کاهش دو معیاره مجموعه آزمون با تحلیل خوشه‌ای الگوهای اجرایی

علیرضا خلیلیان[1] و آرمان مهربخش[2]

[1] کارشناس ارشد نرم‌افزار، دانشگاه آزاد اسلامی واحد دماوند
akhalilian@gmail.com

[2] عضو هیأت علمی دانشگاه آزاد اسلامی واحد لاهیجان
ar_mehr@damavandiau.ac.ir

چکیده

یکی از عملیات مهم در چرخه‌ی حیات یک نرم‌افزار، آزمون رگرسیون است که در مرحله‌ی نگهداری نرم‌افزار به‌دفعات انجام می‌شود. آزمون رگرسیون در هر اجرا باید تعداد انبوهی از موارد آزمون را روی نرم‌افزار اجرا نماید. با گذشت زمان، حجم مجموعه آزمون آن‌قدر بزرگ می‌شود که اجرای همه‌ی آن‌ها غیر عملی می‌گردد. برای حل این مشکل از فنون کاهش مجموعه آزمون استفاده می‌شود. متأسفانه کاهش حجم، منجر به از دست رفتن کارایی مجموعه در کشف خطا می‌گردد. برای برطرف نمودن این مشکل، در این مقاله یک الگوریتم کارا ارائه شده است. این الگوریتم با استفاده از خوشه‌بندی الگوهای اجرایی موارد آزمون، افزونگی را از مجموعه حذف می‌نماید. در جریان نمونه‌گیری از هر خوشه، مورد آزمونی که بیش‌ترین پوشش نیازمندی‌ها را تأمین کند، انتخاب خواهد شد. جهت ارزیابی الگوریتم پیشنهادی، آزمایش‌هایی مشابه مطالعات پیشین روی برنامه‌های محک زیمنس ترتیب یافته است. نتایج آزمایش‌ها نشان می‌دهد که الگوریتم پیشنهادی قادر است ضمن کاهش قابل ملاحظه اندازه‌ی مجموعه‌ها، قدرت کشف خطای آن‌ها را بهبود دهد.

کلمات کلیدی

آزمون رگرسیون نرم‌افزار، معیار آزمون، کاهش مجموعه آزمون، کمینه‌سازی مجموعه آزمون، کارایی در کشف خطا.

۱- مقدمه

آزمون رگرسیون نرم‌افزار یکی از فعالیت‌های مهم مرحله‌ی نگهداری نرم‌افزار است که با فرکانس زیاد در چرخه‌ی حیات نرم‌افزار انجام می‌شود [1]. این آزمون پس از هر بار رفع خطاها و تغییر جزیی کد، با اجرای همه‌ی موارد آزمون موجود بررسی می‌کند که تغییرات صحیح بوده و نیز تاثیر نامطلوب بر قسمت‌های بلا تغییر نداشته باشند. علاوه بر این، ممکن است تغییرات مستلزم افزودن عملکردهای جدید به نرم‌افزار باشند. آزمودن عملکردهای جدید نیاز به طراحی موارد آزمون جدید و افزودن آن‌ها به مجموعه آزمون فعلی دارد. در نتیجه این فرایند، حجم مجموعه آزمون به‌تدریج آن‌قدر زیاد می‌شود که اجرای مجدد همه‌ی موارد آزمون عملاً غیر ممکن می‌گردد. بنابراین، مشکل اساسی آزمون رگرسیون نرم‌افزار، حجم انبوه موارد آزمونی است که پس از چند مرحله اجرای آزمون ایجاد می‌شود [2]. اجرای این حجم انبوه آزمون‌ها، به‌دلیل محدودیت زمان و منابع آزمون و نیز فرکانس بالای اجرای آزمون رگرسیون غیر عملی است.. بنابراین به‌منظور رفع مشکل آزمون رگرسیون، فنونی برای کاهش مجموعه آزمون‌ها مطرح شده است تا به‌طور دائمی موارد آزمون افزونه را از مجموعه آزمون حذف نمایند [4][3][2].

فنون کاهش مجموعه آزمون‌ها، ضمن حذف دائمی موارد آزمون افزونه، سعی می‌کنند تا کاراترین موارد آزمون را از نظر پوشش قسمت‌های مختلف کد نرم‌افزار و آشکارسازی خطاها در مجموعه حفظ نمایند. حذف افزونگی، اگرچه منجر به کاهش چشمگیر اندازه‌ی مجموعه آزمون می‌گردد، اما متقابلاً باعث افت شدید کارایی مجموعه‌ی کاهش یافته در کشف خطا می‌گردد [3]. اما باید توجه کرد که هدف اصلی آزمون، آشکارسازی خطاهاست [5]. از این‌رو فنون کاهش بیشتر از آن‌چه کاهش را مورد توجه قرار دهند، باید قدرت آشکارسازی خطاها را هدف قرار داده و بتوانند موازنه‌ی منطقی میان اندازه‌ی مجموعه و قدرت آشکارسازی خطای آن برقرار سازند. عدم توجه به این

مسأله، خلأ بزرگی در اغلب تحقیقات موجود پیرامون کاهش مجموعه آزمون‌ها بوده است. در این مقاله، یک روش کارا برای کاهش مجموعه آزمون‌ها ارائه شده است که سعی می‌کند با ساز و کاری به‌خصوص، موارد آزمون افزونه را شناسایی کرده و از مجموعه حذف کند. ضمن این‌که در فرایند انتخاب برای مجموعه‌ی کاهش‌یافته، موارد آزمونی را در نظر می‌گیرد که حداکثر پوشش کد را داشته باشند تا به‌این ترتیب مؤثرترین آن‌ها را در کشف خطا برگزیند. در ادامه مقاله در بخش ۲، تعاریف، پیشینه تحقیق و برخی روش‌های موجود معرفی می‌گردد. پس از بیان مشکل روش‌های موجود، در بخش ۳ روش پیشنهادی ارائه می‌شود. بخش ۴ به ارزیابی و تحلیل نتایج اختصاص داشته و نتیجه‌گیری در بخش ۵ ارائه می‌شود.

۲- پیشینه تحقیق و کارهای مرتبط

روشن است که هر چه تعداد موارد آزمون در فرایند آزمون بیشتر باشد، احتمال تأمین نیازمندی‌های آزمون بیشتر می‌گردد. نیازمندی آزمون، خصوصیت از برنامه است، که باید در جریان آزمون نرم‌افزار برآورده شود [6]. به‌عنوان مثال، یک انشعاب، تعریف و استفاده‌ی یک متغیر، یک بلاک اولیه یا ورودی یک تابع، نیازمندی آزمون محسوب می‌شود. مشکل این‌جا است موارد آزمون جدید ممکن است تولید و در مجموعه درج شود تا نیازمندی‌های بیشتری را تأمین نماید. در نتیجه موارد آزمون موجود در مجموعه آزمون، ممکن است بیشتر از حد نیاز برای تأمین نیازمندی‌های آزمون باشد. مسأله اینجاست که اگر تعدادی از موارد آزمون از مجموعه آزمون حذف شود، این مجموعه کماکان همه‌ی نیازمندی‌های آزمون را که مجموعه‌ی اولیه تأمین می‌کرد، برآورده می‌نماید. این مسأله به "کاهش مجموعه آزمون" مشهور بوده و زیرمجموعه‌ی حاصل، مجموعه‌ی نماینده یا شاخص نامیده می‌شود. [7][3]. تعریف رسمی مسأله‌ی کمینه‌سازی مجموعه آزمون به‌صورت زیر است [2]:

فرض: یک مجموعه آزمون T از موارد آزمون $\{t_1, t_2,..., t_m\}$، یک مجموعه از نیازمندی‌های آزمون $\{r_1, r_2, ..., r_n\}$، که باید برآورده شوند تا پوشش مطلوب از برنامه حاصل شود، و زیرمجموعه‌های $\{t_1, t_2,..., t_n\}$ از T که هرکدام به یکی از r_i ها منتسب است به‌قسمی که هر یک از موارد آزمون t_j متعلق به T_i ، r_i را تأمین نماید.

مسأله: زیرمجموعه‌ی کمینه‌ای از T پیدا کنید که همه‌ی r_i هایی که مجموعه‌ی غیرکمینه‌ی T تأمین می‌کند را برآورده نماید.

تحقیقات نسبتاً زیادی در ارتباط با کاهش مجموعه آزمون صورت گرفته است. این تحقیقات دو دسته‌ی کلی از فنون را ارائه کرده است: ۱) فنون بهینه که با هزینه اجرایی زیاد، مجموعه‌هایی را تولید می‌نمایند که به‌صورت بهینه‌ای کمینه شده‌اند و ۲) روش‌های ابتکاری که مجموعه‌های تولید شده آن‌ها تقریباً بهینه هستند اما سرعت اجرایی آن‌ها بیشتر بوده و سریع‌تر به جواب می‌رسند. از آن‌جایی‌که مسأله‌ی کمینه‌سازی مجموعه آزمون‌ها، مشابه مسأله‌ی پوشش مجموعه‌ها بوده و از این‌رو NP کامل می‌باشد [8]، اکثر تحقیقات و کارهای انجام شده در این مقوله به دنبال روش‌های ابتکاری کمینه‌سازی بوده‌اند. از مهمترین مطالعات و روش‌های ابتکاری کاهش مجموعه آزمون‌ها می‌توان موارد زیر را نام برد: کاهش مجموعه آزمون دو معیاره مبتنی بر برنامه‌نویسی صحیح خطی [9]، الگوریتم حریصانه اچ‌جی‌اس [2]، مطالعات تجربی با الگوریتم اچ‌جی‌اس [3]، کمینه‌سازی با پوشش حساسیت احتمالی دستورالعمل [10]، کمینه‌سازی مبتنی بر تحلیل مفهومی [11]، کمینه‌سازی با پوشش تصمیم / شرط تغییر یافته [12]، مطالعه برای مقایسه چهار فن معمول کمینه‌سازی [4]، کاهش بر اساس پشته فراخوانی [13] و کاهش بر اساس معیار افزونگی انتخابی [5].

اغلب روش‌هایی فوق مبتنی بر پوشش هستند. اما پوشش کد به‌تنهایی برای انتخاب موارد آزمون مناسب کافی نیست. زیرا اغلب می‌توان با انتخاب چند مورد آزمون ساده به پوشش کاملی همانند مجموعه اصلی دست یافت. اما آزمون‌های منتخب رفتار برنامه را در شرایط واقعی منعکس نخواهند کرد [14]. لذا مشکل فنون مبتنی بر پوشش، درنظر گرفتن پوشش کد به‌تنهایی است که منجر به ایجاد مجموعه‌های کوچکتر می‌شود، اما این مجموعه‌ها کارایی خوبی در کشف خطا نخواهند داشت. دسته‌ای دیگری از روش‌های کاهش، فنون مبتنی بر توزیع هستند [15] که از الگوهای اجرایی موارد آزمون استفاده می‌کنند تا آن‌ها را بر حسب شباهتشان دسته‌بندی نمایند. یک الگوی اجرایی، مؤلفه‌هایی از برنامه را مشخص می‌کند که با اجرای مورد آزمون، پوشیده شده‌اند. علی‌رغم کارایی این دسته از فنون در کشف خرابی‌ها، مشکل اساسی آن‌ها این است که لزوماً پوشش کامل ایجاد نمی‌کنند [15].

۳- روش پیشنهادی

اگرچه تقریباً همه‌ی روش‌های موجود کاهش مجموعه آزمون قادرند به‌خوبی مجموعه آزمون را کاهش دهند [4]، اما مشکل اغلب آن‌ها این است که خطاهایی که در اثر این کاهش ممکن است آشکار نشده باقی بمانند، موجب افت کیفیت نرم‌افزار خواهند شد. لذا فن کاهشی ایده‌آل است که بتواند به‌شکلی هوشمندانه، موارد آزمونی را انتخاب نماید که هم منحصر بفرد بوده و هم قدرت آشکارسازی خطای آن‌ها بالا باشد. چنین انتخابی یک مشکل عمده دارد: آزمونگران قبل از اجرای آزمون از وجود خطاها، محل آن‌ها و شدت آن‌ها بی‌خبرند. پس چگونه می‌توان قدرت آشکارسازی خطای موارد آزمون را به‌دست آورد؟ محققان برای حل این مشکلات سعی می‌کنند تا روش‌های ابتکاری را به‌کار گیرند و در آن‌ها به‌کمک معیارهایی، رفتار واقعی برنامه و نیز

رفتار خطاها را تخمین بزنند [1].

روش پیشنهادی این مقاله ایده‌های دو رویکرد مبتنی بر توزیع و مبتنی بر پوشش را به‌کار می‌گیرد تا ضمن برطرف نمودن ایرادات این دو روش، بتواند از مزایای آن‌ها در تولید یک مجموعه آزمون کاهش‌یافته کارا بهره ببرد. ایده‌ی کلی این روش اینست که ابتدا الگوهای اجرایی حاصل از موارد آزمون را با استفاده از یک روش تحلیل چند متغیری مثل خوشه‌بندی دسته‌بندی نماییم. سپس با استفاده از یک روش حریصانه از فنون مبتنی بر پوشش، آن‌قدر از این دسته‌ها نمونه‌برداری نماییم تا پوشش تجمعی موارد آزمون منتخب همانند پوشش مجموعه اصلی شود. اما نکته‌ی مهمی در این میان وجود دارد. موارد آزمونی که با توجه به یک معیار آزمون افزونه هستند، ممکن است نسبت به سایر معیارهای آزمون افزونه نباشند. مثلاً ممکن است دو مورد آزمون دستورات انشعابی یکسانی را اجرا نمایند که در نتیجه یکی از آن‌ها با توجه به معیار آزمون انشعاب افزونه خواهد بود. اما همین دو مورد آزمون ممکن است در اجرای دستورات تعریف و استفاده متغیرها با هم کمی تفاوت داشته باشند. در این‌صورت این دو مورد آزمون با درنظر گرفتن معیار آزمون تعریف و استفاده دیگر افزونه نبوده و وجود هر دو در مجموعه‌ی کاهش‌یافته ضروری است. ویژگی مهم الگوریتم پیشنهادی این‌است که مجموعه‌ی کاهش‌یافته حاصل نسبت به دو معیار آزمون مورد استفاده، کفایت آزمون را تأمین خواهد کرد. شبه‌کد الگوریتم پیشنهادی در شکل ۱ مشاهده می‌شود.

تحلیل بدترین زمان اجرا: فرض کنید که تعداد موارد آزمون را با nt، تعداد نیازمندی‌های معیار اول را با m، تعداد نیازمندی‌های معیار دوم را با m' نشان دهیم. همچنین فرض کنید که MC بیشترین تعداد موارد آزمونی باشد که هر نیازمندی از معیار اول را می‌پوشاند و نیز MC' بیشترین تعداد موارد آزمونی باشد که هر نیازمندی از معیار دوم را می‌پوشاند. زمان کل الگوریتم پیشنهادی پس از ساده‌سازی نهایی به‌صورت $O(nt.m.MC) + O(nt.m'+MC')$ در می‌آید.

۴- ارزیابی

به‌منظور ارزیابی کارایی روش ارائه شده و مقایسه با روش‌های پیشین، آزمایش‌های تجربی ترتیب داده شده است. در آزمایش‌های انجام گرفته از مجموعه آزمون زیمنس (جدول ۱) شامل ۷ برنامه به زبان C [17]، استفاده شده است. مشابه آزمایش‌هایی که در [5] و [10] انجام شده است، در این تحقیق نیز برای برپایی آزمایش‌ها، مجموعه آزمون‌های کافی در پوشش یال‌ها (انشعاب‌های برنامه) ایجاد شده است. در آزمایشات انجام شده، ۱۰۰۰ مجموعه آزمون کافی در پوشش یال برای هر برنامه در هر یک از شش محدوده فوق (جمعاً ۶۰۰۰ مجموعه آزمون) به این روش تولید شده است. این مجموعه‌ها را B، B1، B2، B3، B4 و B5 می‌نامیم. برای استخراج پوشش انشعاب موارد آزمون، کد برنامه‌های زیمنس به‌طور دستی مستندگذاری گردید. به‌منظور ارزیابی و مقایسه‌ی روش‌های پیشنهادی با روش‌های موجود، پس از کاهش اطلاعات زیر در مورد هر مجموعه آزمون ثبت شده است:

۱- تعداد موارد آزمون موجود در مجموعه آزمون اصلی، $|T|$

۲- تعداد موارد آزمون موجود در مجموعه‌ی کاهش‌یافته، $|T_{red}|$

۳- تعداد خطاهای متمایزی که توسط مجموعه‌ی اصلی قابل کشف است، $|F|$

۴- تعداد خطاهای متمایزی که توسط مجموعه‌ی کاهش‌یافته قابل کشف است، $|F_{red}|$

۵- درصد کاهش اندازه‌ی مجموعه (٪کاهش):

$$\frac{|T|-|T_{red}|}{|T|}*100 \qquad (۱)$$

۶- درصد فقدان کارایی کشف خطا (٪فقدان خطا):

$$\frac{|F|-|F_{red}|}{|F|}*100 \qquad (۲)$$

جدول (۱): مشخصات برنامه‌های زیمنس

نام برنامه	تعداد نسخ خطادار	تعداد موارد آزمون
ptok	۷	۴۱۳۰
ptok2	۱۰	۴۱۱۵
replace	۳۲	۵۵۴۲
sched	۹	۲۶۵۰
sched2	۱۰	۲۷۱۰
tcas	۴۱	۱۶۰۸
totinfo	۲۳	۱۰۵۲

در آزمایش‌های انجام شده، از نمودارهای جعبه‌ای، استفاده شده است. که به کمک ابزار SAS/GRAPH تولید شده است [18]. علاوه بر این، از پیاده‌سازی الگوریتم کِلوپ موجود در ابزار Weka به‌منظور خوشه‌بندی داده‌ها استفاده شده است [19]. به‌منظور تعیین این‌که آیا کاهش فقدان کشف خطا برای الگوریتم‌های پیشنهادی نسبت به الگوریتم مورد مقایسه از لحاظ آماری معنی‌دار است، از یک روش آماری به‌نام *آزمون فرض برای تفاضل دو میانگین* استفاده شده است [20]. با استفاده از فرمول زیر، مقدار z را محاسبه کرده و آنرا در جدول مقادیر بحرانی در مرجع [20] جستجو می‌کنیم. رد فرض صفر دلالت می‌کند بر این‌که کاهش فقدان کشف خطا توسط روش‌های پیشنهادی از لحاظ آماری معنی‌دار است و درصد اطمینان، نشان می‌دهد که چند درصد می‌توان به این کاهش در مورد برنامه‌های واقعی اطمینان داشت.

$$Z=\frac{(\overline{x_1}-\overline{x_2})-\delta}{\sqrt{\frac{\sigma_1^2}{n_1}+\frac{\sigma_2^2}{n_2}}} \qquad (۳)$$

تحقیقات نشان می‌دهند، در بین فنون معمول و کاربردی کاهش مجموعه آزمون، روش اچ‌جی‌اس با در نظر گرفتن معیارهایی چون اندازه‌ی مجموعه‌ی تولیدی، قدرت کشف خطا، سرعت اجرایی، پیچیدگی زمانی، سادگی و سهولت، انتخاب اول در بین سایر فنون موجود خواهد بود [4]. لذا الگوریتم پیشنهادی با الگوریتم اچ‌جی‌اس مقایسه شده است. نتایج حاصل از این آزمایش در نمودار جعبه‌ای شکل ۲ مشاهده می‌شود که توزیع درصد کاهش اندازه‌ی مجموعه آزمون اصلی (SR) و درصد فقدان کشف خطا (FL) را در بزرگ‌ترین محدوده‌ی موارد آزمون،B5، نشان می‌دهد. در این شکل برای هر برنامه ۲ جفت جعبه در نظر گرفته شده است. جفت جعبه‌های سفید، درصد کاهش اندازه‌ی مجموعه‌ی اصلی و جفت جعبه‌های خاکستری درصد فقدان کشف خطا را نشان می‌دهند. در هر جفت، جعبه‌ی سمت چپ، برای الگوریتم پیشنهادی و جعبه‌ی سمت راست برای الگوریتم اچ‌جی‌اس در نظر گرفته شده است. به‌علاوه، در جدول ۲ مقادیر z محاسبه شده برای مجموعه آزمون‌های محدوده‌ی B5 و درصد اطمینان برای رد فرض صفر نمایش داده شده است.

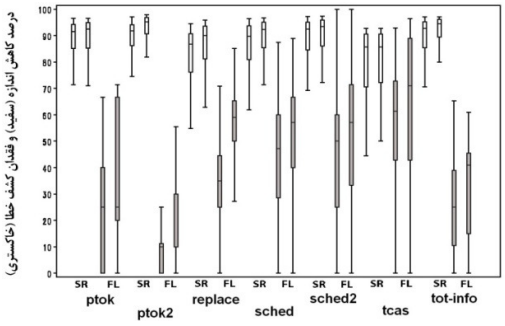
```
define:
   requirement^i: set of coverage requirements for minimization with respect to the ith criterion:
                  r_1^i, r_2^i,..., r_n^i, i =1,2
input:
   t_1,t_2,...,t_m: test cases present in the unreduced test suite
   cv_i[m,n]: coverage matrix representing the ith requirement coverage of each test case
   clusters_i: array[1..k] of cluster instances for the ith requirement, each containing similar test cases
output:
   RS: a reduced suite of test cases from the original test suite.

algorithm TestSuiteReduction
begin
STEP 1:   currentClusterIndex := 0;
          redundant := redundant1 := redundant2 := { }; RS := { };
          foreach criterion i and requirement j do marked_i[j] := false;
          for each test case t and criterion i do
             numUnmarked^i[t] := number of the elements in cv_i[t,n] for which cv_i[t,j] == true;
          foreach t_i do numCovered[i] is sum of the the number of elements for which
                cv_1[t_i, r_j^1 ] == true or cv_2[t_i, r_k^2 ] == true;
          Sort(clusters,ascending);
STEP 2:   while there exists r_i^1 ∈ requirements^1 s.t. marked_1[i] == false do
             if currentClusterIndex == k then currentClusterIndex := 0;
             list := all t_j ∈ clusters_1[currentClusterIndex];
             if Card(list) == 1 then
                test := t ∈ list;
                if exists r_j^1 ∈ requirements^1 s.t. cv_1[test, r_j^1 ] == true and marked_1[j] == false then
nextTest := test;
                else nextTest := 0;
                endif
             else
                nextTest := SelectTest(list, numUnmarked^1, numUnmarked^2, numCovered);
             endif
STEP 3:      if nextTest ≠ 0 then
                RS := RS ∪ {nextTest};
                clusters[currentClusterIndex] := clusters[currentClusterIndex] – nextTest;
                foreach criterion c do
                   foreach r_j^c ∈ requirements^c s.t. cv_c[nextTest, r_j^c ] == true and marked_c[j] == false do
                      marked_c[j] := true;
                      foreach test case t in the test suite s.t. cv_c[t, r_j^c ] == true do
                         numUnmarked^c[t] := numUnmarked^c[t] – 1;
                         if numUnmarked^c[t] == 0 and t ∉ RS then
                            redundant_c := redundant_c ∪ t;
                      endfor
                   endfor
                endfor
             currentClusterIndex := currentClusterIndex + 1;
             redundant := redundant_1 - redundant_2;
STEP 4:      if Card(redundant) != 0 then
                redundant := redundant ∩ list;
                SelectSomeMoreTestCases(redundant, RS);
                redundant := redundant_1 := redundant_2 := { };
             endif
          endwhile
          return RS;
end TestSuiteReduction
```

شکل (۱): شبه کد بدنه اصلی الگوریتم پیشنهادی

کاهش اندازه مجموعه: درصد کاهش اندازه‌ی مجموعه‌های کاهش‌یافته توسط روش پیشنهادی در مورد همه‌ی برنامه‌ها، به‌طور میانگین کمی از درصد کاهش اندازه‌ی مجموعه‌های کاهش‌یافته توسط روش اچ‌جی‌اس کمتر است. علت این مسأله این است که روش پیشنهادی مورد آزمون‌های بیشتر و مؤثرتری را با در نظر گرفتن معیار دوم انتخاب می‌کند.

فقدان کشف خطا: میانگین فقدان کشف خطا در مجموعه‌های کاهش‌یافته توسط روش پیشنهادی دوم کمتر از میانگین فقدان کشف خطا در مجموعه‌های کاهش‌یافته توسط روش اچ‌جی‌اس است. ضمن این‌که مشاهده می‌شود که نتایج کاهش و فقدان کشف خطا در مورد مجموعه آزمون‌های بزرگتر بهتر شده است.



شکل (۲): نمودار جعبه‌ای درصد کاهش اندازه مجموعه آزمون و درصد فقدان کشف خطا برای الگوریتم پیشنهادی

دلیل این مسأله این است که مجموعه آزمون‌های بزرگتر، تعداد بیشتری مورد آزمون افزونه نسبت به معیار انشعاب دارند، و بنابراین الگوریتم پیشنهادی هنگام انتخاب، گزینه‌های بیشتری از موارد آزمون را می‌تواند انتخاب کند. علاوه بر این، جدول ۲ نشان می‌دهد که میزان بهبود در نرخ کشف خطا توسط روش پیشنهادی دوم، برای همه‌ی برنامه‌ها از لحاظ آماری معنی‌دار است و درصد اطمینان این معنی‌داری بالاتر از ۹۹٪ می‌باشد.

جدول (۲): مقادیر z محاسبه شده و درصد اطمینان رد فرض صفر

نام برنامه	مقدار z محاسبه شده	درصد اطمینان رد فرض صفر
ptok	۳/۷۲	>۹۹/۹۵٪
ptok2	۳/۵۰	>۹۹/۹۵٪
replace	۹/۹۶	>۹۹/۹۹٪
sched	۲/۵۰	>۹۸/۷۵٪
sched2	۵/۴۴	>۹۹/۹۹٪
tcas	۵/۲۱	>۹۹/۹۹٪
totinfo	۵/۶۸	>۹۹/۹۹٪

۵- نتیجه گیری

در این مقاله یک الگوریتم کارا برای حل مشکل فنون کاهش مجموعه آزمون ارائه شد. الگوریتم ارائه شده توانسته است موازنه‌ی صحیحی میان اندازه‌ی مجموعه و کارایی آن در کشف خطا برقرار سازد. به منظور ارزیابی کارایی الگوریتم‌های پیشنهادی، آزمایش‌هایی مشابه با مطالعات معتبر این حوزه، ترتیب داده شده است. در این آزمایش‌ها، مجموعه‌های کاهش‌یافته ، همواره نرخ کشف خطای بالاتری نسبت الگوریتم موجود از خود نشان دادند. معنی‌دار بودن میزان بهبود در کشف خطا نیز از لحاظ آماری نیز تأیید شده است.

مراجع

[1] S. Mirarabbaygi, A Bayesian Framework for Software Regression Testing, Master Thesis of Applied Science, Waterloo, Canada, 2008.
[2] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," ACM Transactions on Software Engineering and Methodology, vol. 2, No. 3, pp. 270-285, 1993.
[3] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical Studies of Test-Suite Reduction," Journal of Software Testing, Verification, and Reliability, vol. 12, Iss. 4, pp.219-249, 2002.
[4] H. Zhong, L. Zhang, H. Mei, "An experimental study of four typical test suite reduction techniques," Journal of Information and Software Technology, Vol. 50, No. 6, pp. 534-546, 2008.
[5] D. Jeffrey and N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction," IEEE Trans. Softw. Eng., vol. 33, pp. 108-123, 2007.
[6] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, UK, 2008.
[7] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in the 8th International Symposium on Software Reliability Engineering, pp. 230-238, 1997.
[8] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, NY, Freeman and Company, 1979.
[9] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-Criteria Models for All-Uses Test Suite Reduction," in the 26th International Conference on Software Engineering, pp. 106-115, 2004.
[10] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," in International Conference of Software Maintenance, Bethesda, Maryland, pp. 34-43, 1998.
[11] S. Sprenkle, S. Sampath, E. Gibson, A. Souter, and L. Pollock, "An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications," Technical Report 2005-009, Computer and Information Sciences, University of Delaware, 2004.
[12] J. A. Jones and M. J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," IEEE Transactions on Software Engineering, vol. 29, No. 3, pp. 195-209, 2003.

[13] S. McMaster and A. Memon, "Call Stack Coverage for Test Suite Reduction," in 21st IEEE International Conference of Software Maintenance, pp. 539-548, 2005.
[14] B. Marick, The Craft of Software Testing: Subsystem Testing, Prentice Hall, Englewood Cliffs, NJ, 1995.
[15] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," in 14th International Symposium on Software Reliability Engineering, Denver, Colorado, pp. 17-20, 2003.
[16] D. Leon, A. Podgurski and L. J. White, "Multivariate visualization in observation-based testing," in Proceedings of the 22nd international conference on Software engineering. ACM, Limerick, Ireland, pp. 116-125, 2000.
[17] G. Rothermel, S. Elbaum, A. Kinneer, H. Do. Software-artifact infrastructure repository. http://www.cse.unl.edu/~galileo/sir.
[18] SAS 9.1.3 Documentation, SAS/GRAPH 9.1 Reference, http://sup
port.sas.com/documentation/onlinedoc/91pdf/index_913.
[19] I. H.Witten, E. Frank, Data mining: practical machine learning tools and techniques, 2nd ed., (Morgan Kaufmann series in data management systems) 2005.
[20] J. E. Freund, Mathematical Statistics, 5th ed., Prentice-Hall, 1992.