علوم رایانش و فناوری اطلاعات
نشریه علمی انجمن کامپیوتر ایران
مجلد ۱۶، شماره ۱، ۱۳۹۷
صفحات ۴۷- ۵۸
مقاله عادی

نشریه علوم رایانش و فناوری اطلاعات

الگوریتم موازی برای کاوش زیرگراف‌های منسجم در گراف‌های حجیم

مهدی عالمی   حسن حقیقی

دانشکده مهندسی و علوم کامپیوتر، دانشگاه شهید بهشتی، تهران، ایران

چکیده

یافتن زیرگراف‌های منسجم در بسیاری از کاربردها از جمله موتورهای جستجو، شبکه‌های اجتماعی و شبکه‌های بیولوژی جایگاه ویژه‌ای دارد. در این بین، زیرگراف K-Truss به دلیل سادگی و کاربردهایی مانند یافتن نقاط پرچگال و ترسیم گراف کاربرد اهمیت دارد. با توجه به حجم زیاد گراف‌ها، توسعه‌ی الگوریتم‌های موازی برای افزایش سرعت در پاسخ‌دهی نیازی اساسی است. برای این منظور، در این مقاله یک الگوریتم موازی برای کاوش زیرگراف‌های منسجم در گراف‌های حجیم در ۲ فاز ارائه شده است؛ در فاز اول، ابتدا با یک الگوریتم موازی مثلث‌های گراف یافت می‌شوند و خروجی‌ها در یک ساختار جدید با نام رأس‌های مثلثی برای نگهداری مجموعه رأس‌هایی که با هر یال تشکیل یک مثلث می‌دهند، تولید می‌شود. در فاز دوم، در یک حلقه به صورت موازی یال‌های نامعتبر به تدریج از مجموعه یال‌های موجود حذف می‌گردند تا زمانی که هیچ یالی که خصوصیت K-Truss را نقض می‌کند، پیدا نشود. کارایی روش پیشنهادی در مقایسه با دیگر روش‌ها و مقیاس‌پذیری آن، با انجام آزمایش‌هایی بر روی یک ماشین ۱۲ هسته‌ای با استفاده از مجموعه گراف‌های استاندارد نشان داده شده است.

کلمات کلیدی: اجتماعات منسجم، شبکه‌های اجتماعی، پردازش موازی، شمارش مثلث، k-Truss.

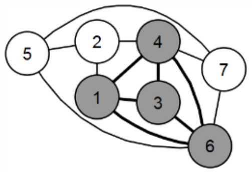
۱- مقدمه

یافتن زیرگراف‌های منسجم[1] در گراف‌های حجیم با میلیون‌ها یال تبدیل به موضوعی پرکاربرد در زمینه‌های مختلف از جمله شبکه‌های اجتماعی [۱]، گراف وب [۲]، شبکه‌های بیولوژی [۳] و غیره گردیده است. منسجم‌ترین زیرگراف کلیک[2] است که در آن همه‌ی رئوس با هم در ارتباط هستند. تعریف کلیک خیلی سخت‌گیرانه است و تعداد زیرگراف‌های یافت شده که دارای این خصوصیت باشند، کم است. برای این منظور مشتقات دیگری از کلیک که دارای سخت‌گیری کمتری هستند، ارائه شده است. در N-Clique [۴] فاصله‌ی بین هر دو رأس از ۱ به N تغییر یافته است. زیرگراف N-Clan [۵] یک N-Clique است که در آن قطر زیرگراف (حداکثر فاصله بین هر دو رأس در زیرگراف) باید کمتر از N باشد؛ در حالی که، N-Club[۵] زیرگرافی است که تنها قطر زیرگراف را محدود به N می‌کند. در یک زیرگراف K-Plex[۶] با n رأس، هر رأس با حداقل n − K رأس دیگر همسایه است. متاسفانه، یافتن این زیرگراف‌ها در دسته مسائل NP-Hard قرار می‌گیرند و یافتن آن‌ها زمان‌بر می‌شود؛ یا همه‌ی آن‌ها در زمان محدود یافت نمی‌شوند.

علاوه بر زیرگراف‌های ذکر شده، زیرگراف‌های دیگری از جمله K-Core[۷] و K-Truss که دارای پیچیدگی زمانی چندجمله‌ای هستند، نیز ارائه شده است. در زیرگراف K-Core درجه‌ی همه رأس‌ها باید حداقل K باشد. از آنجا که زیرگراف‌های K-Core انسجام کافی ندارند، با توجه به سادگی و پیچیدگی زمانی پایین K-Core، بیشتر از آن به عنوان یک مرحله پیش‌پردازش در الگوریتم‌های دیگر استفاده می‌شود [۸، ۹] . بر خلاف آن، زیرگراف K-Truss محبوبیت زیادی پیدا کرده است. زیرا، علاوه بر منسجم بودن این زیرگراف‌ها، کاربردهایی نیز از آن گزارش شده است. به عنوان نمونه، با توجه به سادگی و پیچیدگی زمانی پایین، این زیرگراف به عنوان یک مرحله پیش‌پردازش در مسائل دیگر مثل یافتن کلیک [۸] به کار رفته است. از کاربردهای دیگر آن می‌توان به یافتن نقاط پرچگال در گراف اشاره کرد [۱۰] که در ترسیم و نمایش گراف‌های بزرگ [۹] و تشخیص تقلب [۱۱] نقش مهمی دارد. همچنین، در تشخیص انتشار دانش در گراف دانش

[۱۲، ۱۳، ۱۴]، تحلیل اتصال[۳] در شبکه [۱۵] و جستجوی اجتماعات منسجم [۱۶، ۱۷] نیز راه‌حل‌هایی مبتنی بر K-Truss ارائه شده است.

اولین تعریف از K-Truss توسط کوهن[۴] در [۸] ارائه شده است. طبق این تعریف، برای یک گراف $G = (V, E)$، زیرگراف K-Truss، بزرگ‌ترین زیرگرافی است که در آن هر یال $e \in E$ در حداقل $(K-2)$ مثلث (یک کلیک با سه رأس) درون زیرگراف شرکت داشته باشد. به عنوان مثال، اگر در شبکه‌های اجتماعی، هر فرد با یک گره و ارتباط دوستی بین دو فرد با یک یال مدل شود، آنگاه زیرگراف K-Truss مشخص کننده اجتماعی است که در آن هر دو دوست دارای $K-2$ دوست مشترک هستند. در گراف شکل ۱ رأس‌های توپر و یال‌های پررنگ مشخص‌کننده زیرگراف 4-Truss هستند که در آن رأس‌های متصل به هر یال حداقل دارای ۲ همسایه مشترک هستند.



شکل ۱- رأس‌ها و یال‌های پررنگ نشان‌دهنده زیرگراف 4-truss است

در [۸] اولین روش برای یافتن K-Truss با ارائه یک الگوریتم تکرارشونده[۵] ترتیبی ارائه شده است. در هر تکرار از این روش، ابتدا مثلث‌های گراف شمرده می‌شوند و سپس یال‌هایی که خصوصیات K-Truss را ارضاء نکنند، از گراف حذف می‌گردند. گراف به‌روز شده به عنوان ورودی برای تکرار بعدی الگوریتم به کار می‌رود. راه‌حل‌های مختلفی سعی در بهبود کاوش زیرگراف K-Truss داشته‌اند تا کارایی را افزایش و حجم وسیع‌تری از گراف‌ها را نیز پوشش دهند. رویکردهای موجود شامل راه‌حل‌های توزیع‌شده، چندهسته‌ای و ورودی/خروجی-کارا[۶] است که در بخش ۳ مرور شده‌اند. با وجود کارهای انجام شده، هنوز کارایی این روش‌ها به اندازه کافی بهبود نیافته است. بنابراین، در این مقاله یک الگوریتم موازی با نام PKT ارائه شده است که می‌تواند با استفاده از ظرفیت تمامی هسته‌های پردازشی ماشین با سرعت بیشتری نسبت به دیگر روش‌ها زیرگراف K-Truss را استخراج نماید.

الگوریتم PKT در دو فاز متوالی زیرگراف K-Truss را استخراج می‌کند. در فاز اول به صورت موازی تمام مثلث‌ها کاوش می‌شوند. برای این منظور، رویکرد توزیع‌شده‌ای که توسط نویسندگان [۱۸] برای شمارش مثلث‌ها ارائه شده است، به صورت چندهسته‌ای و کاراتر پیاده‌سازی شده است. برای این منظور، ساختار داده FONL که در [۱۸] پیشنهاد شده است، از روی لیست همسایگی توسط نخ‌های پردازشی[۷] به صورت موازی ساخته می‌شود و در حافظه‌ی مشترک نگهداری می‌شود. در FONL همسایه‌های هر رأس که درجه‌ی آن‌ها بزرگتر از آن رأس است، انتخاب می‌شوند و بر حسب درجه به صورت صعودی مرتب می‌شوند. این ساختار، یافتن مثلث‌ها در مراحل بعدی را کاراتر می‌کند و علاوه بر آن، موجب ریز-دانگی[۸] محاسبات و به تبع آن افزایش موازی‌سازی می‌گردد. از این طریق، توازن بار بین هسته‌ی پردازشی افزایش می‌یابد که سبب افزایش مقیاس‌پذیری می‌گردد. در حین یافتن مثلث‌ها، به ازای هر یال، پشتیبان آن (که مشخص‌کننده تعداد مثلث‌هایی است که آن یال در آن‌ها شرکت دارد) به صورت سراسری به‌روزرسانی می‌شود. همچنین، در هر نخ پردازشی، یک ساختار داده با نام رأس‌های مثلثی تعریف شده است که به‌منظور نگهداری رأس‌هایی است که با یک یال تشکیل یک مثلث می‌دهند. هر نخ پردازشی، پس از یافتن یک مثلث، رأس‌های مثلثی مربوط به یال‌های آن مثلث را به صورت محلی بروزرسانی می‌کند.

فاز دوم از الگوریتم، با دریافت پشتیبان یال‌ها به صورت سراسری و رأس‌های مثلثی به صورت محلی در هر نخ پردازشی کار خود را آغاز می‌کند. این الگوریتم در یک حلقه به صورت تکرارشونده بخشی از یال‌ها که خصوصیت K-Truss را ارضاء نکنند، حذف می‌کند. برای این منظور، در هر تکرار، نخ‌های پردازشی به صورت موازی اقدام به بروزرسانی دو ساختار پشتیبان یال‌ها و رأس‌های مثلثی می‌کنند و این کار تا زمانی ادامه پیدا می‌کند که هیچ یالی پیدا نشود که خصوصیت K-Truss را نقض کند. یال‌های باقی‌مانده به عنوان یک زیرگراف K-Truss از گراف ورودی بازگردانده می‌شوند.

برای بررسی کارایی PKT، آزمایش‌های مختلفی با استفاده از گراف‌های واقعی استاندارد بر روی یک ماشین ۱۲ هسته‌ای انجام شده است. با توجه به نتایج بدست آمده، راه حل ارائه شده در این مقاله توانسته است به خوبی از ظرفیت هسته‌های پردازشی استفاده کند و زیرگراف K-Truss را با سرعت بالاتری نسبت به دیگر روش‌ها استخراج کند؛ به‌طوری‌که، در برخی گراف‌ها سرعت اجرا بیش از ۱۰ برابر افزایش یافته است. همچنین، مقیاس‌پذیری روش پیشنهادی با افزایش تعداد پردازنده‌ها نشان داده شده است.

در ادامه، مفاهیم پایه‌ای موردنیاز در بخش ۲ تعریف شده‌اند. کارهای مرتبط در بخش ۳ مرور گردیده‌اند. بخش ۴، الگوریتم پیشنهادی را، که دارای فازهای «یافتن پشتیبان و رأس‌های مثلثی» و «استخراج K-Truss» است، در دو زیربخش جداگانه تشریح می‌کند. آزمایش‌ها و نتایج بدست آمده از اجرای روش پیشنهادی در بخش ۵ نشان داده شده است. در نهایت، نتیجه‌گیری و کارهای آتی در بخش ۶ بیان شده است.

۲-تعاریف و مفاهیم پایه‌ای

فرض کنیم یک گراف $G = (V.E)$ داریم که در آن V مجموعه‌ی رأس‌ها و E مجموعه‌ی یال‌ها باشد. تعداد رأس‌های گراف را با n و تعداد یال‌های آن را با m نشان می‌دهیم. برای هر رأس $v \in V$، همسایه‌های آن با $N(v) = \{u \in V: (u.v) \in E\}$ مشخص می‌شوند. همچنین درجه‌ی رأس v با $d(v) = |N(v)|$ نشان داده می‌شود. در ادامه تعریف‌های موردنیاز در طول این مقاله ارائه شده‌اند.

تعریف ۱) ترتیب سراسری (Total Order): به ازای دو رأس u و $v(u \neq v)$، رابطه $u \prec v$ نشان‌دهنده کمتر بودن ترتیب سراسری u نسبت به v است. در این حالت، شرط مشخص شده در فرمول ۱ باید برقرار باشد. این شرط بیان می‌کند که درجه رأس u باید کمتر از درجه رأس v باشد و در شرایطی که درجه این دو رأس با هم برابر باشد، شناسه‌ی رأس u باید کمتر از شناسه رأس v باشد.

$$u \prec v \Leftrightarrow d(u) < d(v) \text{OR} \\ id(u) < id(v) \text{if} d(u) = d(v) \quad (۱)$$

تعریف ۲) مثلث: به ازای سه رأس $u \neq v \neq w$، مطابق با فرمول ۲ یک مثلث $\Delta_{u.v.w}$ تشکیل می‌شود اگر و فقط اگر رأس‌های u و v و w با هم همسایه باشند.

$$\Delta_{u.v.w} \Leftrightarrow \{(u.v).(v.w).(u.w)\} \subseteq E \quad (۲)$$

تعریف ۳) مجموعه رأس‌های مثلثی[۹]: به ازای هر یال $e = (u.v)$، رأس‌های مثلثی آن با $TV(e)$ نشان داده می‌شود که شامل مجموعه رأس‌هایی است که با u و v همسایه هستند. در این حالت، به ازای یک مثلث $\Delta_{u.v.w}$ داریم: $w \in$

$TV(u.v)$، $v \in TV(u.w)$ و $u \in TV(v.w)$. در فرمول ۳ این تعریف به صورت صوری ارائه شده است.

$$TV(e) = \{w|\Delta_{u.v.w}\} \quad (۳)$$

تعریف ۴) پشتیبان[۱۰] یال: برای هر یال $e = (u.v)$، پشتیبان آن با $Sup(e)$ نشان داده می‌شود که مشخص‌کننده تعداد مثلث‌هایی است که یال $(u.v)$ در آن شرکت دارد. طبق این تعریف در فرمول ۴ داریم:

$$Sup(e) = |\{\Delta_{u.v.w}| \{(u.w) . (v.w)\} \subset E\}| \quad (۴)$$

تعریف ۵) زیرگراف K-Truss: گراف $G_{KT} = (V_{KT}. E_{KT})$ ($V_{KT} \subseteq V$ و $E_{KT} \subseteq E$)، یک زیرگراف K-Truss از G است، اگر مطابق با فرمول ۵ تمامی یال‌های موجود در این زیرگراف دارای پشتیبان حداقل برابر $K - 2$ باشند.

$$G_K = \{e|e \in E and Sup(e) \geq K - 2\} \quad (۵)$$

تعریف ۶) یال نامعتبر[۱۱] در K-Truss: هر یال e از گراف G که جزء زیرگراف K-Truss نباشد، یال نامعتبر در آن زیرگراف شناخته می‌شود. در مقابل، یال‌های موجود در زیرگراف K-Truss، یال‌های معتبر شناخته می‌شوند. اگر مجموعه یال‌های نامعتبر را با invalids نشان دهیم، شرط ارائه شده در فرمول ۶ این تعریف را به صورت صوری بیان می‌کند.

$$e \in invalids \Leftrightarrow e \in G and e \notin G_K \quad (۶)$$

۳- کارهای مرتبط

برای استخراج زیرگراف K-Truss نیاز است که الگوریتم به صورت تکرارشونده عمل کند و در هر تکرار بخشی از یال‌های نامعتبر حذف شود. گراف بروز شده در هر تکرار به عنوان ورودی در تکرار بعدی مورد استفاده قرار می‌گیرد. علت تکراری بودن الگوریتم این است که تا برخی از یال‌ها حذف نگردند، ممکن است نامعتبر بودن یال‌های دیگر مشخص نشود.

در [۸] که اولین تعریف از زیرگراف K-Truss در آن ارائه شده است، ابتدا یک الگوریتم ساده با پیچیدگی زمانی $O(nm^2)$ معرفی گردیده است که در آن در هر تکرار، الگوریتم به ازای یک یال بررسی می‌کند که اشتراک همسایه‌های دو رأس آن کمتر از $K-2$ باشد. از این طریق یال‌های نامعتبر شناخته می‌شوند و از گراف حذف می‌گردند. در همین مقاله، یک روش بهبود یافته در زمان $O(\sum_{v \in V} d^2(v))$، با شمارش مثلث‌ها و تشخیص پشتیبان به ازای هر یال در هر تکرار الگوریتم، یال‌های نامعتبر را شناسایی و حذف می‌کند. علاوه بر آن، هرس کردن رأس‌ها با استفاده از الگوریتم K-Core، به عنوان یک مرحله پیش‌پردازش در هر تکرار، مدنظر قرار گرفته است؛ زیرا، طبق تعریف، هر زیرگراف K-Truss یک زیرگراف $(K-1)$-Core نیز محسوب می‌شود، درحالی‌که عکس این قضیه صحیح نیست [۸].

با توجه به زمان‌بر بودن شمارش مثلث‌ها، اجرای آن در هر تکرار برای محاسبه پشتیبان یال‌ها، سبب کند شدن اجرا می‌شود. برای جلوگیری از شمارش مثلث‌ها در هر تکرار، راه‌حل‌هایی در [۱۹، ۲۰، ۲۱، ۲۲، ۲۳] ارائه شده است که در آن‌ها ابتدا پشتیبان یال‌ها محاسبه می‌شود و در ادامه الگوریتم، مقادیر پشتیبان در تکرارهای بعدی بروزرسانی می‌شود. برای بروزرسانی پشتیبان یال‌ها، این راه‌حل‌ها نیاز دارند که لیست همسایگی گراف را در حافظه نگهداری کنند تا از طریق اشتراک‌گیری بین همسایه‌های دو رأس یک یال نامعتبر، بتوانند دیگر یال‌هایی را، که پشتیبان آن‌ها باید بروزرسانی شود، پیدا کنند. بنابراین، پس از حذف یک یال $(u.v)$، دو یال دیگر مثلث مربوطه با پیچیدگی زمانی $O(d(v) + d(u))$ جستجو می‌گردد که در این حالت زمان بروزرسانی پشتیبان یال‌ها به شدت افزایش می‌یابد. در ادامه، هر یک از این روش‌ها با جزئیات بیشتری تشریح می‌گردند.

در [۱۹] یک الگوریتم ترتیبی برای پردازش گراف‌های حجیم با تمرکز بر افرازبندی و بارگذاری گراف از/به دیسک ارائه شده است. در این مقاله، یک روش ترتیبی با پیچیدگی زمانی $O(m^{1.5})$ ارائه شده است. در این روش، پس از محاسبه پشتیبان، یال‌ها بر حسب پشتیبان‌شان مرتب می‌شوند و در هر تکرار، یال‌های نامعتبر شناسایی و حذف می‌شوند. در این الگوریتم حذف هر یال نامعتبر منجر به کاهش پشتیبان دو یال دیگر، که با یال نامعتبر تشکیل یک مثلث می‌دهند، می‌شود. برای یافتن این دو یال، همسایه‌های دو رأس یال نامعتبر مورد بررسی قرار می‌گیرند و با یافتن همسایه‌های مشترک، دو یال دیگر تشخیص داده می‌شود. همان‌طور که ذکر شد، تشخیص همسایه‌های مشترک در این مرحله دارای سربار زمانی است و در نتیجه، انجام آن در هر تکرار از الگوریتم به ازای هر یال نامعتبر، سبب کندی اجرا می‌شود. در همین مقاله، برای پردازش گراف‌های حجیم نیز یک روش مبتنی بر دیسک ارائه شده است که در آن ابتدا رأس‌های گراف افرازبندی می‌شوند، که هر افراز حاوی بخشی از رأس‌های کل گراف است. در هر تکرار تمامی افرازها به ترتیب پردازش می‌شوند. افرازها به ترتیب خوانده می‌شوند و به ازای آن‌ها زیرگراف همسایگی ساخته می‌شود. زیرگراف همسایگی حاوی رأس‌های افراز جاری و کلیه همسایه‌های آن‌ها است. در اینجا، ساختن گراف همسایگی هزینه زیادی دارد، زیرا برای ساختن آن باید کل گراف از دیسک خوانده شود. همچنین، افرازهایی که حاوی رأس‌های با همسایه‌های زیاد هستند، سبب می‌شوند که گراف همسایگی خیلی بزرگ شود و کمبود حافظه رخ دهد.

در [۲۰] محاسبه K-Truss برای گراف‌های غیرقطعی تعریف شده است. در این راستا، ابتدا یک الگوریتم درون-حافظه‌ای[۱۲] با پیچیدگی زمانی $O(m^{1.5}Q)$ ارائه شده است. سپس، برای پردازش گراف‌های بزرگ که در حافظه جای نگیرند، یک الگوریتم برون-حافظه‌ای[۱۳] با رویکرد مشابه [۱۹] نیز پیشنهاد شده است. در این روش نیز مشکلات مطرح شده در [۱۹] وجود دارد.

در [۲۱] یک روش چندهسته‌ای با رویکرد تقسیم کار بر حسب یال بین نخ‌های پردازشی (پردازش مجموعه‌ای از یال‌ها توسط هر نخ) ارائه شده است. در این روش، پس از اتمام کار شمارش مثلث‌ها، به صورت ترتیبی، یال‌ها با توجه به پشتیبان‌شان مرتب‌سازی و تعداد مثلث‌های هر یال در یک ساختار فشرده نگهداری می‌شود. سپس به صورت موازی یال‌های با پشتیبان کم حذف می‌شوند و اطلاعات یال‌ها به‌روزرسانی می‌گردد.

در [۲۴] روش کوهن بر مبنای انتزاع نگاشت-کاهش[۱۴] [۲۵] ارائه شده و با استفاده از هدوپ[۱۵] [۲۶] پیاده‌سازی شده است. در این رویکرد، داده‌ها به صورت کلید - مقدار[۱۶] مدل شده‌اند که در آن، کلید مشخص کننده یک یال منحصر به‌فرد و مقدار بیانگر دو یال دیگر است که با کلید تشکیل یک مثلث می‌دهند. در اینجا به دلیل نیاز به فضای ذخیره‌سازی بالا در نگهداری کلید-مقدارها و نگهداری داده‌های میانی مربوط به هر تکرار در دیسک، سرعت اجرا به شدت کند می‌شود. علاوه بر آن، این روش دارای همان مشکل کوهن در شمارش مثلث‌ها در هر تکرار می‌باشد که البته این مشکل در [۲۲] برطرف گردیده است.

در [۲۷] یک روش بر مبنای انتزاع پریگل[۱۷] [۲۸] پیشنهاد شده است. در انتزاع پریگل، هر رأس تنها قادر به ارتباط با همسایه‌های خود است و می‌توان مقادیری را در رأس‌ها و یال‌ها نگهداری کرد. این روش به صورت تکرارشونده عمل می‌کند و در هر تکرار، ۴ گام را به ترتیب انجام می‌دهد. گام ۱) هر رأس شناسه‌ی رأس‌های همسایه‌ی خود را به تمام همسایگانش می‌فرستد. گام ۲) رأس‌های

دریافت کننده پیام، مثلث‌ها را تشخیص می‌دهند و در پاسخ، مشخصات مثلث‌ها را به رأس فرستنده پیام ارسال می‌کنند. گام ۳) هر یالی که پشتیبان کمی داشته باشد، حذف می‌گردد و پیام مربوط به این حذف به دو رأس آن یال ارسال می‌شود. گام ۴) اگر هیچ یالی پیامی دریافت نکرد، تکرار جاری به پایان می‌رسد. در این روش، توازن بار به درستی صورت نمی‌گیرد و پیام‌های زیادی بین رأس‌ها رد و بدل می‌شود. علاوه بر آن، مشکل شمارش مثلث‌ها در هر تکرار همچنان وجود دارد.

در نهایت در [۲۳] یک روش توزیع‌شده با معرفی انتزاع زیرگراف ارائه شده است (در این انتزاع، موجودیتی به نام زیرگراف وجود دارد). این روش کار خود را با تولید زیرگراف‌ها از گراف ورودی شروع می‌کند، به‌طوری‌که کل گراف پوشش داده شود، گرچه زیرگراف‌ها ممکن است با یکدیگر همپوشانی داشته باشند. سپس یال‌های نامعتبر در هر زیرگراف به صورت تکرارشونده حذف می‌شوند. هر تکرار دارای دو مرحله است: ابتدا به ازای هر زیرگراف، یال‌های با پشتیبان کم، که تمام مثلث‌های آن در همان زیرگراف قرار دارند، حذف می‌شوند. سپس، در خصوص حذف بقیه یال‌های گزارش شده (که امکان تشکیل مثلث با یال‌های موجود در سایر زیرگراف‌ها را دارند) از هر زیرگراف، به صورت سراسری تصمیم‌گیری می‌شود. در این روش ایجاد زیرگراف‌ها نیاز به زمان زیادی دارد. همچنین ممکن است از یک تکرار به بعد، توازن بار در حین پردازش محلی زیرگراف‌ها از بین برود.

به‌طور کلی، در روش‌های ارائه شده که در آن‌ها، شمارش مثلث‌ها در هر تکرار الگوریتم انجام می‌شود، زمان زیادی از الگوریتم صرف این عملیات می‌گردد. برای روش‌هایی که یک‌بار شمارش مثلث را انجام می‌دهند، بروزرسانی پشتیبان یال‌ها در حین تکرار الگوریتم زمان‌بر است. در این مقاله، مرحله شمارش مثلث تنها یک‌بار انجام می‌گیرد. همچنین، ساختارهای مناسبی ارائه شده است که سبب می‌شوند، بروزرسانی پشتیبان یال‌ها با سرعت بالایی انجام شود. ساختارهای پیشنهادی علاوه بر داشتن قابلیت موازی‌سازی، دارای حداقل عناصر اطلاعاتی هستند تا کمترین میزان مصرف حافظه را داشته باشند.

۴- الگوریتم موازی برای یافتن K-Truss

در این بخش، روش پیشنهادی برای یافتن زیرگراف K-Truss ارائه می‌گردد. برای این منظور، ابتدا در بخش ۴-۱ یک الگوریتم موازی برای یافتن پشتیبان و مجموعه رأس‌های مثلثی (بخش ۲- تعریف ۳) ارائه می‌شود. سپس، در بخش ۴-۲ زیرگراف K-Truss با استفاده از یک روش موازی استخراج می‌گردد. به‌منظور درک بهتر الگوریتم‌های پیشنهادی، گراف شکل ۱ به عنوان یک مثال اجرایی درنظر گرفته شده است و خروجی‌های مرتبط با هر مرحله با توجه به این گراف نمایش داده می‌شود.

۴-۱- یافتن پشتیبان و رأس‌های مثلثی

فرض می‌کنیم گراف ورودی به صورت لیست همسایگی ذخیره شده است. در ابتدا از روی لیست همسایگی، یک ساختار داده به نام FONL در قالب کلید-مقدار ساخته می‌شود. در این ساختار، به ازای هر رأس، یک کلید-مقدار ایجاد می‌شود، که کلید آن برابر با شناسه رأس و مقدار آن شامل همسایه‌هایی است که ترتیب سراسری آن‌ها (بخش ۲- تعریف ۱) بزرگ‌تر از رأس جاری باشد. همچنین، این همسایه‌ها به صورت صعودی بر حسب ترتیب سراسری مرتب می‌شوند. الگوریتم ۱ نحوه‌ی ساختن این ساختار داده را نشان می‌دهد. در این الگوریتم، ورودی NL مشخص کننده لیست همسایگی است که به صورت یک آرایه‌ی دو بعدی می‌باشد. اندیس بُعد اول مشخص‌کننده شناسه رأس و همسایه‌های هر رأس مقابل آن در بُعد دوم ذخیره شده‌اند.

خط اول الگوریتم، آرایه‌ی دو بعدی FONL را ایجاد می‌کند که اندازه‌ی بعد اول آن برابر با n است. سپس در خط دوم به ازای هر رأس به طور موازی دستورات خطوط ۳ تا ۵ اجرا می‌شود. در خطوط ۳ و ۴، همسایه‌هایی که ترتیب سراسری آن‌ها بزرگ‌تر از u است، به $FONL_u$ اضافه می‌شوند. سپس در خط ۵ همسایه‌های انتخاب شده مرتب می‌شوند. پس از اتمام کار همه‌ی نخ‌ها، نتیجه در خط ۶ بازگردانده می‌شود. با توجه به توزیع یکنواخت رأس‌ها بین P نخ پردازشی، هر نخ تعداد $\frac{n}{P}$ رأس را پردازش می‌کند. خطوط ۳ و ۴ به اندازه درجه هر رأس نیاز به عملیات دارد و در خط ۵ مرتب‌سازی در زمان حداکثر $\sqrt{m}$ انجام می‌گیرد، زیرا در بدترین حالت اگر گراف ورودی یک گراف کامل باشد، بین تمامی رئوس گراف یال وجود دارد؛ یعنی، $m = O(n^2)$ است. در این حالت رأس u در $FONL_u$ می‌تواند به طور متوسط تعداد $\sqrt{m}$ رأس بزرگ‌تر از خود داشته باشد. بنابراین، پیچیدگی زمانی الگوریتم ۱ در بدترین حالت برابر با $O(\frac{n}{P} \times \sqrt{m})$ = $O(\frac{m}{P})$ است.

الگوریتم ۱- ایجاد ساختار داده FONL به صورت موازی

ورودی: گراف G به صورت لیست همسایگی NL

خروجی: FONL

```
1 Set FONL to be a two-dimensional array of size |V|
2 for each u ∈ V do in parallel
3    for each v ∈ NLu do
4        if u ≺ v then add v to FONLu
5    sort vertices in FONLu based on Total Order
6 return FONL
```

رأس u	درجه	همسایه ها
۱	۴	۲, ۳, ۴, ۶
۲	۳	۱, ۴, ۵
۳	۳	۱, ۴, ۶
۴	۵	۱, ۲, ۳, ۶, ۷
۵	۳	۲, ۶, ۷
۶	۵	۱, ۳, ۴, ۵, ۷
۷	۳	۴, ۵, ۶

P1 → (رأس‌های ۱ و ۲)، P2 → (رأس‌های ۳ و ۴)، P3 → (رأس‌های ۵، ۶ و ۷)

رأس u	$FONL_u$
۱	۴, ۶
۲	۵, ۱, ۴
۳	۱, ۴, ۶
۴	۶
۵	۶, ۷
۶	۴
۷	۴, ۶

شکل ۲- ایجاد ساختار داده FONL از لیست همسایگی به صورت موازی در سه نخ پردازشی P1، P2 و P3

الگوریتم ۲- محاسبه پشتیبان و رأس‌های مثلثی یال‌ها

ورودی: FONL

خروجی: پشتیبان یال‌ها (Sup)، آرایه‌ای از نگاشت یال به مجموعه رأس‌های مثلثی (TVs)

```
Set P to be the number of threads
Set Sup to be a map from an edge e to its support as an atomic integer
Set TVs to be an array of maps with size P
for each u ∈ V do in parallel
   for each v ∈ FONLu do
      let vi be the index of v in FONLu
      let FLu be the length of FONLu
      let SFuv be a sub-array of FONLu as FONL [vi + 1 … FLu]
      c = intersection between SFuv and FONLv
      for each w in c do
         increment sup (u, v)
         increment sup (u, w)
         increment sup (v, w)
         let t be Id of the current thread
         add w to   TVs [t] (u, v)
         add u to   TVs [t] (v, w)
         add v to   TVs [t] (u, w)
return Sup and TVs
```

در شکل ۲ ساختار داده FONL حاصل از لیست همسایگی گراف شکل ۱ نشان داده شده است. همان‌طور که ملاحظه می‌شود، هر نخ به صورت مستقل بخشی از لیست همسایگی را دریافت می‌کند و در اندیس معادل لیست همسایگی، خروجی خود را که بخشی از ساختار FONL است، می‌نویسد. همان‌طور که در

شکل نشان داده شده است، مجموعه رئوس {۱, ۲}، {۳, ۴} و {۵, ۶, ۷} به ترتیب توسط سه نخ پردازشی P1، P2 و P3 مورد پردازش قرار می‌گیرند.

پس از ایجاد FONL، الگوریتم ۲ یافتن مثلث‌ها و محاسبه پشتیبان و بروزرسانی رأس‌های مثلثی به ازای هر یال را انجام می‌دهد. با توجه به خطوط ۱ تا ۳، متغیر Sup یک نگاشت از یال e به پشتیبان آن را، که یک متغیر صحیح تفکیک‌ناپذیر[18] است، فراهم می‌کند. متغیرهای اتمیک ساختار داده‌ای را فراهم می‌کنند که مقادیر آن‌ها توسط نخ‌های متفاوت قابل تغییر است؛ به‌طوری‌که در هر لحظه از زمان تنها یک نخ پردازشی می‌تواند مقدار آن را تغییر دهد. بنابراین، با توجه به اینکه بروزرسانی Sup توسط همه‌ی نخ‌های پردازشی صورت می‌گیرد، نگهداری و تغییر پشتیبان یال‌ها باید به صورت تفکیک‌ناپذیر انجام می‌شود. بر روی متغیر تفکیک‌ناپذیر ذکر شده، دو عملگر افزایش (increment) و کاهش (decrement) مقدار عدد ذخیره شده را افزایش یا کاهش می‌دهند. علاوه بر پشتیبان که به صورت سراسری قابل دسترس است، هر نخ به صورت محلی، رأس‌های مثلثی یال‌های مربوط به خود را نگهداری می‌کند. برای این منظور، آرایه TVs تعریف شده است و هر اندیس از این آرایه، تنها توسط یک نخ مورد استفاده قرار می‌گیرد.

در ادامه، هر نخ بخشی از رئوس پردازش نشده در FONL را انتخاب می‌کند (خط ۴) و به ازای هر رأس u، دستورات خطوط ۵ تا ۱۷ را اجرا می‌نماید. بنابراین هر نخ به طور متوسط تعداد $\frac{n}{p}$ رأس را پردازش می‌کند. برای هر رأس v در $FONL_u$، ابتدا SF_{uv} در خط ۸ تعیین می‌شود که شامل رأس‌های با اندیس بزرگتر از v در $FONL_u$ است. برای یافتن مثلث‌ها، بین SF_{uv} و $FONL_v$ اشتراک گرفته می‌شود و نتیجه‌ی این اشتراک (متغیر c در خط ۹) مشخص کننده تمام رئوس w است که با u و v تشکیل یک مثلث می‌دهند. با توجه به مرتب بودن رئوس در FONL، این اشتراک در زمان $O(|FONL_u| + |FONL_v|)$ انجام می‌گردد که با توجه به متوسط اندازه هر آیتم FONL در بدترین حالت که برابر $\sqrt{m}$ است، پیچیدگی زمانی انجام اشتراک برابر با $O(\sqrt{m} + \sqrt{m})$ می‌باشد. از آنجا که به ازای هر رأس در FONL، تعداد $\sqrt{m}$ رأس مقابل آن باید مورد بررسی قرار گیرد، تعداد تکرارهای حلقه for در خط ۱۰ برابر با $O(\sqrt{m})$ است. بنابراین پیچیدگی زمانی حلقه در خط ۵ برابر با $O(\sqrt{m} \times \sqrt{m}) = O(m)$ می‌باشد.

در خطوط ۱۱ تا ۱۳، به ازای هر رأس w در c (نتیجه اشتراک)، پشتیبان یال‌های $(u.v)$، $(u.w)$ و $(v.w)$ یک واحد افزایش داده می‌شود، زیرا مجموعه رئوس $\{u.v.w\}$ تشکیل یک مثلث می‌دهند. همچنین هر نخ t به صورت محلی، مجموعه رأس‌های مثلثی مربوط به این یال‌ها را در TVs[t] مربوط به خود (خانه t از آرایه TVs) بروزرسانی می‌کند (خطوط ۱۵ تا ۱۷). پیچیدگی زمانی حلقه for در خط ۱۰ وابسته به اندازه اشتراک یعنی اندازه مجموعه c (تعداد مثلث‌های یافت شده) است. بنابراین، اگر در بدترین حالت اندازه c را برابر با $\sqrt{m}$ فرض کنیم، آنگاه، پیچیدگی زمانی الگوریتم ۲ برابر با $O\left(\frac{n}{p} \times m + m\right) = O(\frac{m\sqrt{m}}{p})$ است.

این الگوریتم پس از اتمام پردازش تمام رأس‌های FONL توسط تمام نخ‌ها، خروجی‌های TVs و Sup را در خط ۱۸ بازمی‌گرداند. TVs بیانگر یک آرایه است که در هر اندیس آن، نتایج مربوط به یک نخ (که به رئوس مثلثی مرتبط است) در حین اجرا ذخیره می‌شود. همچنین، آرایه سراسری Sup که حاوی مقادیر تفکیک‌ناپذیر (تعداد مثلث‌های هر یال) است توسط همه‌ی نخ‌ها بروزرسانی می‌شود.

در شکل ۳ نحوه عملکرد الگوریتم ۲ بر روی FONL نشان داده شده در شکل ۲ و نیز خروجی‌های مربوطه نمایش داده شده است. در شکل ۳-الف، هر نخ بخشی از آیتم‌های FONL را دریافت می‌کند و پس از یافتن SF_{uv}، مثلث‌های موجود را پیدا می‌کند. به عنوان مثال، به ازای رأس ۲، مجموعه رأس‌های v به ترتیب برابر با ۵، ۱ و ۴ است. برای رأس ۵، آرایه SF_{uv} برابر با [۱، ۴] است و برای تشخیص مثلث، باید کنترل شود که آیا هر یک از رئوس ۱ و ۴ با رأس ۵ همسایه هستند یا خیر. برای این منظور، آرایه [۱، ۴] با $FONL_5$ که حاوی [۶، ۷] است، اشتراک گرفته می‌شود؛ نتیجه این اشتراک تهی است. در ادامه و به ازای رأس ۱، آرایه SF_{uv} برابر با [۴] است و نتیجه اشتراک آن با $FONL_1$، که حاوی [۴، ۶] است، برابر با {۴} می‌شود. بنابراین، رأس ۴ به عنوان یک همسایه مشترک بین دو رأس ۱ و ۲ محسوب شده و مثلث {۴، ۱، ۲} یافت می‌شود. پس از یافتن هر مثلث، رئوس مثلثی (TVs) به ازای هر نخ به صورت محلی بروزرسانی می‌شود و مقدار پشتیبان یال‌های آن مثلث همان‌طور که در شکل ۳-ب نشان داده شده است، به صورت سراسری بروزرسانی می‌شود.

۴-۲- استخراج K-truss به صورت موازی

پس از ایجاد پشتیبان یال‌ها به صورت سراسری و نگاشت یال‌ها به رأس‌های مثلثی مربوط به آن‌ها به صورت محلی، الگوریتم ۳ تمامی یال‌های نامعتبر را به صورت تکرارشونده حذف می‌کند. برای این منظور در هر تکرار، هر نخ پردازشی یک دسته از یال‌ها را انتخاب و آن‌هایی را، که پشتیبان کمتر از $K - 2$ داشته باشند، پیدا می‌کند. این یال‌ها که یال‌های نامعتبر شناخته می‌شوند، باید از مجموعه یال‌های موجود در Sup حذف گردند. حذف هر یال $(u.v)$ منجر به حذف مثلث‌های $\Delta_{u,v,w1}$، $\Delta_{u,v,w2}$، ... و $\Delta_{u,v,wn}$ می‌گردد که در آن رابطه $w_i \in TVs(u, w)$ برقرار است. حذف مثلث‌ها سبب کاهش پشتیبان تمامی یال‌های مثلث‌های حذف شده می‌گردد.

رأس u	$FONL_u$
۱	۴, ۶
۲	۵, ۱, ۴
۳	۱, ۴, ۶
۴	۶
۵	۶, ۷
۶	۴
۷	۴, ۶

P1 P2 P3

رأس u	مجموعه (v, SF_{uv})	اشتراک SF_{uv} با $FONL_u$	TVs	نگاشت یال به رئوس مثلثی
۱	(۴, [۶])	(۴, [۶])	۱	(۱, ۴) -> {۶, ۲}, (۱, ۶) -> {۴}, (۴, ۶) -> {۱}, (۲, ۱) -> {۴}, (۲, ۴) -> {۱}
۲	(۵, [۱, ۴]) (۱, [۴])	(۱, [۴])		
۳	(۱, [۴, ۶]) (۴, [۶])	(۱, [۴, ۶]) (۴, [۶])	۲	(۳, ۱) -> {۴, ۶}, (۴, ۶) -> {۳}, (۱, ۴) -> {۳}, (۳, ۶) -> {۱, ۴}, (۱, ۶) -> {۳}, (۳, ۴) -> {۱, ۶}
۴	–	–		
۵	-		۳	(۷, ۴) -> {۶}, (۷, ۶) -> {۴}, (۴, ۶) -> {۷}
۶	–	–		
۷	(۴, [۶])	(۴, [۶])		

الف) مراحل یافتن رئوس مثلثی با استفاده از FONL در هر نخ پردازشی

یال	پشتیبان
(۱, ۴)	۳
(۱, ۶)	۲
(۴, ۶)	۳
(۲, ۱)	۱
(۲, ۴)	۱
(۳, ۱)	۲
(۳, ۴)	۲
(۳, ۶)	۲
(۷, ۴)	۱
(۷, ۶)	۱

ب) پشتیبان سراسری هر یال

شکل ۳- نحوه عملکرد الگوریتم ۲ بر روی FONL شکل ۲

پس از حذف یال‌های نـامعتبر و بروزرسـانی پشـتیبان بقیـه یال‌هـا، در تکـرار بعدی، دوباره یال‌های نامعتبر شناسایی و بروزرسانی‌ها انجام می‌شود. در صورتی که در یک تکرار هیچ یال نامعتبری کشف نشود، الگوریتم به پایان می‌رسـد و یال‌هـای حذف نشده مشخص کننده زیرگراف K-Truss هستند.

الگوریتم ۳- استخراج K-Truss به صورت موازی

ورودی: Sup ،K ،TVs

خروجی: مجموعه یال‌های با پشتیبان بیشتر یا مساوی $K-2$

```
1  set P to be the number of threads
2  set LocalInvalids to be an array of sets with size P
3  set Invalids to be a set of edges with size |Sup|
4  for each (u, v) ∈ sup do in parallel
5     let t be Id of the current thread
6     if Sup (u, v) < K − 2 then add (u, v) to LocalInvalids[t]
7  do
8      Invalids = ∪t LocalInvalids[t]
9      clear LocalInvalids
10     remove Invalids from Sup
11     for each thread t do in parallel
13         for each (u, v) ∈ Invalids do
14             for each w in TVs[t] (u, v) do
15                 for each z in {u, v} do
16                     if w ≺ z then swap (z, w)
17                     set = TVs[t] (z, w)
18                     if set does not contain z then continue
19                     remove z from set
20                    newSup = decrement sup (z, w)
21                   if newSup = K – 3 then add (z, w) to
                                           LocalInvalids [t]
22 while (Invalids is not empty)
23 return {(u, v) | (u, v) ∈ Sup}
```

شبه کد الگوریتم استخراج زیرگراف K-Truss در الگوریتم ۳ ارائه شده است. این الگوریتم حاوی سه ورودی است: ۱) TVs: رأس‌های مثلثی استخراج شده به ازای هر یال در هر نخ پردازشی؛ ۲) K: ورودی کاربر که عددی صحیح بزرگتر از ۲ است؛ ۳) Sup: پشتیبان سراسری به ازای هر یال. خروجی مورد انتظار نیز شامل زیرمجموعه‌ای (زیرگرافی) از یال‌های گراف اصلی است که در آن پشتیبان همه‌ی یال‌ها بزرگتر یا مساوی $K-2$ باشد. خطوط ۱ تا ۳ از الگوریتم به تعریف متغیرها پرداخته است. به این صورت که P تعداد نخ‌های پردازشی، LocalInvalids یک آرایه به اندازه P است که هر خانه‌ی آن حاوی یک مجموعه برای نگهداری یال‌های نامعتبر در هر نخ پردازشی می‌باشد؛ در نهایت Invalids مجموعه‌ی کل یال‌های نامعتبر پیدا شده در بین تمامی نخ‌های پردازشی است.

پس از تعریف و مشخص شدن متغیرها، در خطوط ۴ تا ۶، هر نـخ t بخشـی از یال‌های موجود در Sup را برمی‌دارد و آن یال‌هایی را کـه دارای پشـتیبان کمتـر از $K-2$ باشند، به مجموعه یال‌هـای نـامعتبر در خانـه tام از آرایـه LocalInvalids اضافه می‌نماید. در اینجا هر یال تنها توسط یک نخ مـورد پـردازش قـرار می‌گیـرد. پیاده‌سازی این بخش از کد به این صورت انجام می‌شود که هر نخ یک بخـش ثابـت (مثلاً ۱۰۰ خانه) پشت سر هم از مجموعه Sup را که توسـط دیگـر نخ‌هـا انتخـاب نشده است، پردازش می‌کند. این کار سبب می‌شود که نرخ اصـابت حافظـه نهـان[۱۹] افزایش یابد. همچنین، هر نخ به طور میانگین تعداد $\frac{m}{p}$ یال را برای پردازش دریافت می‌کند. بنابراین پیچیدگی زمانی این بخـش از الگـوریتم برابـر بـا $O(\frac{m}{p})$ اسـت. در شکل ۴-ب یال‌های نامعتبر یافته شده توسط هر نخ با توجه به پشـتیبان یال‌هـا در شکل ۴-الف مشخص شده است. در این مثال فرض شده است که مقدار K برابر بـا ۴ است. بنابراین، یال‌هایی که مقدار پشتبان آن‌ها کمتر از ۲ است (یعنـی پشـتیبان آن‌ها برابر با ۱ باشد)، به عنوان یال نامعتبر شناخته می‌شوند.

پس از مشخص شدن یال‌های نامعتبر، در یک حلقـه در خطـوط ۷ تـا ۲۲ بـه صورت تکرارشونده، در هر مرحله یال‌های نامعتبر از مجموعه یال‌های موجود حذف می‌شوند، تا زمانی‌که هیچ یال نـامعتبری (شـرط while در خـط ۲۲) پیـدا نشـود. اولین دستور در این حلقه (خط ۸) مجموعه یال‌های نامعتبر را در متغیر Invalids ادغام می‌کند. یک پیاده‌سازی بهینه از این عملیات به این صورت است که در ابتـدا اندازه هر LocalInvalids مشخص شود و اندیس شروع هر نـخ بـرای کپـی کـردن یال‌های نامعتبر در آرایه Invalids تعیین گـردد. سـپس نخ‌هـا بـه صـورت مـوازی یال‌های نامعتبر را در محل تعیین شده در Invalids بنویسند. از آنجا که هر نخ در حالت متوسط می‌تواند $\frac{m}{p}$ یال نامعتبر داشته باشد، پیچیـدگی زمـانی ایـن عملیـات برابر با $O(\frac{m}{p})$ است. این پیچیدگی زمانی برای بـدترین حالـت هـم صـدق می‌کنـد. زیرا، در مراحل قبلی برای تشخیص یال‌های نامعتبر، هـر نـخ در بـدترین حالـت $\frac{m}{p}$ تعداد یال را باید پردازش کند. نتیجه ادغام یال‌های نامعتبر محلـی شـکل ۴-ب در شکل ۴-ج مشخص شده است.

پس از تجمیع یال‌های نـامعتبر از همـه‌ی نخ‌هـا در متغیـر Invalids، خـط ۹ کلیه مجموعه‌های LocalInvalids را تهی می‌کند تا برای مراحل بعـدی الگـوریتم آماده باشند و بتوانند مقادیر جدیـد را در خـود ذخیـره نماینـد. خـط ۱۰ یال‌هـای نامعتبر را از متغیر Sup حذف می‌کند. در کلیه تکرارهای حلقه، دسـتور خـط ۷ در بدترین حالت به تعداد $O(m)$ بـار اجـرا می‌شـود. زیـرا، اگـر همـه یال‌هـا نـامعتبر شناخته شوند، دیگر الگوریتم به پایان می‌رسد. خطوط ۱۱ تـا ۲۱ نشـان می‌دهنـد که همه‌ی نخ‌ها، یال‌های نامعتبر موجـود در آرایـه Invalids را بـه صـورت مـوازی می‌خوانند و با توجه به آن یال‌ها، رأس‌های سوم (متغیر w خط ۱۴) مثلـث حـاوی یال نامعتبر را تشخیص می‌دهند. اگر یال نـامعتبر، $\{u.v\}$ باشـد (خـط ۱۵)، آنگـاه یال‌های $(u.w)$ و $(v.w)$، یال‌های دیگر مثلث $\Delta_{u,v,w}$ می‌باشند.

پشتیبان	یال
۳	(۱, ۴)
۲	(۱, ۶)
۳	(۴, ۶)
۱	(۲, ۱)
۱	(۲, ۴)
۲	(۳, ۱)
۲	(۳, ۴)
۲	(۳, ۶)
۱	(۷, ۴)
۱	(۷, ۶)

الف) پشتیبان یال‌ها

یال نامعتبر
(۲, ۱)

یال نامعتبر
(۲, ۴)

یال نامعتبر
(۷, ۴)
(۷, ۶)

ب) یال‌های نامعتبر در هر نخ

یال نامعتبر
(۲, ۱)
(۲, ۴)
(۷, ۴)
(۷, ۶)

ج) کلیه یال‌های نامعتبر

یال‌های دیگر مثلث	رئوس مثلثی
(۲, ۴) -> {۱}, (۱, ۴) -> {۲}	(۲, ۱) -> {۴}
(۲, ۱) -> {۴} (۱, ۴) -> {۲}	(۲, ۴) -> {۱}
(۷, ۶) -> {۴} (۴, ۶) -> {۷}	(۷, ۴) -> {۶}
(۴, ۶) -> {۷} (۷, ۴) -> {۶}	(۷, ۶) -> {۴}

بروز رسانی‌ها
(۱, ۴) -> {۲}
(۴, ۶) -> {۷}

د) استخراج دیگر یال‌های مثلث مربوط به یال نامعتبر و مشخص کردن بروزرسانی‌های محلی

پشتیبان	یال
۲	(۱, ۴)
۲	(۱, ۶)
۲	(۴, ۶)
۲	(۳, ۱)
۲	(۳, ۴)
۲	(۳, ۶)

ه) پشتیبان نهایی یال‌ها

شکل ۴- مراحل اجرای الگوریتم ۳

بنابراین، با توجه به حذف این مثلث، باید رأس u از مجموعه رئوس مثلثی مربوط به یال $(v.w)$ و رأس v از مجموعه رئوس مثلثی مربوط به یال $(u.w)$ حذف گردند. در شکل ۴-د نشان داده شده است که چطور هر نخ با توجه به رئوس مثلثی محلی مربوط به خود، رئوس مثلثی دیگر یال‌ها را، که باید بروزرسانی شوند، پیدا می‌کند (در ستون سمت راست). همان‌طور که در این شکل مشاهده می‌شود، برخی یال‌ها مثل (۱، ۲)، (۴، ۲)، (۴، ۷) و (۶، ۷) قبلاً از مجموعه یال‌های معتبر در Sup حذف شده‌اند. بنابراین در پایین شکل ۴-د مجموعه یال‌هایی که باید بروزرسانی شود، نشان داده شده است. این بدین‌معنی است که رئوس ۲ و ۷ به ترتیب از مجموعه رئوس مثلثی یال‌های (۴، ۱) و (۶، ۴) باید حذف گردند.

خط ۱۹ حذف یک رأس از رئوس مثلثی یال مربوطه را نشان می‌دهد. اگر رأسی از مجموعه رئوس مثلثی یک یال حذف شود، پشتیبان یال مربوطه باید یکی کم شود (خط ۲۰). هر گاه پشتیبان بروز شده یالی به $K-3$ رسید، به این معنی است که آن یال دیگر یک یال معتبر نیست و باید به LocalInvalids مربوط به نخ جاری اضافه شود (خط ۲۱) تا در آغاز تکراری بعدی در متغیر Invalids قرار گیرد. در این مجموعه دستورات، تنها یک نخ، نامعتبر بودن یک یال را تشخیص می‌دهد، زیرا شرط موجود در خط ۲۱ تنها برای یک نخ اتفاق می‌افتد. شکل ۴-ه پشتیبان سراسری یال‌های معتبر باقیمانده از مثال اجرایی را نشان می‌دهد. در این شکل مشخص می‌شود که پس از یک تکرار، پشتیبان یال‌های باقیمانده همگی بزرگتر از ۲ هستند. بنابراین، برای $K=4$، الگوریتم مدنظر پس از یک تکرار بر روی مثال اجرایی، زیرگراف 4-Truss را تشخیص می‌دهد. برای تحلیل پیچیدگی زمانی این الگوریتم، کافی است به این نکته توجه کنیم که متغیر Invalids توسط همه‌ی نخ‌ها پردازش می‌شود و در طول کل حلقه (خطوط ۷ تا ۲۲)، حداکثر m یال (همه یال‌ها) به عنوان یال نامعتبر در Invalids قرار می‌گیرند. لذا، پیچیدگی زمانی الگوریتم ۳ در بدترین حالت برابر با $O(m)$ است.

۴-۳- تحلیل پیچیدگی

برای استخراج زیرگراف K-Truss، الگوریتم‌های ۱، ۲ و ۳ ارائه شده‌اند که به ترتیب به دنبال هم باید اجرا شوند. بنابراین، پیچیدگی زمانی کلی روش ارائه شده برابر با مجموع پیچیدگی زمانی هر یک از این الگوریتم‌ها است. با توجه به تحلیل‌های صورت گرفته در بخش‌های قبلی، پیچیدگی زمانی الگوریتم‌های ۱، ۲ و ۳، در بدترین حالت، به ترتیب معادل $O(\frac{m}{p})$، $O(\frac{m\sqrt{m}}{P})$ و $O(m)$ محاسبه شده است. با این شرایط، پیچیدگی زمانی کلی روش ارائه شده برابر است با $O\left(\frac{m}{P}+\frac{m\sqrt{m}}{P}+m\right) = O(\frac{m\sqrt{m}}{P}+m)$. در نتیجه، پیچیدگی زمانی روش ارائه شده در بدترین حالت برابر با $O\left(\frac{m}{P}\times\left(\sqrt{m}+P\right)\right)$ است. این پیچیدگی نشان می‌دهد که زمان اجرای الگوریتم با افزایش تعداد پردازنده‌ها کاهش پیدا می‌کند که نشان از مقیاس‌پذیری روش پیشنهادی از نقطه‌نظر تئوری است. با توجه به آزمایش‌های انجام شده در بخش بعدی، این مقیاس‌پذیری در عمل نیز نشان داده شده است.

۵- نتایج آزمایش‌ها

روش پیشنهادی با زبان جاوا پیاده‌سازی و بر روی یک ماشین ۱۲ هسته‌ای با ۱۲۰ گیگابایت حافظه‌ی اصلی اجرا شده است. جدول ۱ مشخصات گراف‌های ورودی در آزمایش‌ها را نشان می‌دهد که از مجموعه گراف‌های واقعی استاندارد موجود در مخازن SNAP [۲۹] و [۳۰] انتخاب شده‌اند. در این بین گراف lj با ۳۵ میلیون یال و ۱۷۷ میلیون مثلث بزرگترین گراف است که متعاقباً انتظار می‌رود به زمان بیشتری برای پردازش نسبت به بقیه گراف‌ها نیاز داشته باشد. همچنین گراف pn با ۱/۴ میلیون یال کوچکترین گراف است. البته با توجه به تعداد مثلث‌های آن، این گراف نسبت به سایر گراف‌ها پرچگال‌تر است و به همین دلیل، زمان پردازش آن نباید اختلاف زیادی با زمان پردازش گراف‌های yb، sk و cp داشته باشد.

جدول ۱- مشخصات مجموعه گراف‌های ورودی

نام گراف	تعداد رأس‌ها	تعداد یال‌ها	تعداد مثلث‌ها
live-journal (lj)	۳/۹ میلیون	۳۵ میلیون	۱۷۷ میلیون
youtube (yb)	۱/۲ میلیون	۳ میلیون	۳ میلیون
as-skitter (sk)	۱/۷ میلیون	۱۱ میلیون	۲۸/۷ میلیون
penn94 (pn)	۴۱/۵ هزار	۱/۴ میلیون	۲۱/۶ میلیون
cit-patents (cp)	۳/۸ میلیون	۱۷ میلیون	۷/۵ میلیون

در این بخش ابتدا رفتار الگوریتم PKT را با اجرای آن بر روی گراف‌های ورودی، با مقادیر مختلف K بررسی می‌کنیم. سپس، زمان اجرای روش پیشنهادی با زمان روش موازی پیشنهاد شده در [۲۱] و روش توزیع شده مبتنی بر چارچوب نگاشت-کاهش در [۲۲] مقایسه می‌شود. برای این کار، از نتایج گزارش شده در مقاله‌های مذکور بهره گرفته‌ایم. با توجه به اینکه تنظیمات آزمایش‌ها در [۲۱] حاوی یک ماشین ۳۲ هسته‌ای با ۲۵۶ گیگابایت حافظه‌ی اصلی و همچنین در [۲۲] حاوی سه ماشین با ۸ هسته و ۱۲۰ گیگابایت حافظه‌ی اصلی است، استفاده از نتایج گزارش شده در این مراجع، مشکلی از منظر مقایسه ایجاد نمی‌کند. زیرا، میزان قدرت پردازشی و حافظه‌ی اصلی استفاده شده در طی اجرای PKT نسبت به دو روش دیگر بسیار کمتر است. در نهایت، مقیاس‌پذیری روش پیشنهادی با بکارگیری تعداد مختلف هسته‌های پردازشی برای گراف‌های ورودی ارزیابی شده است.

زمان اجرای الگوریتم پیشنهادی PKT با مقادیر مختلف K (از ۳ تا ۲۰) بر روی گراف‌های ورودی در نمودار شکل ۵ نشان داده شده است. در این نمودار مشاهده می‌شود که افزایش K، نه تنها سبب افزایش زمان اجرا نمی‌شود، بلکه در برخی موارد به کاهش زمان اجرا نیز منجر می‌گردد. علت این امر این است که با افزایش K، ممکن است تعداد بیشتری یال‌های نامعتبر در یک تکرار یافت شوند و این موضوع موجب کاهش تعداد تکرارهای الگوریتم گردد. همان‌طور که ملاحظه می‌شود، گراف lj بیشترین زمان اجرا را به خود اختصاص داده است. از آنجا که الگوریتم K-Truss وابستگی شدیدی به تعداد یال‌ها و مثلث‌ها دارد و نسبت تعداد یال‌ها و مثلث‌های این گراف به یال‌ها و مثلث‌های سایر گراف‌ها خیلی بیشتر است، چنین نتیجه‌ای منطقی به نظر می‌رسد.

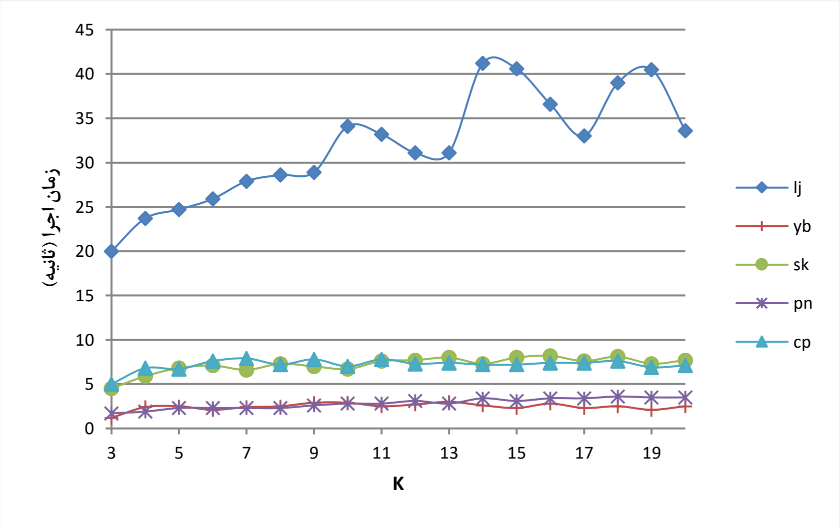
در نمودار شکل ۵، مشاهده می‌شود که گراف‌های sk و cp از یک سو و همچنین گراف‌های yb و pn از سوی دیگر، دارای زمان‌های اجرای مشابهی هستند. با توجه به پیچیدگی زمانی الگوریتم، که برابر با $O(\frac{m}{P}\times(\sqrt{m}+P))$ است و آمارهای داده شده در خصوص تعداد یال‌های گراف‌ها، می‌توان دلیل نزدیکی زمان اجرا را در نزدیک بودن تعداد یال‌های گراف‌های مذکور بیان کرد. البته، تعداد مثلث‌ها و تعداد تکرار الگوریتم نیز، که وابسته به توزیع گراف است، در زمان اجرا تاثیر دارد. در مقایسه دو گراف cp و sk، هرچند تعداد یال‌های cp نسبت به sk بیشتر است، ولی با توجه به بیشتر بودن تعداد مثلث‌های sk نسبت به cp، زمان اجرای sk نسبت به cp بالاتر است. همین تحلیل در خصوص زمان اجرای مربوط به دو گراف yb و pn نیز برقرار است.

به منظور بررسی بیشتر رفتار الگوریتم با توجه به گراف‌های ورودی، در نمودار شکل ۶، تعداد تکرارهای الگوریتم پیشنهادی (الگوریتم ۳) با توجه به مقادیر مختلف K و در نمودار شکل ۷، زمان اجرا در تکرارهای مختلف نشان داده شده است.

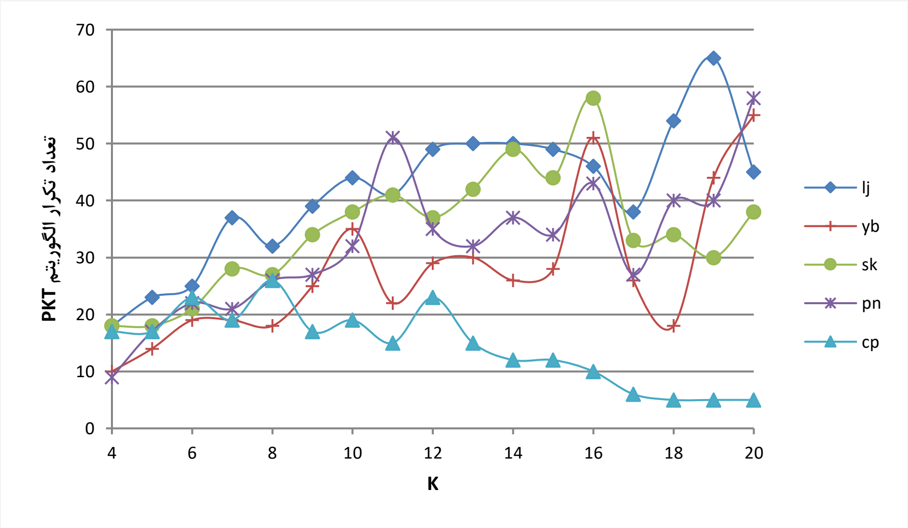
نمودار شکل ۶ نشان می‌دهد که به ازای مقادیر مختلف K، تعداد تکرارهای الگوریتم متفاوت است. کاهش تعداد تکرار به ازای افزایش K، نشان‌دهنده این است که تعداد بیشتری از یال‌های نامعتبر، با افزایش K، در یک تکرار کشف شده‌اند. بنابراین، از آنجا که تعداد یال‌های نامعتبر بیشتری در یک تکرار از گراف حذف می‌شوند، تعداد تکرارهای الگوریتم کاهش می‌یابد. این وضعیت بیان‌گر تاثیر توزیع گراف بر تعداد تکرارها در مقادیر مختلف K است. زیرا، تشخیص نامعتبر بودن با توجه به پشتیبان یال‌ها صورت می‌گیرد و پشتیبان یال‌ها وابسته به ارتباط بین رأس‌ها و مثلث‌های ایجاد شده است. در این نمودار مشاهده می‌شود که در گراف cp، با افزایش K از یک بخش به بعد، تعداد تکرار کاهش یافته است. تحلیل این رفتار نیز با توجه به توزیع مثلث‌ها در گراف است. این بدین معنی است که نقاط پرچگال در این گراف، تنها در بخشی از گراف قرار گرفته است.

شکل ۷ بیانگر رابطه‌ی بین مدت زمان اجرا و تعداد تکرارهای الگوریتم است. همان‌طور که مشاهده می‌شود، به غیر از گراف lj، در سایر موارد، تعداد تکرارها تاثیر مستقیمی بر زمان اجرا نداشته است. بیشترین تفاوت lj با سایر گراف‌ها در تعداد یال‌ها و مثلث‌های آن است. علاوه بر آن، زیاد بودن تعداد بروزرسانی‌های یال‌ها در هر تکرار نیز می‌تواند سبب افزایش زمان اجرا در lj با افزایش تکرارها باشد.

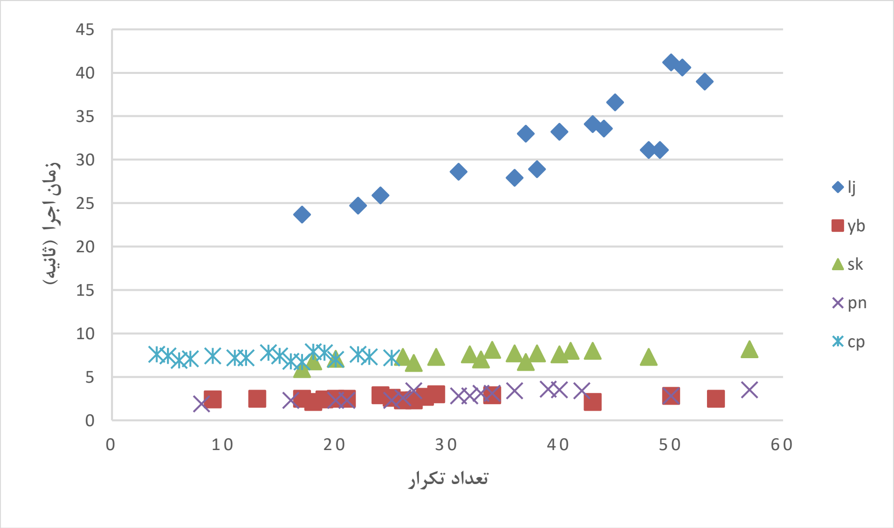
مقایسه زمان اجرای الگوریتم پیشنهادی PKT با دیگر روش‌های ارائه شده در [۲۱، ۲۲] در شکل ۸ نشان داده شده است. در این شکل، با توجه به اینکه از گزارش‌های ارائه شده در مقالات اصلی استفاده شده است، مقایسه با [۲۱] شامل گراف‌های pn، lj و sk و مقایسه با [۲۲] صرفا شامل yb است. روش ارائه شده در [۲۱] همانند PKT یک روش موازی چندهسته‌ای است. با وجود اینکه آزمایش‌های انجام شده در [۲۱] بر روی یک ماشین با ۳۲ هسته و ۲۵۶ گیگابایت حافظه اصلی بوده است، روش پیشنهادی (PKT) با اجرا روی یک ماشین ۱۲ هسته‌ای با ۱۲۰ گیگابایت حافظه، توانسته است در اکثر مواقع از روش [۲۱] پیشی بگیرد. البته، هر چند زمان اجرای PKT در گراف pn با اختلاف بسیار ناچیز بیشتر است، اما، با توجه به کم‌تر بودن منابع سخت‌افزاری در اجرای آزمایش‌های PKT، برتری آن مشخص است؛ زیرا نسبت تعداد هسته‌های پردازشی به زمان اجرا در PKT در این گراف و سایر گراف‌ها، نسبت به [۲۱] کمتر است. با این وجود، برای تحلیل این موضوع که چرا بر خلاف سایر گراف‌ها، در این مورد زمان اجرای PKT بیشتر است، می‌توان به این نکته اشاره کرد که در گراف pn، نسبت تعداد یال‌ها به رأس‌ها بسیار زیاد است.



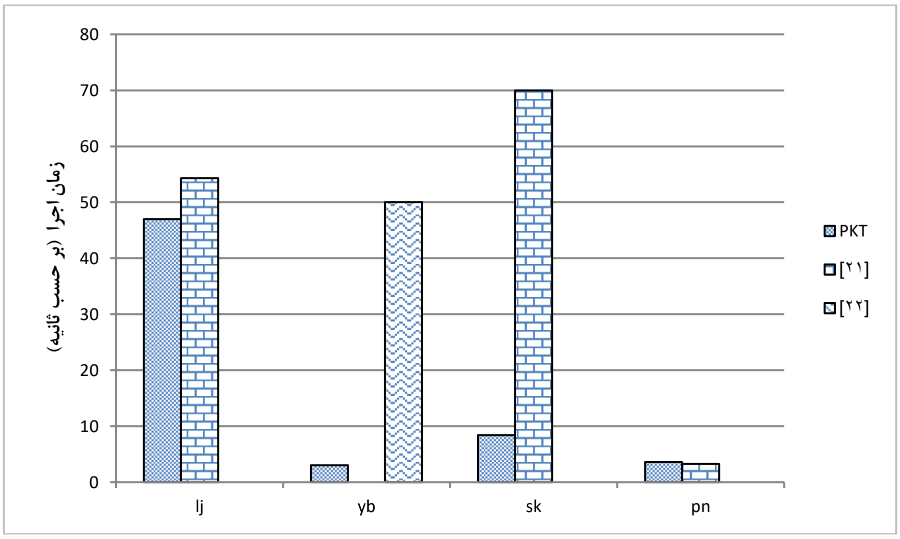
شکل ۵- زمان اجرای PKT بر حسب مقادیر مختلف K



شکل ۶- تاثیر مقادیر مختلف K بر تعداد تکرارهای الگوریتم PKT



شکل ۷- زمان‌های اجرای به دست آمده در تکرارهای مختلف الگوریتم



شکل ۸- مقایسه زمان اجرا بین روش پیشنهادی (PKT) و دیگر روش‌های ارائه شده در [۲۱] و [۲۲]

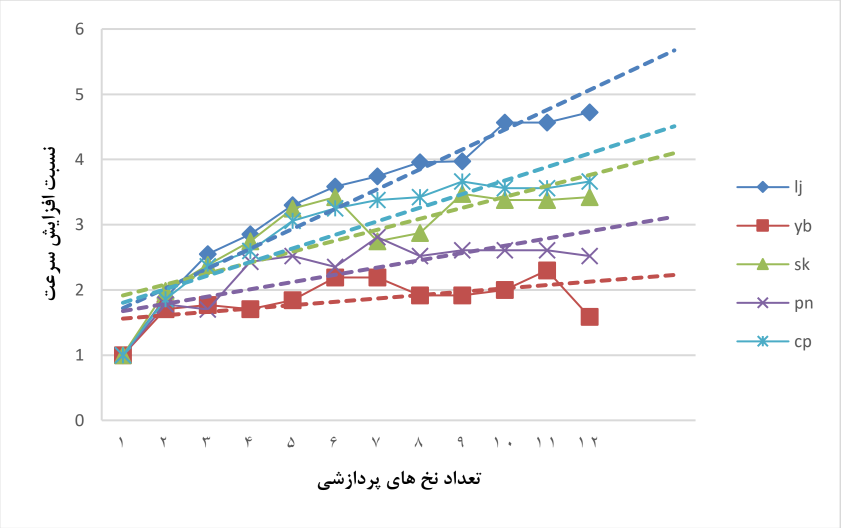
از آنجا که در مرحله یافتن مثلث‌ها، موازی‌سازی در [۲۱] بر حسب یال انجام می‌گیرد، در حالی‌که PKT این کار را بر حسب رأس انجام می‌دهد، می‌توان به این نتیجه رسید که در مرحله یافتن مثلث‌ها و محاسبه‌ی پشتیبان یال‌ها برای گراف pn در مقایسه با سایر گراف‌ها، توزیع بار بین هسته‌های پردازشی در [۲۱] بهتر صورت گرفته است.

بر خلاف lj، در ارتباط با گراف sk، بهبود زمان اجرا در روش PKT نسبت به روش [۲۱] بسیار زیاد است. در روش PKT، بروزرسانی پشتیبان یال‌ها خیلی سریع‌تر از [۲۱] انجام می‌گیرد. زیرا در [۲۱] پس از یافتن هر یال نامعتبر، باید بین همسایگان دو رأس آن یال اشتراک گرفته شود که این عمل هزینه‌بر است. یعنی، در گراف sk، رأس‌های مربوط به یال‌های نامعتبر دارای همسایگان غیرمشترک زیادی هستند.

روش دیگر مقایسه شده [۲۲] با روش پیشنهادی، یک روش توزیع شده است که از چارچوب هدوپ به عنوان بستر اجرا استفاده نموده است. با توجه به اینکه هدوپ نتایج میانی را در دیسک ذخیره می‌کند، زمان اجرای آن به شدت بالا می‌رود. البته مزیت استفاده از هدوپ، قابلیت اجرا روی مجموعه داده‌های خیلی حجیم است؛ ولی ممکن است زمان اجرا ساعت‌ها به طول بیانجامد. در اینجا، مقایسه زمان اجرای PKT و روش ارائه شده در [۲۲] برای گراف yb، به خوبی نشان می‌دهد که روش پیشنهادی با کارایی خیلی بالاتری توانسته است کار خود را به پایان برساند. البته یک روش چندهسته‌ای دارای مزیت استفاده از حافظه‌ی مشترک است که این عامل نقش خیلی مهمی در زمان اجرا دارد.

در نهایت، مقیاس‌پذیری الگوریتم پیشنهادی با استفاده از تعداد مختلف نخ‌های پردازشی (از ۱ تا ۱۲) و مقدار $K = 10$ (با توجه به شکل ۵، زمان اجرا در این مقدار تقریبا برابر با میانگین زمان‌های اجرا به ازای مقادیر مختلف K است) در شکل ۹ نشان داده شده است. در نمودار این شکل، نسبت زمان اجرا در حالت استفاده از چند نخ (یا چند پردازنده) نسبت به زمان اجرا در حالت استفاده از یک نخ (یا یک پردازنده) مشخص شده است؛ محور عمودی این شکل، با عنوان "نسبت افزایش سرعت" موید همین مقدار است.

مطابق با این شکل، در حالت کلی، افزایش تعداد پردازنده‌ها سبب کاهش زمان اجرا می‌شود. هر چه گراف بزرگ‌تر باشد، با توجه به نیاز بیشتر به منابع برای پردازش، استفاده از پردازنده‌های بیشتر سبب افزایش سرعت به شکل محسوس‌تری می‌شود.



شکل ۹- نسبت زمان اجرا در حالت استفاده از چندپردازنده به یک پردازنده

به عنوان مثال، برای گراف lj میزان مقیاس‌پذیری بالاتری مشاهده شده است. برای مجموعه داده‌های کوچک‌تر، گاهی مشاهده شده است که این نسبت با افزایش تعداد نخ‌ها کاهش یافته است. یک دلیل برای این رخداد، غیرمعتبر شدن برخی مجموعه داده‌های موجود در حافظه‌ی نهان پردازنده به علت بروزرسانی عناصری از آن مجموعه داده‌ها توسط یک نخ و نیاز به خواندن عناصر دیگری از همان مجموعه داده‌ها توسط نخ دیگر است. زیرا، داده‌ها در حافظه‌ی نهان به صورت مجموعه‌ای بارگذاری می‌شوند و اگر به ازای یک مجموعه داده، یک عنصر توسط یک نخ بروزرسانی شود و نخ دیگر نیاز به خواندن عنصر دیگری داشته باشد، آنگاه نیاز است که مجموعه داده مذکور در حافظه اصلی نوشته شود و این موضوع به کاهش کارایی منجر می‌شود. دلیل دیگر کاهش کارایی، دسترسی همزمان چندین هسته از طریق یک درگاه مشترک به یک بخش از حافظه و در نتیجه ایجاد گلوگاه کارایی است.

۶- نتیجه‌گیری

در این مقاله، یک الگوریتم چندهسته‌ای برای استخراج زیرگراف K-Truss ارائه شده است. برای این منظور، ابتدا گراف ورودی به یک ساختار مناسب برای افزایش موازی‌سازی تبدیل می‌شود. سپس با استفاده از این ساختار، مثلث‌های گراف به صورت موازی تشخیص داده می‌شوند. موازی‌سازی در این روش بر مبنای رئوس صورت گرفته است. پس از یافتن یک مثلث در هر نخ، دو ساختار داده بروزرسانی می‌شود: ۱- یک ساختار داده سراسری که مربوط به پشتیبان هر یال (تعداد مثلث‌هایی که یال در آن شرکت دارد) بوده و ۲- یک ساختار محلی به ازای هر نخ که نمایانگر تعداد رأس‌های مثلثی (رأس‌های مقابل هر یال در مثلث‌های یال) هر یال است. استخراج این دو ساختار سبب افزایش کارایی در مراحل بعدی الگوریتم می‌شود، به‌طوری‌که، در یک روش تکرارشونده، این ساختارها به صورت موازی بررسی و بروزرسانی می‌شوند. آزمایش‌های صورت گرفته نشان‌دهنده کارایی بالای روش پیشنهادی در مقایسه با دیگر روش‌ها است.

در خصوص کارهای آتی، دو رویکرد می‌تواند مدنظر باشد. یکی نگهداری گراف به صورت فشرده و کاهش حافظه‌ی موردنیاز به منظور اجرای گراف‌های حجیم‌تر است. استفاده از کدگذاری‌های مناسب در نگهداری شناسه‌ی یال‌ها و رأس‌ها می‌تواند سبب کاهش مصرف حافظه گردد. همچنین، با توجه به بسترهای پردازشی داده‌های حجیم موجود مثل هدوپ، می‌توان رویکرد ارائه شده در این مقاله را با توجه به انتزاع‌های معرفی شده در این بسترها بازنویسی کرد و گراف‌های بزرگ‌تری را، که در یک ماشین جای نمی‌گیرند، پردازش نمود.

مراجع

[1] S. Koujaku, I. Takigawa, M. Kudo, and H. Imai, "Dense core model for cohesive subgraph discovery," *Social Networks*, vol. 44, pp. 143-152, 2016.

[2] L. Qin, R. H. Li, L. Chang, and C. Zhang, "August. Locally densest subgraph discovery," *Proc. 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 965-974, 2015.

[3] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek, "May. Finding the hierarchy of dense subgraphs using nucleus decompositions," *Proc. 24th International Conference on World Wide Web*, pp. 927-937, 2015.

[4] R. D. Luce, "Connectivity and generalized cliques in sociometric group structure," *Psychometrika*, vol. 15, no. 2, pp.169-190, 1950.

[5] R. J. Mokken, "Cliques, clubs and clans," *Quality & Quantity*, vol. 13, no. 2, pp.161-173, 1979.

[6] S. B. Seidman, and B. L. Foster, "A graph theoretic generalization of the clique concept*," *Journal of Mathematical sociology*, vol. 6, no. 1, pp.139-154, 1978.

[7] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269-287, 1983.

[8] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, p.16. 2008.

[9] F. Zhao, and A. K Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *Proc. VLDB Endowment*. vol. 6, no. 2, VLDB Endowment, 2012.

[10] A. E. Sariyüce, and A. Pinar, "Fast hierarchy construction for dense subgraphs," *Proc. VLDB Endowment*, vol. 10, no. 3, pp. 97-108, 2016.

[11] U. Redmond, H. Martin, and C. Pádraig, "Mining dense structures to uncover anomalous behaviour in financial network data," *Modeling and mining ubiquitous social media*, pp. 60-76, 2012.

[12] M. G. Rossi, and V. Michalis, "Exploring Network Centralities in Spreading Processes," *International Symposium on Web AlGorithms (iSWAG)*. 2016.

[13] M. G. Rossi, D. M. Fragkiskos, and V. Michalis, "Locating influential nodes in complex networks," *Scientific reports* 6, (2016).

[14] M. G. Rossi, D. M. Fragkiskos, and V. Michalis, "Spread it good, spread it fast: Identification of influential nodes in social networks," *Proc. 24th International Conference on World Wide Web, ACM*, 2015.

[15] R. Bredereck, C. Komusiewicz, S. Kratsch, H. Molter, R. Niedermeier, and M. Sorge, "Assessing the computational complexity of multi-layersubgraph detection," in International Conference on Algorithms and Complexity. Springer, 2017, pp. 128–139.

[16] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng, "Approximateclosest community search in networks," Proceedings of the VLDBEndowment, vol. 9, no. 4, pp. 276–287, 2015.

[17] X. Huang, "Querying k-truss community in large and dynamic graphs," *Proc. ACM SIGMOD international conference on Management of data*. ACM, 2014.

[18] M. Alemi, H. Haghighi, and S. Shahrivari, "CCFinder: using Spark to find clustering coefficient in big graphs," *The Journal of Supercomputing*, pp. 1-28, 2017.

[19] J. Wang, and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endowment*, vol. 5. no. 9, pp. 812-823, 2012.

[20] Z. Zhaonian, and R. Zhu, "Truss decomposition of uncertain graphs," *Knowledge and Information Systems*, vol. 50, no. 1, pp. 197-230, 2017.

[21] R. A. Rossi, "Fast triangle core decomposition for mining large graphs," *Proc. Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 310-322, 2014.

[22] P. L. Chen, C. K. Chou, and M. S. Chen, "Distributed algorithms for k-truss decomposition," *Proc. In Big Data (Big Data), 2014 IEEE International Conference on*, pp. 471-480, 2014.

[23] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," *Proc. the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 613-624, 2014.

[24] J. Cohen, "Graph twiddling in a MapReduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29-41, 2009.

[25] J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp.107-113, 2008.

[26] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[27] L. Quick, P. Wilkinson, and D. Hardcastle, "Using Pregel-like large scale graph processing frameworks for social network analysis," *Proc. the 2012 International Conference on Advances in Social Networks Analysis and Mining*, pp. 457-463, 2012.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," *Proc. the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135-146, 2010.

[29] SNAP: Stanford Network Analysis Project, http://snap.stanford.edu, June 2017.

[30] Network Repository, http://networkrepository.com, June 2017.

مهدی عالمی در حال حاضر کاندیدای دکتری رشته مهندسی کامپیوتر (گرایش نرم افزار) در دانشگاه شهید بهشتی تهران می‌باشد. ایشان در زمینه‌های مربوط به کلان داده در کاربردهایی از جمله موتور جستجو و تحلیل وب و ترافیک شبکه فعالیت دارد. زمینه‌های پژوهشی ایشان شامل زیرگراف کاوی در گراف‌های حجیم، ذخیره و بازیابی اطلاعات و داده‌های حجیم می‌باشد.
آدرس پست‌الکترونیکی ایشان عبارت است از:
m_alemi@sbu.ac.ir

حسن حقیقی در سال ۱۳۸۹ مدرک دکتری خود را در رشته مهندسی کامپیوتر (گرایش نرم‌افزار) از دانشگاه صنعتی شریف دریافت کرد و در حال حاضر دانشیار دانشکده مهندسی و علوم کامپیوتر در دانشگاه شهید بهشتی تهران می‌باشد. زمینه‌های پژوهشی ایشان شامل روش‌های صوری در چرخه توسعه نرم‌افزار، آزمون نرم‌افزار و معماری نرم‌افزار می‌باشد
آدرس پست‌الکترونیکی ایشان عبارت است از:
h_haghighi@sbu.ac.ir

اطلاعات بررسی مقاله:

تاریخ ارسال:۱۳۹۶/۰۴/۱۴
تاریخ اصلاح: ۱۳۹۷/۰۲/۱۲
تاریخ قبول شدن: ۱۳۹۷/۰۴/۰۱
نویسنده مرتبط: مهدی عالمی، دانشکده مهندسی و علوم کامپیوتر، دانشگاه شهید بهشتی، تهران، ایران.

---

[1]Cohesive Subgraphs
[2]Clique
[3]Connectivity
[4]Cohen
[5]Iterative
[6]IO-Efficient
[7]Processing Threads
[8]Fine-Grained
[9]Triangle Vertex Set
[10]Support
[11]Invalid Edge
[12]In-Memory
[13]External-Memory
[14]Map-Reduce
[15]Hadoop
[16]Key-Value
[17]Pregel
[18]Atomic
[19]Cache Hit Rate

The CSI Journal on
Computing Science and
Information Technology
Vol. 16, No. 1, 2018

A Parallel Algorithm for Cohesive Sub-Graph Mining in Big Graphs

Mehdi Alemi **Hassan Haghighi**

Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran

ABSTRACT

Cohesive subgraph detection has an important role in many applications like search engines and biological networks. Among the cohesive subgraphs, K-Truss is getting important because of its simplicity and many usages, e.g., graph visualization and finding dense parts from the graph. Considering big graphs, parallel algorithms should be taken into account in order to get reasonable response time. In this regards, to mine K-Truss subgraphs, we have proposed a parallel algorithm in two phases. For the first phase, a concurrent method has been developed to find all the triangles from the given graph. In addition, a new data structure, called triangle vertices (TVs), has been designed to maintain a set of vertices constructing a triangle with an edge. In the second phase, iteratively, invalid edges are excluded from TVs by a parallel algorithm until no edge disobeying the K-Truss attributes is found. According to the experiments on a machine with 12 cores, using the real graphs, the performance of the proposed approach is proved against the other competitors. Moreover, by configuring different cores, the scalability of our method is presented.

Keywords: Cohesive Communities, Social Networks, Parallel Processing, Triangle Counting, K-Truss.