دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

پایان‌نامه کارشناسی

گرایش نرم افزار

عنوان

پیاده سازی نرم افزار رنگ آمیزی گراف های بزرگ

نگارش

وحید پوریوسف

استاد راهنما

دکتر علیرضا باقری

استادمشاور

دکتر امیرحسین پی براه

شهریور ۱۳۹۵

صفحه فرم ارزیابی و تصویب پایان نامه- فرم تأیید اعضاء کمیته دفاع

در این صفحه فرم دفاع یا تایید و تصویب پایان نامه موسوم به فرم کمیته دفاع- موجود در پرونده آموزشی- **را قرار دهید.**

نکته مهم:

نگارش پایان نامه/رساله باید به *زبان فارسی* و بر اساس آخرین نسخه دستورالعمل و راهنمای تدوین پایان نامه های دانشگاه صنعتی امیرکبیر باشد.(دستورالعمل وراهنمای حاضر)

*** چاپ و صحافی پایان نامه/رساله بصورت دورو بلامانع است.**

به نام خدا

تعهدنامه اصالت اثر

تاریخ:

اینجانب وحید پوریوسف متعهد می‌شوم که مطالب مندرج در این پایان نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آنها استفاده شده است مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتر ارائه نگردیده است.

در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان نامه متعلق به دانشگاه صنعتی امیرکبیر می‌باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه‌برداری، ترجمه و اقتباس از این پایان نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر مآخذ بلامانع است.

وحید پوریوسف

امضا

جا دارد از جناب آقای دکتر علیرضا باقری استاد راهنمای پروژه سپاسگزاری نمایم.

همچنین از راهنمایی‌ها و پیگیری‌های مجدانه جناب آقای دکتر امیرحسین پی‌براه در تمام مراحل این پروژه کمال قدردانی را دارم.

چکیده

در این پروژه با استفاده از چارچوب Spark و Graphx نرم افزاری ساخته شده تا تعدادی از الگوریتم های رنگ آمیزی گراف بر روی هر گراف ساده دلخواهی اجرا شود و خروجی به صورت گرافیکی نمایش داده شود. همچنین تعداد کمترین رنگ لازم برای رنگ آمیزی گراف و زمان پردازش نیز به عنوان خروجی به کاربر نمایش داده می شود. این نرم افزار فارغ از نوع الگوریتم به دو صورت اجرا بر روی یک کامپیوتر و اجرا به صورت توزیع شده را پیشتیبانی می کند که در حالت توزیع شده امکان پردازش بر روی گراف های بزرگ امکان پذیر است و بخشی از پردازش ها به صورت موازی بر روی دستگاههای پردازشگر انجام می شود.

واژه‌های کلیدی:

پردازش توزیع شده، رنگ آمیزی گراف، Spark، Graphx

فهرست عناوین

صفحه

فهرست اشکال

صفحه

صفحه

فهرست جداول

فهرست نمودارها صفحه ها

فهرست علائم

علائم یونانی

$\chi(G)$	عدد رنگی گراف G
$\omega(G)$	عدد دسته یا گروه گراف G
Δ	بزرگترین درجه در گراف

۱

فصل اول

مقدمه

مقدمه

گراف ها هر چند در علوم مختلف فیزیک، زیست شناسی و... کاربردهایی دارند اما شاید در هیچ علم دیگری به اندازه علم کامپیوتر از آن بهره نبرده باشد. از این جهت باید گراف ها را در علم کامپیوتر به صورت خاص مورد بررسی قرار داد و مسائل مختلف آن را حل نمود. دلیل اهمیت فراوان گراف در علم کامپیوتر این است که توصیف بسیاری از ساختارهای داده ای در ساختمان داده های دیگر یا قابل نمایش و ذخیره سازی نیست یا به صورت بهینه این اماکان مهیا نمی باشد. از این رو لازم است بسیاری از اطلاعات به صورت یک گراف ذخیره شود تا ارتباطات بین داده ها ساده تر قابل نمایش باشد و پردازش های آن بهتر و بهینه تر انجام شود.

با اینکه گراف یک نحوه مناسب برای ذخیره سازی انواع مختلفی از اطلاعات است، اما بدست آوردن برخی اطلاعات از برخی گراف ها از نظر پردازشی پرهزینه است. لذا الگوریتم های متنوعی برای مسائل مختلف پیرامون گراف ارائه شده است با این حال بسیاری از مسائل نیز در حال حاضر حل نشده باقی مانده اند و یا از نظر زمانی غیرقابل انجام است. حتی برخی الگوریتم ها با اینکه برای گراف های کوچک به سرعت پاسخ می دهند، اما برای گراف های بزرگ در زمان محدود قابلیت پاسخگویی ندارند. در نتیجه باید به فکر استفاده از قدرت پردازشی متشکل از دسته ای از کامپیوترها باشیم.

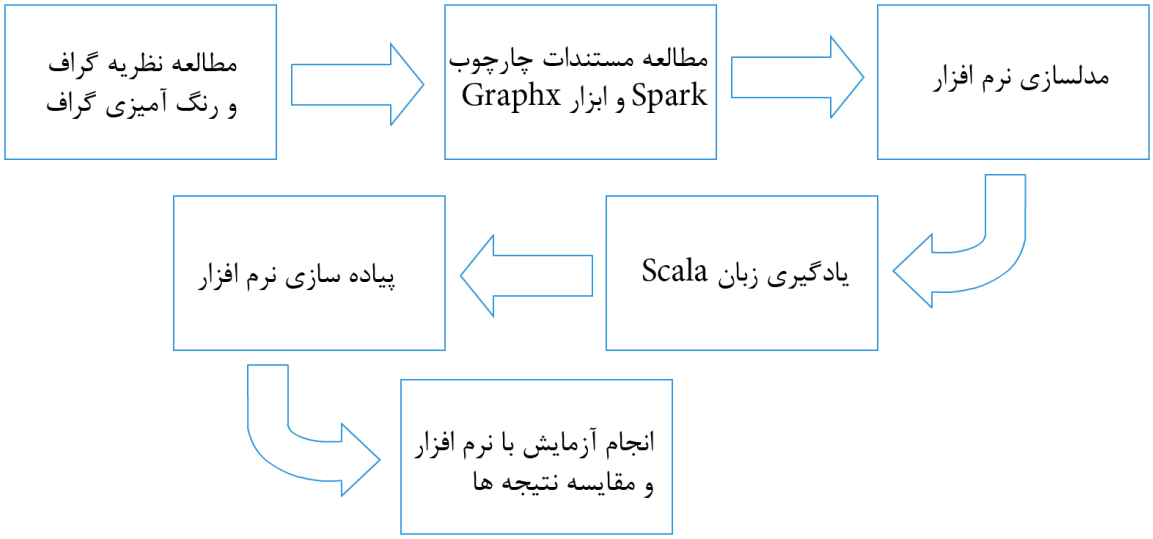
فارغ از زمان لازم برای پردازش گراف های بزرگ، بسیاری از مواقع حتی ذخیره سازی گراف ها بر روی یک کامپیوتر نیز ما را دچار مشکل خواهد کرد. زمانبر بودن خواندن و نوشتن گراف بر روی یک هارد دیسک، اشغال فضای حافظه موقت برای بارگیری گراف از روی هارددیسک و کمبود حافظه هارددیسک برای ذخیره توپولوژی و داده های گراف از جمله مشکلات ذخیره سازی گراف بر روی یک سیستم است.

موارد فوق ما را به این مهم راهنمایی می کند که علاوه بر استفاده از سیستم های قدرتمندتر، باید از قدرت چندین کامپیوتر در یک شبکه استفاده کرد و هر کامپیوتر مسئول ذخیره سازی یا پردازش بخشی از تمامی گراف را بر عهده بگیرد. این راهکار هر چند ظرفیت ذخیره سازی و قدرت پردازش را بسیار گسترش پذیر خواهد کرد، اما مسائل متعددی را نیز به همراه خود خواهد داشت که هدف از سیستم های توزیع شده بررسی و حل اینگونه مسائل است.

در این پروژه که کاری عملی در کنار کاری تئوری می باشد ابتدا لازم است تئوری آن را مورد مطالعه قرار دهیم، تعاریف موجود در این مبحث را ارائه کنیم و پس از روشن شدن فضا و ابعاد کار به پیاده سازی این نرم افزار بپردازیم.

از این رو در فصل دوم مختصری از تئوری رنگ آمیزی گراف گفته خواهد شد و برخی از الگوریتم ها به زبان ساده توضیح داده می شود. در فصل سوم با چارچوب Spark که ابزاری برای اجرای پردازش ها به صورت توزیع شده است و Graphx که به عنوان ابزاری برای اجرای پردازش بر روی گراف ها در بستر توزیع شده Spark است آشنا خواهید شد. در فصل چهارم مراحل مدلسازی و طراحی نرم افزار ارائه می شود. و همچنین مراحل و نکات پیاده سازی نرم افزار و نحوه کار با نرم افزار شرح داده خواهد شد. در فصل پنجم آزمایشاتی با این نرم افزار انجام شده و خروجی نرم افزار برای الگوریتم های مختلف ارائه می شود. در نهایت در فصل ششم جمع بندی و نتیجه گیری از این پروژه ارائه شده و کارهایی که پیرامون این پروژه می توان انجام داد نیز پیشنهاد داده می شود.

فصول ذکر شده تقریبا همان جریان کاری[۱] آبشاری[۲] است که در نمودار ۱ مشاهده می کنید.



نمودار ۱ جریان کاری انجام پروژه

[۱] Work flow

[۲] Waterfall(Sequential)

۲

فصل دوم

مختصری از تئوری رنگ آمیزی گراف

مختصری بر تئوری رنگ آمیزی گراف

رنگ آمیزی گراف ها یکی از مسائل جذاب در نظریه گراف است و از آنجایی که این مسئله از نوع NP-Complete است[۱] ، برای آن میتوان الگوریتم ها زیادی معرفی کرد. ابتدا لازم است مسئله رنگ آمیزی گراف را تبیین کنیم و اینکه در این مبحث چه چیزی را جستجو می کنیم. الگوریتم های مختلفی را معرفی کنیم و پس از پیاده سازی نرم افزار می توانیم آزمایشاتی را انجام دهیم.

۱.۲ تعاریف نظریه رنگ آمیزی گراف

مسئله رنگ آمیزی گراف یا به صورت دقیقتر رنگ آمیزی رئوس گراف به مسئله ای گفته می شود که در آن می خواهیم با کمترین تعداد رنگ تمام رئوس گراف را رنگ آمیزی کنیم به طوری که هیچ دو رأس همجواری همرنگ نباشند. به طور معمول در مسائل رنگ آمیزی گراف گراف بدون جهت[۳] مد نظر است که در این پروژه نیز این نوع گراف مورد بررسی قرار می گیرد.

در رنگ آمیزی گراف چندین اصطلاح پرکاربرد وجود دارد:

عدد رنگی[۴]: به کمترین تعداد رنگی که میتوان گراف G را رنگ آمیزی کرد و با نماد $\chi(G)$ نمایش می دهند.

عدد دسته یا گروه[۵]: به تعداد رئوس بزرگترین دسته از دسته های گراف G و با نماد $\omega(G)$ نمایش می دهند. منظور از دسته یا گروه به زیر مجموعه ای از گراف می گویند که آن زیرمجموعه گراف کامل[۶] باشد.

---

[۳] Undirected Graph

[۴] Chromatic Number

[۵] Clique Number

[۶] Complete Graph

بیشتر مباحث رنگ آمیزی گراف در خصوص رنگ آمیزی رئوس بوده و از این رو عدد رنگ و دیگر مباحث مرتبط در خصوص رنگ آمیزی رئوس گراف است و به اختصار رنگ آمیزی گراف گفته می شود مگر به طور صریح از واژه یال استفاده شود.

۲.۲ نظریه رنگ آمیزی گراف

طبق تعاریف انجام شده در مسئله رنگ آمیزی گراف به دنبال یافتن عدد رنگ $\chi(G)$ هستیم. طبق تعریف رنگ آمیزی از آنجایی که بیشترین درجه Δ است پس با فرض اینکه رأس v دارای Δ رأس مجاور باشد، حداکثر رنگ مصرفی توسط رئوس مجاور به تعداد Δ رنگ خواهد بود و v را می توان با یکی از رنگ های c۱، c۲، ... و $c\Delta+1$ رنگ آمیزی کرد. پس همواره $\chi(G)\leq\Delta+1$ است[۲] .

همچنین طبق تعریف عدد دسته، در یک گراف اگر عدد دسته $\omega(G)$ داشته باشیم یعنی زیرگرافی از $\omega(G)$ رأس داریم که تشکیل یک گراف کامل می دهند. میدانیم هر دو رأسی در یک گراف کامل همجوار هستند لذا نمیتوانند همرنگ باشند. از آنجایی که به تعداد $\omega(G)$ رأس داریم که دو به دو با هم همجوارند، لذا لااقل به $\omega(G)$ رنگ برای رنگ آمیزی آنها نیازمندیم[۳] . پس همواره برای گراف G داریم $\chi(G)\geq\omega(G)$.

در نتیجه هر الگوریتم رنگ آمیزی قابل استفاده ای لااقل باید شرط $\omega(G)\leq\chi(G)\leq\Delta+1$ را داشته باشد.

اما علاوه بر عدد رنگ پارامترهای دیگری همچون پیچیدگی زمان، تعداد و میزان جابه جایی اطلاعات، حافظه و... نیز بسته به کاربرد دارای اهمیت خواهد بود که در بحث نظری بیشتر به پیچیدگی زمانی و عدد رنگ پرداخته می شود اما در پیاده سازی و در عمل نمیتوان از سایر پارامترها چشم پوشی کرد.

۳.۲ معرفی چند الگوریتم متداول در تئوری رنگ آمیزی گراف

همانطور که قبلا گفته شد مسئله رنگ آمیزی گراف یک مسئله NP-Complete است برای همین الگوریتم های مختلفی ارائه شده است که هر کدام نسبت به دیگری مزایا و معایبی دارد و الگوریتم

کاملی که بتواند جواب صحیح را برای تمام گراف ها پیدا کند تاکنون کشف نشده است. در ادامه چندین الگوریتم را مطرح کرده و به طور مختصر شرح داده می شود.

۱.۳.۲ الگوریتم ترتیبی[۷] [۳]

ساده ترین الگوریتم رنگ آمیزی گراف را شاید بتوان الگوریتم ترتیبی دانست. این الگوریتم الگوریتمی حریصانه[۸] است با دید فکر مرکزی به حل این مسئله می پردازد بدین معنی که به صورت توزیع شده و موازی عمل نمی کند. در نتیجه تمامی گراف در اختیار یک کامپیوتر بوده و الگوریتم را به صورت ترتیبی بر روی آن اجرا می کند.

ترتیب انتخاب رئوس بر اساس یک تابع اکتشافی[۹] انجام می شود. در ساده ترین حالت تابع تصادفی را انتخاب میکنیم می توانیم توابع اکتشافی دیگری را برای ترتیب انتخاب رئوس استفاده کنیم، به عنوان مثال: انتخاب رئوس از بزرگترین درجه

نحوه عملکرد الگوریتم (با فرض انتخاب رئوس به ترتیب تصادفی):

گام ۱- مجموعه S شامل تمام رئوس گراف را تعریف میکند

گام ۲- اگر مجموعه S تهی نباشد یک رأس از مجموعه S انتخاب میکند (به طور تصادفی) و اگر تهی بود به گام ۶ می رود.

گام ۳- از بین رنگ های ۱ تا $\Delta+1$ به جز رنگهای رئوس مجاورش (اگر رنگ شده اند)، رنگ با کمترین شماره را انتخاب میکند

گام ۴- رأس رنگ شده، رنگ خود را به تمام رئوس همجوارش اعلام می کند

گام ۵- رأس فعلی از مجموعه S حذف می کنیم و به گام ۲ می رود

---

[۷] Sequential Algorithm

[۸] Greedy Algorithm

[۹] Heuristic Function

گام ۶ – پایان

همانطور که دیده شد این الگوریتم در هر مرحله یک رأس را به صورت قطعی بر اساس ترتیبی که انتخاب کرده رنگ آمیزی می کند و این الگوریتم حریصانه با توجه به تابع اکتشافی خود می تواند بهبود یابد.

۲.۳.۲ الگوریتم حریصانه توزیع شده[۱۰][۳]

این الگوریتم به الگوریتم ترتیبی که الگوریتمی حریصانه بود شباهت زیادی دارد. با این تفاوت که در الگوریتم قبل تمامی رئوس به ترتیب یکی پس از دیگری رنگ آمیزی می شدند اما در این الگوریتم چندین رأس را می توان به صورت همزمان رنگ آمیزی کرد.

برای این الگوریتم نیز می توان توابع اکتشافی مختلفی معرفی کرد. به عنوان مثال شماره دادن به هر رأس و رنگ کردن یک رأس که در بین همسایگانش بزرگترین شماره را دارد.

این بار چون الگوریتم به صورت توزیع شده است، برنامه برای یک رأس می نویسیم و تمام رئوس این برنامه را همزمان اجرا می کنند. از این رو مکانیزم هایی باید داشته باشیم تا هر رأس بداند تمامی رئوس مجاور مرحله جاری را به اتمام رسانده اند یا خیر. لذا یک نوع هماهنگ سازی نیز میان رئوس وجود دارد.

برنامه یک رأس به زبان ساده:

گام ۱- اگر round_over مقدار false داشت به گام ۲ می رود (round_over در ابتدای برنامه مقدار false می گیرد). اگر مقدار true داشت به گام ۷ می رود.

گام ۲- پیامی را دریافت می کند، اگر نوع پیام round بود به گام ۳ میرود. اگر از نوع color بود به گام ۴ می رود و اگر از نوع discard بود به گام ۵ می رود.

گام ۳- چک میکند اگر خودش رنگ نشده بود (colored!=true) و شماره اش از تمام گره های همسایه جاری بزرگتر بود (i>max(curr_neighs))، کوچکترین رنگ آزاد بین ۱ تا $\Delta+1$ را انتخاب میکند (c=min(free_cols)) و رنگ خود را به تمام همسایه های جاری ارسال می کند و همچنین خود را به

[۱۰] The Greedy Distributed Algorithm

عنوان رنگ شده علامت می زند(colored=true). اگر بزرگتر نبود، پیامی از نوع discard به تمام همسایه های جاری ارسال می کند. همچنین در انتهای این قسمت مقدار round_rcvd=true قرار میدهد.

گام ۴- شماره گره مبدا پیام دریافتی را به مجموعه received اضافه می کند و رنگ آن را از مجموعه رنگ های آزاد خود (free_cols) کم می کند و همچنین آن را در مجموعه گره های حذفی قرار می دهد (lost_neighs).

گام ۵- شماره گره مبدا پیام دریافتی را به مجموعه received اضافه می کند.

گام ۶- اگر این گره راند گرفته است (round_recvd==true) و همچنین تعداد پیام های دریافتی برابر تعداد همسایه های جاری است، آنگاه به این معناست که این راند به پایان رسیده و تمام پیام ها دریافت شده است، در نتیجه مقادیر round_over=true و round_recvd=false و همچنین همسایه های جاری را برابر مقدار قبلی منهای گره های حذفی (lost_neighs) قرار میدهد. همچنین مجموعه پیام های دریافتی (received) و مجموعه گره های حذفی (lost_neighs) را تهی می کند. به گام ۱ می رود.

گام ۷- پایان

هر چند به جای دریافت چندین و چندباره پیام discard از همسایه های با عدد کمتر با توجه به ساختار Pregel که راندها را همگام برگزار می کند، در پیاده سازی نیازی به این بخش نیست و توضیحات لازم در پیاده سازی الگوریتم با توجه به روش Pregel ارائه خواهد شد. اما در این بخش به دلیل اینکه روش Pregel توضیح داده نشده و همچنین در خصوص نظریه رنگ آمیزی گراف است، در همین حد بسنده می کنیم.

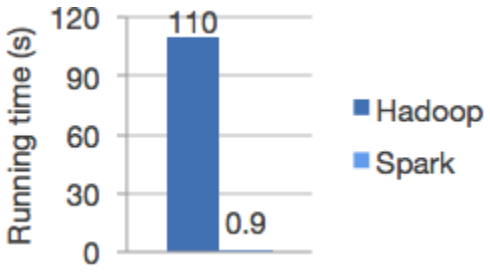
۳

فصل سوم

چارچوب Spark و Graphx

چارچوب[1] Spark و Graphx

چندین چارچوب مطرح در سیستم های توزیع شده توسعه داده شده اند که از آنها مـی تـوان چـارچوب های Hadoop، Spark، Flink و... نام برد که برخی از آنها خـاص منظـوره هسـتند امـا در ایـن میـان چارچوب Spark امروزه به عنوان یک ابزار کامل که بسیاری از امکانات ابزارهـای دیگـر را در خـود قـرار داده و حتی آنها را ارتقا داده است، مورد استفاده قرار می گیرد. مستندسازی کامـل و مطالـب آموزشـی بسیاری که برای Spark وجود دارد، میل به استفاده از این ابزار را دو چندان ساخته است. نمودار ۲ زمان اجرای دو چارچوب معروف Spark و Hadoop برای یک برنامه مشخص را مقایسه می کنـد و همـانطور که مشاهده می شود Spark عملکردی تا بیش از ۱۰۰ برابر سریعتر را دارد. این در حـالتی اسـت کـه از حافظه استفاده شده باشد، حتی در حالتی که از دیسک استفاده شـود نیـز Spark عملکـردی ۱۰ برابـر سریعتر را داراست [۴].



نمودار ۲ مقایسه زمان اجرای دو چارچوب Spark و Hadoop [۴]

در ارتباط کار با گراف ها گوگل سیستمی ارائه داده است به نام Pregel که امکان پردازش گراف هـا بـه صورت توزیع شده را به ما می دهد و البته این سیستم اختصاصی است. از این رو جامعه متن-بـاز[2] ابـزار معادل آن را به نام Giraph معرفی کرد. اما ابـزار Graphx در Spark ایـن ابـزار را بـه صـورت یکـی از امکانات در خود جای داده و آن را بهینه کرده است. لذا کار با Graphx علاوه بر داشتن مزایای Pregel امکانات بیشتری نیز خواهیم داشت.

[1] Framework

[2] Open Source

۱.۳ چارچوب Spark و ساختار آن[۴]

قبل از استفاده از چارچوب Spark لازم است ساختار آن و عملکرد آن را بهتر بشناسیم تا در مدل سازی، طراحی و پیاده سازی دید بهتری داشته باشیم. وظیفه Spark تولید ساختمان های داده توزیع شده[۱] ای است که مجموعه داده های ما را به مجموعه داده های توزیع شده ارتجاعی (RDD)[۲] می کند.

Spark برای اجرای یک برنامه قطعا یک سیستم مرکزی به نام درایور[۳] و تعداد دلخواهی سیستم اجراکننده[۴] تعریف می کند (می تواند سیستم اجرا کننده نداشته باشد). برنامه و داده ها در اختیار درایور است و درایور RDD را تقسیم بندی و به اجراکننده ها می دهد. همچنین در هر عملی که نیاز باشد کد یا دستوری برای اجرا بر روی مجموعه داده[۵] های RDD انجام شود دستور را به اجراکننده ها ارسال می کند و هر اجرا کننده بر روی مجموعه داده خود اعمال می کند. در نهایت درایور خروجی مورد نظر خود را از اجراکننده ها جمع آوری می کند.

بر اساس توضیحات فوق دو اصطلاح در Spark برای کار با RDD ها وجود دارد:

۱- تبدیل[۶]: عملیاتی که بر روی هر مجموعه داده در هر اجراکننده قابل اجرا است و این عمل در تمام اجراکننده ها به صورت موازی قابل اجراست

۲- عمل[۷]: عملیاتی که برای جمع آوری داده های خروجی از اجراکننده ها به سمت درایور است

این مقدمه به صورت مختصر سیستم Spark را توضیح داده است و در توضیح Graphx و پیاده سازی نرم افزار با جزئیات بیشتری آشنا می شویم.

---

۱ Resilient Distributed Datasets

۲ Partition

۳ Driver

۴ Executor

۵ Dataset

۶ Transformation

۷ Action

۲.۳ کاربرد Graphx

یکی از ابزارهای پرکاربرد Spark ابزار Graphx است که کار با RDD ها وقتی کـه RDD از نـوع یـک گراف است را آسان می کند و توابع و کلاس های مناسب این کار را در اختیار ما قرار می دهد و مـا را از پیاده سازی ویژگی های گراف معاف می سازد.

برخی از الگوریتم های مهم و پرکاربرد نظریه گراف به صورت پیشفرض در Graphx پیاده سـازی شـده است. اما الگوریتم رنگ آمیزی گراف به صورت پیش فرض وجود ندارد و باید پیـاده سـازی شـود. بـرای پیاده سازی الگوریتم رنگ آمیزی گراف به صورت توزیع شـده، از رابـط Pregel کـه در Graphx پیـاده سازی شده است استفاده می کنیم.

۱.۲.۳ عملکرد Pregel [۵]

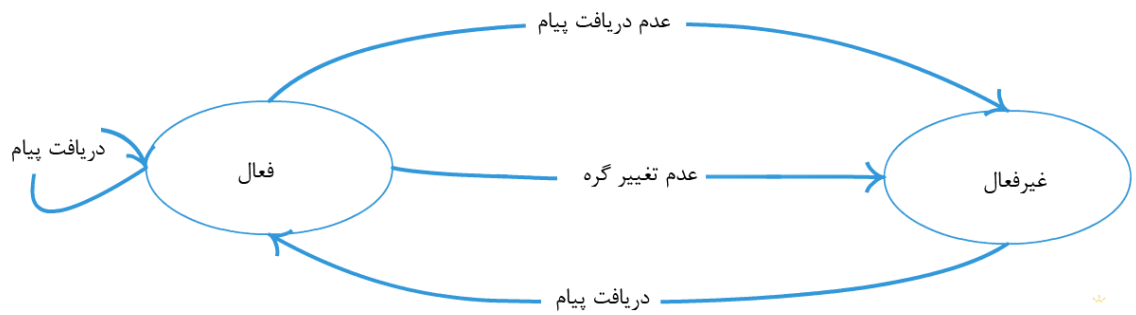
در نگاه کلی Pregel یک عملیات را چندین بار بر روی یک گراف انجام می دهد. اما برای کاهش میـزان پردازش راهکاری ارائه می دهد که باعث می شود رئوس یک گراف در دو حالـت فعـال و غیرفعـال قـرار گیرند.

ابتدا لازم است بدانیم Pregel بر مبنای رئوس یک گراف عمل می کند و پردازش ها بر روی رئوس انجام می شود و ما برای رئوس برنامه می نویسیم و یال ها هیچ برنامه ای برای پردازش ندارند.

در حالت ابتدایی تمامی رئوس فعال هستند. برنامه هر رأس بـه صـورت همزمـان بـا دیگـر رئـوس قابـل اجراست اما هر پیمایش[۱] در تمام گره ها هماهنگ است. یعنی وقتی یک پیمایش در تمـام رئـوس تمـام شد، پیمایش بعدی اجرا می شود. اگر تعداد پیمایش ها به حداکثر راند تعیین شده برسد یا تمامی رئوس غیرفعال شوند، پردازش به پایان رسیده است.

نحوه فعال/غیرفعال شدن یک رأس در گراف و در روش Pregel در نمودار ۳ ترسیم شده است.

[۱] Iteration



نمودار ۳ تغییر حالت یک رأس در روش Pregel

نکته مهم: ممکن است برنامه ما علی رغم اینکه درست به نظر می رسد اما عملکرد مطلوب را نداشته باشد که یکی از علل آن عدم توجه به مفهوم تغییر گره یا رأس در پیاده سازی Pregel در Graphx است. طبق پیاده سازی تغییر برای یک رأس یا گره، تغییر یک یا چند فیلد شیء رأس نیست بلکه باید شیء جدیدی ساخته شود و در چنین صورتی تغییر انجام شده است. علت آن این است که شرط برابری دو متغیر همان شرط برابری اشاره گر به نقطه حافظه است. لذا اگر شیء جدیدی ساخته نشود و شیء قبلی اما با تغییر فیلدهای آن همراه باشد، به عنوان تغییر رأس در نظر گرفته نمی شود و رأس به حالت غیرفعال می رود.

۳.۳ توزیع شدگی

هر چند توزیع شدگی را به معنی اجرای موازی بر روی چندین دستگاه کامپیوتر بدانیم، اما عملا نیازی به این کار نیست. چرا که ساختار Spark به صورت توزیع شده الگوریتم ها و عملیات را اجرا می کند و برنامه ای که بر بستر Spark نوشته شود، لزوما توزیع شدگی را خواهد داشت و اگر کد به گونه ای نوشته شود که این عمل را نقض کند با خطای هنگام اجرا رو برو خواهد شد.

در این پروژه قصد آزمایش چارچوب Spark را نداریم و لذا با توجه به هدف Spark و استفاده فراوان این چارچوب در سیستم های توزیع شده، این را به عنوان یک حقیقت می پذیریم و باید عملکرد برنامه نوشته شده بر این بستر صحیح باشد تا بپذیریم این برنامه به صورت توزیع شده نوشته شده است. هر

برنامه ای که با Spark نوشته شود هر چند به صورت محلی اجرا شـود، ماهیـت توزیـع شـده دارد و بـه صورت توزیع شده نیز قابلیت اجرا شدن را دارد.

در این پروژه پیاده سازی و آزمایشات به صورت محلی صورت گرفته است اما این تضـمین را بـه خـاطر ماهیت Spark می دهیم که بر روی چندین سیستم بدون هیچ تغییری قابل اجراست.

۴

فصل چهارم

طراحی نرم افزار

طراحی نرم افزار

پس از کسب دانش و پیش نیازهای لازم که عبارتند از تئوری رنگ آمیزی گراف ها و همچنین آشنایی با ابزار Graphx برای کار با گراف ها به صورت توزیع شده، اکنون می توانیم نرم افزار خود را طراحی و پیاده سازی کنیم. اما قبل از طراحی و پیاده سازی لازم است مدلی از نرم افزار یا سیستم خود را ارائه کنیم. از آنجایی که این نرم افزار از بخش های زیادی برخوردار نیست و بیشتر تعاملی با سیستم Spark است و Spark عمل های زیادی دارد اما از دید ما پنهان است میتوانیم به صورت یک جعبه سیاه[1] در نظر بگیریم.

همانطور که در مقدمه نیز آمده است، روندهای اجرایی ساخت این نرم افزار به صورت پی در پی یا روش آبشاری[2] است و پس از فصول قبل اکنون به مرحله مدلسازی سیستم (نرم افزار) رسیده ایم.

۱.۴ مدل UML نرم افزار

به خاطر بخش های مستقل و اندک این نرم افزار شاید صرف زمان و انرژی بر روی مدلسازی چندان منطقی نباشد و بیشتر زمان باید در طراحی و پیاده سازی صرف شود. با این حال در حد نیاز مدل هایی برای سیستم ارائه می شود. بخش زیادی از این نرم افزار توسط چارچوب Spark انجام می شود و بخش تعاملی، چگونگی پیاده سازی الگوریتم و نحوه کارکردن با این چارچوب مورد نظر بوده است.

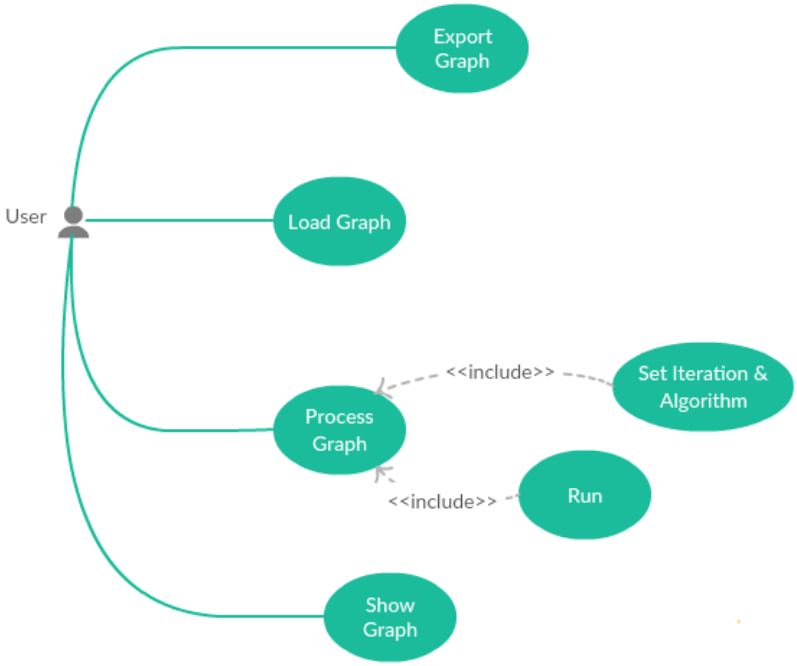
با این حال مدلسازی بخش هایی که در این نرم افزار پیاده سازی شده است و در تعامل با Spark است، ارائه می شود.

[1] Black box

[2] Sequential (waterfall)

۱.۱.۴ نمودار Use Case

این نرم افزار تک کاربره است و در نتیجه تنها یک کاربر به نام User داریـم. همچنـین در نمـودار Use Case تنها رابط کاربری مد نظر است و اینکه کاربر چه کنترلهایی بر روی نـرم افـزار خواهـد داشـت. در نتیجه می توان به صورت خلاصه عملیات خاص نرم افزار را در نمودار ۴ مشاهده کرد.



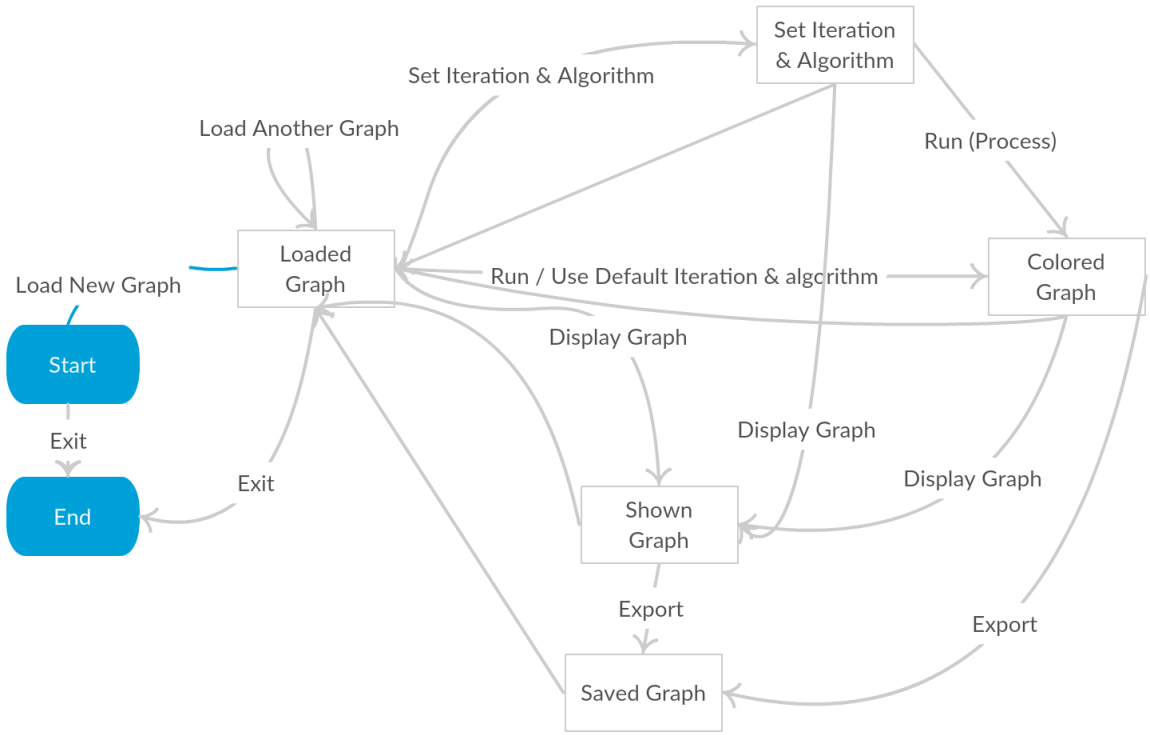
نمودار ۴ Use Case برای نرم افزار رنگ آمیزی گراف های بزرگ

۲.۱.۴ نمودار انتقال حالت (STD)[۱]

نمودار انتقال حالت عملکرد نرم افزار را از نقطه آغازین تا نقطه ای که بتوانیم از نرم افزار خروجی بگیریم را نمایش می دهد. این انتقال حالت، دقیقا در نرم افزار پیاده سازی می شود و این نمـودار حالـت حتـی می تواند یک راهنمای نرم افزار نیز تلقی کرد.

در نمودار ۵ حالات ممکن و عملیاتی که در این نرم افزار قابل انجام است را نمایش می دهد.

[۱] State Transition Diagram



نمودار ۵ انتقال حالت (STD) نرم افزار رنگ آمیزی گراف های بزرگ

در این نرم افزار بدون باز و بسته کردن نرم افزار می توان گراف ها و الگوریتم و مقادیر مختلف را تنظیم و اجرا کرد که اگر غیر از این بود نرم افزار با رابط گرافیکی کاربرپسندی نبود.

از این رو می توانیم از هر حالت به حالت دیگر برویم و دورهایی برای اجرای بینهایت برنامه وجود دارد. تنها حالتی که شرطی است، انتقال از حالت شروع به حالت گراف بارگذاری شده یا حالت پایان است زیرا تا گرافی لود نشده باشد، اجرای الگوریتم بر روی آن یا نمایش آن و یا ذخیره کردن آن معنایی ندارد و این موضوع در پیاده سازی نرم افزار باید مورد توجه قرار بگیرد.

۲.۴ پیاده سازی نرم افزار

هدف از ایجاد این نرم افزار ابزاری برای تعامل با چارچوب Spark و نوشتن الگوریتم یا الگوریتم هایی به صورت توزیع شده است و رابط کاربری برای کار با گراف ها و رنگ آمیزی گراف و مشاهده خروجی و برخی کنترل هایی که برای کار با چنین مسائلی مورد نیاز است. لذا هسته مرکزی این نرم افزار چارچوب Spark است.

چارچوب Spark در حال حاضر برای چهار زبان برنامه نویسی Scala, Java, R و Python منتشر شده است هر چند در زبان های R و Python به تمام امکانات دسترسی نداریم و در حال توسعه است اما Java و Scala کامل است. در اصل Scala نسخه پیشرفته تر Java است که امکاناتی از قبیل برنامه نویسی Functional را به آن اضافه کرده و هر چند قواعد نحوی اندکی متفاوت را دارد اما در نهایت به class. قابل اجرا توسط ماشین مجازی Java ایجاد می کند. همچنین کتابخانه های Java در زبان Scala قابل استفاده است.

تمامی این موارد ما را به سمت انتخاب زبان Scala متمایل می کند. یادگیری این زبان هم اندکی از وقت برنامه نویس را خواهد گرفت، اما در پیاده سازی به کمک او خواهد آمد.

۱.۲.۴ یادگیری پیش نیازها و مطالعه مستندات

در اولین گام قبل از پیاده سازی باید پیش نیازهای آن را مشخص نمود و جهت یادگیری و مطالعه آن اقدام کرد. در فصول قبل برخی از پیش نیازها گذرانده شد و از جمله پیش نیازهای دیگر این پروژه یادگیری زبان Scala و مطالعه مستندات Spark و Graphx و مطالعه نمونه هایی از برنامه های نوشته شده برای این زبان است. یادگیری Scala برای یک برنامه نویس مسلط به زبان Java چندان دشوار نیست و لازم است زمانی برای یادگیری آن اختصاص داده شود.

اما مطالعه مستندات Spark و Graphx از پیش نیازهای ضروری و زمانبر این پروژه بوده است. راه اندازی Spark و اجرای اولین برنامه بر طبق مستندات بوده و سپس با کلاس ها، توابع و گاها پیاده سازی آن باید آشنا شد.

در مطالعه Graphx علاوه بر کلاس ها و توابع پرکاربرد، یادگیری و فهم دقیق پیاده سازی Pregel نیز از اهمیت خاصی برخوردار بود که عدم درک صحیح باعث خطاهای غیرقابل دیباگ کردن در این برنامه می شد. همانطور که می دانید در برنامه های چندنخی[۱] خطایابی به شدت دشوار می شود. حال آنکه در این پروژه علاوه بر چندنخی بودن، توزیع شدگی نیز باید افزود. اما با دانستن نحوه کارکرد روش Pregel

[۱] Multithreading

که در بخش ۱.۲.۳ توضیح داده شد می توان بخش های با پتانسیل وجود اشتباه و خطا را بهتر کشف کرد.

لینک دسترسی به منابع اینترنتی Spark و Graphx در پیوست پ-۱ آورده شده است.

۲.۲.۴ راه اندازی Spark

تنها یک Spark بر روی هر سیستم اجرا می شود و در تنظیمات اولیه ساخت یک شیء SparkContext می توانیم سیستم Master را مشخص کنیم و اینکه برنامه به صورت توزیع شده بر روی چندین کامپیوتر یا به صورت توزیع شده بر روی یک کامپیوتر اجرا شود.

در این پروژه بر روی یک کامپیوتر آزمایشات را انجام داده ایم لذا تنظیمات لازم و نحوه تولید شیء SparkContext در جدول ۱ نشان داده شده است.

جدول ۱ کد تنظیم و ایجاد SparkContext برای اجرا بر روی یک کامپیوتر

```
val conf = new SparkConf().setAppName("AppName").setMaster("local")
val sc = new SparkContext(conf)
```

شیء sc در مواردی که نیاز به موازی سازی داده ها و ایجاد RDD است کاربرد دارد زیرا این شیء داده ها را بین سیستم ها توزیع می کند.

۳.۲.۴ بارگذاری گراف از روی دیسک

گراف ها را می توان به انواع فرمت ها ذخیره کرد. در این پروژه فرمت GEXF[۱] به عنوان یکی از فرمت های رایج توصیف گراف پیاده سازی شده است. همچنین Graphx کلاس GraphLoader را به صورت پیش فرض در خود دارد که نوعی دیگر از فرمت ذخیره سازی گراف را می تواند بخواند و به صورت یک

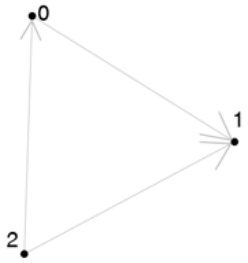
[۱] Graph Exchange XML Format

گراف خروجی دهد. در نتیجه نرم افزار حاضر از دو فرمت برای بارگذاری گراف استفاده می کند. هر چند تنها فرمت ذخیره سازی GEXF برای نوشتن بر روی دیسک استفاده می کنیم.

۱.۳.۲.۴ فرمت ذخیره سازی گراف GEXF

فرمت GEXF از ساختار XML برای توصیف گراف استفاده می کند. این ساختار قابلیت برچسب زدن بر روی رئوس و یالها را دارد و امکان تعریف انواع گراف ها را خواهد داشت. برای خواندن فایل XML از نوع رخداد-محور[۱] استفاده شده است که امکان خواندن فایل های بزرگ را با حافظه اندک و پردازش کمتر را در سریعترین زمان ممکن به ما می دهد هر چند پیاده سازی آن از روش دیگر (DOM) دشوارتر است.

نمونه ای از گراف ترسیم شده در شکل ۱ با فرمت GEXF در جدول ۲ توصیف شده است.



شکل ۱ نمونه گراف جهت دار

جدول ۲ نمونه توصیف گراف با فرمت GEXF

```
<?xml version="۱,۰" encoding="UTF-۸"?>
<gexf xmlns="http://www.gexf.net/۱,۲draft" version="۱,۲">
    <meta lastmodifieddate="۲۰۱۶-۰۷-۲۰">
        <creator>Vahid-PY</creator>
        <description>Sample Graph</description>
    </meta>
    <graph mode="static" defaultedgetype="directed">
        <nodes>
```

[۱] Event-Based

```
            <node id="۰" label="a" />
            <node id="۱" label="b" />
            <node id="۲" label="c" />
        </nodes>
        <edges>
            <edge id="۰" source="۰" target="۱" />
            <edge id="۱" source="۲" target="۱" />
            <edge id="۲" source="۲" target="۰" />
        </edges>
    </graph>
</gexf>
```

۲.۳.۲.۴ فرمت ذخیره سازی گراف با فرمت پیش فرض Graphx

ابزار Graphx از نوعی فرمت توصیف گراف استفاده می کند که امکانات زیادی در اختیار ما قرار نمی دهد اما از سادگی زیادی برخوردار است. با این فرمت ساختار گراف جهت دار را به راحتی می توانیم ترسیم کنیم اما قابلیت برچسب زدن بر رئوس و یال ها وجود ندارد. در این ساختار در هر سطر یک یال تعریف می شود و عدد اول شماره مبداء و عدد دوم شماره مقصد می باشد. همچنین رئوس به صورت اتوماتیک تولید می شود و نیاز به تعریف مجزا ندارد. این فرمت نیز در این نرم افزار تنها برای بارگذاری گنجانده شده اما برای ذخیره سازی در دسترس نیست و علت این کار عدم امکان برچسب زنی در این فرمت است که باعث می شود نتوانیم رنگ رئوس را ذخیره کنیم.

گراف ترسیم شده در شکل ۱ با این فرمت در جدول ۳ توصیف شده است.

جدول ۳ نمونه توصیف گراف با فرمت پیش فرض Graphx

```
#Comment
۰ ۱
۲ ۰
۲ ۱
```

۳.۴ نحوه کار با متد Pregel

مبانی عملکرد Pregel در بخش ۱.۲.۳ گفته شد. اما در این بخش می خواهیم پیاده سازی یک کد بر مبنای این متد را تشریح کنیم. ابتدا با تأمل در کد جدول ۴ می بینیم تابع Pregel چه پارامترهایی دارد.

جدول ۴ هدر تابع pregel

```
def pregel[A](initialMsg: A, maxIter: Int = Int.MaxValue, activeDir:
EdgeDirection = EdgeDirection.Out)
(vprog: (VertexId, VD, A) => VD, sendMsg: EdgeTriplet[VD, ED] =>
Iterator[(VertexId, A)],mergeMsg: (A, A) => A) : Graph[VD, ED]
```

این تابع بر روی یک گراف Graph[VD, ED] اجرا می شود و در خروجی نیز همین گراف را با تغییراتی می دهد. نوع پیام ها را با پارامتر نوع A مشخص می کنیم. یک پیام ابتدایی داریم که به تمام رئوس ارسال می شود و در اولین اجرا این پیام را خواهند داشت. تعداد راندهایی که پردازش انجام می شود را نیز مشخص می کنیم.

توضیحی که در خصوص activeDir باید داده شود آنکه از آنجایی که گراف ها جهت دار است و پیام ها در یال ها جا به جا می شود در این قسمت مشخص می کنیم که تابع sendMsg (که در ادامه توضیح داده می شود) بر روی چه یالهایی از یک رأس فعال اجرا شود. رأسی فعالی که در راند قبل پیام دریافت کرده، چندین یال ورودی و خروجی دارد. activeDir چندین مقدار دارد:

- EdgeDirection.out: تابع sendMsg بر روی یالهایی از رأس فعال اجرا می شود که از رأس فعال خارج شده باشد.
- EdgeDirection.in: تابع sendMsg بر روی یالهایی از رأس فعال اجرا می شود که به رأس فعال وارد شده باشند.
- EdgeDirection.Both: تابع sendMsg بر روی یالهایی از رأس فعال اجرا می شود که هم به آن رأس وارد شده باشند و هم خارج شده باشند (یال طوقه).
- EdgeDirection.Either: تابع sendMsg بر روی یالهایی از رأس فعال اجرا می شود که یا به آن رأس وارد شده باشند و یا خارج شده باشند و یا هردو.

از آنجایی که ما می خواهیم گراف های بدون جهت را پردازش کنیم، بهترین گزینه EdgeDirection.Either خواهد بود.

vprog برنامه ای خواهد بود که در هر پردازش بر روی یک رأس اجرا می شود. پیام و رأس را به عنوان ورودی داریم و به عنوان خروجی رأس تغییر یافته را باز می گردانیم. همانطور که قبلا توضیح داده شد در صورتی که شیء جدیدی ساخته نشود و تنها فیلدهای شیء رأس را تغییر دهیم و بازگردانیم، آن رأس غیرفعال می شود.

پس از پردازش بر روی هر یال می توانیم یک پیام ارسال کنیم. این پیام به چه طریقی ارسال شود و حاوی چه اطلاعاتی باشد توسط تابع sendMsg مشخص می‌گردد.

همچنین پیام های متعددی به یک رأس در گراف از سوی همسایه ها ارسال می شود ولی همانطور که قبلا در توضیح vprog گفته شد، تنها یک بار در هر پردازش بر روی یک رأس اجرا می شود و در ورودی یک پیام دریافت می کند. لذا مکانیزمی لازم است تا این پیام ها را ادغام کند و در یک پیام به رأس مقصد بدهد. این مکانیزم را در تابع mergeMsg مشخص می کنیم.

۱.۳.۴ پیاده سازی الگوریتم حریصانه توزیع شده با استفاده از Pregel

با تئوری این الگوریتم در فصول قبل آشنا شدیم و اکنون برای پیاده سازی طبق تابع Pregel باید یکسری تغییرات در الگوریتم وارد کنیم. به عنوان مثال اولین نکته ای که باید توجه داشت راندها در Pregel به صورت همگام اجرا می شود و نیازی به پیام های discard و... برای دانستن تمام شدن پردازش تمام رئوس نیست. از این رو به این پیامها نیازی نداریم.

در پیاده سازی این الگوریتم دوبار از تابع pregel استفاده شده است. در اولین استفاده که تنها یک راند دارد، همسایه ها به همدیگر معرفی می شوند و هر رأس شماره همسایه های خود را ذخیره می کند.

در نتیجه نوع پیام را یک مجموعه از شماره رئوس در نظر میگیریم. در sendMsg هر رأس را به همسایه آن معرفی می کنیم و در mergeMsg اجتماع مجموعه ها را محاسبه می کنیم و در نهایت در vprog این مجموعه را به همسایه هر رأس اضافه می کنیم و اگر شماره رأس از همسایه هایش بیشتر بود، در همین بخش آن را رنگ کرده و مقدار رنگی آن برابر ۱ خواهد بود. کد این موضوع را نشان می دهد.

در تابع دیگر pregel پیام ها از نوع لیستی از شماره رأس و شماره رنگ خواهد بود. تابع sendMsg به این صورت است که هر رأس تنها به همسایه هایی رنگ و شماره خود را می فرستد که شماره اش بزرگتر از آن ها باشد و همچنین رنگ شده باشد. این رأس پس از این راند غیرفعال خواهد بود و رئوس همسایه اطلاعات این رأس را تنها یکبار دریافت می کنند. تابع mergeMsg نیز این لیست ها را به همدیگر متصل می کند.

اما مهمترین بخش کار را تابع vprog انجام می دهد. در این تابع اگر رأس رنگ شده باشد، همان رأس بازگردانده می شود و درنتیجه رأس غیرفعال خواهد شد. اگر رنگ شده نباشد، در نتیجه بزرگترین در بین همسایگان نبوده آنوقت بررسی می کند اگر پیام دریافتی برابر پیام اولیه بود یعنی هیچ پیامی دریافت نکرده است و آن رأس غیرفعال می شود تا وقتی که پیامی دریافت کند. اگر پیام دریافتی پیام یکی از رئوس همسایه بود، یک شیء جدید از نمونه قبلی ساخته (تا رأس غیرفعال تلقی نشود و در راند بعد sendMsg برای آن اجرا شود تا بتواند رنگ خود را به همسایه ها ارسال کند) و سپس رنگ تمام پیام های دریافتی را از رنگ های آزاد خود حذف می کند (مقدار آن را در آرایه برابر بی نهایت قرار میدهد) و مقدار آن همسایه را در آرایه برابر منفی بینهایت قرار می‌دهد. سپس اگر رأس در بین همسایگان باقی مانده بزرگترین بود، کوچکترین رنگ موجود را انتخاب می کند.

کد این الگوریتم در پیوست پ-۲ درج شده است.

۴.۴ رابط کاربری گرافیکی (GUI)[۱] نرم افزار

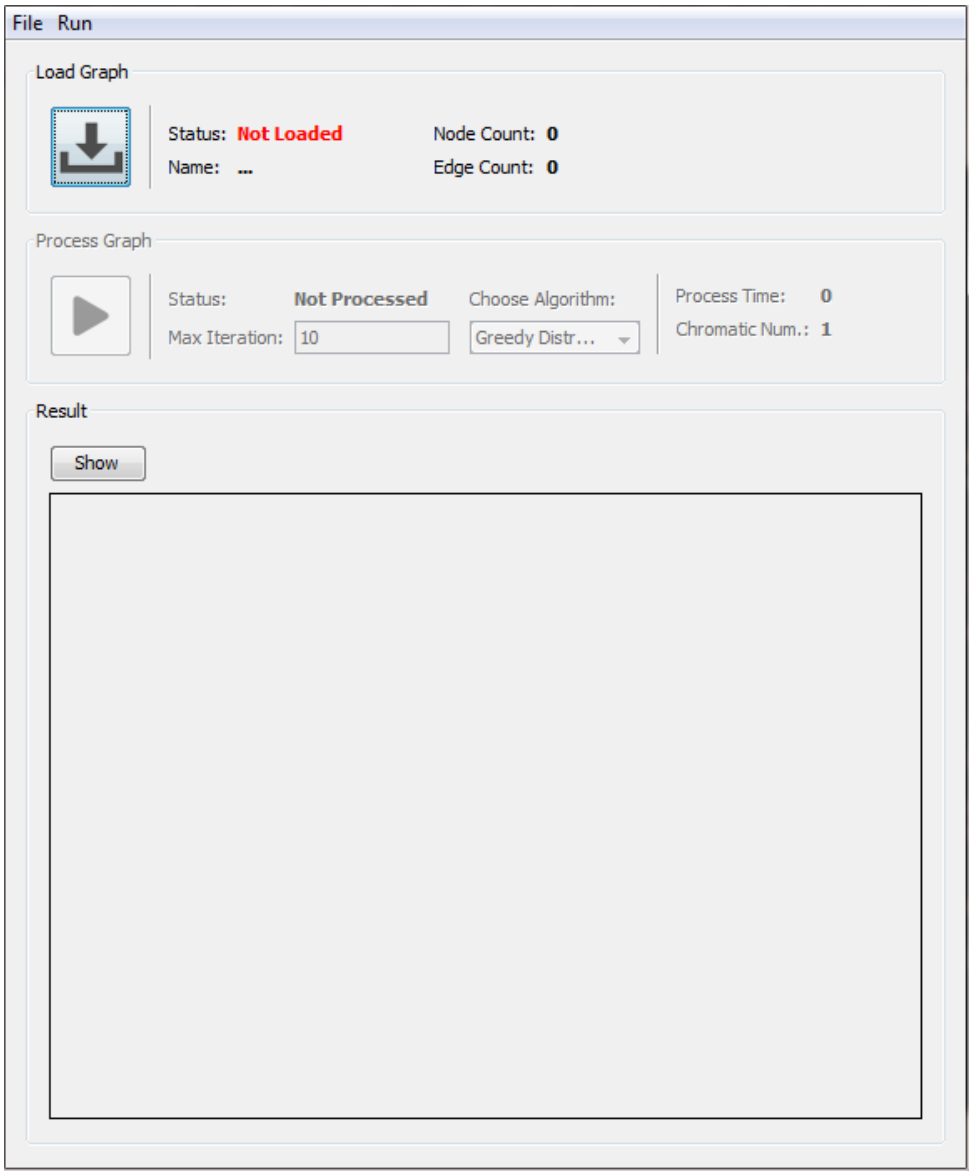
رابط کاربری گرافیکی، یک نرم افزار خشک و بی روح را به یک نرم افزار قابل استفاده و کاربرپسند تبدیل می کند. Scala از همان رابط کاربری گرافیکی Java استفاده می کند و همانطور که گفته شد کتابخانه های Java در Scala قابل استفاده است. رابط کاربری این نرم افزار چندان پیچیده نیست و با استفاده از Swing می توان نیازهای این نرم افزار را مرتفع کرد، هر چند JavaFX امکانات گرافیکی قویتری را در اختیار ما قرار می دهد. به علت استاندارد بودن Swing مشکلی در این بخش وجود ندارد. اما ترسیم گراف خروجی رنگ آمیزی شده نیازمند کتابخانه ای است که در انجام این کار یاری دهد، از بین

[۱] Graphical User Interface

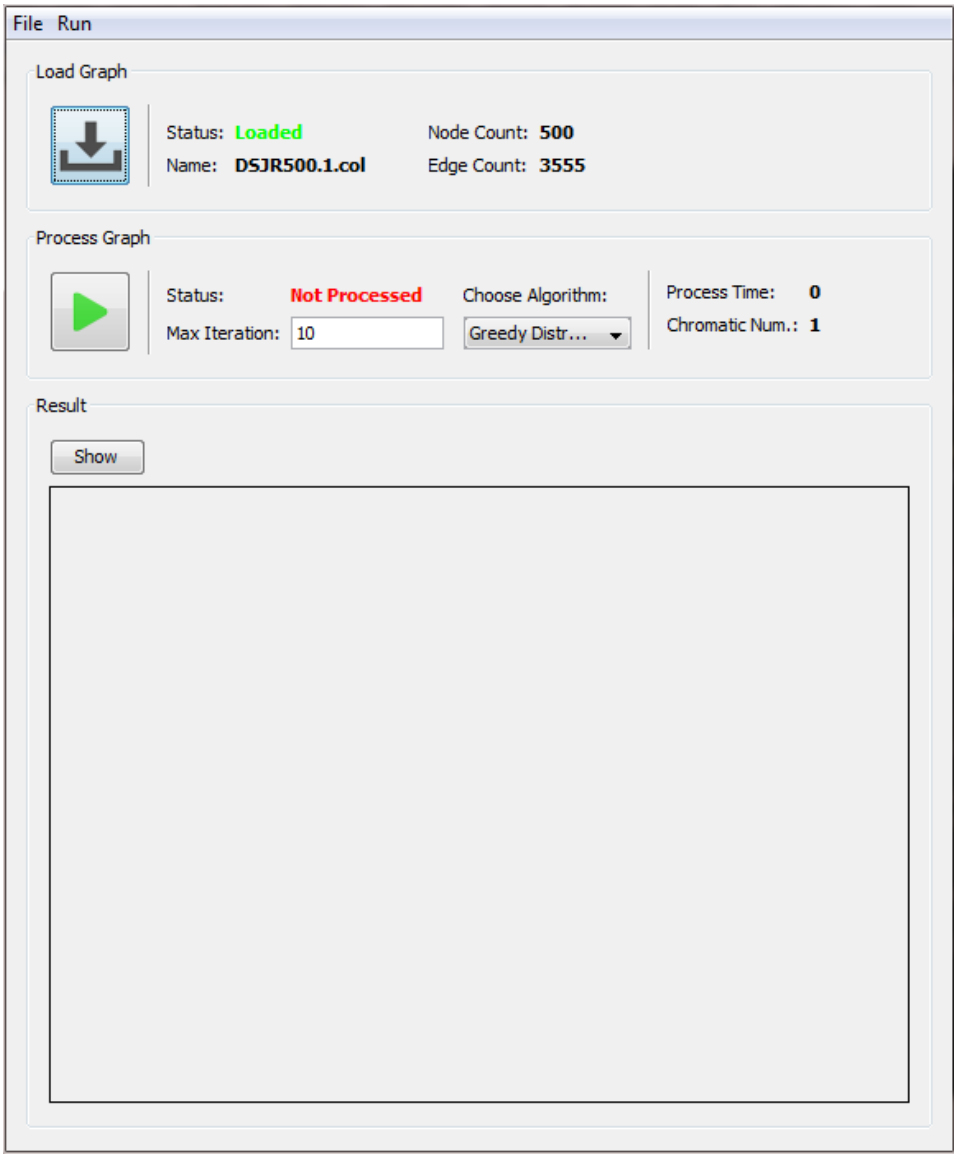
کتابخانه های مختلف GraphStream به دلیل سهولت استفاده و پویا بودن گراف انتخاب شده است، هر چند به دلیل نوپا بودن آن مشکلاتی در طراحی را دارد که با ترفندهایی برطرف گردید.

۱.۴.۴ محیط نرم افزار

در محیط کاربری این نرم افزار هر چیزی در دسترس است و کار با آن ساده خواهد بود. هر چند از طریق منوهای نرم افزار نیز همان عملیات قابل انجام است. همچنین بخش زیادی از ترتیب انجام عملیات که در نمودار انتقال حالت (STD) در بخش ۲.۱.۴ مشخص شده است در این رابط کاربری لحاظ گردیده است. به عنوان مثال تا وقتی گراف بارگذاری نشده است، بخش پردازش، ذخیره و نمایش گراف غیرفعال است. شکل ۲ محیط نرم افزار را وقتی که گراف بارگذاری نشده است و شکل ۳ محیط نرم افزار پس از بارگذاری گراف نشان می دهند.

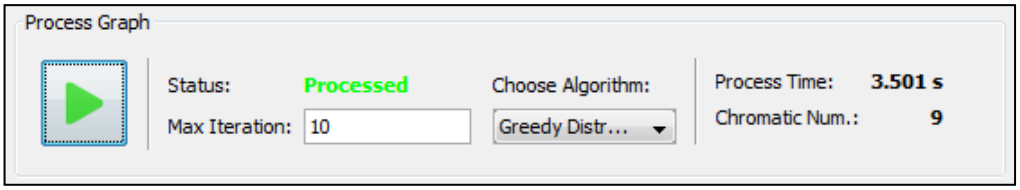


شکل ۲ محیط کاربری نرم افزار قبل از بارگذاری گراف



شکل ۳ محیط کاربری نرم افزار پس از بارگذاری گراف

پس از کلیک بر روی دکمه پردازش (یا انتخاب از طریق منو Run>Process) گراف طبـق Iteration و الگوریتم انتخاب شده اجرا می شود. پس از انجام پردازش، بخش Process به صورت شکل ۴ خواهد شد.

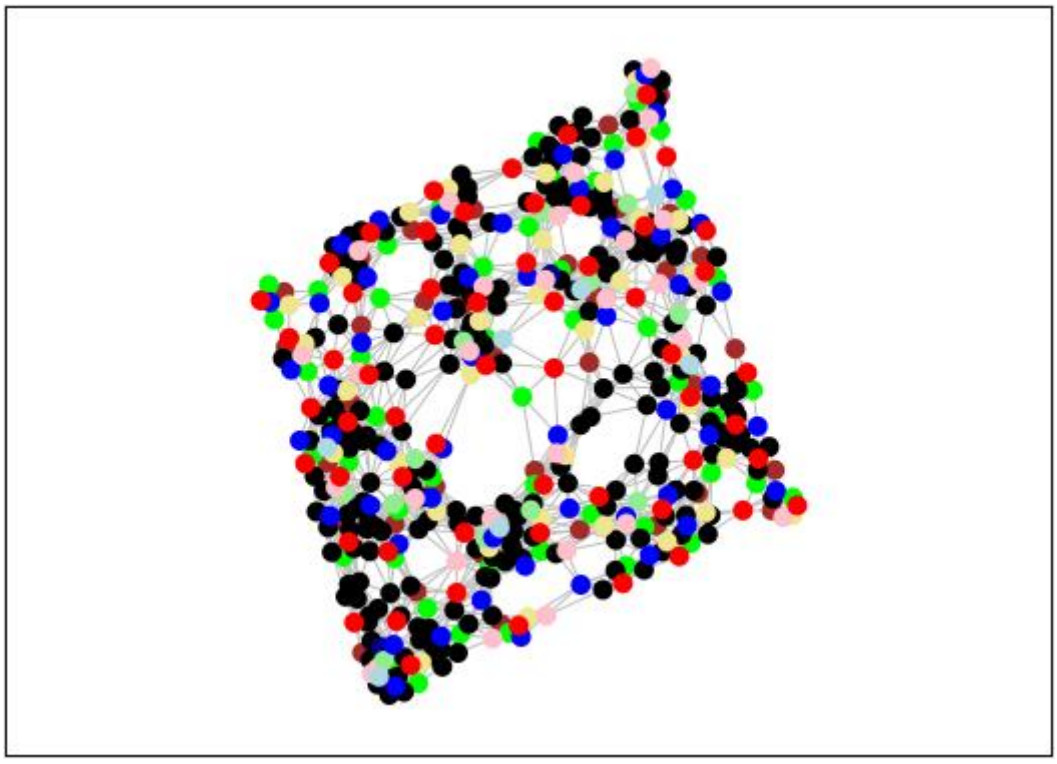


شکل ۴ قسمت Process محیط کاربری پس از پردازش گراف

در این قسمت، عدد رنگی (Chromatic Num.) و زمان پردازش برای گراف داده شده نمایش داده مـی شود. همچنین می توانیم با کلیک بر روی show گراف نهایی به صورت رنـگ آمیـزی شـده را مشـاهده کنیم. نمایش گراف را در بخش بعدی توضیح داده شده است.

۲.۴.۴ نمایش گراف به صورت گرافیکی با کتابخانه GraphStream

از میان کتابخانه های مختلف ترسیم گراف به صورت گرافیکی، با توجه به خروجـی هـای گرفتـه شـده، کتابخانه GraphStream انتخاب شد. این کتابخانه امکان ترسیم گراف به صورت پویا[۱] را به ما می دهد. این ابزار با Swing قابل ادغام است و می توان آن را در یک JPanel نمایش داد. خروجی گـراف بخـش قبل در شکل ۵ نمایش داده شده است.

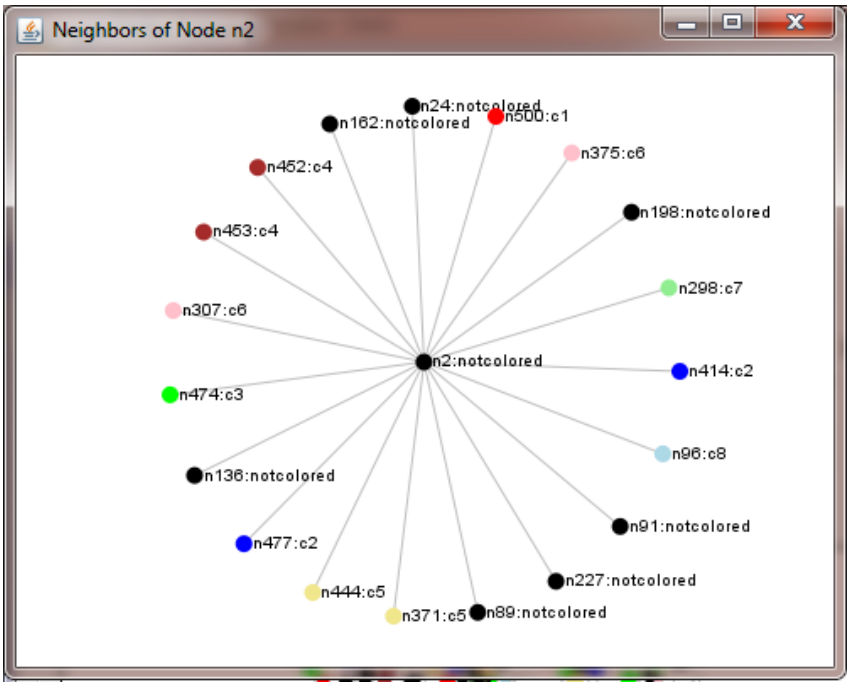


شکل ۵ نمایش گرافیکی گراف رنگ آمیزی شده

[۱] Dynamic

طبق این شکل تعداد زیادی رأس با رنگ مشکی مشاهده می شود. در اصل این رئوس رنگ آمیزی نشده اند. علت آن نیز تعداد راند (Iteration) است که برای رنگ آمیزی این گراف با الگوریتم حریصانه توزیع شده کفایت نمی کند. در فصل بعد این مسائل مورد بررسی قرار می گیرد.

اما از دیگر امکانات این نرم افزار دقیق شدن بر روی بخشی از گراف است. به عنوان مثال می خواهیم یک رأس با همسایه هایش مقایسه کنیم. بر روی یک رأس کلیک کرده و خروجی مشابه شکل ۶ مشاهده می کنیم. همانطور که در این شکل پیداست، شماره هر رأس با رنگ آن را مشاهده می کنیم.



شکل ۶ نمایش گرافیکی همسایگان یک رأس در گراف

یکی از مشکلاتی که در برنامه نویسی این کتابخانه وجود دارد، نحوه کلیک کردن بر روی یک رأس است. در این کتابخانه با کلیک بر روی هر رأس رخدادی اتفاق می افتد اما نمی توان تفاوت میان کشیدن[۱] و کلیک کردن را مشاهده کرد. از طرفی رخداد کلیک بر روی این شیء به صورت سراسری است. یعنی تمامی صفحه نمایش گراف یک JPanel است و کلیک بر هر نقطه ای از این صفحه یکسان خواهد بود.

[۱] Drag

برای رفع این موضوع از ترکیب این دو رخداد استفاده شده و کـد آن در کـلاس NodeClickListener قرار دارد. این کد در پیوست پ-۳ قرار گرفته‌است. لینک دریافت این کتابخانه نیز در پیوست پ-۴ قـرار دارد.

۵.۴ تحمل خطا[۱]

از آنجایی که این نرم افزار بر بستر Spark طراحی شده است و Spark این تضمین را به ما می دهد که در صورت شکست هر گره محاسباتی (اجراکننده ها) بتواند سیستم را پایدار نگهداری کند [۴]. در نتیجه در صورت حذف یک گره محاسباتی هنگام اجرای پردازش، اطلاعات آن گره در گره محاسـباتی دیگـری جایگذاری می شود و پردازش انجام خواهد شد. در نتیجـه بـا اسـتفاده از ایـن نـرم افـزار نگـران اشـتباه محاسباتی در حالت توزیع شده نخواهیم بود.

[۱] Fault Tolerance

۵

فصل پنجم

آزمایش و آنالیز نتایج با استفاده از نرم افزار پیاده سازی شده

آزمایش و آنالیز نتایج با استفاده از نرم افزار پیاده سازی شده

گراف های خاصی تعریف شده اند که برای بررسی الگوریتم های مختلف گراف از آن ها استفاده می شود. در این فصل نیز برخی از این گراف ها را با نرم افزار مورد بررسی قرار می دهیم و با مقدار ثبت شده توسط بنچمارک[۱] های گوگل در مسئله رنگ آمیزی گراف مقایسه می کنیم [۶].

۱.۵ مسائل بازه زمانی چند ثانیه

در جدول ۵ الگوریتم مورد بررسی، حریصانه توزیع شده است. اما بنچمارک بهترین عدد رنگی که تاکنون برای هر گراف یافت شده است را گزارش می دهد. طبق این بنچمارک گراف های مورد بررسی در زمان چند ثانیه قابل رنگ آمیزی است.

جدول ۵ مقایسه الگوریتم حریصانه توزیع شده با بنچمارک رنگ آمیزی گراف با زمان اجرا چند ثانیه

نام گراف	تعداد رئوس و یال‌ها	عدد رنگی بنچمارک	عدد رنگی (توسط نرم افزار)	زمان پردازش (تک سیستم) بر حسب ثانیه	تعداد راند برای رنگ شدن کامل گراف
۱-Fullns_۳	|V|=۱۰۰, |E|=۳۰۰	۴	۴	۱.۸۵	۱۲
anna	|V|=۱۳۸, |E|=۹۸۶	۱۱	۱۱	۵,۱۴	۱۶
۱-Insertions_۴	|V|=۶۷, |E|=۲۳۲	۵	۵	۲,۸۴	۱۱
DSJR۵۰۰,۱	|V|=۵۰۰, |E|=۳۵۵۵	۱۲	۱۴	۵,۲۲	۲۳

[۱] Benchmark

همانطور که در جدول ۵ مشاهده می شود، الگوریتم رنگ آمیزی حریصانه توزیع شده، توانست همان عدد رنگی بنچمارک را بدست آورد اما در مثال DSJR۵۰۰,۱ با اینکه تمام گراف را در ۲۳ راند رنگ آمیزی کرده است اما نتوانسته به بهترین نحو رنگ آمیزی کند. می توان این نتیجه گیری را انجام داد که الگوریتم حریصانه توزیع شده در بحث رنگ آمیزی گراف نمی تواند بهترین عدد رنگی را بیابد.

همچنین با بررسی گراف ها و الگوریتم، به این نتیجه خواهیم رسید که این الگوریتم در گراف های غیرمتراکم بسیار سریع عمل می کند زیرا بخش های مختلف گراف می توانند مستقل از هم رنگ آمیزی شوند اما در گراف های متراکم، به سری بودن رنگ آمیزی متمایل می شود و از رنگ آمیزی موازی کاسته می شود که باعث کندی اجرا رنگ آمیزی خواهد شد. این چالش ها در مسئله رنگ آمیزی گراف موجود است.

۲.۵ مسائل بازه زمانی دقیقه

در جدول ۶ الگوریتم مورد بررسی، حریصانه توزیع شده است. مطابق بخش قبل بنچمارک بهترین عدد رنگی که تاکنون برای هر گراف یافت شده است را گزارش می دهد. طبق این بنچمارک گراف های مورد بررسی در زمان چند دقیقه قابل رنگ آمیزی است.

جدول ۶ مقایسه الگوریتم حریصانه توزیع شده با بنچمارک رنگ آمیزی گراف با زمان پردازش چند دقیقه

نام گراف	تعداد رئوس و یال‌ها	عدد رنگی بنچمارک	عدد رنگی (توسط نرم افزار)	زمان پردازش (تک سیستم) بر حسب ثانیه	تعداد راند برای رنگ شدن کامل گراف
۱-Fullns_۴	|V|=۹۳, |E|=۵۹۳	۵	۵	۲,۹۶	۱۹
۴-Insertions_۳	|V|=۷۹,	۴	۴	۴,۶۳	۱٤

	|E|=۱۵۶				
le۴۵۰_۱۵a	|V|=۴۵۰, |E|=۸۱۶۸	۱۵	۲۲	۵۶	~۱۰۰

با توجه به این آزمایش و جدول مشاهده می کنیم در برخی مثال ها با افزایش چگالی و درجات برخی رئوس زمان اجرا به شدت افزایش می یابد و الگوریتم حریصانه توزیع شده نمی تواند عملکرد چندان مناسبی داشته باشد. هر چند در برخی از این گراف ها که بنچمارک زمان چند دقیقه ای را برای رنگ آمیزی آن ثبت کرده است، این الگوریتم در عرض چند ثانیه به جواب رسیده است که از این نظر برای چنین گراف هایی این الگوریتم از نظر زمانی مناسب است.

یکی از مشکلات الگوریتم حریصانه توزیع شده که از الگوریتم حریصانه متوالی به ارث برده است نحوه شماره دهی به گره ها در رنگ آمیزی، زمان اجرا و نهایتا در عدد رنگی به عنوان نتیجه نهایی موثر خواهد بود.

در این جدول نیز مشاهده می کنیم با اینکه تعداد راندهای اجرا الگوریتم بالاست و زمان اجرا نیز بالاست، با این وجود تعداد رنگ های یافت شده توسط این الگوریتم از مقدار بهینه یافت شده توسط بنچمارک به مقدار زیادی فاصله دارد.

۶

فصل ششم

جمع‌بندی و نتیجه‌گیری

جمع‌بندی و نتیجه‌گیری

در این پروژه تلفیقی از کار تحقیقاتی و تئوری صورت گرفت و در نهایت به صورت یک نرم افزار قابل استفاده درآمد. در ابتدا تئوری هایی از گراف و رنگ آمیزی گراف بررسی شد و با آن آشنا شدیم، سپس با ابزار متداول و پرکاربرد Spark که یکی از ابزارهای پراستفاده در صنعت کنونی کامپیوتر و داده های حجیم است آشنا شدیم و در نهایت به جایی رسید که بتوانیم از آن استفاده کنیم.

این چارچوب در مقاطع کارشناسی ارشد به مراتب مورد استفاده قرار می گیرد و پروژه های بزرگ کامپیوتری به این ابزار و ابزارهای مشابه نیازمندند زیرا حجم داده ها و پردازش ها فراتر از توان یک کامپیوتر معمولی است. همچنین در سیستمی که کاربران زیادی از یک سرویس استفاده می کنند سرعت پاسخگویی بسیار پراهمیت می شود و نیاز داریم درخواست کاربران را بین سیستم های مختلف تقسیم کنیم بدون اینکه در پاسخ آن اشکالی ایجاد شود. برای درک بهتر تصور کنید سیستمی همچون گوگل یا فیس بوک یا واتس اپ که میلیون ها درخواست را در ثانیه دریافت می کنند و در حجم داده های عظیمی پردازش انجام می دهند و در کسری از ثانیه پاسخ مناسب را به کاربر می دهند. تاکنون معماری دیگری جز معماری سیستم های شبکه شده توزیع شده جوابگو چنین کاربردی نبوده است.

امروزه فضاهای ابری[1] نیز اضافه شده اند که حجم داده ها و پردازش ها به مراتب افزایش خواهد یافت. البته از ایده های موجود استفاده از سیستم های سرویس گیرنده به عنوان سرویس دهنده نیز وجود دارد که تشکیل یک سیستم توزیع شده غول پیکر و راه دور را خواهد داد که البته چالش های جدیدی نیز ایجاد خواهد کرد. اما در حال حاضر به عنوان یک امکان مطرح شده است.

امروزه در کشور ایران در زمینه پردازش های توزیع شده نیز نیازمندی های جدی به وجود آمده است. برای مثال بسیاری از سیستم های دولت الکترونیک، بانک ها و سازمان های بزرگ یا اپلیکیشن های پرمخاطب حجم داده های ارسالی/دریافتی و پردازش ها و درخواست های زیادی را دارند و یکی از مشکلاتی که در این زمینه موجب کیفیت پایین شده عدم استفاده از توزیع شدگی است و کاربرها با کندی خدمات روبرو هستند. اما می توان با افزایش اندکی توان سخت افزاری و استفاده از چارچوب هایی

[1] Cloud

همچون Spark و مانند آن کیفیت خدمات را افزایش دهند. هر چند در حال حاضر نیز تلاش هایی در این زمینه در حال انجام است، اما با افزایش کارهای تحقیقاتی و پروژه های دانشگاهی در این زمینه افراد کارشناس در این زمینه افزایش خواهد یافت و خواهند توانست مسائل موجود در صنعت کامپیوتر کشور را در پروژه های دانشگاهی مطرح و حل نمایند.

۱.۶ کارهای آینده با این چارچوب و نرم افزار

این نرم افزار چندین ابزار را کنار هم فراهم آورده است. رابطی ساده برای کار با چارچوب Spark و Graphx که که می تواند جهت استفاده در کارهای تحقیقاتی مورد استفاده قرار گیرد. همچنین با تصویرسازی گراف به صورت پویا، امکان نمایش گراف های اندازه متوسط (کمتر از میلیون یال) وجود دارد که جهت درک نتیجه پردازش به وضوح قابل نمایش است. به راحتی می توانیم یالها و گره ها را رنگ بندی و برچسب زنی کنیم.

با توجه به رعایت نحوه استفاده از pregel به راحتی میتوان انواع الگوریتم های قابل تعریف بر گراف را پیاده سازی کرد و با اندک تغییری در برنامه الگوریتم های دیگری را بر روی گراف به صورت توزیع شده و بر بستر Spark اجرا کنیم.

بهتر است بگوییم این نرم افزار ابزاری برای استفاده از تابع pregel و استفاده از چارچوب Spark است و نمایش گرافیکی آن است و رنگ آمیزی گراف به صورت توزیع شده تنها یکی از کاربردهای آن بوده که در این پروژه پیاده سازی شده است.

توصیه می شود در پروژه های دیگر بر روی بهبود این نرم افزار کار شود. به عنوان مثال نرم افزار را به گونه ای تغییر دهند که بدون تغییر کد بتوان انواع الگوریتم ها بر روی گراف اجرا کرد و امکان تعریف انواع الگوریتم ها وجود داشته باشد. اگر این امر عملی شود یک کارگاه کار با گراف با ابزارهای مفید برای کاربران تهیه شده است.

از موارد دیگری که پیرامون این پروژه می توان انجام داد، نحوه نمایش گراف به صورت گرافیکی است. در این پروژه کل گراف به یک باره بر روی صفحه رسم می شود که برای گراف های خیلی بزرگ ممکن

است برنامه را به بن بست برساند. در نتیجه باید بخش هایی از گراف را به صورت هوشمندی انتخاب کند و نمایش دهد و با پیمایش در بخش های گراف، قسمت های جدید آن نمایش داده شود.

منابع و مراجع

[۱] R. Karp, "Probabilistic recurrence relations," *Proc ۲۳rd annual ACM symposium on theory of computing*, pp. ۱۹۰-۱۹۷, ۱۹۹۱.

[۲] B. Bollobas, Graph theory, New York: Springer, ۱۹۷۹.

[۳] K. Erciyes, Distributed Graph Algorithms for Computer Networks, London: Springer, ۲۰۱۳.

[۴] Apache, "Apache Spark," Jul ۲۰۱۶. [Online]. Available: http://spark.apache.org/.

[۵] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *the ۲۰۱۰ ACM SIGMOD International Conference on Management of data*, ۲۰۱۰.

[۶] S. Gualandi, "Graph Coloring Benchmarks," Google, [Online]. Available: https://sites.google.com/site/graphcoloring/home.

پیوست‌ها

پ-۱. مستندات Spark و Graphx به ترتیب در آدرس های اینترنتی http://spark.apache.org/documentation.html و http://spark.apache.org/docs/latest/graphx-programming-guide.html در دسترس هستند.

پ-۲. پیاده سازی الگوریتم حریصانه توزیع شده با تابع pregel

جدول پ-۱. پیاده سازی الگوریتم حریصانه توزیع شده با تابع pregel

```
def process(graph: Graph[MyVertex, Int], maxIteration:Int):
Graph[MyVertex, Int] = {

MAX_ITERATION=maxIteration

if(MAX_ITERATION<١) return graph;

var neg = graph.pregel[Set[VertexId]](Set(-١), ١,
EdgeDirection.Either)((id, vData, msg) => {

    val x = new MyVertex().addAllNeighbors(msg)

    if (id > x.maxCurrNeighbor) x.colored = true

    return x

}, edge => {

 Iterator((edge.srcId, Set(edge.dstId)), (edge.dstId, Set(edge.srcId)))

}, (a, b) => {

 a ++ b

})

if(MAX_ITERATION<٢) return neg;

neg = neg.pregel[List[(VertexId, Int)]](List((-١L,-١)), MAX_ITERATION -
١, EdgeDirection.Either)(vprog, sendMsg, mergeMsg)

return neg

}
```

```
def sendMsg(edge: EdgeTriplet[MyVertex, Int]): Iterator[(VertexId,
List[(VertexId, Int)])] = {

      if(edge.dstId>edge.srcId) {

            if(edge.dstAttr.colored && !edge.srcAttr.colored)
Iterator((edge.srcId,List((edge.dstId,edge.dstAttr.color))))

            else Iterator.empty

      }else {

            if(edge.srcAttr.colored && !edge.dstAttr.colored)
Iterator((edge.dstId,List((edge.srcId,edge.srcAttr.color))))

            else Iterator.empty

      }

}


def mergeMsg(a: List[(VertexId, Int)], b: List[(VertexId, Int)]):
List[(VertexId, Int)] = {

      a ::: b

}


def vprog(vId: VertexId, vData: MyVertex, msg: List[(VertexId, Int)]):
MyVertex = {

      if (vData.colored) return vData

      if(msg(۰)._۱== -۱ || msg(۰)._۲== -۱) {

            return vData

      }

      val nvData=vData.clone

      msg.foreach(tuple => {

            val indexColor = nvData.free_cols.indexOf(tuple._۲)

            if (indexColor != -۱) {

                  nvData.free_cols(indexColor) = Int.MaxValue

            }

            val indexNeighbor = nvData.neighbors.indexOf(tuple._۱)

            if (indexNeighbor != -۱) {

                  nvData.neighbors(indexNeighbor) = Int.MinValue
```

```
            }
      })
      if (vId > nvData.maxCurrNeighbor) {
            nvData.color = vData.minFreeColor
            nvData.colored = true
      }
      return nvData
}
```

پ-۳.

جدول پ-۲. کد NodeClickListner برای یافتن رخداد کلیک بر روی رئوس در کتابخانه GraphStream

```
class NodeClickListener(pGraph: Graph, pViewer: Viewer, mGraph:
org.apache.spark.graphx.Graph[MyVertex, Int]) extends ViewerListener
with MouseListener {
      val graph = pGraph
      val viewer = pViewer
      val mainGraph = mGraph
      val fromViewer = viewer.newViewerPipe();
      fromViewer.addViewerListener(this);
      fromViewer.addSink(graph);
      var nodeRelease: Boolean = false
      var nodeID: String = ""

      override def buttonReleased(id: String) {
            nodeRelease = true
            nodeID = id
      }
      override def mouseClicked(arg۰: MouseEvent) {
            if (nodeRelease) {
                  showInformation(nodeID)
```

```
            }
      }
      override def mouseReleased(arg٠: MouseEvent) {
            nodeRelease = false
            fromViewer.pump()
      }
      def showInformation(sid: String): Unit = {
            new InfoWin(sid,graph).setVisible(true)
      }

      override def viewClosed(viewName: String) {}
      override def buttonPushed(id: String) {}
      override def mouseEntered(arg٠: MouseEvent) {}
      override def mouseExited(arg٠: MouseEvent) {}
      override def mousePressed(arg٠: MouseEvent) {}
}
```

پ-۴. لینک دسترسی به کتابخانه GraphStream: http://graphstream-project.org