نشریه علمی- ترویجی محاسبات نرم
شماره‌ی سوم/ بهار و تابستان ۱۳۹۲/ صفحه ۴۴ـ۵۹

تاریخ دریافت مقاله: خرداد ۱۳۹۲
تاریخ پذیرش مقاله: بهمن ۱۳۹۲

مکان‌یابی خطاهای پنهان نرم‌افزار با استفاده از آنتروپی متقاطع و مدل‌های N-گرام

مجید حاجی‌بابا[۱]، سعید پارسا[۲]

[۱]کارشناسی ارشد، دانشکده کامپیوتر، دانشگاه علم و صنعت، تهران، ایران
mhajibaba@comp.iust.ac.ir*
[۲]دانشیار، دانشکده کامپیوتر، دانشگاه علم و صنعت، تهران، ایران
parsa@iust.ac.ir

چکیده: هدف، ارائه‌ی راهکاری برای تعیین خودکار محدوده‌ی خطاهای پنهان در متن برنامه‌هاست. می‌توان محدوده‌ی علت خطا را براساس مقایسه‌ی مسیرهای اجرایی صحیح و غلط به‌دست آورد. براساس شباهت مسیرهای اجرایی، می‌توان آن‌ها را دسته بندی کرد. برای به‌دست‌آوردن شباهت مسیرها، مدل‌های N-گرام اجراها را به‌دست می‌آوریم و سپس با استفاده از آنتروپی متقاطع، شباهت بین این مدل‌ها را محاسبه می‌کنیم. برای به‌دست‌آوردن مدل‌های N-گرام که در دسته مدل‌های مارکوف قرار می‌گیرند، از تخمین حداکثر احتمال به‌وسیلهٔ شمارش کلمات یا به‌عبارتی N-گرام‌ها استفاده می‌شود. سپس با تحلیل هر دسته، با کمک آنتروپی متقاطع، یک‌سری مکان‌های مشکوک به‌خطا شناسایی می‌شوند و درنهایت با استفاده از رأی اکثریت بین دسته‌ها، مکان‌های مشکوک به خطا به‌صورت بخش‌هایی از یک زیرمسیر به برنامه‌نویس معرفی می‌شود. راهکار ارائه‌شده در این مقاله، با دقت بالا محدوده‌ی خطا را نشان می‌دهد و نتایج به‌دست‌آمده از اِعمال این راهکار به مجموعه محک زیمنس، گویای آن است.

واژه‌های کلیدی: مکان‌یابی خطا، خطای نرم‌افزار، آنتروپی متقاطع، مدل N-گرام.

۱ . مقدمه

امروزه مشخص شده بخش اعظم بازهزینه‌ی تولید نرم‌افزار، صرف اشکال‌زدایی و رفع خطا[1]های پنهان می‌شود [۱]. خطای پنهان به آن دسته از خطاها اطلاق می‌شود که اجرای برنامه را متوقف نمی‌کند؛ بلکه فقط منجر به نتیجه‌ی نادرست در برنامه می‌شود. بسیاری از خطاهای پنهان نرم‌افزار مشکل‌ساز هستند؛ اما برخی از آن‌ها ممکن است پیامدهای بسیار ناگواری نظیر تبعات جانی یا هزینه‌های گزاف مالی داشته باشند که انفجار موشک آریان ۵، کشته‌شدن سه بیمار توسط دستگاه پرتودرمانی تراک-۲۵ و ازکارافتادن شبکه‌ی تلفنی AT&T نمونه‌هایی ازآن‌هاست [۲،۳].

برای یافتن این دسته از خطاها، اکثر روش‌های موجود، مبتنی بر مقایسه‌ی اجراهای موفق و ناموفق برنامه‌هاست. درواقع، مکان خطا را در ناحیه‌ای از کد باید جویا شد که عامل تفکیک اجرای خطادار از اجرای صحیح است.

مسیرهای اجرایی برنامه را می‌توان در قالب دنباله‌ای از تعیین‌کننده‌ها مشخص کرد. منظور از تعیین‌کننده[2] نقاطی از برنامه است که انعکاس‌دهنده‌ی عملکرد و رفتار برنامه است و رفتار برنامه براساس ارزیابی این نقاط در طی اجراهای مختلف تحلیل می‌شود؛ برای نمونه یک جمله *if* را می‌توان به عنوان یک تعیین‌کننده درنظر گرفت. برای به‌دست‌آوردن اطلاعات لازم از مسیرهای اجرایی برنامه، کدهایی در این نقاط به برنامه‌ی اصلی اضافه می‌شود که اطلاعات مربوط به تعیین‌کننده ها را جمع‌آوری می‌کند. به این کار، اصطلاحاً مستندگذاری[3] می‌گویند.

براساس آمار به‌دست آمده از اجراهای موفق و ناموفق یک برنامه‌ی مستندگذاری‌شده، می‌توان مدل رفتاری برنامه را ایجاد کرد. در این راستا، می‌توان روش‌هایی را نام برد که از مدل‌های رگرسیون، مدل‌های احتمالی و مدل‌های مبتنی بر گراف استفاده می‌کنند.

مدل‌های رگرسیون [۴،۵،۶] از تحلیل رگرسیون برای مکان‌یابی خطا استفاده می‌کنند. مزیت این مدل‌ها درنظرگرفتن وابستگی بین تعیین‌کننده‌ها با نتیجه‌ی برنامه است؛ درحالی‌که در بیشتر روش‌ها، تعیین‌کننده‌ها مستقل از هم فرض می‌شوند. مشکل مدل‌های رگرسیون، محاسبات سنگین و درنتیجه مقیاس‌ناپذیربودن آن‌هاست.

مدل‌های مبتنی بر گراف [۷،۸] که در آن هر گراف نشان‌دهنده‌ی یک اجراست و گره‌های آن تعیین‌کننده‌ها هستند، به‌دنبال زیرگراف متمایزکننده‌ی اجراهای صحیح و غلط هستند. این مدل‌ها به‌خوبی بر مشکلی که حلقه‌های طولانی در محاسبات به‌وجود می‌آورند، فائق آمده‌اند. این روش‌ها نیز رابطه‌ی بین تعیین‌کننده‌ها را درنظر می‌گیرند؛ ولی همچنان مقیاس‌پذیر نیستند.

مدل‌های احتمالی [۹،۱۰،۱۱] از احتمال شرطی برای نسبت‌دادن تعیین‌کننده‌ها یا دستورالعمل‌ها با خطادار‌بودن اجرای برنامه استفاده می‌کنند. این مدل‌ها با اتکا به رابطه‌های آماری و احتمالی، توانایی بالایی در تشخیص خطاها دارند. مشکل این روش‌ها درنظرنگرفتن وابستگی هم‌زمان تعیین‌کننده‌ها به یکدیگر و ناتوانی در فراهم‌سازی زمینه‌ی خطاست. زمینه‌ی خطا، دنباله‌ای از تعیین‌کننده‌های مرتبط با خطا است که به کاربر در کشف ریشه‌ی خطا کمک می‌کند.

روشی که در این مقاله ارائه می‌شود، به نوعی ترکیب چند مدل است. دراین روش از احتمال MLE (تخمین حداکثر احتمال[4]) برای به‌دست‌آوردن مدل N-گرام یک اجرا استفاده می‌کند که حالتی از مدل مارکوفیاست و از آنتروپی متقاطع[5]

1. Fault
2. Predicate
3. Instrumentation
4. Maximum Likelihood Estimation
5. Cross Entropy

برای به‌دست‌آوردن تفاوت دو مدل احتمالی استفاده می‌کند؛ شبیه به آنچه در مدل‌های مبتنی بر گراف برای تعیین تمایز گراف‌ها استفاده می‌شود. استفاده از N-گرام‌ها سبب می‌شود وابستگی بین تعیین‌کننده‌ها نیز درنظر گرفته شود. برای حل مشکل مقیاس‌پذیری در این روش، از دسته‌بندی استفاده شده است.

این روش در ابتدا اجراها را دسته‌بندی می‌کند؛ به‌طوری‌که اجراهای دارای رفتار مشابه در یک دسته قرار گیرند. برای به دست‌آوردن شباهت بین اجراها به‌منظور دسته‌بندی، از مبحث آنتروپی کمک گرفته شده است که از مباحث اساسی در نظریه‌ی اطلاعات است. آنتروپی در مباحث مختلف، ازجمله داده‌کاوی، بیوانفورماتیک و دیگر رشته‌ها، خصوصاً در به دست‌آوردن شباهت بین پروتئین‌ها و بین رشته‌های DNA توسط نوع توسعه‌یافته‌ای از آن به نام آنتروپی متقاطع، کاربردهایی فراوان دارد. پس از دسته‌بندی اجراها با تحلیل هر دسته، یعنی مقایسه‌ی زیرمسیرها، باز هم با کمک آنتروپی متقاطع، زیرمسیرهایی که بیشترین شباهت به اجراهای اشتباه و بیشترین تفاوت با اجراهای صحیح را دارند، به‌عنوان مکان‌های مشکوک به خطا معرفی می‌شوند. پس از تحلیل دسته‌ها، جواب‌هایی که هر دسته به‌عنوان مسیرهای مشکوک به خطا اعلام می‌کنند، ممکن است دارای اولویت‌های متفاوتی باشند و حتی در برخی موارد کاملاً متفاوت باشند؛ بنابراین، با استفاده از رأی اکثریت بین دسته‌ها، زیرمسیرهای مشکوک به خطا به ترتیب اولویت معرفی خواهند شد.

در بخش‌های بعدی این مقاله، ضمن ارائه‌ی مفاهیم اولیه، راهکار پیشنهادی، ارزیابی روش با استفاده از مجموعه محک زیمنس[1] و درنهایت نتیجه‌گیری ارائه خواهد شد.

۲ . ادبیات موضوع

قبل از ارائه‌ی راهکار پیشنهادی، لازم است برخی از مفاهیم مورد استفاده در این راهکار و برخی از راهکارهای موجود در مکان‌یابی خطا توضیح داده شود.

۱.۲ . تعیین‌کننده، مستندگذاری و مورد آزمون

تعیین‌کننده، گزاره‌ای منطقی درباره‌ی خاصیتی از برنامه است. هر تعیین‌کننده در طی اجرای برنامه، ممکن است چندین بار ارزیابی شود. مقادیر حاصل از ارزیابی تعیین‌کننده‌ها، درست یا نادرست هستند. انتخاب نقاطی از برنامه به‌عنوان تعیین‌کننده که بتوانند به‌طور کارآمد رفتار برنامه را منعکس کند، مسئله‌ای مهم است. مهم‌ترین نقاط برنامه که برای تشخیص عملکرد برنامه، به‌عنوان تعیین‌کننده در این راهکار انتخاب شده‌اند، عبارت‌اند از: عبارات شرطی، مقادیر بازگشتی و ورودی توابع.

هدف از مستندگذاری، جمع‌آوری اطلاعات از رفتارهای اجرایی برنامه است. در فرآیند مستندگذاری، دستورات اضافی در کد برنامه در محل‌هایی که به‌عنوان تعیین‌کننده شناسایی شده‌است، وارد می‌شود. درواقع، با استفاده از مستندگذاری می‌توان اطلاعات مهمی مانند مسیر اجرایی برنامه و تعداد دفعات اجرای جمله یا مسیری در برنامه را، در طی اجرا و براساس مجموعه داده ورودی تشخیص داد. اطلاعات حاصل از مستندگذاری، ذخیره و سپس برای تحلیل رفتار برنامه استفاده می‌شود. مستندگذاری کاربردهای زیادی نظیر اندازه‌گیری میزان پوشش آزمون، تشخیص ناهنجاری در جریان داده و مکان‌یابی خطا دارد. عملیات مستندگذاری می‌تواند منجر به تأثیرات جانبی نامطلوبی بر روی برنامه‌ی اصلی شود؛ یعنی به تغییر رفتار اصلی برنامه در زمان اجرا منجر شود؛ بنابراین، مسئله‌ی مهم در مستندگذاری این است که اطمینان یابیم که جملات مستندگذاری اضافه‌شده به برنامه تأثیری بر رفتار اصلی آن نداشته باشد.

1. Siemens Benchmark

مورد آزمون[1]، مجموعه‌ای از شرایط یا متغیرهاست که از داده‌های ورودی و وضعیت خاتمه‌ی اجرای برنامه که می‌تواند موفق یا ناموفق باشد، تشکیل شده است و تعیین خواهد کرد که آیا برنامه براساس ورودی‌های مربوطه به‌درستی عمل می‌کند یا خیر. سازوکاری که تعیین می‌کند نرم‌افزار یا سیستم، مورد آزمونی را گذرانده یا شکست خورده، به‌عنوان پیشگوی آزمون[2] شناخته می‌شود. ممکن است موارد آزمون زیادی برای تعیین درست کارکردن برنامه‌ای مورد نیاز باشد. موارد آزمون برنامه معمولاً درون مجموعه‌ای آزمون[3] جمع‌آوری می‌شوند. راهکارهای مکان‌یابی خطا از روش‌های مختلفی برای شناسایی بخشی از برنامه (معمولاً قسمت معیوب آن) براساس اطلاعات مربوط به اجراهای آن برنامه با استفاده از مجموعه آزمون بهره می‌گیرند. تأثیری که مجموعه آزمون در مؤثربودن راهکارهای مکان‌یابی خطا دارند، امری بدیهی و مورد بررسی بسیاری از محققان است. در این مقاله، برای ارزیابی راهکار ارائه‌شده، از مجموعه آزمون زیمنس استفاده شده است.

۲.۲ . مدل‌های N-گرام

به‌طور ساده N-گرام به رشته‌ای به طول N واحد گفته می شود که واحد آن بسته به کاربرد می‌تواند حرف، کلمه یا هر چیز دیگری باشد. در مدل‌های N-گرام، هدف این است که احتمال کلمه w با پیشینه h، یعنی $P(w|h)$ را به‌دست آوریم؛ برای مثال، فرض کنید پیشینه h رشته کلمات "*water is so transparent that*" می‌باشد و می‌خواهیم بدانیم که احتمال اینکه کلمه‌ی بعدی *the* باشد چقدر است:

$$P\ (the \mid water\ is\ so\ transparent\ that) \qquad (۱)$$

برای محاسبه این احتمال، یک راه این است که آن را از تعداد تکرارهای مرتبط تخمین بزنیم؛ برای مثال، فرض کنید که پیش‌نویسی بسیار بزرگ داریم. تعداد دفعاتی را که رشته کلمات "*water is so transparent that*" مشاهده می‌شود، می شماریم و سپس تعداد دفعاتی را که همین رشته کلمات به‌همراه *the* بعد از آن آمده است، می‌شماریم؛ بنابراین، احتمال مذکور برابر است با نسبت مشاهدات پیشینه h به مشاهدات پیشینه h به همراه w بعد از آن:

$$P(the|waterissotransparentthat) = \frac{C(waterissotransparentthatthe)}{C(waterissotransparentthat)} \qquad (۲)$$

به‌طور رسمی، برای نمایش احتمال متغیری تصادفی مانند X_i که مقدار *"the"* را بگیرد از $P(X_i="the")$ یا به‌طور مختصر از $P(the)$ استفاده می‌شود و برای نمایش یک رشته متشکل از n کلمه از $w_1 \dots w_n$ یا w_1^n استفاده می‌شود. برای احتمال توأم کلمات در یک رشته، با مقداری خاص برای هر متغیر تصادفی، یعنی $P(X=w_1,Y=w_2,Z=w_3,\dots)$، از $P(w_1,w_2,w_3,\dots)$ استفاده می‌شود. یک روش برای محاسبه‌ی احتمال توأم یک رشته مانند $P(w_1,w_2,w_3,\dots)$، این است که می‌توان آن را با استفاده از قانون زنجیره‌ای احتمال تجزیه کرد [۱۲]. با اعمال قانون زنجیره‌ای به کلمات [۱۲]، خواهیم داشت:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)\dots P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned} \qquad (۳)$$

که در آن w_i معادل کلمه، w_n معادل n امین کلمه یا توکن موجود در رشته و w_1^n معادل $\{w_1 w_2 \dots w_n\}$ یعنی رشته متشکل از کلمه‌های w_1 تا w_n است؛ اما در اینجا قانون زنجیره‌ای کمک چندانی نمی‌کند؛ چراکه هیچ راهی برای

1. Test Case
2. Test Oracle
3. Test Suit

محاسبه‌ی دقیق احتمال یک کلمه با داشتن پیشینه‌اش، یعنی $P(w_n|w_1^{n-1})$، وجود ندارد. تنها می‌توان با استفاده از شمارش تعداد دفعات ظاهرشدن هر کلمه، بعد از یک جمله، این احتمال را تخمین زد. ساده‌ترین راه برای تخمین احتمال N–گرام، تخمین حداکثر احتمال یا به‌اختصار MLE است. برای توکن‌های مدل N–گرام، تخمین MLE را می‌توان به‌وسیله‌ی شمارش تعداد کلمات و سپس با تقسیم بر تعداد کل، نرمال کرد تا مقداری بین ۰ و ۱ داشته باشد [۱۲]. رابطه‌ی (٤) احتمال N–گرام را با استفاده از این تخمین نشان می‌دهد.

$$P(w_n|w_1^{n-1}) = \frac{C(w_1^{n-1}w_n)}{C(w_1^{n-1})} = \frac{C(w_1^n)}{C(w_1^{n-1})} \quad (٤)$$

که در آن $C(w_1^n)$ تعداد جمله‌های متشکل از w_1 تا w_n است.

مبنای مدل N–گرام این است که به‌جای محاسبه‌ی احتمال کلمه‌ای که پیشینه‌ی آن داده شده است، پیشینه با استفاده از چند کلمه‌ی آخرش تقریب زده شود؛ برای مثال، در مدل بایگرام[1] (۲–گرام)، احتمال یک کلمه با وجود پیشینه‌اش $P(w_n|w_1^{n-1})$ را با استفاده از احتمال شرطی کلمه‌ی ماقبل آن $P(w_n|w_{n-1})$ تقریب می‌زنند؛ به عبارت دیگر، به‌جای محاسبه احتمال شرطی:

$$P(the|\ water\ is\ so\ transparent\ that) \quad (٥)$$

از تقریب

$$P(the|that) \quad (٦)$$

استفاده می‌کنیم.

این فرض که احتمال هر کلمه تنها به کلمه‌ی قبلی آن وابسته است، فرض مارکوف[2] نامیده می‌شود. مدل‌های مارکوف گونه‌ای از مدل‌های احتمالی هستند؛ با این فرض که می‌توان احتمال برخی از پدیده‌ها را بدون دانستن تاریخچه‌ی خیلی دور آن‌ها، تنها با دانستن حالت قبلی، پیش‌بینی کرد. می‌توان بایگرام را که تنها به یک کلمه‌ی قبلی نگاه می‌کند، به تریگرام[3] که به دو کلمه بلافصل قبل از خود نگاه می‌کند و درنهایت به N–گرام که به N-1 کلمه‌ی ماقبل نگاه می‌کند، تعمیم داد؛ بنابراین، رابطه‌ی کلی برای تقریب N–گرام به احتمال شرطی کلمه‌های ماقبل در یک دنباله، به صورت زیر است:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}) \quad (٧)$$

که در آن N طول مدل و n شماره‌ی کلمه است؛ برای مثال، نمونه رشته‌ی "SAQXYAQUZ" دارای کلمه‌های {SA, AQ, QX, XY, YA, QU, UZ} و دارای توکن‌های {w_1=S,w_2=A,w_3=Q,w_4=X,…,w_9=Z} در مدل ۲–گرام (N=2) و دارای توکن‌های {SAQ, AQX, QXY,…, AQU,…} در مدل ۳–گرام (N=3) است.

با استفاده از فرض بایگرام برای احتمال هر کلمه، می‌توان احتمال یک دنباله از کلمه‌ها را با استفاده از تقریب رابطه (٧) در رابطه (٣) به‌دست آورد:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \quad (٨)$$

برای به‌دست‌آوردن احتمال توکن AQX در مدل ۳–گرام مثال بیان‌شده که در آن پیشینه h رشته کلمات "AQ" است، می‌خواهیم بدانیم احتمال اینکه کلمه‌ی بعدی X باشد، یعنی $P(X|AQ)$ چقدر است. برای محاسبه‌ی مدل ۳–گرام (N=3) براساس فرض مارکوف خواهیم داشت:

$$P(w_n|w_{n-N+1}^{n-1}) = P(w_n|w_{n-2}^{n-1}) \quad (٩)$$

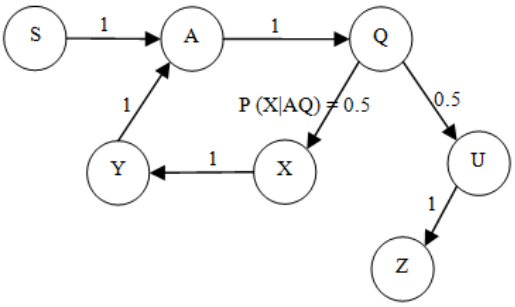
بنابراین، در احتمال $P(X|AQ)$ خواهیم داشت n=4 (با شروع از ۱) و به‌منظور محاسبه‌ی آن تعداد کل کلمات AQX را می‌شماریم و بر تعداد کل کلمات AQ تقسیم می‌کنیم:

1. Bigram
2. Markov
3. Trigram

$$p(X|AQ) = P(w_4|w_2^3) = \frac{count(AQX)}{count(AQ)} = \frac{1}{2} \quad (۱۰)$$

که احتمال ۳-گرام AQX در رشته مفروض است.

با به‌دست‌آوردن احتمال MLE برای همه‌ی توکن‌ها، درنهایت مدل ۳-گرام به‌دست‌آمده به‌صورت شکل (۱) خواهد بود.



شکل (۱): مدل ۳-گرام رشته SAQXYAQUZ

همان‌طور که در شکل (۱) مشاهده می‌شود، حالت X با پیشینه A,Q در مدل ۳-گرام، دارای احتمال وقوع ۰.۵ است که قبلاً توسط رابطه (۱۰) به‌دست آوردیم. از آنجایی‌که مدل ۳-گرام است، S به‌عنوان پیشینه‌ی X به‌حساب نمی‌آید.

۳.۲ . آنتروپی و آنتروپی متقاطع

آنتروپی مقدار بی‌نظمی ذاتی توزیع احتمال (یا مجموعه‌ای از داده‌های مشاهده‌شده) است و مفهومی کلیدی در شاخه‌ای از علم است که نظریه‌ی اطلاعات[1] نامیده می‌شود. متغیر تصادفی X و تابع احتمال آن $P(x)$ را درنظر بگیرید. آنتروپی به‌معنای خطای تخمینی متغیری تصادفی که شانون[2] آن را در سال ۱۹۸٤ ارائه داد، به‌صورت زیر است [۱۳]:

$$H(p) = H(X) = -\sum_{x \in X} p(x) \log_2 p(x) \quad (۱۱)$$

که در آن P تابع توزیع احتمال، X متغیر تصادفی و x مقادیر مختلف برای متغیر تصادفی هستند. با کمک آنتروپی می توان مدل‌هایی را که برای تشریح رفتار زبان ساخته شده‌اند، بهتر کرد؛ ولی مواردی وجود دارد که می‌خواهیم چندین توزیع با هم مقایسه شوند تا مشخص شود کدام‌یک به هم نزدیک‌تر هستند و شباهت رفتاری بیشتری دارند. در اینجا می‌توان از آنتروپی متقاطع استفاده کرد که اجازه می‌دهد دو تابع احتمال با هم مقایسه شوند. آنتروپی متقاطع بین دو توزیع p و q بر روی یک فضای احتمال به‌صورت رابطه (۱۲) است [۱۳]:

$$H(p,q) = E_p[-\log q] = H(p) + D_{KL}(p||q) \quad (۱۲)$$

که در آن E مقدار مورد انتظار تابع p، $H(p)$ آنتروپی p و $D_{KL}(p||q)$ فاصله‌ی کولبک- لیبلر[3] توزیع q از p است. فاصله‌ی کولبک- لیبلر (که به‌عنوان آنتروپی نسبی[4] هم شناخته می‌شود)، بین دو تابع چگالی احتمال g و h توسط رابطه (۱۳) محاسبه می‌شود [13,14]:

$$\mathcal{D}(g,h) = \mathbb{E}_g\left[\ln\frac{g(X)}{h(X)}\right] = \int g(x)\ln\frac{g(x)}{h(x)}dx = \int g(X)\ln g(x)\,dx - \int g(x)\ln h(x)\,dx \quad (۱۳)$$

که در آن g و h دو تابع مفروض و E مقدار مورد انتظار تابع g است. اگر g و h توزیع‌های گسسته باشند، خواهیم داشت:

$$\mathcal{D}(g,h) = \mathbb{E}_g\left[\ln\frac{g(X)}{h(X)}\right] = \sum_x g(x)\ln\frac{g(x)}{h(x)} = \sum_x g(X)\ln g(x) - \sum_x g(x)\ln h(x) \quad (۱٤)$$

که با توجه به رابطه‌ی (۱۱) خواهیم داشت:

$$\mathcal{D}(g,h) = -H(g) - \sum_x g(x)\ln h(x) \quad (۱۵)$$

که در آن $H(g)$ آنتروپی توزیع احتمال g است. با جاگذاری رابطه‌ی (۱۵) درون رابطه‌ی (۱۲)، رابطه‌ی (۱٦)

1. Information Theory
2. Shannon
3. kullback-Leibler
4. Relative Entropy

به‌دست خواهد آمد که آنتروپی متقاطع توزیع p و q را نشان می‌دهد.

$$H(p,q) = -\sum_{x \in X} p(x) \log q(x) \quad (۱۶)$$

یکی از ویژگی‌های آنتروپی متقاطع، متقارن نبودن آن است، یعنی $H(p,q)$ با $H(q,p)$ لزوماً برابر نیست.

$$H(p,q) \neq H(q,p) \quad (۱۷)$$

در تعیین آنتروپی متقاطع، مواردی وجود دارد که توزیع احتمال p ناشناخته است؛ مثلاً در مدل‌سازی زبان، یک مدل براساس یک مجموعه آموزشی[1] ساخته می‌شود و آنتروپی متقاطع آن بر روی یک مجموعه‌ی آزمایشی برای ارزیابی دقت مدل در پیش‌بینی داده‌های آزمایشی، اندازه‌گیری می‌شود. در این حالت، p توزیع حقیقی از کلمات و q توزیع کلمات برای پیش‌بینی توسط مدل است. از آنجایی‌که توزیع حقیقی ناشناخته است، نمی‌توان برای آن آنتروپی متقاطع را حساب کرد. در این حالت، آنتروپی به‌صورت رابطه‌ی (۱۸) تخمین زده می‌شود [۱۳]:

$$H(T,q) = -\sum_{i=1}^{S} \frac{1}{S} \log q(x_i) \quad (۱۸)$$

که در آن S اندازه مجموعه آزمایشی و $q(x)$ احتمال رخداد x است که توسط مجموعه آموزشی به‌دست آمده است. به این ترتیب، می‌توان از آنتروپی متقاطع برای مقایسه‌ی مدل های تقریبی استفاده کرد.

در محاسبه‌ی آنتروپی متقاطع، معمولاً آرگومان اول (p) را به‌عنوان توزیع احتمال هدف و آرگومان دوم (q) را به‌عنوان تخمینی از آنچه می‌خواهیم تناسب آن را با هدف ارزیابی کنیم، درنظر می‌گیریم. ضمناً اگر p=q باشد، آنتروپی متغیر تصادفی X حاصل می‌شود که در معادله (۱۱) بیان شد؛ بنابراین، می‌توان از آنتروپی متقاطع برای مقایسه‌ی مدل‌های تقریبی استفاده کرد.

1. Training Set

هرچه آنتروپی متقاطع به آنتروپی نزدیک‌تر باشد، مدل (q) تقریب بهتری از هدف (p) است [۱۲،۱۳].

۳ . راهکارهای مرتبط

تاکنون کارهای بسیاری در زمینه‌ی کشف خطای پنهان نرم افزار صورت گرفته است که در این بخش به مهم‌ترین آن‌ها و برخی دیگر که شباهت‌هایی به راهکار پیشنهادی دارند، پرداخته خواهد شد.

در [۱۵] از برش‌بندی استفاده شده است که روش معمول مورد استفاده برای اشکال‌زدایی است. این روش و روش‌های مشابه با کاهش دامنه‌ی جستجو از طریق برش، سعی در یافتن محل خطا دارند. اگر آزمایش به‌علت مقدار نادرست یک متغیر در یک دستور دچار شکست شود، آنگاه خطا را می‌توان در برش ایستای مرتبط با جفتِ متغییر– دستور مربوط به آن متغیر یافت. مشکل استفاده از برش ایستا این است که دستوراتی را بیرون می‌کشد که بر متغیرهای مورد استفاده برای هر ورودی تأثیرگذار است، به‌جای آنکه دستوراتی را بیرون کشد که درواقع بر متغیرهای مورد استفاده‌ی یک ورودی خاص تاثیرگذار است. برای حذف چنین دستورات اضافه‌ای از برش، نیاز به استفاده از برش پویا به‌جای برش ایستا است. مطالعاتی مانند [۱۶] و [۱۷] از برش پویا برای اشکال‌زدایی برنامه استفاده کرده‌اند. مشکل استفاده از برش پویا هم این است که گردآوری آن‌ها بیش از حد ممکن زمان و فضا مصرف می‌کند، اگرچه الگوریتم‌هایی برای حل این مسائل پیشنهاد شده‌اند.

برخی، از راهکارهای از طیف برنامه به‌منظور اشکال‌زدایی استفاده می‌کنند. یک طیف برنامه، اطلاعات مربوط به جنبه‌های خاصی از رفتار پویای برنامه در هر آزمون، از جمله نحوه‌ی اجرای دستورها و انشعاب‌های شرطی در زمان اجرای برنامه را ثبت می‌کند. تارانتولا [۹] یکی از شناخته‌شده‌ترین راهکارهای

مکان‌یابی خطا براساس طیف دستورات اجرایی است. این روش‌طیف اجراهایی را که خطا در آن‌ها مشاهده شده و طیف اجراهایی را که هیچ خطایی در آن‌ها مشاهده نشده با هم مقایسه می‌کند تا تفاوت‌ها را به‌دست آورد.

لیبلیت[1] [۱۰] و سوبر[2] [۱۱] دو روش معروف براساس راهکارهای آماری هستند. این روش‌ها بر مستندگذاری و ارزیابی تعیین‌کننده‌های درون برنامه برای رتبه‌بندی تعیین‌کننده های مشکوک به خطا تکیه دارند که برای پیداکردن خطاها بهتر است بازبینی شوند. همچنین این روش‌ها محدود به یافتن خطاها در حیطه‌ی تعیین‌کننده‌ها هستند و هیچ روشی برای نشان‌دادن مشکوک‌بودن هریک از دستورات برنامه ارائه نمی‌دهند و فقط حیطه‌ی آن را نشان می‌دهند. این راهکارها استفاده گسترده‌ای پیدا کرده‌اند. در [۱۸] راهکار آماری دیگری مبتنی بر N-گرام ارائه شده است که در آن مدل N-گرام براساس حرف به‌جای کلمه است و ادعا شده است که این عمل دقت بالاتری دارد. این روش از مستندگذاری استفاده نمی‌کند و با تحلیل بر روی گزارش‌های خطا و کد منبع به‌طور جداگانه و استخراج N-گرام‌ها در قالب مجموعه‌ای از حروف، نتایج را براساس ضریب تشابه بین این مجموعه‌ها رتبه‌بندی می‌کند.

زِلر[3] و همکاران، یک روش اشکال‌زدایی مبتنی بر حالت برنامه به‌منظور کاهش علل شکست برنامه به مجموعه‌ای کوچک از متغیرها پیشنهاد کردند (اشکال‌زدایی دلتا [۱۹]). حالت برنامه مجموعه‌ای از متغیرها به‌همراه مقادیرشان در نقطه‌ای خاص در طول اجرای برنامه است. این روش حالت‌های برنامه را بین آزمایش‌های موفق و شکست‌خورده از طریق نمودارهای حافظه آن‌ها مقایسه می‌کند. براساس اشکال‌زدایی دلتا، زلر و کلیو[4] [۲۰] روشی به نام روش انتقال علت پیشنهاد کردند که برای شناسایی مکان و زمانی است که در آن علت شکست از متغیری به متغیر دیگر تغییر می‌کند. این کار را با مقایسه‌ی حالت‌های برنامه در اجراهای موفق و ناموفق و به‌دست‌آوردن اختلاف این حالت‌ها انجام می‌دهد.

برخی روش‌ها برای مکان‌یابی خطا در تلاش برای یادگیری و استنباط مکان خطا براساس داده‌های ورودی و روش‌های یادگیری ماشین می‌باشند. ونگ و همکاران [۲۱]، روش مکان‌یابی خطا براساس انتشار رو به عقب[5] (BP) شبکه های عصبی پیشنهاد کردند که در عمل یکی از محبوب‌ترین مدل‌های شبکه‌های عصبی است. بریاند[6] و همکاران [۲۲]، با استفاده از الگوریتم درخت تصمیم، روشی ارائه دادند که مجموعه‌ای از قوانین می‌سازد که می‌توانند موارد آزمون را به قسمت‌های مختلف طبقه‌بندی کنند؛ به‌طوری‌که موارد آزمون شکست‌خورده در یک ناحیه به احتمال زیاد به‌علتی مشابه شکست خورده‌اند.

سِلیر[7] و همکاران [۲۳] از ترکیبی از قوانین اتحاد و تحلیل مفهوم فرمال[8]، برای کمک به مکان‌یابی خطا استفاده می‌کنند. در [۲۷] از تکنیک‌های داده‌کاوی و تحلیل N-گرام‌ها برای رتبه بندی مظنون به خطا بودن دستورات اجرایی استفاده شده است که در آن مجموعه N-گرام‌ها و نرخ تکرارشان، توسط کاوش قوانین انجمنی تحلیل می‌شود. روش مطرح‌شده در [۲٤] مبتنی بر مدل است و روش‌های مطرح‌شده در [۲۵] و [۲٦] مبتنی بر

1. Liblit
2. SOBER
3. Zeller
4. Cleve
5. Back Propagation
6. Briand
7. Cellier
8. Formal Concept Analysis(FCA)

ضریب تشابه (مانند آچیایی و جَکارد[1]) هستند. همچنین مطالعاتی نیز در تأثیر آزمایش تصادفی موفق در اثربخشی راهکارهای مکان‌یابی خطا انجام گرفته است [۲۸].

٤. شرح راهکار پیشنهادی

معماری راهکاری که در این مقاله ارائه شده است، در شکل (۲) نشان داده شده است. در این راهکار، ابتدا برنامه (P) مستندگذاری می‌شود. پس از مستندگذاری با اجرای موارد آزمون (T) بر روی برنامه‌ی مستندشده ($\widehat{\boldsymbol{P}}$)، اجراها به‌صورت دنباله‌هایی از نام تعیین‌کننده‌ها (S) به‌دست خواهند آمد. با انتخاب یک اجرای غلط به‌عنوان هدف و سپس به‌دست‌آوردن شباهت دیگر اجراها به آن، تعدادی از مشابه‌ترین اجراها، شامل اجراهای صحیح و غلط، انتخاب شده و به‌عنوان یک دسته نگهداری می‌شوند (C). با تحلیل هر دسته، یک سری مسیر اجرایی (R_C) به‌عنوان مسیر مظنون به خطا به‌دست می‌آید. پس از آن، با اجرای رأی اکثریت برای همه‌ی زیرمسیرها (N–گرام ها)ی موجود در مسیرهای اجرایی، زیرمسیرها اولویت‌بندی نهایی شده و ارائه می‌شوند (P_i).

برای مستندگذاری، در قسمت‌هایی از برنامه که به‌عنوان تعیین‌کننده مشخص شده‌اند، دستوری اضافه می‌شود که به منظور ذخیره مقداری درون فایل که نشان‌دهنده‌ی نام تعیین کننده است، به‌کار می‌رود؛ بنابراین، تمام تعیین‌کننده‌ها درون برنامه باید مشخص شده و نامی به آن‌ها اختصاص داده شود. دانه‌بندی مستندگذاری می‌تواند در سطوح مختلفی انجام شود؛ برای مثال، به‌منظور مستندکردن برنامه در سطح تصمیم برای شرطی همچون *if(a>1 && b>a)* تنها دو تعیین‌کننده، یکی برای درست‌بودن و یکی برای نادرست‌بودن شرط به برنامه اضافه می‌شود؛ ولی برای مستندکردن در سطح شرط، به پنج تعیین‌کننده نیاز است: دوتا برای شرط *a>1* دوتا برای *b>a* و یکی هم برای غلط‌بودن کل شرط. سطوح دیگری نیز وجود دارد که پرداختن به آن‌ها در این مقاله نمی‌گنجد. در این راهکار، از دانه‌بندی در سطح شرط برای بالابردن دقت ارائه‌ی مکان مظنون به خطا استفاده شده است.

پس از مستندگذاری و اجرای موارد آزمون بر روی برنامه‌ی مستندشده، دنباله‌هایی از تعیین‌کننده‌ها را خواهیم داشت که نشان‌دهندی اجراها هستند. در این مرحله، مدل‌های N–گرام اجراها، با استفاده از احتمال MLE محاسبه شده و شباهت این مدل‌ها با استفاده از آنتروپی متقاطع به‌صورت رابطه (۱۹) که ترکیبی از رابطه‌های (٤) و (۱۱) است، به‌دست می‌آید (در [۲۹] ایده‌ی اصلی آنتروپی برای مدل‌های N–گرام به‌کار گرفته شده است):

$$H(X) = -\sum_{W^*} p(w_i^n)\, log\, p(w_n|w_i^{n-1}) \qquad (۱۹)$$

که در آن W^* به‌معنی تمام N–گرام‌های رشته X است.

این مقدار نشان‌دهنده‌ی این است که چه اندازه یک دنباله از کلمه‌ها، توسط مدل N–گرام متناظر با آن نشان داده شده است. اگر این رابطه به دو دنباله اعمال شود، اختلاف آن‌ها نمی‌تواند برای اندازه‌گیری شباهت استفاده شود. کمبود رابطه‌ی فوق با استفاده از آنتروپی متقاطع معادل، برطرف شده و می‌تواند برای تعیین شباهت دو دنباله از کلمات استفاده شود که در رابطه (۲۰) نشان داده شده است.

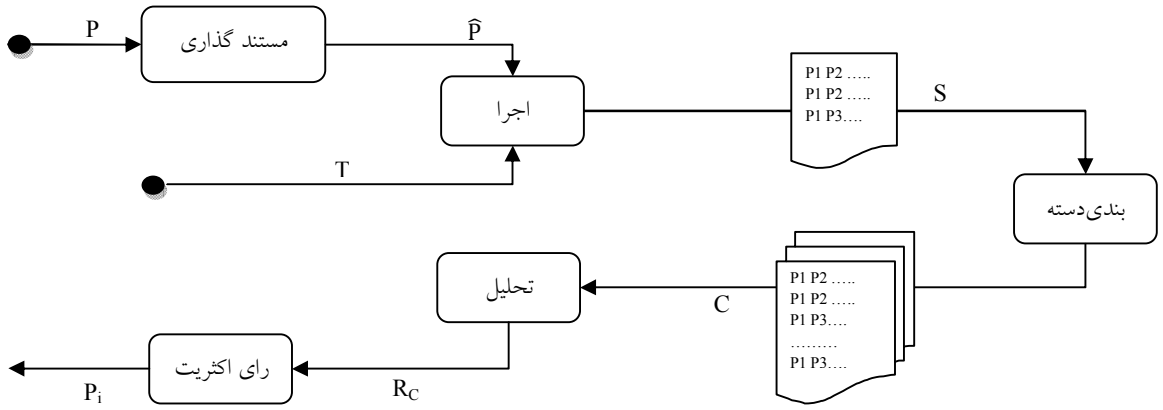
$$H(X,Y) = -\sum_{W^*} P_X(w_i^n)\, log\, P_Y(w_n|w_i^{n-1}) \qquad (۲۰)$$

که W^* در آن به‌معنی تمام N–گرام‌های رشته هدف است.

در این حالت، مدل N–گرام براساس شمارش تعیین‌کننده‌های دنباله‌ی هدف، ساخته شده و تناسبش با دنباله‌ی ثانی توسط رابطه (۲۰) محاسبه می‌شود.

شکل (۳) الگوریتم محاسبه آنتروپی متقاطع برای دو دنباله تعیین‌کننده (مسیر اجرایی) را با استفاده از رابطه (۲۰) نشان می‌دهد.

1. Ochiai & Jaccard



شکل (۲): معماری راهکار پیشنهادی

```
CrossEntropy
Input:
Sequence X, Sequence Y
Output:
e: cross entropy of X and Y
begin
for (int n=1; n<5; n++)
begin
for each (ngram in X.ngrams(n)) //reads each ngram with length n within X
begin
(token, history) = split (ngram);//splits ngram so that ngram = history + token
a = X.P(ngram);//probability ofngram in X sequence
b = Y.ConditionalP(token,history);//probability of tokengiven history in Y sequence
logb = (b!=0 ? Log(b) : Log(1/X.ngramCount));
c = a × logb;
e += c;
end
end
end
```

شکل (۳): الگوریتم محاسبه آنتروپی متقاطع دو مسیر

با دردست‌داشتن اجراها به‌صورت دنباله‌ای از کلمات و استفاده از رابطه‌ی (۲۰) می‌توان شباهت اجراها را به‌منظور دسته بندی به‌دست آورد. جدول (۱) مقادیر به‌دست‌آمده از رابطه‌ی (۲۰) را برای یک دنباله هدف مفروض (که می‌تواند حاصل اجرای یک مورد آزمون بر روی یک نمونه برنامه مستندگذاری شده باشد) و چند دنباله ثانی مفروض نشان می‌دهد.

جدول (۱): آنتروپی متقاطع برای یک رشته هدف با چند رشته مفروض

#	دنباله هدف	آنتروپی
1	P1 P2 P3 P4 P5	1.60
#	دنباله‌های دوم	آنتروپی متقاطع
2	P1 P2 P3 P4	2.28
3	P1 P2 P3 P4 P6	2.46
4	P1 P2 P3 P4 P5 P6	1.79
5	P1 P2 P1 P2 P3 P1 P2 P4	3.65
6	P1 P2 P3 P4 P6 P7 P8 P9	2.84

همان‌طور که مشاهده می‌شود، بیشترین شباهت به اجرای ۱ مربوط به اجرای شماره ٤ است؛ زیرا اجرای شماره ٤ تمام اجرای شماره ۱ را دربر دارد. اجرای شماره ۲ پس از آن شبیه ترین است؛ زیرا زیرمجموعه‌ای از اجرای هدف است. اجرای شماره ۵ و ٦ نشان می‌دهند که طولانی‌تربودن یک زیرمسیر مشترک (P1 P2 P3 P4)، تأثیر بیشتری از متعددبودن یک زیرمسیر مشترک (P1 P2) دارد. از مقایسه‌ی اجرای شماره ۳ و شماره ۲ نیز استنباط می‌شود که طولانی‌بودن اجرا بدون اشتراک با اجرای هدف، نامطلوب است.

پس از به‌دست‌آوردن شباهت اجراها و دسته‌بندی که با انتخاب یک اجرای غلط به‌عنوان هدف و سپس به‌دست‌آوردن شباهت دیگر اجراها به‌عنوان دنباله ثانی به اجرای هدف صورت می‌گیرد، نوبت به تحلیل دسته‌ها می‌رسد. تحلیل به این صورت انجام می‌شود که تمام N-گرام‌های اجرای هدف به‌دست می‌آید و با هرکدام از آن‌ها به‌صورت رشته‌ای جدا رفتار می‌شود و مجدداً توسط رابطه (۲۰) شباهت این زیرمسیر به‌عنوان هدف با اجراهای صحیح و غلط به‌صورت جداگانه به‌دست می‌آید. با تحلیل هر دسته یک‌سری مسیر اجرایی به‌عنوان مسیر مظنون به خطا به‌دست می‌آید.

هرچه مقدار N بیشتر باشد، دقت بالاتر می‌رود؛ ولی با انجام آزمایش‌هایی مشخص شد که مدت‌زمان اجرای برنامه نیز به‌صورت نمایی بالا می‌رود. برای دستیابی به نتیجه‌ای مناسب در زمانی معقول، روش پیشنهادی با Nهای متفاوتی بررسی شد و درنهایت، در این روش برای دسته‌بندی N=4 و برای تحلیل N=3 انتخاب شد. هر زیرمسیری که کمترین شباهت را به اجراهای صحیح و بیشترین شباهت را به اجراهای غلط داشته باشد، به‌عنوان زیرمسیر مشکوک به خطا معرفی می‌شود. برای به دست‌آوردن این امتیاز، مقدار شباهت زیرمسیر به اجراهای صحیح را بر مقدار شباهت آن به اجراهای غلط تقسیم می‌کنیم؛ ولی از آنجایی‌که خروجی رابطه (۲۰) مقدار تفاوت دو رشته است، یک تبدیل انجام می‌دهیم:

$$Point(ngram) = \frac{D_f}{D_p} = \frac{S_p}{S_f} \quad (۲۱)$$

که در آن D_f (S_f) مقدار تفاوت از (شباهت به) اجراهای غلط و D_p (S_p) مقدار تفاوت از (شباهت به) اجراهای صحیح است.

زیرمسیرهایی که هر دسته به‌عنوان زیرمسیر مشکوک به خطا معرفی می‌کند، برحسب این امتیاز اولویت‌بندی می‌شوند. ازآنجایی‌که دسته‌های مختلف که حاصل تحلیل مسیرهای اجرایی مختلف هستند، زیرمسیرها را به‌صورت متفاوتی اولویت بندی می‌کنند و در بعضی موارد زیرمسیرهای متفاوتی ارائه می‌دهند، باید نتیجه‌ی دسته‌ها را با هم ادغام کرد و نتیجه‌ای نهایی استنتاج کرد. برحسب ترتیب ظاهرشدن هر زیرمسیر در هر دسته، مقداری بین ۰ و ۱ به آن زیرمسیر در آن دسته نسبت داده می‌شود که نزدیک‌بودن به ۱ به‌معنای اولویت بالاتر است. این مقدار از تقسیم شماره یا اندیس مکان ظاهرشدن زیرمسیر نسبت به تعداد کل زیرمسیرها به‌دست می‌آید که در رابطه‌ی (۲۲) نشان داده شده است.

$$Priority(Ngram) = \frac{i_{N\text{gram}}}{W} \quad (۲۲)$$

که در آن i_{Ngram} اندیس مکان ظاهرشدن N-گرام در دسته و W تعداد کل N-گرام‌های آن دسته است.

پس از آن، با به‌دست‌آوردن رأی اکثریت بین دسته‌ها توسط رابطه (۲۳)، از مجموع اولویت‌های به‌دست‌آمده از رابطه (۲۲) در هر دسته، برای همه‌ی N-گرام‌های موجود، زیرمسیرها اولویت‌بندی نهایی شده و ارائه می‌شوند.

$$score(Ngram) = \sum_{c=1}^{\text{total}} (Priority(Ngram))_C \quad (۲۳)$$

که در آن total تعداد کل دسته‌ها، c به‌عنوان شماره‌ی دسته و $(Priority(Ngram))_C$ امتیاز N-گرام در آن دسته است.

درنهایت، زیرمسیرها در دسته‌های ۱-گرامی، ۲-گرامی و ۳-گرامی تقسیم می‌شوند و ازآنجایی‌که زیرمسیرهای ۳-گرامی دقت بیشتری دارند، درنتیجه نهایی زیرمسیرهای ۳-گرامی لیست می‌شوند. برای هر زیرمسیر ۳-گرامی، زیرمسیر ۱-گرامی و ۲-گرامی از آن که اولویت‌های بالاتری دارند، قبل از آن نمایش داده می‌شوند که درنهایت لیستی از زیرمسیرهای مشکوک به خطا خواهیم داشت. این زیرمسیرها و تعیین‌کننده‌هایشان به ترتیب اولویت به برنامه‌نویس ارائه می‌شود تا برای مکان‌یابی خطا استفاده شود.

در پیوست الف، برنامه نمونه بسیار ساده‌ای به‌منظور درک عملکرد روش پیشنهادی با استفاده از این روش جهت مکان‌یابی خطا استفاده شده است.

۵. ارزیابی نتایج

در این بخش، برای نشان‌دادن کارایی روش از برنامه‌های محک زیمنس استفاده شده است. مجموعه زیمنس، ۷ برنامه دربر دارد که به زبان C نوشته شده‌اند. روش‌های مکان‌یابی خطا از این مجموعه به‌عنوان معیار سنجش و مقایسه با دیگر روش‌ها استفاده می‌کنند. هرکدام از این برنامه‌ها دارای تعدادی نسخه خطادار هستند که مجموعاً شامل ۱۳۲ نسخه خطادار می‌شود.

پس از مستندگذاری این برنامه‌ها و اعمال راهکار پیشنهادی بر روی آن‌ها، نتایج به‌دست‌آمده به‌صورت خلاصه در جدول (۲) نشان داده شده است. نتیجه در قالب تعداد برنامه‌های زیمنس و درصد خطوطی از برنامه که برنامه‌نویس برای کشف خطا نیاز دارد آنها را بررسی نماید، با شروع از مکانی که راهکار پیشنهادی ارائه داده است، ارزیابی‌شده است.

جدول (۲): خلاصه نتایج روش پیشنهادی بر مجموعه زیمنس

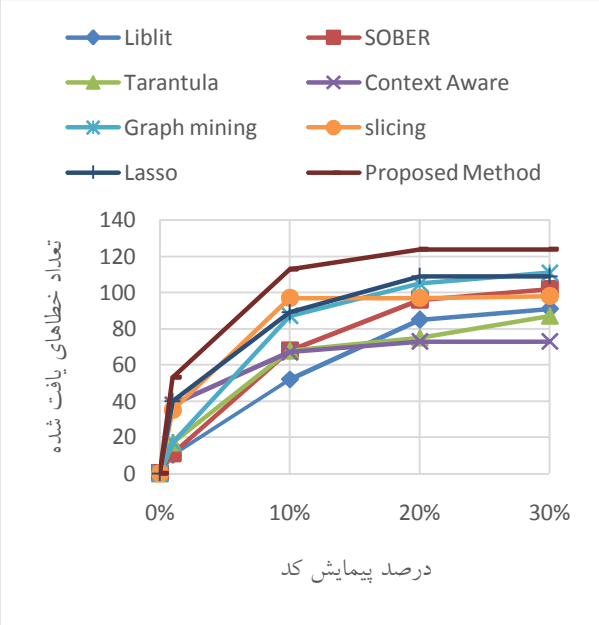
تعداد کل برنامه ها	Teas	Totinfo	Replace	Schedule2	Schedule	Prittokens2	Printtokens	درصد پیمایش کد
۵۳	۳	۱۲	۱۸	۳	۲	۱۰	۵	< 1 %
۸۳	۱۵	۱۸	۲٤	٦	۵	۱۰	۵	< 5 %
۱۱۳	٤۰	۲۱	۲٤	۷	٦	۱۰	۵	< 10 %
۱۲٤	٤۰	۲۳	۲۸	۹	۹	۱۰	۵	< 20 %
٦	۱	۰	٤	۱	۰	۰	۰	بدون اجرای خطا
۲	۰	۰	۰	۰	۰	۰	۲	خطا در سرآیند
۱۳۲	٤۱	۲۳	۳۲	۱۰	۹	۱۰	۷	هاتمام نسخه

جدول (۳): مقایسه تعداد خطاهای کشف‌شده توسط روش پیشنهادی و روش‌های دیگر برای مجموعه زیمنس

آگاه از متن	تارانتولا	لیبلیت	برش بندی	سوبر	رگرسیون لاسو	پیمایش گراف	روش پیشنهادی	درصد پیمایش کد
۳۸	۱۷	۱۰	۳۵	۱۱	٤۰	۱۷	۵۳	< 1 %
٦۷	٦۸	۵۲	۹۷	٦۸	۸۹	۸۷	۱۱۳	< 10 %
۷۳	۷۵	۸۵	۹۷	۹٦	۱۰۹	۱۰۵	۱۲٤	< 20 %
۷۳	۸۷	۹۱	۹۸	۱۰۲	۱۰۹	۱۱۱	۱۲٤	< 30 %

برای نشان دادن کارایی راهکار ارائه‌شده، نتایج به‌دست‌آمده از این روش با روش‌های پیمایش گراف [٨]، رگرسیون لاسو [٦]، سوبر [١١]، برش‌بندی [١٦]، لیبلیت [١٠]، تارانتولا [٩] و آگاه از متن [٣٠] در جدول (٣) مقایسه شده است.

همان‌طور که مشخص است، هم تعداد خطاهای یافت‌شده در مجموعه زیمنس (١٢٤ خطا از ١٣٢ خطا) و هم مقدار پیمایش کد برای یافتن این خطاها (٥٣ خطا در کمتر از ١٪ پیمایش کد) به‌طور چشمگیری کاهش یافته است. برای درک قدرت روش پیشنهادی، نتایج فوق به‌صورت نمودار در شکل (٤) نشان داده شده است.



شکل (٤): مقایسه عملکرد روش پیشنهادی با روش‌های دیگر

٦. نتیجه‌گیری

در این مقاله، راهکاری ارائه شد که نشان داد می‌توان از مباحث مورد استفاده در شباهت رشته‌ها، نه‌تنها در مباحث بیوانفورماتیک برای به‌دست‌آوردن شباهت DNAها، بلکه در مباحث مهندسی نرم‌افزار برای مکان‌یابی خطا نیز استفاده کرد. این راهکار برای اولین‌بار با تغییر مورد استفاده آنتروپی متقاطع و ادغام آن با مدل های N-گرام، به‌کارگیری زیرمسیرها به‌عنوان N-گرام در واحد تعیین‌کننده و دسته‌بندی اجراها، توانست مکان‌یابی خطا را با دقت بالایی انجام دهد. به‌کارگیری این روش در مجموعه آزمون زیمنس این ادعا را نشان می‌دهد. قدرت این روش در یافتن تعداد خطاهای بیشتر با دقت بالاتر (یافتن ٨٣ خطا با پیمایش کمتر از ٥٪ کد) است. ضمناً برای نتایج به‌دست‌آمده از حداکثر قدرت روش به‌منظور مقیاس‌پذیربودن استفاده نشده است و در صورتی که مقیاس‌پذیری قابل چشم‌پوشی باشد، نتایج بهتری هم به‌دست خواهد آمد. همچنین این روش قادر به شناسایی خطاهای چندگانه و خطاهای سگمنت[1] بوده است.

پیوست الف:

برای درک عملکرد روش پیشنهادی در این بخش، برنامه بسیار ساده و کوچکی را که حاوی یک خطا است، مطالعه می‌کنیم. این کد میانه سه عدد را حساب می‌کندو در شکل(الف-١) نشان داده شده است.

```
1  Mid() {
2  int x,y,z,m;
3  read("Enter 3 numbers:",x,y,z);
4  m = z;
5  if (y<z){
6  if (x<y)
7  m = y;
8      Else
9  if (x<z)
10 m = y; //*** bug***
11   }else
12 if (x>y)
13 m = y;
14     Else
15 if (x>z)
16 m = x;
17 print("Middle number is:",m);
18 }
```

شکل (الف-١): نمونه کد حاوی خطا برای به‌دست‌آوردن میانه سه عدد

شکل (الف-٢) نیز مستندگذاری به‌کاررفته در راهکار پیشنهادی برای این قطعه کد را نشان می‌دهد. در اینجا از مستندگذاری در سطح شرط استفاده شده است. هر تعیین‌کننده دارای یک نام است و دستورات مستندگذاری به‌منظور درج نام تعیین‌کننده در فایل اضافه شده‌اند. باید دقت شود که دستورات اضافه‌شده به‌هیچ‌وجه باعث تغییر روند اجرایی برنامه نشود.

1. Segmentation fault

```
1  Mid() {
2  int x,y,z,m;
3  read("Enter 3 numbers:",x,y,z);
4  m = z;
5  if (y<z){
6  if (x<y){
7  m = y; printf(myfile,"P0");}
8  Else
9  if (x<z){
10 m = y; //***bug***// printf(myfile,"P1");} else
   printf(myfile,"P2");
11 }else
12 if (x>y){
13 m = y; printf(myfile,"P3");}
14 Else
15 if (x>z){
16 m = x; printf(myfile,"P4");} else
   printf(myfile,"P5");
17 print("Middle number is:",m);
   printf(myfile,"P6");
18 }
```

شکل (الف-۲): نمونه کد مستندگذاری‌شده کد شکل (الف-۱)

برنامه‌ی مستندگذاری‌شده در مرحله‌ی قبل، با استفاده از یک مجموعه آزمون استاندارد به‌عنوان ورودی اجرا می‌شود. خروجی اجرای این برنامه بسته به مورد آزمونی که اجرا می‌شود و درون فایلی در مجموعه آزمون موجود است، درست یا نادرست است؛ به این معنی که نتیجه اجرای برنامه به‌ازای برخی از موارد آزمون موجود در مجموعه آزمون، به‌دلیل وجود خطای پنهان در برنامه نادرست خواهد بود. اجرای برنامه به‌ازای این دسته از موارد آزمون متوقف نخواهد شد؛ بلکه برنامه به‌طور کامل اجرا می‌شود؛ اما نتیجه‌ی نهایی اجرای آن منطبق با آنچه انتظار می‌رود نیست. این فایلِ نتایج از قبل طی فرآیند آزمون پذیرش[1] به‌دست آمده و در مجموعه آزمون قرار داده شده است؛ برای مثال، جدول (الف-۱) چند مورد آزمون از قطعه کد شکل (الف-۱) را نشان می‌دهد که شامل ورودی‌ها به‌همراه نتیجه آن‌ها می‌باشد که طی آزمون پذیرش به‌دست آمده است.

جدول (الف-۱): نمونه مجموعه آزمون

شماره آزمون	(x,y,z)	نتیجه
t1	(3,3,5)	T
t2	(1,2,3)	T
t3	(3,2,1)	T
t4	(5,5,5)	T
t5	(5,3,4)	T
t6	(2,1,3)	F

1. Acceptance test

با استفاده از مستندگذاری برنامه مطابق با آنچه در بخش پیش گفته شد، شماره هر تعیین‌کننده که در زمان اجرا مشاهده شود، در یک فایل خروجی نوشته می‌شود؛ بنابراین، فایل خروجی هر برنامه پس از اجرای آن برای مجموعه داده، شامل دنباله‌ای از شماره تعیین‌کننده‌هایی است که دستور مربوط به آن تعیین‌کننده اجرا شده است. درواقع این فایل شامل مسیر اجرایی همه موارد آزمون است که برای یافتن مسیر خطادار برنامه مطابق آنچه در بخش اصلی مقاله گفته شد، تحلیل می‌شود؛ برای مثال، فایل به‌دست‌آمده از اجرای موارد آزمون جدول (الف-۱) بر روی قطعه کد مستندشده شکل (الف-۲) به‌صورت جدول (الف-۲) خواهد شد.

جدول (الف-۲): مسیرهای اجرایی حاصل از موارد آزمون

شماره آزمون	مسیر اجرایی
t1	P1 P6
t2	P0 P6
t3	P3 P6
t4	P5 P6
t5	P4 P6
t6	P1 P6

مشاهده می‌شود که به دلیل مستندگذاری تعیین کننده‌ها، مسیرهای اجرایی بسیار بهینه‌تر هستند از زمانی دستورات مستند گذاری می‌شوند. در اینجا *P6* نشان دهنده اجرای دستورات خطوط 1,2,3,4,5,17 و *P1* نشان‌دهنده اجرای دستورات خطوط 6,8,9,10 می‌باشد که در یک گروه اجرایی قرار دارند.

قبلاً نشان داده شد که با در دست داشتن اجراها به‌صورت دنباله‌ای از کلمات و استفاده از رابطه (۲۰) می‌توان شباهت اجراها را به‌منظور دسته‌بندی به‌دست آورد؛ برای مثال، با در دست داشتن دنباله‌های اجرایی جدول (الف-۲)، اجرای شماره ٦ را به‌عنوان اجرای خطادار انتخاب کرده و شباهت دیگر اجراها را به آن به‌دست می‌آوریم.

$$H(t_6,t_1) = -\sum_{w_i^{i}\{P1,P2\},w_i^{i+1}\{P1P2\}} P_{t_6}(w_i^n)\, log\, P_{t_1}(w_n|w_i^{n-1}) =$$

$$-\big(P_{t_6}(w_1)\, log\, P_{t_1}(w_1|\#) + P_{t_6}(w_2)\, log\, P_{t_1}(w_2|\#) + P_{t_6}(w_1^2)\, log\, P_{t_1}(w_2|w_1^1)\big)$$

$$= -\big(P_{t_6}(P1)\, log\, P_{t_1}(P1|\#) + P_{t_6}(P2)\, log\, P_{t_1}(P2|\#) + P_{t_6}(P1P2)\, log\, P_{t_1}(P2|P1)\big)$$

$$= -\big(P_{t_6}(P1)\, log\, P_{t_1}(P1|\#) + P_{t_6}(P2)\, log\, P_{t_1}(P2|\#) + P_{t_6}(P2)P_{t_6}(P1|P2)\, log\, P_{t_1}(P2|P1)\big)$$

$$= -(\,0.5 \times -0.69 + 0.5 \times -0.69 + 0.5 \times 1 \times 0) = -(\,-0.69\,) = 0.69$$

$$H(t_6,t_2) = H(t_6,t_3) = H(t_6,t_4) = H(t_6,t_5) = 1.24$$

حال باید مقادیر را نرمال کنیم؛ به‌طوری‌که شباهت کامل یا به‌عبارتی عدم تفاوت، مقدار صفر را داشته باشد؛ بنابراین، باید کمترین مقدار به‌دست‌آمده از تفاوت مابین دنباله‌ی هدف با دیگر دنباله‌ها، از همه‌ی مقادیر کم شود و از آنجایی‌که کمترین تفاوت بین دنباله هدف با خودش است، مقدار آنتروپی متقاطع دنباله هدف با خودش یا به‌عبارتی آنتروپی دنباله هدف را به‌دست آورده و از همه مقادیر کم می‌کنیم.

$$H(t_6, t_6) = H(t_6) = 0.69$$

بنابراین خواهیم داشت:

$D(t_6,t_1) = 0$,
$D(t_6,t_2) = D(t_6,t_3) = D(t_6,t_4) = D(t_6,t_5) = 1.24 - 0.69 = 0.55$

مشاهده می‌شود که تفاوت اجرای $t1$ و $t6$ صفر است و بدان معناست که این دو اجرا کاملاً شبیه هم هستند و قاعدتاً در یک خوشه قرار خواهند گرفت. این مثال به‌دلیل کوچک‌بودن مسیرهای اجرایی، برای نشان‌دادنِ تأثیر روش، مثال خوبی نیست؛ ولی به‌دلیل حجم زیاد عملیات و محاسبات، به‌کارگیری مثال بزرگ‌تر، از حوصله‌ی این مقاله خارج است. این روش در رشته‌های طولانی‌تر که برنامه‌های واقعی چنین خواهند بود، نتایج بهتری خواهد داد.

با داشتن اجرای t_6 به‌عنوان اجرای غلط و مقایسه زیرمسیرهای آن با دیگر اجراها می‌توان مقدار مظنون‌بودن به خطا را در آن‌ها نشان داد.

جدول (الف-۳): آنتروپی متقاطع زیرمسیرهای اجرای هدف با دیگر اجراها

	تفاوت از اجراهای							
	صحیح					غلط		
زیر مسیر	t1	t2	t3	t4	t5	Dp	t6	Df
P1	0.69	1.09	1.09	1.09	1.09	5.05	0.69	0.69
P6	0.69	0.69	0.69	0.69	0.69	3.45	0.69	0.69
P1 P6	0.69	0.89	0.89	0.89	0.89	4.25	0.69	0.69

ازآن‌جایی‌که اجراهای t_1 تا t_5 همگی بدون شکست بوده اند، اعداد به‌دست‌آمده در ستون D_p نشان‌دهنده‌ی فاصله یا عدم شباهت زیرمسیرها با اجراهای صحیح است. مقدار D_f برای همه‌ی زیرمسیرهای اجرای هدف، به‌دلیل تنها حضور در اجرای هدف معادل آنتروپی آن‌ها با مسیر اصلی‌شان است که به‌دلیل نداشتن مسیر تکراری در این نمونه همگی یکسان هستند. شباهت زیرمسیرها با اجرای خودشان برای صفرنشدن D_f زمانی که تنها اجرای غلط در دسته، اجرای هدف است، ضروری است. برای کاهش سربار محاسبات، می‌توان این عدد را مقداری پیش‌فرض و برای تأثیرگذاری کمتر در تقسیم مقدار ۱ درنظر گرفت. به‌خصوص زمانی که اجرای هدف زیرمسیرهای تکراری ندارد، این کار شایسته‌تر است.

حال با داشتن D_f و D_p می‌توان درمورد مظنون به خطا بودن زیرمسیرها تصمیم گرفت. این مقادیر در جدول (الف-٤) اندازه‌گیری شده است.

جدول (الف-٤): مقدار مظنون‌بودن زیرمسیرهای هدف

زیرمسیر	مقدار مظنون‌بودن $\frac{D_p}{D_f}$	D_f	D_p
P1	۷.۳۱	۰.٦۹	۵.۰۵
P6	۵	۰.٦۹	۳.٤۵
P1 P6	٦.۱۵	۰.٦۹	٤.۲۵

همان‌طورکه مشاهده می‌شود، P1 (که حاوی خطاست) مشکوک‌ترین زیرمسیر به خطا می‌باشد. تقسیم انجام‌گرفته در این مرحله، نقش بسزایی در تعیین زیرمسیر صحیح دارد؛ ولی در این مثال، به‌علت کوچک‌بودن، تأثیر آن پنهان مانده است. همچنین به‌دلیل کوچک‌بودن مسئله تنها یک دسته برای این مثال درنظر گرفته شده و همین امر موجب حذف رأی‌گیری اکثریت نیز شد.

مراجع

[1] A. Zeller, "*Why Program Fail: A Guide to Systematic Debugging*", 1st ed. San Francisco: Morgan Kaufmann, 2006.

[2] M. Dowson, "*The Ariane 5 software failure*", Software Engineering Notes, vol. 22, no.2, p. 84, Mar. 1997.

[3] N. G. Leveson and C. S. Turner, "*An investigation of the Therac-25 accidents*", IEEE Computer Society, vol. 26, no. 7, pp. 18-41, Jul. 1993.

[4] B. Liblit, A. Aiken, X. Zheng, and M. I. Jordan, "*Bug isolation via remote program sampling*", in Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and, San Diego, 2003, pp. 141-154.

[5] W. Dickinson, D. Leon, and A. Podgurski, "*Finding Failures by Cluster Analysis of Execution*", in 23rd Internationall Conference Software Eng. (ICSE 01), 2001, pp. 339-348.

[6] Parsa, S., Asadi-aghbolaghi, M., Vahidi-Asl, M., "*Statistical Debugging Using a Hierarchical Model of Correlated Predicates*", in proc. of the AICI Third International Conference on Artificial Intelligence and Computational Intelligence, Taiyuan, China, pp. 251-256 , 2011.

[7] X. Yan, H. Cheng, J. Han, and P. S. Yu, "*Mining significant graph patterns by leap search*", in International Conference on Management of Data - SIGMOD, Vancouver, BC, Canada, 2008, pp. 433-444.

[8] Z. Mousavian, M. Vahidi-Asl, S. Parsa, "*Finding software fault relevant subgraphs a new graph mining approach for software debugging*", in 24th Canadian Conference on Electrical and Computer Engineering, 2011, pp. 908-911.

[9] J. A. Jones and M. J. Harrold, "*Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique*", in Proc. of the 20th IEEE/ACM international Conference on Automated software engineering, California, USA, 2005, pp. 273-282.

[10] B. Liblit, A. Aiken, M. Naik, and A. X. Zheng, "*Scalable Statistical Bug Isolation*", in Proceeding of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, Illinois, USA, 2005, pp. 15-26.

[11] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "*Statistical Debugging: A Hypothesis Testing-Based Approach*", IEEE Transaction on Software Engineering, vol. 32, no. 10, pp. 831-848, Oct. 2006.

[12] D. Jurafsky and J. H. Martin, "*SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*", 2nd ed. Prentice Hall, 2007.

[13] C. D. Manning and H. Schiitze, "*Foundations of statistical natural language processing*", MIT press. 1999, ch. 2.2, pp. 60-68.

[14] S. Kullback and R. A. Leibler, "*On Information and Sufficiency*", Annals of Mathematical Statistics, vol. 22, no. 1, p. 79–86, 1951.

[15] M. Weiser, "*Programmers use slices when debugging*", Communications of the ACM, vol. 25, no. 7, pp. 446-452, Jul. 1982.

[16] S. Parsa, F. Zareie, M. Vahidi-Asl, "*Fuzzy Clustering the Backward Dynamic Slices of Programs to Identify the Origins of Failure*", in 10th international conference on Experimental algorithms, 2011, pp 352-363.

[17] X. Zhang, H. He, N. Gupta, and R. Gupta, "*Experimental Evaluation of Using Dynamic Slices for Fault Location*", in Proc. of the 6th International Symposium on Automated Analysis-driven Debugging, Monterey, California, USA, 2005, pp. 33-42.

[18] S. Lal, A. Sureka,"*A static technique for fault localization using character n-gram based information retrieval model*", In Proc. of the 5th India Software Engineering Conference (ISEC '12), New York, NY, USA, 2012, pp. 109-118.

[19] A. Zeller and R. Hildebrandt, "*Simplifying and Isolating Failure-Inducing Input*", IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 183-200, Feb. 2002.

[20] H. Cleve and A. Zeller, "*Locating Causes of Program Failures*", in Proc. of the 27th International Conference on Software Engineering, Louis, Missouri, USA, 2005, pp. 342-351.

[21] W. E. Wong and Y. Qi, "*BP Neural Network-based Effective Fault Localization*", International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 4, pp. 573-597, Jun. 2009.

[22] L. C. Briand, Y. Labiche, and X. Liu, "*Using Machine Learning to Support Debugging with Tarantula*", in Proc. of the 18th IEEE International Symposium on Software Reliability, Trollhattan, Sweden, 2007, pp. 137-146.

[23] P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux, "*Formal Concept Analysis Enhances Fault Localization in Software*", in Proc. of the 4th International Conference on Formal Concept Analysis, Montréal, Canada, 2008, pp. 273-288.

[24] F. Wotawa, M. Stumptner, and W. Mayer, "*Model-based Debugging or How to Diagnose Programs Automatically*", in Proc. of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence, Cairns, Australia, 2002, pp. 746-757.

[25] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "*A Practical Evaluation of Spectrum-based Fault Localization*", Journal of Systems and Software, vol. 82, no. 11, pp. 1780-1792, Nov. 2009.

[26] J. A. Jones, J. F. Bowring, and M. J. Harrold, "*Debugging in Parallel*", in Proc. of the 2007 international symposium on Software testing and analysis, London, United Kingdom, 2007, pp. 16-26.

[27] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "*Software Fault Localization Using N-gram Analysis*", in Proc. of the Third International Conference on Wireless Algorithms, Systems, and Applications , Dallas, TX, USA, 2008, pp. 548-559.

[28] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "*Taming Coincidental Correctness: Refine Code Coverage with Context Pattern to Improve Fault Localization*", in Proc. of the 31st International Conference on Software Engineering, Vancouver, Canada, 2009, pp. 45-55.

[29] D. H. Van Uytsel and D. Van Compernolle, "*Entropy-based Context Selection in Variable-Length N-Gram Language Models*", in IEEE Benelux Signal Proc. Symposium, Leuven, Belgium, 1998, pp. 227-230.

[30] L. Jiang, Z. Su, "*Context-aware statistical debugging: from bug predictors to faulty control flow paths*", in Proc. of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, 2007, pp. 184-193.