چکیده

هدف از این پروژه یافتن اجتماع ها در یک گراف . مقایسه ی عملکرد الگوریتم ها از لحاظ زمانی و حافظه ای برا دو ساختمان داده ی ماتریس مجاورت و لیست مجاورت و یافتن عدد n مناسب برای بهینه ترین حالت optimum sort. در این پروژه به دلیل حجم زیاد داده ها و محدودیت های سخت افزاری از جمله حافظه ی RAM برای پیاده سازی ماتریس مجاورت از ماتریس های خلوت استفاده شده است. همچنین پیاده سازی این پروژه با زبان جاوا بوده است.

مقدمه

تشخیص ساختارهای اجتماعاتی یک موضوع مهم در بسیاری از زمینهها میباشد. این موضوع با مفاهیمی مانند شبکههای اجتماعی)روابط بین اعضا(، تحقیقات بیولوژیکی یا مسایل تکنولوژیکی) بهینه سازی زیرساختهای حجیم(در ارتباط میباشد.الگوریتمهای متنوعی برای یافتن ساختارهای انجمنی موجود میباشد

محققان رشته های مختلف روش های متفاوتی را ارایه دادند و رویکرد های متفاوتی نسبت به ان داشته اند ولی کلیت بیشتر این روش ها پیدا کردن زیر گراف هایی است که ارتباط درونی زیاد و ارتباط بیرونی کمی دارند.الگوریتم های خوشه بندی گراف به ۳ دسته ی روش های مبتنی بر فاصله . روش های سلسله مراتبی و روش های پیمانگی یا ماژولاریتی تقسیم میشود. یکی از این الگوریتم های برا خوشه بندی گراف ها که در دسته ی ماژولاریتی قرار میگیرد نیومن-گیروان است که بسیار هزینه بر است و برای تعداد بسیار زیاد داده ناکارا می باشد. الگوریتم استفاده شده در این پروژه بر پایه ی یک مفهوم به نام ضریب خوشه ای یال پایه گذاری شده است که رابطه ریاضی این کمیت در زیر آماده است.

$$C_{ij} = \frac{z_{ij} + 1}{\min[k_i - 1, k_j - 1]}$$

شرح الگوریتم ها و ساختمان داده ها

این پروژه برای پیاده سازی به ۳ قسمت اصلی تقسیم میشود.
قسمت اول:پیاده سازی ساختمان داده های مورد نیاز برای ذخیره سازی گراف
قسمت دوم: نحوه ی پردازش اطلاعات ورودی از فایل و ذخیزه ی ان به صورت ساختمان داده های پیشنهاد شده.
قسمت سوم : پیاده سازی sort های مورد نیاز برای پروژه
قسمت چهارم :پیاده سازی الگوریتم های مورد نیاز برای خوشه بندی و تشخیص اجتماعات گراف

- قسمت اول

ماهیت هر دو ساختمان داده ی نام برده شده یکسان است و عملیات های مختلفی را قرار است بر روی گراف مورد نظر اجرا کنند برای همین یک کلاس parent به نام Graph در نظر میگیریم که دارای ویژگی های مشترک هردو ساختمان داده باشد و این کلاس ساختمان داده های ما از ان ارث میبرند.

گراف (Graph)

این کلاس یک کلاس انتزاعی(abstract) است که فقط ویژگی های مشترک همه ی گراف ها را دارد.

ماتریس مجاورت (GraphSparse)

به دلیل حجم زیاد داده ها ورودی و محدودیت سخت افزار از جمله حافظه RAM برای پیاده سازی این قسمت از ماتریس های خلوت استفاده شده است. به اینصورت که به جای ذخیره سازی تمام ماتریس دو بعدی مجاورت تنها یک های موجود در ان ذخیره شده است.

برای اینکار از کلاسی به نام Edge استفاده کردیم که دارای دو راس(Vertex) و یک متغیر صحیح برای هزینه (ضریب خوشه ای یال) است.

✓ توابع پیاده سازی شده:

DFS :

این متد بوسیله ی یک stack کمکی پیاده سازی شده به این صورت که ابتدا با اولین راس فراخوانی میشود و سپس راس های مجاور ان داخل استک pop میشوند و همین روند تا زمان خالی شدن استک ادامه دارد.
برای پیدا کردن راس مجاور با راس مشخص v باید داخل ارایه یا لیست اسپارس پیمایش انجام بدهیم و رئوس مجاور با v را بیابیم.
در بدترین حالت برای این الگوریتم ما باید به ازای هر راس تمام لیست رو پیمایش کنیم که هزینه ی ان O(VE) میشود.

CountCycle :

متدی برای شمارش دور هایی به طول ۳ که بر روی دو راس مشخص v1 و v2 بنا شده است.
برای پیدا کردن به دور۳ میتوان راس های مشترکی را که دو راس v1 , v2 با ان ها مجاور هستن را بشماریم.
برای این کار با یک پیمایش ابتدا راس های مجاور با هر کدام از راس های v1 و v2 را در یک لیست ذخیره میکنیم و سپس با مقایسه ی دولیست رئوس مشترک را میشماریم و این عدد برابر با تعداد مثلث ها یا دور هایی به طول ۳ است.
پیچیدگی هزینه ی این عملیات برابر با O(E + n1*n2) است که E تعداد یال و n1, n2 برابر با درجه رئوس دو راس v1, v2 است.

DegreeOfVertex:

این متد برای گرفتن درجه ی راس خاص مثل v است که هزینه ی ان O(1) است چون ما در هنگام خوندن فایل درجه ی هر راس را ذخیره می شود و هنگام حذف هم اپدیت می شود.

DeleteEdge :

متدی برای حذف یال میان دو راس مشخص v1, v2 برای انجام این علیات ما یک پیمایش روی لیست اسپارس خود انجام میدهیم تا شماره ی یال را پیدا کنیم و سپس ان را حذ ف میکنیم.

هزینه ی این عملیات O(E) است.

FetchCostOfEdge :

متدی برای حساب کردن هزینه ی همه ی یال ها که طبق فرمول محاسبه ضریب خوشه ای یال بدست می اید و هزینه ی این عملیات O(E + V^2) است.

لیست مجاورت (GraphList)

برای پیاده سازی این ساختمان داده از یک ارایه به اندازه ی تعداد راس های گراف استفاده می شود که هر خانه ی ان یک گره است که دارای یک قسمت اطلاعات و یک قسمت لینک به گره بعدی است.

قسمت اطلاعات ان یک راس(Vertex) است.

✓ توابع پیاده سازی شده:

DFS :

این متد بوسیله ی یک stack کمکی پیاده سازی شده به این صورت که ابتدا با اولین راس فراخوانی میشود و سپس راس های مجاور ان داخل استک pop میشوند و همین روند تا زمان خالی شدن استک ادامه دارد.
برای پیدا کردن راس های مجاور با راس مشخص V به دلیل داشتن دسرسی تصادفی (random_acces) میتوانیم به هزینه ی ۱ به راس مجاور دسرسی پیدا کنیم.
پس هزینه ی این عملیات O(E + V) است.

CountCycle :

برای داشتن تعداد دور ها به طول ۳ برای دو راس v1, v2 کافیست راس سومی وجود داشته باشد تا با هردوی این دو راس مجاورت باشد پس برای چک کردن این مجاورت کافیت راس های مجاور دو راس v1 ,v2 را بررسی کنیم.
اگر فرض کنیم درجه ی این دو راس برابر با n1, n2 باشد هزینه ی عملیات ما O(n1 * n2) خواهد بود.

DegreeOfVertex:

این متد برای گرفتن درجه ی راس خاص مثل v است که هزینه ی ان O(1) است چون ما در هنگام خوندن فایل درجه ی هر راس را ذخیره می شود و هنگام حذف هم اپدیت می شود.

DeleteEdge :

برای حذف کردن یال میان دو راس v1,v2 باید از لیست پیوندی v1 گره v2 را حذف کنیم و بالعکس.

در بدترین حالت هزینه عملیات این متد O(n1 + n2) خواهد بود زیرا گره v1 دارای n1 گره مجاور است و در بدترین حالت باید تمام ان ها پیمایش شوند برای v2 هم همینطور است.

FetchCostOfEdge :
متدی برای حساب کردن هزینه ی همه ی یال ها که طبق فرمول محاسبه ضریب خوشه ای یال بدست می اید و هزینه ی این عملیات O(E + V^2) است.

الگوریتم تشخیص تقسیم گراف :

برای تشخیص وجود ناهمبندی در یک گراف میتوان از الگوریتم های متفاوتی استفاده کرد که ساده ترین ان ها DFS است.برای اینکار کافیست ما این متد را از راس ابتدایی فراخوانی کنیم و در پایان متد ارایه ی visited را که برای برچسب گذاری راس ها استفاده میشد پیمایش کنیم.
در صورتی که تمامی خانه ها مقدارشان True باشد یعنی تمامی رئوس به طریقی از راس ابتدایی قابل دسرسی و مشاهده بوده اند پس گراف همبند است اما اگر حداقل یک خانه مقدارش False باشد به ای معنا است که از طریق راس ابتدایی و دیگر رئوس مجاور با ان قایل مشاهده و دسرسی نبوده است پس گراف ما ناهمبند است.
هزینه ی زمانی این الگوریتم هم دقیقا مشابه الگوریتم DFS است.

الگوریتم تشخیص اجتماع در گراف:

پیاده سازی این الگوریتم دارای 4 قسمت است.

قسمت اول : یک حلقه که همبندی گراف را بررسی میکند و درصورت ناهمبند شدن از ان خارج شده و

نتیجه را اعلام میکند.(DFS)

قسمت دوم: محاسبه ی هزینه های تمامی یال ها(FetchCostOfEdge)

قسمت سوم: مرتب سازی لیست هزینه های هر یال(ضریب خوشه ای یال) بوسیله ی sort های مشخص شده.

قسمت چهارم: حذف یال با کمترین هزینه(DeleteEdge)

مشخصات سخت افزاری :

	Mine	Friend
CPU Model	Intel-core i7-6500U @ 2.5Ghz	
CPU Physical Core	4	
CPU Virtual Core	2	
CPU L1 Cash	64kib	
CPU L2 Cash	512kib	
CPUL3 Cash	4Mib	
RAM Model	SODIMM synchronous 2133 MHz (.5 ns)	
RAM Capacity (GB)	12GB	
RAM Bus	_	
HDD/SSD Write Speed	_	
HDD/SSD Read Speed	_	
OS	Linux – ubuntu – 64bit	

معرفی داده های پروژه :

Name	Vertex Number	Edge Number	Average Vertex Degree
Test1	10000	77048	7
Test2	10000	126903	12
Test3	50000	382559	7
Test4	50000	634553	12

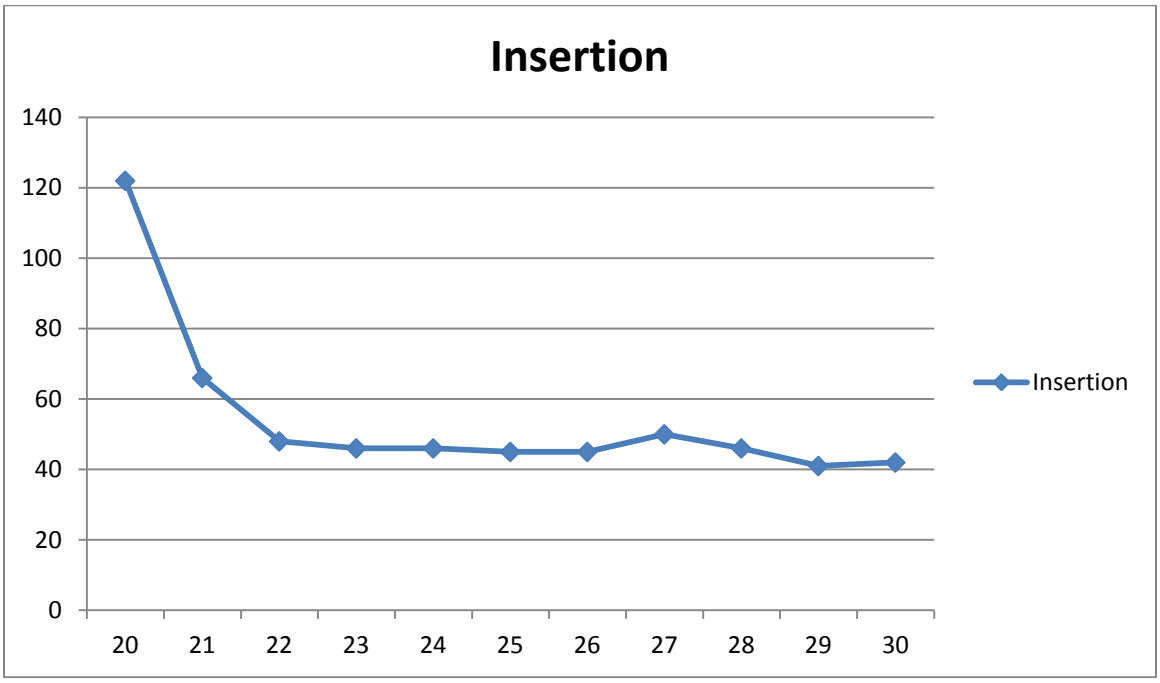
تفاوت زمانی ساختمان داده ها

همانطور که در تحلیل الگوریتم ها توضیح داده شد ماتریس مجاورت بسیار زمان بیشتری نسبت به لیست مجاورت مصرف میکنه و این به دلیل ان است که ما با محدودیت حجم مواجه شدیم و از ماتریس های خلوت استفاده کردیم به همین دلیل برای هر الگوریتم پیچیدگی زمانی ما به تعداد یال ها مربوط میشود و این خیلی هزینه ی بالایی است.

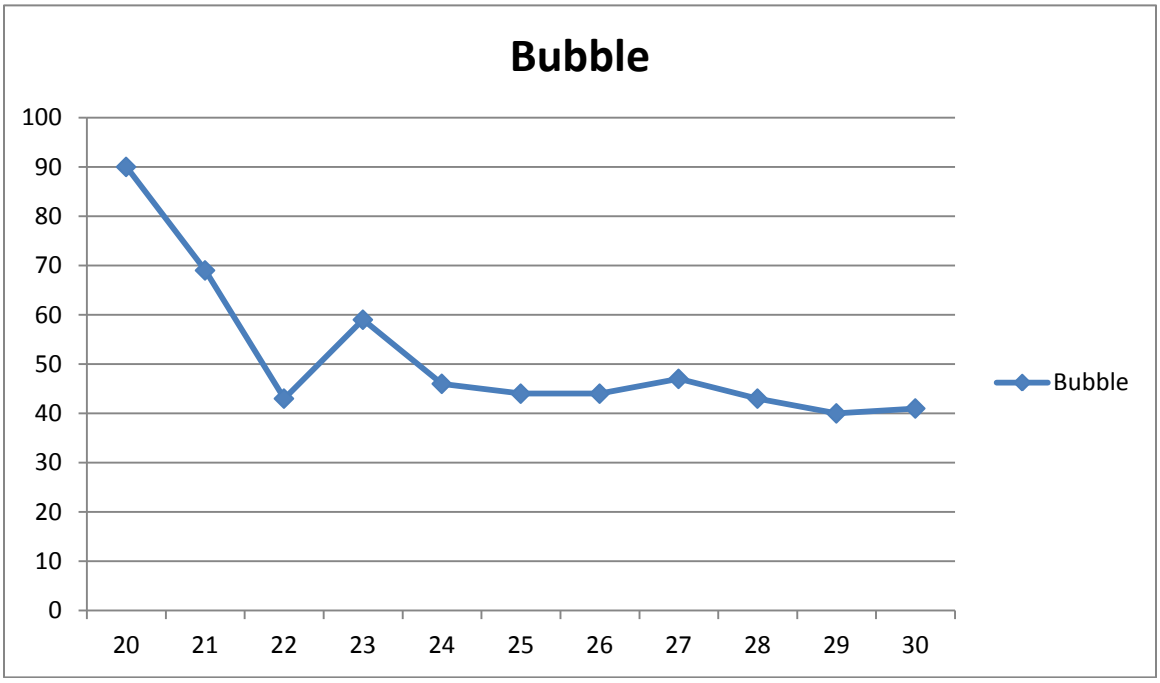
تفاوت حافظه ای ساختمان داده ها

از لحاظ حافظه ای ساختمان داده ی ماتریس خلوت بسیار بهینه تر از لیست مجاورت بود و دلیل ان هم داشتن تنها یک لیست از یال ها اسدت در صورتی که در لیست مجاورت ما علاوه بر لیست یال ها لیستی از گره های مجاور داریم.

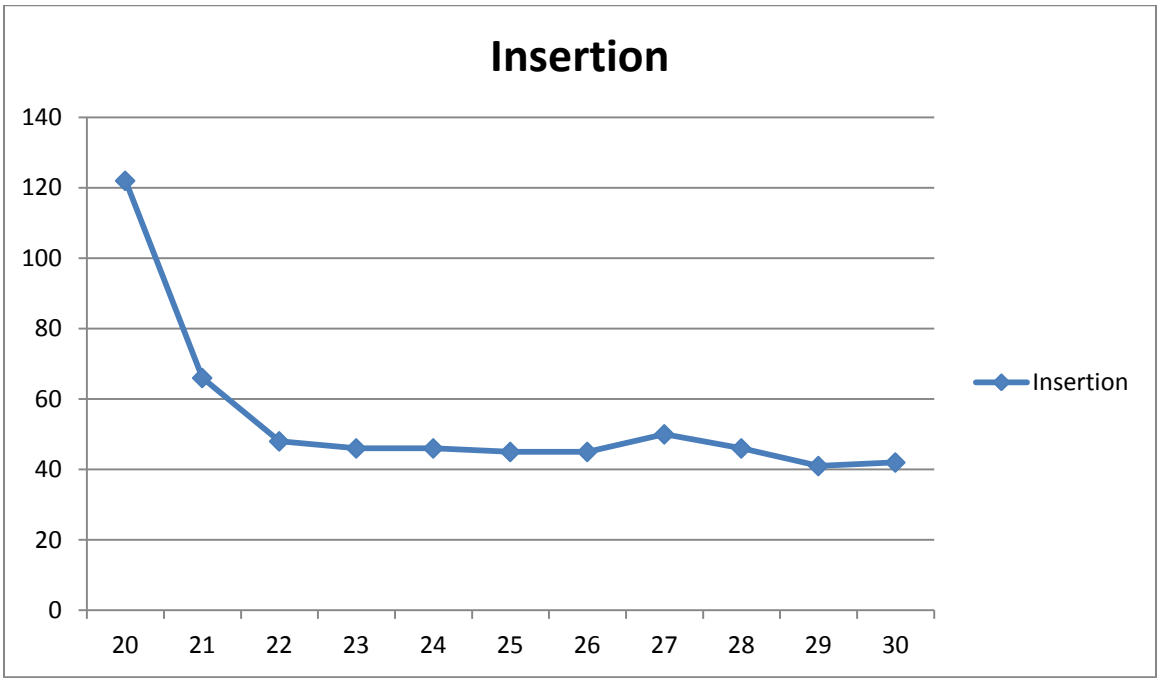
یافتن Nمناسب برای لیست insertion بر روی test4



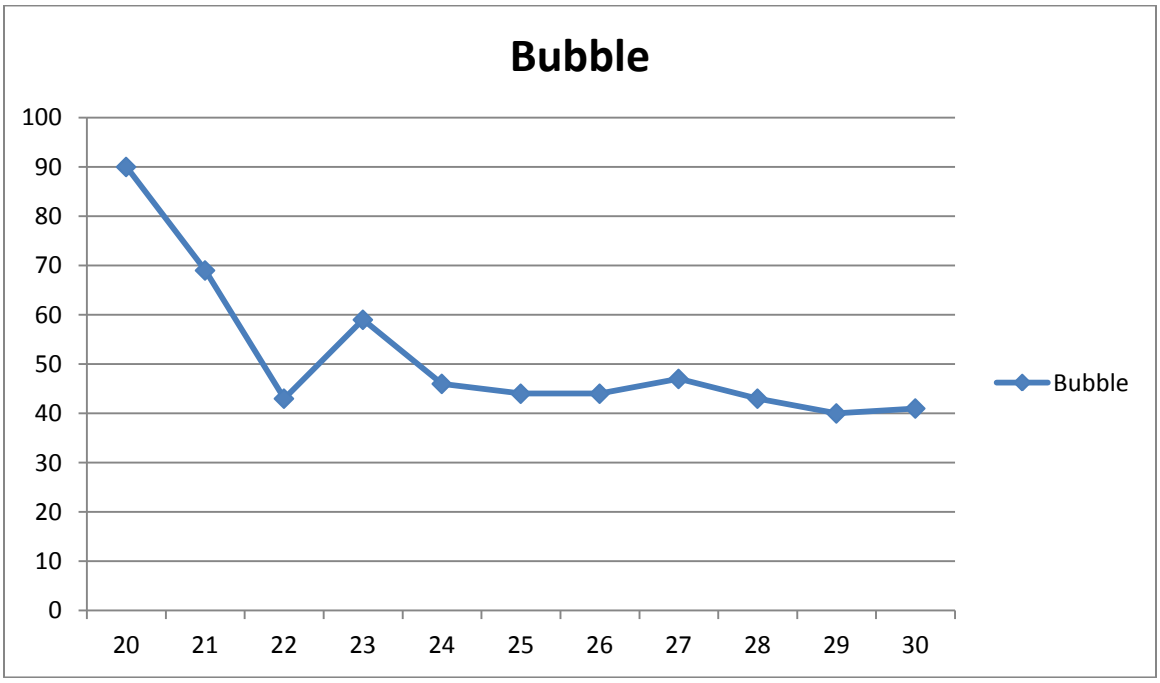
یافتن Nمناسب برای لیست bubble بر روی test4



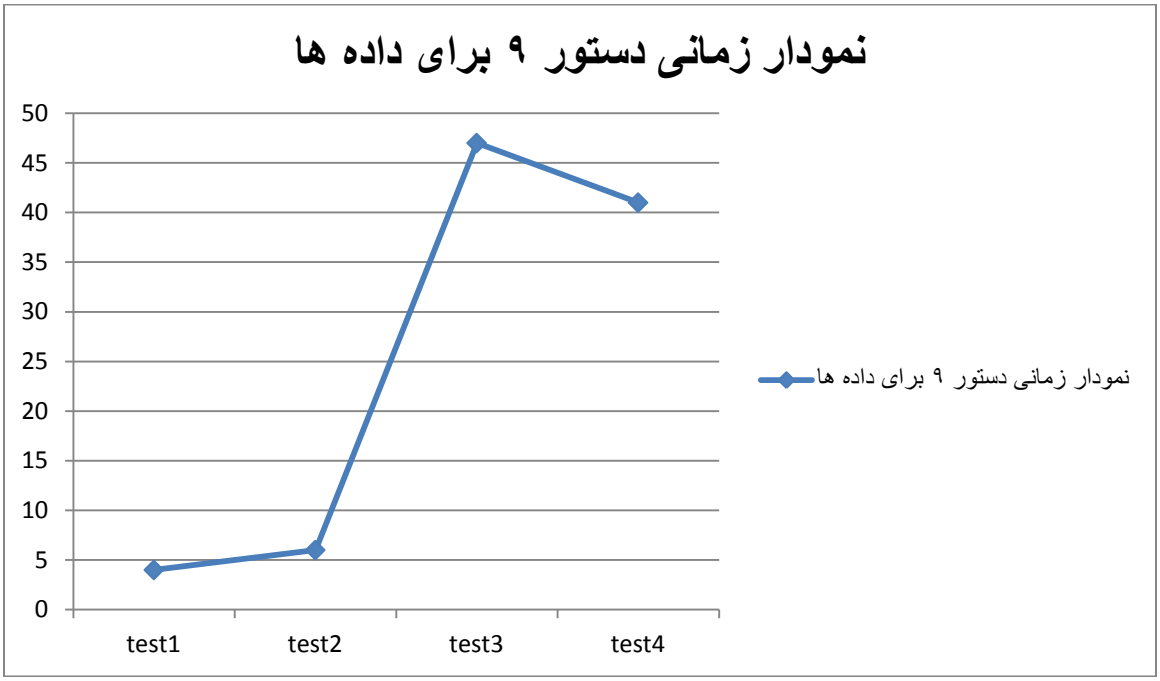
یافتن Nمناسب برای ارایه مجاورت insertion



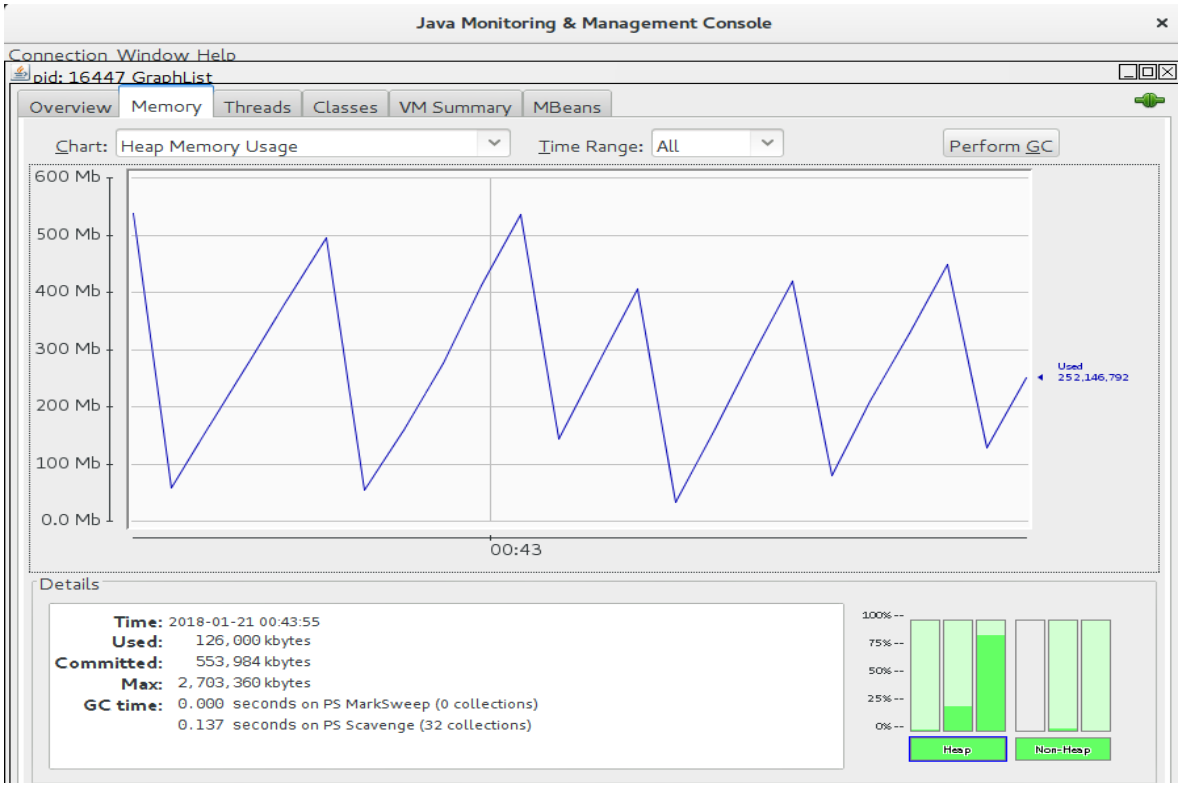
یافتن Nمناسب برای ارایه مجاورت bubble



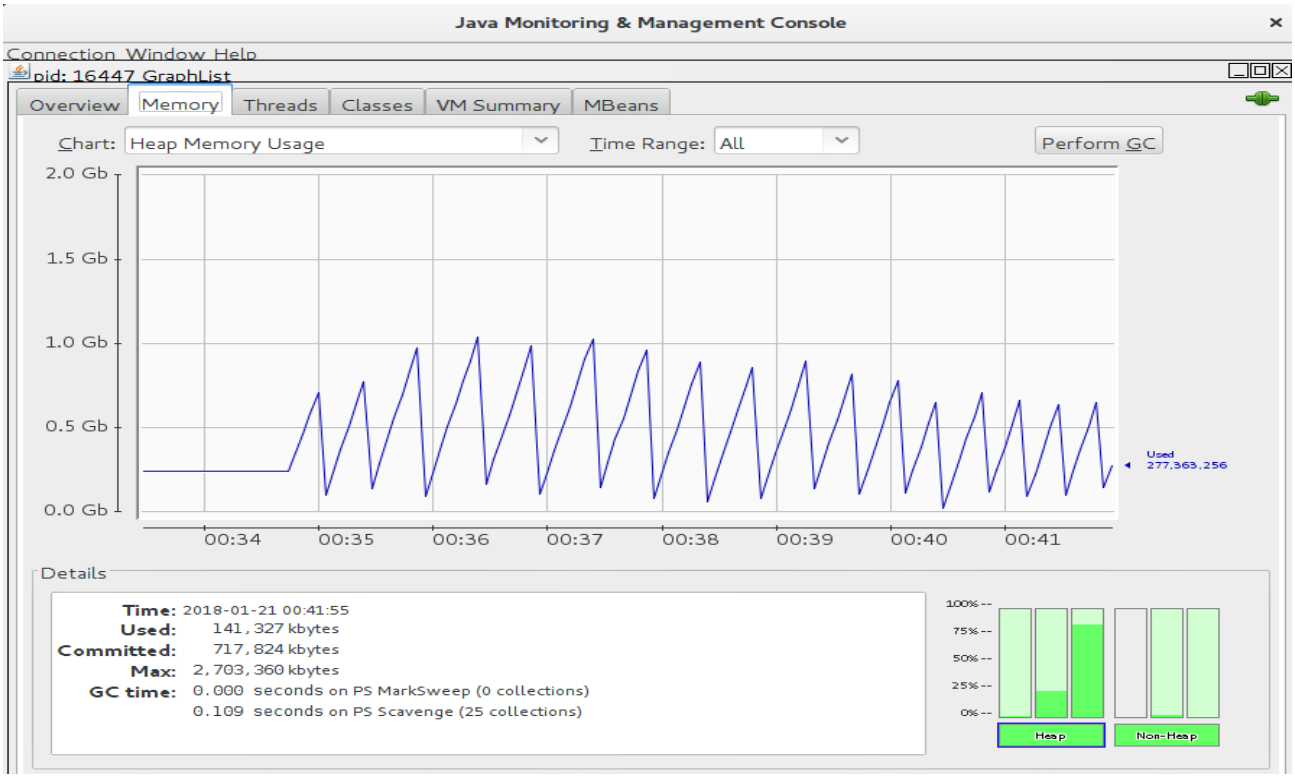
دستور شماره 9 (optimumSort-29 – insertion-List) برای تست های مختلف بر حسب میلی ثانیه



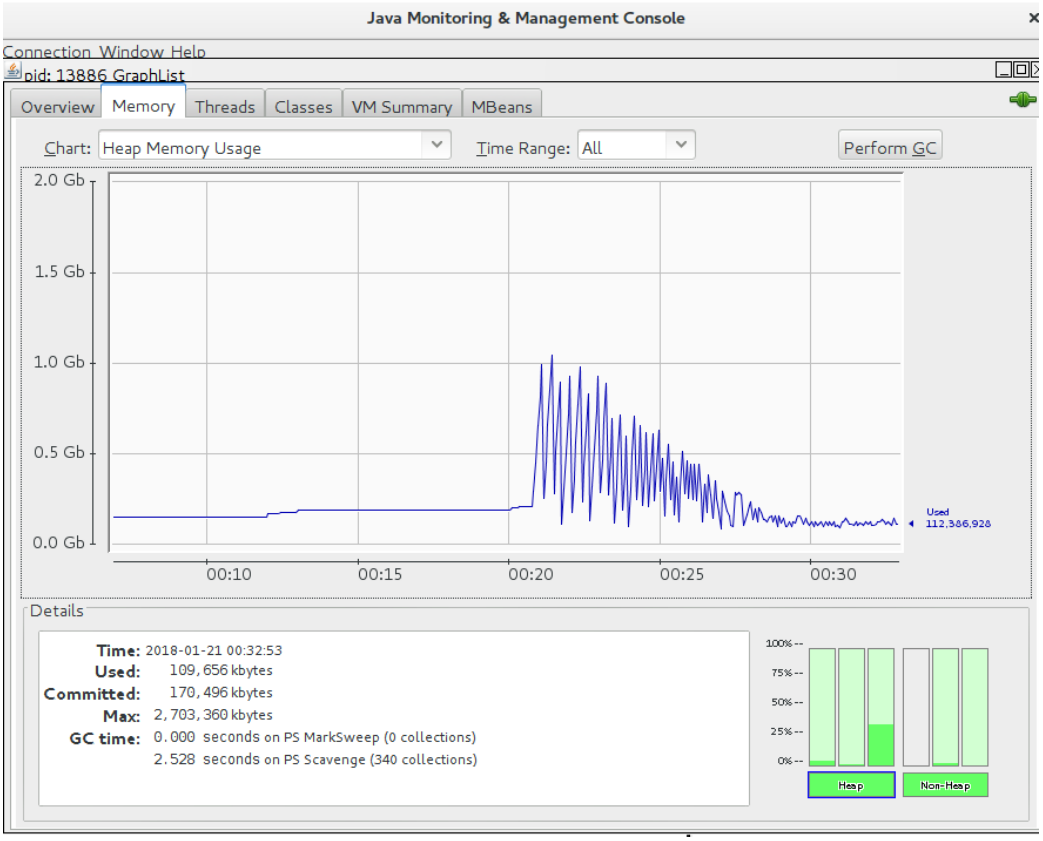
نمودار حافظه مصرفی دستور شماره 9 (optimumSort-29 – insertion-List – test 1)



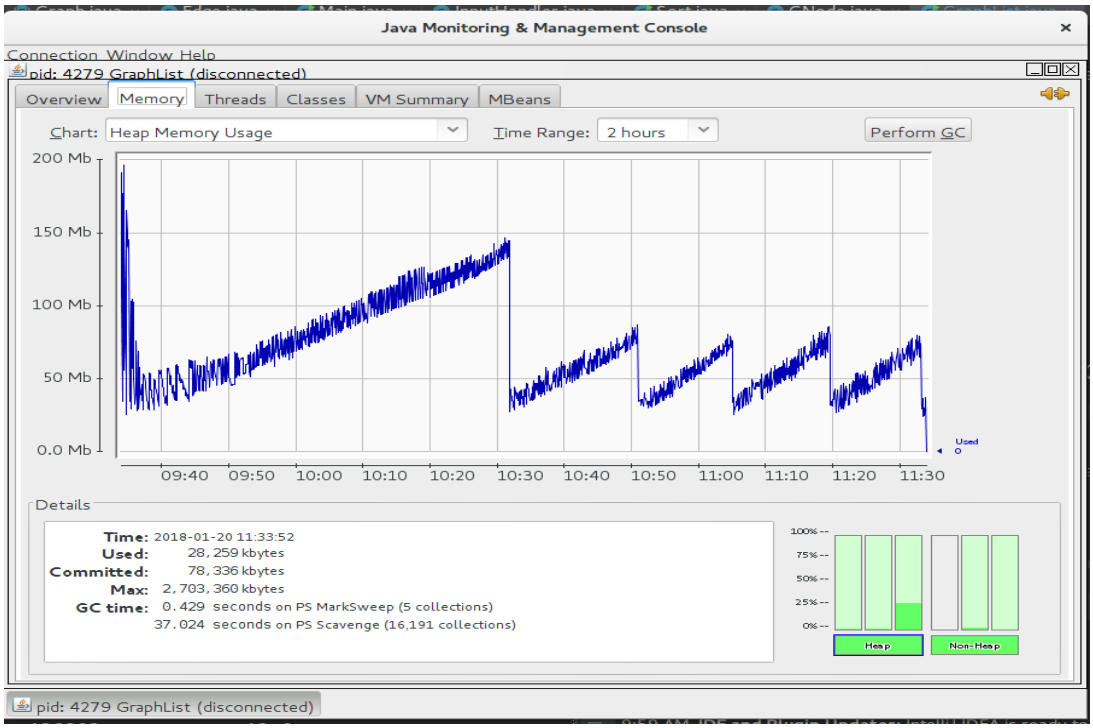
نمودار حافظه مصرفی دستور شماره 9 (optimumSort-29 – insertion-List – test2)



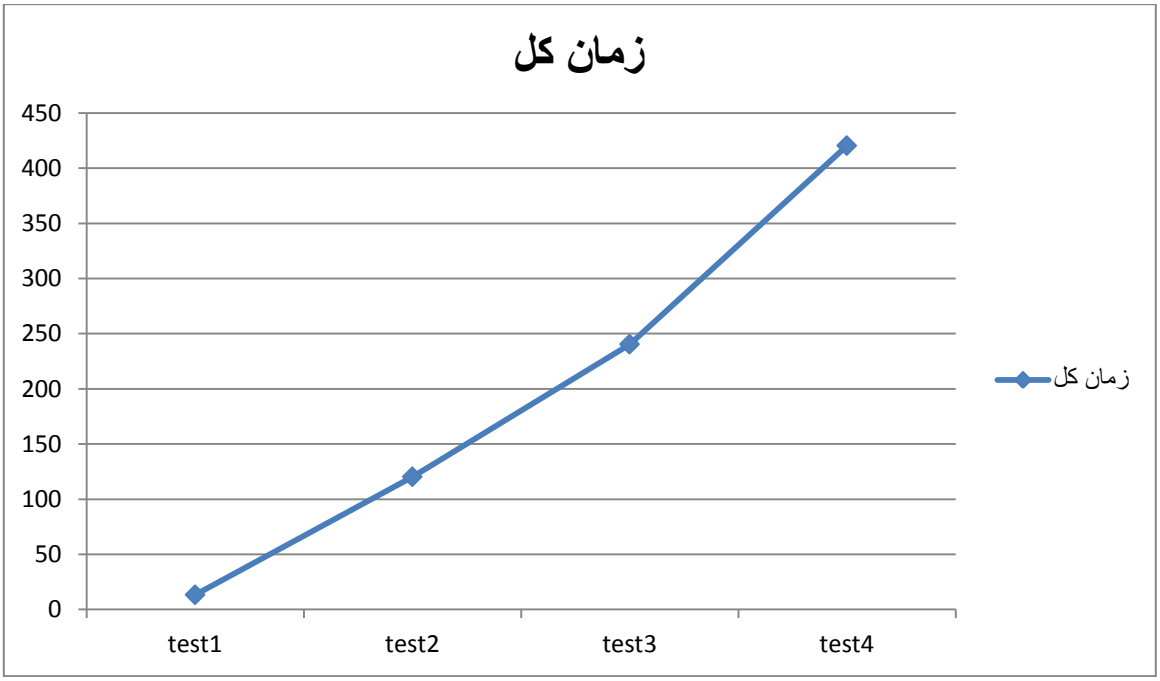
نمودار حافظه مصرفی دستور شماره 9 (optimumSort-29 – insertion-List-test3)



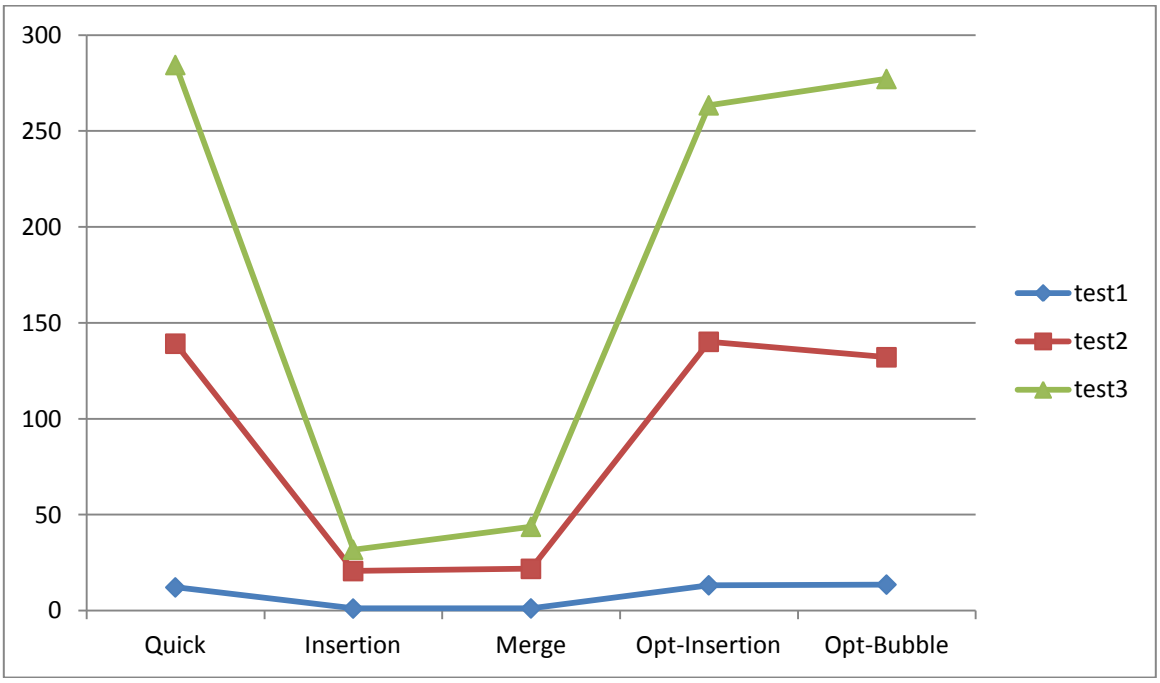
نمودار حافظه مصرفی دستور شماره 9 (optimumSort-29 – insertion-List – test4)



نمودار زمان کل دستور شماره 9 (optimumSort-29 – insertion-List) برای تست های مختلف بر حسب دقیقه



نمودار زمانی دستورات فرد برای تست های مختلف(List) بر حسب دقیقه



نمودار زمانی 5 قدم برای تست های مختلف(List) بر حسب دقیقه