K-mag dekompozíció gráf-adatfolyam alapján

Tanulmány
Ahmet Erdem Sarıyüce, Buğra Gedikz, Gabriela Jacques-Silva,
Kun-Lung Wu, Ümit V. Çatalyürek
Streaming Algorithms for k-core Decomposition [1]
cikke alapján

Adatbázisrendszerek elméleti alapjai – ELTE-IK
Balogh Szabolcs (NIO402)
Beke Balázs (NUH8UX)
Botos Ádám (YFYO78)

2013. november 12.

1. Absztrakt

Egy gráf k-magja egy olyan maximális részgráf, melyben minden csúcsnak legalább k másik részgráfbeli csúccsal van kapcsolata. Az ilyen részgráfok keresése gyakori eleme a nagy adathalmazok elemzésének, úgy mint emberi közösségek meghatározása, fehérjék viselkedésének becslése vagy olyan, gyakorlatban is sűrűn használt gráfelméleti problémák, mint a legnagyobb klikk keresése. A gráfok a valós életben szinte mindig változnak, ezért fontos olyan algoritmusok megalkotása, melyek képesek követni és kezelni a gráfok változásait.

Az alábbi algoritmusok minden élváltozásra megkeresik a gráf azon részét, aminek a maximális k-magját érintheti a változás. Az eredmények nagyságrendekkel javítják a k-magok meghatározásának a hatékonyságát a nemváltozáskövető algoritmusokkal szemben. Egy 16 millió csúcsból álló gráfra a sebesség-különbség már milliós nagyságrendű.

2. Bevezetés

Gyakran használunk az alkalmazásainkban gráfokat. Ábrázolhatjuk a emberek kapcsolati rendszerét, a web szerkezetét, telefonhívásokat, vagy fehérjék felépítését. Klikkek, magok segítségével ismerhetőek fel a gráfban az

összetartozó csoportok, amiket felhasználhatunk célzott hirdetésre, promóciókra, spamek szűrésére, biológiai folyamatok modellezésére, becslésére és számtalan más feladat megoldására.

A valóságot ábrázoló gráfok a legtöbb esetben változnak. Egy szociális hálóban például felhasználók jönnek-mennek és kapcsolatok alakulnak vagy épp bomlanak fel. Új tartalmak a weben, új kapcsolat a telefonkönyvben, mind változó gráfokat követelnek.

A gráf k-magjának nevezünk egy olyan maximálisan összefüggő részgráfot, amiben minden csúcs foka legalább k. A k-magokat gyakran használják a gráf sűrű komponenseinek megkeresésére, gráfok megjelenítésére, klikkek keresésére (mivel egy *k-klikk* egy *k-1-mag*!), stb. A cikk a k-magok nyilvántartására nyújt egy inkrementális megoldást.

A dekompozíció során minden csúcshoz nyilvántartunk egy olyan k értéket, ami azt jelenti, hogy a csúcs egy k-mag része, de nincs olyan $k+1$-mag, ami tartalmazza azt. Ezekből az értékekből könnyedén előállíthatjuk bármelyik magot. A problémára viszonylag egyszerűen írható algoritmus, ám ezek csak statikus gráfokban működnek. Egy változó gráfban tehát minden él-változás esetén újra kellene számolni az összes csúcs k-értékét. Ezt általában nem engedhetjük meg magunknak. Javítható a teljesítmény, ha a változásokat kötegelve ritkábban értékeljük újra az értékeket, így azonban jelentős késéssel reagálhatunk a változásokra.

A cikkben egy olyan szekvenciális algortimus szerepel, ami megoldást kínál a régi algoritmusok problémáira. Az algoritmus minden él-változásakor frissíti a csúcsok k-értékeit (a csúcsok változásai triviálisak). Ehhez le kell küzdeni néhány akadályt. Egyrészt, meg kell határozni minden olyan csúcsot, amit érint a változás. Másrészt, hatékonyan algoritmust kell adni az értékek frissítésére.

A cikkben tehát bemutatásra kerül az első olyan k-mag dekompozíciós algoritmus, amely képes követni a gráf változásait, és meghatározni azt a részgráfot, amit érint egy-egy változás. Több változat is bemutatásra kerül, valamint mérésekkel támasztják alá a szerzők a nem inkrementális algoritmusokkal szembeni teljesítmény-növekedést, ami akár hatos nagyságrendű is lehet egy 16 millió csúcsot tartalmazó gráf esetén.

3. Kapcsolódó munkák

A k-magokat Seidman [2] vezette be 1983-ban a gráfok összefüggő régióinak a jellemzésére. A k-mag dekompozícióra pedig Batagelj [3] írt egy hatékony algoritmust. Alapevetően ezekre a munkákra épül a tanulmány

alapjául szolgáló cikk, aminek célja az első változáskövető k-mag dekompozíciós algoritmus megalkotása.

A k-magok különböző felhasználási területein is születtek munkák, úgy mint a közösségi hálózatok [4, 5] elemzése, bűnözői hálózatok [9] és egyéb közösségek felderítése[4]. Jelentős felhasználói terület a nagy hálózatok vizualizációja [6, 7, 8], meghatározott tulajdonságú gráfok generálása [12], maximális *klikk*-keresés [13], illetve sűrű részgráfok keresése [14].

A jelenlegi cikk irányítatlan és súlyozatlan gráfokkal foglalkozik, találhatók irányított [15] és súlyozott [4] k-magokról szóló cikk is. Ám a jelen cikkben leírtakhoz hasonló, inkrementális tulajdonságú megoldással eddig senki sem foglalkozott.

A cikk megjelenésével nagyjából egy időben jelent meg egy másik, a témával foglalkozó cikk [16]. Azonban Li et al. algoritmusa negyedfokú, a jelenleg taglalt megoldás lineáris komplexitású algoritmust biztosít.

4. Alapok

Ebben a részben fogalmakat definiálunk és megfigyeléseket teszünk, amelyek szükségesek a k-mag dekompozíció pontos megértéséhez. Ezekre a cikk elméleti eredményeiben szereplő tételek bizonyításai is hivatkoznak.

Legyen $G = (V, E)$ egy irányítatlan és súlyozatlan gráf. Egy $H = (V', E')$ gráfban ($H \subseteq G$), ahol $V' \subseteq V$, E' pedig az összes $(u, v) \in V'$-beli csúcsok közötti E-beli csúcs, $\delta(H)$ jelölje a H részgráf minimális fokszámát (azaz $\delta(H) = min\{\delta_H(u) : u \in H\}$ és $\delta_H(u)$ az u fokszáma a H-ban).

1. Definíció. *Ha H egy összefüggő gráf és $\delta(H) \geq k$, hogy H egy k-részmag G-ben. Továbbá, ha H maximális (azaz $\nexists H'$, hogy $H \subset H'$ és H k-részmag G-ben), akkor H egy k-mag G-ben*

1. Megfigyelés. *Legyen $u \in H$, ahol H k-mag. Ekkor H egyértelmű, azaz nincs olyan másik k-mag, ami tartalmazza u-t.*

A fenti megfigyelés alapján ezt az egyértelmű k-magot, ami tartalmazza u csúcsot, jelöljük H_k^u-val.

2. Definíció. *$u \in H$ legnagyobb k-magja, jelöljük H^u-val, az a k-mag, amely tartalmazza u-t és a $k = \delta(H^u)$ értéke maximális (azaz $\nexists H$, hogy $u \in H$ és H egy l-mag, ahol $l > k$). u legnagyobb k-magjának K-értéke: $K(u) = \delta(H^u)$.*

2. Megfigyelés. *Ha H k-mag G-ben, akkor egyértelműen létezik olyan $H' \subseteq H$, amely $(k-1)$-mag G-ben.*

3. Megfigyelés. *$\forall u \in V$ csúcs, ahol $K(u) = k$ - a 2. megfigyelés alapján - része a $H_k^u \subseteq H_{k-1}^u \subseteq \cdots \subseteq H_1^u$ magoknak.*

Tehát azt mondhatjuk, hogy egy G gráf dekompozíciója megfelel a csúcsok maximális k-magjainak a keresésével. A következő kiegészítés alapján, adott K értékek mellett a k-magok könnyedén előállíthatók minden csúcsra.

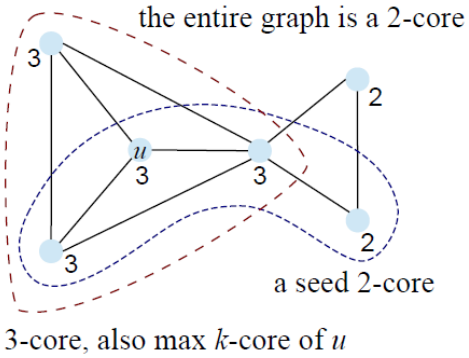
Algorithm 1: FINDKCOREDECOMPOSITION($G(V, E)$)

```
Data: G: the graph
Compute δ_G(v) (i.e., the degree) for all vertices v ∈ V
Order the set of vertices v ∈ V in increasing order of δ_G(v)
for each v ∈ V do
    K(v) ← δ_G(v)
    for each (v, w) ∈ E do
        if δ_G(w) > δ_G(v) then
            δ_G(w) ← δ_G(w) − 1
    Reorder the rest of V accordingly
return K
```

FindKCoreDecomposition: Gráf K értékeinek előállítása.

1. Kiegészítés az 1. algoritmushoz. *Adott egy G gráf, ahol $\forall v \in G$-re $\exists K(v)$, az $u \in G, K(u) \geq k$ csúcs k-magja (H_k^u) tartalmazza u-t, valamint minden olyan $w \in G, K(w) \geq k$ csúcsot, amelyik elérhető u-ból egy olyan P úton, amire $\forall v \in P, K(v) \geq k$. H_k^u előállítható a gráf bejárásával u-ból, ami során minden w csúcsot beveszünk H_k^u-ba, amire $K(w) \geq k$.*

A kiegészítés szerint tehát minden bejárt csúcs része H_k^u-nak és midnen H_k^u-beli csúcsot bejárunk. A k-mag keresési problémát tehát leegyszerűsíthetjük a K értékek kiszámolására. Az 1. algoritmus kiszámolja egy gráf csúcsainak a K értékeit a következő tulajdonság alapján: Egy gráf k-magjainak a megkereséséhez töröljünk minden csúcsot, aminek a foka kisebb, mint k, valamint törökjük ezen csúcsok éleit is.



1. ábra. Egy gráf különböző magjai

Az ábrán egy gráfot látunk, ahol minden csúcs mellett szerepel azok K-értékei. A teljes gráf egy 2-magot alkot ($G = H_2^u$). Vizsgáljuk meg az u jelű

csúcsot. Az ábrán szerepel a csúcs egy 2-részmagja, valamint a 3-magja(H_k^u). A 3-mag a csúcs legnagyobb k-magja is egyben($H_3^u = H^u$). Vegyük észre hogy $H_3^u \subseteq H_2^u$

5. Elméleti eredmények

Alább szerepelnek azok a tételek, amiket az algoritmusok épülnek. A tételek meghatározzák azokat a csúcsokat, amelyek frissülhetnek egy változás esetén, illetve biztosítják, hogy más csúcs k-értéke nem változhat.

4. Megfigyelés. *Legyen $G = (V, E)$ gráf és $u, v \in V$. Ha van él u és v között ($e \in E$), amire $K(u) > K(v)$, akkor – az 1. kiegészítés alapján – $e \notin H^u$, viszont $e \in H^v$.*

1. Tétel. *Egy él beszúrása vagy törlése a $G = (V, E)$ gráfból maximum 1-gyel változtatja meg egy $u \in V$ csúcs K értékét.*

2. Tétel. *Ha egy $G = (V, E)$ gráfban egy (u, v) él megjelenik vagy törlődik, ahol $u, v \in V$ úgy, hogy $K(u) < K(v)$, akkor $K(v)$ értéke nem változhat.*

A 2. Tételből következik az is, hogy egy (u, v) beszúrása vagy törlése $K(u)$-t legfeljebb 1-gyel változtathatja, amennyiben $K(u) \leq K(v)$.

3. Tétel. *$G = (V, E)$ gráfban $u, v \in V$ csúcsok közé szúrunk be egy (u, v) élt. Ebben az esetben azok a csúcsok, amelyek K értéke változik, összefüggő részgráfot alkotnak: $G' \subset G \cup (u, v)$. Hasonlképpen, egy (u, v) él törlése esetén a K értékben változó csúcsok egy összefüggő $G'' \subset G$ részgráfot alkotnak.*

4. Tétel. *Adott $G = (V, E)$ gráfban, egy (u, v) él beszúrása (törlése) esetén ($u, v \in V, K(u) \leq K(v)$) csak azon $w \in V$ élek K értéke nőhet(csökkenhet), amelyekre $K(w) = K(u)$ és elérhető u-ból egy olyan úton, amiben minden csúcsra $K(w) = K(u)$.*

Tehát az (u, v) él beszúrásakor, legyen u a gyökér, ha $K(u) \leq K(v)$ ($K(u) = K(v)$ esetén bármelyik csúcs lehet gyökér). Minden csúcs, aminek a K értéke megváltozhat, elérhető egy úton, amin minden csúcs K értéke megegyezik.

6. Algoritmusok

6.1. Subcore

Az első algoritmus a 4. tételben leírtak szerint frissíti a K értékeket élváltozáskor.

3. Definíció. *Egy $G = (V, E)$ gráfban az $u \in V$ csúcs subcore-ja (jelölése S_u) olyan $w \in V$ csúcsok halmaza, amelyre $K(w) = K(u)$ és elérhető u-ból egy olyan úton, amelyben minden él K értéke megegyezik $K(u)$-val.*

Egy él beszúrására a $G = (V, E)$ gráfba az **InsertEdge**, törlésére pedig a **RemoveEdge** algoritmus frissíti a K értékeket. Mindkettő felhasználja az előbbi definíciót.

Algorithm 2: FINDSUBCORE($G(V, E), K(), u$)

```
Data: G: the graph, K: max-k values, u: the vertex
H(V', E') ← empty graph; Q ← empty queue
cd[v] = 0; visited[v] = false, ∀v ∈ V                    ▷ Lazy init
k ← K(u)                                  ▷ Remember K value of the root
Q.push(u); visited[u] ← true
while not Q.empty() do
    v ← Q.pop(); V'.push(v)
    for each (v, w) ∈ E do
        if K(w) ≥ k then
            cd[v] ← cd[v] + 1
            if K(w) = k and not visited[w] then
                Q.push(w); E'.push((v, w))
                visited[w] ← true
return H and cd
```

FindSubcore: u csúcshoz tartozó subcore halmaz megkeresése, valamint *core degree* értékeinek kiszámítása

A subcore meghatározását a fenti algoritmus végzi. Első lépésként meg kell határozni a gyökércsúcsot. Ez – jelöljük r-rel – legyen a kisebbik K értékű csúcs, egyenlőség esetén bármelyik csúcs lehet. A subcore halmaz meghatározásához az algoritmus egy szélességi bejárást végez u-ból kiindulva és összegyűjti azokat a csúcsokat, amelyek a K értéke azonos a gyökércsúcséval. Egyúttal kiszámolja a csúcsok *core degree* - (cd) értékeit. A cd a csúcs azon szomszédainak száma, amelyek részei a csúcs maximális magjának. Ebből az értékből meghatározható, hogy változhat-e a csúcs K értéke. Ha a csúcs cd értéke kisebb, vagy egyenlő a gyökércsúcs K értékével, akkor azott csúccsal többet nem kell foglalkozni, mert nem lehet része az 1-gyel nagyobb magnak, a K érték változatlan marad. A cd kiszámításához megszámoljuk, hogy az adott csúcsnak hány olyan szomszédja van, aminek nagyobb a K értéke.

Algorithm 3: SUBCORE: INSERTEDGE($G(V, E), K(), u_1, u_2$)

Data: G: the graph, K: max-k values, (u_1, u_2): inserted edge
$r \leftarrow u_1$ ▷ Set the root
if $K(u_2) < K(u_1)$ **then** $r \leftarrow u_2$
$G \leftarrow G \cup (u_1, u_2)$ ▷ Add the edge into G
H, cd $\leftarrow$ FINDSUBCORE(G, K, r) ▷ Find subcore
▷ Now update the K values of the vertices in H
$k \leftarrow K(r)$ ▷ Remember K value of the root
Sort cd values in increasing order (using bucket sort)
for each $v \in H$ *in order* **do**
  if cd$[v] \leq k$ **then** ▷ Cannot be part of a k+1-core
    for each $(v, w) \in H$ **do**
      if cd$[w] >$ cd$[v]$ **then**
        cd$[w] \leftarrow$ cd$[w] - 1$
        Reorder cd values accordingly
  else ▷ All remaining vertices become part of k+1-core
    for each $w \in H$ **do**
      $K(w) \leftarrow k + 1$
    break

InsertEdge: Él beszúrása G gráfba u_1 és u_2 csúcsok közé

A következő algoritmus egy él beszúrásakor fut. Az előbbi algoritmussal meghatározzuk a gyökércsúcshoz (r) tartozó subcore-t. A csúcsokat a subcore-ban *cd* szerint növekvő sorrendbe rendezzük. Minden ciklusban kiválasztjuk a legkisebb *cd*-jű, még feldolgozatlan csúcsot. Ha a említett csúcs *cd* értéke nem nagyobb gyökércsúcs K értékénél, akkor csökkentjük minden szomszédjának a *cd* értékét 1-gyel, hiszen a kiválasztott csúcs nem vehet részt egy $K(r) + 1$-magban. A megmaradt (azaz fel nem dolgozott) csúcsok K értékeit növeljük, majd az algoritmus terminál.

Algorithm 4: SUBCORE:REMOVEEDGE$(G(V,E), K(), u_1, u_2)$

```
Data: G: the graph, K: max-k values, (u1, u2): inserted edge
r ← u1                                              ▷ Set the root
if K(u2) < K(u1) then r ← u2
G ← G \ (u1, u2)                                    ▷ Remove the edge from G
if K(u1) ≠ K(u2) then
    H, cd ← FINDSUBCORE(G, K, r)                    ▷ Find subcore
else
    H1, cd1 ← FINDSUBCORE(G, K, u1)                 ▷ Find subcore of u1
    H2, cd2 ← FINDSUBCORE(G, K, u2)                 ▷ Find subcore of u2
    H ← H1 ∪ H2; cd ← cd1 ∪ cd2
▷ Now update the K values of the vertices in H
k ← K(r)                                            ▷ Remember K value of the root
Sort cd values in increasing order (using bucket sort)
for each v ∈ H in order do
    if cd[v] < k then                               ▷ Cannot be part of a k-core anymore
        K(v) ← k − 1
        for each (w, v) ∈ H do
            if cd[w] > cd[v] then
                cd[w] ← cd[w] − 1
                Reorder cd values accordingly
    else break;                                     ▷ All remaining vertices still in a k-core
```

RemoveEdge: (u_1, u_2) él törlése G gráfból

Ez az algoritmus pedig egy él törlésekor fut. Az előbbi algoritmushoz képest egy fontos változás, ha a törölt él csúcsainak a K értékei megegyeznek, akkor mindkét csúcsnak külön-külön meg kell keresni a subcore halmazát. A ciklus azonban hasonlóképpen fut. A subcore (vagy azok uniója) csúcsai közül a legkisebb *cd* értékűt választjuk ki, és ha az kisebb, mint $k = K(r)$, akkor ez a csúcs már nem lehet része egy k-magnak, tehát csökken a K értéke. Ugyanezért kell a szomszéd csúcsok *cd* értékeit is csökkenteni. Ha a halmazban csak k, vagy annál nagyobb *cd* értékű csúcsok maradnak, az algoritmus terminál.

6.2. Purecore

Az előző részben bemutatott Subcore algoritmus csak a K értékekre épített, amikor meghatározta a csúcshalmazt. Ebben a részben a Purecore algoritmus kerül bemutatásra, ami az egyes csúcsok további adatait is figyelembe veszi, így csökkentve a halmaz méretét. Ehhez először szerepeljen egy definíció:

4. Definíció. *Legyen az $u \in V$ csúcs $MCD(u)$ (maximum-core degree) értéke u csúcs w szomszédainak a száma, amire $K(u) \leq K(w)$.*

5. Megfigyelés. *$G = (V, E)$ gráfban $u \in V$ csúcsra $MCD(u) \geq K(u)$.*

A megfigyelés a definícióból adódik, hiszen $MCD(u) < K(u)$ azt jelentené, hogy a csúcshoz nincs olyan k-mag, ahol $k = K(u)$.

5. Definíció. *$G = (V, E)$ gráfban $u \in V$ csúcs purecore halmaza (P_u) legyen u csúcs és minden olyan $v \in V$ csúcs, amire teljesül, hogy $K(w) = K(u)$ és $MCD(w) > K(u)$, továbbá, v elérhető u-ból egy úton, aminek minden $w \in V$ csúcsára teljesül, hogy $K(w) = K(u)$ és $MCD(w) > K(u)$*

Algorithm 5: FINDPURECORE$(G(V, E), K(), u)$

```
Data: G: the graph, K: max-k values, u: the vertex
H(V', E') ← empty graph; Q ← empty queue
cd[v] = 0; visited[v] = false, ∀v ∈ V                    ▷ Lazy init
k ← K(u)                                  ▷ Remember K value of the root
Q.push(u); visited[u] ← true
while not Q.empty() do
    v ← Q.pop(); V'.push(v)
    for each (v, w) ∈ E do
        if K(w) > k or (K(w) = k and
                                   MCDEGREE(G, K, w)> k) then
            cd[v] ← cd[v] + 1
            if K(w) = k and not visited[w] then
                Q.push(w); E'.push((v, w));
                visited[w] ← true
return H and cd
```

FindPurecore: Egy gráf u csúcsához tartozó purecore halmazt határozza meg. Az **MCDegree** függvény a 4. definíció szerint használandó!

5. Tétel. *$G = (V, E)$ gráfba $(u, v) \in E$, ahol $K(u) \leq K(v)$ él beszúrása esetén csak a $w \in P_u$ csúcsoknak változhat a $K(w)$ értékük.*

A Purecore algoritmus tehát jelentősen csökkentheti a kiértékelendő csúcsok számát él beszúrása esetén. Használjuk az InsertEdge algoritmust a Subcore-nál definiáltak szerint, a **FindSubcore** hívást **FindPurecore**-ra cserélve.

Egy $(u, v) \in E$, ahol $K(u) \leq K(v)$ él törlésekor azonban bármely $w \in S_u$ K értéke változhat, ezért ebben az esetben a Purecore módszer nem használható.

6.3. Traversal

A következő módszerrel tovább szűkíthető a vizsgálandó csúcsok köre, valamint bevezetésre kerül egy módszer az MCD gyorsabb kiszámítására.

Az előbb definiált Purecore algoritmus ugyan szűkíti a csúcsok halmazát, az MCD kiszámítása azonban plusz költséggel jár. Az algoritmus összességében sokkal lassabban fut a Subcore-nál, ha P_u közel akkor, mint S_u. Az MCD-t jó lenne tárolni, így megspórolva a folyamatos újraszámolást. Ehhez vezeti be a szerző a *Purecore Degree*, azaz PCD értéket. Az két értéket együtt *Residential Core Degree*-nek nevezi a cikk, röviden RCD-nek. Ezeket a K-hoz hasonlóan frissíteni és tárolni kell minden csúcsra.

6. Definíció. *Legyen $PCD(u)$ értéke $u \in V$ csúcs azon $w \in V$ szomszédainak a száma, amire teljesül $(K(u) = K(w) \wedge MCD(w) > K(u)$ vagy $K(u) < K(w)$*

A definíció szerint tehát u-nak $PCD(u)$ számú csúcsa van, aminek a K értéke megfelelő ahhoz, hogy az 1-gyel nagyobb k-magban részt vegyen. Minden szomszéd csúcs, aminek a K értéke nagyobb, mint $K(u)$ ilyen. Ha $K(w) = K(u)$, de még $K(w)$ nem érte el az $MCD(w)$-t, akkor az a csúcs is megfelel, hiszen a $K(w)$ még nőhet (lásd az 5. megfigyelést).

6. Megfigyelés. *$G = (V, E)$ gráfban:*

- *$u \in V$ csúcs K értéke megváltozik $\Longrightarrow$ minden szomszédos $v \in V$ csúcs $MCD(v)$ értéke változhat*
- *$u \in V$ csúcs MCD értéke megváltozik $\Longrightarrow$ minden szomszédos $v \in V$ csúcs $PCD(v)$ értéke változhat*

Tehát, egy $u \in V$ csúcs a K értékének megváltozása maximum két lépésen belül okozhat változást a gráfban.

A megfigyelés szerint tehát egy él beszúrása vagy törlése során az algoritmus először meghatározza a gyökércsúcsot, majd újraszámolja az RCD értékeit. Ezután frissíti minden érintett csúcs K értékét, végül a változásoknak megfelelően aktualizálja a csúcsok RCD értékeit.

A teljesítmény javítása érdekében a Traversal algoritmus az **InsertEdge** függvényre bevezeti a gyökér-figyelést. A módszer arra épül, hogy ahhoz, hogy bármely csúcsnak megváltozzon a K értéke egy beszúrás vagy törlés hatására, ahhoz gyökércsúcs K értékének is változnia kell. Ezért a beszúró algoritmus során ügyelünk arra, hogy ha a gyökércsúcs K értéke nem nőhet (azaz $K(u) \geq PCD(u)$), akkor a további felesleges munka megspórolható.

A gyökér figyelés miatt nem biztos, hogy minden *core degree* értékre szükség van az algoritmusnak, ezért a korábbi beszúró algoritmussal ellentétben menet közben számítja, ha szükség van arra.

6. Tétel. *A Traversal algoritmus pontosan megtalálja azokat a $v \in V$ csúcsokat, amelyre $K(v)$ változik.*

Algorithm 6: TRAVERSAL:
INSERTEDGE$(G(V,E), K(), u_1, u_2)$

Data: G: the graph, K: max-k values, MCD: max-core degrees, PCD: purecore degrees, (u_1,u_2): inserted edge
$r \leftarrow u_1$ ▷ Set the root
if $K(u_2) < K(u_1)$ **then** $r \leftarrow u_2$
$G \leftarrow G \cup (u_1, u_2)$ ▷ Add the edge into G
PREPARERCDS
▷ Perform a traversal over vertices that have root's K value, while evicting the ones that cannot be a part of a k+1-core
$S \leftarrow$ empty stack ▷ To perform DFS
`visited`$[v] =$ **false**, $\forall v \in V$ ▷ To perform DFS (lazy init)
`evicted`$[v] =$ **false**, $\forall v \in V$ ▷ To remember evicted vert. (lazy init)
`cd`$[v] = 0, \forall v \in V$ ▷ To find vertices to be evicted (lazy init)
$k \leftarrow K(r)$ ▷ Remember the K value of the root
`cd`$[r] \leftarrow PCD(r)$ ▷ Set `cd` of root
S.push(r); `visited`$[r] \leftarrow$ **true**
while not S.empty() **do** ▷ Do a DFS traversal
  $v \leftarrow S$.pop()
1 **if** `cd`$[v] > k$ **then** ▷ Vertex is currently part of a k+1-core
    for each $(v, w) \in E$ **do**
      ▷ Neighbouring vertex currently part of a k+1-core
      if $K(w) = k$ **and** $MCD(w) > k$ **and not** `visited`$[w]$ **then**
        S.push(w); `visited`$[w] \leftarrow$ **true**
        ▷ Use + as `cd`$[w]$ may be < 0 due to evictions
        `cd`$[w] \leftarrow$ `cd`$[w] + PCD(w)$
  else ▷ Vertex cannot be part of a k+1-core
    if not `evicted`$[v]$ **then** ▷ Recursively perform eviction
      PROPAGATEEVICTION$(G, K,$ `cd`, `evicted`$, k, v)$
for each v s.t. `visited`$[v]$ **do** ▷ Find visited vertices
  if not `evicted`$[v]$ **then** ▷ If not evicted as well
    $K(v) \leftarrow K(v) + 1$ ▷ The vertex is part of a k+1-core
RECOMPUTERCDS

InsertEdge (Traversal):

Algorithm 7: PROPAGATEEVICTION($G(V, E), K(), \texttt{cd}[], \texttt{evicted}[], k, v$)

```
Data: G: the graph, K: max-k values, cd: cd values, evicted:
      evicted values, k: max-k of root, v: evicted vertex
evicted[v] ← true
for each (v, w) ∈ E do
    if K(w) = k then
1       cd[w] ← cd[w] − 1
2       if cd[w] = k and not evicted[w] then
            PROPAGATEEVICTION(G, K, cd, evicted, k, v)
```

PropagateEviction:

Az **InsertEdge** Traversal változata hasonlóképpen indul az elődjéhez. Gyökércsúcs meghatározása után előkészíti az RCD értékeket, majd inicializálja a szükséges adatszerkezeteket: az S vermet, a *visited* tömböt a már bejárt csúcsok eltárolására, az *evicted* tömböt a gyökér figyelés során kihagyott csúcsok számára, a *cd* tömböt a *core degree* értékeknek. Az algoritmus mélységi bejárást használ, ezért egy vermet használa csúcsok gyűjtésére, majd bejárására. Ha egy $v \in V$ csúcs részt vehet egy $k+1$-magban, akkor minden w szomszédját berakja az S verembe. Ellenkező esetben elindítja a rekurzív **PropagateEviction**-t a v csúcsból. A rekurió során a v csúcs minden olyan x szomszédjának csökkenti a *cd* értékét, amire $K(x) = k$. Ha $cd[x] = k$, akkor az a csúcs sem vehet részt a $k+1$-magban, ezért rekurzívan meghívja rá a **PropagateEviction**-t. Amikor a mélységi bejárás végzett, minden nem kihagyott (*evicted*) csúcs K értékét növeli 1-gyel. Végül egyetlen teendő maradt, frissíteni az RCD értékeket.

Az éltörlés a Traversal módszer szerint a következőképpen működik: A beszúráshoz hasonlóképpen előkészíti a változókat és tömböket. Ebben az esetben is bejárás közben frissíti a *cd* értékeket. Ha egy v csúcsra teljesül, hogy $cd[v] < K(v)$, akkor egy rekurzív törlést indul v-ből: minden szomszéd w csúcs *cd* értékét csökkenteni kell 1-gyel, ha $K(w) = k$. Ha w-re is teljesül, hogy $cd[w] < K(w)$, akkor egy újabb rekurzió indul (w-ből). A rekurziók végén az RCD értékek frissítése után az algoritmus terminál.

6.4. Implementáció

Az algoritmus gyorsaságának egyik kulcsa, hogy kerülni kell minden olyan műveletet, amit a teljes gráfon el kell végezni. Ezért az olyan tömbök, mint a $visited, evicted, cd$ és társai inicializálása során érdemes lusta kiértékelésű tömböt használni. Megfelelő hasítással elérhető egy konstans idejű lekérés,

valamint tárterület spórolható azzal, hogy az alapértelmezett értékeket nem tárolja el a program. Ugyanakkor a lusta tömbök gyengébben teljesítenek, ha azok túl nagyra nőnek. A mérések során látható majd, hogy a lusta tömbök használata a legtöbb esetben megtérül.

Több algoritmus alkalmaz *cd* szerinti rendezést. A *cd* értékek többsége egy szűk határon belül lesz, ezért az edényrendezés egy megfelelő hasítással $O(n)$ rendezést és $O(1)$ elérést tesz lehetővé.

Az algoritmus működéséhez a generált és valós gráfokat is használtak a szerzők. A tesztekhez C++ implementációt használtak két Intel Xeon E5520 2.27GHz processzorral és 48 GB RAM memóriával.

A generált adatok:

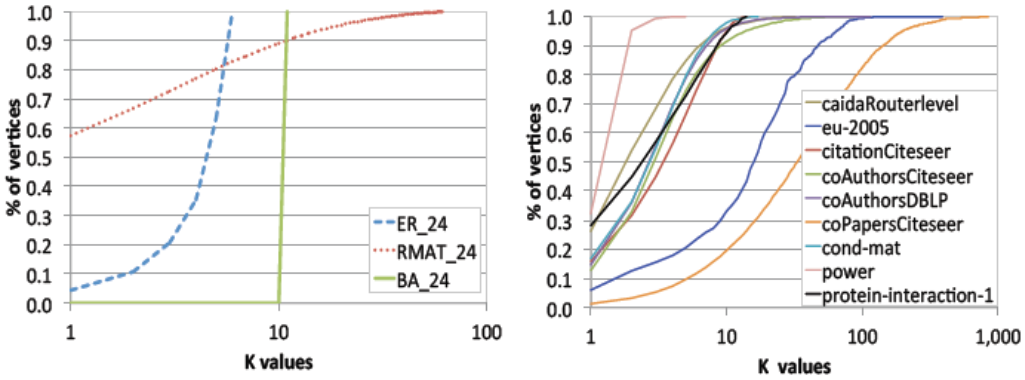
- ER: Erdős-Rényi modellen alapuló gráf: $p = 0.1, |E|/|V| = 8$
- BA: Barabási-Albert modellen alapuló gráf
- $RMAT$: R-MAT generátor: $[0.45, 0.25, 0.20, 0.10]$ valószínűségekkel

Graph file	Number of vertices	Number of edges	Maximum degree	Average degree	Max k
caidaRouterLevel	192,244	609,066	1,071	6.336	32
eu-2005	862,664	16,138,468	68,963	37.415	388
citationCiteseer	268,495	1,156,647	1,318	8.616	15
coAuthorsCiteseer	227,320	814,134	1,372	7.163	86
coAuthorsDBLP	299,067	977,676	336	6.538	114
coPapersCiteseer	434,102	16,036,720	1,188	73.885	844
cond-mat	16,726	47,594	107	5.691	17
power	4,941	6,594	19	2.669	5
protein-interaction-1	9,673	37,081	270	7.667	14

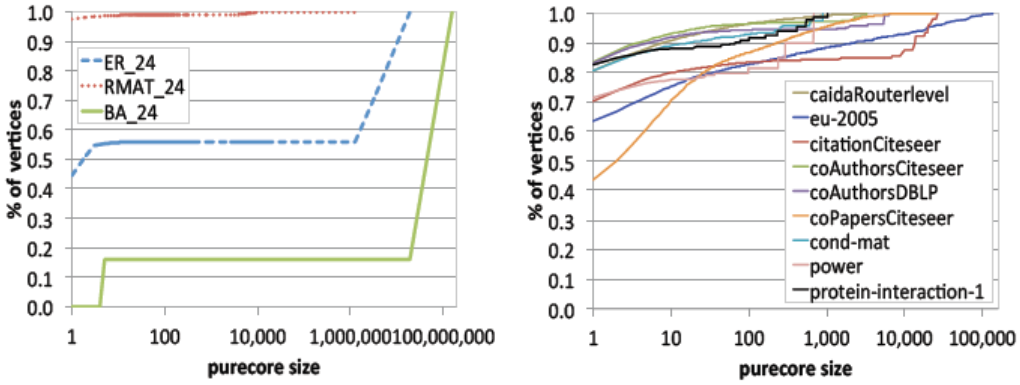
2. ábra. Valós adatokon alapuló gráfok tulajdonságai: csúcsok száma, élek száma, legnagyobb fokszám, átlagos fokszám, maximális k

A valós adatok, forrás: 10th DIMACS[17]

- $eu-2005$ és $caidaRouterLevel$: Európai domain hálózatok
- $citationCiteseer$, $coAuthorsCiteseer$, $coAuthorsDBLP$, $coPapersCiteseer$: társszerzői és hivatkozási gráfok
- $cond-mat$: Condensed matter collaboration graph (akármi is az...)
- $power$: Villamos hálózat
- $protein-iteraction-1$: fehérjék kölcsönhatásai



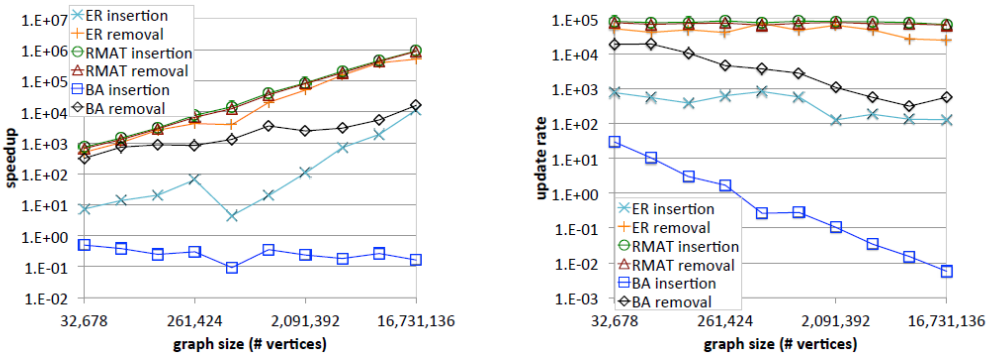
3. ábra. K értékek eloszlása. Bal oldalon a generált, jobb oldalon a valós gráfokban.



4. ábra. Purecore halmazok méreteinek összege a generált (bal) és valós (jobb) gráfokban.

Íme néhány grafikon a különböző gráfok csúcsairól. A szintetikus gráfoknál a Barabási-Albert modell jók kivehető, hiszen minden csúcsra $K = 11$. Ez egyben hatalmas, akár több százmilliós purecore halmazokat is eredményez, ahogy az a következő gráfon látható. Az R-MAT gráf csúcsainak viszont csaknem 60%-a nagyon alacsony K értékekkel rendelkezik. Ennek megfelelően a purecore halmazok is kicsik.

A valós gráfok hasonló képet mutatnak. Mindegyikre jellemző, hogy a csúcsok 80%-ának 100 csúcsnál kisebb purecore halmaza van. Emiatt várhatóan jól teljesít majd az algoritmus ezekben az esetekben. Mivel a valós adatok többnyire statikusak, ezért a váltzásokat egy csúszó ablakkal szimulálták.



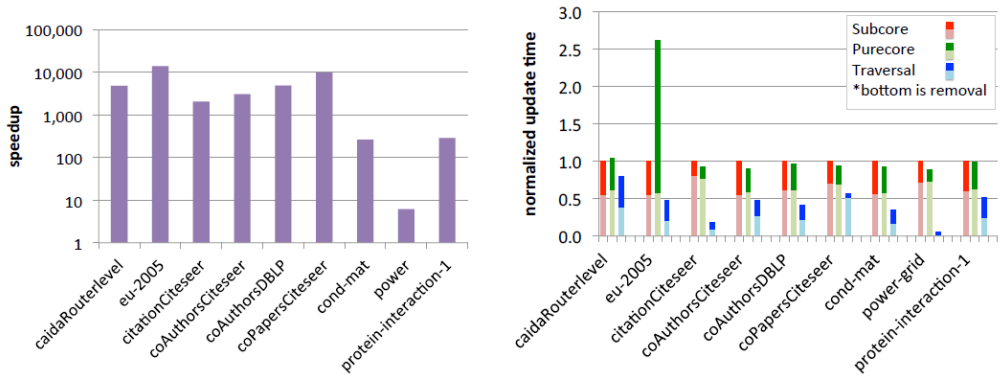
5. ábra. Traversal algoritmusok skálázhatósága.

A következő kísérletben a Traversal algoritmus skálázhatóságát vizsgálták a különböző szintetikus gráfokon.

A bal oldali ábrán az a sebességnövekedés látható, ami a nem inkrementális algoritmusokhoz képest érhető el. A vízszintes tengelyen a gráfok mérete növekszik 2^{15}-től 2^{24}-ig. A legjobb skálázhatóságot (6 nagysegrend) az R-MAT modellű éri el, köszönhetően a nagyon kicsi purecore halmazoknak. A Barabási-Albert gráfokra viszont éppen a nagy összefüggőség jellemző. Ennek megfelelően az algoritmus még a nem inkrementális változatnál is rosszabbul teljesít. Az előző vizsgálatok szerint azonban a valós adatokon alapuló gráfok nem rendelkeznek ilyen tulajdonságokkal. Él törlésért nagyjából hasonlóan viselkedik mindhárom modell.

A jobb oldali grafikon a frissítési rátát mutatja, azaz, hogy másodpercenként hány élt képes feldolgozni az algoritmus. Látható, hogy az R-MAT típusú gráfok teljesítménye nem függ a mérettől, és az Erdős-Rényi gráf is viszonylag stabilan reagál a növekvő csúcsszámra, a Barabási modell azonban ismét drasztikusan lassítja a feldolgozás sebességét.

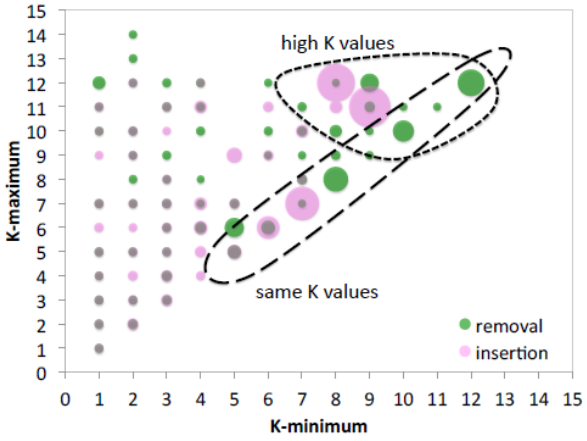
A következő elemzésben az egyes algoritmusok összehasonlítása látható. A mérés egy-egy él beszúrására, illetve törlésére koncentrál. A bal oldali grafikonon a subcore algoritmus teljesítménye látható a nem-inkrementális algoritmushoz viszonítva. Látható, a gyorsulás 6.2-től 14 000-szeresig terjed.



6. ábra. Bal: Subcore algoritmus teljesítménynövekedése a nem-inkrementális algoritmusokhoz viszonyítva. Jobb: Átlagos frissítési idők a különböző algoritmusok esetén.

A jobb oldali ábra a cikkben bemutatott három módszert hasonlítja össze. Látható, hogy a Purecore bizonyos esetekben akár a Subcore algoritmusnál is rosszabban teljesíthet. Bár kisebb halmazzal dolgozik, a Purecore algoritmus lényegesen több számítást végez egy-egy csúcshoz, így a halmaz méretéből adódó előny nem mindig kifizetődő. A traversal módszer minden esetben a legjobbnak bizonyult.

Az utolsó kísérlet a traversal algoritmus futásidejét vizsgálja, ahogyan az reagál az egyes K értékekre. A beszúrás rózsaszín, a törlés zöld színnel van jelölve. Ha $K(u) \leq K(v)$, akkor u az x-tengelyen, egyébként az y-tengelyen látható. Alacsony K értékekre a grafikon nem mutat túl nagy változatosságot. Két érdekes eset azonban látható az ábrán. A futásidő meglehetősen változó képet mutat, ha a két csúcs K értéke nagy, vagy azonosak.



7. ábra. Él beszúrás és törlés futási ideje a végpontok K értékei szerint.

7. További tervek

A cikkben jól látható, hogy a csúcs környezetére vonatkozó értékek felvételével javítható volt az algoritmus teljesítménye. Érdemes lehet tehát meg-

vizsgálni, hogy mekkora környezettel érthető el a legoptimálisabb futásidő.

Érdemes lehet megvizsgálni, hogy hogyan alkothatók olyan modellek, amik egyszerre több él hatékony módosítását teszik lehetővé; van-e hatékonyabb megoldás csúcsok törlésére, mint az élek egyessével történő kitörlése.

A bemutatt módszereket átültethetők-e irányított, illetve összetett gráfokkal kapcsolatos alkalmazásokba is?

Hivatkozások

[1] A. E. Sarıyüce, B. Gedikz, G. Jacques-Silva, K. L. Wu, Ü. V. Çatalyürek. Streaming Algorithms for k-core Decomposition, VLDB 2013

[2] S. B. Seidman. Network structure and minimum degree. Social Networks, 5(3):269–287, 1983.

[3] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. The Computing Research Repository (CoRR), cs.DS/0310049, 2003.

[4] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In International Conference on Advances in Social Network Analysis and Mining (ASONAM), pages 87–93, 2011.

[5] A. Verma and S. Butenko. Network clustering via clique relaxations: A community based approach. 10th DIMACS Implementation Challenge, 2011.

[6] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decomposition: A tool for the visualization of large scale networks. The Computing Research Repository (CoRR), abs/cs/0504107, 2005.

[7] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In IEEE International Conference on Data Engineering (ICDE), pages 1049–1060, 2012.

[8] M. Gaertler. Dynamic analysis of the autonomous system graph. In International Workshop on Inter-domain Performance and Simulation (IPS), pages 13–24, 2004.

[9] F. Ozgul, Z. Erdem, C. Bowerman, and C. Atzenbeck. Comparison of feature-based criminal network detection models with k-core and n-clique. In International Conference on Advances in Social Network Analysis and Mining (ASONAM), pages 400–401, 2010.

[10] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. k-core organization of complex networks. Physical Review Letters, 96, 2006.

[11] T. Luczak. Size and connectivity of the k-core of a random graph. Discrete Math, 91(1):61–68, 1991.

[12] M. Baur, M. Gaertler, R. G¨orke, M. Krug, and D. Wagner. Augmenting k-core generation with preferential attachment. Networks and Heterogeneous Media, 3(2):277–294, 2008.

[13] B. Balasundaram, S. Butenko, and I. Hicks. Clique relaxations in social network analysis: The maximum k-plex problem. Operations Research, 59:133–142, 2011.

[14] R. Andersen and K. Chellapilla. Finding dense subgraphs with size bounds. In Workshop on Algorithms and Models for the Web Graph (WAW), pages 25–37, 2009.

[15] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In IEEE International Conference on Data Mining (ICDM), pages 201–210, 2011.

[16] R.-H. Li and J. X. Yu. Efficient core maintenance in large dynamic graphs. CoRR, abs/1207.4567, 2012.

[17] DIMACS. 10th DIMACS implementation challenge. http://www.cc.gatech.edu/dimacs10.