Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Multidimenzionális gráfok elemzése

Szakdolgozat

Készítette
Boér Lehel

Konzulens
Szárnyas Gábor

2017. május 20.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Boér Lehel*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2017. május 20.

Boér Lehel
hallgató

Kivonat

A modern, elosztott gráffeldolgozó rendszerek nagy adathalmazok gyors feldolgozására képesek és használatuk egyre terjed az internetes alkalmazásokban. Mivel a gráfanalízis viszonylag új tudományterület, kevés az összehasonlítás az egyes rendszerek között. Ha egy webáruházat üzemeltetünk, milyen platformot válasszunk a felhasználói viselkedés elemzésére? Nem tudjuk elég részletesen, hogy melyik gráffeldolgozó rendszer milyen teljesítményre képes, és hogy a használati paraméterektől függően mikor melyiket válasszuk.

Ebben a dolgozatban bemutatom a gráffeldolgozás elméleti hátterét és összehasonlítom az Apache Flink és Apache Giraph gráffeldolgozó rendszerek teljesítményét. Leírom az általam készített Java alkalmazás egyes implementációs részleteit, amely 14 multidimenzionális metrikát valósít meg mindkét rendszerben, leméri azok futási idejét, majd összehasonlítja és elemzi őket.

Az eredmények megmutatják, hogy a metrikákat nem elosztott környezetben, egy számítógépen és közepesen nagy (tízezres nagyságrendű), generált gráfokon futtatva az Apache Flink jobb teljesítményre képes.

Abstract

Modern distributed graph processing frameworks are capable of rapidly processing large datasets and their usage in internet applications is growing constantly. Since graph analysis is a relatively new discipline, there is little comparison among the frameworks. If we run a web store, what platform should we choose to analyze user behavior? We do not know the particular differences in the performance of graph processing systems depending on the choice of parameters.

In this thesis I present the theoretical background of graph processing and compare the performance of Apache Flink and Apache Giraph graph processing systems. I describe the implementation details of the Java application that I have made, which implements 14 multidimensional metrics in both systems, measures their execution time and then compares and analyzes them.

The results show that running metrics in non-distributed environments, on one computer and with moderately large graphs (tens of thousands in order), Apache Flink is capable of better performance.

1. fejezet

Bevezetés

Kontextus. A számítógépek és az internet elterjedésével megjelentek az internetes alkalmazások. A felhasználók számának növekedése és a vállalatok profitközpontú gondolkodása a felhasználói élmény maximalizálására sarkallja őket. Megjelentek az ajánlórendszerek, az egyénre szabott reklámok és a kereskedés felhasználói adatokkal, statisztikákkal. Egyszóval megjelent az igény a felhasználói és hálózati adatok gyűjtésére elemzés céljából.

Ezt később tovább fokozták a kilencvenes években megjelent ismeretségi hálózatok, melyek mára nagy népszerűségre tettek szert. Virtuális közösségek jöttek létre és a felhasználók közötti üzenetküldés, tartalmak megosztása - vagyis az interaktivitás - vált jellemzővé rájuk. A 2004-ben megalakult Facebook napjainkban már több mint egymilliárd felhasználóval rendelkezik, akik az internetfelhasználók kb. 30-40%-át teszik ki. [14] [13]A népszerűség miatt az ismeretségi hálózatok elemzésének igénye is nőtt, olyannyira, hogy ez a terület külön tudományágként(kapcsolatháló-elemzés) is megjelent a szociológiában.

Probléma. Az internetes alkalmazásokban a felhasználók és a hálózati topológia elemzése a felhasználói tapasztalatok növelése céljából rengeteg adat feldolgozását vonja maga után. Mivel az internetes alkalmazások a mai napig folyamatosan fejlődnek, egyre pontosabb és jobb módszerekre van igény a felhasználók viselkedésének elemzéséhez, hálózati metrikák számításához. Ezeket a módszereket használják mind üzleti környezetben a kommunikáció javítására, közösségi hálókon a felhasználók igényeinek kielégítésére, ajánlórendszerek készítésére, hálózati operátorok a hálózati struktúra és teljesítmény optimalizálására és számos más területen is. Szükség lett olyan rendszerekre, melyek képesek nagy számú adatot viszonylag gyorsan feldolgozni. A számítógépes hálózatok és közösségi hálók adatait és a közöttük lévő kapcsolatokat egyszerű és átlátható módon multidimenzionális gráfokkal modellezhetjük. Az adatok ilyen formában történő feldolgozására megjelentek különféle gráffeldolgozó rendszerek, melyekkel kiszámolhatjuk a gráfok számos tulajdonságát. További probléma, hogy jelenleg nem létezik olyan nyílt forráskódú projekt, amely beépített multidimenzionális metrikákat tartalmaz és támogatja az elosztott környezetet, illetve a létező rendszerek teljesítményéről is kevés információ áll rendelkezésre.

Cél. Az adatok feldolgozásának igénye miatt egyre több gráffeldolgozó rendszer jelenik meg. Ugyanakkor a gráfanalízis kutatási területe még viszonylag új, nem kiforrott, ezért nincsenek összehasonlítások az egyes rendszerek között. Hosszú távú cél egy olyan keretrendszer létrehozása, amely jobban támogatja a multidimenzionális metrikákat, továbbá a mai modern gráffeldolgozó rendszerek teljesítményének megmérése éles, elosztott környezetben valós adatokkal, és a mért eredmények összehasonlítása. Mivel ezek a rendszerek különböző modellekre épülnek és eltérő implementációval rendelkeznek, a teljesítményük

elemzése nem csak kideríti, hogy milyen paraméterek mellett melyik teljesít a legjobban, hanem irányt mutathat újabb gráffeldolgozó alkalmazások készítéséhez.

Megvalósítás. A dolgozatomban két modern gráffeldolgozó rendszer, az Apache Flink és az Apache Giraph működését mutatom be. Generált gráf adathalmazon az előbbi rendszerek fölött multidimenzionális metrikák futási idejének mérésével összehasonlítom és elemzem a teljesítményüket.

Kontribúciók. A dolgozatom a következő kontribúciókat tartalmazza:

- 17 multidimenzionális metrika, amelyből 14-et tartalmazza a saját implementációs részleteit.
- A metrikákat lefuttattam Apache Giraph-on, Apache Flinken és lemértem a futási idejüket.
- A futási időket elemeztem, a két rendszer teljesítményét összehasonlítottam.

2. fejezet

Háttérismeretek

Ez a fejezet részletezi az alapvető fogalmakat és definíciókat, melyeket a dolgozat többi részében használok, valamint bevezet egy vezérpéldát az algoritmusok szemléltetésére.

A való életbeli hálózatok legegyszerűbb matematikai reprezentációja a gráf. A hálózat entitásait a hálózatot leíró gráf csúcspontjainak, az entitások közötti kapcsolatok pedig az egyes csúcspontok között húzódó éleknek feleltethetjük meg.

2.1. Multidimenzionális gráfok

2.1.1. A gráf fogalma

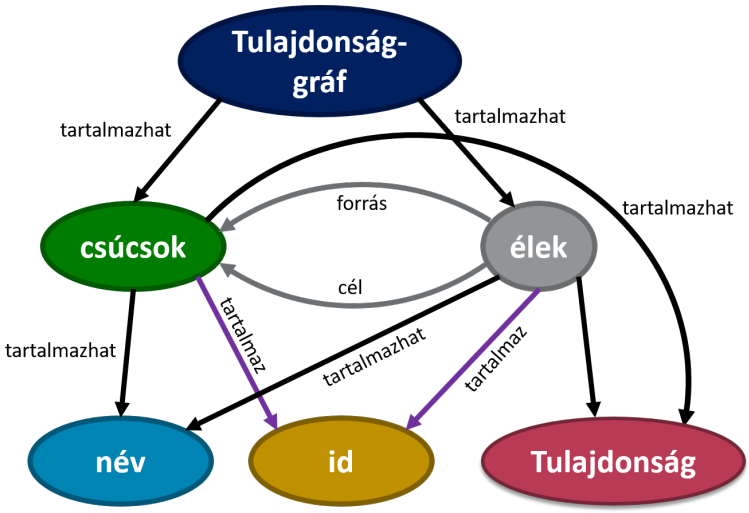
A gráf csomópontok és a köztük futó élek halmaza. Matematikailag a gráf egy A halmaz és a felette értelmezett $\rho \subseteq A \times A$ bináris reláció. Ekkor a $G = (A, \rho)$ párt, vagyis az A halmaz feletti relációs struktúrát az A halmaz feletti gráfnak nevezzük. Az A halmazt a gráf csúcshalmazának, a csúcsok között értelmezett bináris relációt pedig a gráf élhalmazának hívjuk. Ez a definíció nem követeli meg a ρ reláció szimmetriáját, ezért csak irányított gráfokra igaz. Egy irányított gráf élei irányítottak, kezdő -és végponttal rendelkeznek, azaz megmondják, hogy az él melyik csúcsból indul és melyikbe tart. A továbbiakban a dolgozatban irányított gráfokról lesz szó.

2.1.2. Tulajdonsággráfok

A tulajdonság gráf az alapvető gráfdefiníció kibővítése. A csúcsok és élek rendelkezhetnek egyéb információkkal. A tulajdonsággráf csúcsok és élek halmazából áll, melyben minden csúcsnak és élnek kötelezően rendelkeznie kell egy azonosítóval, továbbá rendelkezhetnek nevekkel és tulajdonságokkal is. Egy csúcs nevét címkének, egy él nevét típusnak vagy éltípusnak hívjuk. Az élek a típusukkal meghatározzák két csúcs kapcsolatát. A tulajdonságok tulajdonságnév és tulajdonságérték párok. A csúcsok neveit, mivel azok valamilyen entitást jelölnek, gyakran hierarchiába rendezik. Például egy csúcs lehet ember és azon belül tanár is. Ennek a jelölése általában Tanár:Ember, ahol mindkettő létező címke. A tulajdonsággráf metamodellje a 2.1 ábrán látható. [9]

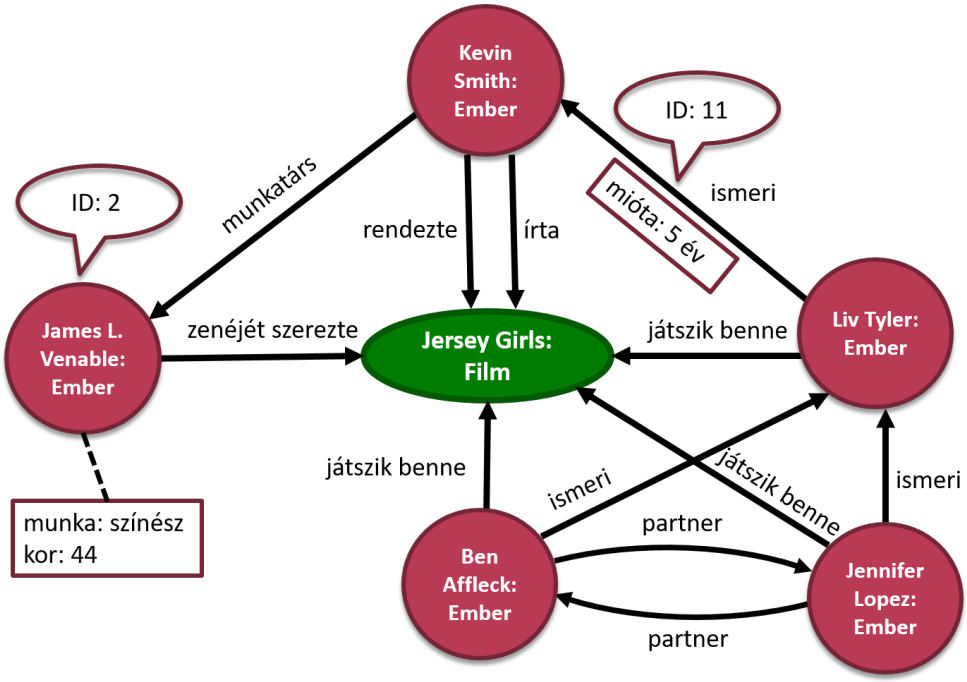
2.1.3. Multidimenzionális gráfok

A egydimenziós gráfok gráfelméleti szempontból annyit jelentenek, hogy a gráf csúcspontjai között egyetlen él futhat. Ezzel szemben a multidimenzionális gráfok [15] csúcsai között nem csak, hogy több él futhat, és a tulajdonsággráf adatmodellt használva meg is különböztethetjük őket élcímkék segítségével. A valóságban - például egy közösségi hálót tekintve - a gráfok általában multidimenzionálisak. Egy ismeretségi hálózatot gráfként



2.1. ábra. Tulajdonsággráf metamodell.

modellezve megkülönböztethetjük az emberek közötti kapcsolatok típusát. Lehetnek rokonok, barátok vagy munkatársak és ezek közül egyszerre több is. A 2.2 ábra egy egyszerű multidimenzionális gráfot mutat be fiktív tulajdonságokkal.



2.2. ábra. Egyszerű multidimenzionális gráf.

A csúcsok embereket vagy filmeket ábrázolhatnak, az élek a közöttük lévő kapcsolatokat írják le. Egy csúcs címkéje egy ember vagy film neve, melyek kettősponttal követnek az és az erre utaló gyűjtőnevek. Ez hasznos az átláthatóság szempontjából, meghatározza, hogy adott típusú címkék között milyen kapcsolatok lehetnek, így a gráfok informatikai feldolgozásnál is segíti a lekérdezéseket. Egy ember egy filmet rendezhetett, játszhatott benne, megírhatta a forgatókönyvét vagy szerezhette a zenéjét. Két ember ismerheti egymást, az egyszerűség kedvéért egyoldalúan is, lehetnek partnerek vagy munkatársak. Min-

den csúcs és él rendelkezik egy egyedi azonosítóval és rendelkezhet tulajdonságokkal. Két csúcs között egyszerre több típusú él futhat, de egyetlen típusból csak egy, azaz a többszörös élek nem megengedettek. Az következő fejezetekben ezt a példát fogom használni a gráfmetrikák bemutatására.

2.2. Gráffeldolgozás

A fejezet bemutatja a gráfanalízis alkalmazási területeit, típusait, a gráfmintaillesztést és a gráffeldolgozási módszereket.

2.2.1. Gráfanalitika

A gráfanalitika, vagy más néven gráfanalízis a gráfok komplex feldolgozása, mely során információkat nyerünk ki a gráf által reprezentált valódi rendszerről és képet kapunk annak felépítéséről, viselkedéséről. Viszonylag új tudományterületnek számít és jelenleg a közösségi hálózatok elemzésében is nagy szerepe van. A mai marketing egyik célja a közösségi hálók befolyásoló hatásainak felderítése, mert a virtuális közösségek a marketing kampányok legújabb célpontjai. Ezen kívül számos más területen is széles körben használt:

- Pénzügyi csalások, pénzmosás felderítése
- Bűnügyek megelőzése, antiterrorizmus
- Elektromos, víz és transzporthálózatok hibáinak felderítése
- Utak optimalizálása légi közlekedésben és logisztikában
- Élettani kutatások (bioinformatika)

A gráf analitikának négy fő típusa van: útvonalkeresés, kapcsolatelemzés, közösségelemzés és központi analízis. [8]

Probléma egy hálózatban megtalálni két pont között a legrövidebb utat. Üzleti környezetben szállítások esetén a megfelelő útvonal megtalálása kiemelt jelentőséggel bír a költségek minimalizálása végett. Ezt számos egyéb paraméter mellett az út hossza is befolyásolja. A legrövidebb utak megtalálására számos gráfelméleti algoritmus létezik. A Dijkstra egy mohó algoritmus, amivel irányított vagy irányítatlan, akár pozitívan súlyozott gráfokban lehet megkeresni a legrövidebb utat két csúcs között egy adott csúcspontból kiindulva. A mélységi keresés (DFS) egy csúcsból kiindulva fát épít ki a gráfban, bejárva az összes csúcsát, így felderíti a fa gyökere és a fa többi pontja közötti legrövidebb távolságokat. A Bellman-Ford algoritmus súlyozott gráfokban keresi meg a legrövidebb utakat egy kiválasztott csúcs és a gráf többi csúcsa között, a Floyd-Warshall módszer pedig a negatív élsúlyokat is megengedi. Jelenleg az $O(|E| + |V| \log |V|)$ futási idejével a Dijkstra algoritmus az egyik leggyorsabb legrövidebb utak keresésére körmentes, pozitívan súlyozott gráfokon.

A kapcsolatelemzés a hálózatok gyenge részeinek a felderítésére és hálózatok összekötöttségének összehasonlítására szolgál. A közösségelemzés távolság és sűrűség alapú elemzés, melynek segítségével megtalálhatjuk az interaktív csoportokat egy közösségi hálózatban. Végül a központi analízis a hálózatok legjelentősebb csúcsaival foglalkozik. Megkeresi a legbefolyásosabb embereket vagy a sokat látogatott weboldalakat valamilyen pagerank algoritmust használva.

2.2.2. Gráfmintaillesztés

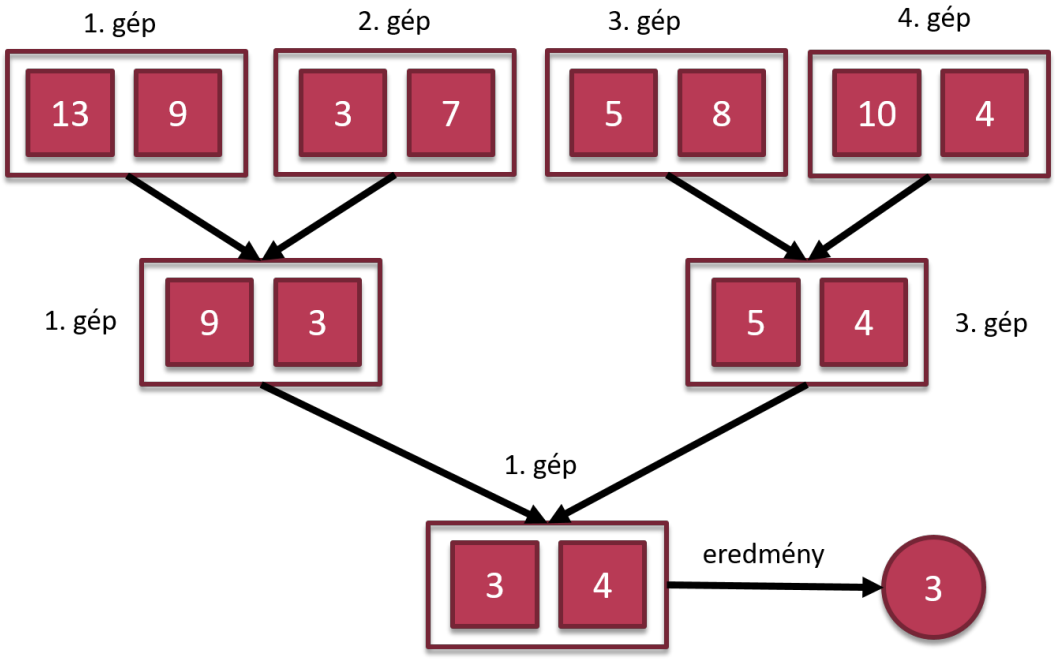
A gráffeldolgozás megjelenik a modell-vezérelt szoftverfejlesztés területén is. A fejlesztési módszertan kulcseleme a modellvalidáció, melynek segítségével a automatikus úton biztosíthatjuk egy modell konzisztenciáját. A modellvalidáció több fázisában transzformáljuk a modellt és megvizsgáljuk, hogy teljesülnek-e a rá vonatkozó jól-formáltsági kényszerek. Mivel mind a modell, mint a kényszerek intuitívan gráfként ábrázolhatók és dolgozhatók fel, valójában egy gráfban (modell) részgráfként keressük egy másik gráf (kényszer) előfordulását. Ez tipikus esete a gráfmintaillesztésnek.

2.3. Gráffeldolgozási módszerek

Ez a fejezet a gráfok feldolgozására alkalmas legelterjedtebb megközelítések működését mutatja be különös tekintettel az ún. *csúcs-centrikus* technikákra. Csúcs-centrikus gráffeldolgozás közben a hangsúly az gráf egésze helyett egy csúcsra helyeződik. Al algoritmusok megvalósításánál a gráf egy csúcsaként kell gondolkodnunk, mert az fogja elvégezni a részszámításokat és az fog kommunikálni a többi csúccsal. Előnye, hogy az ilyen rendszerben megvalósított algoritmusok elosztott rendszereken futtathatók. [2]

2.3.1. BSP és Pregel

A csúcscentrikus iteráció alapja a Bulk Synchronous Parallel (BSP) modell, amit Leslie Valiant fejlesztett ki az 1980-as években [17]. A BSP alkalmas algoritmusok párhuzamos futtatására és szinkronizációjára. Tegyük fel, hogy van négy gépünk, melyek el tudják dönteni, hogy két szám közül melyik a nagyobb. Nyolc pozitív egész szám közül szeretnénk a lehető leghamarabb megkapni a legkisebb számot. A megoldás a számítások párhuzamosítása. Az alábbi ábra erre mutat egy lehetséges megoldást.

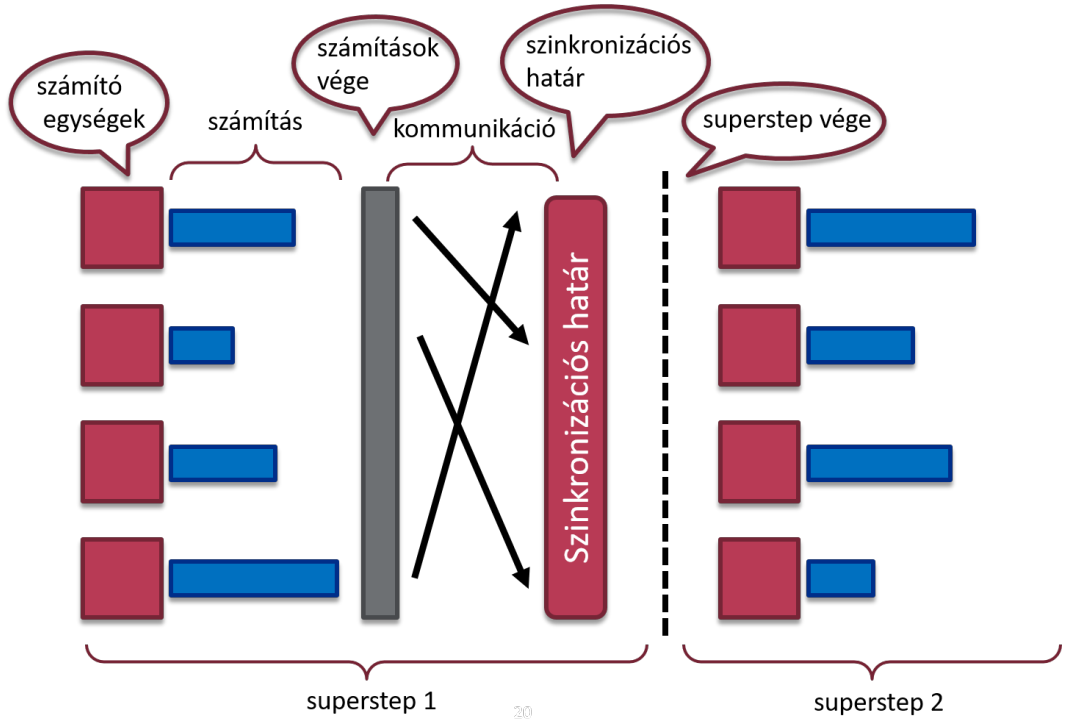


2.3. ábra. Párhuzamos minimumkiválasztás hét gépen.

A számítás három fázisból fog állni és egy fázisban egy gép két számot kap, melyek közül eldönti melyik a kisebb, majd az eredményt elküldi a tőle balra elhelyezkedő gépnek és így tovább.

- Első fázis: Az gépek kiszámolják a minimumot a hozzájuk rendelt két szám közül. A második gép az elsőnek, a negyedik a harmadiknak elküldi az eredményét. Várnak amíg a gépek közötti kommunikáció véget ér.
- Második fázis: A első és harmadik gép rendelkezik a saját eredményével és a szomszédjától kapottal. Kiszámolják a két szám minimumát, majd a harmadik gép elküldi az eredményét a másodiknak. A gépek megvárják a kommunikáció végét.
- Harmadik fázis: Vegyük észre, hogy már csak az első gép számol, akinek van egy saját eredménye és egy a második fázisban a harmadik géptől kapott eredménye. Kiszámolja a minimumot, de már nem tudja senkinek tovább küldeni, mert egyedül van. Ez a minimum a végeredmény.

Az BSP absztrakt módon általánosítja az algoritmusok párhuzamos feldolgozását. Az algoritmus bemeneteit elosztja több számítást végző egység között, melyek lépésekre osztva, egymással párhuzamosan részeredményeket számolnak és kommunikálnak egymással. Ilyen lépések sorozatával kiszámolják az algoritmus végeredményét.



2.4. ábra. A BSP számítási modellje négy számítást végző egységgel.

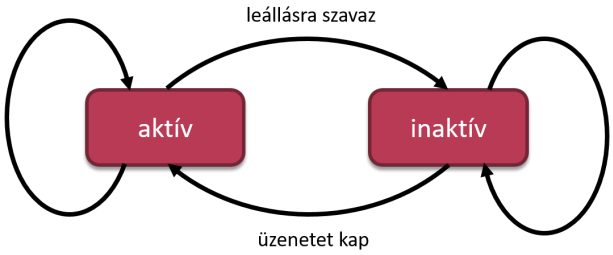
Az egységek a kapott bemeneteken elvégzik a részeredmények kiszámítását. A szürke határ azt jelzi, hogy a BSP modellben az egységek megvárják, amíg mindegyikük befejezte a számításait, és csak utána kezdik el az egymás közötti kommunikációt. Az eredményeik egymás közötti cseréje után szintén egy szinkronizációs lépés következik, ami megvárja, hogy az egységek befejezzék a kommunikációt. Ez hívjuk szinkronizációs határnak. Az

egységek számításainak kezdetétől a szinkronizációs határig eltelt időt superstepnek hívjuk. A superstep a párhuzamos feldolgozás iterációs egysége, amelynek a végét mindig a szinkronizációs határ végzi (Ábra 2.4).

A Pregel modellt a Google fejlesztette ki 2010-ben nagy gráfok feldolgozására. [12] Előnye a gráfalgoritmusok könnyű és intuitív megvalósítása, valamint a gyors számítási képessége az algoritmusok elosztott rendszereken való párhuzamos futtathatósága miatt. A Pregel modell a BSP-n alapszik, de a modellt átértelmezték gráf környezetbe és bővítették a funkcionalitását. A Pregel-beli gráfok csúcsok és irányított élek halmazából áll, amit a következő módon reprezentál:

- csúcs: Egy csúcsnak van egy saját azonosítója (id), egy értéke és az éleinek listája.
- él: Egy él tartalmazza a saját értéket és a végpontjának az azonosítóját.

Egy él nem rendelkezik külön azonosítóval sem a kezdőpontjának az id-jával. Ez a csúcs-centrikus nézőpont miatt van, mivel egy csúcs tudja a saját éleinek a halmazát, és a számítások illetve a kommunikáció mindig egy csúcs környezetében történik. A Pregel-számítások iterációrészekből állnak, amit itt is superstep-nek neveznek. Egy superstep során az összes aktív csúcs egymással párhuzamosan elvégzi a felhasználó által definiált számítást. A számítás során a csúcsok beállíthatják a saját értéküket, módosíthatják a gráf topológiáját és üzeneteket küldhetnek a szomszédainak meg azoknak a csúcsoknak, amiknek ismeri az azonosítóját. A számítás és üzenetküldések után szavazhat a leállásra. Ha egy csúcs leállásra szavaz, inaktív állapotba kerül és amíg nem kap valamelyik másik csúcstól üzenetet, vagy véget nem ér az algoritmus, nem végez számítást.



2.5. ábra. A Pregel modellbeli gráfcsúcsok aktivitásának állapotdiagramja.

Egy aktív csúcs továbbá aktív állapotban marad, amíg nem szavaz leállásra vagy véget nem ér az algoritmus. Az eddigi megvalósított BSP funkcionalitások felett a Pregelben vannak kombinátorok és aggregátorok a számítási teljesítmény növelésére.

- A kombinátorok üzeneteket gyűjtenek össze és egy nagy üzenetbe csomagolják őket,
- az aggregátorok pedig számításokat végeznek a bejövő üzeneteken.

A kommunikáció a csúcsok között teljesítményigényes művelet. Például ha meg szeretnénk keresni a legkisebb értékkel rendelkező csúcsot, ahelyett, hogy a csúcs szomszédjainak küldözgetjük a csúcs értékét és összehasonlítjuk a kapott üzeneteket a saját értékünkkel, az értéket elküldhetjük egy aggregátornak, aki egy lépésben kiszámolja a végeredményt. A BSP és a Pregel közti lényeges különbség, hogy a BSP superstepje három fázisból áll. A BSP megvárja, amíg egy superstepen belül a csúcsok elvégzik a számításokat és csak utána kezdődik az üzenetküldés, végül a szinkronizáció. A Pregelben a csúcsok rögtön elkezdik küldeni az üzeneteiket, amint befejezték a számításaikat. Ez gyorsítja a számolásokat, mert

azoknak a csúcsoknak, amiknek kevés számításra van szükségük, de több üzenetküldésre, nem kell várakozniuk a több számítást igénylőkre.

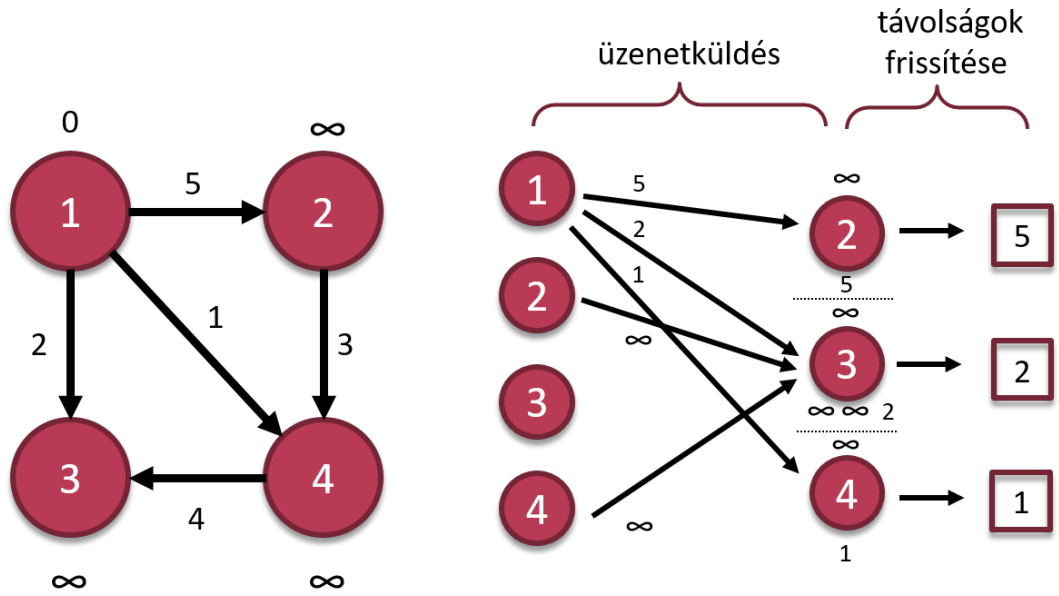
Ide kapcsolódik még a MapReduce modell, ami a Pregelhez hasonlóan működik [12] és ugyanúgy nagy adathalmazok párhuzamos feldolgozásával foglalkozik. A Pregel előnye a MapReduce-al szemben, hogy míg az utóbbi minden superstep után kiírja az eredményeket valamilyen háttértárra, a Pregel a memóriában tárolja őket, ezért gyorsabb.

2.3.2. Scatter-Gather

A Scatter-Gather modell nagyon hasonlít a Pregelre, a számítások itt is superstepekből állnak és a csúcsok kommunikálnak egymással. Egy superstepben egy csúcs üzenetet küldhet más csúcsoknak, és változtathatja az értékét a beérkezett üzenetei alapján. Ez két fázisból áll:

- Scatter: Üzeneteket készít és elküldi a kívánt csúcsoknak.
- Gather: Frissíti az értékét a beérkezett üzenetek alapján.

A legfőbb különbség az előző modellekhez képest, hogy kettéválik az üzenetküldés és a csúcsok értékváltozásának logikája, illetve a csúcsok nem kapnak leállási opciót. Ha egy csúcs nem frissíti az értékét a Gather fázisban, a következő superstep-ben nem küld üzenetet a többi csúcsnak. A Scatter-Gather memóriatakarékosabb, mert nem igényel olyan struktúrát, amivel egy fázisban kezeli a ki -és bemenő üzenetek egyszerre. Az egy csúcsból induló legrövidebb út megkeresése a gráfban nagyon jól szemlélteti a modell működését.



2.6. ábra. Az 1-es csomópontból induló legrövidebb utak keresése.

A bal oldali négy csúcsból álló gráfon az 1-es csúcstól a többi csúcs legrövidebb távolságát szeretnénk megtalálni. A Scatter fázisban a csúcsok elküldik az értékük és a címzettől való távolság összegét a kimenő éleken. A Gather fázisban a címzettek megkapják a leveleket és értéküket frissítik a kapott legkisebb távolság alapján. Ha valamelyik csúcsnak nem frissült az értéke, ő nem küld tovább üzeneteket a következő superstepben.

Az ábra a folyamat első superstepjét ábrázolja. A következő superstepben az 1-es csúcs nem küld üzeneteket, mert a Gather fázisban nem változott az értéke. A 2-es csúcs elküldi a 4-esnek a saját értékét és a köztük lévő út távolságát, ha esetleg az 1-es és 4-es csúcs közötti legrövidebb út rajta vezetne keresztül. De nem így történik, ezért a 4-es csúcsnak nem változik az értéke. A 4-es csúcs hasonlóan a 3-asnak küld üzenetet, de annak sem változik az értéke. Mivel egyik csúcsnak sem frissítette az értékét a Gather fázisban, az algoritmus véget ér.

2.3.3. Gather-Sum-Apply

A Gather-Sum-Apply szintén a BSP modellt követi és háromfázisú superstep-ekből áll.

- Gather: A csúcsok a felhasználó által definiált függvényt hajtják végre párhuzamosan.
- Sum: Az előző fázisban kiszámolt részeredményeket egy ún. reducer egyetlen értékké alakítja.
- Apply: A csúcsok frissítik az értékeiket a jelenlegi értékük és az aggregált érték alapján.

A fő különbség a Scatter-Gather és a Gather-Sum-Apply között, hogy az utóbbi párhuzamos küldi el az üzeneteit a reducernek és csak a csúcs szomszédaival operál. Scatter-Gather és Pregel esetén olyan csúcsoknak is küldhetünk üzeneteket, akiknek ismerjük az id-ját, de nem feltétlenül szomszédosak velünk. Bonyolultabb számítások esetén, ahol sok üzenetet kell küldeni, érdemes ezt a módszert használni. A legrövidebb útkeresés során először a Gather fázisban a csúcsok kiszámolják a szomszédaik fél vezető út hosszát a saját értékük és az út távolsága alapján. A Sum fázisban a reducer a csoportosított csúcs id-k alapján kiszámolja az egyes csúcsokhoz tartozó legrövidebb távolságokat, majd az Apply fázisban a csúcsok frissítik a távolságértékeiket. Fontos, hogy ebben a modellben a kommunikáció kizárólag a csúcsok és a reducer között történik. Ha egy csúcs nem frissítette az értékét, a következő superstepben nem fog távolságot számolni. Ha egy superstepben egy csúcs sem frissítette az értékét, az algoritmus leáll.

3. fejezet

Gráffeldolgozó rendszerek

Ez a fejezet az Apache Flink és Apache Giraph gráffeldolgozó rendszerek felépítésével, belső működésével, illetve a Java API-juk bemutatásával foglalkozik.

3.1. Apache Flink

3.1.1. Bevezetés

Az Apache Flink egy nyílt forráskódú adatfolyam feldolgozó keretrendszer elosztott, magas rendelkezésre állású alkalmazások számára.[1] Az Apache Flink pontosabb működésének megértéséhez szükséges az ide kapcsolódó alapvető fogalmak definiálása. Ha egy közösségi hálózat felhasználóinak adatait szeretnénk feldolgozni, új felhasználó regisztrálásakor, vagy egy fiók törlésekor teljesen új adat keletkezik vagy meglévő adat tűnik el, változik az adatok számossága. A felhasználók adatai is változhatnak, így szükség lehet a már feldolgozott adatok módosítására. Tehát egy adathalmaz a mérete szempontjából lehet korlátos vagy korlátlan, az adatok szempontjából pedig statikus vagy változó. A rendszer a gráffeldolgozás során kétféle adathalmazt használ ennek a problémának a megoldására:

- korlátos adathalmaz (Bounded dataset): Olyan adatok halmaza, melyek elemeinek száma és az elemek állandók, nem változnak az adatfeldolgozás során.
- korlátlan adathalmaz (Unbounded dataset): Végtelen adatok halmaza, melyek időközben változhatnak.

A határozatlan adathalmaz modellje jobban leírja a való világ adatainak viselkedését. Az ilyen adathalmazok egyaránt megjelennek a mobil és webalkalmazások felhasználói interakcióiban, folyamatos környezeti szenzorok által szolgáltatott mérésekben, naplózórendszerekben, valamint általánosan minden olyan adatot produkáló rendszerben, ami egy időszakra nézve folyamatosan produkálja az adatokat. Tegyük fel, hogy egy könyv szavairól szeretnénk statisztikát készíteni az előfordulási gyakoriságuk alapján informatikai eszközökkel. Ebben az esetben adatfeldolgozásról beszélünk, és maguk a szavak az adatok. Ha a könyv már meg van írva, akkor egy statikus és korlátos adathalmazról van szó, amire használhatjuk a Flink adatköteg feldolgozási modelljét. Ellenben ha a könyvet még írják - és már a könyv írása közben is szükségünk van a statisztikákra-, az adathalmaz korlátlan és változó, hiszen nem ismerjük az idejét a könyv elkészültének és az írása közben korábbi fejezetekben is történhet változás. Így határozatlan ideig folyamatos adatfeldolgozásra van szükség, amit a Flink adatfolyam feldolgozó modellje biztosít. A határozott és határozatlan adathalmazok különböző feldolgozási mechanizmusokat igényelnek. A Flink a következő végrehajtási modellekkel rendelkezik:

[1] `http://flink.apache.org`

- Adatfolyam feldolgozás (Streaming): A feldolgozás folyamatos és addig tart amíg van adat. Ezt korlátlan adathalmazok feldolgozására használják.
- Adatköteg feldolgozás (Batch): Korlátos adathalmazok feldolgozására való, a feldolgozás véges időn belül befejeződik.

3.1.2. Az Apache Flink előnyei

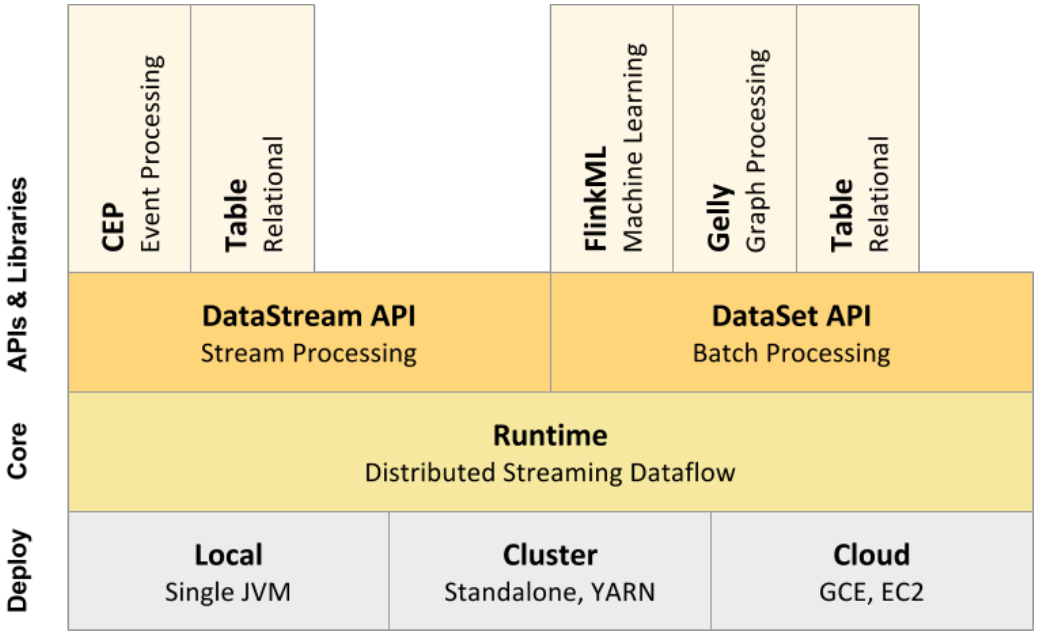
Az Apache Flink számos új funkciót hozott, amivel az addigi adatfeldolgozó rendszerek (például Apache Storm[2]) nem rendelkeztek.

- Elosztott rendszerekre tervezték: A Flink hatékonyan működik elosztott rendszereken, képes az adatokat párhuzamosan több számítógépen feldolgozni, ami növeli a számítások teljesítményét.
- Pontos eredmények: A hálózatokban az egyes adatcsomagok nem feltétlenül ugyanazon az úton jutnak el a címzetthez, ezért más sorrendben is megérkezhetnek, mint ahogy továbbításra kerültek. Tekintve, hogy a Flink nagyteljesítményű, sok csomópontos klaszterekre van tervezve, az adatfeldolgozás során az adatok különböző utakat járhatnak be a hálózatban, így az adat feldolgozásának ideje különbözhet az adat megérkezésének az idejétől. A Flink megoldja ezt a problémát és a fejlesztőknek biztosít egy programozási felületet az adatfeldolgozáshoz kapcsolódó időműveletek kezelésére.
- Támogatja az ablakok használatát: Képzeljük el, hogy forgalmi adatokat mérünk és egy szenzor percenként megszámolja, hogy hány autó haladt át előtte. Mivel az adatok folyamatosan jönnek, az érzékelt autók száma minden perc végén hozzáadódik az eddigi összeghez, ami egy összesített statisztika, nem percenkénti. A Flink az ablakok használatával diszkretizálja adatfolyamot, tehát ablakokra bontjuk és ezeket külön feldolgozandó egységként kezeli. Egyperces időablakok használatával már ki tudjuk számolni a percenként mért autók számát. További probléma, ha egyszerre több, térben különböző szenzorunk van, melyek folyamatosan mérik a forgalmat és szeretnénk egyenként statisztikákat számolni rajtuk. Ebben az esetben a Flink képes arra, hogy az időablakokat egy további szenzorazonosítóval lássa el, így azonosítani tudjuk a különböző forrásokból származó adatokat. [10]
- Hibatűrő: Ha a Flinket elosztott rendszeren használjuk, ún. jobokat indíthatunk. Egy job mindig tartalmaz valamilyen algoritmust, ami az adatokat manipulálja, és egyéb beállításokat (például hány szál futassa). A Flink verziókezeli a rendszer állapotát az egyes jobok között, így bármilyen hiba esetén visszaállíthatjuk azt egy korábbi időpontra.
- Nagy teljesítményre képes: Képes nagyon sok adatot nagyon gyorsan feldolgozni kis késeltetéssel.

3.1.3. Az Apache Flink architektúra

A Flink architektúra három fő rétegből áll. A program magja - a Core réteg- egy elosztott, adatvezérelt adatfolyam-feldolgozó motor. Elosztott, mivel általában számítógépfürtökön fut és adatvezérelt, mert eseményszerűen dolgozza fel az adatokat, amikor azok rendelkezésre állnak. A Deploy réteg az alkalmazás futtatási környezetét mutatja be. Futhat lokálisan, saját Java virtuális gépen (JVM), futhat Hadoop vagy Akka Actor klaszteren [1], de akár felhőben is.

[2] `http://storm.apache.org`



3.1. ábra. Az Apache Flink architektúrája

Az APIs & Libraries réteg tartalmazza a főbb API-kat. A DataStream API az adatfolyam-feldolgozásra, a DataSet API, az adatköteg-feldolgozásra alkalmas. A Gelly a FLink gráf API-ja, melyek segítségével gráfokat dolgozhatunk fel, erről a későbbiekben részletesebben lesz szó.

3.1.4. A DataSet API

A 4 fejezetbeli metrikák implementálása során a DataSet API-t használtam, mert véges, nem változó gráfon mértem a teljesítményt. A DataSet API végzi a véges adathalmaz objektumok előállítását és az adatok transzformálását. DataSet objektumokat számos forrásból létrehozhatunk: például CSV, szöveges vagy Hadoop fájlból és Java kollekciókból is. A DataSet transzformációk egy DataSet objektumon egy másik, új DataSet objektummá transzformálják. A legfontosabb transzformációk röviden:

- Map: Egy felhasználói metódust alkalmaz sorban minden elemen, és egy elem pontosan egy másik elemmé transzformálódik.
- FlatMap: Szintén egy metódust alkalmaz az elemeken, de egy bemenetre nulla, egy vagy több kimenetet adhat.
- Filter: Szűri az adathalmaz elemeit valamilyen felhasználói feltétel szerint.
- Reduce: Elempárokat egyetlen elemmé transzformál egészen addig amíg egy elem marad, vagy egy sem.
- GroupReduce: Végigiterál az adathalmazon és több elemből több elem transzformációt végez.
- Join: Összekapcsol két adathalmazt valamelyik kulcsuknál fogva.

A transzformációs függvények mindig egy DataSet objektummal térnek vissza, de a paraméterük változó. A join transzformáció egy másik DataSet-et vár paraméternek, a

GroupReduce és a legtöbb interfészt, ami megvalósítja az adott funkciót. Ezt a legegyszerűbb helyben anonim osztályként implementálni.

3.1.5. Gelly

A Gelly a Flink gráf API-ja, amit gráffeldolgozó alkalmazások készítésére hoztak létre. Segítségével létrehozhatjuk, módosíthatjuk vagy transzformálhatjuk a gráfokat, és tartalmaz egy könyvtárat különféle gráfalgoritmusokkal. A Gelly a gráf éleit és csúcsait DataSet objektumokban tárolja, a csúcsokat egy egyedi azonosítóból és egy értékből, az élek két csúcs azonosítójából - melyek között futnak - és egy értékből állnak. A gráfok létrehozhatók beépített metódusokkal az éleket és csúcsokat tartalmazó DataSet-ekből, CSV fájlokból és Java kollekciókból. A fontosabb gráftulajdonságok szintén elérhetők beépített módon: A csúcsok és élek halmaza, a csúcsok egyedi azonosítóinak és az élek azonosítópárjainak halmaza, mind DataSet formában. A kimenő, bemenő és összesített fokszámok azonosító-fokszám alakban, szintén DataSet-ben. A csúcsok és élek száma, illetve kezdőcsúcs, végcsúcs, él hármasok.

A gráftranszformációk bemutatása előtt megjegyzem, hogy a Flink generikus tupleöket használ összetartozó adatok tárolására, amit Tuple-Tuple5 osztályok írnak definiálnak. A Flink tuple-ök legalább egy és legfeljebb öt objektumot tárolhatnak, melyek különbő típusúak lehetnek, illetve képesek módosítani és lekérdezni az adatmezőiket. Visszatérve, a Gelly a következő fontosabb gráftranszformációkat implementálja:

- Map, Translate: Végrehajt a gráfon egy felhasználói metódust, az id-kat változatlanul hagyja, a csúcsok vagy élek értékeit pedig a megadott metódus szerint lecseréli.
- Filter: Három filterfüggvénnyel rendelkezik. Közös bennük, hogy ha egy él vagy csúcs nem felel meg a szűrő feltételnek, nem jelenik meg az eredményben. A `filterOnEdges` a megadott feltétel szerint kiszűri az éleket és egy részgráfot hoz létre a megmaradt élekkel, de a csúcsokat változatlanul hagyja. A `filterOnVertices` az előzőhöz hasonlóan részgráfot hoz létre és a csúcsokra alkalmazza a szűrőfeltételt. A `subgraph` metódusnak az előző szűrők paraméterül adhatók, így az egyszerre tudja szűrni az éleket és a csúcsokat is.
- Join: A Gelly join függvényei az élek és csúcsok DataSet-jein végeznek join transzformációkat. A `joinWithVertices` a csúcsokat kapcsolja össze egy `Tuple2` bemenő adathalmazzal a csúcsok id-ja és a bemenő tuple-ök első mezője szerint. A transzformáció egy megadott felhasználói metódus által a csúcsok értékeit frissíti. A `joinWithEdges` hasonló logikát követ, de a join a két csúcs id-jából álló összetett kulcs által azonosítja az összetartozó párokat, és bemenetnek egy `Tuple3` adathalmazt vár. Ezt megkerülve a `joinWithEdgesOnSource` és `joinWithEdgesOnTarget` bemenetnek `Tuple2` halmazt vár és az éleket vagy a kezdő vagy a végcsúcsaik azonosítóin kapcsolja össze a kettesekkel. Az összes metódus új gráffal tér vissza.
- Reverse: Megfordítja a gráfban az élek irányítását és új gráffal tér vissza.
- Undirected: A Gelly csak irányított gráfokat támogat. Ugyanakkor egy irányítatlan gráf szimulálható, ha az összes létező él mellé behúzzuk ellentét, ami ugyanazon két csúcs között fut, de fordított irányítással. Így két csúcs között oda-vissza járható utat hoz létre, ami az irányítatlan gráfok lényege. Tehát a metódus behúzza a gráfban a megfelelő éleket, hogy az irányítatlan gráfként viselkedjen.
- Union: A Gelly képes két gráf éleinek és csúcsainak unióiból egy új gráfot készíteni.
- Difference: Két gráf él -és csúcshalmazának a különbségeit adja meg.

- Intersect: Két gráf éleinek metszetéből egy új gráfot készít. Két él akkor azonos, ha ugyanazok a csúcs azonosítóik és az értékük. Az új gráfban a csúcsok már nem rendelkeznek értékkel, de ezeket a `joinOnVertices` metódussal vissza lehet szerezni.

A Gelly arra is biztosít különböző beépített metódusokat, hogy módosítsuk egy letező gráf topológiáját. Egy gráfhoz hozzáadhatunk csúcsokat és éleket, vagy törölhetünk belőle. Ezeket gráfmutációs metódusnak(graph mutation) hívják és mindig új gráfpéldánnyal térnek vissza. További fontos funkció a szomszédossági eljárások (neighborhood methods). A szomszédsági függvények egy aggregátort és a feldolgozandó élek irányítását várják paraméterül, majd a csúcsok szomszédságán végeznek számításokat. Ezek a számítások minden csúcson egyszerre, párhuzamosan hajtódnak végre.

- `reduceOnEdges`: Egy kommutatív, asszociatív aggregátorfüggvényt vár, ami egy csúcs éleiből egyetlen értéket aggregál eredményül.
- `groupReduceOnNeighbors`: Szintén kommutatív és asszociatív aggregátort vár paraméterül, de a csúcsok szomszédos csúcsait képezi le egyetlen értékre.
- `groupReduceOnEdges`, `groupReduceOnNeighbors`: Ha az aggregátorfüggvény nem kommutatív vagy asszociatív, illetve ha egy csúcs esetén több értéket szeretnénk visszaadni, a `groupReduce` változat a megoldás. Ez a metódus szintén a csúcs szomszédos környezetét aggregálja, de nulla, egy vagy több kimenetet vár egy `Collector` nevű gyűjtőben.

A Gelly tartalmaz egy gráfvalidációs funkciót is, aminek van beépített és szabadon definiálható része is. A `Graph` osztály `validate` metódusa egy, a `GraphValidator` absztrakt osztály `validate` metódusát implementáló osztály vár paraméter, ami megadja, hogy egy gráf éleire és csúcsaira milyen feltételeknek kell teljesülnie. A metódusnak beépített validátorok is paraméterül adhatók és visszatérési értéke mindig `boolean` típusú. Végül a transzformációk, mutációk és validáción túl vannak beépített gráfalgoritmusok a legegyszerűbbektől kezdve a komplexebbekig. Ezek közül egy pár összetettebb algoritmus:

- Community Detection, Label Propagation: A gráf csúcsait közösségekre osztja összekötöttségi szempontok alapján.
- Connected Components: Gyenge komponensekre bontja a gráfot. Két csúcs egy komponensbe tartozik, ha az egyikből el lehet jutni a másikba élirányítástól függetlenül.
- PageRank: A PageRank algoritmust weboldalak fontosságának az osztályozására használják. Egy csúcs annál több pontot ér el, minél többen mutatnak felé, ugyanakkor a pontszám függ attól, hogy a felé mutató csúcs hány másik csúcs felé is mutat.
- Singe Source Shortest Paths: Egy kiválasztott csúcsból a többi csúcsba vezető legrövidebb utak meghatározása.
- Triangle Count: Megszámolja a gráfban a háromszögeket. Irányított gráfban azon csúcs -és élhármasok számítanak háromszögnek, melyek közül bármely csúcsból elindulva az élein keresztül visszajutunk oda ahonnan elindultunk. Az algoritmust implementálták irányított és irányítatlan gráfokra is. A Gelly gráffelépítése irányított, de a `getUndirected` metódussal az éleket megfelelően megduplázva elérhető az irányítatlan funkcionalitás.

Ezeket az algoritmusokat mind Scatter-Gather vagy Pregel típusú csúcscentrikus iterációkkal implementálták, mert összetettségük miatt könnyebb részfeladatokra bontatni őket, és intuitívabb ez a fajta megvalósítás.

3.2. Giraph

A Apache Giraph egy 2012-es nyílt forráskódú projekt gráffeldolgozásra, ami a Pregel modellen alapszik és Hadoop klaszterek felett fut. A Giraph-ot először a Yahoo fejlesztette, az Apache a projektet csak később vette át. Egyre népszerűbb gráffeldolgozó rendszernek számít, a Facebook a 2013-as Graph Search nevű keresőmotorját a Giraph segítségével készítette el. Az Apache Hadoop egy skálázható keretrendszer nagy adatmennyiség elosztott feldolgozására.

Az internetes alkalmazások növekvő használata a cégeket egyre nagyobb mennyiségű adat feldolgozására készteti. Az adatmennyiség nagymértékű növekedése miatt már nem lehetséges, vagy nagyon drága egy számítógépen és adattárolón végezni a felhasználói információk feldolgozását. Az Apache lehetővé tette, hogy olcsó hardverekből álló számítógépfürtökön az adatfeldolgozás master-slave architektúrában párhuzamosan történjen. Ez azt jelenti, hogy egy kitüntetett gép kezeli, a master, kezeli a kéréseket és továbbítja az akár több ezer slave gép felé. Mivel az adatok szét vannak osztva ezen gépek között, egyszerre, egymással párhuzamosan tudnak dolgozni, így gyorsítják a folyamatot. Tehát az Apache Hadoop-al nincs szükségünk a legdrágább hardverekere, mert több olcsó segítségével felülmúljuk a teljesítményt. [6] A Hadoop a nagymennyiségű adat hatékony, elosztott kezelését a MapReduce modell alapján implementálta és egy Java API-t is közzétett, amivel ún. MapReduce job-okat futtathatunk a keretrendszerben. A Hadoop megismerése fontos, mert a Giraph erre támaszkodik. Kifejezetten arra tervezték, hogy Hadoop klaszterek felett fusson és az algoritmusait, a Giraph jobokat, a Hadoop MapReduce job-okként futtassa. Mindemelett a Hadoop saját fájlrendszerrel is rendelkezik. A HDFS (Hadoop Distributed File System) egy hibatűrő és magas áteresztőképességű fájlrendszer adatok elosztott kezelésére tervezve. Ezt használja a Giraph a gráfok beolvasására és kiírására.

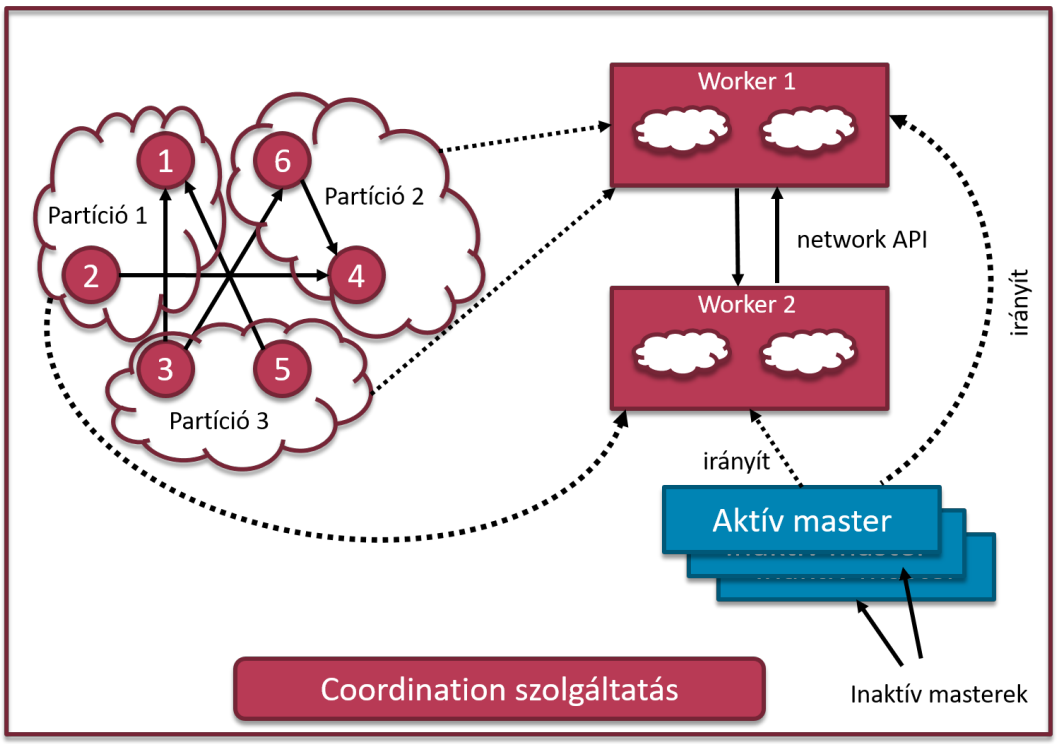
Ugyanakkor az Apache Giraph nem csak elosztottságot biztosít. A Pregel modellen (ld. 2.3.1) alapuló gráffeldolgozó keretrendszere elfedi az előbbieket és intuitív, kényelmes módot ad gráfalgoritmusok implementálására Java, Scala és Python nyelveken. A programozó feladata a saját algoritmusát a Giraph feltételei szerint leírni, a többi mögöttes, de fontos dologgal - mint az elosztottság és hibatűrés - nem szükséges foglalkoznia. A Giraph nem összekeverendő a gráfadatbázisokkal. A gráfadatbázisok célja a gyors lekérdezés. Ezzel ellentétben a Giraph komplex gráfalgoritmusok futtatására tervezték, nem a gyorsaság az elsődleges célja. A programozási modellt tekintve a Pregelt számos új funkcióval bővítették. A Giraph Zookeepert használ. Ez kezeli a konfigurációs adatokat, a szinkronizációt és a hibatűrést. Emellett új még a master computation számítás bevezetése, ami a superstepeket vezérli, és az él-orientált bemenet. Összefoglalva az Apache Giraph egy elosztott gráffeldolgozó rendszert biztosít, amit elfed egy kényelmes csúcs-centrikus gráffeldolgozási modellel.

3.2.1. A Giraph szolgáltatás-architektúra

A Hadoop hiába biztosítja a fájlrendszerszintű elosztottságot, a Giraph számításoknak szüksége van valamilyen belső logikára annak érdekében, hogy a gráf csúcsait és éleit particionálja, és ezeken a partíciókon az algoritmus futtatható legyen. Egy Giraph számítást háromféle szolgáltatás hajt végre: masterek, workerek és coordinatorok.

A csúcsok és élek partíciókba kerülnek párhuzamos feldolgozásra. A szolgáltatások részletes feladatai a következők:

- Master: Alapvetően ez a szolgáltatás koordinálja a worker-eket. Minden superstep kezdetén kiosztja a partíciókat, jelzi a superstep-ek végét és felügyeli a worker-ek állapotát. Egyszóval kezeli az alkalmazás életciklusát. Ő futtatja a



3.2. ábra. Az Apache Giraph szogáltatások architektúrája

`MasterCompute` algoritmust, amiről később lesz részletesebben szó, és ő implementálja a `MasterGraphPartititoner` interfészt, ami megmondja, hogy a superstep-ek elején milyen szabály alapján rendeli hozzá az egyes worker-hez a partíciókat. Egyszerre több master szolgáltatás fut a hibatűrés miatt. Ezek közül az egyik aktív, a többi inaktív állapotban van, és ha az aktív master meghibásodik, valamelyik átveszi a szerepét.

- Worker: A worker szolgáltatások hajtják végre a `compute` metódusban megírt felhasználói kódot a hozzájuk rendelt partícióban lévő csúcsokon. Képesek egymással egy, a Giraph-ba beépített hálózati API-n keresztül kommunikálni, egymás adait elérni és módosítani.
- Coordination: Ez a szolgáltatást az Apache Zookeeper[3] biztosítja, és konfigurációs adatokat, szinkronizációt illetve névtereket biztosít az alkalmazás számára. A worker statisztikák alapján a master ezen a szolgáltatások keresztül frissíti a partíciók kiosztását.

3.2.2. A Giraph telepítése

Mivel a Giraph Hadoop felett fut, érdemes megismerni a Hadoop telepítési lehetőségeket:

- Standalone: Ebben a módban a Hadoop egy gépen, egy Java alkamazásként fut. Ezt a módot használtam én is, mert megkönnyíti a telepítést és a konfigurációt. Emiatt általában fejlesztésre és tesztelésre használják.
- Pseudo-distributed: A pseudo-distributed mód egy gépen szimulálja az elosztottságot. Ebben az esetben a Hadoop-démonok külön Java folyamatokban futnak.

[3] `http://zookeeper.apache.org`

- Fully distributed: Ilyenkor a Hadoop egy számítógép klaszteren fut, teljesen elosztott módban. Éles környezetben ezt a módot használják.

A Giraph alapvetően parancssorban adja át a Hadoopnak a jobokat. A telepítést és a fejlesztést leegyszerűsíti, hogy a Giraph-ot Eclipse-ből és IntelliJ-ből is futtathatjuk, ha valamilyen buildelő keretrendszerrel (pl. Maven, Gradle) feloldjuk a `hadoop-core` és `giraph-core` dependenciákat.

3.2.3. Gráfok Giraph-ban

Az élek és csúcsok interfészei azonosak a Pregel modellben leírtakkal. Egy él rendelkezik egy értékkel és a végpontjának az azonosítójával. Egy csúcsnak van értéke, azonosítója, és tartalmazza a kimenő élei halmazát. Mivel ezek az interfészek generikusak, fontos, hogy az implementáció során az él interfész végpont azonosítójának és a csúcs interfész azonosítójának a típusa megegyezzen.

Egy élnek két metódusa van. A `getTargetVertexValue` viszaadja az él végpontjának azonosítóját, a `getValue` pedig az él értékét. Az csúcs interfész fontosabb metódusai a következők:

- `getId`: Visszaadja a csúcs azonosítóját.
- `getValue`: Visszaadja a csúcs értékét.
- `getNumEdges`: A csúcs kimenő élei számát adja vissza.
- `voteToHalt`: A számítás során ezzel a hívással a csúcs inaktív állapotba kerül és abban is marad amíg nem kap új üzenetet, véget nem ér a számítás, vagy fel nem ébresztik a `wakeUp` metódussal.
- `wakeUp`: Egy inaktív csúcsot aktív állapotba helyez.

Ezeken kívül még létezik metódus a csúcsok létrehozására és a gráf topológiájának a módosítására is.

3.2.4. A Computation interfész

A Computation interfész a Giraph számítások legalapvetőbb eleme, `compute` metódusában implementáljuk a gráfalgoritmusunkat, ami minden superstepben lefut. Az metódus paraméterei a csúcs és a bejövő üzenetek. Mivel egy csúcs implementálja a `Vertex` interfészt, eléri a kimenő éleit, vagyis a számításokat nem csak a csúcs értéke, hanem a kimenő élei és a kapott üzenetei alapján végezzük. Az kiszámolt eredmények a mindig a csúcs értékeként tároljuk el. A `compute`-on belül megkülönböztethetjük az egyes superstepeket, ha feltételeket alkalmazunk a superstep-ek számára, így meghatározhatjuk, hogy melyikben milyen algoritmus fusson. A `getSuperStep` függvény az aktuális superstep sorszámát adja vissza (nullától kezdődik). A Computation interfész legtöbbet használt metódusai:

- `getSuperstep`: Visszatér az aktuális superstep sorszámával.
- `getTotalNumEdges`: Megadja az összes él számát, ami az előző superstep-ben létezett. Ha az első superstep-ben hívjuk meg, -1-gyel tér vissza.
- `sendMessage`: Egy üzenetet küld egy csúcsnak. A csúcsazonosítót és az üzenetet paraméterben kapja meg.

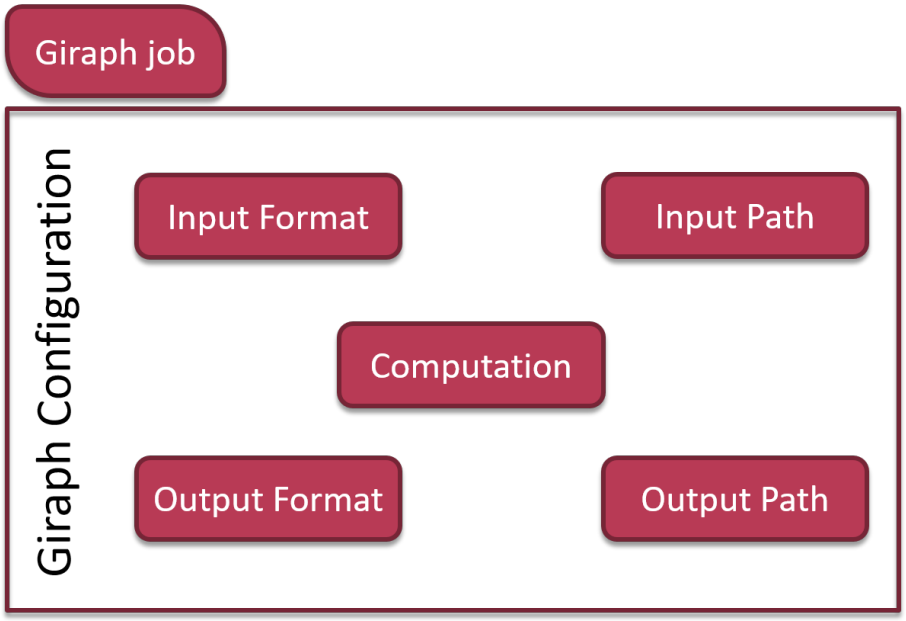
- `sendMessageToAllEdges`: Elküldi a csúcs összes kimenő élén a paraméterül kapott üzenetet.

Üzenetet küldhetünk a csúcsok egy halmazának is a `sendMessageToMultipleEdges` metódussal, illetve kérések intézhetünk a programhoz csúcsok és élek hozzáadásáról vagy törléséről. Ezek mindig csak a kérést követő superstep-ben hajtódnak végre. Ha számításokat szeretnénk végezni a `compute` függvény meghívódása előtt vagy után, a `preSuperstep` és `postSuperstep` rendelkezésünkre állnak, bár csak akkor hívódnak meg, ha a csúcs aktív állapotban van, tehát a `compute` is valóban meghívódik.

3.2.5. A Giraph job

Egy Giraph job hat dologból áll.

- Az alapja egy konfigurációs objektum, ami tartalmazza azokat a paramétereket, melyekre szüksége van a Giraph-nak a job futtatásához.
- A konfigurációnak tartalmaznia kell az algoritmust implementáló osztályt.
- A bemeneti és kimeneti elérési út megmondja a Giraph-nak, hogy honnan olvassa be a fájlt és miután lefutott a job hova írja ki.
- A bemeneti formátum specifikálja a fájl tartalmát, hogy milyen formátumban legyenek a csúcsok és élek leírva, illetve ennek a segítségével tudja a Giraph beolvasni a fájlt a megadott útvonalról, majd létrehozni a gráfot a memóriában.
- A kimeneti formátum megadja, hogy milyen formátumban írja ki a memóriából a kimeneti útvonalra a gráfot a job befejezése után.



3.3. ábra. A Giraph job felépítése

3.2.6. A Driver program

A driver program a master szolgáltatás része és egyfajta interfészt biztosít a Giraph job-ok és a Hadoop MapReduce job-ok között. A feladata, hogy egy Giraph konfigurációs objektum alapján létrehozza a job-ot, majd odaadja a Hadoop `JobTracker` szolgáltatásának futtatásra. Erre Java kódban többféle implementáció létezik, én `ToolRunner` osztályt használtam. A megvalósítás fázisai:

- Az osztálynak rendelkeznie kell egy `Configuration` Hadoop konfigurációs objektummal.
- A `ToolRunner` csak olyan osztályokból képes Giraph job-ot futtatni, melyek implementálják a `Tool` interfészt, ezért kötelepző annak az implementálása.
- A `Tool` interfész maga után vonja a `run` metódusának az implementálását is. A run metódusban egy `GiraphConfiguration` objektumot hozunk létre, és megadjuk a job beállításait. Például az algoritmust futtató osztály, a bemeneti és kimeneti formátum és elérési út, a `MasterCompute` osztály, vagy a workerek száma.
- Végül az osztály `main` metódusában a `ToolRunner` statikus osztály `run` metódusának paraméterül adva a job-ot tartalmazót osztályt és a parancssori paraméterek tömbjét, a job készen áll a futtatásra.

3.2.7. InputFormat: A bemeneti formátum

A Giraph `InputFormat` ősosztály a gráfok bemeneti formátumát írja le. Ezek nagyon specifikusak, mivel a Giraph a Hadoop fájlrendszeréről, a HDFS-ről olvas be. Alapvetően két fő formátum létezik a gráf leírására; a `VertexInputFormat`, és az `EdgeInputFormat` absztrakt osztályok által. Mindkét ősosztálynak vannak beépített, előre implementált leszármazottjai, melyeket használhatunk, de van egy fő különbség a kettő között. A `VertexInputFormat` a gráf éllistás leírása. Ez azt jelenti, hogy soronként a

$$\text{csúcs}_1 \text{ id},\ \text{csúcs}_1 \text{ érték},\ \text{él}_1 \text{ id},\ \text{él}_1 \text{ érték},\ \text{él}_2 \text{ id},\ \text{él}_2 \text{ érték}, ...$$

általános formátumot követi, de a leszármazott osztály szerint egyes mezőket, például az értékek nem szükséges leírni, vagy vessző helyett más írásjelekkel van tagolva a formátum. Az `EdgeInputFormat` az élekkel írja le a gráfot. Az általános formátuma a következő:

$$\text{csúcs}_i \text{ id},\ \text{csúcs}_j \text{ id},\ \text{él}_1 \text{ érték}$$

Természetesen ebben is lehetnek változások, például a `IntNullTextEdgeInputFormat` osztály nem vár értéket az élekhez.

3.2.8. MasterCompute és optimalizálás

A `MasterCompute` osztály feladata a superstep-ek koordinálása. Segítségével megadhatjuk, hogy a job melyik superstep-ben épp melyik `Computation` osztályt használja (ezekkel implementáljuk a gráfalgoritmusokat). A `MasterCompute` használatához le kell származtatnunk a `MasterCompute` vagy `DefaultMasterCompute` osztályokat és felülírnunk a `compute` függvényét. Ez a függvény teljesen hasonló a `Computation` osztály `compute` metódusához. Innen is elérjük a gráf adatait és módosíthatjuk azokat, üzeneteket küldhetünk, de nem csúcsonként, hanem az egészre gráfra nézve egyszer fut le. Mivel a `MasterCompute` kódja egy master szálon fut, az minden superstep előtt meghívódik, ezért innen kooridnálhatjuk, hogy

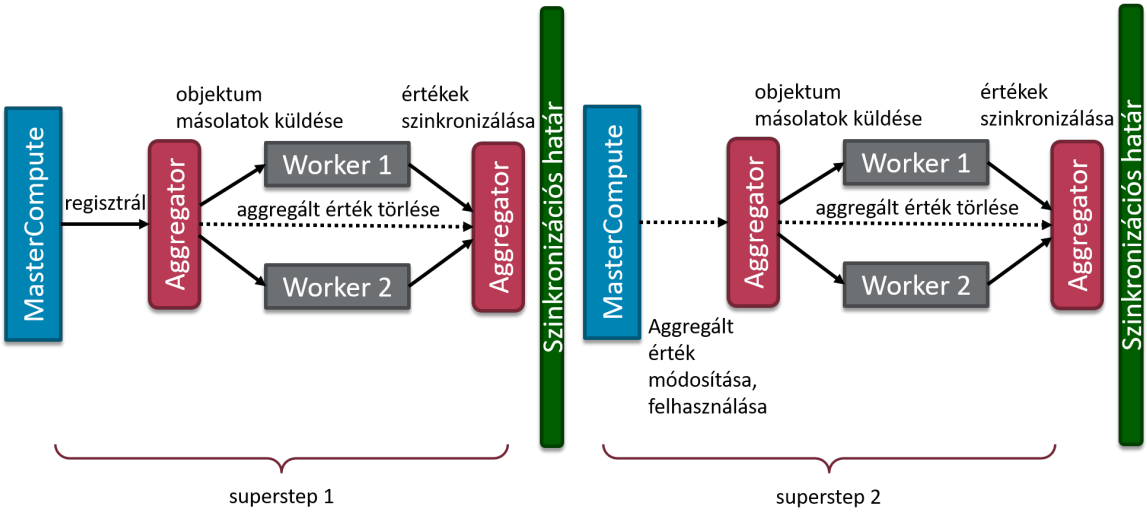
a workerek melyik lépésben melyik algoritmust futtassák. Ez azért hasznos, mert bonyolultabb algoritmusok implementálásakor több superstep-en át dolgozunk. Az egyes superstepek kódjait külön `Computation` osztályokba szervezhetjük a jobb átláthatóság érdekében, és a `MasterCompute` osztály használva beállíthatjuk, hogy az algoritmus melyik része melyik superstep-ben fusson. Egy job konfigurálása közben a `GiraphConfiguration` objektum `setMasterComputeClass` metódusával be kell állítanunk a job-hoz tartozó `MasterCompute` osztályt. További hasznos funkciója, hogy a `haltComputation` metódusával leállíthatjuk a számítást.

3.2.9. További optimalizációs technikák

A Giraph három technológiát támogat a master és a worker node-ok közötti kommunikációra: a broadcastot az aggregátort és a `ReduceOperation`-t. Ezek közül a két előbbit mutatom be részletesebben. A broadcast, aggregátor és `ReduceOperation` objektumokat a `MasterCompute` osztályban használat előtt egyedi `String` azonosítóval regisztrálni kell.

Broadcast: Abban az esetben, ha minden csúcsnak szeretnénk üzenetet küldeni, energiatakarékosabb megoldás, ha mindegyik tényleg csak egyszer kapja meg. Ha az üzenetet `Computation` osztályban küldenénk szét, mindegyik csúcs annyiszor kapná meg, ahány szomszédja van. A broadcastot a `MasterCompute` osztályban használhatjuk globális üzenetküldésre.

Aggregator: Az aggregátorfüggvények szintén globális kommunikációra valók, de nem csak a master, hanem a worker node-ok is elérhetik őket. Az aggregátor regisztrálása után



3.4. ábra. Az aggregátor működése

a master aggregátor objektumokat küld a worker node-oknak, akiknek olvasási az aktuális példányra. A worker-ek kezdeményezhetik az aggregátorérték változtatását, de mivel több van belőlük, ez a superstep végére szinkronizálódik. Az aggregátor begyűjti a változásokat és a belső logikája alapján kiszámol egy végértéket. A következő superstepben már ezt az értéket látják a worker-ek és a master is. Az aggregátorokkal erőforrást takaríthatunk meg. Tegyük fel, hogy meg szeretnénk találni a gráfban a legkisebb értékű csúcsot. Ezt aggregátor nélkül úgy lehetne implementálni, hogy egy a csúcsok elküldik a saját értékeiket a szomszédaiknak, akik kiválasztják a kapott értékek közül a legkisebbet. Ez addig folytatódik, amíg minden érték megegyezik a gráfban, tehát az algoritmus lefutása több

superstepből és rengeteg üzenetből állhat. Ehelyett ha létrehozunk egy aggregátort egyszerű minimumkiválasztó logikával, a csúcsok elküldik neki az értékeiket, az pedig egy superstep alatt kiválasztja a legkisebb értéket. Az aggregátoroknak két fajtája létezik: a perzisztens és a nem-perzisztens aggregátor. A különbség a kettő között, hogy míg előbbi minden superstep elején törli az értékét, az utóbbi a számítás végéig megtartja.

4. fejezet

Gráf metrikák

A fejezet bemutatja, hogy mik a gráf metrikák és miket hívunk annak, utána bemutatja a megvalósított gráf metrikákat, majd az egyes megvalósításokat az Apache Flink és Apache Giraph gráffeldolgozó rendszerekben.

4.1. Mit jelent a metrika?

Definíció 1. A metrikus tér egy olyan (X, d) rendezett pár, ahol X tetszőleges nemüres halmaz, $d : X^2 \to \mathbb{R}_0^+$ pedig olyan nemnegatív valós értékű függvény, melyre tetszőleges $x, y, z \in X$ esetén:

1. $d(x, y) \geq 0$ (pozitivitás)
2. $d(x, y) = 0 \iff x = y$ (egyenlőségi tulajdonság);
3. $d(x, y) = d(y, x)$ (szimmetria);
4. $d(x, z) \leq d(x, y) + d(y, z)$ (háromszög-egyenlőtlenség)

Ha (X, d) metrikus tér, X elemeit pontoknak, a d függvényt metrikának hívjuk. ▪

Definíció 2. Ha egy G gráf irányítatlan és összefüggő, továbbá V csúcsainak halmazán értelmezett a $d(x, y)$ függvény, ahol $x, y \in V$ és $d(x, y)$ eleget tesz a fenti három tulajdonságnak, akkor a gráfban a V csúcsok halmaza metrikus tér, a rajta értelmezett $d(x, y)$ függvény pedig metrika. ▪

Erre egy jó példa, ha $d(x, y)$ egy gráfokon értelmezett távolságfüggvény, amely az x és y csúcsok közötti legrövidebb távolságot adja meg, mert teljesül rá mind a szimmetriai, egyenlőségi, és háromszög-egyenlőtlenségi tulajdonság. Ez felveti a kérdést, hogy más, összetettebb számítások a gráfon metrikának számítanak-e, illetve mi a helyzet, ha a gráf nem összefüggő, nem irányítatlan, esetleg súlyozott. Nem összefüggő gráfok esetén két független csúcs között a $d(x, y)$ távolság vagy nem értelmezhető, vagy végtelen nagy. Az olyan metrikákat, amelyek felvehetnek végtelen nagy értéket, kiterjesztett metrikáknak [3] hívjuk. Ha a gráf irányított, nem feltétlen teljesül a metrikákra a szimmetriai tulajdonság, hiszen ha két pont között csak egy irányított él fut, a köztük lévő legrövidebb út csak egy irányból érhető el. Az ilyen metrikákat kvázi-metrikáknak hívjuk. Ha megengedjük a pozitív élsúlyokat, adódhat olyan eset, amikor nem teljesül a háromszög-egyenlőtlenség. Ezek az esetek a szemi-metrikák körébe tartoznak, melyek teljesítik az első három tulajdonságot kivéve az utolsót.

Látható, hogy ahhoz függvények és gráfok halmaza, melyek megfelelnek a metrikus tér és a metrika (vagy valamelyik módosított metrikafogalom) feltételeinek, igen szűk. Gráf

metrikák alatt általánosan a gráfot leíró tulajdonságokat értjük, melyek algoritmusokkal kiszámolhatók, és lehetnek kifejezetten topológiai, de egyéb vonatkozásúak is. Topológiai metrikák közé tartoznak azok, amik kifejezhetők a gráfok csúcsai közötti távolságokkal, tehát egy gráf szomszédsági mátrixából kiszámolhatók. Nem topológia metrikák, amik felhasználják a gráf egyéb tulajdonságait is, például a címkéket tulajdonsággráfok vizsgálat közben.

4.2. Előismeretek

A multidimenzionális gráfokat a tulajdonsággráfokra értelmezzük. Maga a multidimenzionális azt jelenti, hogy az élek több dimenzióval rendelkezhetnek, azaz több értékük lehet. Ezen értékek halmaza a gráf dimenziója. A metrikák definiálásához a következő fogalmakat és jelöléseket használom:

- Egydimenziós metrikák

  Legyen a G tulajdonsággráf csúcsainak és éleinek halmaza V és E. Legyen a csúcsok száma $v = |V|$ és az élek száma $e = |E|$. A gráf $v \in V$ csúcsnak fokszámát a be -és kimenő élek függvényében jelölje $d_{be}(v)$ és $d_{ki}(v)$. Az összesített fokszám legyen $d(v)$.

- Többdimenziós metrikák

  Legyen a G tulajdonsággráf csúcsainak és éleinek halmaza V és E, és legyen $d \in D$ G egy dimenziója, ahol D a dimenziók halmaza.

  Definíció 3. Összefüggőség: A v és w csúcs a d dimenzióban összefüggő, ha a köztük vezető valamilyen irányú él a d dimenzióban aktív és eleme a gráf élhalmazának.

  $$\text{Connected}(v, w, d) \iff (v, w, d) \in E \vee (w, v, d) \in E$$

  Definíció 4. Aktivitás

  $$\text{Active}(v, d) \iff \exists w \in W : \text{Connected}(w, v, d)$$

4.3. Az implementált metrikák

Egydimenziós metrikák:

- Kimenő fokszám: A kimenő fokszáma vagy kifoka egy csúcsnak a kifele irányuló éleinek száma. Jelölése: $d_{ki}(v)$
- Bejövő fokszám: A bejövő fokszáma vagy befoka egy csúcsnak a befele irányuló éleinek száma. Jelölése: $d_{be}(v)$
- Fokszám: A gráf egy csúcsának a fokszáma a gráf azon éleinek száma, melyek illeszkednek az adott csúcsra. Jelölése $d(v)$.
- Lokális klaszterezettségi együttható: A gráf lokális klaszterezettsége megméri, hogy egy csúcs szomszédai milyen mértékben összekötöttek egymással, vagyis az általuk feszített részgráf mennyiben tér el a teljes gráftól. A topológiai metrikák között az átlagos távolság és a fokszámeloszlás mellett az egyik legfontosabbnak számít. Jelölje a $G(V, E)$ gráfban e_{ij} a v_i és v_j csúcsokat összekötő élt. Ekkor a v_i csúcs szomszédainak halmaza $N_i = \{v_j : e_{ij} \in E \vee e_{ji} \in E\}$. Tehát N_i olyan csúcsok halmaza, melyek

és a v_i csúcs között létezik él. Az e_{ij} és e_{ji} megkülönböztetés irányított gráfokra érvényes. A lokális klaszterezettségi együttható a szomszédos csúcsok között futó és lehetséges élek aránya. Irányítatlan gráfok esetén:

$$C_i = \frac{2 \cdot |\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)},$$

ahol $k_i = |N_i|$. A metrika irányított gráfok esetén a számlálóban lévő kettes szorzóban különbözik, ugyanis ebben az esetben a szomszédok közötti lehetséges élek duplája az előzőnek.

$$C_i = \frac{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}$$

Multidimenzionális metrikák:

- Dimensional degree: A $v \in V$ csúcs $d \in D$ dimenziójú éleinek száma.

  $$\text{Degree}(v, d) = |\{w \in V | \text{Connected}(v, w, d)\}|$$

  A metrika dimenziók halmazára is értelmezhető.

- Node dimension activity (NDA): Megadja, hogy a d dimenzióban hány aktív csúcs van. Azok a csúcsok aktívak, amelyek rendelkeznek d dimenziójú éllel.

  $$\text{NDA}(d) = |\{v \in V | \text{Active}(v, d)\}|$$

- Node dimension connectivity (NDC): Kiszámolja egy adott dimenzióban aktív csúcsok arányát az összes dimenzióhoz képest. Feltesszük, hogy mindegyik él rendelkezik dimenzióval, így az összes dimenzióban aktív csúcsok száma megegyezik a gráf csúcsainak számával.

  $$\text{NDC}(d) = \frac{\text{NDA}(d)}{|V|}$$

- Node exclusive dimension activity (NEDC): Azon csúcsok száma, melyek kizárólag a megadott dimenzióhoz tartoznak, azaz csak egyféle dimenziójú élük lehet.

  $$\text{NEDC}(d) = \frac{|v \in V | \text{Active}(v, d) \wedge \neg\text{Active}(v, \{D \setminus \{d\}\})|}{|V|}$$

- Edge dimension activity (EDA): Megadja azon élek számát, melyek a d dimenzióban aktívak.

  $$\text{EDA}(d) = |\{(v, w, d) \in E | v, w \in V\}|$$

- Edge dimension connectivity (EDC): A d dimenziójú élek aránya az összes élhez képest:

  $$\text{EDC}(d) = \frac{\text{EDA}(d)}{|E|}$$

- Node activity (NA): Azon dimenziók száma, melyben a v csúcs aktív.

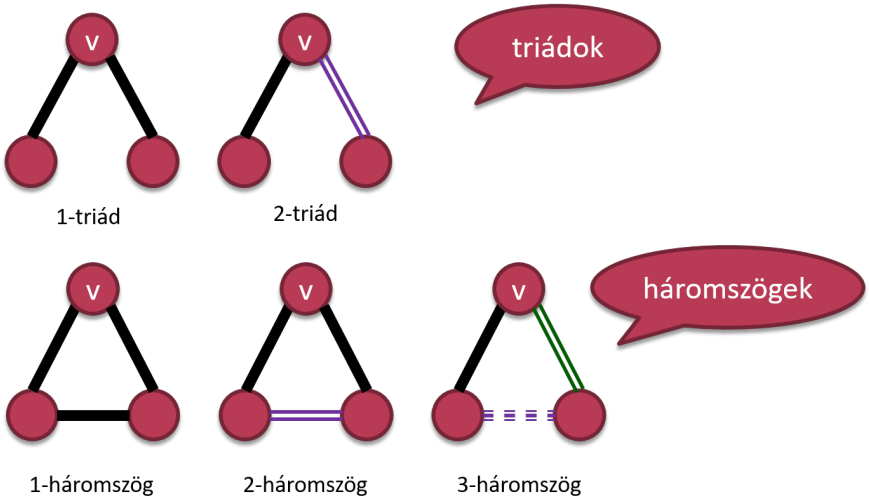
  $$\text{NA}(v) = |\{d \in D | \text{Active}(v, d)\}|$$

- Multiplex Participation Coefficient (MPC): Megméri, hogy a v csúcs dimenziói mennyire egyenletesen oszlanak el a D dimeziók között.

$$\mathrm{MPC}(v) = \frac{|D|}{|D|-1}\left[1 - \sum_{d \in D}\left(\frac{\mathrm{Degree}(v,d)}{\mathrm{Degree}(c,D)}\right)^2\right]$$

  Eredménye egy 0 és 1 közötti érték, amely 0 ha v összes éle egy dimenzióba tartozik, és 1 ha v élei mind különböző dimenziójúak.

- Dimensional clustering coefficient (DC): Egy v csúcsra értelmezve méri a multidimenzionális háromszögek arányát a gráfban. Hívjuk triádoknak azokat a csonka háromszögeket, melyek egyik pontja v, két másik pontja és v között vezet él, de nincs olyan éle, ami v-re nem illeszkedik. Ezek közül legyenek 1-triádok azok a triádok, amelyek élei egy dimenziósak és 2-triádok, amelyek több dimenzióval rendelkeznek. A gráfban háromszögnek nevezünk három olyan csúcsot, hogy bármely kettő között fut él. Ezek közül dimenziók szerint szintén vannak 1-háromszögek, 2-háromszögek és 3-háromszögek. Ezeket felhasználva két féle DC metrikát különböztetünk meg.



4.1. ábra. Triádok és háromszögek

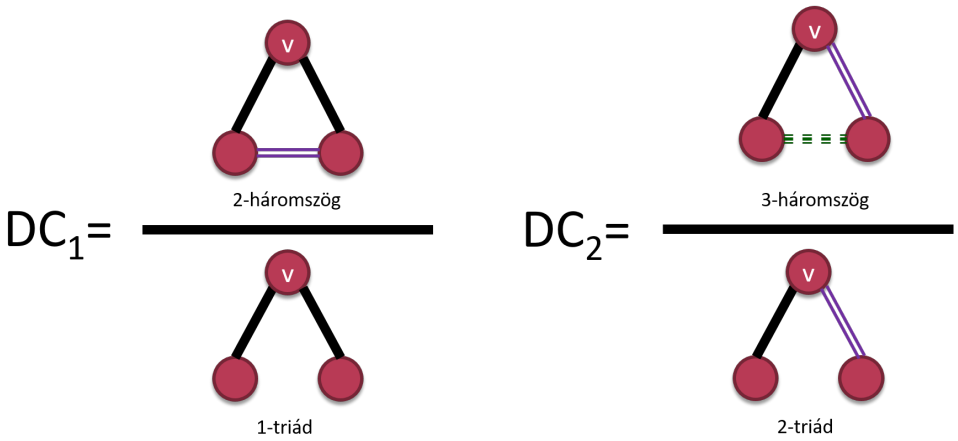
  A DC_1 metrika a v csúcs középpontú 2-háromszögek és 1-triádok, a DC_2 metrika pedig a 3-háromszögek és 2-triádok aránya. A v középpont a színezést illetően a 2-háromszögek esetén számít, ahol ahol az eltérő színű él a v-vel szemközti kell legyen, ahogy az alábbi képen látható.

4.4. Metrikák Flinkben

Ez az alfejezet részletesen bemutatja a 4.3 fejezetben tárgyalt metrikák Flink-beli implementációját.

- Be -és kimenő fokszám: A `Graph` osztály beépített `inDegrees` és `outDegrees` metódusait használtam.

- Átlagos fokszám: Mivel egy gráfban a fokszámok összege megegyezik az élek számának kétszeresével, a `(double) graph.numberOfEdges() * 2 / graph.numberOfVertices()` formulával ez könnyen kiszámítható.



4.2. ábra. Dimensional clustering metrikák

- Lokális klaszterezettségi együttható (LCC): A gráf beépített LCC metódusát először példányosítani kell, majd a `Graph` osztály `run` függvényének paraméterül adva egy `GraphAnalytic` objektumban tárolja el az algoritmus eredményét.

- Dimensional degree: A metrika kiszámolásához a `Graph` osztály `groupReduceOnEdges` metódusát használtam, ami csúcsok szerint csoportosít és `GroupReduce` transzformációt hajt végre az egyes csúcsok élein. A `GroupReduceOnEdges` paraméterül egy `EdgesFunction` vagy `NeighborsFunction` interfészt implementáló osztályt vár. Ezeknek az osztályoknak az `iterateEdges` vagy `iterateNeighbors` függvényét felülírva implementálhatjuk a `GroupReduce` transzformációt. A fő különbség a kettő között ezen függvény paraméterezésében áll. Az `EdgesFunction` interfész `iterateEdges` metódusa a csúcsok élein, a `NeighborsFunction` interfészé a csúcsok szomszédos csúcsain iterál végig. A két interfésznek van egy-egy további változata is: a `EdgesFunctionWithVertexValue` és a `NeighborsFunctionWithVertexValue`, amelyek iterációs metódusában extra paraméterként az adott csúcs is elérhető. Általánosan egy `GroupReduce` transzformációt megvalósító osztály iterációs függvényének kettő vagy három paramétere van. A csúcs, aminek éppen az élein vagy a szomszédain iterálunk, amennyiben a csúcsértékkel kiegészített interfészeket implementáljuk, a csúcshoz tartozó élek iterálható halmaza, és egy ún. generikus kollektor, ami csúcsonként egy kimeneti eredményt tárol el. A `DimensionalDegree` osztály implementálja az `EdgesFunctionWithVertexValue` interfészt és az `iterateEdges` függvényét. Rendelkezik egy `id` és egy `List<DimensionType>` tagváltozóval, melyekkel eldönthetjük, hogy egy csúcs vagy az összes csúcs dimenzionális fokára és melyik dimenzióra vagyunk kíváncsiak. Az `iterateEdges` metódus megvizsgálja, hogy a konstruktorban megadott `id` paraméter értéke `null`-e. Ha igen, akkor az összes csúcsra, egyébként csak a megadott `id`-jú csúcsra számolja ki a metrikát. Az algoritmus ezután végigiterál az adott csúcs élein és megnézi, hogy az egyes élek értékei megegyeznek-e az osztály konstruktorában megadott dimenzió típussal. Ha igen, egy cikluson kívüli változóban számolja.

- Node dimension activity (NDA): A metrika azt számolja meg, hogy hány aktív csúcs van egy megadott dimenzióban. Szintén `GroupReduce` transzformációt használtam az `EdgesFunctionWithVertexValue` interfésszel, melyet a `NodeDimensionActivity` osztály implementál. Az osztály a konstruktorában pontosan egy dimenzió típust vár. Az `iterateEdges` metódusában végigiterál a csúcs élein, összehasonlítja az él dimen-

zióját a konstruktorban megadottal, ha egyezést talál, egy cikluson kívüli `boolean isActive` változót igazra állít és megszakítja a ciklust. Ezután a kollektor az aktuális csúcsot és az `isActive` változó értéke szerinti 0-ból vagy 1-ből álló `Tuple2` objektumot kap eredményül. Az NDA metrika futtatása egy `DataSet` objektumot vissza. Ezt a második mezője, az értékek szerint összegezve (`dataSet.sum(1).collect`), megkapjuk összesen hány csúcs aktív a megadott dimenzióban.

- Node dimension connectivity (NDC): Az implementáláshoz a már meglévő NDA metrikát használtam. Ezt kellett elosztani a gráf csúcsainak számával.

- Node exclusive dimension connectivity (NEDC): Az előzőkhöz hasonlóan `GroupReduce` transzformációt használtam. A különbség az `iterateEdges` metódusban van. A függvény végigiterál egy csúcs élein és ellenőrzi, hogy mindegyik éle a konstruktorban megadott dimenzióba tartozik-e. Ha talál más dimenziójú élt, az `isExclusive` boolean változót hamisra állítja és kilép a ciklusból. A kollektor egy csúcs azonosító és egy számból álló `Tuple2`-t kap, ahol a szám az `isExclusive` értéke alapján 0 vagy 1 lehet.

- Edge dimension acitvity (EDA): Az éldimenziós aktivitás kiszámolásához a `Graph` osztály `filterOnEdges` metódusát használtam, ami paraméterül egy a `FilterFunction` interfészt megvalósító osztályt vár. Ennek az osztálynak az interfész `filter` metódusát szükséges implementálni, amivel definiálja a szűrő logikát. Az `EdgeDimensionActivityFilter` osztályt használtam a szűrő implementálására. Az osztály ezen kívül rendelkezik egy `DimensionType` tagváltozóval, amit a konstruktorban lehet beállítani. A `filter` metódus paraméter egy él, a visszatérése egy boolean típus. A függvény törzsében megvizsgálom, hogy az él dimenziója egyezik-e a konstruktorban megadotta. Eszerint igaz vagy hamis értéket ad a szűrő. A metrika futtatása egy új gráf objektumot ad vissza, amelyben a csúcsok változatlanok, az élek pedig a filter osztály szerint szűrtek. Az életekt összeszámolva - `edgeFilteredGraph.getEdges().count()` - megkapjuk az eredményt.

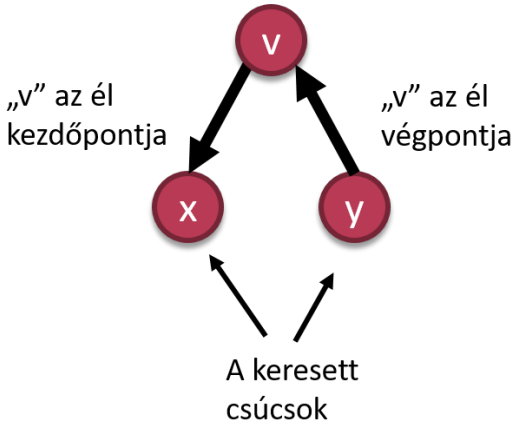
- Edge dimension connectivity (EDC): Itt az EDA filterét használtam. A különbség, hogy az eredményt el kell osztani a gráf éleinek számával: `(double) edgeFilteredGraph.getEdges().count() / originalGraph.getEdges().count()`.

- Node Activity (NA): A node activity metrika kiszámításához újra a `Graph` osztály `groupReduceOnEdges` metódusát és az `EdgesFunctionWithVertexValue` interfészt implementáló `NodeActivity` osztályt használtam. A `NodeActivity` osztály a konstruktorában egy csúcs azonosítót kap. Az `iterateEdges` metódusa először megvizsgálja, hogy éppen a konstruktorban megadott id-jú csúcsnál tartunk-e. Ha igen, létrehoz egy `distinctEdgeDimensions` nevű `String` típusú halmazt. Ezután végigiterál az aktuális csúcs élein, és ha a halmaz még nem tartalmazza az iterált él értékét, hozzáadja. A kollektor megkapja a csúcs id-ját és a halmaz méretét. A halmaz mérete jelzi, hogy hány különböző dimenziójú éllel rendelkezik a hozzátartozó csúcs.

- Multiplex participation coeffiecient (MPC): A metrika használja a `dimensionalDegree(Integer id, List<DimensionType> dimensions)` metódust, amit korábban bemutattam. A dimenziók számát a `DimensionType` enum osztályból nyeri ki a `DimensionType.value().length` metódussal. Utána végigiterál a dimenziókon és a `dimensionalDegree`-t használva kiszámolja a szumma

utáni részt. Végül behelyettesít a képletbe és a szabványos kimenetre kiírja a végeredményt.

- Dimensional clustering 1: A DC metrikák implementálásához több segédosztályt is használtam. A `EdgeUtils` segédosztály a gráf beolvasása után létrehoz egy statikus `Map`-et, amelynek kulcsai a csúcsok azonosítói, értékei a csúcsokhoz tartozó kimenő éllisták. Két fontos metódusa van: a `getEdgeMap` visszaadja a csúcsok szerinti éleket tároló `Map`-et, a `getEdgeMap` paraméterül egy csúcs id-t vár, és a megadott id-hoz tartozó éllistát adja vissza. A másik egy `Fraction` nevű osztály, amiben törteket lehet tárolni számláló és nevező alakban, és a `result` metódusa kiszámolja a tört értékét. Ha a számláló és nevező értéke is nulla, a `result` értéke is nulla lesz. A metrika osztálya, a `DC1Metrics` implementálja az `EdgesFunctionWithVertexValue` interfészt és két metódussal rendelkezik. Az egyik az interfésztől örökölt `iterateEdges`, a másik a `getTriadCloserEdgeLabels` segédfüggvény. Az utóbbi sorban két élt és egy csúcsot vár paraméterül, úgy hogy a két élnek illeszkednie kell a csúcsra, majd kiszámolja a megadott csúcs azon szomszédai közötti élek dimenzióinak számát, melyek szintén illeszkednek a paraméterként kapott két élre, tehát bezárják a csúcsból és a két élből álló triádot. Mivel nem ismerjük az élek irányítását, nem tudjuk, hogy az élek kezdő vagy végpontjához tartozó csúcs a megadott szomszédja. A metódus megvizsgálja, hogy a paraméterként kapott élek kezdőpontjának vagy végpontjának az id-ja egyezik meg a paraméter csúcs id-jával, majd kiválasztja a másik csúcsot és eltárolja. Ha



4.3. ábra. A megfelelő csúcsok keresése

megtalálta a két csúcsot, az `EdgeUtils` osztály `getEdgeList` metódusával elkéri az egyik csúcshoz tartozó élek listáját. Végigiterál az éllistán és megvizsgálja, hogy az aktuális él végpontja a másik csúcs-e. Amennyiben igen, ez azt jelenti, hogy talált egy, a két csúcsra illeszkedő élt, de azt is meg kell vizsgálnia, hogy a triádot háromszöggé záró él dimenziója ne legyen azonos a triád szárát alkotó éle dimenziójával. Ha ez megtörtént, eltárolja az él értékét egy listában, és végül a listával tér vissza.

Az `iterate` metódus (az interfésztől jön) paraméterként mindig egy csúcsot, az éleit és egy kollektort kap. Duplán végigiterál az aktuális csúcs élein, és megvizsgálja, hogy a két iterált él dimeziója egyezik-e. Ha igen, elkéri a `triadCloserEdgeLabels`-től az triádot bezáró élek listáját. Amennyiben a lista mérete nulla, egy tört objektum nevezőjét növeli eggyel, egyébként a tört objektum számlálóját és nevezőjét a lista méretével. A kollektor az aktuális csúcs azonosítóját és a hozzátartozó tört értékét tárolja el egy `Tuple2` objektumban.

- Dimensional Clustering 2: A metrika teljesen a DC1 megoldási menetét követi. A különbség abban rejlik, hogy itt a DC1-gyel ellentétben 2-háromszögek helyett 3-háromszögek és 1-triádok helyett 2-triádok arányát mérjük, így a `DC2Metrics` osztály `iterateEdges` metódusában az élek dupla iterációjában azokat kell az élpárokat kell vizsgálni, amelyek dimenziói különböznek.

4.5. Metrikák Giraph-ban

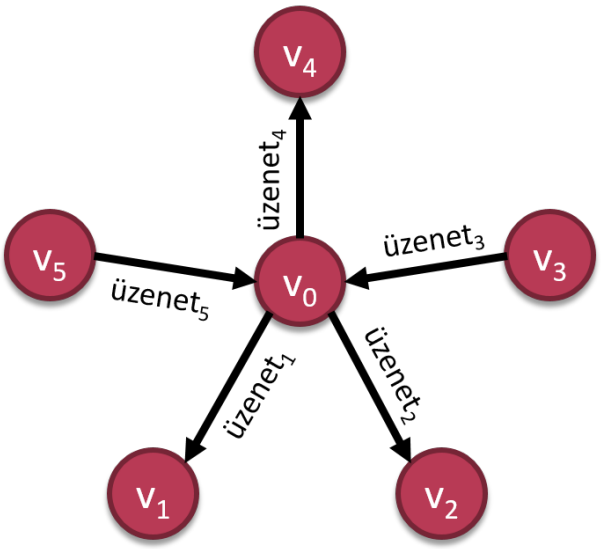
Ez a fejezet a metrikák Apache Giraph keretrendszerbeli implementációját mutatja be.

- Kimenő fokszám: A metrika az `OutDegree` osztályban került implementálásra, amely a `BasicComputation` absztrakt osztály leszármazottja. Az felülírandó metódus - `compute` - két paramétert kap; egy csúcsot és az előző superstepben kapott üzeneteket. Fontos, hogy itt egy csúcs szemszögéből írjuk le a csúcsokat. A `compute` metódus nem csak azt mondja meg, hogy egy, hanem hogy az összes csúcs mit csináljon. Az első (valójában nulladik) superstepben a paraméter `vertex` objektum `setValue` metódusát használva a `compute` beállítja a csúcs értékének önmaga élei számát. Az élek száma a csúcs `getNumEdges` függvénnyel érhető el, és ezt kapja paraméterül a `setValue` metódus. Ezután a `vertex.setValue()` metódussal a csúcs leállásra szavaz. Tehát egy superstep alatt az összes csúcs beállította értékül az éleinek számát, majd passzív állapotba került, így az algoritmus véget ért. A `OutDegree`-ből a `GiraphAppRunner` driver oszály készít egy Giraph jobot és elküldi végrehajtásra. A job kimeneti formátuma határozza meg, hogy mi az algoritmus eredménye. Minden metrika implementálásánál az `IdWithValueTextOutputFormat OutputFormat` osztályt használtam, ami a kimeneti útvonalra egy fájlba kiírja a csúcsok azonosítóit és a hozzájuk tartozó csúcs értékeket.

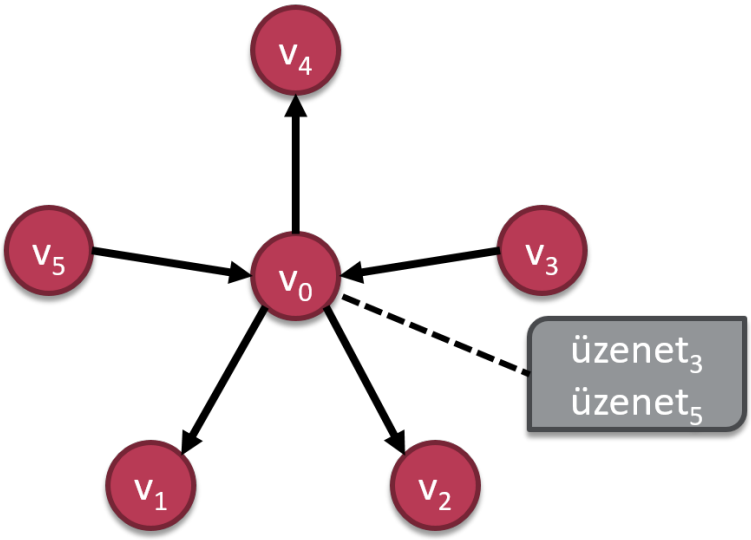
- Bemenő fokszám: Az metrika két superstepből áll és a `InDegree` osztály valósítja meg. Az algoritmus egyszerűsége miatt nem használtam `MasterCompute` osztályt a superstepek kezelésére, hanem az `OutDegree compute` metódusában a `getSuperstep`el vizsgáltam, hogy éppen melyikben superstepben vagyunk. A nulladik superstepben a csúcsok üzenetet küldenek a kimenő éleiken a szomszédaiknak a `sendMessageToAllEdges` metódussal, amely első paraméterként a küldő csúcs objektumot, másodikként az üzenetet várja. Az üzenetet értéke ebben az esetben nem számít, mert a következő superstepben nem kerül feldolgozásra. A típusa az ősosztály generikus paraméterében adjuk meg, ami jelen esetben `LongWritable`. Az első superstepben a csúcsok összeszámolják a kapott üzeneteket és az eredményt beállítják értékül a `setValue` metódussal, majd leállásra szavaznak.

  A 4.4 ábrán az algoritmus a v_0 csúcs szemszögéből mutatja a nulladik superstepbeli történéseket. A $v1$, v_2 és v_4 csúcsoknak üzenetet küld a kimenő élein, a v_3 és v_5 csúcsok pedig neki küldenek üzenetet. A következő superstep-ben a csúcsok megkapják az üzeneteket, a 4.5 ábrán látható a v_0 csúcs v_3 és v_5-től kapott üzenete. Az üzenetek száma megegyezik a bemenő fokszámmal.

- Lokális klaszterezettségi együttható: A metrika implementálása során három segédosztályt használtam. A `LongArrayListWritable` a Giraph `ArrayListWritable` leszarámazottja és `LongWritable` objektumokból álló listát implementál az ezeket tartalmazó üzenetek küldéséhez. Az üzenetek és a `Computation` osztály egyéb generikus paraméterei (csúcsok, élek típusa) valamilyen módon mindig implementálják a Hadoop `Writable` interfészét, annak érdekében, hogy a Giraph jobok kompatibilisek legyen a Hadoop `MapReduce` jobokkal. A második `MessageWrapper` szerepe, hogy egy-



4.4. ábra. Kimenő fokszám számítása: Nulladik superstep



4.5. ábra. Kimenő fokszám számítása: Első superstep

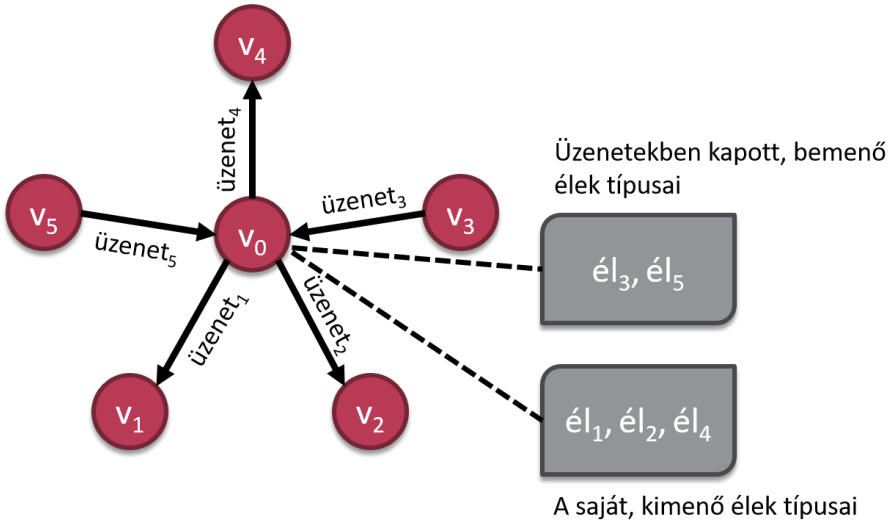
részt egységes formába foglalja az üzeneteket, másrészt tárolja a küldő csúcs azonosítóját és az él értékét, amelyen az üzenet érkezett. A harmadik `LongIdFriendsList` segédosztály a generikus `MessageWrapper` leszármazottja és beállítja és beállítja az üzenet feladójának az azonosítótípusát, az üzenet típusát és az élek értékének típusát.

Az algoritmus főosztálya a `LocalClusteringCoefficient` osztály, amely öt statikus `BasicComputation` és egy `MasterCompute` leszármazottat tartalmaz. Az egyes `BasicComputation` leszármazottak implementálják az algoritmust részeit superstepek szerint, a `MasterCompute` pedig megmondja, hogy melyik superstep-ben melyik belső algoritmus osztály fusson.

- Dimensional degree: A `MessageWithSenderAndEdgeType` egy `MessageWrapper` leszármazott, segédosztály és az üzenetek típusa. Segítségével `Text` típusú üzeneteket küldhetünk, amelyek megegyeznek a csúcsok értékének típusával, így kiküszöböli a kasztolást az értékek beállításakor. A `DimensionalDegree` - az algoritmust imple-

mentáló osztály - két statikus belső osztályban két superstep-re bontja a metrikát. A `InOutEdgesComputation` belső osztály a `compute` metódusában végigiterál a csúcs élein. Az iteráció közben létrehoz egy `MessageWithSenderAndEdgeType` objektumot, amelynek beállítja a forrását a csúcs azonosítójára, az üzenetet a csúcs értékére, az él típusát pedig egy üres `Text` objektumra, mert az nem lényeges. A `sendMessage` metódust az él végpontjával és az aktuális `MessageWithSenderAndEdgeType` objektummal felparaméterezve elküldi az üzenetet. A `DimensionalDegreeComputation` osztály a `compute` metódusában végigiterál a csúcs élein és a kapott üzeneteken, miközben megvizsgálja, hogy az él típusok illetve az üzenetben kapott él típusok (amelyik élen jött az üzenet) megegyeznek-e a megadott dimenzióval.

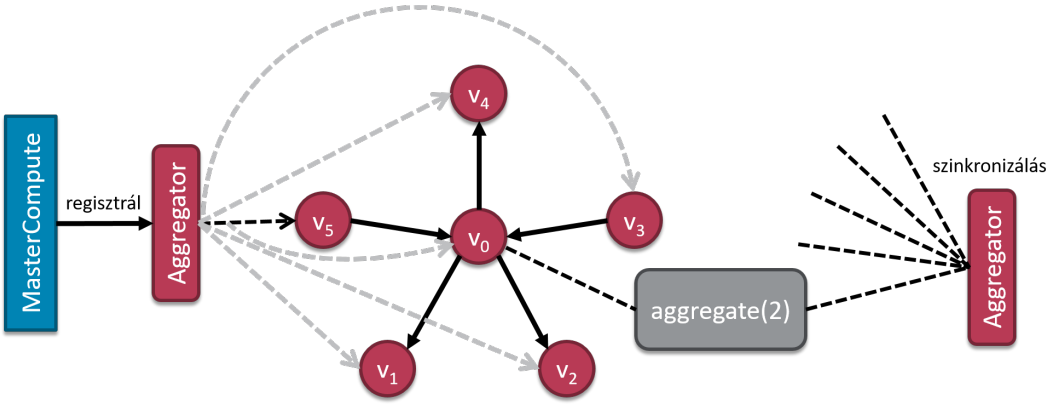
Itt probléma volt, hogy a dimenzió típusát nem tudtam paraméterül adni a `DimensionalDegree` osztálynak, mivel a Giraph job létrehozásakor nem példányosítjuk az algoritmust implementáló osztályt, hanem a rá vonatkozó `class` értéket adjuk meg. Jobb lehetőség hiányában a dimenziót a `Computation` osztályban lehet beállítani.



4.6. ábra. Dimensional degree számítása: Első superstep

Az összeszámolt dimenziókat a csúcs beállítja saját értékének majd leállásra szavaz.

- Node dimension activity (NDA): A metrika a megadott dimenzióban aktív csúcsok számát méri. Az ötlet az volt, hogy ha egy csúcs kimenő élein iterálunk végig, mindegyik csúcs csak a kimenő élein iterál végig, így minden élt egyszer számolunk. A `NodeDimensionActivity` osztályban a `NodeDimensionActivityComputation` statikus belső osztály implementálja a metrikát a `BasicComputation`-től örökölt `compute` metódusában. A `MasterCompute` osztály az `initialize` metódusában egy `LongSumAggregator`-t regisztrál, de a superstep-eket nem irányítja, mert az algoritmus egyszerűsége nem követelte meg külön belső osztályok létrehozását. A `compute` metódus a nulladik superstep-ben a csúcs élein, és ha az él értéke megegyezik a megadott dimenzióval, az `aggregate` metódusával egy 2 értékű `LongWritable` objektumot aggregál. Azért 2-es értéket küldd az aggregátornak, mert egy élhez két csúcs tartozik, és mindkettőbe beleszámít az él dimenziója, vagy kimenő vagy bemenő élként. Az első superstep-ben a csúcsok beállítják az értéküket az aggregátor már szinkronizált értékére, majd a harmadik superstep-ben a `MasterCompute` osztály `compute` függvénye leállítja az algoritmust a `haltComputation` hívással. A 4.7 ábra



4.7. ábra. Node dimension activity számítása: Nulladik superstep

mutatja, ahogy a `MasterCompute` szétküldi az aggregátor másolatokat a csúcsoknak, akik kérést küldenek a módosításra, majd az aggregátor szinkronizálja a kéréseket.

- Node dimension connectivity (NDC): Az NDC metrikát a `NodeDimensionConnectivity` osztály implementálja. Az NDA-hoz hasonlóan `LongSumAggregator` aggregátort használ, de az első superstep-ben, amikor a csúcsok beállítják az értéküket a szinkronizált aggregátor értékére, elosztják az a csúcsok számával. A csúcsok száma a `BasicComputation` örökölt `getTotalNumVertices` függvényével érhető el az algoritmust implementáló `NodeDimensionConnectivityComputation` osztályban.
- Node exclusive dimension connectivity (NEDC):
- Edge dimension activity (EDA): Az NDA és EDA kiszámolása között egyetlen különbség van. Utóbbi az aggregátornak 1 értékű `LongWritable` objektumot ad, mivel ebben az esetben az élek dimenzióbeli aktivitását vizsgáljuk, nem a csúcsokét.
- Edge dimension connectivity (EDC): A metrikát az NDC-hez hasonlóan implementálja a `EdgeDimensionConnectivity` és `EdgeDimensionConnectivityComputation` osztály. Az első superstep-ben az aggregátortól a `getAggregatedValue` metódussal elkért értéket elosztja az élek számával. Az élek száma a `BasicComputation` leszármazott osztályban elérhető a `getTotalNumEdges` metódussal.
- Node activity (NA): Azon dimenziók számát kell megtalálnunk, melyben az adott csúcs aktív. A `NodeActivity` osztály a `SendEdgeValues` és `NodeActivityComputation BasicComputation` leszármazottakkal két superstepben implementálja a metrikát. A `SendEdgeValues compute` metódusa végigiterál a csúcsok élein, és a kimenő éleken elküldi szomszédjainak az él értékét egy üzenetben. A `NodeActivityComputation compute` metódusa ezután létrehoz egy `DimensionType` halmazt, végigiterál a kapott üzeneteken, majd a csúcs sjaát élein, és berakja az értékeket a halmazba. Végül a csúcs beállítja az értéket a halmaz méretére és leállásra szavaz. A halmaz kiszűri az ugyanolyan dimenziókat, ezért a mérete a csúcshoz tartozó különböző dimenziójú élek száma, ami a megoldás.
- Multiplex participation coefficient (MPC): A `MultiplexParticipationCoefficient` osztály a `SendEdgeValues` és `MultiplexParticipationCoefficientComputation BasicComputation` leszármazottakkal implementálja a metrikát. Az előbbi belső osztály a node activity-hez hasonlóan szétküldi a csúcsok éleinek értékét a kimenő

éleken. Az MPC számítása a 4.3 fejezetben leírtaknak megfelelően a következő formulával történik:

$$MPC(v) = \frac{|D|}{|D|-1}\left[1 - \sum_{d \in D}\left(\frac{Degree(v,d)}{Degree(v,D)}\right)^2\right]$$

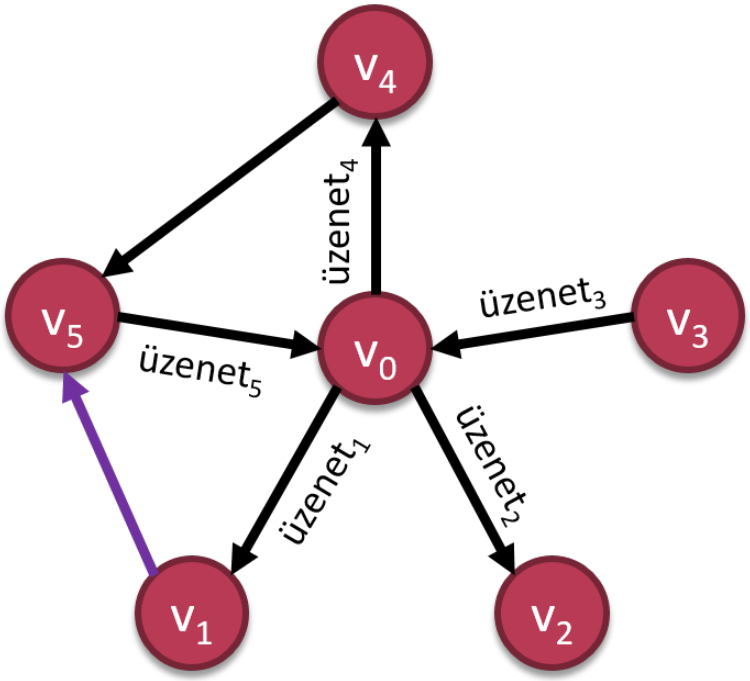
A szumma utáni $Degree(v, D)$ a v csúcs dimenzionális fokszámát jelenti az összes dimenzióra nézve. Ez az eredeti definíció szerint

$$Degree(v,d) = |\{w \in V | Connected(v,w,D)\}|,$$

vagyis az csúcs éleinek száma. A `MultiplexParticipationCoefficientComputation` osztály `computation` függvénye a csúcs kimenő élei és az előző superstep-ből kapott üzenetei alapján eltárolja az élek számát. Végigiterál az üzeneteken és a csúcs élein, majd az élek dimenzióit egy `Map`-be gyűjti dimenziótípus szerint. Végül dimenziók szerint végigiterál a `Map`-en és kiszámolja a szumma utáni részt. Ezt követően behelyettesít a képletbe, az eredményt beállítja a csúcs értékének és leállásra szavaz.

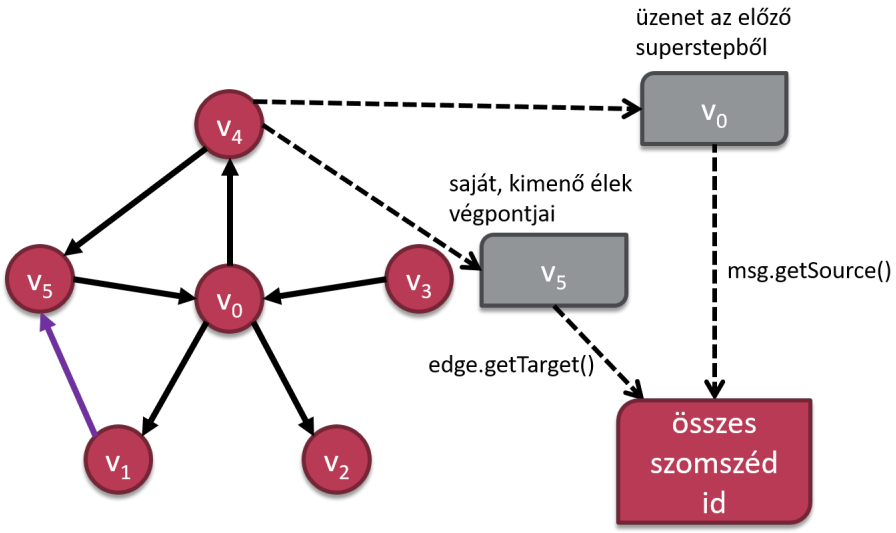
- Dimensional clustering 1 (DC1): Az üzenetek formátumára a `MapFriendList` segédosztályt használtam, ami a `MessageWrapper` leszármazottja. Giraphban egy számítás alatt kizárólag egyféle típusú üzenetet lehet használni. Az wrapper-ben az üzenet típusa `MapWritable`, a csúcsoké továbbra is `LongWritable`, az élek értéke pedig `Text` típusú. A `DimensionalClustering1` tartalmazza az algoritmust implementáló négy `BasicComputation` leszármazottat. Ezek sorban:

  A `SendOutEdges` létrehoz egy `MapFriendsList` objektumot, beállítja a küldőt a csúcs azonosítójára, az üzenetet és az éltípust pedig üres objektumokra, mert egyelőre nem számítanak. A `sendMessageToAllEdges` metódussal elküldi az összes szomszédjának az üzenetet.



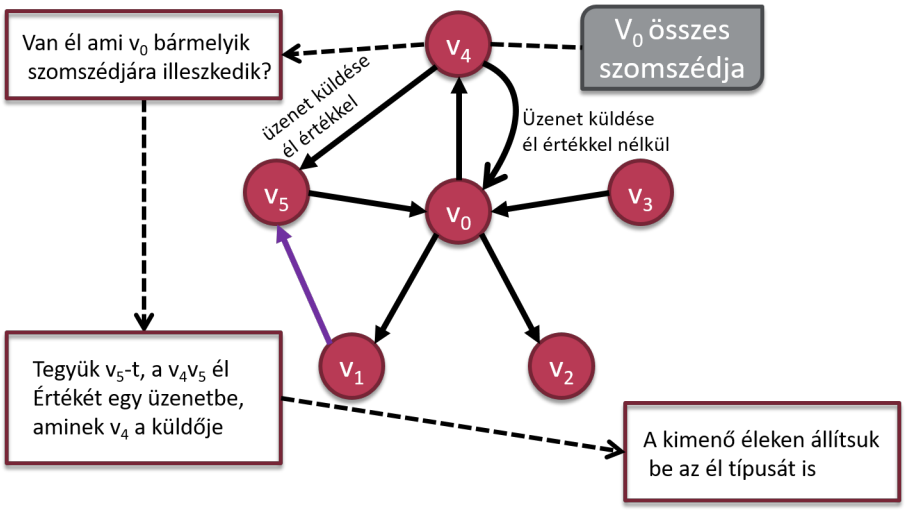
4.8. ábra. Dimensional clustering 1 számítása: Nulladik superstep v_0 szempontjából

A `SendFriendsListToAllEdges` létrehoz egy `MapWritable` objektumot, végigiterál az éleken és belerakja kulcsként a kimenő élek másik csúcsát a `getTargetVertexId` metódussal, értéknek pedig egy üres `Text` objektumot ad meg. Ezután a kapott üzeneteken a wrapper `getSourceId` metódusával kideríti, hogy ki melyik csúcs küldte azt, és hasonlóan belerakja a `MapWritable` objektumba. Így az utóbbi tartalmazza a csúcs összes szomszédos csúcsának az azonosítóit élirántól függetlenül. A következő lépésben egy `MapFriendsList` üzenetet-csomagot készít, amelyben maga az üzenet a szomszédokat tartalmazó `MapWritable`, a küldő a csúcs, és az élek típusa egy üres `Text`, mert arra még mindig nincs szükség. Az üzenetet a `MapWritable` kulcsai segítségével elküldi az összes szomszédjának.



4.9. ábra. Dimensional clustering 1 számítása: Első superstep v_4 szempontjából

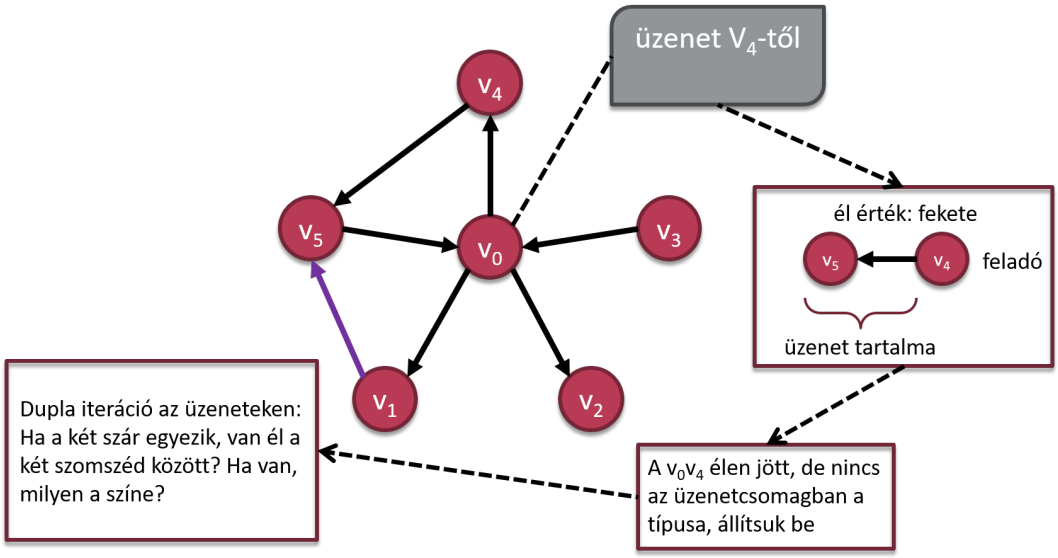
A második superstep-et a `NeighboursLinkComputation` osztály implementálja. Megkapta az előző superstep-ből az összes szomszédjától azok szomszédai listáját. Ez azért jó, mert ha összeveti a saját éleit valamelyik szomszédjától kapott listával, el tudja dönteni, hogy a küldő csúcs szomszédságában van-e éle. Ehhez végigiterál az üzeneteken, majd az üzeneten belül a kapott csúcslistákon. Ha csúcs `getEdgeValue` metódusa egy csúcs id-t kap paraméterül és nincs a két csúcs között él, `null` értékkel tér vissza. Ez a metódus ezért alkalmas annak a vizsgálatára, hogy van-e éle az aktuális csúcsnak a küldő csúcs szomszédságában. Ha van, akkor eltárolja egy új `MapWritable` objektumban; a csúcslista aktuális id-ját kulcsként, a hozzátartozó él dimenzióját értékként. Tehát egy bejegyzés az új `MapWritable`-ben a küldő csúcs szomszédja, és az aktuális csúcsból a szomszéd fele mutató él értéke. Továbbra is az üzeneteken iteráló ciklusban, de már az egy üzeneten belüli iteráción kívül létrehoz egy új `MapFriendsList` objektumot, beállítja magát az üzenet küldőjének, az új `MapWritable`-t pedig az üzenet tartalmának. Az üzenetet szeretnénk visszaküldeni a küldő csúcsnak úgy, hogy az üzenete-csomag tartalmazza az él típusát is, amin utazik. Ezt csak akkor tudjuk megtenni, ha kimenő élről van szó, mert az előző superstep-ből jött üzenetek tartalmazzák a küldő id-t, így az `vertex.getEdgeValue(msg.getSource())` hívással kimenő élek esetén megkapjuk az él értékét, bemenő élek esetén pedig `null`-t. A `NeighboursLinkComputation` ezzel a módszerrel beállítja az élek értékét, és az üzenetettel válaszol az összes feladónak.



4.10. ábra. Dimensional clustering 1 számítása: Első superstep v_4 szempontjából

A `DimensionalClustering1Computation` osztály az algoritmus következő lépését dolgozza fel. Megkapja az üzeneteket a szomszédságában lévő élekkel és az élek végpontjainak azonosítóival, illetve az élek dimenzióival, amelyeken az üzenet utazott (az üzenet-csomag/wrapper metódusával kérhető el). Végigiterál az üzeneteken és a `vertex.getEdgeValue(msg.getSourceId())` hívással beállítja azoknak az éleknek az értékét, melyeket a küldő csúcs az irányításuk miatt nem tudott beállítani. Ezt a lépést követően egy üzenet-csomag tartalmazza, hogy melyik élen keresztül jött, melyik csúcs küldte, a küldő csúcs egy élét az aktuális csúcs szomszédságában, és az utóbbi él végpontját, ami az aktuális csúcs egy szomszédja. Ezután duplán végigiterál az üzeneteken, és megvizsgálja, hogy a két él között, amelyeken a két üzenet jött, van-e összekötő harmadik él. Meg tudja vizsgálni, mert mindkét üzenet tartalmazza a szomszédsági élek listáját, továbbá tudjuk az összes érintett él típusát, melynek segítségével a triádok és háromszögek dimenzióira kiszabott feltételek vizsgálata teljesíthető.

- Dimensional clustering 2 (DC2): A metrika kiszámításához az előző, DC1-nél alkalmazott módszert használtam, csak a triádok és háromszögek szárainál változott a feltétel. A szárak értékei különbözők kell legyenek, és ha van köztük bezáró él, annak is különböznie kell.



4.11. ábra. Dimensional clustering 1 számítása: Harmadik superstep v_0 szempontjából

5. fejezet

Értékelés

5.1. Adathalmaz

5.1.1. LDBC SNB

A Linked Data Benchmark Council (LDBC) szervezet célja, hogy gráfadatbázisok és gráffeldolgozó rendszerek teljesítménymméréséhez benchmarkokat szolgáltasson. A Social Network Benchmark (SNB) egy közösségi hálót definiál [7], amelyben személyek, fórumok, üzenetek, városok stb. találhatók. A benchmark keretrendszer DATAGEN modulja[1] különböző méretű közösségi hálókat generál.

5.1.2. Adatok átalakítása

A DATAGEN modul CSV és Turtle formátumban képes menteni a generált közösségi hálókat. Ezek közül azonban egyik sem tölthető be közvetlenül a Flink és a Giraph rendszerekbe, ezért a mérés előtt az adatokat a megfelelő formátumba konvertáltam. A konverzió során eltávolítottam az egyes csomópontok és élek attribútumait, ezeket ugyanis nem vesszük figyelembe a metrikaszámítás során. A konvertálás idejét nem számítottam bele a mérési eredményekbe.

5.2. Mérési elrendezés

A teljesítménymérés során az Apache Flink és Apache Giraph gráffeldolgozó rendszereken implementált metrikák futási idejét mértem le és hasonlítottam össze. Az Apache Giraphban a metrikák egy számítógépen egy standalone módban futó Hadoop fölött hajtódtak végre `MapReduce` jobokként, amelyet a Giraph Gradle dependenciákból oldott fel. A Flink szintén támogatja a Hadoopot - és külön konfigurációval el lehet érni, hogy a fölötte fusson -, de van saját futtatási környezete is, melyben az Akka Actor keretrendszert használja a számítások párhuzamosítására. A mérés során a Flink egy számítógépen futott a saját környezetében. A metrikák futási idejének méréséhez használt számítógép főbb paraméterei az 5.1 táblázat mutatja be.

[1] `https://github.com/ldbc/ldbc_snb_datagen`

Paraméter	Érték
Processzor gyártó	AMD
Processzor modell	FX-4100
Processzor órajel	3,6 Ghz
Processzor magok száma	4
Alaplap gyártó	ASROCK
Alaplap modell	970 Extreme3
Memória gyártó	Corsair
Memória modell	CMZ4GX3M1A1600C9
Memória típus	DDR3
Memória méret	4096 MB
Memória sebesség	1334 MHz
Merevlemez gyártó	Western Digital
Merevlemez modell	WDC WD10EZEX-08MN2NA0
Merevlemez fordulatszám	7200 rpm
Operációs rendszer	Ubuntu 17.04

5.1. táblázat. Számítógép paraméterek

5.3. Mérési eredmények

Az 5.2 táblázat első sorában a metrika neve, második és harmadik sorában a Flink és Giraph rendszereken mért futtatási idő látható másodpercben. A metrikákat implementáló metódusok idejét a `System.nanoTime` Java függvény segítségével mértem le. Az 5.3 táblázat a Giraph jobok végrehajtási fázisainak idejét tartalmazza, melyek a log fájlban megtalálhatók. Ellentétben ezzel, a Flink nem tárol információt a job végrehajtási idejéről. A Giraph végrehatási fázisainak idejét az 5.4 fejezet mutatja be részletesen.

Metrika	Apache Flink (sec)	Apache Giraph (sec)
Kimenő fokszám	0.93101578	13.073073498
Bemenő fokszám	1.755186127	14.480148964
Átlagos fokszám	0.383415794	13.07081995
Lokális klaszterezettségi együttható	2.138349009	15.071808708
Dimensional degree	1.044030647	13.052500864
Node dimension activity	1.110209992	13.042950934
Node dimension connectivity	1.101649479	13.050596872
Node exclusive dimension connectivity	1.608005846	13.043943519
Edge dimension activity	0.596544413	13.04745898
Edge dimension connectivity	1.101649479	13.049813614
Node activity	1.110209992	13.041468733
Multiplex participation coefficient	23.241666889	13.135025658
Dimensional clustering 1	0.863860928	16.053990732
Dimensional clustering 2	0.766273432	16.055239004

5.2. táblázat. Metrikák futási ideje Flinken és Giraph-on

Metrika	Setup	Input ss.	0. ss.	1. ss.	2. ss.	3. ss.	Shutdown	Total
Kimenő fokszám	0.024	0.44	0.09	-	-	-	8.901	9.456
Bemenő fokszám	0.022	0.468	0.134	0.174	-	-	8.88	9.678
Átlagos fokszám	0.023	0.471	0.099	-	-	-	8.791	9.384
LCC	0.026	0.527	0.21	0.409	0.437	0.484	8.854	10.947
Dimensional degree	0.025	0.466	0.176	0.186	-	-	8.877	9.734
NDA	0.025	0.455	0.111	0.112	0.112	-	8.747	9.5647
NDC	0.024	0.473	0.11	0.092	0.245	-	8.733	9.678
NEDC	0.025	0.498	0.192	0.159	0.112	0.253	8.856	10.099
EDA	0.023	0.458	0.124	0.113	-	-	8.856	9.575
EDC	0.025	0.468	0.121	0.243	-	-	8.764	9.623
NA	0.025	0.495	0.147	0.215	-	-	8.894	9.779
MPC	0.023	0.523	0.132	0.469	-	-	8.9	10.052
DC1	0.028	0.465	0.379	1.345	1.17	0.56	8.861	12.809
DC2	0.032	0.478	0.377	1.182	1.11	0.534	8.868	12.584

5.3. táblázat. Giraph jobok részletes futási idői

5.4. Mérési eredmények elemzése

5.2 táblázat idői azt mutatják meg, hogy a metrikát implementáló függvény meghívásának kezdetétől mennyi idő telik el a használt erőforrások lezárásáig. Egy felhasználó szemszögéből az az idő, amennyit az illető vár a metrika végrehajtásának kezdetétől az eredmény elkészültéig. Ebben benne van a gráf betöltésének az ideje, a job elkészítésének az ideje, a konfigurációs lépések, a jobok végrehajtása és az erőforrások lezárása is.

A Flinken és Giraph-on mért eredmények körülbelül 12-13 másodperces eltérést mutatnak. A nagy különbség két okból adódik. A Giraph ZooKeepert használ a jobok és az alkalmazás konfigurációjának és életciklusának kezelésére. Egy ismert bug miatt amikor a Giraph-ot Eclipse-ből vagy IntelliJből használják, nem képes elsőre csatlakozni a ZooKeeperhez. A másik ok, hogy a job végrehajtása és az eredmény fájlba írás után sokáig tart az erőforrásainak lezárása.

A Giraph jobok végrehajtásának fázisait az 5.3 táblázat tartalmazza az egyes metrikák függvényében.

- Setup: A legelső lépésektől a bemeneti gráf beolvasásáig eltelt idő.
- Input superstep (Input ss.): A bemeneti gráf beolvasása, particionálása, és a partíciók kiosztása a workerekhez.
- Superstep (ss.): Az adott superstep tényleges végrehajtási ideje.
- Shutdown: Az eredmény fájlba írása után az erőforrások (például hálózati kapcsolatok, stb.) lezárásáig.
- Total: A teljes végrehajtási idő, az előzők összege.

Az 5.3 táblázat Total oszlopában lévő idők és az 5.2 táblázat Apache Giraph oszlopa körülbelül 2-3 másodpercben térnek el. Ennek egy lehetséges magyarázata, hogy a végrehajtás legelején történő sikertelen csatlakozás a ZooKeeper-hez még a Setup idő előtt történik, így ezt az 5.2 táblázat beleszámolja a metrika futási idejébe, az 5.3-es pedig nem.

Az 5.2 táblázat oszlopaiban megfigyelhető, hogy a nagyobb számítási igényű metrikák futási ideje 1-2 másodperccel több mint az átlagos, pl. a lokális klaszterezettségi

együttható számolása. Ugyanakkor érdekes, hogy míg a dimensional clustering Giraphan a legösszetettebbnek számít és a legtöbb ideig is fut, FLinkben lényegesen egyszerűbb volt az implementálása, és gyorsabb is a futási ideje. A Flink oszlopban mért idők közül a multiplex participation coefficient értéke kiugró. Ez a metrika implementációjából adódik, mivel a már előre megírt dimensional degreet többször is használja. A Flink dimensional degree jobot készít, átadja az aktor rendszerének, ami végrehajtja és visszatér az eredménnyel, ami minden egyes hívásnál sok időt vesz el.

Az 5.3 táblázatban látszik, hogy a setup és az input idő minden metrika esetében közel azonos. Ennek magyarázata, hogy mindegyik ugyanolyan körülmények között, ugyanabban a környezetben futott, illetve ugyanaz a gráf volt a bemenete. A superstepek ideje kb. 0,01 és 2 másodperc között mozog, és az figyelhető meg, hogy azok a superstepek melyekben sok üzenetküldés történt, jóval hosszabbak a többinél. Ilyen például a dimensional clustering metrikák első és második superstepje. Ha összeadjuk a superstepek idejét, de a shutdown időt nem számolnánk bele(Total - Shutdown - Setup), a futási idők mind a Giraph-ban és a Flinkben egy nagyságrendre kerülnének.

Összességében elmondható, hogy ezeken a hardvereken és ezen környezetekben futva a Flink egy nagyságrenddel gyorsabb a Giraphnál, aminek az oka a Giraph erőforrások lezárására fordított ideje. Ugyanakkor ez nem jelenti azt, hogy a Flink minden esetben - akár csak ezeket a metrikákat tekintve - gyorsabb lenne. Az Apache Flinket és Giraphot elosztott számítógépes rendszerekre tervezték, de a mérés nem elosztott környezetben történt. A metrikák futási idejének éles, elosztott számítógép hálózaton való lemérése a jövőbeli tervek közé tartozik.

6. fejezet

Kapcsolódó munkák

Ez a fejezet röviden ismerteti a munkámhoz kapcsolódó gráfanalízist támogató keretrendszereket és példákat mutat egyéb gráfmetrikákra.

6.1. Metrikák

A multidimenzionális gráfelemzés viszonylag új területnek számít a gráfanalízisben, ezért a tudományos cikkekben leírt metrikák nem alkotnak egységes képet, nincsenek összegyűjtve és rendszerezve, illetve attól függően, hogy a gráf milyen tulajdonságait vizsgáljuk, újabbak és újabbak jöhetnek szóba. A dolgozatomban a 4.3 fejezetben bemutatottakon kívül további metrikák [4] is léteznek:

- Dimensional degree entropy: Megmutatja egy csúcs dimenzionális fokának eloszlását a dimenziók között.

$$H(v) = -\sum_{d \in D} \frac{Degree(v,d)}{Degree(c,D)} \ln\left(\frac{Degree(v,d)}{Degree(c,D)}\right)$$

  Az entrópia értéke nulla, ha a csúcs összes éle egy dimenzióban aktív és maximális, ha az élek a dimenziók között egyenletesen eloszlanak.

- Interdependence: Ez a metrika a csúcsok közötti legrövidebb utakat vizsgálja a dimenziók függvényében. Kiszámolja, hogy a két csúcs közötti legrövidebb utak közül azok, amelyek legalább két dimenzióban aktívak, hogyan aránylanak az összes legrövidebb úthoz. Az v_i és v_j csúcsok közötti λ_i interdependencia definíciója:

$$\lambda_i = \sum_{j \neq i} \frac{\psi_{ij}}{\sigma_{ij}},$$

  ahol ψ_{ij} az i és j csúcs közötti olyan legrövidebb utak száma, amelyek legalább két dimenzión keresztül haladnak, σij pedig az i és j közötti összes legrövidebb út száma.

- Edge overlap: Az edge overlap kiszámolja, hogy mekkora valószínűséggel létezik az i és j között él a d' dimenzióban, ha létezik közöttük él d-ben. A formális definíció feltételes valószínűséggel:

$$P\left(v_{ij}^{[d']} | v_{ij}^{[d]}\right) = \frac{\sum_{ij} v_{ij}^{[d']} v_{ij}^{[d]}}{\sum_{ij} v_{ij}^{[d]}})$$

6.2. Eszközök

- Giraph++: A Giraph++ egy Giraph-ra épülő nyílt forráskódú Pregel-implementáció és az első olyan gráffeldolgozó rendszer, amely egyszerre támogatja az aszinkron végrehajtást és a gráfmutációkat is. [16] Előnye több optimalizációs technika, amelyeket a Giraph nem támogat.

- Graphalytics: A Graphalytics egy olyan benchmark, amely különböző gráffeldolgozó rendszerek teljesítményeit méri. [5] Képes eltérő platformokon és különböző adathalmazokon algoritmusok futási idejének lemérésére. Az általuk implementált mérések során hat metrikát használtak:

  1. BFS
  2. PageRank
  3. Weakly connected components
  4. Communitiy detection
  5. Lokális klaszterezettségi együttható
  6. Legrövidebb utak

- GraphLab: A GraphLab egy nagy teljesítményű, elosztott keretrendszer gráfanalízisre és gépi tanulás modellezésére. [11] Eszközöket biztosít topológiai modellezésre, gráfanalitikára, ajánlórendszerek készítésére és gráfok, statisztikák vizuális megjelenítésére.

7. fejezet

Összefoglalás és jövőbeli tervek

7.1. Összefoglalás

Az internet terjedésével egyre több adat keletkezik és egyre nagyobb az igény adatok feldolgozására. Az internetes alkalmazásokban a felhasználók növekvő mennyisége, a felhasználói tapasztalatok javításának célja, az ajánlórendszerek terjedése és a hálózati topológiák optimalizálásának igénye maga után vonja olyan eszközök fejlesztését, amelyek segítségével nagy mennyiségű adatok gyorsan feldolgozhatók.

A dolgozatomban bemutattam a mai modern gráffeldolgozó rendszerek működését elméleti és gyakorlati szempontból, valamint multidimenzionális metrikákkal lemérem két elterjedt keretrendszer teljesítményét. Az elméleti részek leírják a kapcsolatot az adatok és a gráfok között, ismertetik a multidimenzionális gráf fogalmát és példát mutatnak a tulajdonsággráf adatmodellre. Bemutatják gráffeldolgozás ontológiáját és a legelterjedtebb gráffeldolgozási modelleket. A dolgozat foglalkozik az Apache Flink és Apache Giraph felépítésével és fontosabb tulajdonságaival, majd bevezeti a metrika fogalmát és leírja a metrikák implementálásnak részleteit az előbbi két rendszeren. Végül a mérési környezet és a metrikák futási idejének elemzése következik, amit a kapcsolódó munkák bemutatása zár le.

Eredmények. A dolgozatban bemutatott metrikákat megvalósítottam Java nyelven az Apache Flink és Apache Giraph gráffeldolgozó rendszerek felett. A metrikák futási idejét lemértem és elemeztem, majd ezek alapján összehasonlítottam a két rendszer teljesítményét. Az elemzés során arra a következtetésre jutottam, hogy ha az Apache Flink a saját futtatási környezetében, az Apache Giraph pedig standalone Hadoop felett fut, és adathalmaz közepesen nagy, a multidimenzionális metrikák az Apache Flinken egy nagyságrenddel hamarabb lefutnak.

Következtetés. Gráfmetrikák mérésére otthoni és teszt környezetben az Apache Flink alkalmasabb, mivel könnyebb telepíteni, saját futtató környezettel rendelkezik, illetve kis és közepesen nagy gráfokon jobb teljesítményt mutat.

Ajánló. A dolgozatom bevezetést nyújt a modern gráffeldolgozó rendszerek működésébe és az Apache Flink illetve Apache Giraph gráfanalízis platformok programozásába elméleti és gyakorlati szempontból, ezért segítséget nyújthat azoknak, akik nem jártasak a témában, vagy csak érdeklődnek ezen rendszerek működése iránt.

7.2. Jövőbeli tervek

Jövőbeli tervek közé tartozik az Apache Flink és Apache Giraph teljesítményének mérése elosztott rendszereken, a multidimenzionális metrikák rendszerezése és implementálása. További terv az eredmények összevetése különböző méretű adathalmazok és elosztottság szempontjából, illetve a metrikák futási idejének lemérése más keretrendszerekben, mint a Giraph++, vagy a GraphLab.

Köszönetnyilvánítás

Köszönöm Szárnyas Gábornak a sok segítséget és átadott tudást.

Irodalomjegyzék

[1] Apache: Akka and actors. https://cwiki.apache.org/confluence/display/FLINK/Akka+and+Actors.

[2] Apache: Apache flink documentation. https://ci.apache.org/projects/flink/flink-docs-release-1.2/.

[3] Samer Assaf: Generalized metrics. *arXiv preprint arXiv:1603.01246*, 2016.

[4] Federico Battiston – Vincenzo Nicosia – Vito Latora: Metrics for the analysis of multiplex networks. *arXiv preprint arXiv:1308.3182*, 2013.

[5] Mihai Capotă – Tim Hegeman – Alexandru Iosup – Arnau Prat-Pérez – Orri Erling – Peter Boncz: Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15*, GRADES'15 konferenciasorozat. New York, NY, USA, 2015, ACM, 7:1–7:6. p. ISBN 978-1-4503-3611-6.
URL http://doi.acm.org/10.1145/2764947.2764954. 6 p.

[6] Andrew Comstock: *What is Apache Hadoop?* 2013. http://qr.ae/TbZ3Hi.

[7] Orri Erling – Alex Averbuch – Josep-Lluís Larriba-Pey – Hassan Chafi – Andrey Gubichev – Arnau Prat-Pérez – Minh-Duc Pham – Peter A. Boncz: The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (konferenciaanyag). 2015, 619–630. p.
URL http://doi.acm.org/10.1145/2723372.2742786.

[8] Mike Ferguson: What is graph analytics? http://www.ibmbigdatahub.com/blog/what-graph-analytics.

[9] Thomas Frisendal: Graph data modeling for nosql and sql. 2016.

[10] Fabian Hueske: Introducing stream windows in apache flink. https://flink.apache.org/news/2015/12/04/Introducing-windows.html.

[11] Yucheng Low – Danny Bickson – Joseph Gonzalez – Carlos Guestrin – Aapo Kyrola – Joseph M Hellerstein: Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5. évf. (2012) 8. sz., 716–727. p.

[12] Grzegorz Malewicz – Matthew H. Austern – Aart J. C. Bik – James C. Dehnert – Ilan Horn – Naty Leiser – Grzegorz Czajkowski: Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010* (konferenciaanyag). 2010, 135–146. p.
URL http://doi.acm.org/10.1145/1807167.1807184.

[13] Brahima Sanou: The world in 2013: Ict facts and figures. *International Telecommunications Union*, 2013.

[14] Aaron Smith – Laurie Segall – Stacy Cowley: Facebook reaches one billion users. *CNN Money*, 2012.

[15] Gábor Szárnyas – Zsolt Kővári – Ágnes Salánki – Dániel Varró: Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016* (konferenciaanyag). 2016, 87–94. p. URL `http://dl.acm.org/citation.cfm?id=2976786`.

[16] Yuanyuan Tian – Andrey Balmin – Severin Andreas Corsten – Shirish Tatikonda – John McPherson: From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7. évf. (2013) 3. sz., 193–204. p.

[17] Leslie G. Valiant: A bridging model for parallel computation. *Commun. ACM*, 33. évf. (1990. augusztus) 8. sz., 103–111. p. ISSN 0001-0782.
URL `http://doi.acm.org/10.1145/79173.79181`. 9 p.