انجمن کامپیوتر ایران
Computer Society of Iran

مرکز توسعه فناوری نیرو (متن)

بهینه‌سازی و تسریع فرایند کشف خطا در آزمون رگرسیون نرم‌افزار

علیرضا خلیلی‌یان
دانشگاه آزاد اسلامی واحد دماوند
akhalilian@gmail.com

آرمان مهر بخش
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.

واژه های 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. از آن‌جایی که تعدادی از موارد آزمون جدیدی که به مجموعه آزمون اضافه می‌شوند، ممکن است مسیرهای اجرایی را پوشش دهند که توسط سایر موارد آزمون نیز پوشش داده می‌شوند، لذا تعدادی از موارد آزمون موجود در مجموعه افزونه هستند. 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]. بنابراین مسأله مهم در فنون کاهش مجموعه آزمون، افت غیر قابل قبول قابلیت آشکارسازی خطای مجموعه آزمون است. 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. بنابراین، مشکل اساسی که هم‌اکنون با آن مواجه هستیم، یافتن فنونی است که موارد آزمون منتخب آن‌ها، دو ویژگی را به‌طور همزمان برآورده سازند: اولاً مؤثرترین موارد آزمون در کشف خطا باشند و ثانیاً در پوشش مسیرهای اجرایی برنامه، کمترین هم‌پوشانی را داشته باشند که منجر به حذف بیشینه افزونگی خواهد شد. 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، یک مورد مشخص، موجودیت، خصوصیت یا رفتاری از برنامه یا محصول نرم‌افزاری 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 که رشد بی‌رویه‌ی مجموعه آزمون می‌تواند مشکل‌ساز شود. به‌عبارت دیگر، موارد آزمون جدید ممکن است تولید و در مجموعه درج شود تا نیازمندی‌های بیشتری را تأمین نماید. در نتیجه موارد آزمونِ 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" (کمینه‌سازی مجموعه آزمون) مشهور بوده و زیرمجموعه‌ی حاصل، مجموعه‌ی نماینده یا شاخص نامیده می‌شود. اگر هیچ زیرمجموعه‌ای از مجموعه‌ی نماینده، تمام نیازمندی‌ها را تأمین نکند، آنرا مجموعه‌ی بهینه یا مجموعه‌ی کمینه می‌نامیم [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. در این بین، تحقیقاتی که مرتبط با این مقاله هستند به دو دسته‌ی کلی تقسیم می‌شوند: ۱) کمینه‌سازی مجموعه آزمون و ۲) کارایی در کشف خطا. این دو زمینه تحقیقاتی از هم مجزا نیستند چراکه در بعضی موارد مفاهیم کارایی در کشف خطا برای تحقیقات در زمینه کمینه‌سازی مجموعه آزمون‌ها به‌منظور ارزیابی و مقایسه فنون مختلف کمینه‌سازی به‌کار گرفته شده است. تحقیقاتی که در ارتباط با کمینه‌سازی مجموعه آزمون‌ها انجام یافته است دو دسته‌ی کلی از فنون کمینه‌سازی 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 اچ‌جی‌اِس [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. این دسته از فنون سعی می‌کنند موارد آزمون افزونه را از مجموعه حذف کرده و پوشش کد را بیشینه نمایند. به این ترتیب اطمینان حاصل می‌کنند که مسیرهای اجرایی مختلف برنامه اجرا خواهد شد. 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. مثلاً الگوی اجرایی قادر است انشعاب‌ها یا جریان داده‌هایی که با اجرای یک مورد آزمون پوشیده شده است را نشان دهد. این الگوها اطلاعات اضافی را راجع به موارد آزمون منعکس می‌کنند که در فرایند تشکیل مجموعه‌ی کاهش‌یافته مفید خواهند بود. فنون کاهش مبتنی بر توزیع نمونه‌هایی از *آزمون مبتنی بر مشاهده* [16] هستند. علی‌رغم کارایی این دسته از فنون در کشف خرابی‌ها، مشکل اساسی آن‌ها این است که لزوماً پوشش کامل ایجاد نمی‌کنند [15]. علاوه بر این مطالعات تجربی [15] نشان داده است که عملکرد این‌گونه روش‌ها با اندازه‌ی مجموعه

آزمون تغییر می‌کند. نتایج این مطالعات همچنین حاکیست کـه مـوارد آزمون منتخب توسط این روش‌ها و روش‌های مبتنی بـر پوشـش اغلـب شبیه نیستند. از این‌رو می‌توان دریافت که فنـون مبتنـی بـر پوشـش و مبتنی بر توزیع مکمل یکدیگرند.

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. به‌عبارت دیگر مشکل اساسی فنون جاری کاهش مجموعه آزمون این است که حذف دائمی موارد آزمون افزونه، اگرچه منجر به کاهش چشمگیر اندازه‌ی آن می‌شود، اما متقابلاً کارایی مجموعه را در کشف خطا به مخاطره می‌اندازد. 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 دو مشکل عمده دارد: اول این‌که چگونه می‌توان موارد آزمون منحصر بفرد را تشخیص داد یا به‌عبارت بهتر معیار تشخیص منحصر بفرد بودن موارد آزمون چیست. مسأله دوم این است که 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. به‌این ترتیب موارد آزمونی که از لحاظ پوشش مؤلفه‌های برنامه یکسان یا شبیه هستند، در دسته‌های یکسانی قرار می‌گیرند. 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. مجموعه‌ای که به‌این طریق بدست می‌آید، دو ویژگی دارد: اول این‌که کفایت مجموعه آزمون اصلی در پوشش مؤلفه‌های برنامه (نسبت به یک معیار پوشش خاص) حفظ شده و نتیجتاً کارایی آن در کشف خطا افت چندانی نخواهد داشت. در ثانی دسته‌بندی موارد آزمون موجب تسهیل در شناسایی موارد آزمون منحصر بفرد و ضروری می‌گردد. زیرا موارد آزمونی که در دسته‌ی یکسانی قرار دارند از لحاظ پوشش عناصر برنامه یا شبیه هستند یا کاملاً یکسان. پس منطقی است که فرض کنیم کارایی موارد آزمون موجود در هر دسته در کشف خطا تقریباً یکسان خواهد بود. شبه‌کد الگوریتم پیشنهادی در شکل ۱ مشاهده می‌شود.

گام اول (مقداردهی اولیه): الگوریتم پیشنهادی برای نمونه‌برداری تک‌تک خوشه‌ها را مورد بررسی قرار می‌دهد. اما از آن‌جایی‌که هر خوشه حاوی موارد آزمونی است که نوع خاصی از مسیرهای اجرایی را می‌پوشانند، لذا نمونه‌برداری باید از خوشه‌هایی شروع شود که تعداد موارد آزمون آن‌ها کمتر است تا ابتدا مسیرهای اجرایی خاص‌تر پوشیده شوند. لذا در ابتدا آرایه‌ی خوشه‌ها را بر حسب تعداد موارد آزمون درون آن‌ها به‌صورت صعودی مرتب می‌کنیم. سپس متغیری را که شماره‌ی خوشه تحت بررسی را نگهداری می‌کند برابر صفر قرار می‌دهیم. مقدار صفر به‌معنای شروع از اولین خوشه است که حاوی کمترین مورد آزمون می‌باشد. در نهایت برای هر مورد آزمون نیز تعداد نیازمندی‌هایی که می‌پوشاند محاسبه شده و در آرایه‌ای قرار می‌گیرد. این آرایه بعداً در انتخاب مورد آزمون از هر خوشه مورد استفاده قرار خواهد گرفت.

گام دوم (تکرار نمونه‌برداری تا کفایت پوشش): در این گام حلقه‌ای در نظر گرفته شده است. شرط تکرار حلقه وجود دست‌کم یک نیازمندی پوشیده نشده تا آن زمان است. به‌عبارت دیگر، آرایه‌ای در نظر گرفته شده است (*marked*) که پوشیده‌شدن نیازمندی‌ها را نگهداری می‌کند. به‌محض انتخاب مورد آزمونی که یک نیازمندی را بپوشاند، درایه‌ی مورد نظر آن در این آرایه علامت‌گذاری می‌شود. شرط حلقه بررسی می‌کند که اگر درایه‌ای در این آرایه وجود دارد که هنوز علامت نخورده است، پس هنوز پوشش مجموعه‌ی کاهش‌یافته همانند مجموعه اصلی نشده و نمونه‌برداری باید ادامه یابد. جهت نمونه‌برداری از یک تابع کمکی به‌نام *SelectTest* استفاده می‌شود.

گام سوم (بروزرسانی متناظر با نمونه‌ی منتخب): در این مرحله اگر مورد آزمونی انتخاب شده باشد، ابتدا در مجموعه‌ی کاهش‌یافته درج شده و نیز از خوشه جاری حذف می‌گردد. سپس تمام نیازمندی‌هایی که مورد آزمون منتخب می‌پوشانند در آرایه *marked* علامت‌گذاری می‌شوند. در نهایت متغیر شاخص شماره خوشه یک واحد افزایش می‌یابد تا در تکرار بعد نمونه‌گیری از آن انجام شود.

نمونه‌برداری با تابع کمکی: این تابع (*SelectTest*) از میان موارد آزمون موجود در خوشه‌ی تحت بررسی، یک نمونه انتخاب می‌کند. عملکرد آن به‌شرح زیر است: ابتدا به‌ازاء تمام موارد آزمون موجود در خوشه‌ی تحت بررسی تعداد نیازمندی‌های پوشش‌نیافته‌ای که هر کدام می‌پوشانند محاسبه می شود. به‌عبارت دقیق‌تر، با مراجعه به ماتریس مورد آزمون-نیازمندی و نیز آرایه‌ی *marked* بررسی می‌شود که چند نیازمندی تاکنون توسط موارد آزمون موجود در مجموعه‌ی کاهش‌یافته تاکنون پوشیده نشده‌اند و از بین آن‌ها، چه تعداد توسط هر مورد آزمون درون خوشه پوشیده می‌شود. پس از این محاسبه، مورد آزمون با بیشترین تعداد انتخاب می‌شود. اگر در انتخاب، بین تعداد دو یا چند مورد آزمون تساوی ایجاد شود، در این‌صورت تعداد کل نیازمندهایی که

هر مورد آزمون در ماتریس مورد آزمون-نیازمندی می‌پوشاند، ملاک انتخاب خواهد شد. اگر مجدداً تساوی رخ دهد، یک مورد آزمون به تصادف انتخاب خواهد شد.

تحلیل بدترین زمان اجرا: فرض کنید که m تعداد نیازمندی‌های آزمون و nt تعداد موارد آزمون باشد. در بدترین حالت تعداد خوشه‌های موارد آزمون برابر تعداد موارد آزمون خواهد بود که در این‌صورت تعداد تکرارهای حلقه while نیز به همین تعداد و nt خواهد بود. اگر خوشه‌ی مورد بررسی تنها یک مورد آزمون داشته باشد، در این‌صورت زمان مورد

```
define:
    requirement: set of coverage requirements for minimization: r1,r2,...,rn
input:
    t1,t2,...,tm: all test cases present in the test pool
    cv[m,n]: coverage matrix representing requirement coverage of each test case. TRUE for covered, FALSE for uncovered
    clusters: array[1..k] of cluster instances, each containing similar test cases according to some dissimilarity metric
output:
    RS: a reduced suite of test cases from the test pool.
declare:
    nextTest: one of test cases
    currentClusterIndex: index of currently processing cluster
    marked: array[1..n] of boolean, initially FALSE
    numCovered: array[1..n] of the number of requirements that each test case in testCaseSet covers
    list: list of ti's
    Card(): returns the cardinality of a set, Sort(): sorts the input array

algorithm TestSuiteReduction
begin
 STEP 1:        currentClusterIndex := 0;  // initialization
                Sort(clusters,ascending); // sorts the input array ascending or descending
              foreach ti do compute numCovered[i], the number of  requirements rj that ti covers;
 STEP 2:        while there exists ri such that marked[i] == FALSE do
                   if currentClusterIndex == k then currentClusterIndex := 0;
                   list := all tj ∈ clusters[currentClusterIndex];
                   nextTest := 0;
                   if Card(list) == 1 then
                      test :=  t ∈ list;
                      if there exists rj ∈ requirements where cv[test,rj] == TRUE and  marked[j] == FALSE then
                         nextTest := test;
                      endif
                   else if Card(list) > 1
                      nextTest := SelectTest(list, numCovered);
                   endif
 STEP 3:           if nextTest ≠ 0 then
                      RS := RS ∪ {nextTest};
                     clusters[currentClusterIndex] := clusters[currentClusterIndex] – nextTest;
                      foreach rj ∈ requirements where cv[nextTest,rj] == TRUE do marked[i] := TRUE;
                   endif
                   currentClusterIndex := currentClusterIndex + 1;
                endwhile
                return RS;
end TestSuiteReduction

function SelectTest(testCaseSet, numCovered)
declare:  /*This function selects the next test case to be in RS*/
    n: the number of test cases in the testCaseSet
    numUnmarked: array[1..n] of the number of unmarked requirements that each test case in testCaseSet covers
    testCase: selected test case from test case set, initially 0
    testList1, testList2: list of ti's
begin
    foreach ti in testCaseSet do
            compute numUnmarked[i], the number of unmarked requirements rj from requirements that ti covers;
    testList1 := all ti from testCaseSet for which numUnmarked[i] is the  maximum;
    if Card(testList1) ≠ 0 then
       if Card(testList1) == 1 then testCase := the test case in testList1;
       else
          testList2 := all ti from testList1 for which numCovered[i] is the maximum;
          if Card(testList2) ≠ 0 then
             if Card(testList2 == 1) then testCase := the test case in testList2;
             else testCase := any test case in testList2;
             endif
          endif
```

```
    endif
  endif
  return testCase;
end SelectTest
```

شکل ۱: الگوریتم پیشنهادی برای کاهش مجموعه آزمون‌ها

نیاز جهت بررسی این‌که آیا نیازمندی پوشیده نشده‌ای وجود دارد که توسط آن پوشیده می‌شود یا خیر، حداکثر $O(nt.m)$ خواهد بود. در غیر این‌صورت تابع *SelectTest* فراخوانی می‌شود. این تابع حدکثر nt مورد آزمون انتخاب می‌کند. برای انتخاب هر مورد آزمون نیز باید حداکثر $nt.m$ محاسبه انجام دهد. لذا زمان کل این تابع $O(nt.nt.m)$ خواهد بود. علامت‌گذاری نیازمندی‌های پوشیده شده توسط مورد آزمون منتخب نیز حداکثر به زمان n نیاز دارد. بنابراین بدترین زمان اجرای الگوریتم پیشنهادی اول عبارت است از $O(nt.nt.m)$.

۴- ارزیابی

به‌منظور ارزیابی کارایی روش ارائه شده و مقایسه با روش‌های پیشین، آزمایش‌های تجربی ترتیب داده شده است. برپایی آزمایش‌های لازم در این حوزه، مقدمات خاصی نیاز دارد تا بتوان به‌شکل صحیحی روش پیشنهادی را ارزیابی نمود. علاوه بر این، برای این‌که نتایج حاصل از آزمایشات با نتایج سایر مطالعات قابل مقایسه باشد، فرایند برپایی آزمایش مشابه سایر مطالعات کلیدی در این زمینه صورت گرفته است. در آزمایش‌های انجام گرفته از مجموعه آزمون زیمنس شامل ۷ برنامه به زبان C است، استفاده شده است. این مجموعه برای هر برنامه، شامل مخزن موارد آزمون به‌صورت ورودی‌ها و خروجی‌های مورد انتظار، اسکریپت‌هایی برای اجرای موارد آزمون و نسخ تک خطایی برای هر برنامه، که در هر نسخه خطاها به‌طور دستی وارد شده‌اند. هر نسخه‌ی خطادار در این برنامه‌ها معادل برنامه‌ی اصلی است به‌جز این‌که یک خطای به‌خصوص در آن کاشته شده است. همه‌ی برنامه‌ها، نسخ خطادار و مخزن آزمون‌های مورد استفاده در آزمایشات توسط محققان زیمنس از طریق سایت [17] در دسترس است. اطلاعات خلاصه‌ای از مجموعه‌ی زیمنس در جدول ۱ نشان داده شده است.

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

مشابه آزمایش‌هایی که در [5] و [10] انجام شده است، در این تحقیق نیز برای برپایی و انجام آزمایش‌ها مجموعه آزمون‌های کافی در پوشش یال‌ها (لبه‌ها، انشعاب‌های برنامه) با روشی مشابه ایجاد شده است. روال مورد استفاده جهت تولید مجموعه آزمون‌های کافی در پوشش یال به‌شرح زیر است: ابتدا به‌طور تصادفی از مخزن آزمون‌ها موارد آزمون انتخاب شده و به مجموعه آزمون اضافه می‌شود. سپس تعدادی مورد آزمون اضافی که به‌طور تصادفی انتخاب شده‌اند، به مجموعه آزمون اضافه می‌شود تا مادامی‌که پوشش تجمعی یال‌ها در مجموعه آزمون همانند مخزن آزمون شده و مجموعه، کافی در پوشش یال گردد. تعداد تصادفی از موارد آزمون که در ابتدا در مجموعه‌ی تولیدی قرار گرفته است، متغیر بوده و بین ۰ تا ۰/۵، ۰ تا ۰/۴، ۰ تا ۰/۳، ۰ تا ۰/۲ و ۰ تا ۰/۱ برابر تعداد خطوط کد هر برنامه 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. فرآیندی را که در طی آن با اضافه نمودن کد به برنامه اصلی اطلاعات مربوط به پوشش عناصر یا مؤلفه‌های مختلف برنامه در حین اجرای یک مورد آزمون جمع‌آوری می‌شوند، اصطلاحاً مستندگذاری (تجهیز کد) می‌گویند. 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، $|F_{red}|$.

با در نظر گرفتن اطلاعات به‌دست آمده‌ی فوق، اطلاعات مهم زیر برای هر مجموعه آزمون محاسبه شده است:

۱- درصد کاهش اندازه‌ی مجموعه (٪کاهش). این مقدار از تقسیمِ اختلاف اندازه‌ی مجموعه اصلی با مجموعه‌ی کاهش‌یافته بر اندازه‌ی مجموعه اصلی به‌دست می‌آید. فرمول محاسبه‌ی آن به‌صورت زیر است:

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

۲- درصد فقدان کارایی کشف خطا (٪فقدان خطا). این مقدار از تقسیمِ اختلاف تعداد خطاهای قابل کشف توسط مجموعه اصلی و مجموعه‌ی کاهش‌یافته بر تعداد خطاهای قابل کشف توسط مجموعه‌ی اصلی به‌دست می‌آید. فرمول محاسبه‌ی آن به‌صورت زیر است:

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

در آزمایش‌های انجام شده، برای نمایش بصری داده‌های حاصل از محاسبه‌ی مقادیر درصد کاهش و درصد فقدان خطا و نیز مقایسه‌ی بین فنون کاهش، از نمودارهای جعبه‌ای، رایج در مطالعات این حوزه استفاده شده است. جهت تولید نمودارهای جعبه‌ای، از ابزاری به‌نام SAS/GRAPH استفاده 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]. در این آزمون نمونه‌های دو جامعه، تعداد خطاهای متمایزی است که توسط هر یک از ۱۰۰۰ مجموعه آزمون کاهش‌یافته در محدوده‌ی اندازه‌ی ۰ تا ۰/۵ توسط روش پیشنهادی و روش مورد مقایسه آشکار می‌شود. در این‌صورت فرض صفر را به این‌صورت در نظر می‌گیریم که هیچ اختلافی بین میانگین تعداد خطاهای آشکار شده توسط الگوریتم پیشنهادی و الگوریتم مورد مقایسه وجود ندارد. با استفاده از فرمول زیر، مقدار z را محاسبه کرده و آنرا در جدول مقادیر بحرانی در مرجع [20] جستجو می‌کنیم. در این‌صورت درصد اطمینان برای رد فرض صفر به‌دست می‌آید. 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 و مطالعات 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. صحت این موضوع در مقاله‌ی [4] به اثبات رسیده است. بنابراین، ما نیز روش پیشنهادی خود را با این روش مقایسه کرده‌ایم. برای این منظور الگوریتم پیشنهادی و نیز الگوریتم اچ‌جی‌اس با زبان جاوا پیاده‌سازی شده است و نتایج کاهش مجموعه آزمون‌ها با هم 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 (به‌دلیل کمبود فضا). 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.

کاهش اندازه‌ی مجموعه: نتایج شکل ۲ نشان می‌دهند که الگوریتم پیشنهادی مجموعه‌های آزمون مربوط به برنامه‌های *replace*، *sched*، *tcas* و *totinfo* را به‌طور متوسط اندکی کمتر کاهش داده است ولی میزان کاهش در مورد همه‌ی برنامه‌ها در کل بالاست. در مورد برنامه‌ی *sched2*، هر دو روش تقریباً یکسان عمل کرده‌اند. برای برنامه‌های *ptok* و *ptok2*، روش پیشنهادی به‌طور میانگین مجموعه‌ها را بیشتر از الگوریتم اچ‌جی‌اس کاهش داده است.

فقدان کشف خطا: میانه‌ی درصد فقدان کشف خطا مربوط به برنامه‌های *replace*، *sched*، *tcas* و *totinfo* به‌طور چشمگیری کاهش یافته یا محدوده‌ی آن پایین‌تر از درصد فقدان کشف خطای مجموعه‌ها با روش اچ‌جی‌اس قرار گرفته است. در مورد برنامه‌ی *sched2*، محدوده‌ی درصد فقدان کشف خطا توسط روش پیشنهادی اول اندکی کمتر است ضمن این‌که باید توجه کرد که این برنامه فقط ۱۰ نسخه‌ی خطادار دارد و باعث می‌شود قدرت آشکارسازی خطای مجموعه‌ی کاهش‌یافته به‌درستی نمایان نشود. برای برنامه‌های *ptok* و *ptok2*، میانه‌ی درصد فقدان کشف خطا با روش اچ‌جی‌اس تقریباً برابر است. اما همان‌طور که از جدول ۲ بر می‌آید، برتری روش پیشنهادی اول در بهبود نرخ کشف خطا، با اطمینان بالاتر از ۹۸٪ برای همه‌ی برنامه‌ها بدیهی و از لحاظ آماری معنی‌دار است.

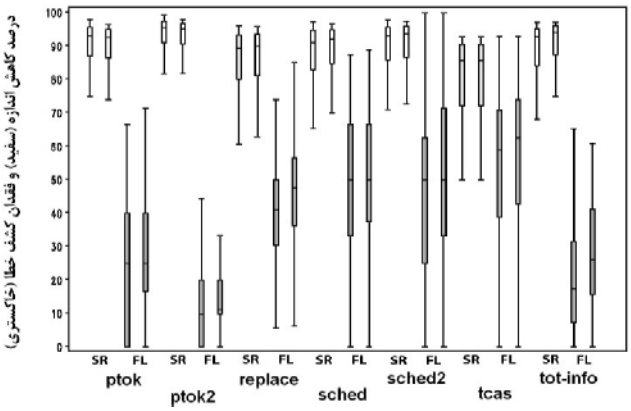
جدول ۲: مقادیر z محاسبه شده و درصد اطمینان رد فرض صفر برای هر برنامه در آزمایش

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، الگوریتم‌های پیشنهادی با کاراترین و معروف‌ترین الگوریتم کمینه‌سازی موجود مقایسه شده است. در این آزمایش‌ها، مجموعه‌های کاهش‌یافته توسط الگوریتم پیشنهادی، همواره نرخ کشف خطای بالاتری نسبت الگوریتم موجود از خود نشان دادند. معنی‌دار بودن میزان بهبود در کشف خطا توسط الگوریتم پیشنهادی، از لحاظ آماری نیز بررسی و تأیید شده است.



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

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.