

## پروژه‌ی نهایی

لایک‌های مشترک، بین پیج‌های verify شده‌ی

### • مقایسه‌ی ابزارها

#### • سه ابزار انتخابی

در وله‌ی اول، دو نوع کلی از بزارها را در اختیار داریم: ۱-کتابخانه interactive و رسم گراف کارایی دارند. البته اکثر آن‌ها نرم‌افزارها به طور کلی بیشتر برای کارهای گرافی دارند. برنامه‌نویس کار کردن با خود کتابخانه‌ها به طور مستقیم آزادی عمل بیشتری حاصل خواهد کرد.

بعد از بررسی ابزارهای موجود، کتابخانه‌های networkkit و graph-tool و نرم‌افزار gephi به نظر ما مناسب‌ترین ابزارها رسیدند. در ادامه ابتدا دلیل انتخاب این سه ابزار را بیان می‌کنیم و بعد برخی الگوریتم‌ها را در این سه ابزار مقایسه می‌کنیم.

برای رسم گراف، یک ابزار کافی بود. چرا که همواره می‌توان الگوریتم‌ها را با استفاده از کتابخانه‌ها اجرا کرد، و نتیجه را با استفاده از ابزار، رسم کرد. از طرفی، gephi برای رسم گراف‌هایی با میلیون‌ها نود مناسب نیست، اما برای گراف انتخابی ما در این پروژه، با ۱۰۰ نود می‌تواند رسم را به خوبی انجام دهد. gephi یکی از قدیمی‌ترین ابزارهای رسم گراف است و علاوه بر documentation خوب، تعداد زیادی از معیارها برای انجام clustering، فیلتر کردن گراف و ... را پیاده‌سازی کرده. همچنین می‌توان به راحتی نتیجه‌ی الگوریتم‌های اجرا شده توسط کتابخانه‌ها را در gephi رسم کرد.

اما در مورد کتابخانه‌های انتخابی، از بین networkkit و graph-tool از آنجایی که wrapper ای بر روی این کتابخانه‌ها در زبان C++ هستند، سرعت بسیار بیشتری در مقایسه با کتابخانه‌های خود پایتون، مانند NetworkX دارد. البته از نظر documentation بسیار قوی‌تر است. اما برای انجام محاسبات در این مقیاس، استفاده از آن تقریباً غیر ممکن است. هر یک از کتابخانه‌های graph-tool و networkkit هم تفاوت‌هایی دارند که در قسمت مقایسه، با تفصیل بیشتری بررسی خواهیم کرد.

## • مقایسه

به طور کلی، رسم گراف با دو کتابخانه‌ی networkkit و graph-tool کار بسیار پیچیده و زمانبری است و در مقابل gephi ابزار بسیار مناسبی برای این کار است در مقابل برای اجرا کردن الگوریتم‌ها گزینه‌ی خیلی مناسبی نیست. در رابطه با مقایسه‌ی این دو کتابخانه هم، در حالت کلی هر کدام در برخی از بخش‌ها توابع بیشتری را پیاده‌سازی کرده‌اند. برای مثال، در زمینه‌ی community detection الگوریتم‌های مشهور بیشتر در networkkit قابل پیگیری بودند تا graph-tool. همچنین به نظر می‌رسد networkkit در بسیاری از موارد سرعت بیشتری دارد.

برای بررسی با جزئیات بیشتر، جدول‌های بعدی زمان اجرای الگوریتم‌ها توسط هرکدام را نشان میدهد. برای این کار ما از دیتاست‌های زیر مجموعه‌ی داده‌ی اصلی که مربوط به facebook استفاده کردیم و ۵ subgraph با نام و تعداد نودهایی که در زیر هر جدول نوشته شده است به کار گرفتیم.

زمان‌های اجرا در jupyter notebook با استفاده از magic command %%timeit با سینتکس اندازه‌گیری شده‌اند که با چندین بار اجرای یک خط کد (بسته به زمان اجرای هرکدام، تعداد تکرار متفاوت است) میانگین زمان اجرا را در آن‌ها بدست می‌آورد. همچنین هر بار ۷ دور اجرا می‌کند، که در هر دور تعداد تکرار متفاوت است (ممکن است ۱۰۰ بار لوب تکرار کند، یا ۷ بار ۱ لوب). در نتیجه تعداد تکرار در این جدول‌ها، تعداد لوب در هر یک از آن ۷ دور است.

الگوریتم	networkkit	زمان متوسط در networkkit	تعداد تکرار (networkkit)	graph-tool	زمان متوسط در graph-tool	تعداد تکرار (graph-tool)
label propagation	5.54 ms	100	—	—	—	—
louvain method	4.71 ms	100	—	—	—	—
betweenness centrality	2.03 s	1	728 ms	—	1	—
closeness centrality	296 ns	1000000	1.03 s	—	1	—
degree centrality	207 ns	1000000	—	—	—	—
eigen vector centrality	201 ns	1000000	1.99 ms	—	100	—
page rank centrality	205 ns	1000000	2.93 ms	—	100	—
katz centrality	211 ns	1000000	4.37 ms	—	100	—
HITS	—	—	3.34 ms	—	100	—

زیرگراف TV با ۳۸۹۲ نود و ۱۷۲۶۲ یال

الgoritم	زمان متوسط در networkit	تعداد تکرار (networkit)	زمان متوسط در graph-tool	تعداد تکرار (graph-tool)
label propagation	19.9 ms	10	—	—
louvain method	16.5 ms	10	—	—
betweenness centrality	18.3 s	1	7.63 s	1
closeness centrality	252 ns	1000000	8.75 s	1
degree centrality	188 ns	1000000	—	—
eigen vector centrality	185 ns	1000000	9.66 ms	100
page rank centrality	199 ns	1000000	18.1 ms	10
katz centrality	211 ns	1000000	4.37 ms	100
HITS	—	—	16.1 ms	10

زیرگراف با ۷۰۵۷ نود و ۸۹۴۵۵ یال governents

الgoritم	زمان متوسط در networkit	تعداد تکرار (networkit)	زمان متوسط در graph-tool	تعداد تکرار (graph-tool)
label propagation	22.3 ms	10	—	—
louvain method	22.9 ms	10	—	—
betweenness centrality	35.5 s	1	14.9 s	1
closeness centrality	279 ns	1000000	16.6 s	1
degree centrality	202 ns	1000000	—	—
eigen vector centrality	204 ns	1000000	6.23 ms	100
page rank centrality	211 ns	1000000	16 ms	100
katz centrality	211 ns	1000000	196 ms	10
HITS	—	—	9.94 ms	100

زیرگراف با ۱۱۵۶۵ نود و ۶۷۱۱۴ یال public figures

الگوریتم	زمان متوسط در networkkit	تعداد تکرار (networkkit)	زمان متوسط در graph-tool	تعداد تکرار (graph-tool)
label propagation	112 ms	10	—	—
louvain method	75.7 ms	10	—	—
betweenness centrality	—	—	—	—
closeness centrality	272 ns	1000000	1 min 55 s	1
degree centrality	201 ns	1000000	—	—
eigen vector centrality	204 ns	1000000	45.1 ms	100
page rank centrality	203 ns	1000000	55.3 ms	10
katz centrality	196 ns	1000000	183 ms	10
HITS	—	—	67.5 ms	10

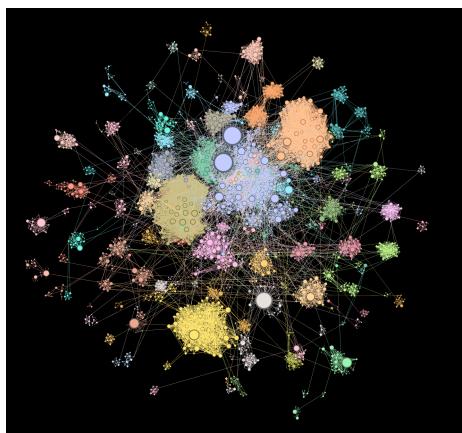
زیرگراف new sites با ۲۷۹۱۷ نود و ۲۰۶۲۵۹ یال

الگوریتم	زمان متوسط در networkkit	تعداد تکرار (networkkit)	زمان متوسط در graph-tool	تعداد تکرار (graph-tool)
label propagation	215 ms	1	—	—
louvain method	241 ms	1	—	—
betweenness centrality	—	—	—	—
closeness centrality	269 ns	1000000	—	—
degree centrality	187 ns	1000000	—	—
eigen vector centrality	179 ns	1000000	321 ms	1
page rank centrality	200 ns	1000000	197 ms	1
katz centrality	196 ns	1000000	8.25 s	1
HITS	—	—	511 ms	1

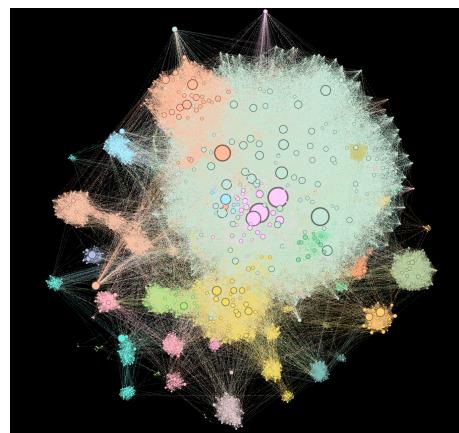
زیرگراف artists با ۸۱۹۳۰۶ نود و ۵۰۵۱۵ یال

در دو زیرگراف آخری، به دلیل بزرگ بودن گراف، برخی از الگوریتم‌ها امکان اجرا بر روی سیستم ما را نداشتند. اما از روند رشد زمان آن‌ها در گرافهای قبلی و نسبت دو کتابخانه به یکدیگر، می‌توان به نتایج مطلوبی رسید.

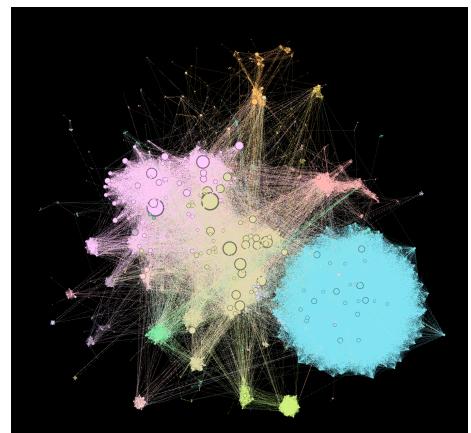
شكل‌های زیر هم گرافهای بررسی شده هستند که در gephi رسم شده‌اند. رنگ نودها با معیار modularity و اندازه‌ی آن‌ها با امتیاز page rank مشخص شده است. و برای آزمون روش‌های مختلف layout از این‌هاهای متفاوتی برای رسم آن‌ها استفاده شده است.



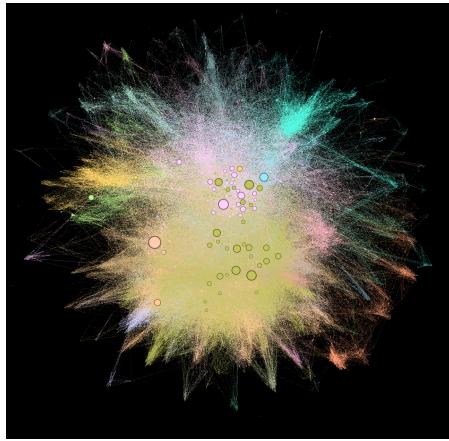
TV shows



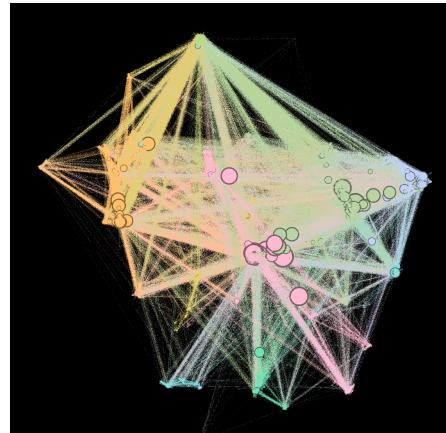
governments



public figure



new sites

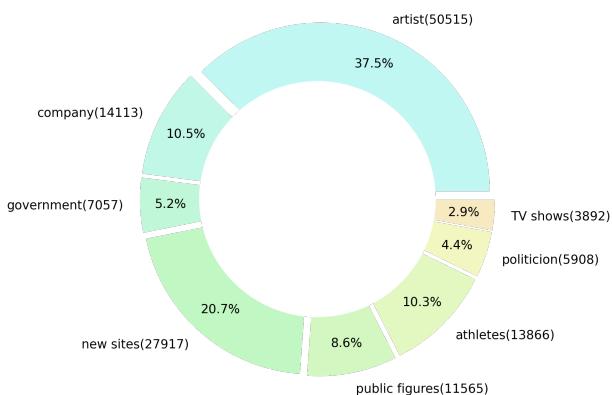


artist

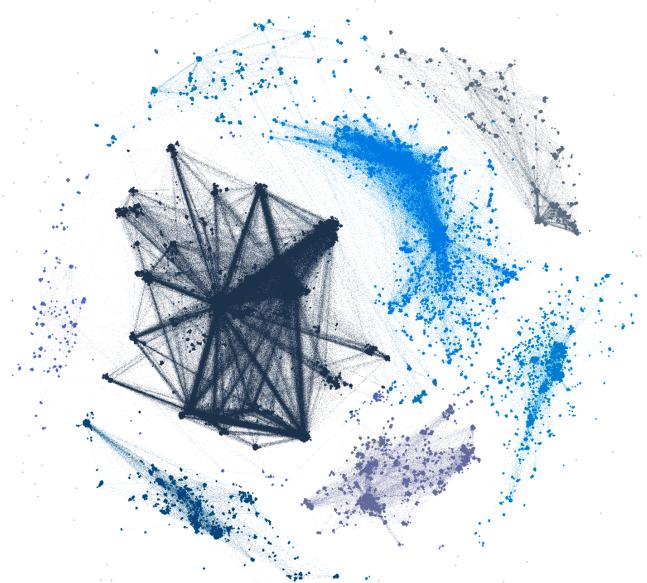
# • تحلیل گراف اصلی

## 1. معرفی گراف

گراف انتخابی من، دیتاست لایک‌های مشترک بین صفحات verify شده‌ی facebook (صفحات تیک‌آبی‌دار!) بود که در مجموع ۱۳۴۸۳۳ نود و ۱۳۸۰۲۹۳ یال دارد. این صفحات در ۸ دسته قرار دارند که در نمودار زیر می‌توان نام و سهم آن‌ها در این گراف را بررسی کرد.



همچنین تصویری کلی که با نرم‌افزار gephi رسم شده و هر رنگ مخصوص به یک دسته از ۸ دسته‌ی گفته شده است، در زیر قابل مشاهده است.



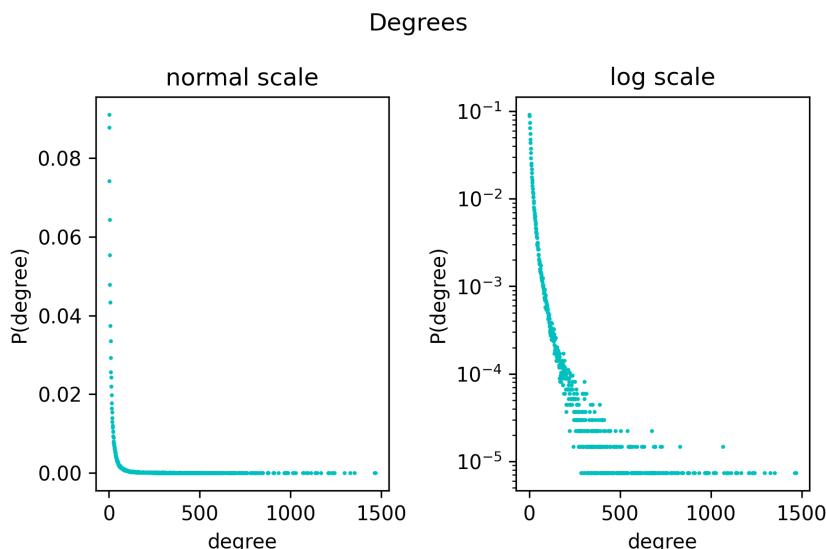
\*\* فایل کدها: بررسی گراف اصلی (بهغیر از تحلیل‌هایی که به کمک gephi صورت گرفته) داخل فایل Facebook.ipynb و ۵ گراف بررسی شده برای قسمت اول، همگی در ۵ فایل notebook به نام همان کلاستر اصلی‌شان، قابل دسترس است.

## 2. پارامترهای توپولوژی در گراف تجمعی

در جدول زیر، پارامترهای توپولوژی گراف که مقدار عددی دارند به طور خلاصه آورده شده است.  
در ادامه با جزئیات بیشتر، توپولوژی بررسی خواهد شد.

مقدار	پارامترها
134833	تعداد نودها
1380293	تعداد یالها
1	کمترین درجه نود
1469	بیشترین درجه نود
353.63	میانگین درجات در گراف
2.81	گاما
0.000152	چگالی گراف
0.256	ضریب خوشبندی
8	تعداد مولفه های همبند
0.074	ضریب همبستگی درجه نودها

توزیع درجات این گراف، همانطور که در شکل زیر مشاهده می کنید، به نظر می رسد که از توزیع پیروی می کند، که در نتیجه یعنی گراف ما، یک گراف scale-free powerlaw است.



کتابخانه powerlaw به ما امکان بررسی و مقایسه توزیع‌های مختلف بر روی توزیع درجات را می‌دهد. و ما با استفاده از تابع `distribution_compare` دو توزیع `powerlaw` و `exponential` را مقایسه کردیم. نتیجه‌ی این مقایسه به این صورت است:

$632.69 = R$  (نسبت likelihood دو توزیع)

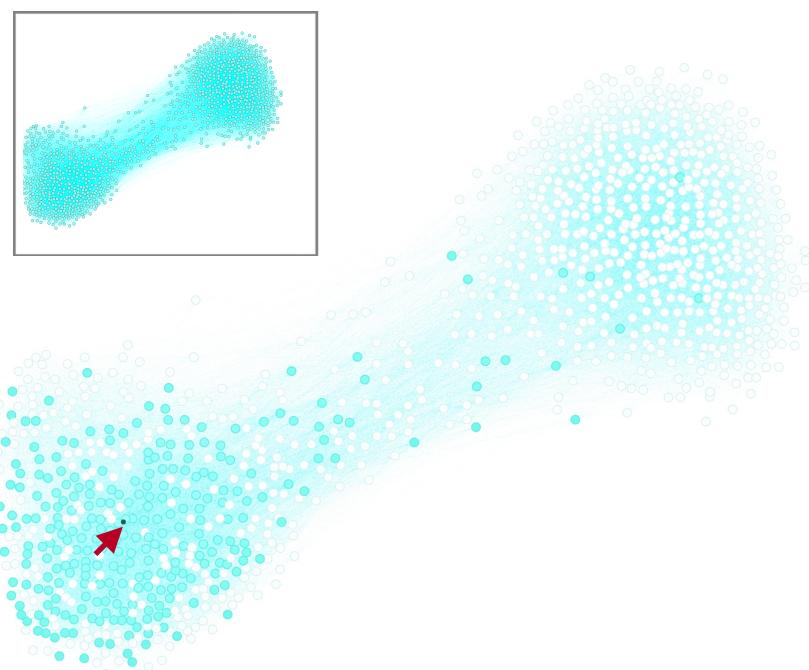
$P = 2.39 \times 10^{-30}$  (p-value) (مقدار قابل اعتنا بودن R یا در واقع)

از نتایج بالا، مشخص است که توزیع درجات با `powerlaw distribution` به مراتب قابل تفسیرتر است.

### 3. معیارهای مرکزیت

#### k-shell

در این حالت، هر نод، مقدارش برابر با بزرگ‌ترین  $k$ -shell‌ای است که در آن جای می‌گیرد. هر یک  $k$ -shell از نودهای گراف است که هر نod عضو آن، حداقل  $k$  یال به نودهای همان  $k$ -shell دارد. در نتیجه زمانی که ما تنها ادرصد بالای این معیار را نمایش می‌دهیم، توقع دیدن کلاسترها بزرگ و `dense` را داریم. در تصویر سمت چپ، ادرصد بالای نودها از نظر عضویت در بزرگ‌ترین  $k$ -shell آورده شده و در سمت راست، نودهای رنگی همسایه‌های نود مشخص شده هستند که تاییدی بر ارتباط تنگاتنگ این نودها و `dense` بودن  $k$ -shell آنها است. البته با توجه به اینکه کلاستر `artist` بیش از ۳۷ درصد از گراف را تشکیل داده، به نظر می‌رسد که این نودها همگی متعلق به Giant Component گراف می‌باشند.



زمانی که تعدادی نود در شبکه‌ی اجتماعی مانند *facebook*، لایک‌های مشترک زیادی با یکدیگر دارند، می‌تواند چند دلیل داشته باشد؛ از جمله:

- تعدادی صفحه، که احتمالاً از یکدیگر حمایت می‌کنند در راستای بالا بردن بازدیدهای صفحه تنها به *interaction* با یکدیگر می‌پردازند. البته به نظر می‌رسد که در صفحات *verify* شده، این رفتار در دسته‌های ۵۰ یا ۶۰ تایی ( $k$ -core ۶۹) هم در گراف موجود است) بعید به نظر می‌رسد. به دو علت؛ اول اینکه افراد *verify* شده معمولاً به اندازه‌ی کافی سرشناس هستند و دوم اینکه حتی اگر صفحات عادی هم بودند، دسته‌هایی با این بزرگی کمی بعید به نظر می‌رسد.

- رفتار دیگری که در صفحات تایید شده منطقی‌تر به نظر می‌رسد، خطهای فکری هستند. برای مثال، صفحه‌ی فعالان سیاسی را در نظر بگیریم، از آنجایی که لایک‌ها و رفتار آنان به دقت توسط سایر کاربران بررسی می‌شود، احتمالاً تایید آن‌ها شامل پست‌هایی خواهد شد که از خط فکری آن‌ها حمایت می‌کند. پس به احتمال خیلی زیاد، برای مثال، افراد تاییدشده‌ی اصلاح طلب پست‌های لایک‌گرفته‌ی مشترک زیادی با یکدیگر دارند و بالعکس. می‌توان برای دسته‌ی صفحات هنری هم، این حدس را بررسی کرد. برای مثال در کشور ما، احتمالاً نوازنده‌های موسیقی سنتی کمتر پستی از نوازنده‌های موسیقی دارک‌متال لایک می‌کنند.

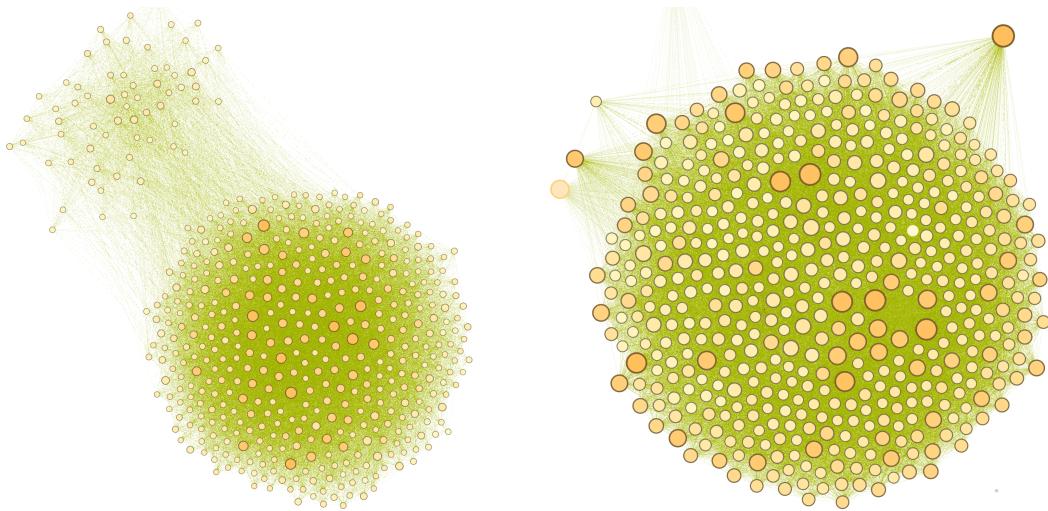
به نظر می‌رسد که برای این *dataset*، تعبیر دوم منطقی‌تر باشد. اگر داده‌هایی مانند نام صفحات یا حتی تخمینی از زمینه‌ی فعالیت آن‌ها داشتیم، می‌توانستیم درستی این موضوع را بررسی کنیم.

حالا با فرض صحت تحلیل دوم، علت مرکزی بودن این نودها چیست؟ و چرا باید برای ما مهم باشند؟ نودهایی که معیار  $k$ -core آن‌ها بالاست، به معنای عضویت در یک جامعه‌ی نسبتاً فشرده هستند. حالا برای مثال، در گروه‌های سیاسی، زمانی که جامعه‌ی یک نگرش خاص شکل بگیرد، می‌تواند به تدریج مطالب مربوط به آن نگرش را ترویج بیشتری کند و همین خود منجر به رشد بیشتر آن جامعه می‌شود. این مسئله مزايا و معایب دارد که از جمله مهم‌ترین معایب شاید دیده‌نشدن نگرش‌های اقلیتی و مرگ آن جامعه به صورت تدریجی باشد.

## HIT •

این روش اکثرا برای گراف‌های جهت‌دار معنا دارد. زمانی که این الگوریتم بر روی گراف‌های بدون جهت اجرا می‌شود، دیگر *hub* و *authority* معنایی ندارد و عملاً هر نod امتیاز مساوی‌ای در هر دوی

این‌ها به دست خواهد آورد. در دو شکل بعدی که با دو الگوریتم مختلف رسم شده‌اند، می‌توانیم ارتباط ۵۰۰ نودی که از نظر این معیارها، نسبت به دیگران برتری داشتند را ببینیم.



در این تصویر رنگ‌های تیره‌تر نمایان‌گر درجه‌ی بیشتر و سایز نودها تخمینی از امتیاز آن‌ها با الگوریتم HIT است.

با توجه به ماهیت گراف و همچنین جهت‌دار نبودن آن، نودهایی که امتیاز بالایی از این معیار بدست آورده‌اند در واقع لایک‌هایشان با صفحات زیادی همپوشانی داشته‌است. به عبارتی پستی را لایک کرده است، که تعداد زیادی پیج دیگر هم آن را لایک کرده‌اند. به نظر نمی‌رسد که هاب‌ها در این گراف اهمیت ویژه‌ای داشته باشند. نوعی تایید همگانی در آن‌ها دیده می‌شود که احتمالاً هدف‌دار هم نیست.

در شکل سمت چپ، که سعی شده دسته‌های مختلف از یکدیگر جدا شوند، به نظر می‌رسد نودهایی موفق به کسب امتیاز بالا شده‌اند که در کل درجه‌ی زیاد و یا `dense` باشد نداشته‌اند. ممکن است دسته‌ی بالای این عکس، از طریق حمایت از یک نظر خاص و لایک پست‌های هم‌راستا با نظرات همان‌گروه توانسته‌اند امتیاز بالای بدست بیاورند و تقریباً از شاخه‌ی دیگر مستقلند. البته با استفاده از الگوریتم HIT نودها در کلاستری بهم پیوسته و نسبتاً چگال هستند. این نودها احتمالاً عضو بزرگترین کلاستر ما یعنی صفحات هنرمندان هم بوده‌اند. زمانی که از این الگوریتم استفاده می‌کنیم، نودهای مهم کلاسترها کوچکتر شناسی چندانی برای قرار گرفتن در ردیف ادرصد بالای نودهای مهم، نخواهند داشت. در ادامه، با توجه به نتایج `page rank` باز هم به این مطلب برمی‌گردیم.

## page-rank •

الگوریتم page-rank هم در واقع برای گرافهای جهت دار طراحی شده، اما می توان مشابه HIT آن را روی گرافهای بدون جهت هم اجرا کرد. این الگوریتم با دخیل کردن احتمال، می تواند مشکلات الگوریتم HIT را پوشش بدهد. برای مثال، نودهایی که تقریباً از بقیه گراف - آن هم در یک کلاستر - مستقل هستند و با تایید و لایک یکدیگر به کار ادامه می دهند شانس کمتری برای بلعیدن توجهها در نتورک خواهند داشت! شکل زیر نتیجه اجرای این الگوریتم بر روی گراف است. (باز هم ۵۰۰ نودی که بالاترین امتیاز از نظر این معیار را داشته اند، رسم شده است).



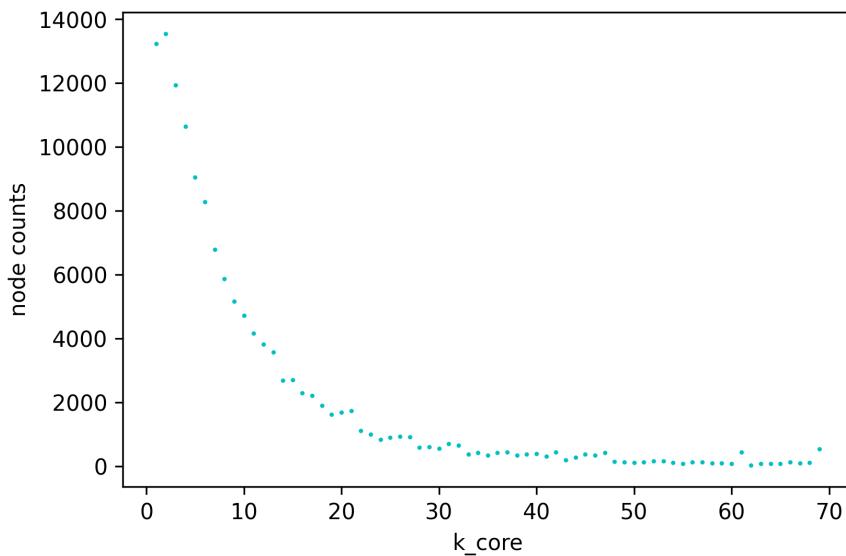
این معیارها هم سهم بیشتر خواهد داشت، اما به هر صورت ما موفق شدیم نودهای مهم در سایر کلاسترها را هم پیدا کنیم.

اگر به شکل اولیه ای که از گراف ارائه شد برگردیم؛ و با در نظر گرفتن اینکه ما در واقع ۸ کلاستر در گراف داشتیم، به نظر می رسد این الگوریتم برخلاف الگوریتم HIT موفق شده نودهایی که لایک های مشترک زیادی با سایر نودها داشته اند را در **هر کلاستر** پیدا کند. البته قطعاً زمانی که ما همه گراف را مدنظر قرار می دهیم کلاستری که سهم بیشتری از نظر تعداد نود دارد، در

## 4. انجمن‌های شبکه

### k-core

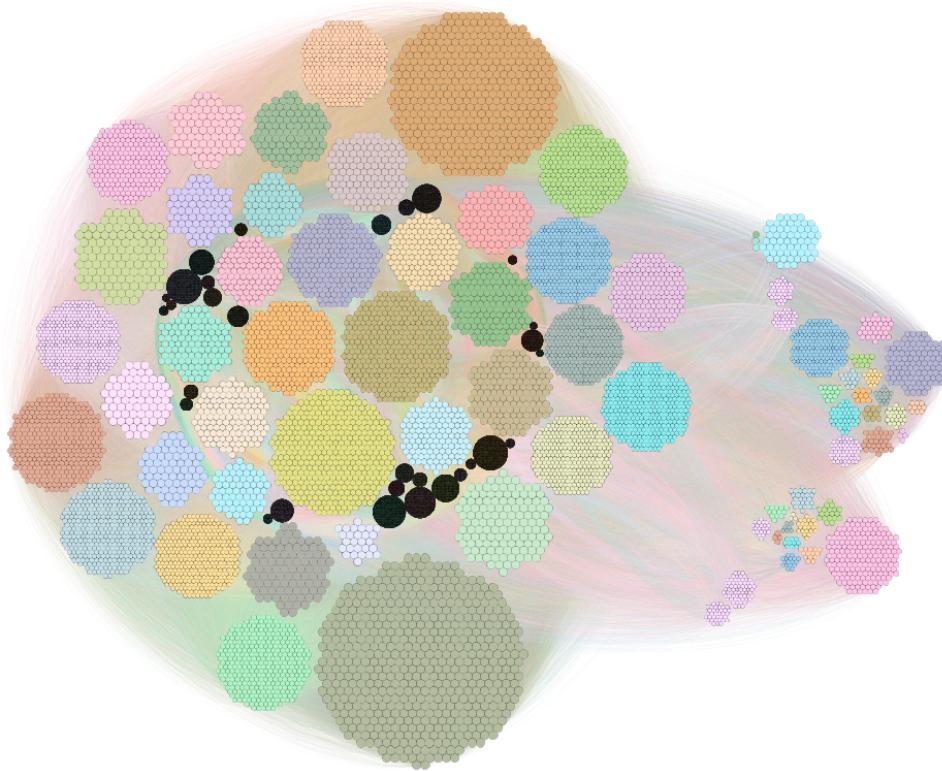
توزيع تعداد نودهای عضو در هر k-core در نمودار زیر آورده شده است:



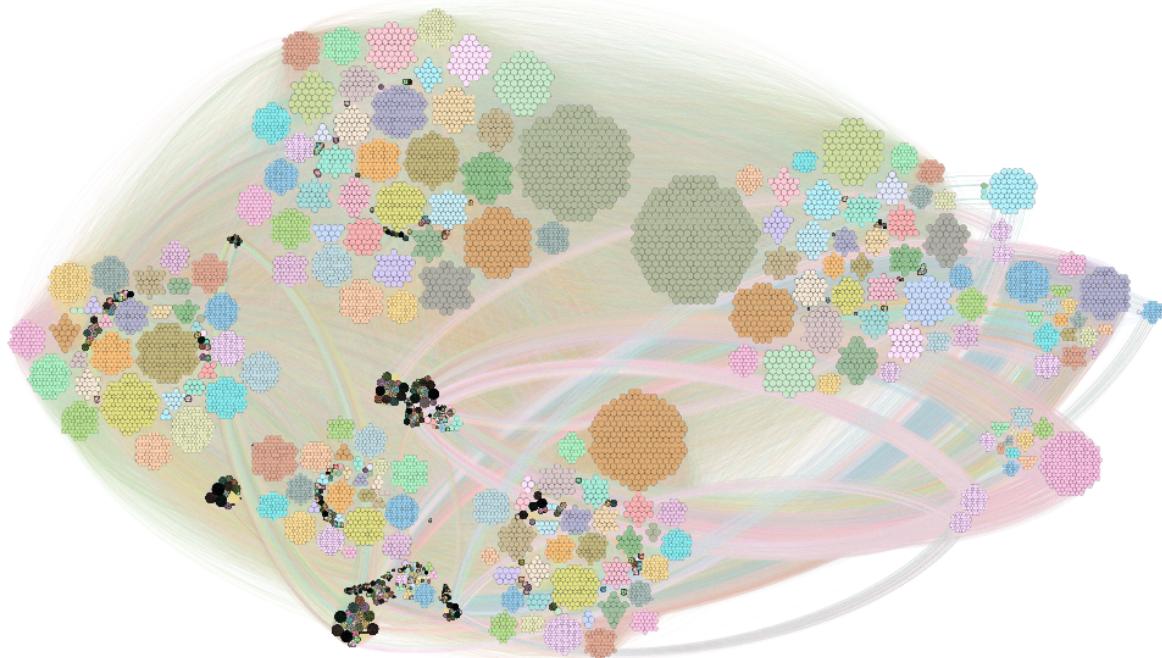
این نمودار نشان می‌دهد که ما تعداد زیادی دسته‌های کوچک داریم و تعداد دسته‌های بزرگ‌تر در مقایسه با آن‌ها کمتر است.

تصویر اولی که در اینجا می‌بینیم، اجرای circle pack layout در نرم‌افزار gephi است که در وهله‌ی اول، نودها را بر اساس ۸ دسته‌بندی گراف جدا کرده و بعد در داخل هر دسته، نودهای با  $k$  برابر را در کنار هم قرار داده است. با این روش می‌توانیم تخمین خوبی از فراوانی  $k$ ‌های مختلف در هر دسته داشته باشیم. همچنین سایز نودها، تابعی از  $k$ -core آن‌هاست.

تصویر دوم اما با کمک الگوریتم louvain رسم شده. به این ترتیب که در ابتدا نودها با توجه به دسته‌بندی‌ها جدا شده‌اند، سپس با استفاده از امتیاز modularity که با این الگوریتم محاسبه شده‌است در hirarchy دوم در کنار یکدیگر قرار گرفته‌اند؛ و در نهایت در هرکدام از این دسته‌ها، نودهای با  $k$ -core مشابه تشکیل دوایر ریزتر را داده‌اند. در این حالت، دو کلاستر با  $k$  نسبتاً کوچک، صرف اینکه در یکی از آن ۸ دسته قرار گرفته‌بودند با هم در یک دایره قرار نخواهند گرفت.



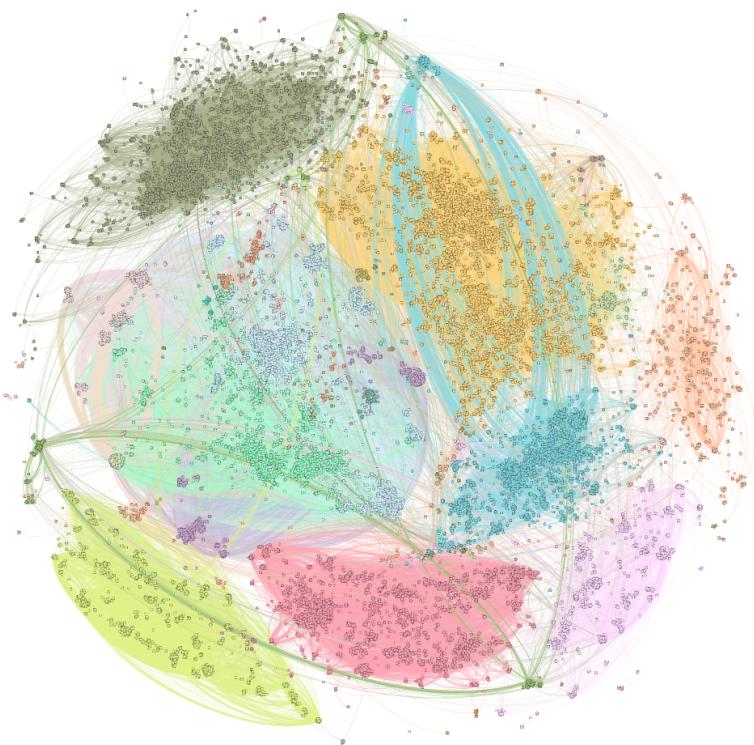
دسته‌بندی بدون معیار modularity



دسته‌بندی با کمک modularity

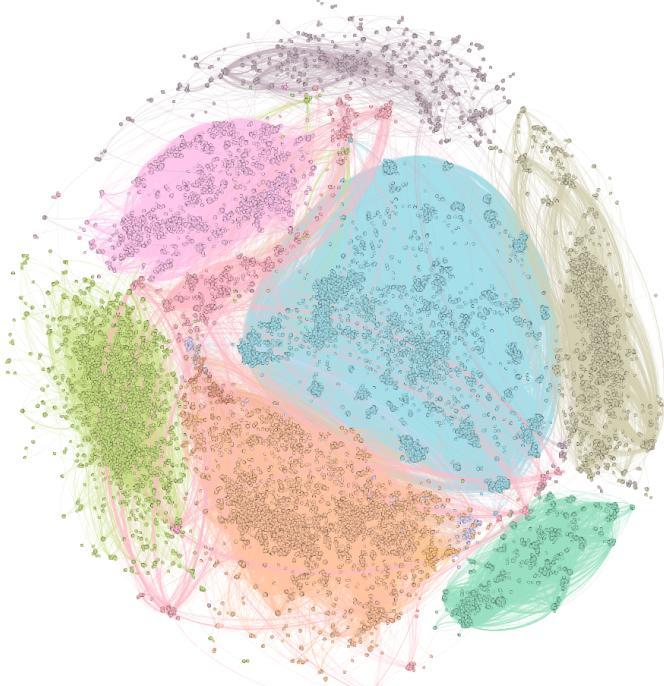
## louvain •

این روش که به خوبی در Gephi هم پیاده‌سازی شده، با دقت قابل قبولی کلاسترها اولیه‌ی این گراف را از هم جدا کرد و می‌توان علاوه بر آن‌ها، انجمن‌های داخلی هر دسته را هم مشاهده کرد: کلاستر اصلی به خوبی در این رنگ‌بندی قابل مشاهده هستند. این روش مجموعاً ۸۹ کلاستر در این گراف پیدا کرد که البته با تغییر پارامترها می‌توان آن را اضافه یا کم کرد.



## infomap •

هیچ یک از سه ابزار networkit، graph-tool و gephi و این الگوریتم را پیاده‌سازی نکرده‌بودند. به این منظور از کتابخانه‌ی cdlib در پایتون استفاده کردیم و نتیجه‌ی آن را به gephi برای رسم انتقال دادیم. کد این بخش در فایل infomap.py موجود است. از آنجایی که در گراف ما، ۸ کلاستر ذکر شده تقریباً مستقلند، این الگوریتم‌ها همگی تا حد خیلی خوبی قادر به جداسازی آن‌ها بوده‌اند. تصویر زیر نتیجه‌ی رنگ‌آمیزی با نتایج Infomap است:



## مقایسه

برای تحلیل هوموفیلی، هیچ داده‌ای اختیار نداشتیم اما به اعتقاد من، نکاتی که در بخش centrality مربوط به k-core گفته شد تا مقدار خیلی زیادی در این بخش هم صدق می‌کند و برای ما یافتن کلاسترها بعهم پیوسته اهمیت زیادی خواهد داشت. اینکه این صفحات، در ژانرهای کاملاً مختلفی فعالیت می‌کنند، تعیین کلاسترها اصلی گراف را برای الگوریتم‌های خوشبندی بسیار ساده‌تر کرده است. اما با توجه به نکاتی که در بخش centrality ذکر شد، روش k-shell decomposition که قادر است کلاسترها بسیار بهم تنیده‌ی گراف را جدا کند، به نظر می‌رسد که بهترین اطلاعات را در اختیار ما قرار می‌دهد. استفاده از این روش‌ها برای یافتن خطاهای فکری قدرتمند و بررسی تاثیرات آن‌ها در طول زمان، می‌تواند بسیار مهم باشد. و از بین دو روش دیگر، louvain کلاسترها جزئی‌تر و کوچک‌تری را نسبت به infomap پیدا کرده و این موضوع می‌تواند به همان دلیل قبلی، حائز اهمیت باشد. چون تعیین کلاسترها کلی، اطلاعات جدیدی در اختیار ما قرار نخواهد داد.

\* پیاده‌سازی روش Girvan-Newman در این گراف، به خاطر تعداد نودها و یال‌ها، میسر نبود.

## 5. پیاده‌سازی روش‌های مبتنی بر label propagation

یکی از الگوریتم‌های مشتق شده از این روش‌ها، که برای clustering بر روی گراف‌ها هم بسیار مناسب است، روش layered label propagation است. در این روش، در ابتدا به هر نود یک برچسب به صورت رندم اختصاص پیدا می‌کند. بعد از آن، در هر دور، نودها -با ترتیبی تصادفی- برچسب خود را آپدیت می‌کنند. به این صورت که هر نود همسایه‌های خودش را بررسی می‌کند و برچسبی که بیشترین تکرار در میان همسایگانش دارد را برمی‌گزیند. این فرآیند ادامه پیدا می‌کند تا زمانی که یا برچسب هیچ نودی تغییر نکند و یا به تعدادی کافی iteration داشته باشیم.

پیچیدگی زمانی این الگوریتم از درجه‌ی خطی نسبت به تعداد یال در گراف است. به طور کلی با تعداد محدودی iteration می‌توان به نتیجه‌ی مطلوب رسید در نتیجه از الگوریتم‌های سریع برای کلاسترینگ به حساب می‌آید.

یکی دیگر از خوبی‌های این الگوریتم این است که در صورتی که تعدادی داده با کلاستر مشخص داشته باشیم (فرض بر اینکه داده‌ی  $\mathcal{I}_{train}$  موجود باشد) با تغییراتی می‌توانیم این الگوریتم را به صورت supervised هم به کار ببریم.

جدول زیر پیچیدگی زمانی هر یک از الگوریتم‌های خوشه‌بندی را ارائه می‌کند:

پیچیدگی زمانی	الگوریتم
$O(N + E)$	K-shell decomposition
$O(E^2 * N)$	Girvan–Newman algorithm
برای گراف‌های sparse برابر $O(N \log N)$ و در حالت کلی $O(E \log E)$	Louvain method
برای گراف‌های sparse برابر $O(N \log N)$ و در حالت کلی $O(E \log E)$	Infomap
$O(E)$	Layered label propagation

با توجه به جدول بالا، این الگوریتم در ردیف بهترین الگوریتم‌ها از نظر پیچیدگی زمانی قرار دارد. هرچند پیاده‌سازی آن با زبان مناسب بسیار در زمان اجرایی تاثیرگذار خواهد بود. برای مثال، پیاده‌سازی gephi از الگوریتم louvain. از پیاده‌سازی ما از الگوریتم LLP سریع‌تر اجرا شد. چرا که اصولاً زبان پایتون که ما برای پیاده‌سازی استفاده کردیم زبان بسیار کندتری خواهد بود.

از نظر نتیجه‌ی الگوریتم هم به نظر می‌رسد که به ترتیب پیاده‌سازی louvain الگوریتم در نرم‌افزار gephi، بعد Infomap با کتابخانه‌ی cdlib و بعد پیاده‌سازی ما از LLP بهترین نتیجه‌ها را بر روی این

گراف خاص داشتند. البته LLP با افزایش تعداد iterationها می‌تواند به دقت بیشتری هم دست پیدا کند.

برای پیاده‌سازی این الگوریتم، ما از کتابخانه‌ی networkkit load تنها برای کردن دیتابست استفاده کردیم و پیاده‌سازی آن تماماً در فایل layered\_label\_propagation.py که ضمیمه شده است قابل بررسی است. همچنین برای تسربی انجام این الگوریتم، با استفاده از کتابخانه‌ی multiprocessing در python آپدیت کردن labelها را به صورت موازی انجام دادیم.

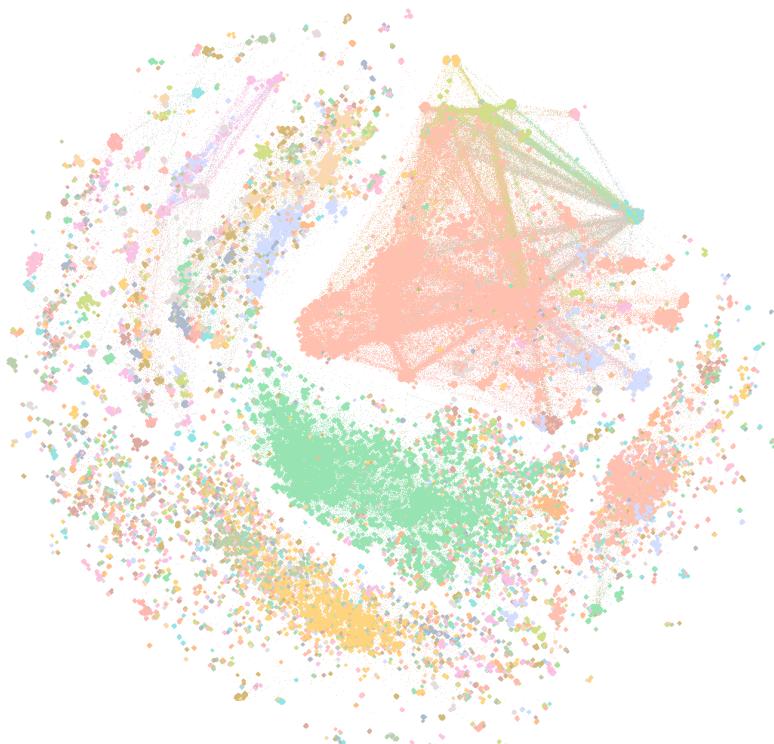
```
def fit(n_clusters, graph, n_threads=4, result_file='llp-labels.csv', max_iter=1000):

    label_dict = _init_labels_(n_clusters, graph)
    node_list = [v for v in graph.iterNodes()]
    pool = mp.Pool(n_threads)

    end_iteration = False
    iteration = 0
    while not end_iteration and (iteration < max_iter):
        np.random.shuffle(node_list)
        new_node_labels = pool.map(_find_new_node_label_, [(label_dict, n) for n in node_list])
        label_dict, end_iteration = _update_labels_(new_node_labels, node_list, label_dict)
        iteration += 1
        if iteration % 10 == 0:
            print('iter: ', iteration)

    save_labels(result_file, label_dict)
```

تصویر زیر با استفاده از اجرای این الگوریتم با تعداد ۳۰۰ iteration در gephi رسم شده است. (خروجی این الگوریتم در فایلی با فرمت csv ذخیره شده و در gephi به عنوان node property load شده است).



## 6. نمایش گراف

در شکل زیر، رنگ نودها با توجه component‌های پیدا شده با روش louvain رسم شده‌اند؛ و سایز نودها بر اساس امتیازی است که از معیار page rank بدست آورده‌اند. اگر دقت کنید در هر کلاستر بزرگ، تعدادی نod که نسبت به دیگران شاخص هستند خواهد دید.

