# ارائه رویکردی نوین و خودکار به منظور تولید داده های تست مبتنی بر روشهای تصادفی

**آرش صباغی**[1*]

1- مربی گروه کامپیوتر، واحد سمنان، **دانشگاه آزاد اسلامی**، سمنان، ایران
* a.sabbaghi@semnaniau.ac.ir

**چکیده**

فرآیند تست 50% کل هزینه توسعه نرم افزار را شامل می شود. به منظور تست نرم افزار، باید مجموعه ای از داده‌های تست ایجاد شوند که تولید این مجموعه، کاری بسیار زمانبر بوده و تاثیر مستقیمی بر هزینه فرآیند تست خواهد داشت. بدون خودکارسازی، این فرآیند، کند، پرهزینه و همراه با خطا خواهد بود. در این مقاله روش جدیدی به منظور تولید داده های تست بر مبنای الگوریتمهای تصادفی با ترکیب روشهای ایستا و پویا ارائه شده است. روش پیشنهادی با بهره برداری از ماهیت ساده تست تصادفی و همچنین استفاده از اطلاعاتی که می‌توان از کد منبع برنامه بدست آورد به تولید داده های تست پرداخته و کارایی تست تصادفی را افزایش می دهد. نتایج بدست آمده حاکی از افزایش سرعت تولید و همچنین کیفیت مجموعه تست می باشد.

**کلیدواژگان**

تست نرم افزار، تست تصادفی، تولید داده های تست

# A Novel Automated Random Based Test Case Generation Approach

**Arash Sabbaghi 1**

1- Department of Computer Engineering, Semnan Branch, **Islamic Azad University**, Semnan, Iran.
a.sabbaghi@semnaniau.ac.ir

**Abstract**

Testing activities takes 50% of overall cost of software development process. In order to perform software testing, a set of test cases should be generated. Generating this set is so time consuming and have a direct impact on the cost of software testing. Without automation, this process is slow, expensive and error prone. In this paper, a new approach in order to generate test cases based on random testing by combining static and dynamic information is proposed. The proposed approach generates test cases by utilizing the simple nature of random testing and also using the information which can be gained by the source code that leads to improve in the performance of random testing. The experimental results indicate an increase in the test case generation speed and also quality of test suite.

**Keywords**

Software Testing. Random Testing, Test Case Generation

## 1- مقدمه

تست نرم افزار فرآیندی است که کیفیت نرم افزار کامپیوتری را مشخص میکند. تست، شامل فرآیند اجرای یک برنامه با هدف یافتن خطاهای نرم افزاری است، اما محدود به آن نمی باشد. به صورت دقیق تر فرآیند تست را می توان به این صورت تعریف کرد: آزمون یک برنامه کاربردی به منظور کشف خطاها، تضمین برآورده کردن نیازمندیهای موجود و سازگاری با سخت افزار مشتری(محیط) است [12]. تولید داده های تست از دشوارترین و زمان بر ترین مراحل فرآیند تست ساختاری نرم افزار بوده و پوشش کد معیاری به منظور سنجش کافی بودن تعداد اعضای مجموعه تست می باشد[4 ,1]. درحال حاضر هزینه بسیار زیادی برای محصولات نرم افزاری پرداخت می شود که در صورت شکست خوردن، تولیدکنندگان و مصرف کنندگان متحمل هزینه‌های زیاد و بعضا غیر قابل جبران می‌شوند. بیشتر از یک سوم این هزینه قابل اجتناب است، اگر فرآیند تست نرم افزار بهتر انجام شود.

فرآیند تست 50% کل هزینه توسعه نرم افزار را شامل می‌شود. از نظر مهندسان نرم افزار نوشتن کدهای تست، به خودی خود، مثل توسعه خود محصول، وقت گیر و گران است. امروزه سازمان‌های نرم افزاری زمان و منابع زیادی را در تحلیل و تست نرم افزار صرف میکنند.

با توجه به توضیحات فوق به نظر می رسد خودکار سازی روال تست امری اجتناب ناپذیر است. در این مقاله روش نوینی برای خودکارسازی تولید داده های تست با کارایی بالا ارائه شده و با توجه به روش پیشنهادی، امکان تعیین معیاری به منظور تشخیص ماژول هایی که دارای پیچیدگی بیشتری جهت تولید داده های تست هستند فراهم گردیده است.

در رویکرد پیشنهادی، الگوریتمی کارا مبتنی بر تولید داده تست به روش تصادفی با توجه به معیار پوشش شاخه[17] و ترکیب تحلیل ایستا و پویا ارائه شده است. البته به منظور افزایش بهره وری و کارایی سیستم تولید کننده داده تست، در استفاده از هر کدام از روشهای ایستا و پویا تغییراتی اعمال شده است. درواقع بخشی از این رویکردها که بار محاسباتی بالا یا کارایی پایینی دارند مورد استفاده قرار نگرفته و جایگزینی مناسب برای آنها ارائه شده است.

در تست تصافی، داده های تست به صورت تصادفی از دامنه ورودی برنامه تحت تست انتخاب شده و سپس بررسی می شود که آیا برنامه با اجرای داده مربوطه رفتار درستی ارائه می دهد یا خیر. تست تصادفی در بسیاری از سیستمهای نرم افزاری قابل اعمال بوده، ساده، ارزان و قابل انعطاف است[14 ,13]. از طرف دیگر، ماهیت ساده و تصادفی آن که ممکن است

منجر به پوشش کد پایین و داده های تست افزونه شود در معرض انتقاد قرار دارد[15, 16]. در روش پیشنهادی به منظور بهره مندی از مزایای تست تصادفی و افزایش کارایی آن، از تمام اطلاعاتی که می‌توان در زمان کامپایل برنامه استفاده کرد، استفاده می‌شود. در واقع در این روش به شکل خاصی، از اجرای نمادین[3, 4, 6] استفاده می‌شود. این استفاده جزئی خواهد بود و با اجرای پویا [2, 5] ترکیب می‌شود. اطلاعاتی که به صورت ایستا در هنگام کامپایل ایجاد می‌شوند به غنی‌تر سازی اطلاعات اولیه مورد نیاز برای تولید داده های تست کمک شایانی کرده و پروسه تست را با سرعت بیشتری در راستای تولید داده های تست با معیار پوشش شاخه همگرا می‌کند.

ادامه مقاله بدین شکل سازماندهی شده است. بخش دوم به مرور کارهای گذشته در این حوزه پرداخته است. در بخش سوم روش پیشنهادی را ارائه می کنیم. بخش چهارم، پیاده سازی انجام شده و نتایج آنرا شرح می دهد و نهایتا فصل پنجم به نتیجه گیری اختصاص یافته است.

## 2- کارهای گذشته

استفاده از روش های تولید داده تست تصادفی به تنهایی کارا نیست و در واقع هوشمندی لازم را ندارد[20]. بنابراین روشهایی ارائه شده‌اند که با اعمال تغییراتی در آن، باعث افزایش هوشمندی و هدفمند شدن تولید داده های ورودی شده‌اند. نمونه ای از این روشها، تولید داده تست تصادفی تطبیق یافته[1] می باشد[21, 22]. ایده این روش این است که داده های تست باید از کل دامنه ورودی انتخاب شوند. در واقع وقتی یک مورد تست، خطایی را آشکار نمی‌کند، مورد تست بعدی باید دورتر از آن انتخاب شود. به عبارت دیگر در روش تطبیق یافته، هدف این است که از الگوهای خطایی که در بسیاری از برنامه ها دیده شده است استفاده کنیم. به این الگوها در[11] اشاره شده است.

روش دیگری که به منظور افزایش کارایی الگوریتم های تصادفی ارائه شده است، تست پارتیشن[2] [23, 24] نام دارد. در این روش دامنه ورودی بر اساس دانشی که وجود دارد یا بدست می‌آوریم به چند زیر دامنه تقسیم و داده های تست از هر زیر دامنه انتخاب می‌شوند. نحوه تقسیم بندی می‌تواند به صورت زیر باشد:

1- یک زیر دامنه ممکن است شامل ورودی هایی که منجر به اجرای یک شاخه مشخص می‌شوند باشد.
2- یک زیر دامنه ممکن است شامل ورودی های هم ارز از یک دیدگاه و خصوصیت مشخص باشد.
3- یک زیر دامنه ممکن است شامل ورودی هایی که منجر به یک خطای خاص می‌شوند باشد.

روشهای فوق که به منظور هوشمند کردن الگوریتم های تصادفی معرفی شده اند کیفیت داده های تست را نسبت به تولید کاملا تصادفی بالا می‌برند اما هزینه سنگینی به تولید کننده داده تست اعمال می‌کنند. به عنوان مثال، تعریف چگونگی افراز دامنه ورودی کار چندان ساده‌ای نیست. مقالات متعددی در مقایسه روشهای تصادفی با روشهای سیستماتیک[18, 19, 20] و تست پارتیشن یا تطبیق یافته ارائه شده[8] که همگی به برتری روشهای تصادفی با توجه به سادگی و قابلیت بالا اذعان کرده اند.

فقط در موارد خاصی، الگوریتمهای تصادفی به خوبی عمل نمی‌کنند و باید در این زمینه فکری کرد[8]. به عنوان مثال ارضاء شرط های تساوی با روشهای تصادفی به سختی و با احتمال کمی‌امکان پذیر است. ایراد دیگری که به روشهای تصادفی وارد است، این است که برای پوشش تمامی‌مسیرها، تعداد داده های تست زیادی باید تولید شوند. در واقع در روشهای تصادفی مجموعه تست دارای اعضای زیادی است که بسیاری از آنها هدف مشترکی را ارضا می‌کنند. این امر سبب می‌شود در هنگام اجرای تست با داده های تولید شده، منابع به هدر روند و داده های تستی اجرا شوند که تاثیری در یافتن خطاها جدید ندارند و می‌دانیم مجموعه تستی مرغوب است که با کمترین تعداد اعضا، بیشترین تعداد خطا را کشف کنند. بنابراین کاری که باید انجام داد این است که از سادگی و کارایی الگوریتمهای تصادفی در تولید داده های تست استفاده کرد و ایرادات آنرا به حداقل رساند.

در روش پیشنهادی جدید از تمام اطلاعاتی که می‌توان در زمان کامپایل برنامه استفاده کرد، استفاده می‌شود. در واقع در این روش به شکل خاصی، از اجرای نمادین استفاده می‌شود. این استفاده جزئی خواهد بود و با اجرای پویا ترکیب می‌شود. اطلاعاتی که به صورت ایستا در هنگام کامپایل ایجاد می‌شوند به غنی‌تر سازی اطلاعات اولیه مورد نیاز برای تولید داده های تست کمک شایانی کرده و پروسه تست را با سرعت بیشتری در راستای تولید داده های تست با معیار پوشش شاخه همگرا می‌کند.

یکی از مشکلات اجرای نمادین، نیاز به حل کننده های محدودیت است[4]. در واقع اجرای نمادین در طول اجرا، محدویتهای شاخه ها را جمع آوری کرده و یک محدودیت کلی تولید می‌کند. این محدودیت عملا یک مساوی با نا مساوی می‌باشد. در نهایت برای حل این محدودیت، احتیاج به ابزارهای حل کننده محدودیت می‌باشد. محاسبات مربوط به حل محدودیتها سنگین بوده و در مواردی حل کننده محدودیت قادر به حل یک معادله نیست. همچنین باید یک حل کننده محدودیت متناسب با زبان برنامه تحت تست تولید گردد. همه اینها از انگیزه هایی است که به سمت استفاده صرف از اجرای نمادین به عنوان یک روش پر هزینه نرویم.

مقالاتی نیز در زمینه استفاده از الگوریتم های جستجو به منظور حل کردن محدودیتها و رسیدن به مقادیر واقعی متغییرهای محدودیت ارائه شده است[3,9,10]. ایده اصلی این روشها این است که با استفاده از گراف کنترل جریان، محدودیت مسیر را تولید کرده و سپس با استفاده از الگوریتم ژنتیک یا PSO به جستجو در فضای ورودی پرداخته و مقادیری که محدودیت را ارضا می‌کنند را می‌یابند.

مهمترین عامل موفقیت الگوریتم های جستجو تعریف صحیح تابع برازش می‌باشد. این مورد در تولید داده های تست از اهمیت بیشتری برخوردار است[7]. در موارد شرط های ساده، تعریف تابع برازش با توجه به معیار فاصله شاخه ساده است و الگوریتم جستجو به خوبی عمل می‌کند، اما بزرگ شدن محدودیت و اضافه شدن شرط ها و متغییر های مختلف، تعریف تابع برازش را با دشواری مواجه کرده و زمان همگرا شدن آنرا به شدت افزایش می‌دهد.

## 3- روش پیشنهادی

در این بخش به بررسی روش پیشنهادی پرداخته، ایده اصلی و مزایای آنرا شرح می‌دهیم. شکل 2 بلوک دیاگرام کلی روش پیشنهادی را نشان می دهد. روال کار به این ترتیب خواهد بود که ابتدا با تحلیل ایستا و تولید گراف وابستگی داده، تمامی‌گزاره های شرط برنامه استخراج شده و در یک جدول

[1] Adaptive Random Test Case Generator
[2] Partition Testing

تحت عنوان جدول ارضا[1] ذخیره می‌شوند. در این مرحله تمامی‌متغییرهای استفاده شده در گزاره، استخراج شده و در این جدول قرار می‌گیرد. با توجه به اینکه در این جدول شرطهای ساده ذخیره می‌شوند ارضاء آنها ساده بوده و به صورت تصادفی با کارایی مناسبی انجام خواهد شد. خروجی دیگر فاز تحلیل ایستا، جدول مسیر[2] می‌باشد. در این جدول مسیرهای مورد نظر به همراه گزاره های خود که باید مقدار True یا False داشته باشند، ذخیره می‌گردد.

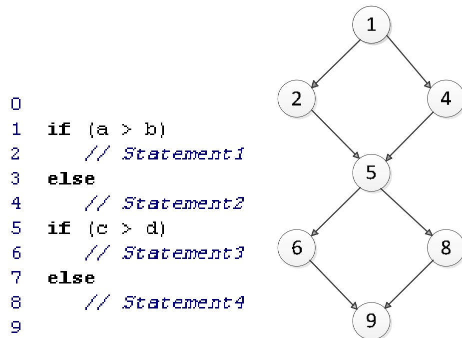
در مرحله بعد، هر شرط(به جز شرطهای تساوی)، از جدول ارضا استخراج شده و با تولید داده های تصادفی ارضا می‌شوند. در این مرحله، دو داده تست جهت ارزیابی شرط به مقادیر True و False تولید می‌کنیم. همانطور که اشاره شد، با توجه به سادگی شرط ها، معمولا در اولین گام، داده های مورد نیاز جهت ارضای شرط به مقادیر True و False بدست می‌آیند. این داده ها، در واقع یک بهینه جزئی هستند.

با توجه به توضیحات قبلی، در روشهای تصادفی، بدست آوردن داده های تستی که شرطهای تساوی را ارضا کنند بسیار دشوار بوده و با احتمال کمی همراه است. به این منظور در این مرحله جهت ارضای شرطهای تساوی تلاش می‌کنیم. به این منظور می‌توان از حل کننده های محدودیت یا الگوریتم ژنتیک استفاده کرد. دلیل پیشنهاد الگوریتم ژنتیک برای ارضای اینگونه شرطها احتمال وجود محاسبات غیرخطی یا ماجولهای غیر قابل دسترس در گزاره های شرط می باشد که ممکن است خارج از قابلیت حل کننده محدودیت مورد استفاده باشد[4]. با توجه به سادگی تک شرط تساوی، الگوریتم ژنتیک یا حل کننده محدودیت، به سرعت و با کمترین حجم محاسبات، عمل می‌کنند.

بعد از بدست آوردن داده هایی که هر شرط را True یا False می‌کند، آنها را به جدول ارضا اضافه می‌کنیم. در نهایت پس از سپری شدن این مرحله، در جدول ارضا با کمترین هزینه، هر شرط و داده هایی که آنها را True و False می‌کنند را خواهیم داشت.

یک مسیر اجرایی، یک دنباله از True و False ها است. نکته قابل توجه این است که حاصل دنباله، که and منطقی تک تک گزاره ها است باید True باشد. در مرحله بعد، از جدول مسیر، هر مسیر، گزاره های مربوط به آن و مقداری که باید داشته باشند(T یا F) را استخراج می‌کنیم. سپس از جدول ارضا، داده های تست مربوط به هر گزاره را با توجه به مقدار مورد نظر استخراج می‌کنیم. در ادامه، باید داده های تستی که هر گزاره را True یا False می‌کند را با هم ترکیب کرد و به داده تستی برسیم که کل مسیر را پیمایش کند. نکته قابل توجه در این بخش این است که داده های تستی که دو گزاره را به تنهایی True می‌کند الزاما and منطقی آنها را True نمی‌نماید. البته در غالب برنامه های دنیای واقعی شرط ها چنین شرایطی ندارند. برای بررسی دقیق تر این موضوع ذکر نکاتی الزامیست.

اگر گزاره هایی که محدودیت یک مسیر را شکل می‌دهند، هیچ متغییر مشترکی نداشته باشند، مسلما داده های تست ذخیره شده در جدول ارضا، با احتمال 1، کل مسیر را ارضا می‌کند. هر چه تعداد متغییرهای مشترک گزاره های یک محدودیت مسیر بیشتر باشد، احتمال اینکه داده های تست جدول ارضا نتوانند آن مسیر را ارضا کنند بیشتر خواهد بود. بنابراین با توجه به وابستگی بین گزاره ها می‌توان معیاری جهت پیچیدگی روال تست یک مسیر یا به صورت کلی یک ماجول ارائه داد.



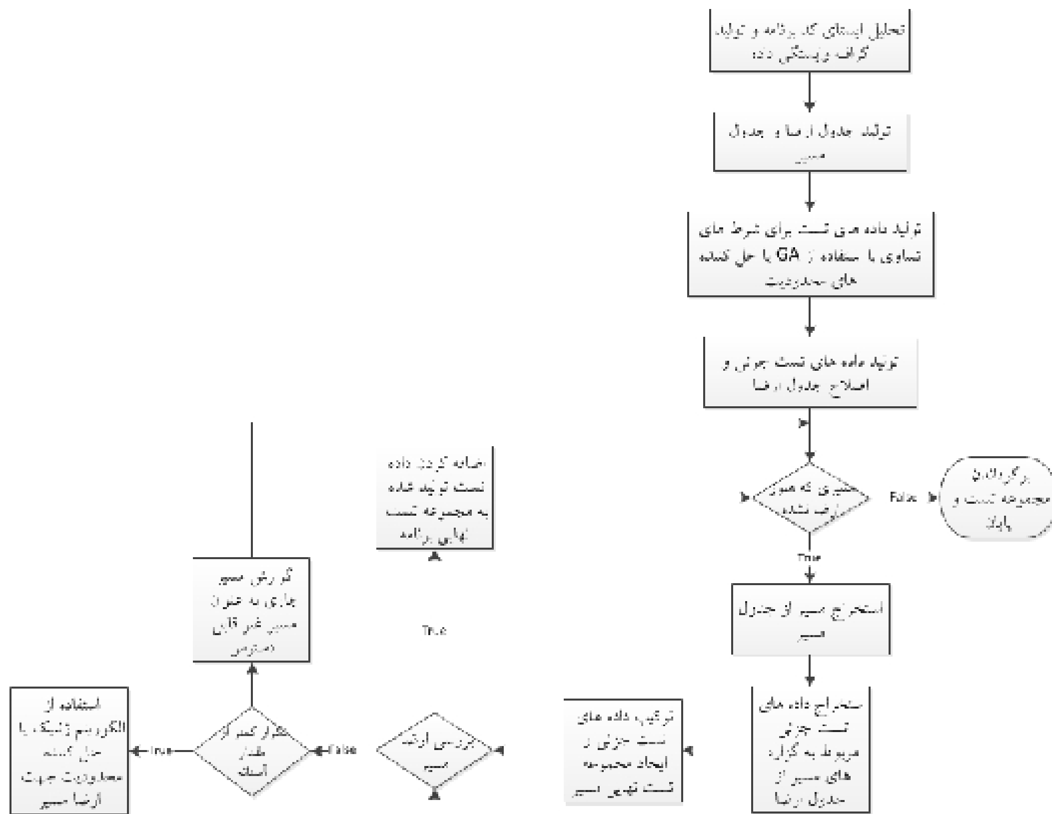

**شکل 1** کد برنامه تحت تست و CFG مربوطه

به عنوان مثال، کد شکل 1 و CFG مرتبط با آنرا در نظر بگیرید. در حین تحلیل ایستا، گراف کنترل جریان برنامه ایجاد شده و جدول ارضا و جدول مسیر برای آن ایجاد می‌گردد. در ابتدا مقادیرمربوط به داده های تست خالی می‌باشد. سپس به صورت تصادفی مقادیری که هر گزاره را True و False می‌کند را بدست می‌آوریم. با توجه به ساده بودن گزاره ها در این مرحله، مقادیر تصادفی با احتمال زیادی داده های صحیحی تولید می‌کنند. جدول شماره 1، جدول ارضا برای تکه کد شکل 1 را نشان می دهد.

با توجه به گراف CFG، تکه کد مشخص شده در شکل 1، 4 مسیر خواهد داشت. جدول مسیر نیز در مرحله تحلیل ایستا و با توجه به سیاستهای مربوطه مطابق جدول 2 ایجاد می‌گردد. سپس با توجه به گزاره های استفاده شده در هر مسیر که در جدول مسیر موجود است، داده تست نهایی برای این مسیر با ترکیب موارد تست جزئی ساخته می‌شود. در مثال شکل 1 با توجه به اینکه گزاره های مسیر، هیچ متغییر اشتراکی ندارند، داده تست مسیر به سادگی ایجاد می‌گردد. این عمل برای تمام مسیرها دنبال خواهد شد. بنابراین با تولید یک مجموعه تست با اندازه 4 داده تست، تمام مسیرها پیمایش خواهند شد.

اما در صورت وجود داده اشتراکی در گزاره های شرط مسیر، وضعیت اندکی متفاوت خواهد بود. به عبارت دیگر ترکیب داده های تست ممکن است محدودیت مسیر را ارضا نکنند. فرض کنید تغییری در تکه کد شکل 1، مطابق شکل 3 ایجاد کنیم. به این ترتیب که در شرط دوم متغییر اشتراکی a وجود داشته باشد. جدول ارضا به شکل جدول شماره 3 تغییر خواهد کرد.

---
1 Satisfy Table
2 Path Table




**شکل** 2 نمودار جریان روش پیشنهادی

**جدول**1 جدول ارضاء

| شماره | گزاره | شاخه True | | | | شاخه False | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | d | a | b | c | d |
| 1 | a>b | 631518 | 354486 | - | - | 195234 | 381216 | - | - |
| 2 | c>d | - | - | 970438 | 76488 | - | - | 619097 | 780101 |

**جدول** 2 جدول مسیر

| شماره مسیر | محدودیت مسیر | شاخه | | داده های تست | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | a | b | c | d |
| 1 | (a>b) and (c>d) | T | T | 631518 | 354486 | 970438 | 76488 |
| 2 | (a>b) and not (c>d) | T | F | 631518 | 354486 | 619097 | 780101 |
| 3 | not (a>b) and (c>d) | F | T | 195234 | 381216 | 970438 | 76488 |
| 4 | not(a>b) and not(c>d) | F | T | 195234 | 381216 | 619097 | 780101 |

```
0
1  if (a > b)
2     // Statement1
3  else
4     // Statement2
5  if (a > c)
6     // Statement3
7  else
8     // Statement4
9
```

**شکل** 3 کد برنامه تحت تست با متغییر اشتراکی در گزاره های شرطی

**جدول** 3 جدول مسیر تکمیل شده

| شماره | گزاره | شاخه True | | | شاخه False | | |
|---|---|---|---|---|---|---|---|
| | | a | b | c | a | b | c |
| 1 | a>b | 7126 | 215 | - | 45918 | 172423 | - |
| 2 | a>c | 790386 | - | 639113 | 125456 | - | 54506 |

حال فرض کنید بخواهیم داده تست پیمایش مسیر 14569 (not (a>b) and (a>c))را تولید کنیم. بنابراین شرط اول باید با Fasle و شرط دوم با True ارضا شود تا حاصل شرط کل مسیر True گردد. دو داده تست مربوط به این گزاره ها را از جدول ارضا استخراج می‌کنیم. این داده ها در جدول 4 مشخص شده اند.

**جدول** 4 جدول داده های تست جزئی

| داده های تست جزئی | | |
|---|---|---|
| a | b | c |
| 45918 | 172423 | - |
| 790386 | - | 639113 |

با ترکیب کردن این داده های تست، نهایتا دو داده تست کامل بدست می‌آید که در جدول 5 مشخص شده اند.

**جدول** 5 جدول داده های تست نهایی

| داده های تست نهایی | | |
|---|---|---|
| a | b | c |
| 45918 | 172423 | 639113 |
| 790386 | 172423 | 639113 |

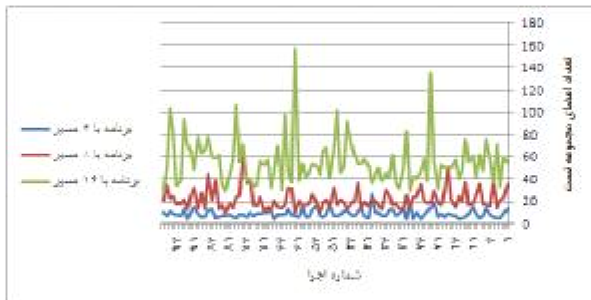
جالب توجه است که داده تست اول محدودیت مسیر را ارضا نمی‌کند و داده تست دوم آنرا ارضا می‌کند. در پیاده سازی انجام شده مشخص گردید که در بیشتر از 73% موارد، حداقل یکی از داده های تست تولید شده، شرط مسیر را ارضا می‌کند. با توجه به توضیحات فوق مشخص شد، که با هزینه بسیار پایین تر از تست تصادفی و بدون نیاز به حل کننده محدودیت جهت حل محدودیتهای بسیار بزرگ که در غالب برنامه های دنیای واقع اتفاق می‌افتد، می‌توان مجموعه تستی ایجاد کرد که هدف مورد نظر را ارضا کند.

## 4- پیاده سازی و نتایج تجربی

ایده پیشنهادی با استفاده از نرم افزار Delphi پیاده سازی شده و به منظور تحلیل ایستا بر روی کد منبع برنامه، از CIL[1] [33] استفاده شده است. جهت ارزیابی نتیجه گزاره ها و محدودیتهای مسیر با داده های ورودی، یک ارزیاب نوشته شده که عبارات منطقی و ریاضی را با داده های ورودی محاسبه کرده و نتیجه را تولید می‌نماید. به منظور مقایسه روشهای تصادفی و روش پیشنهادی، از تعداد اعضای مجموعه تست استفاده شده است. تعداد اعضای مجموعه تست برابر با تعداد تمام ورودی های تولید شده در طول پروسه تولید داده های تست به منظور پوشش تمام مسیرهای موجود در جدول مسیر، در نظر گرفته شده است.

نتایج پیاده سازی، از کاهش چشمگیر اندازه مجموعه جستجو، نسبت به الگوریتم های تصادفی حکایت دارد. این کاهش در شرایطی که گزاره ها، متغییرهای اشتراکی ندارند مشهود تر است. در واقع این وضعیت ساده ترین حالت یک برنامه می‌باشد و الگوریتم پیشنهادی در این موارد مجموعه تستی با حداقل اعضا تولید می‌کند.

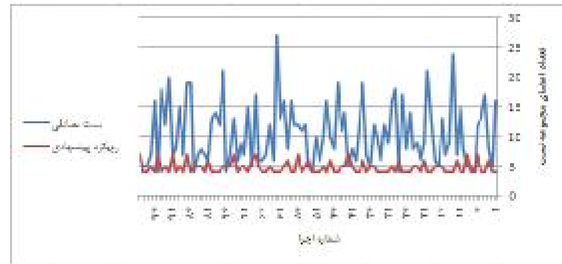
نمودار شکل 4، تعداد اعضای مجموعه تست، در 100 بار اجرای الگوریتم تولید داده تست تصادفی در سه برنامه مختلف با تعداد کل مسیرهای ممکن 4، 8 و 16 را نشان می‌دهد. گزاره های این برنامه ها هیچ اشتراکی با هم ندارند. همانطور که مشاهده می‌شود، در بعضی از اجراها برای برنامه‌ای که تنها 4 مسیر دارد تعداد اعضای مجموعه تست به 25 مورد و برای برنامه‌ای که 16 مسیر دارد، به 157 مورد نیز می‌رسد. اما روش پیشنهادی، برای این نوع برنامه ها مجموعه های تستی با اندازه حداقل یعنی 4، 8 و 16 مورد تولید می‌کند.



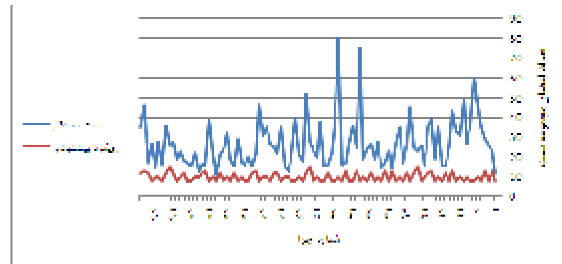

**شکل** 4 تعداد اعضای مجموعه تست، در تست تصادفی در برنامه های بدون گزاره اشتراکی

نمودار اشکال 5، 6 و 7، تعداد اعضای مجموعه تست، در 100 بار اجرای الگوریتم تولید داده تست تصادفی و روش پیشنهادی در سه برنامه مختلف با تعداد کل مسیرهای ممکن 4، 8 و 16 را نشان می‌دهد که دو گزاره این برنامه ها با هم متغییر اشتراکی دارند. همانطور که مشاهده می‌شود، با افزایش ارتباط بین متغییرهای محدودیت مسیر، تعداد اعضای مجموعه تست نیز افزایش می‌یابد. همانطور که مشاهده می شود، روش پیشنهادی در تولید مجموعه تست کارایی بسیار بالاتری از خود نشان داده است.

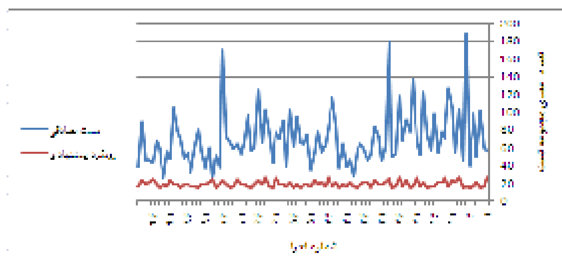
[1] http://cil.sourceforge.net/



**شکل** 5 مقایسه تست تصادفی و روش پیشنهادی در برنامه ای با چهار مسیر به همراه گزاره اشتراکی



**شکل** 6 مقایسه تست تصادفی و روش پیشنهادی در برنامه ای با هشت مسیر به همراه گزاره اشتراکی



**شکل** 7 مقایسه تست تصادفی و روش پیشنهادی در برنامه ای با شانزده مسیر به همراه گزاره اشتراکی

به صورت کلی مزایای روش جدید ارائه شده به صورت زیر می‌باشد:

- استفاده از مزایای تست تصادفی
- کاهش بسیار زیاد نیاز به استفاده از حل کننده های محدودیت. در پیاده سازی انجام شده تنها در 20% مسیرها نیاز به استفاده از حل کننده محدودیت یا الگوریتم ژنتیک است. این امر سربار الگوریتم را به شدت کاهش می‌دهد.
- ایجاد پوشش بالا با توجه به معیار پوشش شاخه
- عدم نیاز به ایجاد تغییر در ساختار برنامه تحت تست
- کاهش چشمگیر تعداد اعضای مجموعه تست

## 5- نتیجه گیری

با توجه به دغدغه های موجود در حوزه تست نرم افزار و هزینه های زیاد این فرایند، تحقیقات زیادی به منظور خودکارسازی آن انجام شده است. در فرآیند تست، از مهمترین این بخشها، تولید داده تست و پیش بینی محل خطا می‌باشد. در واقع می‌توان گفت این بخش ها از مهمترین و پرهزینه‌ترین قسمتهای فرآیند تست نرم افزار هستند و تمرکز زیادی بر روی خودکارسازی آن شده است.

در طول این مقاله روش جدیدی مبتنی بر الگوریتم های تصادفی ارائه شد که با قابلیت بالایی داده های تست مناسب، به تعداد کافی و با توجه به پوشش شاخه ایجاد می‌کند. نتیجه عملی و مقایسات انجام شده حکایت از موفقیت این الگوریتم دارد.

## 6- مراجع


[1] Ferrer, Javier, Francisco Chicano, and Enrique Alba. "Estimating software testing complexity." *Information and Software Technology* 55.12 (2013): 2125-2139.

[2] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.

[3] Baars, Arthur, et al. "Symbolic search-based testing." *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011.

[4] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." *Communications of the ACM* 56.2 (2013): 82-90.

[5] Jamrozik, Konrad, et al. "Generating Test Suites with Augmented Dynamic Symbolic Execution." *Tests and Proofs*. Springer Berlin Heidelberg, 2013. 152-167.

[6] Softwaretechnologie, A. K., Bernhard Aichernig, and Markus Bauer. "Symbolic Execution and Program Testing."

[7] Baresel, André, Harmen Sthamer, and Michael Schmidt. "Fitness Function Design To Improve Evolutionary Structural Testing." *GECCO*. Vol. 2. 2002.

[8] Ntafos, Simeon. "On random and partition testing." *ACM SIGSOFT Software Engineering Notes*. Vol. 23. No. 2. ACM, 1998.

[9] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers", Information Sciences, vol. 178, no. 15, (2008) August, pp. 3075-3095.

[10] A. Sharma, A. Jadhav, P. R. Srivastava and R. Goyal, "Test cost optimization using tabu search", J. Soft. Eng. Appl., vol. 3, no. 5, (2010), pp. 477–486.

[11] Cai, K.-Y., 2002. Optimal software testing and adaptive software testing in the context of software cybernetics. Information and Software Technology, 44(14), 841–855.

[12] Van Vliet, Hans, Hans Van Vliet, and J. C. Van Vliet. Software engineering: principles and practice. Vol. 3. Wiley, 1993.

[13] Ciupa, Ilinca, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. "On the number and nature of faults found by random testing."Software Testing, Verification and Reliability 21, no. 1: 3-28, 2011.

[14] Arcuri, Andrea, Muhammad Zohaib Iqbal, and Lionel Briand. "Random testing: Theoretical results and practical implications." Software Engineering, IEEE Transactions on 38, no. 2: 258-277, 2012.

[15] Myers, Glenford J., Corey Sandler, and Tom Badgett. The art of software testing. John Wiley & Sons, 2011.

[16] Ahmad, Mian Asbat. "New Strategies for Automated Random Testing." PhD diss., Ph. D. thesis, Department of Computer Science, The University of York, York, England, 2013.

[17] Yang, Qian, J. Jenny Li, and David M. Weiss. "A survey of coverage-based testing tools." The Computer Journal 52, no. 5: 589-597, 2009.

[18] Ciupa, Ilinca, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. "Experimental assessment of random testing for object-oriented software." In Proceedings of the 2007 international symposium on Software testing and analysis, pp. 84-94. ACM, 2007.

[19] Ciupa, Ilinca, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. "On the number and nature of faults found by random testing."Software Testing, Verification and Reliability 21, no. 1: 3-28, 2011.

[20] Arcuri, Andrea, Muhammad Zohaib Iqbal, and Lionel Briand. "Random testing: Theoretical results and practical implications." Software Engineering, IEEE Transactions on 38, no. 2: 258-277, 2012.

[21] Huang, Rubing, Huai Liu, Xiaodong Xie, and Jinfu Chen. "Enhancing mirror adaptive random testing through dynamic partitioning." *Information and Software Technology* 67 (2015): 13-29.

[22] Sabor, Korosh Koochekian, and Mehran Mohsenzadeh. "Adaptive random testing through dynamic partitioning by localization with distance and enlarged input domain." *International journal of innovative technology and exploring engineering* (2012): 1-5.

[23] Khalsa, Sunint Kaur, and Yvan Labiche. "An analysis and extension of Category partition testing for constrained systems." In *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*, pp. 55-56. IEEE, 2015.
[24] Bai, Xiaoying, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. "Ontology-based test modeling and partition testing of web services." In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pp. 465-472. IEEE, 2008.