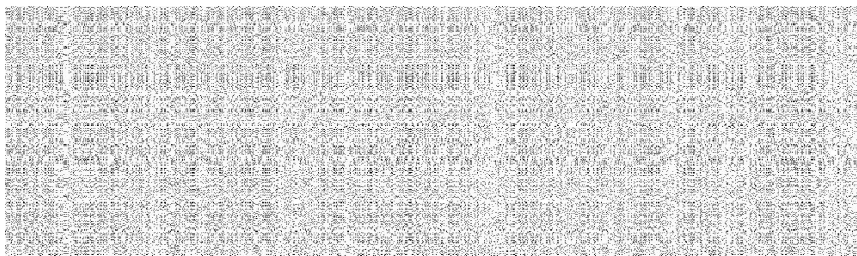
(2.3. ábra: példa valós eredményre)

2. általam generált random mátrix (2.4. ábra) : paraméterben megadott méretű mátrixot állít elő, ahol a függvények neve fX lesz -ahol az X a sor sorszámát mutatja és 0-tól a paraméterben megadott értékig terjed, a tesztesetek neve pedig tY lesz- ahol az Y az oszlop sorszámát adja 0-tól indulva. Emellett szintén paraméterben kell megadni, hogy mekkora százalékkal legyen 1 érték a pozíciókban és random generátort használva állítom be az egyes mezők értékeit.



(2.4. ábra: általam generált random kép)

3. random klaszterezett generált mátrix (2.5. ábra) : ez annyiban különbözik a „sima” random mátrixtól, hogy itt meg kell adnunk, hogy hány klasztert szeretnénk létrehozni és azt, hogy a klaszteren belül, valamint azon kívül mekkora százalékos valószínűséggel legyen 1 érték a pozícióban. Ezek a random mátrixnál jobban klaszterezhető mátrixot eredményeznek, ugyanakkor eléggé meg vannak keverve ahhoz a sorok és az oszlopok, hogy a rendezés ne legyen triviális (vagyis szabad szemmel látható).



(2.5. ábra: speciálisan előállított random kép)

3 A klaszterezésről

A klaszterezés lényege kicsit pongyolán megfogalmazva az, hogy olyan csoportokat alakítsunk ki egy halmazból, ahol az egy csoportba tartozó elemek -a vizsgált szempont alapján- hasonlóbbak lesznek egymáshoz, jobban hasonlítanak egymásra, mint a csoporton kívüli elemekhez.

Egy jó klaszterező algoritmusnak három tulajdonsága van [3] [4]:

1. skála invariancia (vagyis invariáns a klaszterek közötti eltérést, távolságot megadó függvény pozitív konstanssal való szorzására)
2. konzisztencia (vagyis invariáns a klaszteren belüli távolságok csökkentésére, illetve a klaszterköziek növelésére)
3. gazdagság (vagyis minden felosztás előállítható alkalmas távolságfüggvényt választva)

Erre vonatkozik Kleinberg lehetetlenségi tétele, amely kimondja, hogy nem adható olyan eljárás, amely a fenti három tulajdonsággal egyszerre rendelkezne [5].

Magukat az algoritmusokat többféleképpen csoportosíthatjuk. Az egyik ilyen a „stílus” szerinti megkülönböztetés. A legelterjedtebb ilyen megközelítések:

- hierarchikus módszerek
- particionáló módszerek
- sűrűség- ill. rács-alapú
- spektrál módszerek

Dolgozatomban csak az első kettővel fogok a továbbiakban foglalkozni.

Hierarchikus klaszterezés esetén a pontok klaszterstruktúráját a klaszterek egymásba ágyazott rendszereként adjuk meg. Két fő típusa létezik: a lentől felfelé haladó „építő eljárások”, illetve a fentről lefelé haladó „lenyeső eljárások”. Én csak a lentől felfelé haladó eljárásokkal foglalkoztam. Ezeknél kezdetben minden elem önálló klasztert alkot. Amíg a megállási

feltétel nem teljesül addig vontjuk össze a két, összevonásra leginkább alkalmasnak ítélt klasztert (képezzük a két klaszter unióját) [6]. A megálláskor „megmaradt” csoportok fogják majd a végső klasztereket alkotni. Sok szempont, elgondolás szerint ki lehet választani, hogy milyen tényező(k) alapján határozzuk meg azt, hogy mikor mondjuk összevonás szempontjából legalkalmasabbnak a két klasztert (pl.: Hamming távolság). Megállási feltétel megadásának is több lehetséges megoldása van. Pl.: klaszterek száma.

Particionáló klaszterezéskor a halmaz optimális felosztását keressük bizonyos célfüggvény optimalizálása révén. Ilyen lehet például a klaszterközpépponttól vett távolság minimalizálása. Az algoritmus végrehajtása során az elemeket közvetlenül rendeljük az általunk megadott számú klaszter egyikébe. Az egyik legelterjedtebb ebbe a csoportba tartozó algoritmus a K-means alg., amelyre a későbbiekben részletesebben is ki fogok térni.

A bemeneti mátrix klaszterezésekor mind a sorok, mind az oszlopok kapnak egy-egy osztálycímkét. Az alapkoncepcióm az volt, hogy minél többféle megközelítést implementáljak és ezeket aztán a definiált metrikák szerint összehasonlítsam.

Először a sorokat klasztereztem valamelyik módszer szerint, majd ezután következtek az oszlopok. Itt ismét kétféle megoldást próbáltam ki. Az egyik szerint ugyanazt az algoritmust hajtottam végre, mint a sorokra, a másik szerint pedig az oszlopban leggyakrabban szereplő klasztercímkét adom az oszlop indexének.

	2	1	2	2	3	3	2	3	2	3	3	1	1	3	3	3	2	3	3	3
1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
2	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	1	1	1
2	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0
2	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1	1
2	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0
3	0	0	0	0	1	1	0	1	1	1	1	0	0	1	1	1	0	1	1	1
1	1	1	1	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0

(2.6. ábra: példa a sorcímkék alapján történő oszlopklaszterezésre)

3.1 Hamming távolságon alapuló eljárás

Az egyik népszerű megközelítés amikor a Hamming távolságot vizsgáljuk és ez alapján klaszterezzük az elemeket. Ilyenkor két vektor -amik az elemeket reprezentálják- azonos pozícióban vett eltérését határozzuk meg.

"toned" and "roses" is 3.
1011101 and 1001001 is 2.
2173896 and 2233796 is 3.

(3.1. ábra: példa Hamming távolságra [7])

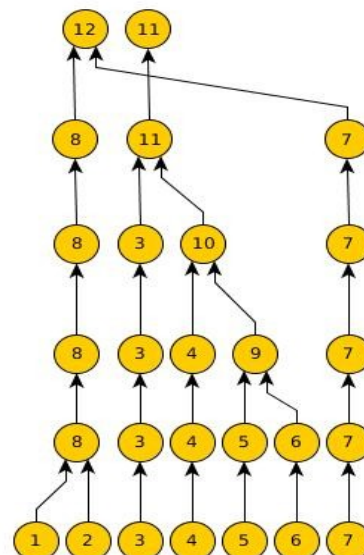
Több programomban is közvetve vagy közvetlenül de ez képezi az összehasonlítás alapján.

3.1.1 Struktúrával történő klaszterezés

Ez az algoritmus a hierarchikus megközelítést használja, vagyis kezdetben minden sor egy struktúraelem lesz.

Maga a struktúra négy adatot fog tárolni:

- a struktúra azonosítója
- melyik másik struktúra volt az apja ill. az anyja (kezdetben -1 az érték)
- klaszterelemek, azaz mely sorok tartoznak abba a klaszterbe



(3.2. ábra: példa struktúrák összevonására)

Magának a programnak a pszeudokódja a következő:

- 1: mátrix beolvasása és a kiindulási struktúrák létrehozása
- 2: minden sorhoz meghatározom, hogy melyik másik sor van hozzá a legközelebb és az mekkora (Hamming) távolságra van
- 3: for(amíg el nem érjük a kellő klaszterszámot){
- 4: összevonandó struktúrák megadása (vagyis a legkisebb Hamming távolság megkeresés)
- 5: a két elemből új elem képzése -az új struktúra tulajdonságainak megfelelő beállításával
- }
- 6: klasztercímkék hozzárendelése a struktúraelemekhez

A ciklust a paraméterben megadott klaszterszámig futtatva a végeredménynek megkapjuk (a klaszterelemek halmazban), hogy mely sorok lettek egy klaszterbe sorolva.

Ugyanezt az algoritmust az oszlopokra is lefuttathatjuk, ezáltal megkapva azok címkeit vagy pedig a fentebb vázolt módszer alapján a sorindexekből számíthatjuk ki.

A program futtatásakor kevés paramétert kell megadni. Elég csak bináris fájlt, az előállítani kívánt klaszterszámot, valamint a kimeneti fájl nevét (aminek majd az eredmény fekete-fehér képen történő megjelenítésében lesz fontos szerepe).

3.1.2 Hamming távolságon alapuló particionáló eljárás

Ez egy viszonylag egyszerű elven működő, ugyanakkor sok paramétert használó és ezáltal sok változótól függő algoritmus.

Miután beolvassuk a mátrixot megnézzük, hogy az egyes sorok, illetve oszlopok mennyi 1-t tartalmaznak. Ha ez az érték nagyobb mint a paraméterben megadott küszöb, akkor ezek az elemek a 0. címkét kapják és a későbbi vizsgálatban már nem játszanak szerepet. Ennek, mármint a nullklaszter definiálásának majd a metrikáknál lesz fontos szerepe.

Ezután egyesével megnézem az összes – még nem klaszterezett – sorra, hogy az mekkora Hamming távolságra van a többi, nem klaszterezett sortól. Ha az kisebb, mint az

argumentumban százalékosan megadott érték – amit sorok számára vetíttek, akkor a két sor címkéje azonos lesz.

Ebben az esetben nem tudjuk előre, hogy hány klaszter lesz, ugyanis az nagyban függ a megadott paraméterektől és csak a futási időben, a ciklus végén fog kiderülni a klaszterszám. Ezáltal a futási idő is a beállított argumentumoktól függ. Pl.: ha mindegyik sor nagyon közel van egymáshoz akkor elég csak egyszer bejárni a mátrixot, de ha olyan a mátrix szerkezete vagy olyan kis toleranciaszintet adunk meg akkor akár minden sor külön klaszterbe kerülhet. Itt több paramétert kell megadnunk, mint a struktúrával történő klaszterezés esetében. Ezek egy része magára a klaszterezésre vonatkozik (ezeket feljebb említettem), mások viszont a mátrix vizualizációját szolgálják (ezekről később lesz szó).

3.2 Halmazok közötti kapcsolaton alapuló eljárások

Ennél a megközelítésnél a Hamming távolság helyett a halmazokra és azok metszetére valamint uniójára építünk, ezt fogjuk felhasználni a közös klaszterbe tartozó elemek meghatározására.

A tesztesetek esetében azok a függvények kerülnek majd az adott halmazokba, ahol 1 érték szerepel a mátrix megfelelő sorában. Függvények esetén ehhez hasonlóan választjuk ki a hozzájuk tartozó a teszteseteket, de itt az oszlopokat vizsgáljuk.

Három különböző megoldást készítettem el. Ezeknél a k-means algoritmust használom a klaszterezésre -amelyhez a GNU Octave [8] nevű programot használtam. Ennek az a lényege, hogy minden elemet ahhoz a klaszterhez fogunk sorolni, amely középpontjához a legközelebb van. Az Octave *k-means* függvényének segítségével a kiindulási mátrixból előállított új mátrixokat az általam megadott számú klaszterek valamelyikébe sorolja és megadja azok középpontjait.

Az algoritmus lépései a következők (MacQueen, 1967) [9] :

1. Kiválasztja a klaszterek számát (k).
2. Véletlenszerűen létrehoz k számú klasztert, és meghatározza minden klaszter közepét, vagy azonnal létrehoz k véletlenszerű klaszterközéppontot.

3. Minden egyes pontot abba a klaszterbe sorol, amelynek középpontjához a legközelebb helyezkedik el.
4. Kiszámolja az új klaszterközéppontokat.
5. Addig ismétli az előző két lépést (iterál), amíg valamilyen konvergencia kritérium nem teljesül (általában az, hogy a besorolás nem változik).

Ez az algoritmus nagyon elterjedt, hála az egyszerűségének és a sebességének. Hátránya viszont, hogy a véletlen kezdő középpontok miatt egymás utáni -azonos paraméterű-futtatásra különböző eredményeket ad, illetve előre meg kell adnunk a kívánt klaszterszámot (noha bizonyos problémák esetében ez akár előnye is lehet). Emellett minimálisra csökkenti a klasztereken belüli varianciát, de nem eredményezi összességében a legkisebb varianciát.

3.2.1 Jaccard hasonlóságon alapuló eljárások

Az első ilyen típusú megoldás a Jaccard hasonlóságra épít. A képlet egy 0 és 1 közötti számot ad vissza, amely megadja a két elem viszonyát.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

(3.3. ábra: a Jaccard hasonlóság képlete [10])

Minden sor- és oszloppárra elvégzem a fenti képlet kiszámolását, amely megadja, hogy az egyes elemek hogyan viszonyulnak egymáshoz. Ennek az eredménye egy 0 (vagyis nincs közös elemük) és 1 (vagyis megegyezik a két elem) közötti érték lesz.

A bemeneti $n \times m$ -es mátrix eredménye egy $n \times n$ -es és egy $m \times m$ -es mátrix lesz, benne a fent említett értékekkel.

Ezután az Octave *k-means* algoritmusával klaszterezem a két mátrixot és az eredményét – amely egy-egy oszlopvektor, benne a klaszterindexekkel – kimentem külön egy-egy fájlba.

Ha ezzel megvagyunk akkor az általam implementált, az eredmény vizualizációra alkalmas programot meghívva és a megfelelő paramétereket megadva lehetőségünk nyílik az

eredményeket megjeleníteni. A korábban említett kétféle oszlopcímkezés egyaránt elvégezhető, ehhez elég csak a sorokra számított Jaccard hasonlóságokat tartalmazó fájlban végrehajtani az algoritmust, majd egy erre alkalmas „megjelenítőprogramot” elindítani.

3.2.2 Ochiai együtthatón alapuló klaszterezés

Az Ochiai együttható a koszinusz hasonlóságnak egy speciális esete. Ez is a két elempár és azokból létrehozott halmazokra, azok viszonyára mond egy 0 és 1 közötti számot.

A képlete:

$$K = \frac{n(A \cap B)}{\sqrt{n(A) \times n(B)}}$$

(3.4. ábra: Ochiai együttható képlete [11])

Ebben az esetben is hasonlóan fog előállni az eredmény, mint a Jaccard hasonlóságra épülő megoldásnál, ugyanazon lépéseket kell végrehajtani a klaszterezéshez.

3.2.3 Dice hasonlóságra épülő klaszterezés

Egy másik, hasonló elven működő eljárás a Dice hasonlóságot fogja felhasználni a klaszterezéshez. Ez nagyon hasonlít a fentebb említett két algoritmushoz, de itt egy másik képletet használunk az elemek viszonyának leírásához.

$$QS = \frac{2C}{A+B} = \frac{2|A \cap B|}{|A| + |B|}$$

(3.5. ábra: Dice hasonlóság)

Ebben az esetben is a korábban alkalmazott műveleteket kell végrehajtani és itt is elkészült a kétféle megközelítése az algoritmusnak.

3.3 Hash függvényen alapuló eljárások

Ez egy egészen újfajta és a klaszterezésben viszonylag szokatlan megközelítése a klaszterezésnek. Ennél azzal a feltételezéssel élünk, hogy a hasonló emelek hasonló értékeket adnak a hash függvényekre. Ehhez azonban kellően sok függvény kell.

A függvények $a * x + b \bmod n$ alakúak. Ezekből paraméterben megadott darabot generálunk, illetve az a és b értékeket is úgy állítjuk elő, hogy az jól tudja „szórni” az elemeket.

Ezután megnézzük, hogy a megfelelő értékeket behelyettesítve -vagyis azokat a sor- ill. oszlopindexeket, sor- ill. oszlopsorszámokat ahol 1 érték szerepel- milyen eredményeket kapunk. Ezen módszer esetében egy felső korlátot tudunk adni a klaszterszámmra, ami a modulus-1 érték lesz.

Lássunk rá egy egyszerű példát:

sorindex																			
0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
1	1	1	1	1	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0
2	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	1	1
3	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0
4	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	1	1	0	1	0	1	1	0	0	1	1	1	0	0	1
6	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0
7	0	0	0	0	1	1	0	1	1	1	1	0	0	1	1	1	0	1	1
8	1	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
9	0	0	0	0	1	0	0	1	0	1	1	0	0	1	1	1	0	1	1

$3x + 3 \bmod 4$	$5x + 2 \bmod 4$	$3x + 7 \bmod 4$	$9x + 1 \bmod 4$	$7x + 5 \bmod 4$	$4x + 3 \bmod 4$
------------------	------------------	------------------	------------------	------------------	------------------

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$3x + 3 \bmod 4$	3	3	2	2	2	1	2	2	0	2	2	3	3	2	2	1	2	0	2	2
$5x + 2 \bmod 4$	2	2	3	3	3	0	3	3	1	3	3	2	2	3	3	0	3	1	3	3
$3x + 7 \bmod 4$	3	3	2	2	2	1	2	2	2	2	2	3	3	2	2	1	2	0	2	2
$9x + 1 \bmod 4$	1	1	2	2	2	3	2	2	0	2	2	1	1	2	2	3	2	0	2	2
$7x + 5 \bmod 4$	1	1	0	0	0	3	0	0	2	0	0	1	1	0	0	3	0	2	0	0
$4x + 3 \bmod 4$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Segéd-indexek	3	3	2	2	2	3	2	2	0	2	2	3	3	2	2	3	2	0	2	2
Végző indexek	1	1	2	2	2	1	2	2	3	2	2	1	1	2	2	1	2	3	2	2

(3.6. ábra: a kiindulási mátrix, a függvények és az oszlopklaszterezés eredmény)

Minden sor (ill. oszlop) esetén a függvényekbe – az X helyén – behelyettesítjük az elem indexét ha az adott pozícióban 1 érték szerepel majd elvégezzük a függvényt műveletet. Egy elem címkéje a hash függvények leggyakoribb eredményei közül a leggyakoribb lesz.

Az így keletkezett eredményt még kicsit át kell alakítani, hogy folytonos legyen a címkézés, vagyis ne legyen olyan, hogy két egymást követő -lexikografikusan rendezett- index között kimaradjon egész érték.

Az így előálló indexvektorokat kimentve rendelkezésre áll a későbbi feldolgozáshoz szükséges fájl, ami majd a vizualizációnál, a metrikaszámolásnál és a klasztermérek meghatározásánál fog majd fontos (nélkülözhetetlen) szerepet játszani.

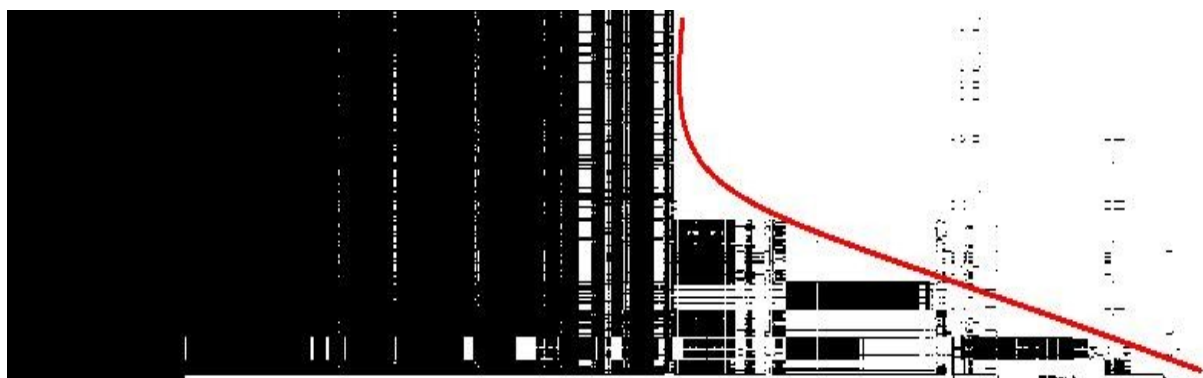
4. Eredmények és azok megjelenítései

Az algoritmusok végrehajtása után lehetőségünk van megjeleníteni az eredményt. Van olyan algoritmus, amibe „bele van építve” a rajzolás megvalósítása is – pl. Hamming távolságon alapuló eljárás esetében-, illetve egyes eljárásoknál a hozzá tartozó mátrix-rajzoló programot kell elindítani a megfelelő paraméterekkel.

4.1 Vizualizáció

A library-ban implementálva lett egy függvény, amely segítségével a mátrixból egy fekete-fehér képet tudunk készíteni, azáltal mintegy vizualizálni a klaszterezés eredményeit.

Az szeretnénk elérni, hogy a így keletkezett képben a főátló mentén sötét blokkok jöjjenek létre, azokon kívül részben pedig minél kevesebb legyen a sötét rész, vagyis az olyan pozíció, ahol 1 érték szerepel. Ha a klaszterezés eredményét rögtön kiíratnánk képpé, akkor a sötét blokkok ugyan megjelennének, de azok szanaszét helyezkednének el a mátrixban. Ezért szükség volt egy előrendezésre is, amely a gyakorlatilag a klaszterek kirajzolásának a sorrendjét hivatott meghatározni.



(4.1. ábra: Augeas mátrix klaszterezése struktúrák segítségével)

A mátrix megjelenítésére a SoDA -ban megvalósított *write* függvényt használtam. A *CArrayBitMatrix* osztály *set* metódusával beállítom a megfelelő pozíciókba a kívánt

értékeket. Magát a rendező algoritmust, vagyis hogy milyen sorrendben írjuk ki a sorokat és az oszlopokat- én implementáltam szemelőt tartva a főátlóba történő rendezést. Ennek alapja, hogy a sorindex-oszlopindex párokban mennyi 1 érték szerepel. Ez alapján kiválasztottam, hogy minden sorklaszterhez melyik oszlopklaszter tartozik. Ezek a klaszterpárokhoz tartozó sorok illetve oszlopok lesznek a főátló mentén.

Amennyiben a felhasználó szeretné a rendezett mátrix bináris kiíratására is van lehetőség, benne a tesztesetek, a függvények és magának a mátrixnak a megfelelő sorrendben való elmentésével.

4.2. Metrikák

Ha csak a képekből indulnánk ki, ha csak ez szolgálna a klaszterezés „minőségének” a megítélésére, akkor az eredmény elég gyenge lábakon állna, nagyon pontatlan és szubjektív lenne. Ezért döntöttünk úgy, hogy keresünk valamiféle metrikát arra, ami számszerűsíti az algoritmusok hatékonyságát és ezáltal megbízhatóan összemérhetővé tenné az eredményeket. Mivel nincs – pontosabban nem találtam olyan- metrikát, amely egyedül és valamiféle hátrány nélkül jellemezné a mátrixot, azért úgy döntöttünk, hogy három különböző mérőszámot definiálunk és ezek alapján végezzük az összehasonlítást.

A klaszterezés során előfordulhatnak olyan sorok, valamint olyan oszlopok – magából a valós programokon mért coverage mátrix természetéből fakadóan, hogy nagyon sok 1 érték szerepel bennük. Ha a hívási fából indulunk ki, akkor ezek a függvények tipikusan a gyökér közelében helyezkednek el és az alattuk levő csúcsok tesztelésekor ezek is meghívódnak. Erre jó példa a 2.1. ábra.

Ezek a blokkok viszont nagyon elrontják az eredményeket, ugyanis mivel a legtöbb elemnél ezekben a pozíciókban 1 érték szerepel ezért az elemek közti eltéréssel kapcsolatban semmilyen információt nem szolgáltatnak, viszont klaszterezéskor „felbomlanak” és szétszóródnak a mátrixban ezzel is rontva a metrikák hatékonyságát.

Ezért készítettem egy kapcsolót – amit futtatáskor paraméterben kell megadni – ami segítségével beállíthatjuk, hogy ezek a nullklaszterbe kerülő blokkok beleszámítódjanak-e a metrikába vagy sem.

Mikor is mondjuk, hogy egy mátrix jól klaszterezhető? Ez nem egy könnyű kérdés. Több olyan feltétel van, amelyeknek minél jobban teljesülniük kell ahhoz, hogy ki merjük jelteni, hogy jól és hatékonyan lehet klaszterekbe sorolni az elemeket.

Ilyen feltétel például:

- minden sorklaszterhez találni kell olyan oszlopklasztort ahol maximális a benne levő 1 száma
- az előbb említett maximalizálást úgy elvégezni, hogy az globális maximumot adjon, vagyis olyan kitüntetett sorklaszter-oszlopklaszter párok találása, amelyeknél nincs jobb, azaz olyan, hogy a másik lehetséges párosításban több 1 érték legyen a párokon belül, mint az előző esetben
- megfelelő klaszterszám megadása/megtalálása
- ideális klaszterméretre való bonthatóság

Ezek mind nehéz kérdések és nincs általánosan elfogadott koncepció ezekkel kapcsolatban. Általában feladat-, mátrix-, és algoritmus-specifikus, hogy melyik az az érték ami jónak számít.

A mérések elvégzésekor többször, különböző paraméterekkel kellett lefuttatnom az algoritmusokat mire egy viszonylag optimális-közeli megoldáshoz jutottam.

4.2.1 Metrika 1

Az első mérőszám amit definiáltam a jól klaszterezett és a mátrixban levő pontok arányára mond egy -1 és 1 közötti számot. Ebben az esetben csak az 1 értékű mezőket vizsgáljuk.

Akkor mondjuk, hogy egy pont jól klaszterezett, ha a pont oszlopindexe a sorindex kitüntetett klaszterpárja. Vagyis -pongyolán fogalmazva- az a jó, ha a sűrű klaszteren belül van a pont. A jól klaszterezett pontok számát csökkentjük a rosszul klaszterezettekével és kiszámoljuk, hogy ez milyen arányban van az összes pont darabszámához képest.

Ezzel az aránnyal lehet szemléltetni, hogy hogyan viszonyulnak a jól klaszterezhető pontok az összes ponthoz. Minél jobban közelít ez az érték az 1-hez annál jobb az algoritmus.

$$M1 = \frac{\text{jól klaszterezett pontok száma} - \text{rosszul klaszterezett pontok száma}}{\text{összes pontok száma}}$$

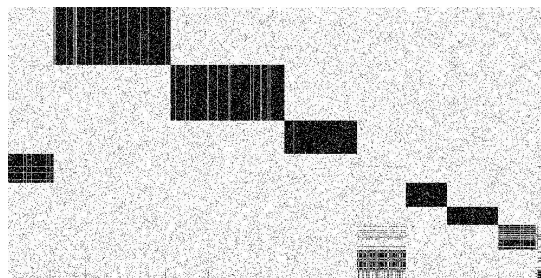
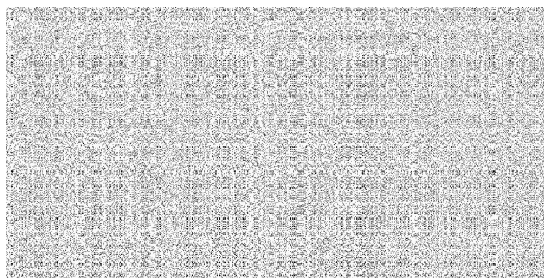
(4.2. ábra: az első metrika képlete)

		Eljárások					
		1	3	5	2	4	6
Tesztesetek	5	1	1	1	0	0	0
	2	0	1	1	0	0	0
	8	0	1	1	0	1	0
	3	0	1	0	1	1	0
	7	0	0	0	1	1	0
	1	0	0	0	1	0	0
	9	0	0	0	0	1	0
	4	0	0	0	0	1	1
	6	0	0	0	0	0	1
	10	1	0	0	0	0	1

(4.3. ábra: példa klaszterekre [12])

A fenti példán be lettek színezve a kitüntetett klaszterpárok. Ebben a példában az 1. eljáráshoz tartozó két teszteset is rosszul van klaszterezve, rossz klaszterben van (ez a 2. és a 8. teszteset). Így összeszámolva a másik két klaszterre is azt kapjuk, hogy az első metrika értéke $(16 - 4) / 20$, vagyis 0.6 lett.

Egy másik, nagyobb, sok pontot tartalmazó általam előállított példára (4.4. ábra) ez a szám 0.374235 lett -Hamming távolság alapuló klaszterezés esetén.



(4.4. ábra: példa mátrix klaszterezésre)

4.2.2 Metrika 2

A következő, szintén -1 és 1 közé eső mutató már nem csak a jól klaszterezett pontok arányát adja meg az összes többi ponthoz, hanem az egész mátrixra „ráülteti” a definíciót, vagyis a 0 értéket tartalmazó mezőket is figyelembe veszi.

Itt a pont jól klaszterezettségének a fogalmát kiterjesztettük azokra a pozíciókra is, ahol 0 szerepel a mátrixban. Ez gyakorlatilag az inverze lesz annak, mint amikor 1 szerepel egy adott pozícióban.

Például a 4.3. ábrán a klaszteren kívüli pontok közül a 6. tesztesethez és a 2. eljáráshoz tartozó mező „jó helyen van”, ugyanis a kitüntetett klaszterpáron kívül van és az értéke 0. Viszont a 4. tesztesethez és a 4. eljáráshoz tartozó pont már rosszul klaszterezett, ugyanis nem sikerült a benne levő 1 értéket egyik kitüntetett klaszterpárba sem elmozdítani.

Ezek alapján az ehhez a metrikához tartozó képlet a következőképpen néz ki:

$$A = \text{jól klaszterezett 1 értékű pontok} + \text{jól klaszterezett 0 értékű pontok}$$

$$B = \text{rosszul klaszterezett 1 értékű pontok} + \text{rosszul klaszterezett 0 értékű pontok}$$

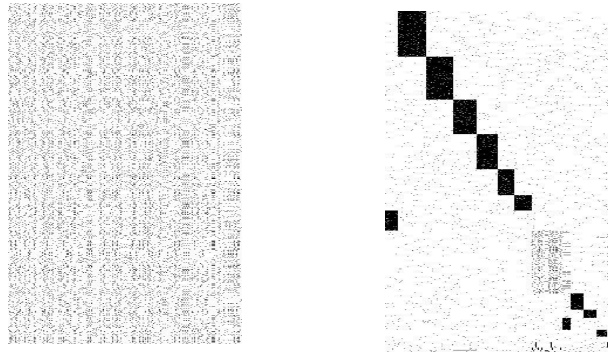
$$M2 = \frac{|A - B|}{|\text{sorok száma} \cdot \text{oszlopok száma}|}$$

(4.5. ábra: a metrika képlete)

Ezek alapján a 4.3. ábrán szereplő példára az $A = 52$, $B = 8$ és így az $M2 = 44/60 = 0.7333$

Ez a mérőszám egy globális értéket ad a teljes mátrix klaszterezettségére, figyelembe véve az összes pont helyzetét.

Egy nagyobb méretű mátrixra megnézve -szintén a Hamming távolságot alapul véve- a 4.6. ábrán levő képre 0.93221 lett a kapott érték.



(4.6. ábra: példa a második metrika kiszámításához)

4.2.3 Metrika 3

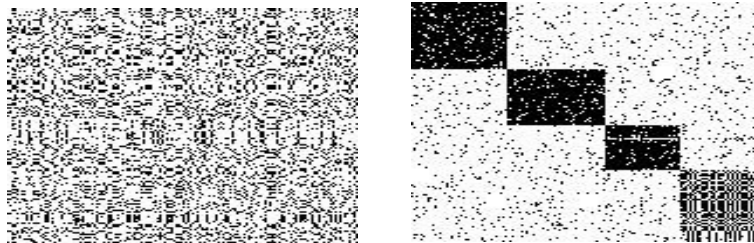
A harmadik metrika a jól és a rosszul klaszterezett pontok egymáshoz viszonyított arányát hivatott mutatni. Összeszámoljuk, hogy mennyi azon pontok száma száma ami jól klaszterezett -legyen az értéke akár 0, akár 1 – és ezt az összeget elosztjuk a rosszul klaszterezett pontok számával. Ez egy 0 vagy nagyobb számot fog adni. Minél nagyobb ez az érték annál jobb az algoritmus (ilyen szempontból).

$$M3 = \frac{\text{jól klaszterezett pontok száma}}{\text{rosszul klaszterezett pontok száma}}$$

(4.7. ábra: a harmadik metrika képlete)

Ennek a metrikának az értéke a 4.3. ábrában levő példára $M3 = 52 / 8 = 6.5$ lesz.

Egy másik, nagyobb példára (4.8. ábra) ez az érték 9.65846 lett.



(4.8. ábra: példa a harmadik metrikához)

A metrikák mellett vannak még más, kevésbé objektíven megítélhető szempontok is. Ilyen például a klaszterszám. Ha az algoritmus kevés halmazt különít el akkor nagyon nagy lesz az eltérés a halmazokon belül és olyan elemek is egy csoportba kerülnek amelyek között szinte nincs vagy csak nagyon kicsit a kapcsolat. Azonban nem szabad átesni a ló túloldalára sem! Ha sok klasztert engedünk meg vagy olyan toleranciaértékeket állítunk be amelyek hatására sok klaszter keletkezik akkor azokban csak a nagyon szoros kapcsolattal rendelkező elemek lesznek és ezekből nehéz komolyabb következtetéseket levonni. Nehéz ideális klaszterszámot mondani, függ a mátrix sűrűségétől, annak belső szerkezetétől, dinamikájától valamint a klaszterező algoritmustól is. A dolgozatnak nem célja az ideális klaszterszám megállapítása, ezért nem térnek ki rá részletesebben.

Egy másik, ehhez hasonló probléma a klaszterek méretének meghatározása. Itt is hasonló problémák merülnek fel, ha túl nagyra vagy ha túl kicsire alkotjuk meg a halmazokat. Ebben az esetben sem célom ideális méretet meghatározni vagy az ehhez szükséges elveket felállítani.

5 Mérések és az eredmények értékelése

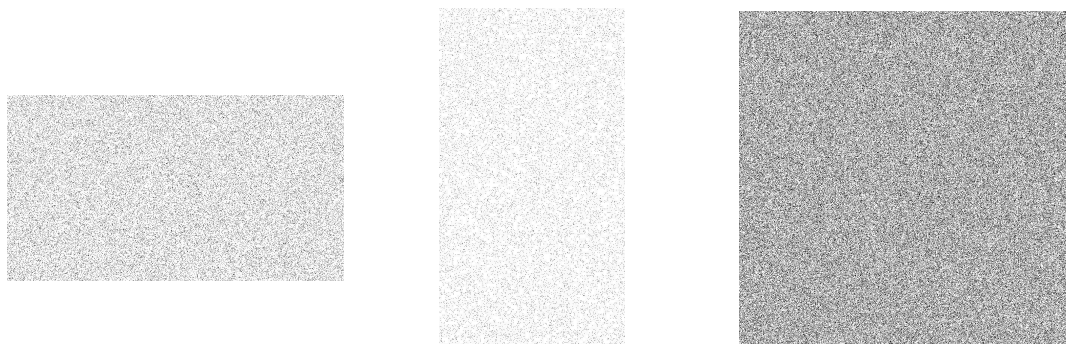
Mivel három különböző típusú mátrixom van -amit a 2.2 fejezetben már bemutatattam- ezért úgy döntöttem, hogy célszerű lenne ezeket külön értékelni.

Az általam generált kétféle mátrixtípust direkt úgy készítettem el, hogy legyen benne sűrű, közepesen sűrű és ritka mátrix is, ezáltal megnézni, hogy azokra milyen értékeket kapunk.

Emellett metrikák szerint is szétbontottam az elemzést, ezáltal részletesebb képet kapva az egyes algoritmusok illetve a mátrixok tulajdonságairól. Az értékeléshez szükséges eredményeket, metrikaértékeket és különböző, futással kapcsolatos (pl.: futási idő) adatokat a konzolon is megjeleníttem, ezáltal jobban követhető, hogy a program épp melyik fázisban van, valamint a legfontosabb értékek tükrében adott esetben (pl.: rossz klaszterszám) újabb futtatást lehet indítani. Ezen futtatások az alábbi eredményeket hozták...

Kezdjük először a random mátrixokkal...

Három különböző mátrixot állítottam elő. Volt egy 500x1000 méretű, amelyben 10% volt az 1 értékek aránya, volt egy 1000x500 méretű amiben 2%-a volt 1 értékű és volt egy 1000x1000-es mátrix, amiben 30% volt az 1-ek aránya (5.1. ábra). Ennél nagyobb százalékos értéket már nem volt értelme választani, ugyanis azokban a mátrixokban már szinte lehetetlen lenne hatékonyan klaszterezni.



(5.1. ábra: általam generált mátrixok)

Mind a 6 algoritmust ugyanazokon a mátrixokon futtattam, többféle paraméterezés mellett. A halmazok közötti kapcsolatra épülő 3 eljárásnál (lásd 3.2 fejezetben) azonos argumentumokat használtam a jobb összehasonlíthatóság miatt.

Itt jegyezném még, hogy a későbbiekben a táblázatokban szereplő *Tree* oszlop a struktúra alapú klaszterezés eredményeit fogja tartalmazni.

	Metrika_1	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
500x1000	Mátrix_1	-0,616751	0,322125	-0,456479	-0,607608	-0,554408	0,368727	0,368727
1000x500	Mátrix_2	-0,676554	-0,444654	-0,259876	-0,574019	-0,640016	-0,810705	-0,259876
1000x1000	Mátrix_3	-0,684293	-0,991969	-0,658914	-0,634197	-0,624581	0,981416	0,981416

	Metrika_2	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
500x1000	Mátrix_1	0,720706	0,783924	0,531432	0,71708	0,724038	0,654676	0,783924
1000x500	Mátrix_2	0,877576	0,942664	0,4851	0,878054	0,885278	0,910996	0,942664
1000x1000	Mátrix_3	0,35631	0,380386	0,263344	0,35554	0,357042	-0,34103	0,380386

	Metrika_3	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
500x1000	Mátrix_1	-0,0183043	8,256	-0,0211022	-0,0225537	-0,0161912	-0,0873052	8,256
1000x500	Mátrix_2	-0,0185053	28,5613	-0,0216918	-0,0218717	-0,0187934	-0,0160224	28,5613
1000x1000	Mátrix_3	-0,0156448	2,22782	-0,0225678	-0,0166446	-0,016833	-19,2584	2,22782

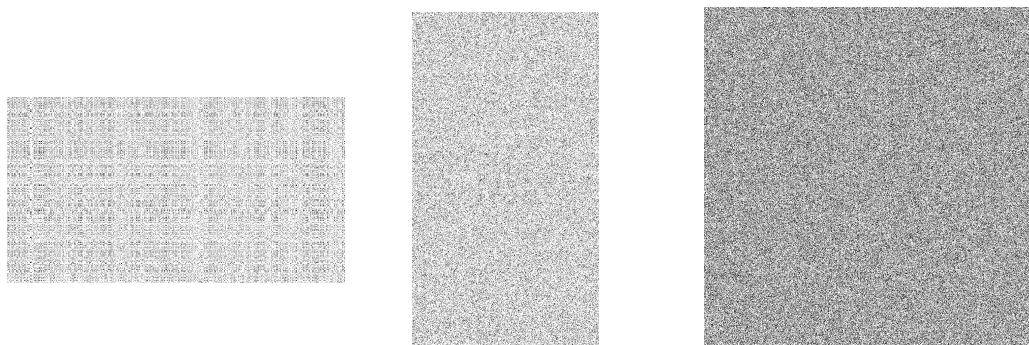
(5.2. ábra: a random mátrix klaszterezésének az eredményei)

Noha egyes metrikákra egészen szép eredmények jöttek ki érdemes egy pillantást a log fájlokra is vetni (amit a mérete miatt nem teszek bele a dolgozatba). Abból ugyanis kiderül, hogy a korábban említett klaszterszám-probléma miatt nagyban torzulhatnak és torzultak is az eredmények.

A *k-means* eljárást használó programok ezt a problémát jobban kezelték, mint a többiek. A *k-means* módszerrel a felhasználó adja meg a klaszterek számát, szóval ezzel ez a probléma megoldódott, maguk a klaszterméretek pedig -a random generálásnak köszönhetően- viszonylag egyenletes oszlanak el. A log -okat nézve látszik, hogy a legjobb metrikaértékekhez tartozó klaszterméretek esetében vagy néhány kiugróan nagy és nagyon sok kicsi méretű klaszter szerepel – pl.: a struktúra alapú megközelítésnél, vagy az összes halmaz egészen kicsi méretű lesz – pl.: Hamming algoritmus esetében. Ezzel szemben ahol számszerűsítve rosszabb eredmények születtek – pl.: Jaccard alg. – ott közel azonos méretű klaszterekbe lehet rendezni a függvényeket és a teszteseteket.

Most térjünk át a speciálisan generált random mátrixok értékelésre... Ebben az esetben is három különböző méretű és sűrűségű mátrixot állítottam el (5.3. ábra). Készítettem egy 500x1000-es méretűt, ahol a kitüntetett klaszterpárokon belüli sűrűség 90%,

az azon kívüli 1%. Emellett elkészült egy 1000x500-as méretű, ahol 35% a „belső” és 15% a „külső” pontok valószínűsége. A harmadik mátrix 1000x1000-es és itt fordított az arány, „belül” 15%, kívül 35% a sűrűség. Ezeken a bemeneteken futtatott algoritmusok az 5.2. ábrán látható eredményeket adták.



(5.3. ábra: speciálisan generált mátrixok)

Megfigyelhető, hogy ezeknél a mátrixoknál is majdnem ugyanazokra az algoritmusokra jöttek ki a legmagasabb metrikaértékek, mint a random mátrixok esetében. Maguknak a mátrixoknak az összehasonlítása már problémásabb, ugyanis az eltérő sűrűség és speciálisan előállított mátrixokba generált mesterséges összefüggések miatt nehéz közöttük párhuzamot vonni.

	Metrika_1	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
500x1000	Mátrix_1	-0,219916	0,803177	-0,459686	-0,27441	-0,547807	0,368727	0,803177
1000x500	Mátrix_2	-0,573864	-0,988793	-0,496739	-0,599538	-0,66503	-0,810705	-0,496739
1000x1000	Mátrix_3	-0,681417	-0,994208	-0,615322	-0,630858	-0,692145	0,981416	0,981416

	Metrika_2	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
500x1000	Mátrix_1	0,708204	0,940132	0,49915	0,717912	0,71614	0,654676	0,940132
1000x500	Mátrix_2	0,586202	0,645072	0,42376	0,605932	0,59306	0,910996	0,910996
1000x1000	Mátrix_3	0,291652	0,311448	0,237628	0,286268	0,289496	-0,34103	0,311448

	Metrika_3	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
500x1000	Mátrix_1	0,0186204	32,4068	-0,0160848	0,0236919	0,0176028	-0,0873052	32,4068
1000x500	Mátrix_2	-0,0176099	4,63495	-0,0241015	-0,0162356	-0,0155406	-0,0160224	4,63495
1000x1000	Mátrix_3	-0,0163408	1,90465	-0,0183052	-0,0168119	-0,0167545	-19,2584	1,90465

(5.4. ábra: a speciális random mátrixok klaszterezésének eredményei)

A log fájlokból viszont egy érdekes észrevételt lehet tenni. Ezeknél a mátrixoknál már nem csak a *k-means* -t használó algoritmusokra jellemző az egyenletes klaszterszám, -és méreteloszlás, hanem például a Hamming távolságon alapulóra is. Ez azért is érdekes, mert a legtöbb esetben ehhez tartozott a legmagasabb metrikaérték. Persze ha belegondolunk a mátrix szerkezetébe, annak generálásába akkor logikusan következik belőle, hogy megfelelő küszöbérték mellett egyenletesen osztja szét az elemeket.

A harmadik és talán legizgalmasabb mátrixtípus következik, ez pedig a valós programokon mért eredményekből állt elő (5.5. ábra). Ezeket különböző opensource projekteken mérték le a SoDA segítségével. Ezekből is hármat szeretnék bemutatni.



(5.5. ábra: Augeas, Dateutils és Bison mátrixa)

Már első ránézésre látszik, hogy ezen mátrixoknak teljesen más struktúrájuk van, mint a mesterségesen előállított véletlen mátrixoknak. Megfigyelhető egy jellegzetes, függőleges, viszonylag vastag fekete rész. Ezek lesznek azok a függvények, amelyek a hívási fában viszonylag a gyökérhez, a belépési pontokhoz közel vannak. Ezek lesznek azok, amelyekből meghívódnak a többi függvények. Ezek lesznek azok az elemek, amelyek a nullklaszterbe fognak tartozni és a metrikaszámításkor meg kell adni egy paraméterben, hogy ezek beleszámítódjanak-e vagy sem a metrikákba (ezek ugyanis kevés információval szolgálnak az elemek közötti különbséget illetően, mivel a legtöbb elemben szerepel).

Emellett a pontok eloszlása sem egyenletes a mátrixon belül, vannak olyan részek ahol nagy sűrűség alakult ki, viszont vannak olyanok is ahol csak egészen kevés pont található. Ezekre a következő eredményeket kaptam:

	Metrika_1	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
Augeas	Mátrix_1	0,00646891	0,05085	-0,0000114	-0,0837536	0,0201141	-0,849698	0,05085
Bison	Mátrix_2	-0,539036	-0,0954952	-0,00573682	-0,236044	-0,374462	0,543281	0,543281
Dateutils	Mátrix_3	-0,57837	0,341802	-0,0222797	-0,582489	-0,528489	-0,826212	0,341802

	Metrika_2	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
Augeas	Mátrix_1	0,202754	0,274366	0,20299	0,236311	0,348135	-0,107603	0,348135
Bison	Mátrix_2	-0,168617	0,245443	0,0975962	-0,131817	-0,166763	0,446763	0,446763
Dateutils	Mátrix_3	0,557278	0,682835	0,18513	0,492917	0,488924	0,620073	0,682835

	Metrika_3	Dice	Hamming	Hash	Jaccard	Ochiai	Tree	MAX
Augeas	Mátrix_1	0,0274323	4,2768	0,00873274	0,324438	0,427125	-0,0290935	4,2768
Bison	Mátrix_2	0,00632541	4,18224	0,012182	3,30022	0,0784423	4,28825	4,28825
Dateutils	Mátrix_3	-0,008291	7,21481	-0,0164271	-0,0548427	-0,0537194	0,0022599	7,21481

(5.6. ábra: valós mátrixok eredményei)

A kapott eredményekből látszik, hogy a legjobb metrikaértékkel rendelkező algoritmusok azok, amelyek a random generált mátrixok esetében is a legjobbak voltak. Ugyanakkor ezeknél a mátrixoknál is fontos volt a megfelelő paraméterek megtalálása -amit többszöri futtatás során tudtuk közelíteni-, ugyanis a klaszterméret és a klaszterszám problémája ezekben az esetekben is fennállt. Ez a probléma egy esetleges későbbi kutatás alapja lehet, aminek az eredményét később beépíthetjük a klaszterező algoritmusokba. Ez azonban túlmutat a dolgozat keretein.

Összességében ha nagy hangsúlyt fektetünk arra, hogy az általunk megadott számú és közel azonos méretű klasztereket kapjunk akkor a *k-means* eljárást használó algoritmusokat, vagyis a Jaccard hasonlóságra, a Dice hasonlóságra vagy az Ochiai együtthatóra építő megoldásokat ajánlom. Ezeknél -amint azt korábban említettem- lehetőségünk van megadni a klaszterszámot is, megakadályozva azt, hogy nagyon sok vagy nagyon kevés klaszter alakulhasson ki.

Ha megengedünk magasabb klaszterszámot és nincs a sorok között nagyon szoros kapcsolat -mint például a Dateutils esetében, ahol jóval keskenyebb a sötét függőleges sáv, mint a másik kettő esetében- akkor egy jó megoldás lehet a struktúrákkal történő klaszterezés használata. Ebben az esetben sok klaszterünk lesz és ezek mind kis méretűk.

Amennyiben azt szeretnénk elérni, hogy legyen néhány nagyobb méretű klaszter, valamint több apróbb akkor ezekre a megfelelő argumentumokkal futtatott Hamming távolságon és a hash függvényeken alapuló eljárások a megfelelőek.

Az így előállított klaszterek segítségével lehetőség nyílik a tesztesetek priorizálására valamint a tesztesetek szelekciójára. Ezek a halmazok szolgáltat(hat)ják az alapját a tesztelés hatékonyságának és minőségének későbbi kutatásához és javításához.

Amennyiben lehetőségem nyílik rá szeretném a későbbiekben is folytatni az ilyen irányú kutatásokat és a dolgozatomat mintegy alapként felhasználva tovább boncolgatni ezt a problémakört...

Irodalomjegyzék

- [1] : International Software Testing Qualifications Board: Certified Tester Foundation Level Syllabus, 2011
- [2] : <http://www.inf.unideb.hu/kmitt/konvkmitt/szoftvertesztes/book.xml.html>
- [3] : <http://www.cs.cornell.edu/home/kleinber/nips15.pdf>
- [4] : Kovács Máté: Klaszterezés ,Budapesti Műszaki és Gazdaságtudományi Egyetem, 2012
- [5] : Jon Kleinberg: An Impossibility Theorem for Clustering, Advances in Neural Information Processing Systems 15, 2003
- [6] : Berend Gábor: Adatbányászat c. kurzus előadásjegyzete, SZTE, 2014
- [7] : http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Hamming_distance.html
- [8] : <https://www.gnu.org/software/octave/>
- [9] : <http://hu.wikipedia.org/wiki/Klaszteranal%C3%ADzis>
- [10] : <http://www.cs.utah.edu/~jeffp/>
- [11] : http://en.wikipedia.org/wiki/Cosine_similarity
- [12] : Pacsika Barbara Liza: Kód lefedettségi osztályozás és annak felhasználás a teszt priorizálás területén, Szeged, 2013

Köszönetnyilvánítás

Ezúton is szeretném megköszönni a segítséget dr. Gergely Tamásnak, aki rendelkezésemre bocsátotta a SoDA library-t és aki az egész kutatás alatt észrevételeivel, iránymutatásával és építő szándékú kritikájával nagyban hozzájárult ahhoz, hogy ez a dolgozat elkészülhetett.
Köszönet érte!