4th International Conference on Web Research
April 25-26, 2018; Tehran, IRAN
ICWR2018
کنفرانس بین المللی وب پژوهی
چهارمین
۵ و ۶ اردیبهشت ۱۳۹۷؛ تهران، ایران

شناسایی کلون‌های معنایی با استفاده از حالت حافظه انتزاعی و گراف وابستگی برنامه

فاطمه عظیم زاده [۱]، حمید نصیرلو[۲]

[۱] عضو هیات علمی مرکز اطلاعات علمی جهاد دانشگاهی (sid) ،
f.azimzadeh@gmail.com

[۲] دانشگاه علم و فرهنگ
hamid.nasirlou@gmail.com

چکیده

در مهندسی نرم‌افزار مهمترین مساله، ارائه نرم‌افزارهای با کیفیت و با کارایی بالا و خدمات پس از فروش آن است. به همین دلیل مهندسان نرم‌افزار، شاخه‌ی بخصوصی را با نام تکامل نرم‌افزار (Software Evolution) معرفی کردند که در آن هدف، ارتقای نرم‌افزارها پس از تولید آن‌ها است. یکی از مباحث پایه در تکامل نرم‌افزار، تشخیص کلونی‌های کد (Code Clone) یا همان تکه کدهای تکراری در نرم‌افزارها است. در حقیقت شاید بتوان تشخیص کلونی‌ها را پایه تکامل نرم‌افزار معرفی نمود، چرا که بیشتر مباحث تکامل نرم‌افزار، به نوعی به تشخیص کلونی‌ها وابسته هستند. تاکنون روش های متنوعی، از جمله دو روش مبتنی بر رفتار (کندتر و دقیقتر) و مبتنی بر حالت حافظه (سریعتر با دقت متوسط) ارائه شده‌اند. در این پژوهش، هدف یافتن کلون‌های بیشتر با دقتی مناسب نسبت به روش مبتنی بر حالت حافظه است(کاهش False Negative). برای انجام این کار از ترکیب دو روش حالت حافظه انتزاعی(Abstract Memory State) و گراف وابستگی برنامه(Program Dependency Graph) استفاده شده است. ضمنا از روش اجرای تکه کدها با مقادیر تصادفی نیز بهره برده شده است. روش ارائه شده در این پژوهش با روش مبتنی بر حالت حافظه مقایسه شده و در نهایت، ارزیابی‌ها نشان می‌دهند که این پژوهش توانسته است کلون‌های نوع ۱،۲،۳ و ٤ را شناسایی کند و False Negative را کاهش دهد.

کلمات کلیدی

مهندسی نرم‌افزار، تکامل نرم‌افزار، کلونی‌های کد، کلون‌های معنایی

۱- مقدمه

یکی از زمینه‌های تحقیقاتی مهندسی نرم‌افزار ، تکامل نرم‌افزار است. در تکامل نرم‌افزار بحث‌های مربوط به توسعه و نگهداری نرم‌افزار، پس از تولید آن، مطرح می‌شود. یکی از بخش‌های مهم در تکامل نرم‌افزار، تشخیص کلونی‌های کد (Code Clones) در نرم‌افزار است. با توجه به تعریف ارائه شده توسط Roy و همکارانش [۱]، کلونی‌های کد، تکه کدهایی هستند که به علت کپی و تکرار آن توسط توسعه دهندگان نرم‌افزار به صورت خواسته و یا ناخواسته، در نقاط مختلف نرم‌افزار به وجود می‌آیند. تحقیقات گذشته نشان داده که ۷ الی ۲۳ درصد از سیستم‌های نرم‌افزاری دارای کدهای تکراری هستند [۱,۲,۳,۴,۵,۶] و این میزان تا ۵۹ درصد نیز گزارش شده است[۷]. بنابراین حجم قابل توجهی از نرم‌افزار را کلونی‌های کد تشکیل می‌دهند. کلونی‌های کد می‌توانند فرآیند توسعه و نگهداری نرم‌افزار را با مشکل روبروکنند که در نهایت منجر به کاهش رضایت‌مندی مشتریان و افزایش هزینه‌ها خواهد شد. بنابراین مساله اصلی یافتن این تکه کدها و سپس حذف، مدیریت و کنترل آن‌ها می‌باشد. از جمله مزایای تشخیص کلونی‌های کد می‌توان به تشخیص دزدی‌های ادبی (Plagiarism Detection)، تشخیص

خطا (Bug Detection)، شناسایی کدهای کاندید برای تبدیل برنامه به جنبه گرایی (Aspect Mining)(Aspect-Oriented)، شناسایی ویروس‌ها، تشخیص نقض گواهی مالکیت (License Violation Detection) و ادغام کدهای منبع اشاره کرد (Merging)[۸]. در بخش زیر تعاریف اولیه و انواع کلونی‌های کد ارائه شده است.

انواع کلونی‌های کد

تاکنون مراجع مختلف، تقسیم‌بندی‌های مختلفی را برای کلونی‌های کد ارائه کرده‌اند:

نوع ۱ (Exact Clones)

کپی عینا کد از لحاظ متنی و دارای تفاوت در فضاهای خالی (Spaces)، توضیحات (Comments) و قالب [Layout] [۹,۱۰]

نوع ۲ (Renamed Clones)

کپی کد از لحاظ نحوی (Syntactically)، شامل تغییرات بیشتر بر روی متغیرها (Variables)، نوع‌ها(Types)، شناسه‌های توابع (Function Identifier) و ثابت‌ها [۹,۱۰] (Literals)

نوع ۳ (Gapped Clones)

کپی کد، شامل تغییرات بیشتر همچون حذف، تغییر و یا افزودن دستورات[۹,۱۰]

نوع ٤ (Semantic Clones)

قطعات کد مشابه از لحاظ عملکرد، اما متفاوت از لحاظ نحوی، ساختاری و متنی[۱۰]

تا کنون سه روش کلی برای شناسایی کلون‌های معنایی ارائه شده است. این روش‌ها عبارت‌انداز:

روش‌های مبتنی بر گراف [۱۱,۱۲,۱۳]

روش‌های مبتنی بر رفتار[۱۴]

روش‌های مبتنی بر حالت حافظه [۱۵]

مشکل اصلی‌ای که در روش‌های مبتنی بر گراف وجود دارد این است که تنها وابستگی خطوط را در نظر می‌گیرند و در حالت کلی تنها توانایی شناسایی کدهایی با ترتیب خطوط متفاوت را دارند و اگر دو تکه کد که شباهت ساختاری ندارند (و یا شباهت کمی دارند) و عملکرد برابری دارند، به این تکنیک‌ها داده شود، نمی‌توانند آن کلون‌ها را تشخیص دهند (همانند دو کد مرتب‌سازی اعداد با روش کاری متفاوت). روش‌های مبتنی بر رفتار، در صورتی که بتوانند از تمام تکه کدها، با توجه به تمام ورودی‌های ممکن، اجرا بگیرند می‌توانند روش‌های بسیار دقیقی باشند. با توجه به توضیحات ارائه شده توسط Kim و همکارانش [۱۵]، روش ارائه شده در [۱۴] که مبتنی بر رفتار می‌باشد، دارای این مشکل است که از تمام ورودی‌های ممکن اجرا نمی‌گیرد که منجر به کاهش دقت آن می‌شود. از سوی دیگر روش‌های مبتنی بر رفتار بسیار زمان‌بر هستند و همچنین نیاز به تجهیزات محاسباتی بسیار قوی دارند. روش‌های مبتنی بر حالت حافظه انتزاعی دارای دقتی بیشتر از روش‌های مبتنی بر گراف و کمتر از روش‌های مبتنی بر رفتار می‌باشند. این روش‌ها دارای مزایایی همچون: زمان اجرای بسیار کم و دقتی نسبتا مناسب، می‌باشند. روش ارائه شده در [۱۵]، که مبتنی بر حالت حافظه انتزاعی می‌باشد، تنها کلون‌های تابعی را تشخیص می‌دهد و سطوح دانه‌بندی کوچکتر از تابع را نمی‌تواند تشخیص دهد. مشکل دیگر روش[۱۵] این است که کدهای نامرتبط را در فرآیند ساخت حافظه انتزاعی آن تکه کد لحاظ می‌کند. روش‌های ترکیبی که توانایی تشخیص کلونی‌های نوع ۴ را دارا باشند، تاکنون بسیار کم ارائه شده‌اند و تکنیک‌هایی هم که این توانایی را دارند (همچون روش [۱۳]) تنها از روش‌های مبتنی بر گراف بهره می‌برند. در این پژوهش سعی بر این است که مشکلات مطرح شده در بالا، برطرف گردد. روش کار به این صورت است که عمل تشخیص کلونی‌های موجود، با استفاده از روش حالت حافظه انتزاعی در سطوح دانه‌بندی کوچکتری (نسبت به تابع [۱۵]) انجام شود. همچنین گراف وابستگی برنامه، نیز استخراج گردد و کلونی‌های آن، تشخیص تشخیص داده شود که در نتیجه این دو عمل، انتظار کاهش False Negative می‌رود. در ادامه تحقیق به این موضوع پی برده شد که تشخیص برابری گزاره‌ها و عبارت‌های محاسباتی تنها با توجه به متن آن‌ها می‌تواند کاهش دقت را به دنبال داشته باشد. با توجه به این مساله از اجرای تکه کدها با مقادیر تصادفی نیز بهره گرفته شد.

۲- ادبیات تحقیق

با توجه به انواع کلونی‌های کد، روش‌های بسیاری برای یافتن این تکه کدها ارائه شده است. مراجع تقسیم‌بندی‌های تقریبا مشابهی را برای این روش‌ها ارائه کرده‌اند[۹,۱۱,۱۶,۱۷]، اما در یک جمع‌بندی کلی، تقسیم‌بندی زیر را می‌توان ارائه نمود (از آنجائیکه تمرکز روش ارائه شده در این پژوهش بر روی روش‌های ترکیبی و معنایی است، بیشتر ارجاعات به این روش‌ها می‌باشد)

۲-۱- مبتنی بر متن یا رشته (Text or String based)

دراین روش‌ها متن خام کدهای مختلف با یکدیگر مقایسه می‌شوند و (معمولا) هیچگونه تبدیل و یا نرمال‌سازی، در این روش‌ها انجام نمی‌گیرد. کاملا واضح است که در این روش‌ها کلون‌های نوع ۱ شناسایی می‌شوند[۱۸].

۲-۲- روش‌های مبتنی بر معیار (Metric based)

در این روش‌ها ابتدا یک سری معیار (Metric) مشخص می‌شود، سپس تکه‌های کد بر اساس این معیارها دسته‌بندی می‌شوند. در نهایت هر کدام از دسته‌ها شامل کلونی‌های کد خواهند بود. این‌که این روش‌ها بتوانند کدام‌یک از انواع کلون‌ها را تشخیص دهند، رابطه‌ی مستقیمی با معیارهای درنظر گرفته شده دارد، اما معمولا در شناسایی کلون‌های نوع ۱، نوع ۲، نوع ۳ به‌کار برده می‌شوند[۱۹, ۱۸].

۲-۳- روش‌های مبتنی بر گراف (Graph based)

در این روش‌ها، ابتدا برای کدهای منبع، گراف‌های وابستگی برنامه PDG (Program Dependency Graph) ایجاد می‌شود که دارای یال‌هایی برای مشخص کردن وابستگی کنترلی (Control Dependence) و وابستکی داده‌ای (Data Dependence) میان دستورات برنامه می‌باشند. سپس این گراف‌ها با یکدیگر مقایسه شده و کلونی‌های کد شناسایی می‌شوند. در گذشته روش‌های مبتنی بر گراف را جزء روش‌های شناسایی کننده کلون‌های نوع ۴ دسته بندی می‌کردند، اما امروزه، این روش‌ها در دسته

روش‌های تشخیص دهنده کلون‌های نوع ۱، نوع ۲ و نوع ۳ قرار گرفته‌اند. هرچند همچنان برخی محققان این قبیل روش‌ها را جزء روش‌های تشخیص دهنده‌ی کلون‌های نوع ۴ قرار می‌دهند[۱۱,۱۲,۱۳,۲۰].

۲-٤- روش‌های مبتنی بر حالت حافظه (Memory (State based

در این روش‌ها حالت‌های حافظه انتزاعی (Abstract Memory State of Code) با یکدیگر مقایسه می‌شوند. این روش‌ها نیز در تشخیص کلون‌های نوع ۴ به کار گرفته شده‌اند. هرچند سرعت آن‌ها نسبت به روش‌های مبتنی بر رفتار بیشتر بوده و هزینه‌ی کمتری دارند، اما تاکنون دقت و قدرت آن‌ها نسبت به روش‌های مبتنی بر رفتار کمتر بوده است[۱۵].

۲-۵- روش‌های مبتنی بر رفتار (Behavior based)

در این روش‌ها از تکه کدها اجرا گرفته می‌شود، به این صورت که به تکه کدها ورودی‌های یکسان داده می‌شود، سپس اگر خروجی این تکه کدها با هم یکسان باشد، این تکه کدها به عنوان کد کلون معرفی می‌شوند. این روش‌ها در تشخیص کلون‌های نوع ۴ به کار گرفته می‌شوند. از مزایای این روش می‌توان دقت و قدرت بالا در تشخیص کلون‌های کد و از معایب آن می‌توان به هزینه و زمان زیاد، اشاره کرد[۱۴].

۲-٦- روش‌های مبتنی بر نشانه و یا لغوی (Lexical (or Token based

در این روش‌ها کد منبع به یک توالی (Sequence) از نشانه‌ها تبدیل می‌شود و بعد از آن نشانه‌ها با هم مقایسه می‌شوند. این روش‌ها در شناسایی کلون‌های نوع ۲ بسیار قدرتمند هستند که در آن‌ها تغییرات اندک و در حد فضاهای خالی، توضیحات و قالب است[۲۱,۲۲].

۲-۷- روش‌های مبتنی بر درخت (Tree Based)

در این روش‌ها ابتدا کدهای منبع به تعدادی درخت نحو انتزاعی (Abstract Syntax Tree) AST تبدیل می‌شوند. سپس این ASTها با یکدیگر مقایسه شده و کلونی‌های کد شناسایی می‌شوند. این روش‌ها می‌توانند کلون‌های نوع ۱، نوع ۲ و تا حدودی نوع ۳ را شناسایی کنند[۲۲,۲۳].

۲-۸- روش‌های ترکیبی (Hybrid)

این روش‌ها، از ترکیبی از روش‌های ۲-۱-۱ تا ۲-۱-۷ برای تشخیص کلون‌ها استفاده می‌کنند. همچنین مراجع، هرتکنیکی را که از روش‌های ۲-۱-۱ تا ۲-۱-۷ استفاده نمی‌کنند، جزء این روش‌ها به حساب می‌آورند[۳۱, ۳۰, ۲۹, ۲۸, ۲۷, ۲۶, ۲۵, ۲۴, ۲۲, ۱۸, ۱۳].

۳- شرح کلی سیستم

ابتدا همانند تمام روش‌های تشخیص کلون، می‌بایستی یک نرمال‌سازی اولیه برروی کدهای ورودی انجام گیرد تا زوائد از متن ورودی حذف گردد. سپس کدهای نرمال شده به مرحله آنالیز لغوی ارسال می‌شوند. در این مرحله تمام متغیرهای تعریف شده به تفکیک نوع آن‌ها استخراج می‌شوند. سپس شرط‌های موجود استخراج شده و گزاره برای متغیرها با توجه به عبارت انتساب یافته به آن‌ها ایجاد می‌شود. حال با توجه به گزاره استخراج شده و عبارت منتسب به هر متغیر، یک گزاره ایجاد و برای هر متغیر ذخیره می‌شود. یک گراف وابستگی نیز برای برنامه ایجاد می‌گردد که هر کدام از گره‌های آن مربوط به یکی از خطوط برنامه است. سپس وارد مرحله نرمال‌سازی ثانویه می‌شود. در این مرحله برای هر خط از کد، یک مقدار هش کد تولید می‌شود و تا حد امکان سعی می‌شود اثر جابه‌جایی متغیرها و تفاوت نام آن‌ها از بین برود تا کد هش تولید شده در دو کد ورودی به یکدیگر نزدیکتر باشند. در ادامه خروجی دو مرحله آنالیز لغوی و نرمال‌سازی ثانویه به مراحل "مقایسه گراف وابستگی برنامه دو کد منبع" و همچنین "مقایسه حافظه انتزاعی متغیرهای دو کد منبع" ارسال می‌شوند. در مرحله مقایسه گراف وابستگی برنامه دو کد منبع، مقادیر کد هش تمام خطوط به صورت دو به دو با یکدیگر مقایسه می‌شوند (برابری گراف) و تا حد امکان، این دو گراف پیمایش شده و مقادیر هش خطوط با یکدیگر مقایسه می‌شوند و در نهایت این شباهت‌ها به عنوان کلون در خروجی به چاپ می‌رسند. در مرحله مقایسه حافظه انتزاعی متغیرهای دو کد منبع، حافظه انتزاعی متغیرها به صورت دوبه‌دو با یکدیگر مقایسه می‌شوند و در صورت برابری، وابستگی‌های کنترلی و داده‌ای آن‌ها استخراج شده و به صورت جداگانه در خروجی به چاپ می‌رسند. الگوریتم‌های (۱،۲و۳) عملیات شرح داده شده را نشان می‌دهد. در ادامه تمام مراحل مذکور به همراه عملکرد الگوریتم شرح داده خواهند شد.

```
Algorithm 1: DSCCD
Input: SourceCode
Output: CodeClones
-----------------------------------------------------------------
Foreach line in SourceCode do
  Remove Comments and Spaces

Foreach variable in SouceCode
  Create Predicate

Foreach line in SourceCode
  Create Control Dependency Gragh and Data Dependency Gragh

Foreach Variable in SourceCode
  Remove Variable Names and Create HashCode

Call FindAMSClones(SourceCode)
Call FindPDGClones(SourceCode)
```

الگوریتم DSCCD (۱)

```
Algorithm 2: FindAMSClones
Input: SourceCode
Output: AMSCodeClones
-----------------------------------------------------------------
Foreach pi in Predicates
  Foreach pj in Predicates
    IF i≠j Then
      cpr = ComparePredicate(pi, pj)
      IF cpr = True Then
        csr = CompareStatements(pi, pj)
        IF csr > Similarity Then
          Write Related Lines For pi and pj
```

الگوریتم FindAMSClones (۲)

```
Algorithm 3: FindPDGClones
Input: SourceCode
Output: PDGCodeClones
-----------------------------------------------------------------
Foreach hi in HashCodes
  Foreach hj in HashCodes
    IF i≠j and hi=hj Then
      Compare and Write Related Lines For hi and hj
```

الگوریتم FindPDGClones (۳)

۳-۱- نرمال‌سازی اولیه

ابتدا فایل‌ها به صورت متنی خوانده می‌شوند و در یک آرایه‌ی رشته‌ای که هر مدخل آن مربوط به یکی از خطوط کد است ذخیره می‌گردد. سپس این آرایه به مرحله نرمال‌سازی اولیه منتقل می‌شود. در این مرحله بخش‌های اضافی هر کد منبع حذف می‌شود. این بخش‌ها شامل توضیحات (همچون کاراکترهای "//" و "*/" و "*/")، فضاهای خالی و خطوط اضافی می‌باشد. در مرحله بعد هر خط از کد به صورت مجزا خوانده شده و به همراه یک شماره خط در آرایه‌ای کاراکتری ذخیره می‌شود. این آرایه کاراکتری، یک آرایه یکپارچه است که تمام کاراکترهای موجود در کد منبع را در بر می‌گیرد. همچنین برای هر خط از کد یک نمونه خالی از کلاس LineHashed ایجاد می‌شود که در ادامه به ساختار و کاربرد آن اشاره خواهد شد. تمام این مراحل در یک پروسه که خواندن فایل نامیده می‌گردد، انجام می‌شود. شکل (۱) و شکل (۲)، نشان دهنده این موضوع است.

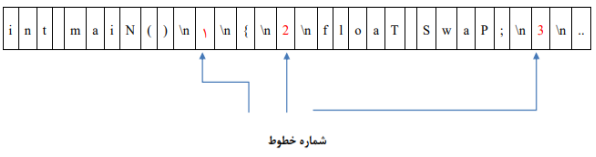
```
int main() /* this Comment is for Test*/
{
  float swap;
  int b = 9, a; /* this Comment is for Test*/
  فضای خالی
  if(((a + 24) * b) > 8 && (a * b) < 90)
  {
    a = b * 4;// this Comment is for Test
    if((b - a) == 20)
    {
      swap = (24 * a) / b;
    }
  }
  return 0;
}
```

شکل (۱) : کد ورودی قبل از نرمال سازی اولیه

```
int main()
{
  float swap;
  int b = 9, a;
  if(((a + 24) * b) > 8 && (a * b) < 90)
  {
    a = b * 4;
    if((b - a) == 20)
    {
      swap = (24 * a) / b;
    }
  }
  return 0;
}
```

شکل (۲) : کد ورودی بعد از نرمال سازی اولیه

حال تمام کاراکترهای این خطوط به یک آرایه‌ی سراسری به نام Token منتقل می‌شوند. این آرایه برای تکه کد بالا به صورت شکل (۳) خواهد بود.



شکل (۳)

این آرایه کمک می‌کند تا پردازش بر روی کد منبع به صورت کاراکتر به کاراکتر انجام شود. همانطور که ملاحظه می‌شود، شماره خطوط نیز در این آرایه ذخیره شده است تا در طی مراحل بعدی مورد استفاده قرار گیرد. در ادامه، این آرایه به مرحله آنالیز لغوی منتقل می‌شود.

۳-۲- آنالیز لغوی (Tokenization)

۳-۲-۱- بررسی تابع ورودی

در این مرحله ابتدا تابع اصلی ورودی کد منبع (Main) شناسایی می‌شود. سپس متغیرهای تعریف شده در تابع خوانده می‌شوند و به مرحله شناسایی و استخراج متغیرها منتقل می‌شوند.

۳-۲-۲- شناسایی و استخراج متغیرها

در این مرحله برای هر یک از متغیرها، یک نمونه از کلاس OriginalVariable ساخته می‌شود که به شرح شکل (۴) است:

```
public class OriginalVariable
    {
        public string name;
        public string type;
        public List<int> lineNumbers = new List<int>();
        public string predicateValue = new List<string>();
        public OriginalVariable() { }
        public OriginalVariable(string name)
        {
            predicateValues.Add(name);
            this.name = name;
        }
    }
```

شکل (٤) : کلاس OriginalVariable

در اینجا صفت Name مشخص کننده نام متغیر است. صفت Type مشخص کننده نوع متغیر است. آرایه LineNumbers شماره خطوط مقداردهی آن متغیر (انتساب) و وابستگی‌های کنترلی‌اش را ذخیره می‌کند که در استخراج خطوط وابسته به حافظه انتزاعی متغیر کاربرد دارد. صفت predicateValue نیز مقدار گزاره نهایی را برای متغیر ذخیره خواهد نمود. در ادامه نحوه ساخت گزاره‌ها شرح داده خواهد شد. در ادامه این مرحله، تمامی کد منبع خوانده شده و متغیرهای موجود در آن شناسایی و یک نمونه از کلاس متغیرها(کلاس فوق) برای هر کدام از آن‌ها ساخته می‌شود. سپس تمامی این متغیرها در متغیری با نام originalVariables که لیستی از نمونه‌های این کلاس را نگهداری می‌کند، قرار داده می‌شوند.

۳-۲-۳- استخراج شرط‌ها و ساخت گزاره متغیرها

در این مرحله هر کدام از دستورات شرطی همچون if و else شناسایی می‌شوند. به محض برخورد با این دستورات، مقدار گزاره‌ی شرطی موجود در آن‌ها استخراج شده و در یک متغیر به نام Predicate ذخیره می‌شود. لازم به ذکر است که دستورات شرطی تودرتو به خوبی کنترل شده‌اند. این به این معنی است که اگر چندین if تودرتو وجود داشته باشد، شرط موجود در گزاره حاصل شده در داخلی‌ترین if شامل تمام شرط‌های if سطوح بالاتر است. ساختار گزاره‌ها به شکل زیر می‌باشد.

{عبارت محاسباتی متغیر , شرط متغیر}

باید دقت شود که عبارت‌های محاسباتی از ترکیب متغیرها و عملگرها ساخته شده‌اند. بنابراین عبارت‌های محاسباتی نیز می‌توانند شامل تعدادی گزاره‌ی دیگر باشد.

برای روشن شدن موضوع به شکل (۵) توجه کنید

```
1- if(((a + 24) * b) > 8 && (a * b) < 90)
2- {
3-    a = b * 4;
4-    if((b - a) == 20)
5-    {
6-       swap = (24 * a) / b;
7-    }
8- }
```

شکل (۵)

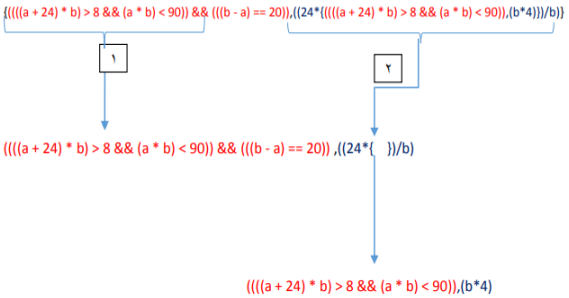
اگر قرار باشد شرطی برای گزاره متغیر Swap نوشته شود، آن شرط به صورت زیر خواهد بود

$$((((a + 24) * b) > 8 \&\& (a * b) < 90)) \&\& (((b - a) == 20))$$

در حقیقت، این شرط ترکیبی از دو شرط if بیرونی است. شرط‌های موجود در دستورات else if هم به این شکل کنترل می‌شوند که شرط‌های موجود در آن‌ها (در صورت وجود) با شرط if اصلی && (And) می‌شوند و در ابتدای گزاره شرط if اصلی یک علامت ! قرار می‌گیرد تا گزاره‌ی else if را به درستی شکل دهد. در ادامه مثالی ارائه شده است که در آن یک گزاره کامل ساخته می‌شود.

به شکل (۶) و مقدار گزاره متغیر Swap توجه کنید

```
1-  int main()
2-  {
3-     float swap;
4-     int b = 9, a;
5-     if(((a + 24) * b) > 8 && (a * b) < 90)
6-     {
7-        a = b * 4;
8-        if((b - a) == 20)
9-        {
10-          swap = (24 * a) / b;
11-       }
12-    }
13-    return 0;
14- }
```



شکل (٦)

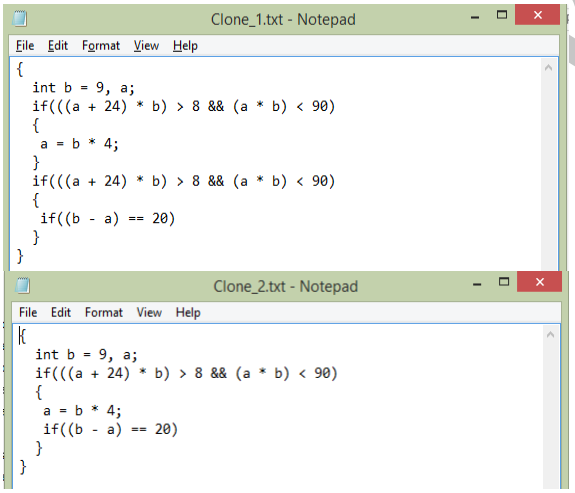
در شکل (۶) قسمت‌های قرمز رنگ، مربوط به شرط و قسمت‌های آبی مربوط به عبارت محاسباتی می‌باشد. دقت در شرط ۱ نشان میدهد این شرط از ترکیب دو شرط if بیرونی ساخته شده است (خطوط ۵ و ۸). عبارت محاسباتی متغیر Swap برابر (24*a)/b می‌باشد که در آن، خود متغیر a نیز دارای یک گزاره‌ی دیگر می‌باشد که در آن هم شرط و عبارت محاسباتی دیگری قرار دارد. این گزاره‌ها برای تمام متغیرها ایجاد می‌شود.

یکی از مزیت‌هایی که استفاده از این گزاره‌ها دارند در مثال زیر قابل مشاهده است. به شکل (۷) توجه نمایید

```
1-  int main()
2-  {
3-     float swap;
4-     int b = 9, a;
5-     if(((a + 24) * b) > 8 && (a * b) < 90)
6-     {
7-        a = b * 4;
8-     }
9-     if(((a + 24) * b) > 8 && (a * b) < 90)
10-    {
11-       if((b - a) == 20)
12-       {
13-          swap = (24 * a) / b;
14-       }
15-    }
16-    return 0;
17- }
```

شکل (۷)

این تکه کد مشابه با تکه کد قبلی است که گزاره‌ها از آن استخراج شدند، اما تفاوت آن در این است که عبارت a=b*4 در شرطی جداگانه مقداردهی شده است که مقدار شرط آن با شرطی که متغیر sawp در آن مقداردهی شده است دارای اشتراک است. روش AMS به خوبی این موارد را پوشش می‌دهد، در حالی که روش PDG توانایی تشخیص این چنین کلونی‌هایی را ندارد. اما در طی تستی که بر روی این دو تکه کد مذکور انجام شد، نتایج جالبی به دست آمد. الگوریتم PDG، کلونی‌ای را تشخیص داد که پیش‌بینی نمی‌شد. در شکل (۸) این کلونی مشاهده می‌شود.



شکل (۸)

این شکل فایل خروجی مربوط به دو تکه کد را که تشکیل کلونی می‌دهند به نمایش گذاشته است. الگوریتم PDG ابتدا مبدا این کلونی را در دستور if((b-a)==20) یافته است که در هر دو تکه کد برابر است. سپس با مقایسه وابستگی‌های کنترلی و داده‌ای این دو تکه کد، خطوط مشابه دیگر را نیز شناسایی کرده است. علت چنین تشخیصی در آن است که در PDG عمل تشخیص بر اساس کد هش تولید شده، صورت می‌گیرد و از آنجاییکه متغیر a در زیر مجموعه شرطی برابر در دو تکه کد است هیچ‌گونه تداخلی در این تشخیص ایجاد نمی‌کند.

۳-۲-٤- استخراج حافظه انتزاعی متغیر یا AMS

حال باید به این سوال پاسخ دهید که گزاره‌های به دست آمده در مرحله قبل برای چه استخراج شده و در کجا استفاده می‌شوند؟ همانطور که پیش‌تر نشان داده شد، این گزاره‌ها برای ساخت گزاره اصلی هر متغیر به کار می‌روند. بنابراین در ابتدای هر بلاک if و یا else if، می‌بایستی این شرط‌ها استخراج شوند تا در هنگام ساخت گزاره اصلی هر متغیر از آن‌ها استفاده شود.

در این مرحله کد منبع خوانده شده و هر کدام از دستورات انتساب در آن بررسی می‌شود. سپس شرط‌های ساخته شده در مرحله قبل به کار گرفته می‌شوند تا عمل ساخت گزاره اصلی برای هر متغیر انجام شود.

یکی از نکاتی که به طور دقیق در الگوریتم آنالیز لغوی رعایت شده است، توجه به سطوح بلاک‌ها در کد منبع است. توجه به این نکته در دو قسمت از کار اهمیت دارد: اول اینکه همانطور که پیش‌تر هم عنوان شد، برای ساخت گزاره اصلی هر متغیر باید گزاره شرطی هر سطح با گزاره شرطی سطوح دیگر And شود که برای این کار باید بدانید در کدام سطح قرار دارید تا گزاره شرطی صحیحی انتخاب شود. برای مثال به شکل (۹) توجه کنید.

```
1-  int main()
2-  {
3-    if(X)                    شرط ۱
4-    {
5-      if(Y)                  شرط ۲
6-      {
7-          a = b % 2;
8-      }
9-      if(Z)                  شرط ۳
10-     {
11-         a = b * 4;
12-         if(K)              شرط ۴
13-         {
14-             swap = (24 * a) / b;
15-         }
16-     }
```

شکل (۹)

در این تکه کد، وقتی آنالیز لغوی وارد شرط ۱ می‌شود مقدار شرط برای گزاره متغیرها برابر X می‌شود. وقتی وارد شرط ۲ می‌شود، مقدار این شرط به X && Y تغییر می‌کند. سپس در هنگام خروج از شرط ۲، مقدار این شرط مجددا X می‌شود. همچنین هنگامی‌که وارد شرط ۳ می‌شوید مقدار شرط برابر X && Z و در آخر هنگام ورود به شرط ۴، مقدار شرط برابر X && Z && K خواهد شد. باید دقت داشت، هنگامی که آنالیزور با یک عبارت محاسباتی روبه‌رو می‌شود که یک متغیر در آن مقداردهی شده است، شروع به ساختن گزاره آن می‌کند که طی آن شرط جاری به دست آمده بر اساس توضیحات فوق را در آن قرار می‌دهد.

بنابراین برای داشتن مقدار صحیح گزاره‌های شرطی باید به سطوح توجه کرد. اهمیت دیگر توجه به سطوح، در تشکیل گراف وابستگی برنامه است. چراکه وابستگی کنترلی خطوط به بلاک‌های شروع و پایانی که در آن‌ها قرار دارند می‌باشد.

۳-۲-۵- استخراج گراف وابستگی برنامه یا PDG

همانطورکه در بخش نرمال سازی اولیه توضیح داده شد، برای هرخط از کد یک نمونه از کلاس LineHashed ساخته می‌شود. شکل (۱۰) ساختار این کلاس را نمایش می‌دهد.

```
public class LineHashed
    {
        public LineHashed() { }
        public LineHashed(int lineHashValue)
        {
            this.lineHashValue = lineHashValue;
        }
        public int lineHashValue;
        public List<int> controlDependenceLineNumbers = new List<int>();
        public List<int> dataDependenceLineNumbers = new List<int>();
    }
```

شکل (۱۰)

صفت LineHashedValue مقدار کد هش خط جاری را ذخیره می‌کند که البته این مقدار در مرحله (نرمال سازی ثانویه) بدست می‌آید. صفت controlDependenceLineNumbers وابستگی‌های کنترلی و صفت dataDependenceLineNumbers وابستگی‌های داده‌ای هر خط از کد را ذخیره می‌کند. دقت شود که هر دو این صفات وابستگی، لیستی از اعداد صحیح را در خود ذخیره می‌کنند.

در ادامه، متغیرهای موجود در هر گزاره، استخراج می‌شود. سپس یک وابستگی داده‌ای از خط جاری کد منبع که گزاره در آن قرار گرفته به آخرین مقداردهی (و یا تعریف اولیه متغیر) ایجاد می‌شود. لازم به ذکر است که شماره خطوط در این کلاس ذخیره نمی‌شود. اینکه الگوریتم از کجا متوجه می‌شود که هر نمونه از کلاس مربوط به کدام خط می‌باشد، از روی اندیس آرایه‌ی LineHashedList می‌باشد، یعنی اندیس ۱ مربوط به خط ۱ و ... می‌باشد. برای درک بهتر مطالب، فرض کنید که قرار است وابستگی کنترلی و داده‌ای برای خط شماره ۱۵ در تکه کد شکل (۱۱) ساخته شود.

```
1-  int main()
2-  {
3-  float swap;
4-  int b = 9;
5-  int a;
6-    if(b % 2 == 0)
7-    {
8-          a = b % 2;
9-    }
10-   if(((a + 24) * b) > 8 && (a * b) < 90)
11-   {
12-         a = b * 4;
13-         if((b - a) == 20)
14-         {
15-             swap = (24 * a) / b;
16-         }
17-   }
18- return 0;
```

شکل (۱۱)

در اینجا خطوط آبی نشان دهنده وابستگی‌های کنترلی و خطوط قرمز نشان دهنده وابستگی‌های داده‌ای می‌باشند. به این نکته مهم باید دقت شود که این وابستگی‌ها برای خطوط کد ایجاد می‌شود، نه برای متغیرهای آن. ایجاد وابستگی‌های کنترلی به این شرح است: همانطور که از شکل مشخص است، وابستگی‌های کنترلی مربوط به بلاک‌های شروع و پایان و همچنین مربوط به دستورات آن بلاک‌ها می‌باشد (همچون if و else).

داشتن وابستگی کنترلی از هر خط به بلاک‌ها و دستورات آن‌ها بستگی به این دارد که آن خط کد در زیرمجموعه حوزه آن بلاک‌ها و دستوراتشان حضور داشته باشد. برای مثال خط ۱۵ هیچ وابستگی کنترلی‌ای به خطوط ۶، ۷ و ۹ ندارد.

ایجاد وابستگی‌های داده‌ای به این شرح است: اگر در یک خط از کد یک متغیر به کار رفته باشد که آن متغیر پیش از آن در جایی (در سطح بالاتری از حوزه بلاکی آن خط کد) مقداردهی شده باشد، یک وابستگی داده‌ای به آن متغیر ایجاد می‌شود. برای مثال خط ۱۵ سه وابستگی به خطوط ۳ و ۴ و ۱۲ دارد و خط ۱۲ نیز دو وابستگی داده‌ای به خط ۴ و ۵ دارد (رسم نشده است).

نکته اینجاست که هیچگونه وابستگی داده‌ای بـه خـط ۸ کـه متغیـر a در آن مقداردهی شده است وجود ندارد که این به دلیل عدم قرار گیری خطوط ۱۵ و ۸ در زیر مجموعه حوزه بلاکی یکدیگر است. بنابراین می‌تـوان نتیجـه‌گیـری کرد که دو شرط موجود در خطوط ۶ و ۱۰ زیرمجموعه بلاکی یکدیگر نیستند و بنابراین دستورات موجود در این دو شرط نمـی‌تواننـد هـیچ‌گونـه وابسـتگی کنترلی و داده‌ای به یکدیگر داشته باشند.

یکی دیگر از دستوراتی که در مرحله آنالیز لغوی در نظر گرفته مـی‌شـود دستورات Printf و Scanf موجود در زبان C است. همـانطور کـه مشـخص است، این دستورات در حقیقت توابعی هستند که می‌تواننـد متغیرهـایی را بـه عنوان ورودی خود بگیرند. برای آنکه الگـوریتم بتوانـد PDG دقیقـی داشـته باشد، می‌بایستی برای این دستورات نیز مقادیر وابسـتگی داده‌ای و کنترلـی را در نظر بگیرد. بنابراین در هنگام برخورد با این دستورات ابتدا بررسی می‌شـود که آیا متغیرهای کد منبع را به عنوان ورودی می‌گیرند یا خیر که اگـر ورودی دارند، وابستگی داده‌ای و کنترلی خطـوط شـامل ایـن دسـتورات نیـز هماننـد عبارات شرطی و یا محاسباتی مقداردهی می‌شود. هر چند در اینجا تنها به این دو دستور توجه شده است، اما توجه به نکته یاد شـده مـی‌توانـد در دسـتورات بسیاری به کار گرفته شود که نتایج خوبی را در بر خواهد داشت.

۳-۲-۶- نرمال‌سازی ثانویه

پس از اتمام آنالیز لغوی، تمام خطوط کد که وابستگی داده‌ای و کنترلی آن‌هـا مشخص شده است، آماده‌ی تولید کد هش می‌باشند. برای ایـن کـار، خطـوط کدی که در ابتدای کار، نرمال‌سازی اولیه بر روی آن‌ها انجـام شـده بـود، بـه مرحله نرمال‌سازی ثانویه ارسال می‌شوند. علت این که نرمال‌سازی ثانویه پس از آنالیز لغوی قرار گرفته، این است که در این نرمال‌سـازی بـه نتـایج مرحلـه آنالیز لغوی احتیاج است. نتایجی همچون متغیرهای تعریف شده در کد منبع.

در نرمال‌سازی ثانویه، هر خط از کد با توجه به آنکـه شـامل دسـتورات شرطی، تعریف متغیر، عبارت محاسباتی و یا دستورات ورودی و خروجی است به طور مجزا بررسی می‌شود. اگر خط موردنظر شامل دستور ورودی و خروجی باشد، پارامترهای ورودی آن حذف می‌شود (مانند شکل (۱۲))

```
printf("an example of printf x = %d, y = %d, z = %d", x, y, z);  ⟶  printf("",,,);
printf("an example of printf a = %d, b = %d, c = %d ", a, b, c);  ⟶  printf("",,,);
```

شکل (۱۲)

همانطور که مشاهده مـی‌شـود، هـر دو دسـتور printf، پـس از حـذف پارامترهایشان به یک شکل درآمده‌اند. این امر یک مزیت محسوب مـی‌شـود که در ادامه توضیح داده می‌شود.

اگر خط کد مورد نظر شامل دستورات شـرطی و یـا عبـارت محاسـباتی باشد، یک عملیات فیلترسازی برروی آن انجام می‌گیـرد کـه طـی آن اسـامی متغیرها از گزاره‌های شرطی حذف می‌شود. (شکل (۱۳))

```
if(((a + 24) * b) > 8 && (a * b) < 90)  ⟶  if(((+)*)>&&(*)<)
swap = (24 * b * 4) / b;  ⟶  =(**)/;
```

شکل (۱۳)

اگر خط شامل دستور تعریف متغیر باشد، آن خط حذف می‌شـود (شـکل (۱۴))

```
float X;  ⟶  حذف
int Y, Z;  ⟶  حذف
```

شکل (۱۴)

در انتهای این مرحله، هر خط از کد به یک کد هش تبدیل می‌شود و در صفت LineHashValue مربوط بـه آن خـط در لیسـت LineHashList ذخیره می‌شود. باید به این نکته توجه کرد که ایـن تمیزکـاری‌هـای پـیش از ساختن کد هش مزیت‌های بسیاری دارد: اول اینکه در دو تکه کد مختلف که عملکرد یکسانی دارند و اسامی متغیرها در آن‌ها متفاوت است، بتوان کـدهای مشابه را پیدا کرد. برای مثال به دو تکه کد در شکل (۱۵) و دستور printfای که در چند پاراگراف قبل نشان داده شده است توجه کنید

```
if(((a + 24) * b) > 8 && (a * b) < 90)  ⟶  if(((+)*)>&&(*)<)
if(((x + y) * z) > 8 && (x * y) < 90)  ⟶  if(((+)*)>&&(*)<)
```

شکل (۱۵)

دوم اینکـه بـه دلیـل حـذف کاراکترهـای فضـای خـالی (space) از عبارت‌های محاسباتی و ...، کد هش تولید شده دقت بیشتری خواهـد داشـت. برای روشن شدن موضوع به دو تکه کد شکل(۱۶) توجه کنید

```
swap = (24 * b * 4) / b;  ⟶  =(**)/; => GetHashCode => 1684170598
swap = (24    * b *    4) / b;  ⟶  =(      **    )/; => GetHashCode => 340924639
```

شکل (۱۶)

همانطور که مشاهده می‌شود مقادیر کد هـش ایـن دو عبـارت یکسـان نیستند و این به این معنی است که فضاهای خالی در کد هش تولید شده تاثیر بسیاری دارند و بنابراین حذف آن‌ها دقت را افزایش خواهد داد.

پس از انجام مراحلی که شرح داده شد، همه چیز آماده‌ی آن اسـت کـه حافظه‌ی انتزاعی متغیرها با یکـدیگر و گـراف وابسـتگی خطـوط کـد نیـز بـا یکدیگر مقایسه شوند.

۳-۲-۷- مقایسه حافظه انتزاعی متغیرهای دو کد منبع

در این مرحله می‌بایستی حافظه‌ی انتزاعی متغیرها در دو کد منبع بـا یکـدیگر مقایسه شوند. ابتدا متغیرهای دو کد منبع به صورت دوبه‌دو به مرحله مقایسـه حافظه انتزاعی فرستاده مـی‌شـوند کـه CompareAMS نامیـده مـی‌شـود. خروجی این مرحله یک مقدار شباهت است. اگر این مقدار از حد معینی بـالاتر باشد، تمام خطوطی که در صفت LineNumbers هـر متغیـر ذخیـره شـده است، در یک فایل متنی (برای هر متغیر جداگانه) در خروجی چاپ مـی‌شـود. این خروجی به تفکیک پوشه‌ها (Folders) (که نشان دهنده گروه‌های کلونی متفاوت هستند) و در فایل‌هایی مجزا، قرار می‌گیرد. به این پوشـه‌هـا پیشـوند Clone به همراه یک اندیس تعلق می‌گیرد تا گـروه‌هـای کلـونی مختلـف از یکدیگر متمایز شوند. ضمنا به هر فایل متنی نیز یک اندیس اختصاص میابد تا کلون مربوط به هر کد منبع از دیگر کلون‌های درون آن گروه متمایز شود.

در مرحله CompareAMS هر دو گزاره مربوط به دو متغیر به صورت همزمان و همروند بررسی می‌شوند. با توجه به ساختار گـزاره‌هـا، ابتـدا شـرط آن‌هـا بررسـی مـی‌شـود. ایـن بررسـی، خـود توسـط مرحلـه‌ای بـه نـام ComparePredicate انجام می‌شود که در ادامه شرح داده خواهد شد. اگر شرط‌ها برابر بود، عبارت‌های محاسباتی گزاره‌ها بررسی می‌شود. همانطور کـه مشخص است خود عبارت محاسباتی نیز می‌تواند شامل گزاره‌هایی دیگر باشد که آن‌ها نیز باید به صورت جداگانه‌ای بررسی شـوند. ایـن عمـل بـه صـورت بازگشتی انجام می‌شـود و در هـر بازگشـت، مقـدار شـباهت از متـد مربوطـه بازمی‌گردد (return می‌شود).

مقایسه‌ی عبارت‌های محاسباتی نیز توسط مرحله‌ای به نام CompareStatements انجام می‌شود. در این مرحله ابتدا به ازای تمام متغیرهای تعریف شده در هر کد، مقادیری تصادفی از انواع مختلف متغیرها تولید می‌شود. سپس این مقادیر به همراه دستورات مربوطه به الگوریتمی که نام آن StatementsExecutionLooping است فرستاده می‌شوند. در این الگوریتم به ازای تمام ترکیب‌های ممکنی که متغیرها می‌توانند با یکدیگر داشته باشند، متد StatementsExcution فراخوانی می‌شود. این عملیات به صورت بازگشتی انجام می‌شود. در متد StatementsExcution هر کدام از دستورات بررسی می‌شوند و متغیرهای آن‌ها شناسایی می‌شوند. سپس با توجه به نوع هر متغیر یکی از مقادیر تصادفی انتخاب و به متغیر مربوط به آن نوع اختصاص میابد. در ادامه، هر دستور با مقادیر جایگزین شده اجرا می‌شود و نتیجه اجرای دو دستور با یکدیگر مقایسه و از متدها و مراحل نام برده بازگشت داده می‌شود تا در نهایت به مرحله CompareStatements باز گردد. برای مثال به دو عبارت موجود در شکل (۱۷) توجه کنید.

Statement1 ⟶ swap = (24 * A * B) / C;

Statement1 ⟶ swap = (24 * B * A) / F;

شکل (۱۷)

در این دو عبارت چهار متغیر (F، C، B و A) به کار رفته‌اند که برای محاسبه عبارت منتسب به متغیر swap می‌بایستی با مقادیر تصادفی مقداردهی شوند. متغیرهای این دو عبارت ممکن است نام‌های یکسانی داشته و یا نداشته باشند. صرفا از روی نام متغیرها نمی‌توان تصمیم گرفت که به متغیرهای هم‌نام مقادیر تصادفی یکسان داد، چرا که این معیار نمی‌تواند قطعا تناظر متغیرها را بیان کند و بنابراین می‌بایستی تمام ترتیب‌های ممکن برای یافتن تناظر صحیح میان متغیرها در دو عبارت را آزمایش و اجرا نمود. در این مثال تناظر متغیرها به صورت شکل (۱۸) است.

A در عبارت اول ⟶ B در عبارت دوم

B در عبارت اول ⟶ A در عبارت دوم

C در عبارت اول ⟶ F در عبارت دوم

شکل (۱۸)

وقتی اجرا و مقایسه دو دستور به اتمام می‌رسد، نتیجه به مرحله CompareStatements باز می‌گردد. از آنجاییکه CompareStatements هم به صورت بازگشتی پیاده‌سازی شده است، نتیجه با دستور return بازگشت داده می‌شود تا در نهایت به عنوان مقدار شباهت (similarity) به کار گرفته شود.

مرحله ComparePredicate نیز تقریبا همانند CompareStatements است، با این تفاوت که در آن مقادیر تصادفی بیشتری مورد بررسی و اجرا قرار می‌گیرند. علت آن این است که شرط‌ها در نهایت به مقادیر True و False ترجمه می‌شوند، این در حالی است که مقادیر تصادفی بسیاری هستند که تغییری در نتیجه شرط‌ها ایجاد نمی‌کنند. برای مثال دو شرط x>100 و x>200 یکسان نیستند، در حالی که تمام اعداد تصادفی بالای ۲۰۰، نتیجه این دو شرط را True می‌کنند. با این وضعیت باید آنقدر اعداد تصادفی تولید شود که حداقل یکی از آن‌ها بین ۱۰۰ تا ۲۰۰ باشد تا عدم برابری این دو شرط اثبات شود. شاید تا به حال چندین روش برای یافتن برابری شرط‌ها ارائه شده باشد، اما در این میان روشی از سوی محققان دانشگاه استنفورد ارائه شده که پیش از این مورد استفاده قرار گرفته است. در این روش ابتدا هر عبارت شرطی ساده سازی می‌شود. سپس این عبارت‌های ساده شده با یکدیگر مقایسه می‌شوند. هر چند این روش به صورت ثابت (Static) این کار را انجام می‌دهد و در نتیجه سرعت بیشتری دارد، اما همیشه ساده‌سازی‌ها نتیجه صحیحی ندارند. برای مثال با استفاده از این روش دو عبارت (a^b)∨(a^c) و (a^(b∨c)) در فرم ساده شده خودشان قرار دارند و همانطور که مشاهده می‌شود کاملا از لحاظ متنی متفاوت هستند، در حالی که به لحاظ معنایی هر دو یکسان هستند. این یک ضعف برای این روش تلقی می‌شود. پس از قدری تامل در خصوص این مساله و سایر مسائل، برای این مسئله راه‌حلی یافته شد که در آینده مورد استفاده قرار خواهد گرفت. در این روش ابتدا به جای اعداد ثابت یک شناسه مشخص قرار داده می‌شود. یعنی x>100 و x>1000 هر دو به عبارت x>id تبدیل خواهند شد. سپس هر شرط به صورت یک معادله تصور می‌شود و شروع به حل این معادلات با هدف یافتن هر کدام از متغیرهای آن‌ها می‌شود. برای مثال شرط x^2+2*x+y=0 دو نتیجه در شکل (۱۹) را در بر خواهد داشت (یکی برای x و یکی برای y)

$$x = - \sqrt{1-y} - 1,\ x = \sqrt{1-y} - 1$$

$$y = - x^2 - 2 * x$$

شکل (۱۹)

البته در اینجا برای وضوح بیشتر، مقادیر ثابت با شناسه id جایگزین نشده است. در اینجا دو متغیر x و y وجود دارد که می‌بایستی معادله را برای هر دوی آن‌ها حل کرد. در نهایت کافیست متغیرهای هر شرط را بدون توجه به نام آن‌ها با یکدیگر مقایسه کرد. در صورت برابری مقدار متغیرها، شرط‌ها برابر خواهند بود. باید به این نکته توجه داشته باشید که هر شرط خود می‌تواند شامل چندین شرط باشد که در اینجا به توضیح یک شرط واحد پرداخته شده است. برای مثال دو شرط (a^b)∨(a^c) و (b^c)∨(b^a) را در نظر بگیرید و فرض کنید هر کدام از a و b و c، خود شرط‌هایی دیگر هستند. بنابراین این دو شرط دارای چند شرط داخلی است که در دو شرط اصلی، به صورت جابه‌جا قرار گرفته‌اند. هر دو شرط با یکدیگر برابرند، در حالی که شرط‌هایشان با یکدیگر جابه‌جا هستند. برای حل این مساله، درختی برای شرط اصلی و انشعاب‌هایی در آن برای شرط‌های فرعی ایجاد می‌شود و نتایج شرط‌های فرعی با یکدیگر مقایسه می‌شود تا اثر جابه‌جایی شرط‌ها از بین برود.

۳-۲-۸- مقایسه گراف وابستگی خطوط دو کد منبع

در این مرحله مقادیر کد هش تمام خطوط در هر دو کد منبع با یکدیگر مقایسه می‌شوند، اگر دو کد هش با یکدیگر یکسان بودند، آنگاه وابستگی‌های کنترلی و داده‌ای دو خط کد با یکدیگر مقایسه می‌شوند. آنقدر این کار به صورت بازگشتی ادامه میابد که یا خطوط کد تمام شود و یا وابستگی کنترلی یا داده‌ای برابری میان انشعاب‌های خط اصلی مورد مقایسه وجود نداشته باشد.

در نتیجه مقایسه دو کد هش، یک سری خطوط از هر دو کد منبع انتخاب می‌شوند که وابستگی کنترلی و داده‌ای آن‌ها یکسان است. سپس این خطوط به یک متد به نام WritePDGsIntersect ارسال می‌شوند. این متد، خطوط مورد نظر را از دو کد منبع استخراج کرده و در پوشه‌هایی جداگانه و فایل‌هایی درون آن (به صورت مجزا)، کدهای مشابه را چاپ می‌کند. به این پوشه‌ها پیشوند PDG به همراه یک اندیس تعلق می‌گیرد تا گروه‌های کلونی مختلف از یکدیگر متمایز شوند. ضمنا به هر فایل متنی نیز یک اندیس

اختصاص میابد تا کلون مربوط به هر کد منبع از دیگر کلون‌های درون آن گروه متمایز گردد.

٤- ارزیابی

این کار پژوهشی در قالب ابزاری به نام DSCCD (Dynamic & Static Code Clone Detector) پیاده سازی شده است. در اینجا، DSCCD با MeCC [۱۵] بر اساس معیار تعداد کلون‌های استخراج شده مقایسه شده‌اند. همچنین زمان استخراج کلون ها برای DSCCD در جدول (۲) مشاهده می‌شود که در آن S نماد ثانیه است. متاسفانه به دلیل آن‌که آنالیزور ، تعداد محدودی از دستورات زبان C را تشخیص می‌دهد، برای انجام آزمایش، نتوانستیم از ورودی‌های ارائه شده در مراجع استفاده کنیم و به جای آن، تعدادی برنامه را به عنوان نمونه ورودی ایجاد کرده و مورد مقایسه قرار داده‌ایم. در جدول (۱)، هر مثال به همراه نتایج حاصل از اجرای روش‌های MeCC [۱۵] و DSCCD مشاهده می‌شود.

باید اشاره نمود که در این آزمایشات، معیار شباهت نزدیک به ۱۰۰ در نظر گرفته شده است.

جدول (۱) : تعداد کلون‌های یافت شده توسط دو روش MeCC و DSCCD

	MeCC	DSCCD	تعداد کلون های واقعی
مثال ۱	۱	۳	۶
مثال ۲	۲	۵	۸
مثال ۳	۱	۲	۴
مثال ۴	۱	۱۰	۱۲

همانطور که در جدول بالا مشخص شده است، روش MeCC [۱۵] تنها توانسته است در حدود ۱۶ درصد از کلون‌ها را شناسایی کند در حالی که DSCCD توانسته در حدود ۶۶ درصد کلون‌ها را شناسایی کند. در برخی از این مثال‌ها، دستورات حلقه مانند For و While قرار داده بودیم که الگوریتم ارائه شده، توانایی تشخصی آن‌ها را نداشت که بخشی از کاهش درصد تشخیص مربوط به این دستورات است. هدف در این کار پژوهشی کاهش False Negative بود که این هدف در این نتایج تحقق یافته است. البته باید توجه داشت که تکه کدهای ورودی نسبتا ساده بوده‌اند و دارای کدهایی در حد مثال‌های مطرح شده در طول مقاله بوده است.

این آزمایشات بر روی یک کامپیوتر با CPU CORE i7 Intel و ۸ Gig Ram انجام شده است. زمان اجرای DSCCD برروی این نمونه کدها در جدول زیر قابل مشاهده است.

جدول (2) : زمان اجرای نمونه کدها برروی DSCCD

	DSCCD
مثال ۱	S۱.۱
مثال ۲	S۱.۷
مثال ۳	S۰.۸
مثال ۴	S۴۲۶

علاوه بر مثال‌های فوق کدهای دیگری نیز آزمایش شده است. نتایج نشان می‌دهد که هر چه حجم کد افزایش میابد، زمان اجرای آن نیز بیشتر می‌شود. همچنین ترتیب تعریف متغیرها در دو تکه کد بر روی سرعت اجرا تاثیرگذار است. علت آن این است که برای تشخیص برابری شرطها و عبارات محاسباتی بخشی از آن‌ها اجرا می‌شود. هر چه ترتیب تعریف متغیرها در دو تکه کد متفاوت‌تر باشد، ابزار نیز می‌بایستی اجراهای بیشتری را داشته باشد تا به نتیجه مطلوب برسد.

٥- جمع بندی و کارهای آتی

یکی از کارهایی که می‌توان در الگوریتم PDG انجام داد این است که، عمل مقایسه شرطها براساس کد هش انجام نشود و در عوض از روش مقایسه شرطها استفاده شود. این عمل موجب افزایش دقت روش PDG خواهد شد. همچنین می‌توان فایل‌های ورودی و حتی دستورات ورودی را براساس تعداد متغیرها، تعداد توابع، نوع آن‌ها، تعداد خطوط و ... دسته‌بندی نمود، سپس در میان هرگروه به یافتن کدهای مشابه پرداخت. این کار باعث افزایش سرعت در یافتن کدهای مشابه می‌شود.

از طرفی دیگر اگر بتوان موازی سازی را در اجرای ابزار به کار گرفت، می‌توان سرعت اجرای ابزار را افزایش داد. برای انجام این کار می‌بایستی از Hadoop و یا کتابخانه‌هایی همچون CloudSim که توانایی اجرای موازی را در اختیار ما قرار می‌دهند استفاده کرد.

یکی از محدودیت‌های موجود در DSCCD، عدم پشتیبانی از دستورات حلقه است. راه‌حلی که پیش از این برای کنترل چنین دستوراتی ارائه شده بود، اجرای آن دستورات با مقادیر تصادفی بسیار زیاد بوده است که مشکل آن هزینه بسیار بالای اجرا است که در برخی موارد روزها به طول انجامیده است [۱۴]. باید دقت شود که در این پژوهش نیز از روش اجرای عبارت‌ها با مقادیر تصادفی استفاده شده است، اما به دلیل نداشتن دستورات حلقه سرعت اجرا مناسب بوده است. به هر حال باید توجه داشت که زمان عامل مهمی است، چرا که برنامه‌نویسان علاقه‌ای به این ندارند که امروز متد تشخیص کلون را اجرا کنند و هفته آینده جواب آن را مشاهده کنند. بنابراین راه حل‌هایی نیاز است که در زمان بسیار کم، کلون‌ها را تشخصی دهند.

مراجع

[1] Roy, C.K., Cordy, J.R. and Koschke, R., 2009. *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming*, 74(7), pp.470-495.Sannella, M. J., Constraint Satisfaction and Debugging for Interactive User Interfaces, Ph.D. Thesis, University of Washington, Seattle, WA, 1994

[2] Baker, B.S., 1995, July. *On finding duplication and near-duplication in large software systems*. In Reverse Engineering, 1995., Proceedings of 2nd Working Conference on (pp. 86-95). IEEE

[3] Baxter, I.D., Yahin, A., Moura, L., Anna, M.S. and Bier, L., 1998, November. *Clone detection using abstract syntax trees*. In Software Maintenance, 1998. Proceedings., International Conference on (pp. 368-377). IEEE

[4] Kapser, C.J. and Godfrey, M.W., 2006. *Supporting the analysis of clones in software systems*. Journal of Software Maintenance and Evolution: Research and Practice, 18(2), pp.61-82

[5] Mayrand, J., Leblanc, C. and Merlo, E.M., 1996, November. *Experiment on the automatic detection of function clones in a software system using metrics*. In Software Maintenance 1996, Proceedings., International Conference on (pp. 244-253). IEEE

Token-Based Methods. Journal of Software Engineering and Applications, 10(13), p.891

[23] Jiang, L., Misherghi, G., Su, Z. and Glondu, S., 2007, May. *Deckard: Scalable and accurate tree-based detection of code clones*. In Proceedings of the 29th international conference on Software Engineering (pp. 96-105). IEEE Computer Society

[24] Murakami, H., Hotta, K., Higo, Y., Igaki, H. and Kusumoto, S., 2013, May. *Gapped code clone detection with lightweight source code analysis*. In Program Comprehension (ICPC), 2013 IEEE 21st International Conference on (pp. 93-102). IEEE

[25] Hummel, B., Juergens, E., Heinemann, L. and Conradt, M., 2010, September. *Index-based code clone detection: incremental, distributed, scalable*. In Software Maintenance (ICSM), 2010 IEEE International Conference on (pp. 1-9). IEEE

[26] Sajnani, H., Saini, V. and Lopes, C., 2015. *A parallel and efficient approach to large scale clone detection*. Journal of Software: Evolution and Process, 27(6), pp.402-429

[27] Krutz, D.E. and Shihab, E., 2013, October. *Cccd: Concolic code clone detection*. In 2013 20th Working Conference on Reverse Engineering (WCRE) (pp. 489-490). IEEE

[28] Stankov, E., Jovanov, M. and Bogdanova, A.M., 2013, June. *Source code similarity detection by using data mining methods*. In Information Technology Interfaces (ITI), Proceedings of the ITI 2013 35th International Conference on (pp. 257-262). IEEE

[29] Kaur, R. and Singh, S., 2014. *Clone detection in software source code using operational similarity of statements*. ACM SIGSOFT Software Engineering Notes, 39(3), pp.1-5

[30] Dang, Y., Zhang, D., Ge, S., Chu, C., Qiu, Y. and Xie, T., 2012, December. *XIAO: tuning code clones at hands of engineers in practice*. In Proceedings of the 28th Annual Computer Security Applications Conference (pp. 369-378). ACM

[31] Sheneamer, A. and Kalita, J., 2015, December. *Code clone detection using coarse and fine-grained hybrid approaches*. In Intelligent Computing and Information Systems (ICICIS), 2015 IEEE Seventh International Conference on (pp. 472-480). IEEE.

[6] Roy, C.K. and Cordy, J.R., 2008, October. *An empirical study of function clones in open source software*. In Reverse Engineering, 2008. WCRE'08. 15th Working Conference on (pp. 81-90). IEEE

[7] Ducasse, S., Rieger, M. and Demeyer, S., 1999. *A language independent approach for detecting duplicated code*. In Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on (pp. 109-118). IEEE

[8] Roy, C.K., 2009, September. *Detection and analysis of near-miss software clones*. In Software Maintenance, 2009. ICSM 2009. IEEE International Conference on (pp. 447-450). IEEE

[9] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E., 2007. *Comparison and evaluation of clone detection tools*. Software Engineering, IEEE Transactions on, 33(9), pp.577-591

[10] Roy, C.K. and Cordy, J.R., 2007. *A survey on software clone detection research*. Technical Report 541, Queen's University at Kingston

[11] Li, J. and Ernst, M.D., 2012, June. *CBCD: Cloned buggy code detector*. In Proceedings of the 34th International Conference on Software Engineering (pp. 310-320). IEEE Press

[12] Krinke, J., 2001. *Identifying similar code with program dependence graphs*. In Reverse Engineering, 2001. Proceedings. Eighth Working Conference on (pp. 301-309). IEEE

[13] Milea, N.A., Jiang, L. and Khoo, S.C., 2014, July. *Scalable detection of missed cross-function refactorings.* In Proceedings of the 2014 International Symposium on Software Testing and Analysis (pp. 138-148). ACM

[14] Jiang, L. and Su, Z., 2009, July. *Automatic mining of functionally equivalent code fragments via random testing*. In Proceedings of the eighteenth international symposium on Software testing and analysis (pp. 81-92). ACM

[15] Kim, H., Jung, Y., Kim, S. and Yi, K., 2011, May. *MeCC: memory comparison-based clone detector*. In Software Engineering (ICSE), 2011 33rd International Conference on (pp. 301-310). IEEE

[16] Roy, C.K. and Cordy, J.R., 2008, June. *Scenario-based comparison of clone detection techniques*. In Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on (pp. 153-162). IEEE

[17] Roy, C.K. and Cordy, J.R., 2007. *A survey on software clone detection research*. Technical Report 541, Queen's University at Kingston

[18] Kodhai, E. and Kanmani, S., 2014. *Method-level code clone detection through LWH (Light Weight Hybrid) approach.* Journal of Software Engineering Research and Development, 2(1), p.12

[19] Abd-El-Hafiz, S.K., 2012, July. *A metrics-based data mining approach for software clone detection*. In Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual (pp. 35-41). IEEE

[20] Alrabaee, S., Shirani, P., Wang, L. and Debbabi, M., 2015. *Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code.* Digital Investigation, 12, pp.S61-S71

[21] Toomey, W., 2012, June. *Ctcompare: Code clone detection using hashed token sequences*. In Software Clones (IWSC), 2012 6th International Workshop on (pp. 92-93). IEEE

[22] Ami, R. and Haga, H., 2017. *Code Clone Detection Method Based on the Combination of Tree-Based and*