ITRC
ICT Research Institute
(Iran Telecommunication Research Center)

Journal of Information and Communication Technology

Iranian Society of Information and Communication Technology

Volume 16, Issue 61-62, Autumn and Winter 2024, pp. 180-194

Investigating an Approach to Identify and Classify Mutants Based on the Characteristics of Mutants with Machine Learning Algorithms

Zeinab Asghari[1], Bahman Arasteh[2*], Abbas Koochari[3]

[1] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.
[2] Department of Computer Engineering, Tabriz Branch, Islamic Azad University, Tabriz, Iran.
[3] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran.

Received: 16 March 2024, Revised: 30 March 2024, Accepted: 02 May 2024
Paper type: Research

Abstract

Software mutation testing is one of the effective methods to evaluate code quality and detect hidden errors. However, this method faces challenges such as producing equivalent mutants and the process being time-consuming. The test cases designed for software testing must have the necessary sufficiency. For this purpose, the criterion of mutation test score is used. One of the main issues related to software mutation testing is the generation of equivalent mutants. Equivalent mutants are mutants that do not change the behavior of the program and have the same output as the original program. Identifying these mutants can make the mutation testing process time-consuming and costly. It is also possible that these mutants are mistakenly classified as hard to kill mutants. While hard to kill mutants can be rejected by changing the test items and strengthening them. In this paper, we present an efficient method for classifying program mutants to identify and separate equivalent mutations from hard to kill mutants. Using machine learning algorithms, we intend to optimize this process and improve the efficiency and effectiveness of the method. This is done by extracting different features from the mutants and using them to train machine learning models.

Keywords: Software Testing, Mutation Testing, Equivalent Mutants, Test Adequacy, Mutation Score.

* Corresponding Author's email: bahman.arasteh@istinye.edu.tr

دوفصلنامه
فناوری اطلاعات و ارتباطات ایران

سال شانزدهم، شماره‌های ۶۱ و ۶۲، پاییز و زمستان ۱۴۰۳، صفحه ۱۸۰ الی ۱۹۴

ارائه رویکردی برای شناسایی و طبقه‌بندی جهش‌ها براساس ویژگی‌های جهش‌ها با الگوریتم‌های یادگیری ماشین

زینب اصغری[1]، بهمن آراسته[2]*، عباس کوچاری[3]
[1]دانشجوی دکترای دانشگاه آزاد اسلامی واحد علوم و تحقیقات، تهران، ایران
[2]دانشیار دانشگاه آزاد اسلامی واحد تبریز، تبریز، ایران
[3]استادیار دانشگاه آزاد اسلامی واحد علوم و تحقیقات، تهران، ایران

تاریخ دریافت: ۱۴۰۲/۱۲/۲۶ تاریخ بازبینی: ۱۴۰۳/۰۱/۱۱ تاریخ پذیرش: ۱۴۰۳/۰۲/۱۳
نوع مقاله: پژوهشی

چکیده
آزمون جهش نرم‌افزار، یکی از روش‌های موثر برای ارزیابی کیفیت کد و تشخیص خطاهای پنهان است. با این حال، این روش با چالش‌هایی مانند تولید جهش‌های معادل و زمان‌بر بودن فرایند مواجه است. موردآزمون‌های طراحی شده برای آزمون نرم‌افزار باید از کفایت لازم برخوردار باشند. برای این کار از معیار امتیاز آزمون جهش استفاده می‌شود. یکی از مسائل اصلی مرتبط با آزمون جهش نرم‌افزار، تولید جهش‌های معادل است. جهش‌های معادل، جهش‌هایی هستند که باعث تغییر در رفتار برنامه نمی‌شوند و خروجی یکسانی با برنامه اصلی دارند. شناسایی این جهش‌ها می‌تواند فرایند آزمون جهش را زمان‌بر و هزینه‌بر کند. همچنین امکان دارد این جهش‌ها به اشتباه در دسته جهش‌های سرسخت قرار بگیرند. درحالی که جهش‌های سرسخت را می‌توان با تغییر موردآزمون‌ها و تقویت آنها پیدا کرد. در این مقاله، ما به بررسی روشی برای طبقه‌بندی جهش‌های برنامه برای شناسایی و جداسازی جهش‌های معادل از جهش‌های سرسخت می‌پردازیم. با استفاده از الگوریتم‌های یادگیری ماشین، ما قصد داریم که این فرایند را بهینه‌سازی کنیم و کارآمدی و کارایی روش را بهبود دهیم. این کار با استخراج ویژگی‌های مختلف از جهش‌ها و استفاده از آنها برای آموزش مدل‌های یادگیری ماشین انجام می‌شود.

کلیدواژگان: آزمون نرم‌افزار، آزمون جهش، جهش‌های معادل، کفایت آزمون، امتیاز جهش.

* رایانامه نویسنده مسئول: bahman.arasteh@istinye.edu.tr

۱- مقدمه

در دنیای امروزی پر از نرم‌افزارهای پیچیده، اطمینان از کیفیت و عملکرد صحیح آنها امری بسیار حیاتی است. هر گونه خطا یا نقص در یک نرم‌افزار می‌تواند عواقب جبران‌ناپذیری را به دنبال داشته باشد، از از دست دادن اعتماد کاربران و مشتریان گرفته تا خسارات مالی و حتی اجتماعی. در این راستا، آزمون جهش به عنوان یکی از روش‌های موثر برای ارزیابی کیفیت آزمون‌های نرم‌افزاری به ویژه برنامه‌های حیاتی و حساس مورد توجه قرار گرفته است [۱و۲].

آزمون جهش در اصل با تولید نسخه‌های متغیر از کد منبع (جهش‌های مصنوعی) و سپس اجرای آزمون‌ها روی این نسخه‌های جهش یافته، سعی در بررسی توانایی آزمون‌ها برای شناسایی خطاهای موجود دارد. برای اعمال این خطاهای هوشمند، یک سری عملگر بنام عملگرهای جهشی بکار گرفته می‌شوند تا با ایجاد تغییرات نحوی کوچک، سعی در تغییر برنامه اصلی داشته باشند. هر نسخه جدیدی از برنامه اصلی که ایجاد می‌شود، برنامه جهش نام دارد. به عنوان مثال، عبارت باینری x+y را در نظر بگیرید. این عبارت می‌تواند به عنوان یک عبارت جهشی مورد آزمون قرار بگیرد. جدول ۱، چند نمونه عملگر جهشی را نشان می‌دهد.

جدول ۱. عملگرهای جهشی پرکاربرد [۷]

نام عملگر	تعریف عملگر
AODS: Shortcut Arithmetic Operator Deletion	{(x, remove(x)) | x ∈ {++, --}}
AODU: Unary Arithmetic Operator Deletion	{(-v, v)}
AOIS: Shortcut Arithmetic Operator Insertion	{(v, --v) , (v, v--), (v, ++v), (v, v++)}
AOIU: Unary Arithmetic Operator Insertion	{(v, -v)}
AORB: Binary Arithmetic Operator Replacement	{(x,y) | x,y ∈ {+, -, *, /, % } ∧ x ≠ y}
AORS: Shortcut Arithmetic Operator Replacement	{(x,y) | x,y ∈ {++, --} ∧ x ≠ y}
ASRS: Shortcut Assignment Operator Replacement	{(x,y) | x,y ∈ {+=, -=, *=, /=, %=} ∧ x ≠ y}
CDL: Constant DeLetion	{(op c, remove(op c)) | op ∈ {+, -, *, /, %, >, >=, <, <=}}
COD: Conditional Operator Deletion	{(!(e), e) | e ∈ {if(e), while(e), for(s; e; s)}}
COI: Conditional Operator Insertion	{(e, !(e)) | e ∈ {if(e), while(e), for(s; e; s)}}
COR: Conditional Operator Replacement	{(x,y) | x,y ∈ {&&, ||, ^} ∧ x ≠ y}
LOI: Logical Operator Insertion	{(v, ~v)}
ODL: Operator DeLetion	{(v op, remove(v op)), (op v, remove(op v)) | op ∈ {+, -, *, /, %, <, <=, >, >=}}, {(v++, v), (v--, v) , (--v, v) , (++v, v) | op ∈ {++, --}}
ROR: Relational Operator Replacement	{(x,y) | x,y ∈ {>, >=, <, <=, ==, !=} ∧ x ≠ y}
SDL: Statement DeLetion	{(s, remove(s))}
VDL: Variable DeLetion	{(v [op], remove(v [op])) | op ∈ {+, -, *, /, %, ++, --, <, <=, >, >=}

عملگر جایگزین حسابی (AORB)، عملگری است که جهش‌های زیر را می‌تواند تولید کند: x+y → x-y , x+y → x*y , x+y → x / y , x+y → x%y.

مثال دیگر برای عملگرهای جهشی، عملگری بنام VDL است که در عبارت موجود، یک عملگر را حذف می‌کند. نمونه‌ای از عملیات این عملگر به شرح زیر است: x+y → x , x+y → y. اگر مورد آزمون‌های طراحی شده، توانایی شناسایی این نسخه‌های جهش یافته را نداشته باشند، به این معنی است که آن آزمون‌ها ناکارآمد هستند و نیاز دارند تا بهبود داده شده و قوی‌تر شوند. از این رو، تولید جهش‌های معادل (جهش‌هایی که عملکرد برنامه را تغییر نمی‌دهند اما توسط آزمون‌ها تشخیص داده می‌شوند) و جداسازی آنها از جهش‌های سرسخت، یکی از چالش‌های اساسی در این زمینه محسوب می‌شود [۳و۴] .

در این مقاله، به دنبال ارائه رویکردی نوین برای شناسایی و جداسازی جهش‌های معادل از جهش‌های سرسخت هستیم. برای این منظور، به کمک الگوریتم‌های یادگیری ماشین و با استفاده از ویژگی‌های اصلی و تاثیرگذار جهش‌ها، سعی در توسعه یک مدل دقیق و کارآمد برای طبقه‌بندی جهش‌ها خواهیم داشت. این رویکرد، بهبود قابل توجهی در کارآمدی و کارآیی فرآیند آزمون جهش نرم‌افزار دارد.

۲- کارهای پیشین

در این بخش، به بررسی جدیدترین روش‌های شناسایی جهش‌های معادل در آزمون جهش نرم‌افزار پرداخته شده است. این روش‌ها به شش دسته کلی تقسیم شده‌اند که هرکدام به اختصار توضیح داده شده است.

۲-۱- روش مبتنی بر تحلیل پویا

در این روش، نرم‌افزار به صورت پویا اجرا می‌شود و رفتار آن در طول اجرا مشاهده می‌شود. تحلیل پویا از جریان کنترل، استفاده از حافظه، ورودی‌ها و خروجی‌ها و... برای شناسایی جهش‌های معادل استفاده می‌کند [۵]. به عنوان مثال، اگر یک جهش معادل توسط یک آزمون تشخیص داده نشود و در زمان اجرا و تحلیل پویا، اثرات مشابهی با جهش اصلی داشته باشد، ممکن است نشان‌دهنده معادل بودن آن جهش باشد. این روش برای شناسایی جهش‌های معادل از تحلیل جریان کنترل، استفاده از حافظه، ورودی‌ها و خروجی‌ها و دیگر جنبه‌های پویای نرم‌افزار استفاده می‌کند [۶]. با تحلیل پویای رفتار نرم‌افزار در زمان اجرا، می‌توان اثرات جهش‌های معادل را با جهش‌های اصلی مقایسه کرد. به عنوان مثال، اگر یک جهش معادل

توسط یک آزمون شناسایی نشود، اما در زمان اجرا و تحلیل پویا اثرات مشابهی با جهش اصلی داشته باشد، این ممکن است نشان‌دهنده معادل بودن آن باشد. به عبارت دیگر، اگر جهش معادل در زمان اجرا همان رفتار یا تاثیرات مشابهی که جهش اصلی ایجاد می‌کند، داشته باشد، می‌توان آن را به عنوان جهش معادل شناسایی کرد. میزان انتشار خطا توسط دستورالعمل‌ها نیز برای شناسایی جهش‌های معادل، موثر است [۷].

۲-۲- روش مبتنی بر مدل‌سازی فرآیند

این روش بر اساس مدل‌سازی دقیق فرآیند تولید و اجرای جهش‌ها استوار است[۸]. با تحلیل این فرآیند، جهش‌های معادل را می‌توان با دقت شناسایی کرد. به عنوان مثال، با مدل‌سازی جزئیاتی از فرآیند تولید جهش‌ها و شناسایی الگوهای خاص، می‌توان جهش‌های معادل را پیدا کرد این مدل‌سازی ممکن است شامل مراحل مختلف فرآیند تولید جهش‌ها، از جمله ایجاد جهش‌ها، اعمال جهش‌ها بر روی کد، اجرای آزمون‌ها، و تحلیل نتایج آزمون‌ها باشد[۹]. با تحلیل این فرآیند به دقت، الگوهایی که به وجود جهش‌های معادل اشاره می‌کنند، شناسایی می‌شوند. به عنوان مثال، اگر در مدل‌سازی فرآیند تولید جهش‌ها الگوهایی مشخص برای جهش‌های معادل شناسایی شوند، می‌توان از این الگوها برای شناسایی جهش‌های معادل استفاده کرد [۱۰]. این الگوها ممکن است شامل ترتیب اجرای جهش‌ها، نحوه تولید و اعمال جهش‌ها، و تاثیر آنها بر رفتار برنامه باشد. با استفاده از این روش، می‌توان به طور دقیق و کارآمد جهش‌های معادل را شناسایی کرده و از این طریق به بهبود آزمون جهش و افزایش کیفیت و قابلیت اطمینان برنامه کمک کرد [۱۱].

۲-۳- روش مبتنی بر تحلیل خروجی آزمون

این روش بر اساس تحلیل و مقایسه نتایج آزمون‌ها برای جهش‌های معادل و جهش‌های اصلی تمرکز دارد. با تحلیل این تفاوت‌ها، می‌توان جهش‌های معادل را شناسایی کرد [۱۲]. برای مثال، اگر خروجی آزمون برای یک جهش معادل با خروجی آزمون برای جهش اصلی تفاوتی نداشته باشد، ممکن است این جهش معادل باشد. شود. به عنوان مثال، اگر خروجی یک آزمون برای یک جهش معادل با خروجی آزمون مربوط به جهش اصلی تفاوتی نداشته باشد، این ممکن است نشان‌دهنده جهش معادل باشد. در این صورت، معادل بودن دو جهش با توجه به نتایج آزمون مشخص می‌شود. در این روش، ممکن است از معیارهای مختلفی برای تحلیل نتایج آزمون استفاده شود، از جمله مقایسه خروجی‌های آزمون، مقایسه زمان اجرا، تحلیل تغییرات در رفتار برنامه و سایر معیارهای مشابه. این تحلیل‌ها می‌تواند به شناسایی جهش‌های معادل کمک کند و از طریق آن‌ها می‌توان به بهبود آزمون جهش و افزایش صحت و کارآیی آن دست یافت [۱۳،۱۴].

۲-۴- روش مبتنی بر تحلیل کد

این روش بر اساس تحلیل ساختار و کد منبع نرم‌افزار استوار است [۱۵]. با تحلیل و مقایسه کدهای مربوط به جهش‌های معادل و جهش‌های اصلی، می‌توان جهش‌های معادل را شناسایی کرد. به عنوان مثال، اگر کدهای منبع مربوط به دو جهش با یکدیگر تطابق داشته باشند، این ممکن است نشان‌دهنده معادل بودن آنها باشد .این تطابق ممکن است به مواردی اشاره کند که علاوه بر تغییراتی که جهش‌ها ایجاد کرده‌اند، تفاوتی در کد اصلی و کد معادل وجود ندارد [۱۶]. علاوه بر تطابق کدها، در این روش می‌توان به دنبال تحلیل تفاوت‌های جزئی در کدها بود که ممکن است نشان‌دهنده جهش‌های معادل باشد. این تفاوت‌ها می‌توانند شامل اختلاف در نحوه پیاده‌سازی یک قسمت از کد، محل یا روش فراخوانی یک تابع، یا سایر جزئیات فنی باشند [۱۷،۱۸].

۲-۵- روش مبتنی بر تحلیل تاریخچه اجرا

این روش بر اساس تحلیل تاریخچه اجرای برنامه و نتایج آزمون‌ها استوار است. با مشاهده و تحلیل تاریخچه اجرای برنامه و تفاوت‌های آن بین جهش‌های معادل و جهش‌های اصلی، می‌توان این جهش‌ها را شناسایی کرد. به طور کلی، این روش شامل مراحل زیر می‌شود:

- تهیه آزمون‌ها: ابتدا، آزمون‌هایی طراحی می‌شود که برای بررسی عملکرد و صحت برنامه مورد استفاده قرار می‌گیرند. این آزمون‌ها می‌توانند از انواع مختلفی باشند که بسته به نیازهای آزمون معین می‌شوند.
- اجرای آزمون‌ها: در این مرحله، آزمون‌ها روی برنامه اجرا می‌شوند و نتایج اجرا ثبت می‌شود. این نتایج می‌توانند شامل خروجی‌های برنامه، اطلاعات لاگ، متریک‌های عملکرد، و غیره باشند.
- تحلیل نتایج: در این مرحله، نتایج اجرا و آزمون‌ها مورد بررسی قرار می‌گیرند. با مقایسه نتایج مربوط به جهش‌های اصلی با نتایج مربوط به جهش‌های معادل، تفاوت‌ها و الگوهای خاصی که ممکن است به نشان دادن وجود جهش‌های معادل بیانجامد، شناسایی می‌شود.
- شناسایی جهش‌های معادل: با توجه به تفاوت‌ها و الگوهایی که در نتایج مشاهده می‌شود، جهش‌های معادل ممکن است شناسایی شوند. این تفاوت‌ها ممکن است شامل تغییرات در

خروجی برنامه، الگوهای خاص در تاریخچه اجرا، یا میزان تاثیر جهش بر عملکرد برنامه باشد [۱۹،۲۰].

۲-۶- روش مبتنی بر تحلیل ویژگی‌های جهش

روش‌های مبتنی بر تحلیل ویژگی‌های جهش به منظور شناسایی جهش‌های معادل، از تحلیل و مقایسه ویژگی‌های خاصی که جهش‌ها دارند، استفاده می‌کنند[۲۱]. این ویژگی‌ها می‌توانند ویژگی‌های کد، ساختار، یا رفتاری از جهش‌ها باشند. این روش‌ها بر اساس فرضیه‌ای عمل می‌کنند که جهش‌های معادل باید ویژگی‌های خاصی را با جهش‌های اصلی به اشتراک داشته باشند. این روش بر اساس تحلیل ویژگی‌های جهش و تفاوت‌های آنها با جهش‌های اصلی تمرکز دارد[۲۲]. با مشاهده و تحلیل ویژگی‌های خاصی که جهش‌های معادل از جهش‌های اصلی دارند، می‌توان این جهش‌ها را شناسایی کرد[۲۳]. به عنوان مثال، اگر یک جهش معادل ویژگی‌های خاصی مانند تغییرات محلی در کد را نشان دهد، ممکن است معادل بودن آن را نشان دهد[۱۵].

۳- روش پیشنهادی

در بخش پیشین به روش‌های مختلف شناسایی و حذف جهش‌های معادل پرداخته شده است. در این بخش، براساس تحقیق انجام گرفته در [۷]، تحلیل ایستا و پویای کد برنامه انجام گرفته و در نتیجه آن، دستورالعمل‌هایی که نسبت به نرخ انتشار خطا، اثربخشی کمتری دارند شناسایی شده است. دستورالعمل‌هایی که نرخ توزیع خطای کمتری دارند، با احتمال بسیار بالایی جزء جهش‌های معادل طبقه‌بندی شده‌اند. یعنی هرچقدر تاثیرپذاری دستورالعمل کمتر باشد، جهش‌های تزریق شده به آن دستورالعمل‌ها، معادل و بلااثر خواهند بود. برطبق این نتیجه گیری، ما در این مقاله به بررسی جهش‌های ایجاد شده در دستورالعمل‌های سطح پایین پرداخته‌ایم. به این صورت که دستورالعمل‌های سطح پایین، احتمال اینکه جهش تزریق شده به آن از نوع بلااثر باشد مورد هدف این مقاله قرار گرفته است.

در این روش، ابتدا تمامی دستورالعمل‌های سطح پایین موجود در برنامه‌های محک شناسایی شده و سپس تمامی جهش‌هایی که برروی آن دستورالعمل‌ها اعمال شده‌اند، مورد بررسی قرار گرفته‌اند. شناساسس این نوع دستورالعمل‌ها با کمک الگوریتم‌های یادگیری ماشین انجام شده است. هنگامی که جهشی بروری دستورالعملی اعمال می‌شود چند حالت ممکن است اتفاق بیفتد: اول این که ممکن است آن جهش توسط موردآزمون‌های طراحی شده شناسایی شود که به این حالت می‌گوییم جهش کشته شده است. حالت دوم زمانی رخ می‌دهد که، جهش تزریق شده توسط هیچکدام از مورد آزمون‌های موجود در بستر آزمون طراحی شده شناسایی نشود. به این حالت گوییم جهش زنده مانده است. بر طبق رابطه ۱ که امتیاز جهش را نشان می‌دهد، هدف از آزمون جهش، بیشینه کردن مقدار امتیاز جهش با شناسایی تمام جهش‌های قابل کشتن و به حداقل رساندن جهش‌های معادل است.

$$Mutation\ Score = \frac{Kill\ Mutants}{All\ Mutants - Equivalent\ Mutants} \qquad (۱)$$

در ابتدا لازم است یک سری تعاریف از مشخصه‌های جهش‌ها را توضیح دهیم.

Mutant Severity: شدت جهش به میزان پتانسیل تأثیر یا جدیت یک جهش بر رفتار یا عملکرد یک نرم‌افزار اشاره دارد. این موضوع، درجه اهمیت یک جهش را از نظر قابلیت ایجاد خطاها یا اشتباهات در کد بررسی می‌کند. شدت یک جهش می‌تواند بر اساس عواملی مانند نوع جهش، مکان جهش و زمینه کد مورد نظر متغیر باشد. در زیر چند نکته اساسی در تعریف شدت جهش آمده است:

نوع جهش: اپراتورهای جهش مختلف درجات متفاوتی از شدت دارند. به عنوان مثال، جهش‌هایی که شامل تغییر اپراتورهای حسابی هستند، احتمالاً شدت کمتری نسبت به جهش‌هایی دارند که شامل تغییرات در بیانیه‌های شرطی یا ساختارهای کنترل حلقه می‌شوند.

تأثیر بر رفتار برنامه: جهش‌هایی که بر رفتار برنامه تأثیر معنی‌داری دارند، مانند تغییر جریان منطقی یا ایجاد خطاهای معنی‌دار، نسبت به جهش‌هایی با تأثیر کمتر، شدت بیشتری دارند.

پتانسیل ایجاد خطا: جهش‌هایی که احتمال بالاتری برای وارد کردن خطاها یا اشتباهات به کد دارند، معمولاً به عنوان جهش‌هایی با شدت بیشتر محسوب می‌شوند. این شامل جهش‌هایی است که نقاط تصمیم‌گیری اصلی را تغییر می‌دهند یا سازوکارهای دستورات اشتباه را تغییر می‌دهند.

سهولت شناسایی: جهش‌هایی که با استفاده از آزمون‌ها یا مرور کد به راحتی قابل شناسایی نیستند، ممکن است به عنوان جهش‌هایی با شدت بالاتر محسوب شوند، زیرا احتمالاً در طول اجرا یا برخورد با ورودی‌های غیرمنتظره به شکل اشکال به نمایش خواهند گذاشت.

از آنجایی که رابطه استانداردی برای محاسبه شدت جهش وجود ندارد، مفهوم شدت جهش اغلب وابسته به زمینه است. این امر، شامل ارزیابی عوامل مختلف مرتبط با تأثیر بالقوه یا جدیت یک جهش بر روی سیستم نرم‌افزاری است. با این حال، ما می‌توانیم یک رابطه مفهومی را تعریف کنیم که این عوامل را در خود جای دهد. شایان ذکر است که این رابطه ممکن است براساس زمینه و الزامات

خاص نرم‌افزار مورد آزمایش، نیاز به تغییراتی داشته باشد. رابطه پیشنهادی برای شدت جهش در اینجا آمده است:

$$MS= \sum_{i=1}^{n}(w_i \times f_i) \quad (۲)$$

در رابطه ۲، n، تعداد عوامل در نظر گرفته شده در ارزیابی شدت هر جهش است. w_i، وزن‌های اختصاص داده شده به هر عامل است تا اهمیت نسبی آنها را منعکس کند. مجموع این وزن‌ها باید در نهایت، ۱ باشد. f_i امتیازاتی هستند که به هر عامل بر اساس سطح شدت آن اختصاص داده می‌شود. فاکتور f_i در رابطه در نظر گرفته شده، شامل موارد زیر است:

- Functional Impact: امتیازی که میزان تأثیر جهش بر عملکرد یا رفتار مورد نظر برنامه را نشان می‌دهد.
- Fault Detection Difficulty: امتیازی که منعکس کننده دشواری تشخیص خطای معرفی شده توسط جهش با استفاده از مجموعه آزمایشی است.
- Potential Consequences: امتیازی که نشان‌دهنده پیامدهای بالقوه خطا، مانند خرابی سیستم، خرابی داده‌ها یا آسیب‌پذیری‌های امنیتی است.
- Frequency of Execution: امتیازی که تعداد دفعات اجرای کد جهش را در طول اجرای برنامه معمولی نشان می‌دهد.

وزن‌های اختصاص داده شده به هر عامل باید اهمیت نسبی آنها را در فرآیند ارزیابی شدت جهش منعکس کند. این وزن‌ها را می‌توان بر اساس تحلیل تجربی تعیین کرد. با جمع‌بندی امتیازهای وزنی این عوامل، رابطه یک نمایش کمی از شدت جهش ارائه می‌کند و به ما این امکان را می‌دهد تا جهش‌های تولید شده را بر اساس سطح شدت جهش‌ها اولویت‌بندی کنیم.

در اینجا یک مثال عددی برای رابطه پیشنهادی برای شدت جهش نشان می‌دهیم. برای سادگی، اجازه دهید سناریویی را با سه عامل در نظر بگیریم: تأثیر عملکردی (FI)، مشکل تشخیص خطا (FDD) و پیامدهای بالقوه (PC). فرض کنید برای هر عامل به صورت زیر وزن تعیین می‌کنیم:

w_{FI}=0.4 (40%), w_{FDD}=0.3 (30%), w_{PC}=0.3 (30%)

همچنین فرض کنیم که هر عامل را در مقیاس ۱ تا ۱۰ درجه‌بندی می‌کنیم که ۱۰ شدیدترین آنها است:

Functional Impact (FI): 8
Fault Detection Difficulty (FDD): 5
Potential Consequences (PC): 7

حال شدت جهش (MS) را با استفاده از رابطه ۲ محاسبه کنیم:

$MS=(w_{FI}\times f_{FI})+(w_{FDD}\times f_{FDD})+(w_{PC}\times f_{PC})$
$MS=(0.4\times 8)+(0.3\times 5)+(0.3\times 7)$
$MS=(3.2)+(1.5)+(2.1)$
$MS=6.8$

در این مثال، امتیاز شدت جهش ۶,۸ است. این امتیاز نشان‌دهندهٔ شدت کلی جهش بر اساس ترکیب وزنی عوامل است. نمرات شدت بالاتر نشان دهنده جهش‌هایی است که شدیدتر هستند و به طور بالقوه تأثیر بیشتری بر روی برنامه دارند.

Mutant Diversity: تفاوت جهش، شامل ارزیابی تفاوت آن با سایر جهش‌های مجموعه با استفاده از فاکتور فاصله است. رابطه ۳، رابطه تفاوت جهش را نشان می‌دهد. در اینجا نحوه محاسبه تنوع برای هر جهش آمده است:

متریک فاصله: متریک فاصله، تفاوت بین جهش‌ها را کمیت می‌کند و می‌تواند بر اساس عوامل مختلفی مانند تعداد عبارات جهش یافته، نوع عملگر جهش اعمال شده یا هر عامل مرتبط دیگری باشد. متریک فاصله به صورت *d(mi,mj)* نشان داده می‌شود که در آن، m_i و m_j دو جهش هستند و d، فاصله بین این دو جهش را نشان می‌دهد. برای هر جهش m_i، تفاوت جهش با تمامی جهش‌های دیگر موجود را با کمک متریک فاصله، محاسبه می‌کنیم. تفاوت جهش m_i را با $div(m_i)$ نشان می‌دهیم و به صورت زیر محاسبه می‌شود:

$$div(m_i) = \sum_{j\neq i}(m_i \text{ و } m_j) \quad (۳)$$

رابطه ۳، مجموع فواصل بین جهش m_i و تمام جهش‌های موجود در مجموعه جهش‌ها را محاسبه می‌کند.

با یک مثال، فاکتور تفاوت جهش را محاسبه می‌کنیم. فرض کنید یک مجموعه جهش برای یک دستورالعمل مفروض به صورت $\{m_1,m_2,m_3\}$ داریم. می‌خواهیم، معیار تفاوت جهش را برای هر جهش براساس فاکتور فاصله بین جهش‌ها محاسبه کنیم. در اینجا مجموعه‌ای فرضی از جهش‌ها به همراه تعداد عبارات جهشی برای هر جهش آمده است:

m_1: در ۵ عبارت جهشی ظاهر شده است
m_2: در ۳ عبارت جهشی ظاهر شده است
m_3: در ۷ عبارت جهشی ظاهر شده است

اکنون، تفاوت جهشی را برای هر جهش محاسبه می‌کنیم:

For m1:
Diversity of m1 with respect to m2: |5−3|=|5−3|=2
Diversity of m1 with respect to m3: |5−7|=|5−7|=2
Total diversity for m1: 2+2=42+2=4
For m2:
Diversity of m2 with respect to m1: |3−5|=|3−5|=2.
Diversity of m2 with respect to m3: |3−7|=4|3−7|=4
Total diversity for m2: 2+4=62+4=6
For m3:
Diversity of m3 with respect to m1: |7−5|=2|7−5|=2
Diversity of m3 with respect to m2: |7−3|=4|7−3|=4
Total diversity for m3: 2+4=62+4=6.

بنابراین، بر اساس متریک فاصله، تفاوت جهش، برای هر جهش به این صورت خواهد بود:

Diversity(m1)= 4

Diversity(m2)= 6
Diversity(m3): 6

Mutant Coverage: پوشش جهش، براساس رابطه ۴ محاسبه می‌شود.

$$\text{Mutant Coverage for One Mutant}=(\text{Total Number of Test cases in the Test Suite}/\text{Number of Test Cases that Detect the Mutant})\times 100 \qquad (۴)$$

به عنوان مثال، اگر مجموعه آزمایشی شما از ۵۰ مورد آزمون تشکیل شده باشد و ۱۰ مورد از این موارد آزمایش، جهش خاصی مانند m_i را که مدنظر دارید شناسایی کند، پوشش جهش برای آن جهش خواهد بود :

$$\text{Mutant Coverage for } m_i=(10/50)/100\%$$

مقدار ۲۰% به این معنی است که ۲۰% از موارد آزمون در مجموعه آزمایشی، جهش خاص مدنظر را تشخیص می‌دهند.

Mutant Visibility: ارزیابی نمایان بودن یک جهش، به معنای این است که چقدر یک جهش قابل مشاهده یا قابل تشخیص توسط مجموعه آزمون است. در واقع، نمایان بودن یک جهش نشان دهنده این است که آیا تغییر معنایی که توسط جهش ایجاد شده است، توسط مجموعه آزمون تشخیص داده می‌شود یا خیر. رابطه ۵، نمایان بودن جهش را نشان می‌دهد.

$$MV=(W_{Scope}\times S_{Scope})+(W_{Change}\times S_{Change})+(W_{Readability}\times S_{Readability}) \qquad (۵)$$

در رابطه ۵، میزان رویت یک جهش را با استفاده از یک مثال ساده نشان می‌دهیم. فرض کنید ما در حال ارزیابی نمایان بودن یک جهش هستیم که شامل تغییر نام متغیر در یک تابع است. ما سه عامل را در نظر خواهیم گرفت: محدوده جهش، ماهیت تغییر و خوانایی کد. محدوده جهش (Scope)، دامنه جهش را ۵ از ۱۰ رتبه‌بندی می‌کنیم. این تغییر تنها بر یک متغیر منفرد در یک تابع تأثیر می‌گذارد که در مقایسه با تغییر کل تابع یا کلاس، دامنه نسبتاً کوچکی است. ماهیت تغییر (change) را ۷ از ۱۰ ارزیابی می‌کنیم. تغییر نام متغیر یک تغییر نسبتاً قابل توجه است، زیرا نحو کد برنامه را بدون تغییر اساسی منطق یا رفتار آن تغییر می‌دهد. خوانایی کد (Readability) را ۸ از ۱۰ رتبه‌بندی می‌کنیم. کد جهش یافته واضح و قابل درک باقی می‌ماند، زیرا تغییر فقط شامل یک نام متغیر است و پیچیدگی یا سردرگمی ایجاد نمی‌کند. در مرحله بعد، به هر عامل وزنی اختصاص می‌دهیم تا اهمیت نسبی آنها را منعکس کند. بیایید وزن‌های مساوی را برای سادگی فرض کنیم:

$W_{Scope}=0.33$ (33%)
$W_{Change}=0.33$ (33%)
$W_{Readability}=0.33$ (33%)

اکنون می‌توانیم میزان نمایان بودن جهش را با استفاده از رابطه ۵ محاسبه کنیم:

$$MV=(W_{Scope}\times s_{Scope})+(W_{Change}\times S_{Change})+(W_{Readability}\times S_{Readability})$$
$$=(0.33\times 5)+(0.33\times 7)+(0.33\times 8) = (0.33\times 5) + (0.33\times 7) + (0.33\times 8)$$
$$=(1.65)+(2.31)+(2.64) = (1.65)+(2.31)+(2.64)$$
$$\approx 6.60.$$

Mutant Complexity:

در آزمون جهش، پیچیدگی جهش به سطح دشواری یا پیچیدگی مرتبط با ایجاد جهش در کد برنامه اشاره دارد. آزمون جهش، تکنیکی است که برای ارزیابی کیفیت مجموعه‌های آزمایشی با وارد کردن تغییرات کوچک به نام جهش در کد منبع و سپس اجرای مجموعه آزمایشی برای مشاهده اینکه آیا هر یک از جهش‌ها شناسایی شده‌اند، استفاده می‌شود. تکه کد محاسبه مساحت زیر را در نظر بگیرید.

```
public static double calculateArea (double length, double width) {
    return length * width;
}
```

حال، فرض کنید می‌خواهیم جهش‌ها را برای اهداف آزمایش جهش وارد این تابع کنیم. جهش‌ها با ایجاد تغییرات کوچک در کد اصلی ایجاد می‌شوند. در مثال خود، جهش‌ها را در نقاط مختلف تابع محاسبه مساحت معرفی می‌کنیم و پیچیدگی آن‌ها را با استفاده از معادله ارائه شده در رابطه ۶ مطرح می‌کنیم:

$$MCom=w_I\times I+ w_I\times C+\times w_I\times D \qquad (۶)$$

برای سادگی، سه عامل را در نظر بگیریم که در پیچیدگی جهش نقش دارند:

- *Importance of the instruction (I)*: دستورالعمل برای عملکرد کلی برنامه چقدر مهم است؟
- Complexity of the instruction (C): دستورالعمل چقدر پیچیده است؟
- *Dependency level of the instruction (D)*: دستورالعمل چقدر به قسمت‌های دیگر کد وابسته است؟ حال به این عوامل وزن اختصاص می‌دهیم:

$w1=0.5$ for importance,
$w2=0.3$ for complexity,
$w3=0.2$ for dependency level

در اینجا نحوه محاسبه پیچیدگی جهش (*MCom*) برای هر جهش آمده است:

برای تابع اصلی I=1. (تابع اصلی برای محاسبه ضروری است.)، C=1 (عملیات ضرب ساده) و D=1 (بدون وابستگی به قسمت‌های دیگر کد). تابع اصلی دارای پیچیدگی جهش ۱ است.

جهش شماره ۱: تغییر عملیات ضرب به جمع

I=0.8: (هنوز مهم است اما کمتر از عملکرد اصلی مهم است.)

C=1: عملیات حسابی ساده

D=1: بدون وابستگی

$MCom = w_I \times I + w_I \times C + \times w_I \times D$

$MCom = 0.5\times0.8+0.3\times1+0.2\times1=0.8\times0.5+1\times0.3+1\times0.2 =0.5+0.3+0.2=1$

جهش شماره ۱، دارای پیچیدگی جهش ۱ است، زیرا تغییری را معرفی می‌کند که بر عملکرد حیاتی تابع تأثیر می‌گذارد.

جهش شماره ۲: اضافه کردن یک عبارت شرطی

I=0.7: (بسیار مهم، اما کمتر از نسخه اصلی)

C=2: (افزایش پیچیدگی به دلیل بیان شرطی)

D=1: بدون وابستگی

$MCom=0.5\times0.7+0.3\times2+0.2\times1=0.7\times0.5+2\times0.3+1\times0.2 =0.35+0.6+0.2=1.15$

جهش شماره ۲ دارای پیچیدگی جهش یافته ۱,۱۵ است که نشان دهنده افزایش پیچیدگی معرفی شده توسط عبارت شرطی است.

Mutant Impact:

تاثیر جهش به میزان اثرگذاری جهش در کد برنامه برروی رفتار برنامه اشاره دارد. هر جهش نشان دهنده تغییر کوچکی است که در کد منبع ایجاد شده است و تأثیر این تغییر با مشاهده چگونگی تأثیر آن بر نتیجه مجموعه مورد آزمون‌ها ارزیابی می‌شود. در برنامه زیر، یک جهش با تغییر عملگر ضرب به جمع ایجاد می‌کنیم. حال میزان تاثیر جهش در رابطه ۷ رل محاسبه می‌کنیم.

```
public class FactorialCalculator {
   public int calculateFactorial(int n) {
      if (n == 0)
         return 1;
      else
         return n * calculateFactorial(n - 1); mutant injection→ return n + calculateFactorial(n - 1);
}
}
```

$$MI\ (\text{Mutant Impact}) = FI\times(1-CD)\times CC \qquad (۷)$$

به طوری که:

MI is the mutation impact.
FI is the functionality impact
CD is the fault detection difficulty.
CC is the code coverage affected by the mutation.

حال برای مثال مفروض، مقادیری را به این فاکتورها اختصاص دهیم:

FI=1

به این معناست که جهش به‌طور کامل، عملکرد روش را تغییر می‌دهد

CD=0.8

به این معنا که تشخیص خطای ایجاد شده توسط جهش به راحتی انجام نمی‌پذیرد.

CC=20%

به این معناست که جهش تنها بخش کوچکی از کد را تحت تأثیر قرار می‌دهد، زیرا محدود به فراخوانی بازگشتی است.

حالا این مقادیر را به فرمول اضافه می‌کنیم:

$MI=1\times(1-0.8)\times0.2=0.2\times0.2=0.04$

در این مثال، تاثیر جهش ۰,۰۴ است. با توجه به اینکه جهش به طور قابل توجهی رفتار روش را تغییر می‌دهد، نشان‌دهندهٔ تأثیر متوسط است، اما تنها بخش کوچکی از کد را تحت تأثیر قرار می‌دهد و ممکن است شناسایی از طریق آزمون تا حدودی چالش برانگیز باشد.

Mutant Coupling:

به میزان وابستگی متقابل یا تعامل بین جهش‌ها در یک برنامه نرم‌افزاری در طول آزمون جهش اشاره دارد. این فاکتور، تغییرات ایجاد شده در یک جهش که بر رفتار، تشخیص یا اثربخشی سایر جهش یافته‌ها تأثیر می‌گذارد را اندازه می‌گیرد. به عبارت دیگر، میزان همبستگی جهش‌ها با یکدیگر را از نظر تأثیر آنها در آزمایش ارزیابی می‌کند. بیایید Mutant coupling را با یک مثال جاوا نشان دهیم و سپس یک فرمول مفهومی برای محاسبه آن ارائه کنیم.

یک کلاس جاوا ساده را در نظر بگیرید که روشی را برای محاسبه مربع یک عدد پیاده سازی می‌کند:

```
public class SquareCalculator {
   public int square(int x) {
      return x * x;
   }
}
```

حال، فرض کنید دو جهش را به این روش معرفی می‌کنیم:

Mutant 1: عمل ضرب (×) را به جمع (+) تغییر می‌دهد.

Mutant 2: به جای ضرب، یک عمل تفریق (-) را معرفی می‌کند. در اینجا نحوه اعمال این جهش‌ها آمده است:

Mutants 1

```
public int square(int x) {
   return x + x;
}
```

Mutant 2:

```
public int square(int x) {
   return x - x;
}
```

حال، بیایید تأثیر تشخیص یک جهش بر دیگری را تجزیه و تحلیل کنیم. فرض کنید مجموعه آزمایشی ما برای تشخیص انحرافات از رفتار صحیح مربع کردن یک عدد طراحی شده است. اگر مجموعه آزمایشی به طور موثر جهش ۱ (عملیات جمع) را تشخیص دهد اما نتواند جهش ۲ (عملیات تفریق) را شناسایی کند، ما جفت جهش را مشاهده می‌کنیم. در این مورد، تشخیص جهش ۱ بر تشخیص یا

اثربخشی جهش ۲ تأثیر می‌گذارد، که نشان دهنده درجه‌ای از وابستگی متقابل بین جهش‌ها است. از آنجایی که فرمول استانداردی برای محاسبه Mutant Coupling وجود ندارد، می‌توانیم آن را به صورت مفهومی به صورت رابطه ۸ نمایش دهید:

$$MC=\frac{N_{\text{detected_together}}}{N_{\text{detected_}alone}} \quad (۸)$$

بطوری که:

$N_{\text{detected_together}}$، تعداد جهش‌هایی است که با هم شناسایی شده‌اند (یعنی هر دو جهش توسط یک مورد آزمایشی یا مجموعه‌ای از موارد آزمایشی شناسایی شده‌اند).

$N_{\text{detected_alone}}$، تعداد جهش‌هایی است که به تنهایی شناسایی می‌شوند (یعنی هر جهش توسط موارد آزمایش جداگانه شناسایی می‌شود). فرض کنید ما مجموعه‌ای از ۱۰ جهش (M1 تا M10) داریم و علاقه مند به محاسبه مقدار فاکتور Coupling برای M1 هستیم. جهش M1 توسط مورد آزمون A شناسایی می‌شود. مورد آزمون A همچنین جهش M2، M3 و M4 را علاوه بر M1 شناسایی می‌کند. مجموعه جهش‌ها در کل از ۱۰ جهش تشکیل شده است. حالا بیایید مقدار جفت شدن جهش M1 را محاسبه کنیم:

تعداد جهش‌های شناسایی شده توسط مورد آزمون A شامل M1: ۴

تعداد کل جهش یافته‌ها در مجموعه جهش: ۱۰

Amount of Coupling=Total number of mutants/Number of mutants detected by Test Case A×100%

Amount of Coupling=4/10*100%=40%

بنابراین، مقدار coupling برای جهش M1 برابر ۴۰٪ است. این نشان می‌دهد که ۴۰٪ از جهش‌ها در مجموعه جهش توسط همان مورد آزمایشی شناسایی می‌شوند که جهش M1 را شناسایی می‌کند.

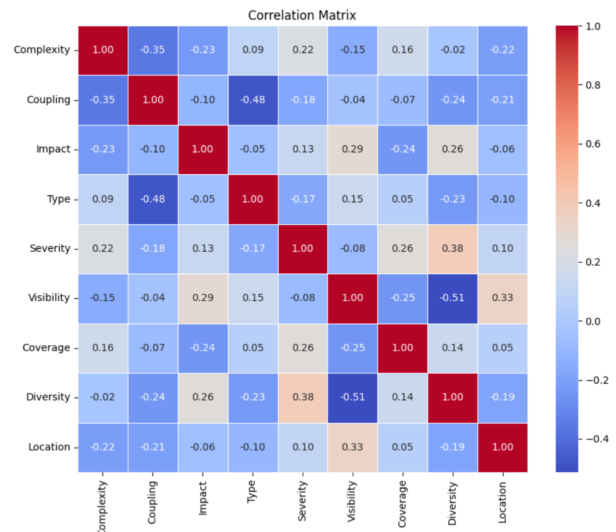
۴- آزمایش‌ها

هدف از روش پیشنهادی، جداسازی جهش‌های معادل از جهش‌های سرسخت با توجه به ویژگی‌های جهش‌ها است. به دلیل اینکه نتیجه اجرای برخی از مورد آزمون‌ها با نتیجه اجرای برنامه اصلی یکسان می‌باشد، آن جهش‌ها را در دسته جهش‌های معادل قرار می‌دهیم. اما در این بین، برخی از جهش‌های دیگری نیز به صورت اشتباهی به‌عنوان جهش‌های معادل ارزیابی می‌شوند. به آن دسته از جهش‌ها، جهش‌های سرسخت گفته می‌شود. شناسایی و جداسازی جهش‌های سرسخت از گروه جهش‌های معادل، نیاز به تلاش دستی زیاد و صرف هزینه و زمان زیادی است. برای همین منظور، روش پیشنهادی با استفاده از بررسی ویژگی‌های جهش‌های ایجاد شده در سطح دستورالعمل‌های بلاتر پیشنهاد شده است تا بتوان بر این اساس، جهش‌های معادل را از جهش‌های سرسخت جدا نمود. همانطور که در بخش قبل، درمورد ویژگی‌های جهش‌ها توضیح داده شده است، هرکدام از این مشخصه‌ها به نوعی، تعیین کننده میزان اهمیت، درجه سختی، جفت شدن و سایر مشخصه‌های مرتبط با جهش است. با توجه به اینکه، جهش‌های تزریق شده به دستورالعمل‌های سطح D، با احتمال بالاتری، جهش معادل هستند، در این مقاله تنها به بررسی دستورالعمل‌های سطح D شناسایی شده می‌پردازیم. درواقع دستورالعلمهای سطوح بالاتر در این بخش مورد تحلیل قرار نمی‌گیرند. بدین ترتیب که برای هر برنامه محک که شامل تعدادی دستورالعمل سطح پایین است، تمام جهش‌های تزریق شده به آنها را استخراج نموده و در جدولی، مشخصه‌های مربوط به آن جهش‌ها را محاسبه کرده و به صورت یک مجموعه داده آماده کرده‌ایم. .بخشی از این مجموعه داده در جدول ۲ نشان داده شده است.

شکل ۱، ماتریس همبستگی ویژگی‌های جهش‌ها را نشان می‌دهد. طبق شکل، همبستگی بین ویژگی‌ها در حد قابل قبولی است.

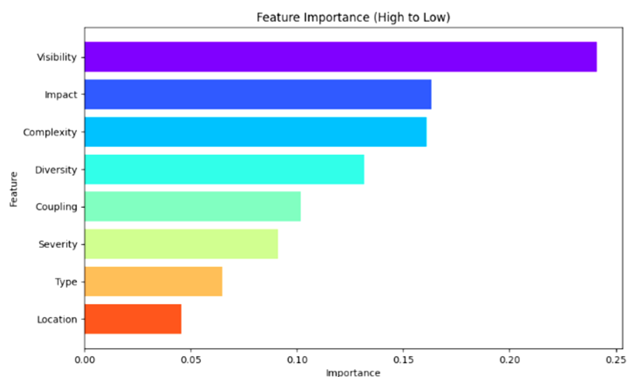
جدول ۲. بخشی از مجموعه داده ایجاد شده

Rank	Location	Diversity	Coverage	Visibility	Severity	Type	Coupling	Complexity	ID	Instruction
1	0	4	80%	9	2	2	50%	0.4	M1	Inst1
1	0	5	70%	5	1	4	20%	0.6	M2	Inst1
0	1	3	90%	7	4	2	10%	0.8	M3	Inst1
1	0	8	80%	2	3	1	70%	0.2	M1	Inst2
0	-1	2	80%	4	4	4	60%	0.4	M2	Inst2
0	-1	4	70%	1	2	3	20%	0.7	M1	Inst3
0	0	5	60%	5	5	1	70%	0.5	M2	Inst3
1	0	6	40%	6	3	3	20%	0.3	M3	Inst3
1	0	4	50%	3	2	3	40%	0.6	M1	Inst4
0	1	2	70%	7	4	2	50%	0.2	M2	Inst4



شکل ۱. ماتریس همبستگی ویژگی‌های جهش

شکل ۲، درجه اهمیت ویژگی‌های جهش‌ها را نشان می‌دهد. همانطور که در شکل دیده می‌شود، ویژگی Visibility بالاترین اهمیت را دارد.



شکل ۲. مقایسه درجه اهمیت ویژگی‌های جهش‌ها

۵- برنامه‌های محک

در این مقاله، هفت برنامه محک مختلف مورد بررسی و آزمایش قرار گرفته است. هرکدام از این برنامه‌ها در سطح واحد و یا تابع بررسی شده‌اند. چون اکثر برنامه‌های کاربردی دنیای واقعی، از توابع مختلف استفاده می‌کنند که همگی آنها درکنار هم، برنامه کاربردی دنیای واقعی را می‌سازد. بنابراین بررسی برنامه‌های کوچک در سطح واحد با همان پیچیدگی برنامه‌های واقعی، نتیجه معتبری را خواهد داشت. درواقع، توابع در میلیون‌ها خط‌کد وجود دارند که برنامه‌های کاربردی دنیای واقعی را تشکیل می‌دهند. اندازهٔ استاندارد یک تابع در دنیای واقعی بین ۵ تا ۵۰ خط‌کد است. برهمین اساس، برنامه‌های محک انتخاب شده برای روش پیشنهادی عبارتند از: Triangle، Factorial، FindMax، Prime، Middle، Bubblesort وDOW. برای تعیین نوع مثلث از برنامه محک Triangle استفاده شده است. ورودی‌های این برنامه شامل سه عدد صحیح است که پس از بررسی شرایط تشکیل مثلث، نوع مثلث بر اساس مقادیر ورودی تعیین می‌شود. یکی از انواع مثلث متساوی الاضلاع، متساوی الساقین و قائم الزاویه به‌عنوان خروجی نشان داده‌می‌شود. اگر شرط تشکیل مثلث برآورده نشود، خروجی یک پیغام خطایی با عنوان عدم تشکیل مثلث چاپ می‌کند. برای محاسبهٔ فاکتوریل یک عدد صحیح از برنامه Factorial استفاده شده است. پس از بررسی شرایط محاسبهٔ فاکتوریل عدد واردشده، خروجی برنامه، نتیجهٔ محاسبهٔ فاکتوریل عدد واردشده خواهد بود. برنامهٔ Prime، با دریافت یک عدد از کاربر، اول بودن عدد را تعیین می‌کند. از برنامهٔ Middle، برای محاسبه و نمایش میانهٔ سه عدد استفاده می‌شود. دراین برنامه، ورودی شامل سه عدد است که مقدار وسط سه عدد وارد شده تعیین شده و به‌عنوان نتیجه برگردانده می‌شود. ورودی برنامهٔ FindMax، شامل مجموعه‌ای از اعداد است. تعداد این اعداد در ابتدا توسط کاربر مشخص می‌شود. خروجی این برنامه، عددی است که بیشینه مقدار اعداد ورودی را مشخص می‌کند. برنامه Bubblesort، لیستی از اعداد صحیح را می‌گیرد و آنها را به ترتیب صعودی با روش مرتب‌سازی حبابی مرتب می‌کند و در خروجی نمایش می‌دهد. در برنامهٔ BubbleSort، مرتب‌سازی با ورودی انجام می‌شود و رشته مرتب شده در خروجی چاپ می‌شود. برنامهٔ DOW شامل سه مقدار ورودی سال، ماه و روز بر اساس تاریخ میلادی است. بعد از دریافت این سه ورودی، روز هفته معادل با تاریخ وارد شده را در خروجی چاپ می‌کند. به‌عنوان مثال، سال ۱۹۸۶ ماه ۴ روز ۱۸، معادل روز جمعه است. جمعه به‌عنوان خروجی چاپ می‌شود. جدول ۳، مشخصه‌های کلی این برنامه‌ها را نشان می‌دهد.

جدول ۳. مشخصات برنامه‌های محک

نام برنامه	اندازهٔ خطوط برنامه	توصیف برنامه
Triangle	۲۸	تعیین نوع مثلث
Factorial	۲۴	تعیین مقدار فاکتوریل عدد ورودی
Mid	۱۴	تعیین عدد میانه سه عدد ورودی
FindMax	۳۰	یافتن بزرگترین مقدار
Prime	۳۳	تعیین اینکه آیا عدد ورودی، عدد اول است یا نه
Bubblesort	۲۱	مرتب سازی لیست ورودی به‌صورت حبابی
DOW	۵۶	تعیین روز هفته معادل با تاریخ ورودی

انتخاب مجموعه مورد آزمون‌ها یکی دیگر از عناصر اصلی تعیین‌کننده در افزایش کارایی روش پیشنهادی محسوب می‌شود. تولید تصادفی داده‌های آزمون، پوشش تمام دستورالعمل‌های برنامه را تضمین نمی‌کند. ازاین‌رو در آزمون‌های انجام شده باید پوشش تمامی دستورالعمل‌ها در نظر گرفته شود. در این روش، برای هر برنامهٔ محک، مجموعه‌های مختلفی از داده‌های آزمون بر اساس معیارهای پوشش تهیه شده است. برای این منظور، گراف جریان

کنترلی هر برنامهٔ محک ایجاد می‌شود. پوشش لبه به‌عنوان یک معیار مبتنی بر نمودار گراف جریان کنترلی برای تولید داده‌های آزمون استفاده می‌شود. پوشش تمام لبه‌ها در این گراف، پوشش کد منبع کل برنامه را تضمین می‌کند. مجموعه‌دادهٔ ایجاد شده در قالب یک فایل اکسل یا csv، توسط الگوریتم یادگیری ماشین در مجموعهٔ ابزار RapidMiner استفاده شده است. بخش اصلی روش پیشنهادی، یک طبقه‌بندی‌کنندهٔ جهش است که با آموزش الگوریتم یادگیری ماشین ایجاد شده است. در این مطالعه، برای تولید برنامه‌های جهش (برنامه‌های خطادار)، از ابزاری بنام Muclipse استفاده شده است. Mujava، افزونه‌ای است که برای اجرای آزمون و ارزیابی امتیاز جهش در ابزار Eclipse گنجانده شده است. کار با ابزار Muclipse برای آزمایش روش پیشنهادی، شامل سه بخش اصلی است. بخش اول آزمایش، مربوط به اجرای مورد آزمون‌های طراحی شده برای هر برنامهٔ محک است. در واقع از بین مورد آزمون‌های طراحی شده، هر بار یکی از موارد آزمون برای اجرا انتخاب خواهد شد و نتیجهٔ آن ذخیره می‌شود. مجری این بخش از ابزار که همان Test runner است، مربوط به Junit است. دو نتیجهٔ مختلف از اجرای مورد آزمون‌ها به دست می‌آید. در حالت اول، اجرای آزمون با موفقیت انجام می‌شود؛ یعنی مقدار مورد انتظار خروجی با نتیجهٔ مورد آزمون تطابق دارد. حالت دوم این است که اجرای آزمون با شکست مواجه شود. در این حالت نسبت به اصلاح مورد آزمون طرح شده اقدام خواهیم کرد. مجری آزمون در این قسمت، Junit4 است. این برنامه‌ها، زیرمجموعهٔ بخش Junit هستند. محتوای هر برنامهٔ Junit، یک مورد آزمون از پنج مورد آزمونی است که شرط پوشش لبه در آن رعایت شده است. برنامه‌های مورد آزمون، در ابزار Muclipse باید با قالب خاصی طراحی شوند. برنامهٔ مورد آزمون، شامل دستورالعمل مقادیر (value)، دستورالعمل مقدار مورد انتظار (expectedOrder)، دستورالعمل مقایسه مقدار حاصل از اجرای مورد آزمون‌ها و مقدار مورد انتظار (assertEquals) است. در این حالت، مورد آزمون مدنظر باید بتواند برنامهٔ اصلی را به‌درستی آزمایش کند. اگر بتواند جهش مدنظر را پیدا کند که همان حالت کشتن جهش است، این جهش به لیست جهش‌های کشته شده اضافه می‌شود. اما اگر نتواند این جهش را کشف کند، گوییم جهش مدنظر زنده مانده است.

شکل۳، کد برنامه FindMax را نشان می‌دهد که یکی از برنامه‌های محک انتخاب شده برای آزمون روش پیشنهادی است و به زبان جاوا نوشته شده است. این برنامه، در بستر Muclipse برای اجرای جهش‌ها نوشته شده است.

```java
public static int Numbers(int[]x)
   {
    int n ;
    n=x.length;
    System.out.println(n+" is number of integers");
          int max=x[0];
          System.out.println(max+" is the first number  ");
     for(int i = 1;i<n;i+=1)
      {
        if(max<x[i])
           max = x[i];
      }
        System.out.println("maximum of  " +n + "
numbers is  " +max);
      return max;
    }
}
```

شکل ۳. برنامه FindMax در بستر Muclipse

شکل ۴ یکی از موردآزمون‌های طراحی شده برای اجرا برروی برنامهٔ محک FindMax را نشان می‌دهد. همان‌طور که مشاهده می‌کنید، مقادیر ورودی آزمون شامل مقادیر {753,564,87,235,123,78} است. مقدار مورد انتظار خروجی عدد ۷۵۳ است. مورد آزمون با مقادیر ورودی فوق، خروجی مد نظر را تولید کرده است.

دستهٔ دوم از برنامه‌های طراحی شده با پسوند M در بخش Muclipse:Mutants قرار می‌گیرند و برنامه‌های جهشی تولید شده در این بخش قرار می‌گیرند. یک نمونه از نتیجهٔ اجرای برنامه‌هایی که حاوی جهش‌های مرتبهٔ اول هستند (هر برنامه تنها یک خطای جهشی دارد)، در شکل ۵ نشان‌داده‌شده است.

```java
package maxx;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.*;
import javax.lang.model.type.ArrayType;
import org.junit.Test;
import junit.framework.TestCase;
public class Max2Test extends TestCase{
          private Max2 c2;
   public void setUp()
    {
          c2 = new Max2();
      }
   @Test
          public void test1() {
          int value []  = {753,564,87,235,123,78};
          int expectedOrder =753 ;
          assertEquals(expectedOrder,c2.Numbers(value));
          }
}
```

شکل ۴. یک مورد آزمون طراحی شده برای برنامهٔ FindMax

```
* Compiling traditional mutants into bytecode
+AOIS_1  +AOIS_10  +AOIS_11  +AOIS_12  +AOIS_13  +AOIS_14
+AOIS_15  +AOIS_16  +AOIS_17  +AOIS_18  +AOIS_19  +AOIS_2
+AOIS_20  +AOIS_21  +AOIS_22  +AOIS_23  +AOIS_24
+AOIS_25  +AOIS_26  +AOIS_27  +AOIS_28  +AOIS_29  +AOIS_3
+AOIS_30  +AOIS_31  +AOIS_32  +AOIS_33  +AOIS_34
+AOIS_35  +AOIS_36  +AOIS_37  +AOIS_38  +AOIS_39  +AOIS_4
+AOIS_40  +AOIS_41  +AOIS_42  +AOIS_43  +AOIS_44  +AOIS_5
+AOIS_6  +AOIS_7  +AOIS_8  +AOIS_9  +AORB_1  +AORB_2
+AORB_3  +AORB_4  +AORB_5  +AORB_6  +AORB_7
+AORB_8  +ASRS_1  +ASRS_2  +ASRS_3  +ASRS_4  +COR_1
+COR_2  +LOI_1  +LOI_10  +LOI_11  +LOI_2  +LOI_3  +LOI_4
+LOI_5  +LOI_6  +LOI_7  +LOI_8  +LOI_9  +ROR_1  +ROR_10
+ROR_11  +ROR_12  +ROR_13  +ROR_14  +ROR_15  +ROR_16
+ROR_17  +ROR_18  +ROR_19  +ROR_2  +ROR_20  +ROR_3
+ROR_4  +ROR_5  +ROR_6  +ROR_7  +ROR_8  +ROR_9
--------------------------
Mutants have been created!
```

شکل ۵. نتیجهٔ اجرای برنامهٔ جهشی FindMax در ابزار Muclipse

در شکل ۵ نتیجهٔ اجرای برنامه‌های آزمون برروی برنامه‌های خطادار برای برنامهٔ محک FindMax ، نشان‌داده‌شده است. ابزار Muclipse، این قابلیت را دارد تا بتواند تعداد جهش‌های کشته شده، تعداد جهش‌های زنده، تعداد جهش‌هایی که فعال مانده‌اند و امتیاز جهش را نشان دهد. شایان ذکر است، جهش‌های زنده، ترکیبی از جهش‌های معادل و جهش‌های سرسخت است که ابزار Muclpise قابلیت تفکیک این دو جهش از هم را ندارد. گزارش نشان داده‌شده در شکل ۶، حاوی اطلاعاتی در زمینهٔ اجرای موردآزمون‌ها و نمایش نتیجهٔ مورد آزمون بر روی تک تک جهش‌های موجود در برنامهٔ خطادار است. گزارش اجرای آزمون‌ها نشان می‌دهند که کدام جهش در برنامه، توسط مورد آزمون کشته شده است و کدامیک توسط مورد آزمون، تشخیص داده نشده است.

شکل ۷ نمایی کلی از محتوای فایل ایجاد شده از عملگرهای جهش تولیدشده برای برنامهٔ FindMax را نشان می‌دهد.

۶- نتایج

هدف از روش پیشنهادی، کاهش تعداد جهش‌های زنده در فرایند آزمون جهش است. جهش‌های زنده از دو جهش مختلف تشکیل شده است. یکی جهش‌های معادل است که این نوع جهش‌ها با هیچ مورد آزمونی قابل شناسایی نیستند. دسته دوم، جهش‌های سرسخت هستند. در روش قبلی، به قرارگیری نادرست جهش‌های سرسخت در دسته جهش‌های معادل پرداخته نشده بود. چون ممکن است به دلیل رفتار مشابه جهش‌های سرسخت با جهش‌های معادل، طبقه‌بندی نادرستی از این دو جهش رخ دهد. در این مقاله، با بررسی ویژگی‌های مهم و اثرگذار جهش‌ها، این ضعف را برطرف کرده‌ایم. نتیجه آزمایش‌ها حاکی از متمایز نمودن جهش‌های سرسخت از دسته جهش‌های معادل به دلیل کاهش تعداد جهش‌های زنده است. این امر همچنین باعث افزایش تعداد جهش‌های کشته شده نیز می‌شود.

```
Analysis of test case
 test 1  kill  ==> AOIS_10 AOIS_11 AOIS_12 AOIS_14
AOIS_15 AOIS_16 AOIS_17 AOIS_19 AOIS_21 AOIS_22
AOIS_23 AOIS_24 AOIS_25 AOIS_26 AOIS_28 AOIS_37
AOIS_38 AOIS_5 AOIS_7 AOIS_9 AOIU_3 ASRS_1 ASRS_2
ASRS_3 ASRS_4 COI_2 LOI_11 LOI_4 LOI_6 LOI_7
--------------------
Live: 26
Dead: 30
Active: 0
--------------------
-------------------------------
Results for class maxx.Max2
-------------------------------
Live mutants: 26
Killed mutants: 30
Mutation Score: 53.0
Writing to file...
```

شکل ۶. خروجی به‌دست‌آمده از اجرای برنامه‌های خطادار در برنامهٔ محک FindMax

```
AOIU_1:27:int_Numbers(int):x.length => -x.length
AOIU_2:33:int_Numbers(int):i => -i
AOIS_9:30:int_Numbers(int):max => ++max
AOIS_10:30:int_Numbers(int):max => --max
AOIS_11:30:int_Numbers(int):max => max++
AOIS_17:31:int_Numbers(int):n => ++n
AOIS_18:31:int_Numbers(int):n => --n
AOIS_19:31:int_Numbers(int):n => n++
AOIS_20:31:int_Numbers(int):n => n--
AOIS_21:32:int_Numbers(int):max => ++max
AOIS_22:32:int_Numbers(int):max => --max
AOIS_23:32:int_Numbers(int):max => max++
AOIS_24:32:int_Numbers(int):max => max--
AOIS_25:32:int_Numbers(int):i => ++i
AOIS_26:32:int_Numbers(int):i => --i
AOIS_27:32:int_Numbers(int):i => i++
AOIS_28:32:int_Numbers(int):i => i--
AOIS_29:33:int_Numbers(int):i => ++i
AOIS_30:33:int_Numbers(int):i => --i
AOIS_31:33:int_Numbers(int):i => i++
AOIS_32:33:int_Numbers(int):i => i--
AOIS_33:36:int_Numbers(int):n => ++n
AOIS_34:36:int_Numbers(int):n => --n
AOIS_35:36:int_Numbers(int):n => n++
AOIS_36:36:int_Numbers(int):n => n--
AOIS_37:36:int_Numbers(int):max => max++
AOIS_38:36:int_Numbers(int):max => max--
AOIS_39:37:int_Numbers(int):max => max++
AOIS_40:37:int_Numbers(int):max => max--
COI_1:31:int_Numbers(int): i < n  =>  !(i < n)
COI_2:32:int_Numbers(int): max < x[i]  =>  !(max < x[i])
LOI_1:27:int_Numbers(int):x.length => -x.length
LOI_2:28:int_Numbers(int):n => ~n
LOI_3:30:int_Numbers(int):max => ~max
LOI_4:31:int_Numbers(int):i => ~i
LOI_5:31:int_Numbers(int):n => ~n
LOI_6:32:int_Numbers(int):max => ~max
LOI_7:32:int_Numbers(int):i => ~i
LOI_8:33:int_Numbers(int):i => ~i
LOI_9:36:int_Numbers(int):n => ~n
LOI_10:36:int_Numbers(int):max => ~max
LOI_11:37:int_Numbers(int):max => ~max
ASRS_1:31:int_Numbers(int):i += 1 => i /= 1
ASRS_2:31:int_Numbers(int):i += 1 => i *= 1
ASRS_3:31:int_Numbers(int):i += 1 => i -= 1
ASRS_4:31:int_Numbers(int):i += 1 => i %= 1
```

شکل ۷. بخشی از عملگرهای جهشی ایجاد شده برای برنامهٔ محک FindMax

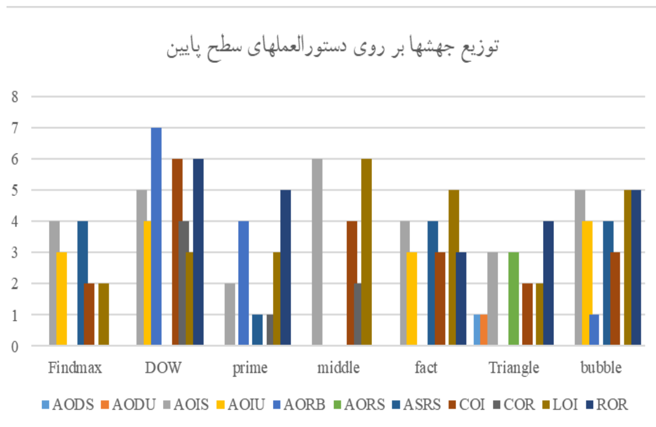
شکل ۸ تعداد کل عملگرهای تزریق شده در تمامی دستورالعمل‌های برنامه‌های محک را نشان می‌دهد. همانطور که در نمودار مشخص است، برنامه DOW، بیشترین تعداد دستورالعمل‌های سطح پایین و در نتیجه آن، بیشترین مشارکت عملگرها را دارد. جدول ۴، برای هر برنامه، تعداد کل جهش‌های تولید شده، تعداد جهش‌های تولید شده برای دستورالملهای سطح بالا و تعداد جهش‌های تولید شده برای دستورالعمل‌های سطح پایین را نشان می‌دهد. تعداد کل جهش‌ها، به تفکیک نوع دستورالعمل‌ها، در سطر آخر مشخص شده است.

در روش پیشنهادی، آزمایش‌های انجام گرفته بر روی دستورالعمل‌های هفت برنامه محک با دو روش سنتی و روش آگاه به انتشار خطا مقایسه شده است. نتایج حاکی از بهبود کارایی روش پیشنهادی با کاهش تعداد جهش‌های زنده است. شکل ۹ نتایج پنج مورد آزمون مختلف برروی برنامه Triangle را نشان می‌دهد. شکل ۱۰، تعداد جهش‌های زنده در برنامهٔ BubbleSort برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون را نشان می‌دهد. در شکل ۱۱، تعداد جهش‌های زنده در برنامهٔ Factorial برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون نمایش داده شده است. تعداد جهش‌های زنده در برنامهٔ Prime برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون، در شکل ۱۲ آمده است. لازم به ذکر است، برنامه‌های محک انتخاب شده برای آزمایش روش پیشنهادی، براساس آزمونهای مختلف استاندارد جهانی انتخاب شده است.

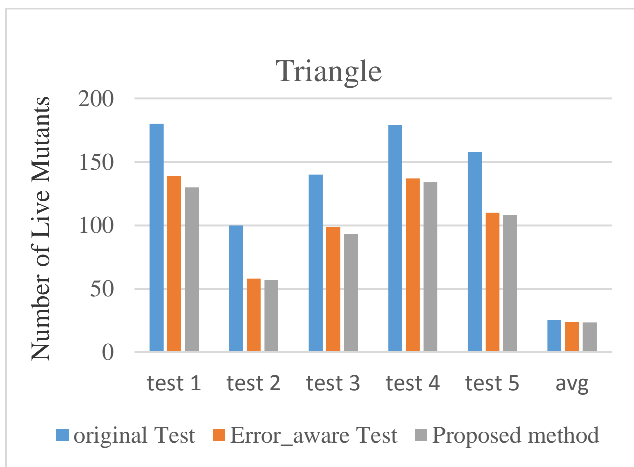
شکل ۱۳، تعداد جهش‌های زنده در برنامه Middle را نشان می‌دهد. نتایج حاکی از کاهش محسوس تعداد جهش‌های زنده است. شکل ۱۴ کاهش تعداد جهش‌های زنده برنامه FindMax را نشان می‌دهد. شکل ۱۵ نیز تعداد جهش‌های زنده در هر سه روش سنتی، آگاه به انتشار خطا و روش پیشنهادی را برای برنامه DOW نشان می‌دهد.

جدول ۴. تعداد جهش‌های تولید شده براساس نوع دستورالعمل‌های برنامه محک

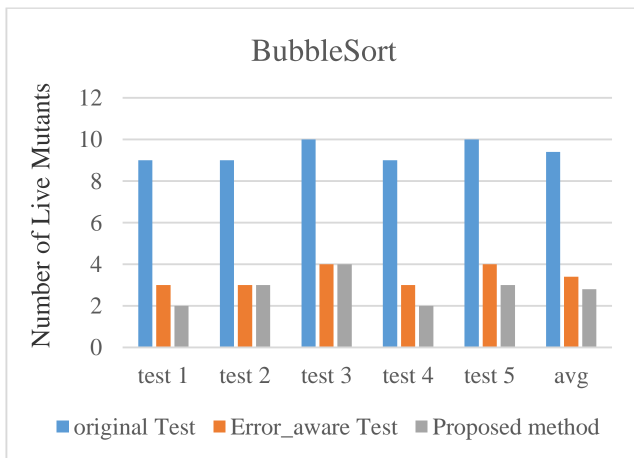
نام برنامه	تعداد کل جهش‌های تولید شده برای هر برنامه	تعداد جهش‌های تولید شده برای دستورالعمل‌های سطح بالا	تعداد جهش‌های تولید شده برروی دستورالعمل‌های سطح پایین
Triangle	252	208	44
Bubble Sort	97	87	10
Factorial	68	50	18
Prime	66	51	15
Middle	121	106	15
FindMax	56	38	18
Dow	294	260	34
Total Mutants	954	800	154



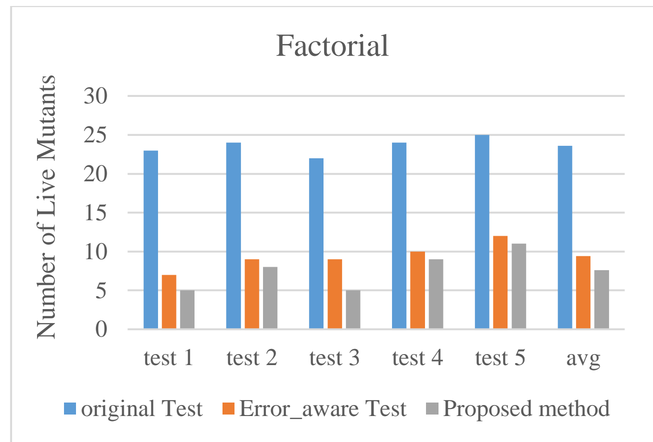
شکل ۸. توزیع عملگرهای جهشی برای هفت برنامه محک



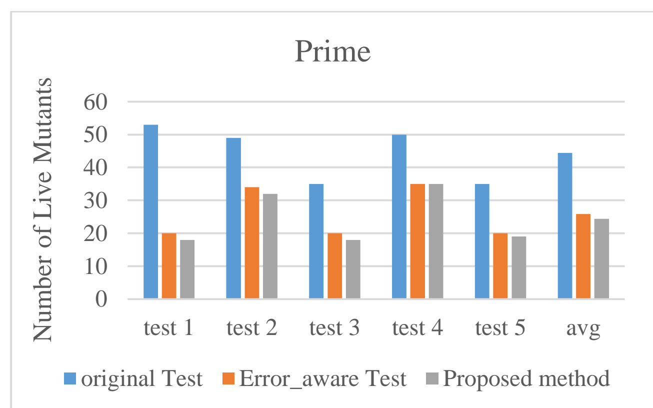
شکل ۹. تعداد جهش‌های زنده در برنامهٔ Triangle برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون



شکل ۱۰. تعداد جهش‌های زنده در برنامهٔ BubbleSort برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون

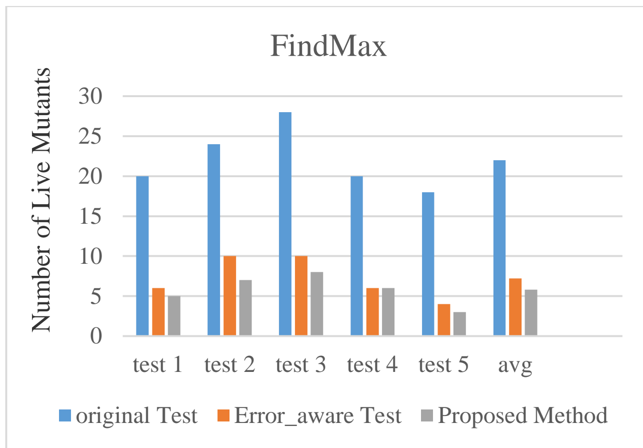


شکل ۱۱. تعداد جهش‌های زنده در برنامهٔ Factorial برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون

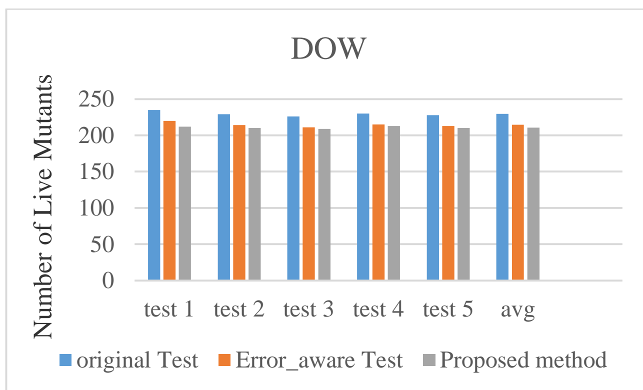


شکل ۱۲. تعداد جهش‌های زنده در برنامهٔ Prime برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون

شکل ۱۳. تعداد جهش‌های زنده در برنامهٔ Middle برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون



شکل ۱۴. تعداد جهش‌های زنده در برنامهٔ FindMax برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون



شکل ۱۵. تعداد جهش‌های زنده در برنامهٔ DOW برای سه روش سنتی، آگاه به انتشارخطا و روش پیشنهادی با پنج مورد آزمون

مراجع

[1] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. "Hints on test data selection: Help for the practicing programmer." IEEE Computer, 11(4): 34-41, 1978.

[2] Jeff Offutt and Ammei Lee. (1998) "An empirical evaluation of weak mutation."IEEE Transactions on Software Engineering,649-660.

[3] Mark Harman. "The current state and future of search-based software engineering." In Proceedings of the 28th International Conference on Software Engineering (ICSE'06), pages 342-351. ACM, 2006.

[4] Simon Poulding and James A. Jones. "An empirical study of the robustness of MacOS applications using random testing." In Proceedings of the 10th International Workshop on Dynamic Analysis (WODA'12), pages 35-40. IEEE, 2012.

[5] López, J., Kushik, N., & Yevtushenko, N. (2018). Source Code Optimization using Equivalent Mutants. Inf. Softw. Technol., 103, 138-141. https://doi.org/10.1016/j.infsof.2018.06.013.

[6] Ghiduk, A., Girgis, M., & Shehata, M. (2019). Employing Dynamic Symbolic Execution for Equivalent Mutant Detection. IEEE Access, 7, 163767-163777. https://doi.org/10.1109/ACCESS.2019.2952246.

[7] Zeinab Asghari, Bahman Arasteh, and Abbas Koochari. 2024. Effective Software Mutation-Test Using Program Instructions

Classification. J. Electron. Test. 39, 5–6 (Dec 2023), 631–657. https://doi.org/10.1007/s10836-023-06089-0

[8] Ghiduk, A., Girgis, M., & Shehata, M. (2019). Employing Dynamic Symbolic Execution for Equivalent Mutant Detection. IEEE Access, 7, 163767-163777. https://doi.org/10.1109/ACCESS.2019.2952246.

[9] Jammalamadaka, K., & Parveen, N. (2021). Equivalent mutant identification using hybrid wavelet convolutional rain optimization. Software: Practice and Experience, 52, 576 - 593. https://doi.org/10.1002/spe.3038.

[10] Souza, B., & Gheyi, R. (2020). A Lightweight Technique to Identify Equivalent Mutants. . https://doi.org/10.5753/CBSOFT_ESTENDIDO.2020.14630.

[11] Tenorio, M., Lopes, R., Fechine, J., Marinho, T., & Costa, E. (2019). Subsumption in Mutation Testing: An Automated Model Based on Genetic Algorithm. 16th International Conference on Information Technology-New Generations (ITNG 2019). https://doi.org/10.1007/978-3-030-14070-0_24.

[12] Fernandes, L., Ribeiro, M., Gheyi, R., Delamaro, M., Guimarães, M., & Santos, A. (2022). Put Your Hands In The Air! Reducing Manual Effort in Mutation Testing. Proceedings of the XXXVI Brazilian Symposium on Software Engineering. https://doi.org/10.1145/3555228.3555233.

[13] Abuljadayel, A., & Wedyan, F. (2018). An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms. International Journal of Intelligent Systems and Applications, 10, 34-45. https://doi.org/10.5815/IJISA.2018.01.05.

[14] Basile, D., Beek, M., Cordy, M., & Legay, A. (2020). Tackling the equivalent mutant problem in real-time systems: the 12 commandments of model-based mutation testing. Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A. https://doi.org/10.1145/3382025.3414966.

[15] Naeem, M., Lin, T., Naeem, H., & Liu, H. (2020). A machine learning approach for classification of equivalent mutants. Journal of Software: Evolution and Process, 32. https://doi.org/10.1002/smr.2238.

[16] Ayad, A., Marsit, I., Loh, J., Omri, M., & Mili, A. (2019). Estimating the Number of Equivalent Mutants. 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 112-121. https://doi.org/10.1109/ICSTW.2019.00039.

[17] Ayad, A., & Mili, A. (2020). Automated Estimation of the Rate of Equivalent Mutants. 2020 International Conference on Computational Science and Computational Intelligence (CSCI), 1794-1799. https://doi.org/10.1109/CSCI51800.2020.00331.

[18] Kazerouni, A., Davis, J., Basak, A., Shaffer, C., Servant, F., & Edwards, S. (2021). Fast and accurate incremental feedback for students' software tests using selective mutation analysis. J. Syst. Softw., 175, 110905. https://doi.org/10.1016/j.jss.2021.110905.

[19] Wu, X., Zhang, J., & Zhao, Y. (2023). Model-based testing and mutation analysis: Recent advances and future directions. Journal of Software: Evolution and Process. https://doi.org/10.1002/smr.2602

[20] Williams, J. L., Gupta, P. S., & Rodrigues, S. C. (2023). A comparative study of mutation testing techniques: Recent developments and future directions. Journal of Software: Evolution and Process. https://doi.org/10.1002/smr.2667

[21] Khan, R., & Amjad, M. (2019). Mutation-based genetic algorithm for efficiency optimisation of unit testing. Int. J. Adv. Intell. Paradigms, 12, 254-265. https://doi.org/10.1504/IJAIP.2019.10019862.

[22] Petrović, G., Ivankovic, M., Fraser, G., & Just, R. (2021). Practical Mutation Testing at Scale: A view from Google. IEEE Transactions on Software Engineering, 48, 3900-3912. https://doi.org/10.1109/ TSE.2021.3107634.

[23] Lee, J., Patel, A., Zhang, K., & Chen, Y. (2023). Advancements in mutation testing using deep learning techniques: A review and future perspectives. IEEE Access,11,158264158283.https://doi.org/10.1109/ACCESS.2023.3 290548.