New Researches in the Smart City
Vol 1 (Issue 4), 53-67
Nrsc.apadana.ac.ir

Apadana Institute of Higher Education

Code Bug Localization Using Genetic Algorithm with Factor Elimination

Farzaneh Zareie *

Head of Planning, Statistics, and Information Analysis Department, Hamedan Municipality, Iran.

Abstract

Considering that the longest phase of the software lifecycle is maintenance, software engineers seek methods to automate the processes in this phase. During maintenance, one of the most critical activities is locating bugs and providing error-free updates. Despite the numerous tests conducted during software development, unknown errors still exist, causing inconvenience to users. Additionally, some errors result from side effects of code changes, making it imperative to find an automated and accurate way to locate bugs. Various works have been done in this area, all of which focus on the impact of statements on output and their behavior in successful and unsuccessful executions. This paper proposes a method for precise bug localization using a defined fitness function for the genetic algorithm, which eliminates the local optimal factor for code statements and assigns scores based on their bug-prone behavior, considering their slicing. The evaluation of this method demonstrates its accuracy.

Keywords: software testing, software maintenance, bug, bug localization, genetic algorithm

ISSN: 2980-8936 Accepted: 15/August/2023 Received: 19/January/2023

* Corresponding Author: FarzanehZareie@gmail.com

فصلنامه پژوهش‌های نوین در شهر هوشمند
دوره ۱، شماره ٤، تابستان۱٤۰۲، ٥۳-٦۷
Nrsc.apadana.ac.ir

خطایابی کد با استفاده از الگوریتم ژنتیک

فرزانه زارعی * رئیس ادارهٔ برنامه‌ریزی، آمار و تحلیل اطلاعات شهرداری همدان، ایران

تاریخ دریافت: ۱۴۰۱/۱۰/۲۹ تاریخ پذیرش: ۱۴۰۲/۰۵/۲۴ ISSN: 2980-8936

چکیده

با توجه به اینکه طولانی‌ترین مرحله از چرخهٔ عمر نرم‌افزار نگه‌داشت آن است، مهندسان نرم‌افزار به دنبال یافتن روش‌هایی جهت خودکارسازی فرایندهای این مرحله هستند. در زمان نگه‌داشت بیشترین فعالیتی که انجام می‌شود مکان‌یابی خطا و ارائهٔ نسخهٔ جدید فاقد خطا است. علی‌رغم آزمون‌های زیادی که در زمان توسعهٔ نرم‌افزار انجام می‌شود، همچنان خطاهایی ناشناخته در نرم‌افزار وجود داشته که کاربران را آزار می‌دهد. همچنین بخشی از خطاها اثرات جانبی ایجاد تغییرات در کد نرم‌افزار است؛ بنابراین یافتن راهی برای مکان‌یابی دقیق خطا به‌صورت خودکار اجتناب‌ناپذیر است. تاکنون کارهای زیادی در این حوزه انجام شده است که همگی به‌نوعی بر اثرگذاری جملات بر خروجی و نیز رفتار متفاوت آن‌ها در اجراهای موفق و ناموفق تمرکز دارند. در این مقاله روشی جهت مکان‌یابی دقیق خطای برنامه ارائه شده که در آن با استفاده از تابع برازندگی تعریف‌شده برای الگوریتم ژنتیک با حذف فاکتور بهینهٔ محلی جملات کد و نیز جملات حاصل از برش‌بندی برحسب میزان خطادار بودنشان امتیازدهی شده و ارزیابی انجام‌شده نشان‌دهندهٔ دقت آن است.

کلیدواژه‌ها: آزمون نرم‌افزار، نگه‌داشت نرم‌افزار، خطا، مکان‌یابی خطا، الگوریتم ژنتیک

* نویسندهٔ مسئول: FarzanehZareie@gmail.com

مقدمه

علی‌رغم پیشرفت در زبان‌های برنامه‌سازی، هم‌چنان بروز خطا در نرم‌افزار، تولیدکنندگان و مصرف‌کنندگان نرم‌افزار را آزار می‌دهد. برای از بین بردن خطا، نرم‌افزار باید اشکال‌زدایی شود. اشکال‌زدایی شامل دو مرحلهٔ تشخیص و تصحیح خطاست. یافتن محل بروز (تشخیص) خطا در برنامه‌های کوچک به‌آسانی صورت می‌گیرد اما در برنامه‌های بزرگ به‌صورت دستی کاری وقت‌گیر و خسته‌کننده است. درنتیجه به‌کارگرفتن روش‌هایی برای خودکارسازی و کوچک‌کردن محدودهٔ جستجو ضروری است. روش‌هایی مانند برش‌بندی، اشکال‌زدایی آماری و اشکال‌زدایی دلتا ازجمله روش‌هایی هستند که به تشخیص خطا کمک می‌کنند. برش‌بندی تکنیکی برای خلاصه‌سازی برنامه است که وابستگی‌های یک نقطهٔ مشخص از کد را به دست می‌دهد. این روش علی‌رغم دقت بالایی که دارد، زمان‌بر است؛ بنابراین در این پروژه ارائهٔ یک روش خطایابی آماری هدف قرار گرفت. در روش پیشنهادی ابتدا با استفاده از مستندگذاری کد برنامه، اطلاعات زمان اجرا شامل بلوک‌های کد اجراشده در مجموعه‌ای از اجراهای موفق و ناموفق جمع‌آوری می‌شود. براساس اطلاعات جمع‌آوری‌شده، بردارهای اجرای برنامه به ازای هر اجرا ایجاد می‌شوند. تعداد عناصر این بردارها برابر با تعداد بلوک‌های کد برنامه است. مقدار هر عنصر نشان‌دهندهٔ تعداد دفعات اجرای بلوک مربوطه در یک اجرای خاص است. با داشتن این بردارها و وضعیت اجرای آن‌ها (موفق و یا ناموفق بودن آن‌ها) در مرحله بعد، با استفاده از الگوریتم ژنتیک قوانین انجمنی (Kantardzic, 2011) استخراج شده و براساس آن‌ها به بلوک‌ها برحسب میزان مشارکت آن‌ها در اجراهای ناموفق و عدم مشارکت آن‌ها در اجراهای موفق امتیاز داده می‌شود. اساس امتیازدهی به‌کاررفته در این پروژه درواقع اساس روش‌های خطایابی آماری است. آن‌چه در میزان قدرت تشخیص خطا مهم است، تابع برازندگی مورداستفاده در الگوریتم ژنتیک است. در روش پیشنهادی از یک فرمول جدید برای محاسبهٔ برازندگی استفاده شده است. هم‌چنین جهت ارزیابی روش پیشنهادی از برنامه‌های مجموعهٔ زیمنس استفاده شده است.

تعریف مفاهیم پایه

الگوریتم تکاملی: الگوریتم تکاملی الگوریتمی است که جنبه‌های انتخاب طبیعی و تداوم هماهنگی را ترکیب می‌کند. الگوریتم تکاملی از جمعیت ساختارهای قوانین انتخاب، ترکیب‌بندی مجدد، تغییر و بقا، حفاظت می‌کند. این ساختارها مبتنی بر عملگرهای ژنتیکی هستند. در این روش، محیط، خود، تعیین‌کنندهٔ هماهنگی یا عملکرد هر یک از افراد جمعیت است و از افراد هماهنگ‌تر برای تولید مجدد استفاده می‌کند.

الگوریتم ژنتیک: الگوریتم ژنتیک یکی از اولین الگوریتم‌های تکاملی تصادفی مبتنی بر جمعیت است. عملگرهای اصلی این الگوریتم عبارت‌اند از انتخاب، اتصال و جهش. این الگوریتم با الهام از نظریهٔ تکاملی داروین معرفی شده و ایدهٔ "بقای موجودات برتر" را شبیه‌سازی می‌کند. هر راه‌حل در این الگوریتم متناظر با یک کروموزوم و هر پارامتر با یک ژن نمایش داده می‌شود. الگوریتم ژنتیک برازندگی هر یک از راه‌حل‌ها (کروموزوم‌ها) را با استفاده از تابعی به نام تابع برازندگی اندازه‌گیری می‌کند. به‌منظور تقویت‌کردن راه‌حل‌های ضعیف، بهترین راه‌حل‌ها به‌صورت تصادفی با استفاده از یکی از مکانیزم‌های انتخاب، برگزیده می‌شوند. با توجه به تصادفی‌بودن الگوریتم ژنتیک، بهترین راه‌حل‌ها ممکن است بهینهٔ محلی باشند؛ بنابراین به‌منظور جلوگیری از انتخاب بهینهٔ محلی و قابل‌اعتمادکردن حاصل الگوریتم ژنتیک، باید به راه‌حل‌های ضعیف نیز به‌صورت تصادفی اجازهٔ بهبود داد. به این منظور از دو عملگر جهش و اتصال استفاده می‌شود. با این دو عملگر، نسل بهتری نسبت به نسل فعلی تولید می‌شود. در عملگر اتصال، از دو راه‌حل به‌عنوان والد برای تولید یک راه‌حل دیگر به‌عنوان فرزند استفاده می‌شود و در عملگر جهش به‌صورت تصادفی روی برخی از

ژن‌های یک کروموزوم (پارامترهای یک راه‌حل) تغییری ایجاد شده و راه‌حل جدیدی به وجود می‌آید. این دو عملگر باعث می‌شود تا الگوریتم گرفتار بهینهٔ محلی نشده و بهینهٔ سراسری تولید کند.

قوانین انجمنی (وابستگی) و کشف آن‌ها: تحلیل وابستگی‌ها به جستجو برای یافتن ارتباط در مجموعهٔ داده‌ها می‌پردازد. به عبارتی دیگر تحلیل وابستگی‌ها، مطالعهٔ ویژگی‌ها یا خصوصیاتی می‌باشد که با یکدیگر همراه هستند. این قوانین به‌صورت کلی $A \Rightarrow B$ تعریف می‌شوند که در آن A مقدم و B تالی نامیده می‌شود. این قوانین به این صورت تفسیر می‌شوند که: "اگر عناصر مجموعهٔ A در یک رکورد ظاهر شوند، عناصر مجموعهٔ B هم با احتمال بالایی ظاهر خواهند شد" و بنابراین $A \cup B$ یک مجموعه پرتکرار است و نیز اشتراک دو مجموعه برابر با تهی است. هدف از کشف و استخراج قوانین انجمنی، یافتن آن دسته از قوانین انجمنی است که ضریب اطمینان و نیز ضریب پشتیبانی آن‌ها بالاتر از یک حد آستانه است. دو مقدار ضریب پشتیبانی و ضریب اطمینان به ترتیب عبارت‌اند از: تعداد عناصر جمعیت که در آن‌ها A و B با هم دیده شده نسبت به کل عناصر (ضریب پشتیبانی) و تعداد عناصر جمعیت که در آن‌ها A و B دیده شده نسبت به تعداد عناصری که در آن‌ها A دیده شده است (ضریب اطمینان) (Narvekar & Syed, 2015; Altay & Alatas, 2020; Parsa et al., 2011 Baro et al., 2020). در کل سه روش عمدهٔ کشف قوانین انجمنی وجود دارد که عبارت‌اند از: بهینه‌سازی، توزیع و گسسته‌سازی (Altay & Alatas, 2020). در روش پیشنهادی از روش بهینه‌سازی با استفاده از الگوریتم ژنتیک استفاده شده است.

خطای معنایی برنامه: خطای معنایی که به آن خطای منطقی نیز گفته می‌شود، خطایی است که از نظر نحوی درست بوده اما باعث می‌شود حاصل اجرای برنامه با آن‌چه مورد انتظار است، متفاوت باشد.

مستندگذاری کد: عبارت است از افزودن بخشی به کد برنامه به‌منظور جمع‌آوری اطلاعات موردنظر از اجرای برنامه.

برش‌بندی: برش‌بندی تکنیکی است که تمام وابستگی‌های یک جملهٔ خاص از برنامه را (بسته به نوع برش‌بندی) به دست می‌دهد. منظور از وابستگی‌ها جملاتی هستند که بر آن جمله و مقادیر محاسبه‌شدهٔ در آن تأثیر گذاشته‌اند (Aho et al., 2007; Agrawal & Horgan, 1990). هدف از به‌کارگیری برش‌بندی در اشکال‌زدایی نرم‌افزار این است که برای وجود خطا فقط باید جملاتی از برنامه بررسی شوند که واقعاً در بروز خطا تأثیر داشته‌اند.

اشکال‌زدایی[1]: اشکال‌زدایی به دنبال آزمون نرم‌افزار انجام می‌شود. زمانی که آزمون نرم‌افزار وجود خطایی را در نرم‌افزار تشخیص می‌دهد، اشکال‌زدایی منشأ خطا را یافته و آن را از بین می‌برد.

داده‌کاوی: عبارت است از استخراج خودکار اطلاعات مفید از بانک‌های اطلاعات بزرگ. ایدهٔ اصلی داده‌کاوی بر این امر استوار است که داده‌های قدیمی حاوی اطلاعاتی هستند که در آینده مورداستفاده قرار گرفته و مفید خواهند بود. هدف داده‌کاوی یافتن الگوهایی در داده‌های پیشین است که نیازها، ترجیحات و تمایلات را روشن‌تر می‌نماید. این حقیقت که الگوها همواره واضح نیستند و علائم دریافت شده از داده‌ها گاهی مبهم و گیج‌کننده هستند کار را سخت‌تر می‌نماید. لذا جداکردن علائم از چیزهای بی‌استفاده یعنی تشخیص الگوهای اساسی در بطن متغیرهای به‌ظاهر تصادفی، یکی از نقش‌های مهم داده‌کاوی است.

مروری بر کارهای انجام شده

در سال ۱۹۸۴ ایدهٔ استفاده از گراف وابستگی برای محاسبهٔ برش‌ها در (Ottenstein & Ottenstein, 1984) مطرح شد. برای خطایابی با این روش، از برش‌بندی پس‌رو برای پیمایش گراف استفاده می‌شد (Ottenstein &

1. debugging

(Ottenstein, 1984; Tip, 1994). با توجه به این‌که روش‌های برش‌بندی ایستا تمام مسیرها را بدون در نظر گرفتن اجرای برنامه موردبررسی قرار می‌دهند، برش حاصل احتمالاً بهینه نخواهد بود. بنابراین نیاز به روشی هست که برش کوچک‌تری تولید کند. در زمان خطایابی برنامه معمولاً برنامه‌نویسان با توجه به ورودیِ منجر به خطا سعی در مکان‌یابی خطا می‌کنند. این موضوع محققان را به سوی روش دیگری هدایت کرد که برش‌بندی پویا نام دارد. این روش در سال ۱۹۸۸ مطرح شد. در این روش با توجه به اجرای برنامه، فقط آن دسته از جملات که اجرا شده‌اند و بر یک متغیر در نقطه‌ای از برنامه تأثیر داشته‌اند، در برش قرار می‌گیرند. الگوریتم اولیهٔ ارائه‌شده بر اساس گراف جریان بود و منجر به تولید برش بزرگی می‌شد. البته این روش برشی کوچک‌تر از برش ایستا تولید می‌کرد اما هم‌چنان جملات اضافی در آن یافت می‌شد. در سال ۱۹۹۰ در (Agrawal & Horgan, 1990) از گراف‌های وابستگی برای به‌دست‌آوردن برش پویا استفاده شد که در آن چهار روند برای برش‌بندی جدید مطرح شد.

خطایابی آماری با اضافه‌نمودن کد به برنامه، اطلاعات زمان اجرای برنامه را جمع‌آوری می‌کند. به فرآیند افزودن کد به برنامه، مستندگذاری گفته می‌شود که عموماً سربار زمانی زیادی را به کاربر تحمیل می‌کند (Ammann & Offutt, 2016). اطلاعات جمع‌آوری‌شده از اجراهای مختلف برنامه (موفق و ناموفق) با استفاده از روش‌های آماری موردتحلیل قرار می‌گیرند تا محدودهٔ بروز خطا مشخص شود. ایراد دیگر این روش‌ها در مستندگذاری پرهزینهٔ آن‌هاست. از روش‌های مهم خطایابی آماری می‌توان به سه روش مهم رنگ‌آمیزی جملات (Jones & Harrold, 2005)، ایزوله‌سازی خطا (Liblit et al., 2005) و آزمون فرض (Liu et al., 2005) اشاره کرد. ضعف‌های مشترک روش‌های آماری در سربار ناشی از مستندگذاری، یافتن محدودهٔ خطا (و نه منشأ اصلی خطا)، بررسی و امتیازدهی به کل جملات اجراشده و یا کل تعیین‌کننده‌ها (که بسیاری از آن‌ها روی تولید خروجی نادرست نقشی نداشته‌اند) و نیاز به داشتن تعداد زیادی اجراهای موفق و ناموفق است.

ترکیبی موفق از برش‌بندی و خطایابی آماری را می‌توان در (Parsa & Zareie, 2013; Parsa et al., 2015; Parsa et al., 2011) ملاحظه کرد. در این روش‌ها سعی شده تا ابتدا با استفاده از برش‌بندی مجموعهٔ کاوش کد کوچک شده و سپس با استفاده از تحلیل آماری اقدام به مکان‌یابی دقیق خطا در کد می‌کند.

در (Hildebrandt & Zeller, 2000) نیز روشی معرفی شده که ابتدا متغیرهای مؤثر در نتیجهٔ اجرا را شناسایی و کمینه کرده و سپس با استفاده از مقادیر آن‌ها در اجراهای موفق، مجموعهٔ مقادیر معتبر را شناسایی می‌کند و سپس از این مقادیر در اجراهای ناموفق استفاده کرده و اقدام به مکان‌یابی خطا می‌کند.

در (Wang et al., 2020) روشی ارائه شده که با استفاده از فراداده‌ها و ردگیری پشتهٔ کنترل، اطلاعاتی به دست می‌آورد که به کوچک‌کردن فضای جستجو کمک می‌کند. در سال‌های اخیر، از روش‌های بازیابی اطلاعات به‌منظور مکان‌یابی خطای برنامه استفادهٔ زیادی شده است. مبنای این روش‌ها تعیین و شناسایی محل خطا با استفاده از شباهت محتوایی آن‌ها به گزارش‌های خطاست. در این روش‌ها دو نوع شباهت بررسی می‌شود: شباهت لغوی و شباهت معنایی. در شباهت لغوی لغات و کاراکترهای مشترک بین دو متن، تعیین‌کنندهٔ میزان شباهت است؛ در شباهت معنایی میزان شباهت معنای دو متن مبنای محاسبه شباهت دو متن است. مشکل موجود در اکثر روش‌های بازیابی اطلاعات این است که معمولاً از یکی از انواع شباهت‌ها استفاده می‌کنند درحالی‌که این دو نوع مکمل یکدیگر هستند. در روش پیشنهادی از فایل‌های مربوط به خطاهای اصلاح‌شده به‌عنوان ناظر در یادگیری مدل استفاده شده است. در این روش هم‌چنین از برخی اطلاعات مانند نوع سیستم‌عامل، بستهٔ مورداستفاده، پیکربندی سیستم و غیره به‌عنوان فراداده استفاده شده و هم‌چنین براساس نام فایل‌ها از پشتهٔ کنترل برنامه، مدلی برای اجراهای ناموفق استخراج شده است. جهت یافتن محل

خطا از هر دو شباهت لغوی و معنایی استفاده شده است. نوآوری دیگری که در این روش به کار گرفته شده، استفاده از تعداد خط کد فایل جهت تعیین احتمال خطا است.

در (Lam et al., 2017) نیز روشی مبتنی بر آموزش عمیق معرفی شده که با استفاده از شبکهٔ عصبی عمیق و بازیابی اطلاعات حاصل از گزارش‌های خطاها اقدام به مکان‌یابی خطا در فایل‌های مختلف منبع می‌کند. در روش پیشنهادی با توجه به اینکه احتمال عدم‌تطابق گزارش خطا با کد منبع (عدم وجود شباهت لغوی) زیاد است، از شبکهٔ عصبی عمیق جهت کشف ارتباطات موجود لغوی بین گزارش‌های خطا استفاده شده و لغات غیرمشابه را نیز در دسته‌های مشابه دسته‌بندی می‌کند. این روش با استخراج شباهت‌های پنهان، با دقت بالاتری فایل‌های خطادار را شناسایی می‌کند.

قوانین انجمنی، یکی از تکنیک‌های اصلی داده‌کاوی است و تقریباً مهم‌ترین شکل کشف و استخراج الگوها در سیستم‌های یادگیری است. قوانین انجمنی نشان‌دهندهٔ وجود ارتباط بین بعضی عناصر هستند به این صورت که این عناصر اغلب همراه با هم دیده می‌شوند. در صورتی که در انتخاب ویژگی از قوانین انجمنی استفاده شود می‌توان گفت وجود یک ویژگی دلیل بر وجود بعضی ویژگی‌های دیگر است (میرزایی و محمودی، ۱۳۹۴).

الگوریتم Apriori یک الگوریتم قدرتمند برای استخراج مجموعهٔ آیتم‌های پرتکرار برای کشف قوانین انجمنی می‌باشد. مجموعهٔ آیتم‌های پرتکرار، مجموعه‌هایی از آیتم‌ها هستند که درجهٔ پشتیبانی بزرگ‌تر-مساوی با حداقل درجهٔ پشتیبانی را دارا می‌باشند. هدف اصلی این الگوریتم یافتن مجموعهٔ آیتم‌های پرتکرار (آیتم‌هایی که زیاد و با هم دیده می‌شوند) است. لازم است یک مجموعهٔ آیتم پرتکرار باشد تا بتوان مجموعهٔ آیتم‌های پرتکرار را با درجه از ۱ تا k به‌صورت تکراری پیدا نمود و از این مجموعهٔ آیتم‌های پرتکرار جهت ایجاد قوانین انجمنی استفاده نمود. Apriori از یک روش تکراری برای یافتن مجموعهٔ عناصر پرتکرار استفاده می‌کند به این صورت که برای یافتن itemset-(k+1)ها از itemset-kها استفاده می‌کند. ابتدا itemset-1ها پیدا می‌شوند که با L1 نمایش داده می‌شوند. L1 برای یافتن L2 که itemset-2ها هستند استفاده می‌شود و همین‌طور این فرآیند ادامه دارد تا هیچ itemset-k یافت نشود. یافتن L_k نیاز دارد تا کل پایگاه یک‌بار پیمایش شود. برای تولید سطح‌به‌سطح مجموعه‌های پرتکرار یک ویژگی مهم به نام ویژگی Apriori به‌صورت زیر معرفی می‌شود که فضای جستجو را کاهش می‌دهد:

Apriori تمام زیرمجموعه‌های ناتهی یک مجموعهٔ پرتکرار خود پرتکرار هستند. از این ویژگی می‌توان این‌چنین استنتاج کرد که اگر یک مجموعهٔ Iحداقل درجهٔ پشتیبانی را نداشته باشد آنگاه I پرتکرار نیست. اگر عنصر A به مجموعهٔ Iاضافه شود آنگاه مجموعهٔ حاصل نمی‌تواند دارای فراوانی بیش از I باشد. پس این مجموعه نیز پرتکرار نیست (Agrawal & Srikant, 1994). یکی از روش‌های مطرح در استخراج الگوهای موجود در یک مجموعهٔ داده، روش HUI می‌باشد. این روش مجموعهٔ مواردی (item set) با اندازه k را که بیشتر از یک حد آستانه (که توسط کاربر تعیین می‌شود) تکرار شده‌اند، شناسایی می‌کند (Liu et al., 2005). روش‌های دیگری نیز در مطالعات بعدی ارائه شدند که این الگوریتم را بهبود داده و قابل‌اعمال روی مجموعه‌های بزرگ و کوچکی از داده‌ها هستند. این الگوریتم‌ها با سرعت بهتری نسبت به HUI قوانین انجمنی را در مجموعهٔ اطلاعات دلخواه استخراج می‌کنند (Fournier-Viger et al., 2014; Fournier-Viger et al., 2016; Sethi et al., 2018). از روش‌های دیگر رایج در استخراج قوانین انجمنی برای مجموعهٔ داده‌های دلخواه می‌توان به روش رشد الگوی رایج (FP Growth) اشاره کرد. در این روش با استفاده از تعداد زیادی الگوی رایج، درخت FP ایجاد شده و با دو بار پیمایش درخت، قوانین انجمنی موجود در یک مجموعهٔ داده شناسایی می‌شوند (Li & Chang, 2004).

شرح روش پیشنهادی

روش پیشنهادی از چهار مرحله تشکیل شده که عبارت‌اند از:

۱) مستندگذاری کد.
۲) اجرای کد مستندگذاری‌شده برای مجموعه‌ای از موارد آزمون.
۳) استخراج قوانین انجمنی.
۴) تعیین محل دقیق خطا.

در ادامه هر یک از مراحل توضیح داده می‌شوند.

مستندگذاری کد

به‌منظور مستندگذاری کد، آرایه‌ای از اعداد صحیح با نام trace در کد منبع برنامه‌های مورداستفاده ایجاد شده و در پایان اجرای برنامه به ازای تمامی موارد آزمون محتوای این آرایه چاپ می‌شود. اندازهٔ این آرایه برابر با تعداد بلوک‌های سازنده کد منبع می‌باشد. درواقع هر درایه از آرایه مذکور، متناظر با یکی از بلوک‌های سازنده کد است. بلوک سازنده به دستورالعمل‌هایی گفته می‌شود که به ترتیب اجرا می‌شوند و پرشی بین آن‌ها وجود ندارد. شکل (۱) نمونه‌ای از مستندگذاری کد را نشان می‌دهد.

```
85
86  int Non_Crossing_Biased_Climb()
87  {
88      int upward_preferred;
89      int upward_crossing_situation;
90      int result;
91      trace[5] = 1;
92      upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
93      if (upward_preferred)
94      {
95          trace[6] = 1;
96          result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation > ALIM()))); /* opertor mutation */
97      }
98      else
99      {
00          trace[7] = 1;
01          result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
02      }
03      return result;
04  }
```

شکل (۱) بخشی از کد مستندگذاری‌شده

همان‌طور که در شکل (۱) ملاحظه می‌شود، تابع مربوطه دارای سه بلوک سازندهٔ کد است. یک بلوک شامل خطوط ۹۲، ۹۳، ۹۸ و ۱۰۳ می‌شود؛ یک بلوک شامل قسمت then-part دستور if (خط ۹۶) است و بلوک دیگر شامل قسمت else-part دستور if (خط ۱۰۱) می‌باشد. در هر بلوک یکی از درایه‌های آرایهٔ trace با یک مقداردهی می‌شود؛ بنابراین اگر در آرایهٔ trace درایه‌ای برابر با یک باشد، به این معنی است که بلوک سازندهٔ متناظر با آن در اجرای مورد آزمون، اجرا شده است. این آرایهٔ ابتدای برنامه با صفر مقداردهی می‌شود. در انتهای برنامه نیز، دستوری برای چاپ آرایه درج شده و براین اساس پس از خاتمهٔ اجرای برنامه به ازای هر مورد آزمون، بلوک‌های سازنده اجرا شده در هر اجرا، قابل ردگیری است.

لازم به ذکر است که فاز مستندگذاری کد به‌صورت دستی انجام می‌شود.

اجرای کد مستندگذاری‌شده

جهت اجرای برنامه به ازای موارد آزمون، اسکریپتی تهیه شده که تصویر آن در شکل (۲) نمایش داده شده است. این اسکریپت از طریق command promt ویندوز اجرا شده و نتیجهٔ آن داخل یک فایل متنی ذخیره می‌شود که بخشی از آن در شکل (۳) نمایش داده شده است.

```
tcas.exe < 958 1 1 2597  574 4253 0  399  400 0 0 1
tcas.exe < 627 0 0  621  216  382 1  400  641 1 1 0
tcas.exe < 549 1 1 4398  133 1445 1  641  639 0 0 1
tcas.exe < 576 0 1 3469  183  381 2  641  501 1 0 1
tcas.exe < 992 1 0 3342   23 4657 1  640  741 0 0 0
tcas.exe < 548 0 1   34  542 3514 2  499  401 1 1 1
tcas.exe < 710 0 0  127  403 4616 3  500  400 0 0 0
tcas.exe < 638 0 1  698  499 2465 3  500  501 0 0 0
```

شکل (۲) اسکریپت ایجاد شده برای اجرای برنامه tcas

```
tcas_v1_result.txt - Notepad
File  Edit  Format  View  Help

C:\Users\Dear User\Desktop>Untitled1.exe 958 1 1 2597  574 4253 0  399  400 0 0 1
1
1      1      1      1      1      1      1      0      1      1      0      1      1      0      1      0      0      0

C:\Users\Dear User\Desktop>Untitled1.exe  627 0 0  621  216  382 1  400  641 1 1 0
0
1      0      0      0      0      0      0      0      0      0      0      1      0      0      0      0      0      0

C:\Users\Dear User\Desktop>Untitled1.exe  549 1 1 4398  133 1445 1  641  639 0 0 1
0
1      0      0      0      0      0      0      0      0      0      0      1      0      0      0      0      0      0

C:\Users\Dear User\Desktop>Untitled1.exe  576 0 1 3469  183  381 2  641  501 1 0 1
0
1      0      0      0      0      0      0      0      0      0      0      1      0      0      0      0      0      0

C:\Users\Dear User\Desktop>Untitled1.exe  992 1 0 3342   23 4657 1  640  741 0 0 0
0
1      0      1      0      1      1      0      1      1      0      1      1      1      0      0      0      1      0
```

شکل (۳) بخشی از فایل متنی حاوی اجرای برنامه به ازای اسکریپت ایجادشده برای برنامه tcas

همان‌طور که ملاحظه می‌شود، این فایل به ازای هر اجرا شامل دستور اجراشده، خروجی برنامه و محتوای آرایهٔ trace می‌شود. در برنامهٔ tcas تعداد بلوک‌های سازنده برابر با ۱۸ بوده و در هر اجرا مشخص است که کدام یک از بلوک‌های سازنده اجرا شده‌اند. برای مثال در اجرای دوم (مورد آزمون شماره ۲) بلوک‌های سازندهٔ شماره ۱ و ۱۲ اجرا شده و مابقی بلوک‌های سازنده اجرا نشده‌اند.

استخراج قوانین انجمنی

جهت استخراج قوانین انجمنی برنامه‌ای به زبان برنامه‌نویسی جاوا در محیط برنامه‌نویسی NetBeans پیاده‌سازی شده است. به‌منظور استخراج قوانین انجمنی از الگوریتم ژنتیک استفاده شده است. مراحل طی‌شده جهت استخراج قوانین انجمنی به ترتیب زیر می‌باشد:

۱) ایجاد جمعیت اولیه: با توجه به اینکه هدف، یافتن بلوک خطادار است، در الگوریتم ژنتیک کروموزوم‌ها با آرایه‌هایی مشابه آرایه‌های رد اجرا برابر هستند؛ بنابراین برای هر برنامه، اندازهٔ کروموزوم با تعداد بلوک‌های سازندهٔ آن برابر است. این کروموزوم‌ها به‌صورت تصادفی در اولین بار (جمعیت اولیه) مقداردهی شده‌اند. اندازهٔ جمعیت اولیه برابر با ۱۰۰ در نظر گرفته شده و جهت رعایت یکنواختی جمعیت اولیه، تمام ژن‌ها شانس برابری برای ۱ شدن دارند. به این منظور برای ژن i–ام عددی تصادفی تولید می‌شود. اگر عدد تصادفی تولیدشده در بازهٔ

$(\frac{i}{\text{اندازه کروموزوم}}, \frac{i+1}{\text{اندازه کروموزوم}})$ باشد، ژن متناظر با آن برابر با ۱ و در غیر این صورت برابر با ۰ خواهد شد. قوانین انجمنی در روش پیشنهادی با هدف یافتن بلوک سازنده حاوی خطا طراحی می‌شوند؛ بنابراین هر قانون تعیین می‌کند که چه بلوک‌های سازنده‌ای با هم اجرا شده‌اند. به‌منظور ارزیابی برازندگی هر قانون از تفاوت بین اجرای بلوک‌های سازندهٔ تشکیل‌دهندهٔ قانون در اجراهای موفق و ناموفق استفاده شده است. هر میزان که مجموعه بلوک‌های سازنده در اجراهای ناموفق، بیشتر و در اجراهای موفق کمتر اجرا شده باشد، احتمال خطادار‌بودن آن‌ها بیشتر است. درواقع هر میزان پشتیبانی قانون در اجراهای ناموفق بیشتر و در اجراهای موفق کمتر باشد، قانون برازندگی بیشتری دارد. پس از آزمایش ضرایب اطمینان، پشتیبانی و گونه‌های متفاوتی از فرمول ارائه‌شده در (Altay & Alatas, 2020) فرمول جدیدی برای محاسبهٔ برازندگی به دست آمد که نتیجهٔ بسیار بهتری از موارد یادشده تولید می‌کند. فرمول (۱) فرمول استفاده‌شده در محاسبهٔ برازندگی کروموزم‌ها در روش پیشنهادی است.

$$fitness = \frac{\frac{f\_siilarity}{failed\ executions}}{\frac{f\_similarity}{failed\ executions} + \frac{p\_similarity}{passed\ executions}} \quad (۱)$$

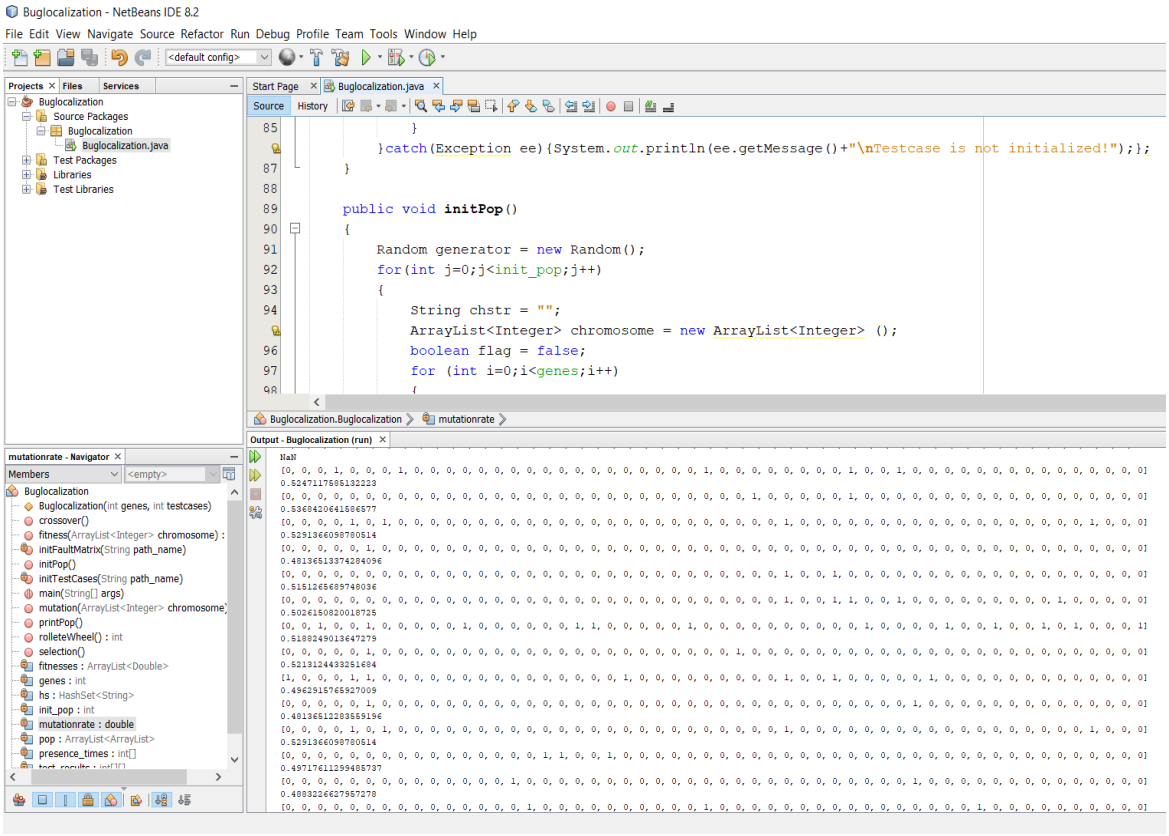
در این فرمول *f_similarity* برابر با میانگین میزان شباهت یک کروموزوم با اجراهای ناموفق و *p_similarity* برابر با میانگین میزان شباهت یک کروموزوم با اجراهای موفق و *failedexecutions* و *passedexecutions* به ترتیب برابرند با تعداد اجراهای ناموفق و موفق. حاصل فرمول پیشنهادی برای کروموزوم‌هایی که شباهت بیشتری به اجراهای ناموفق دارند عدد بزرگ‌تری تولید می‌کند. جهت بررسی شباهت بین دو کروموزوم از ضرب داخلی آن‌ها استفاده شده است. ضرب داخلی دو بردار از فرمول (۲) به دست می‌آید.

$$A.B = \sum_{=1}^{n} A_i \times B_i = |A| \times || \times \cos(\theta) \quad (۲)$$

که در آن n برابر با اندازهٔ بردار (تعداد ژن‌های یک کروموزوم)، |A(B)| برابر با اندازهٔ بردار A(B) و θ برابر است با زاویهٔ بین بردارها.

اگر هر کروموزوم به‌عنوان یک بردار در نظر گرفته شود، شباهت دو بردار را می‌توان با استفاده از فرمول (۲) به دست آورد. با توجه به آنکه هر قدر کسینوس زاویهٔ بین دو بردار بزرگ‌تر باشد، دو بردار، بیشتر به هم شباهت دارند، بنابراین با به‌دست‌آوردن کسینوس زاویهٔ بین دو بردار می‌توان میزان شباهت آن‌ها را به یکدیگر محاسبه کرد.

۲) اجرای الگوریتم ژنتیک: در این مرحله با استفاده از عملگرهای اتصال و جهش که در یک فرآیند تکراری مورداستفاده قرار می‌گیرند، چندین نسل از جمعیت حاصل شده و درنهایت آخرین جمعیت به‌عنوان نتیجهٔ نهایی حاصل می‌شوند. در هر مرحله جهت انتخاب والد و یا انتخاب نسل بعدی از الگوریتم چرخ رولت استفاده شده است. در الگوریتم پیاده‌سازی‌شده، نرخ جهش برابر با ۰/۰۶ و تعداد تکرار عملیات اتصال برابر با ۱۰۰۰ در نظر گرفته شده است. عملگر جهش با احتمال ۶ درصد پس از هر عمل اتصال انجام خواهد شد. شکل (۴) نمونه‌ای از خروجی را نمایش می‌دهد.



شکل (٤) نمونه‌ای از خروجی الگوریتم ژنتیک

در شکل (۴) به ازای هر کروموزوم مقدار تابع برازندگی آن در زیر کروموزوم نمایش داده شده است.

۳) انتخاب راه‌حل برتر: در این مرحله کروموزومی که بالاترین برازندگی را دارد به‌عنوان راه‌حل نهایی انتخاب شده و بلوک‌های سازنده که در آن مقادیری برابر با ۱ دارند، جهت یافتن خطا بررسی می‌شوند. روند بررسی بلوک‌های سازنده به این ترتیب است که ابتدا بلوک‌های موجود در کروموزوم منتخب به ترتیب نزدیکی به محل مشاهدهٔ خطا بررسی می‌شوند. اگر خطا در این بلوک‌ها نبود، بلوک‌های قبل از آن‌ها در اجراهای ناموفق بررسی می‌شود. لازم به ذکر است که برای دقیق‌ترشدن نتیجهٔ جستجو، برش‌بندی کامل بلوک‌های یافت‌شده نیز انجام (به‌صورت دستی) و نتایج بسیار خوبی حاصل کرد. دلیل عقب‌گرد از بلوک شناسایی‌شده توسط قانون انجمنی این است که بسیاری از خطاها در قسمتی از کد قرار می‌گیرند که در بسیاری از اجراهای موفق نیز اجرا می‌شوند و تنها از روی اثر آن‌ها قابل‌شناسایی هستند. برای مثال قطعهٔ کد خطادار شکل (۵) را که از برنامه schedule انتخاب شده است در نظر بگیرید.

```
192  upgrade_process_prio(prio, ratio)
193  int prio;
194  float ratio;
195  {
196      trace[23] = 1;
197      int count;
198      int n;
199      Ele *proc;
200      List *src_queue, *dest_queue;
201      if (prio >= MAXPRIO)
202      {
203          trace[24] = 1;
204          return;
205      }
206      src_queue = prio_queue[prio];
207      dest_queue = prio_queue[prio+1];
208      count = src_queue->mem_count;
209
210          if (count > 1) /* off by one */ {
211              trace[25] = 1;
212              n = (int) (count*ratio + 1);
213              proc = find_nth(src_queue, n);
214              if (proc) {trace[26] = 1;
215                  src_queue = del_ele(src_queue, proc);
216              /* append to appropriate prio queue */
217              proc->priority = prio;
218              dest_queue = append_ele(dest_queue, proc);
219          }
220      }
221  }
```

شکل (۵) قطعهٔ کد خطادار از برنامه schedule

در این قطعه، کد خطا در خط ۲۱۰ قرار دارد. خط ۲۱۰ همان‌طور که از تصویر پیداست در بلوک سازندهٔ ۲۳ قرار دارد که به ازای بسیاری از اجراهای موفق نیز اجرا می‌شود؛ بنابراین رفتار این بلوک در اجراهای موفق و ناموفق شباهت زیادی دارد. تفاوت بین اجراهای موفق و ناموفق زمانی اتفاق می‌افتد که خطا باعث می‌شود ارزیابی شرط count>1 غلط شود؛ بنابراین اگر بلوک شمارهٔ ۲۵ در روش پیشنهادی به‌عنوان یکی از بلوک‌های سازندهٔ جواب نهایی مشخص شود، با عقب‌گرد از آن (و بسیاردقیق‌تر با برش‌بندی کامل آن) خطا در خط ۲۱۰ آشکار می‌شود. روش پیشنهادی در مواردی نظیر این مورد، بسیار دقیق عمل کرده و در این مورد خاص، بلوک ۲۵ به‌عنوان نتیجهٔ ارزیابی مشخص شد.

تعیین محل دقیق خطا

در آخر آزمون‌کنندهٔ نرم‌افزار با بررسی راه‌حل‌ها به ترتیب امتیاز آن‌ها اقدام به یافتن خطا می‌کند.

نتایج به‌دست‌آمده از ارزیابی روش پیشنهادی

جهت ارزیابی روش پیشنهادی از تعدادی از برنامه‌های مجموعهٔ زیمنس استفاده شده است. برنامه‌های مورداستفاده در ارزیابی روش پیشنهادی در جدول (۱) معرفی شده‌اند.

جدول (۱) مشخصات برنامه‌های مورداستفاده در ارزیابی روش پیشنهادی

برنامه	تعداد نسخه	تعداد خطوط کد	تعداد موارد آزمون	تعداد رویه‌ها	توصیف
tcas	۴۱	۱۷۳	۱۶۰۸	۹	سیستم جلوگیری از تصادف اشیاء
print_tokens	۷	۷۲۶	۴۱۳۰	۱۸	تحلیل‌گر لغوی
schedule	۹	۳۷۰	۲۶۵۰	۱۸	زمان‌بند اولویت

برنامه‌های این مجموعه به زبان C پیاده‌سازی شده و به ازای هر یک از برنامه‌ها بسته‌ای شامل کد منبع برنامهٔ اصلی (فاقد خطا)، نسخه‌های مختلف خطادار برنامه، ماتریس شکست، موارد آزمون و غیره ارائه شده است. ماتریس شکست حاوی نتایج آزمون به ازای موارد آزمون مختلف است. در صورتی که برنامه، خروجی نادرست تولید کند (اجرای ناموفق) مقدار نتیجه برای مورد آزمون مورداستفاده برابر با ۱ و در غیر این صورت برابر با ۰ است. جدول (۲) نتایج حاصل از این ارزیابی را با استفاده و بدون استفاده از برش‌بندی کامل نمایش می‌دهد.

جدول (۲) نتایج حاصل از ارزیابی روش پیشنهادی

برنامه	تعداد خط کد	تعداد دستورهای کد برای یافتن خطا		درصد کاوش کد	
		با برش‌بندی	بدون برش‌بندی	با برش‌بندی	بدون برش‌بندی
tcas	۱۷۳	۳	۷	۰/۰۱۷	۰/۰٤
print_tokens	۷۲٦	۷	۱۳	۰/۰۰۹	۰/۰۱۷
schedule	۳۷۰	۸	۱۰	۰/۰۲۱	۰/۰۲۷

همان‌طور که ملاحظه می‌شود، بدون استفاده از برش‌بندی در بدترین حالت با کاوش تنها ۴ درصد از کد برنامه محل دقیق خطا یافت می‌شود و با استفاده از برش‌بندی این مقدار به ۲/۱ درصد تقلیل می‌یابد. با توجه به آنکه هر اندازه برنامه بزرگ‌تر باشد، به احتمال بسیار بالا به ازای موارد آزمون نتایج متفاوت‌تری تولید می‌کند، حاصل ارزیابی روش پیشنهادی نیز بهتر خواهد بود. در جدول (۳) ضرایب اطمینان برای قانون انجمنی حاصل (بهترین جواب) مشخص شده است.

جدول (۳) ضریب اطمینان قانون انجمنی حاصل

برنامه	ضریب اطمینان قانون انجمنی حاصل (قانون انجمنی با بالاترین مقدار برازندگی)
tcas	٪ ۷۵
print_tokens	٪ ۹۲/۱
schedule	٪ ۸۸/۸

همان‌طور که در جدول (۳) مشاهده می‌شود، ضریب اطمینان برای همهٔ قوانین به‌دست‌آمده عدد بالایی نمی‌باشد. دلیل این امر آن است که برخی از خطاها مانند آنچه در شکل (۵) مشاهده می‌شود در بسیاری از اجراها (موفق و ناموفق) اجرا می‌شوند و بنابراین در قانون نتیجهٔ درایهٔ متناظر با آن‌ها ۰ بوده و بلوک‌های تاثیرپذیرنده از آن‌ها یک می‌باشند؛ بنابراین لزوماً در تمام اجراهای ناموفق بلوک مشخص‌شده اجرا نمی‌شود.

بحث و نتیجه‌گیری

پیش‌تر به اهمیت فرآیند آزمون و اشکال‌زدایی نرم‌افزار در دورهٔ نگهداشت یک نرم‌افزار اشاره شد. همان‌طور که گفته شد پس از اعلام گزارش‌های خطا از سوی کاربران، آزمون‌کنندگان نرم‌افزار بر اساس گزارش‌های دریافت‌شده اقدام به مکان‌یابی خطای نرم‌افزار می‌کنند. با توجه به اینکه معمولاً نرم‌افزارها چندین هزار خط کد دارند، یافتن محل دقیق خطا به‌صورت دستی کاری بسیار زمان‌بر و خسته‌کننده است. هر روزه روش‌های جدیدی برای خودکارسازی مکان‌یابی خطا ارائه می‌شود. روش‌های ارائه‌شده ممکن است مجموعهٔ خروجی بزرگی تولید کنند، مکان خطا را به‌صورت دقیق مشخص نکنند، به اطلاعات زیادی نیاز داشته باشند و یا تعداد موارد آزمون زیادی برای اعمال آن‌ها

موردنیاز باشد؛ بنابراین در این پژوهش با بررسی موانع و مشکلات موجود سعی شده است تا روشی ارائه شود که این مشکلات را مرتفع کرده و خروجی دقیقی تولید کند.

ملاحظات مربوط به هر یک از مشکلات یادشده به‌صورت زیر می‌باشد:

۱) کوچک‌بودن مجموعهٔ خروجی: در روش پیشنهادی بلوک‌های اولیهٔ کد مستندگذاری شده و مورد ارزیابی قرار می‌گیرند. در روش پیشنهادی برای انتخاب بلوک‌های اولیه از قوانین انجمنی و برای کشف قوانین انجمنی از الگوریتم ژنتیک استفاده شده است. تابع برازندگی الگوریتم ژنتیک طوری تعریف شده که هر قانون دارای ضریب اطمینان بالایی بوده و شامل تعداد بسیار کمی بلوک اولیه می‌شود؛ بنابراین تعداد جملاتی که باید بررسی شوند بسیار کم می‌باشد. نتایج ارزیابی روش پیشنهادی نشان داد که این روش در اکثر مواقع بلوک کد حاوی خطا را به‌عنوان خروجی مشخص کرده و یا با بلوک کد حاوی خطا فاصلهٔ اجرایی بسیار کمی دارد.

۲) مشخص‌نکردن خطا به‌صورت دقیق: روش‌های آماری

از مزایای روش پیشنهادی می‌توان به موارد زیر اشاره کرد:

۱) هزینهٔ زمانی پایین اجرا: اگر از برش‌بندی استفاده نشود، با توجه به تعداد تکرار اتصال انتخاب‌شده، این روش، هزینهٔ اجرای بسیار پایینی دارد (کمتر از ۵ ثانیه). در چند اجرا تعداد تکرار عملیات اتصال برابر با مقادیر بالاتر (۱۰۰۰۰، ۱۰۰۰۰۰۰ و ۱۰۰۰۰۰۰۰۰) در نظر گرفته شد و نتیجهٔ به‌دست‌آمده نسبت به ۱۰۰۰ بار تکرار بهتر نبود؛ بنابراین با در نظر گرفتن ۱۰۰۰ بار تکرار عمل اتصال، نتیجهٔ به‌دست‌آمده قابل‌قبول بوده و هزینهٔ اجرا نیز پایین می‌باشد. تنها عملیات زمان‌بر در روش پیشنهادی مستندگذاری کد است که به‌صورت دستی انجام شده و نسبت مستقیم با اندازهٔ کد (loc) دارد.

۲) وجود تعداد کمی از بلوک‌های سازنده در کروموزوم‌ها با استفاده از تابع برازندگی پیشنهادی: با توجه به آنکه میزان برازندگی با اندازهٔ بردار متناظر با کروموزوم‌ها نسبت عکس دارد، کروموزوم‌هایی با اندازهٔ کوچک‌تر (تعداد یک کمتر) امتیاز بیشتری کسب می‌کنند و شانس بیشتری برای ورود به مرحلهٔ بعد دارند.

۳) دقت بالای روش در یافتن محدودهٔ خطا: نتایج به‌دست‌آمده از ارزیابی روش پیشنهادی نشان‌دهندهٔ دقت بالای روش در یافتن خطا است.

از معایب روش پیشنهادی می‌توان به موارد زیر اشاره کرد:

۱) دستی‌بودن مستندگذاری.

۲) متناسب‌بودن دقت روش با تعداد و پوشش موارد آزمون موفق و ناموفق. هر قدر تعداد موارد آزمون موفق و ناموفق بیشتر و پوشش کد در هر دو نوع اجرا بالاتر باشد، روش پیشنهادی نتیجهٔ دقیق‌تری خواهد داشت.

خطاهایی مانند کد جاافتاده (کدی که به اشتباه توسط برنامه‌نویس نوشته نشده است) توسط این روش پوشش داده نمی‌شوند. لازم به ذکر است که این مورد در هیچ یک از روش‌های خطایابی پویا پوشش داده نمی‌شود.

منابع

میرزایی، کمال و محمودی، سید مصطفی. (۱۳۹۴). استفاده از قوانین انجمنی جهت کشف عوامل خطر در بروز سرطان معده، *مجلهٔ انفورماتیک سلامت و زیست پزشکی*، ۱(۲)، ۹۵-۱۰۳.

Agrawal, H., & Horgan, J. R. (1990). Dynamic program slicing. *ACM SIGPlan Notices*, *25*(6), 246-256.

Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In Proc. 20th int. conf. very large data bases, VLDB (Vol. 1215, pp. 487-499).

Aho, A. V., Sethi, R., & Ullman, J. D. (2007). *Compilers: principles, techniques, and tools* (Vol. 2). Reading: Addison-wesley.

Altay, E. V., & Alatas, B. (2020). Intelligent optimization algorithms for the problem of mining numerical association rules. *Physica A: Statistical Mechanics and its Applications*, *540*, 123142.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Baró, G. B., Martínez-Trinidad, J. F., Rosas, R. M. V., Ochoa, J. A. C., González, A. Y. R., & Cortés, M. S. L. (2020). A PSO-based algorithm for mining association rules using a guided exploration strategy. *Pattern Recognition Letters*, *138*, 8-15.

Fournier-Viger, P., Lin, J. C. W., Duong, Q. H., & Dam, T. L. (2016, July). FHM: Faster high-utility itemset mining using length upper-bound reduction. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems* (pp. 115-127). Cham: Springer International Publishing.

Fournier-Viger, P., Wu, C. W., Zida, S., & Tseng, V. S. (2014). FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Foundations of Intelligent Systems: 21st International Symposium, ISMIS 2014, Roskilde, Denmark, June 25-27, 2014. Proceedings 21* (pp. 83-92). Springer International Publishing.

Hildebrandt, R., & Zeller, A. (2000, August). Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* (pp. 135-145).

Jones, J. A., & Harrold, M. J. (2005, November). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 273-282).

Kantardzic, M. (2011). *Data mining: concepts, models, methods, and algorithms*. John Wiley & Sons.

Lam, A. N., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2017, May). Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (pp. 218-229). IEEE.

Li, Y. C., & Chang, C. C. (2004, November). A new FP-tree algorithm for mining frequent itemsets. In *Advanced Workshop on Content Computing* (pp. 266-277). Berlin, Heidelberg: Springer Berlin Heidelberg.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. *Acm Sigplan Notices*, *40*(6), 15-26.

Liu, C., Yan, X., Fei, L., Han, J., & Midkiff, S. P. (2005). SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, *30*(5), 286-295.

Liu, Y., Liao, W. K., & Choudhary, A. (2005, August). A fast high utility itemsets mining algorithm. In *Proceedings of the 1st international workshop on Utility-based data mining* (pp. 90-99).

Narvekar, M., & Syed, S. F. (2015). An optimized algorithm for association rule mining using FP tree. *Procedia Computer Science*, *45*, 101-110.

Ottenstein, K. J., & Ottenstein, L. M. (1984). The program dependence graph in a software development environment. *ACM Sigplan Notices*, *19* (5), 177-184.

Parsa, S., Vahidi-Asl, M., & Zareie, F. (2015). Statistical based slicing method for prioritizing program fault relevant statements. *Computing and Informatics*, *34*(4), 823-857.

Parsa, S., Zareie, F., & Vahidi-Asl, M. (2011). Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure. In *Experimental Algorithms: 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings 10* (pp. 352-363). Springer Berlin Heidelberg.

Sethi, K. K., Ramesh, D., & Edla, D. R. (2018). P-FHM+: Parallel high utility itemset mining algorithm for big data processing. *Procedia computer science*, *132*, 918-927.

Tip, F. (1994). *A survey of program slicing techniques* (p. 58). Amsterdam: Centrum voor Wiskunde en Informatica.

Wang, Y., Yao, Y., Tong, H., Huo, X., Li, M., Xu, F., & Lu, J. (2020). Enhancing supervised bug localization with metadata and stack-trace. *Knowledge and Information Systems*, *62*, 2461-2484.

Zareie, F., & Parsa, S. (2013). A non-parametric statistical debugging technique with the aid of program slicing (NPSS). *International Journal of Information Engineering and Electronic Business*, *5* (2), 8.

استناد به این مقاله: زارعی، فرزانه. (۱۴۰۲). خطایابی کد با استفاده از الگوریتم ژنتیک. *فصلنامه پژوهش‌های نوین در شهر هوشمند*، ۱(۴)، ۵۳-۶۷.