Uma análise de grafos de fluxo como abordagem para o ensino de teste de software

Eva Maia Malta[1], Philiphe A. R. Kramer[2], Sérgio Fred Ribeiro Andrade[3]

Ciência da Computação - Departamento de Ciências Exatas e Tecnológica -
Universidade Estadual de Santa Cruz (UESC)
45662-999 – Ilhéus – BA – Brasil

emaiamalta@gmail.com[1], pharkrum@gmail.com[2], sergiof@uesc.br[3]

Abstract. *This work analyzes the association between the cyclomatic complexity and the number of independent paths, which are related in didactic literature, for software structural testing. The tests were performed in a simple program with several input scenarios to compare the approach here called Traditional Path Approach that considers generic inputs and another, used in this work called Calculated Path Approach that also considers global and referenced variables. A tool was developed for didactic purposes and to show the differences between these approaches. The result evidenced the divergence between the approaches in the number of independent paths, which makes it possible to induce errors in the indication of software test cases.*

Resumo. *Este trabalho analisa a associação entre a complexidade ciclomática e a quantidade de caminhos independentes, que são relacionados na literatura didática, para teste estrutural de software. Foram realizados ensaios num programa simples com diversos cenários de entrada com objetivo da comparação entre a abordagem aqui chamada de Abordagem do Caminho Tradicional que considera entradas genéricas e outra, utilizada neste trabalho denominada de Abordagem do Caminho Calculado a qual considera também variáveis globais e referenciadas. Foi desenvolvida uma ferramenta para fins didáticos e mostrar as diferenças entre essas abordagens. O resultado evidenciou a divergência entre as abordagens na quantidade de caminhos independentes, o que possibilita induzir a erros na indicação de casos de teste de software.*

1. Introdução

As atividades de teste de software buscam garantir a qualidade do software. Segundo Pressman (2016), a qualidade do software é inerente à exigência dos requisitos funcionais, das especificações explicitamente descritas e do desempenho estabelecidos.

Um motivo cabível para aplicação de testes de software está indicado por Howden (1976), o qual observa que no processo de construção de software os erros na maioria das vezes são humanos. Apesar do uso de métodos consistentes, ferramentas de suporte apropriadas e profissionais treinados, as falhas acontecem e prejudicam a qualidade dos produtos. Essas falhas acometem, geralmente, na parte do código pouco executada.

Os testes de software mais utilizados são o teste de sistema, teste estrutural e teste de unidade, segundo Sommerville (2007). Em particular, o teste estrutural pressupõe conhecimento da implementação do software, onde são projetados casos de teste para validação das ações estruturais do código representadas por recursos de decisão e repetição no controle sequencial do fluxo.

Duas técnicas importantes de teste estrutural são fundamentadas na teoria dos grafos. A complexidade ciclomática e o teste de caminho independente, ambos descritos em McCabe (1976). O primeiro resulta na quantidade de fluxos possíveis do programa, conta-se todas as estruturas condicionais e soma-se a uma unidade, como também, para o mesmo resultado, conta-se a quantidade de nós e subtrai-se da quantidade de arestas mais duas unidades. O segundo teste parte-se do grafo orientado e destaca-se todos os fluxos de caminhos possíveis, chamados caminhos independentes. Cada fluxo deve incluir pelo menos uma nova aresta que não tinha sido percorrida por outros caminhos antes destacados, ou seja, não deve ser uma combinação de caminhos já atravessados (Pressman, 2016).

Desta forma, conhecendo a quantidade de caminhos independentes e quais são seus fluxos pode-se operacionalizar os testes efetivos na busca de erros ou falhas no programa (McCabe, 1976).

Na literatura encontra-se uma vinculação entre os dois testes citados como complemento, aqui chamada de Abordagem do Caminho Tradicional (ACT). Em Sommerville (2008) é dito que após descobrir o número de caminhos independentes pela complexidade ciclomática projeta-se os casos de teste para esses caminhos, onde o mínimo de casos de teste necessário para todos os caminhos é igual à complexidade ciclomática. Em Pressman (2016), é mencionado que o valor computado para a complexidade ciclomática define o número de caminhos independentes do programa, fornecendo um limite superior para a quantidade de testes. Da mesma forma, essa ideia é encontrada em Howden (1976), Herman (1976), Januz e Korel (1983) e Boris (2003).

Em Maldonado (1991), citando Weyuker (1984), Clarke (1985) e Raps (1985), foi considerada que "a complexidade de um critério *C* é definida como o número máximo de casos de teste requerido pelo critério no pior caso, ou seja, dado um programa qualquer *P*, se existir um conjunto de casos de teste *T* que seja *C* – adequado para *P*, então existe um conjunto de casos de teste T_1, tal que a cardinalidade de T_1 é menor ou igual à complexidade do critério *C*."

Essas afirmações são levadas para o processo ensino-aprendizagem como assertivas, sem direcionar alguma alternativa para conhecer o número de casos de teste necessários ou real limite superior de fluxos no grafo, possíveis de execução ou não, diante de diversas composições de atribuição de variáveis, inicializadas no código ou alocadas em tempo de execução.

Sendo assim, o objetivo deste trabalho foi analisar grafos de fluxo para emprego do cálculo da complexidade ciclomática em diversos cenários, e verificar se realmente são indicativos da quantidade de caminhos independentes, para comprovar as afirmações na literatura didática citadas anteriormente. Ou seja, questionar a abordagem que relaciona os testes estruturais em potenciais aos caminhos independentes

executáveis indicados pela complexidade ciclomática, para dirimir dúvidas no processo ensino-aprendizagem.

Para tanto, foi considerada a família de Critérios Potenciais em Uso que estabelece considerar critérios para os fluxos e caminhos executáveis e a Família de Critérios de Fluxo de Dados que estabelece a presença de todos os caminhos e ramos, inclusive não executáveis (Frankl, 1987).

Diante dessa premissa um algoritmo para resolver um problema simples de comparação, contendo erro semântico, foi estudado durante as aulas de engenharia de software, onde buscou-se analisar se, genericamente, através do cálculo da complexidade ciclomática chega-se aos caminhos independentes de um grafo de fluxo de controle, com o emprego de variáveis alocadas no código globalmente, considerando ausência de caminho livre de definição (Maldonado, 1991), e, com o emprego de variáveis sem inicialização com alocação em tempo de execução. Aqui denominada de Abordagem do Caminho Calculado (ACC).

Para os ensaios, foi desenvolvida uma ferramenta de propósito didático no objetivo de auxiliar no conhecimento do processo de teste estrutural e no entendimento da vinculação entre as técnicas de complexidade ciclomática e caminhos independentes, aplicando-se recursos mais simples possíveis para as operações de reconhecimento de linguagem formal.

2. Materiais e Métodos

Um programa pode ser representado como um fluxograma de algoritmo que pode ser convertido num grafo orientado $G = (V, E, n, k)$. Onde, cada nó V representa um procedimento (declaração, atribuição, operação aritmética e afins) e uma decisão (controle de fluxo para seleção ou repetição), que são associados através de arestas E, que é um par (n, m) de V que representa sequência de controle de n para m. O grafo resultante do programa é um grafo orientado com um único nó de entrada $n \in V$ e um único nó de saída $k \in V$. Um caminho ou fluxo no grafo é uma sequência de nós ($n_1, n_2, n_3, ..., n_k$), sendo $k \geq 2$.

Um caminho simples é um fluxo que contém todos os nós com exceção de n_1 e último n_k, sem possibilidade de retorno, sendo todos os nós distintos, também chamado de caminho livre de laço. Um caminho completo contém n_1 na entrada e n_k como saída. Um caminho independente é um fluxo completo que contém uma nova aresta não percorrida anteriormente e não composta por combinação de fluxos antes visitados (McCabe, 1976).

Em um nó que contém um fluxo condicional, seleção ou repetição, é chamado de nó predicado (P) caracterizado por duas ou mais arestas de saídas. O nó P é aplicado para o cálculo do teste da complexidade ciclomática através de eq. 1.

$$V(G) = P + 1 \qquad (1)$$

Da mesma forma, pode-se conhecer o cálculo ciclomático $V(G)$ pela quantidade de nós e arestas visitadas em todos os caminhos do programa, dada pela eq. 2.

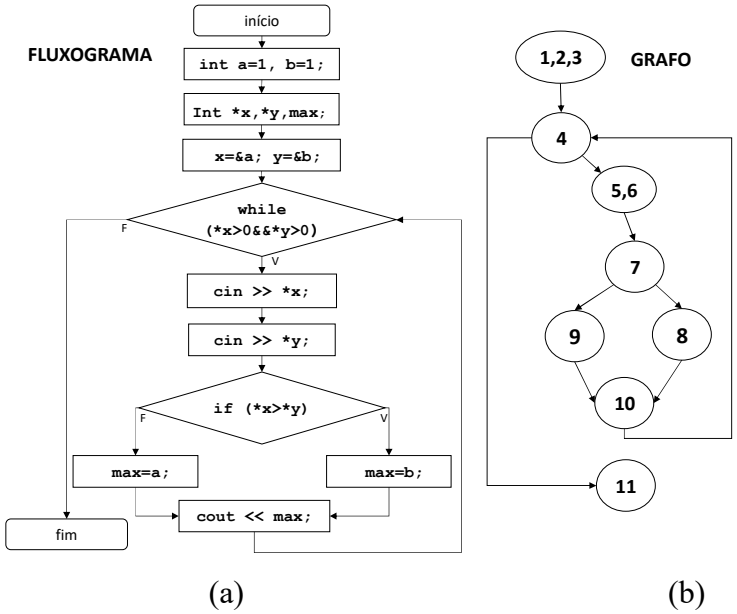
$$V(G) = V - E + 2 \qquad (2)$$

O resultado *V(G)* denota a quantidade de casos de teste que devem ser realizados no limite superior para o número de caminhos independentes do programa (Pressman, 2016).

Um caminho completo é executável quando existem variáveis de entrada definidas, ou seja, inicializadas no código ou de *input* externo. Observa-se a possibilidade de não haver nenhuma entrada para uma variável numa sequência de procedimentos em um determinado fluxo do grafo, fazendo com que o caminho seja não executável, essa situação é conhecida como caminho livre de definição (Frankl, 1987, Rapps e Weyuker, 1985). Para nós do grafo que possuem variáveis globais definidas, existe alocação da variável e ocorre um caminho executável. Para variáveis alocadas em tempo de execução sempre ocorre definição e um caminho executável.

Com base nesses princípios, para ACC, foi empregado um algoritmo simples que recebe dois inteiros, compara-os e resulta no máximo valor entre eles. Esse algoritmo do programa considerado está ilustrado no fluxograma da Figura 1(a) e no grafo orientado correspondente da Figura 1(b).

Na execução do programa, os dois inteiros são inicializados no código e enviados a um procedimento para comparação e impressão de saída. A escolha do algoritmo é justificada pelo uso das variáveis globais inicializadas que forçam sempre o mesmo comportamento inicial, entrar no laço de repetição ou não, dependendo da definição, possibilitando divergências na quantidade de caminhos independentes.



(a) (b)

Figura 1. (a) fluxograma do programa estudado; (b) grafo correspondente ao fluxograma.

O programa foi apresentado em cinco cenários quanto à definição e alocação de variáveis (Figura 2.), que serviram para ensaios.

a) cenário a: com duas variáveis globais inicializadas com 1, atribuídas às variáveis locais tipo ponteiro para endereço de memória, que são enviadas por referência a um procedimento que faz comparação e impressão. Esta versão apresenta erro semântico na atribuição da variável de comparação;

b) cenário b: com duas variáveis globais inicializadas com 1, atribuídas às variáveis – objeto de memória, que são enviadas por valor a um procedimento que faz comparação e impressão. Esta versão apresenta erro semântico na atribuição da variável de comparação;

c) cenário c: com duas variáveis globais inicializadas com 0, atribuídas às variáveis locais tipo ponteiro – endereço de memória, que são enviadas por referência a um procedimento que faz comparação e impressão. Esta versão apresenta erro semântico na atribuição da variável de comparação;

d) cenário d: com duas variáveis globais inicializadas com 0, atribuídas às variáveis – objeto de memória, que são enviadas por valor a um procedimento que faz comparação e impressão. Esta versão apresenta erro semântico na atribuição da variável de comparação;

e) cenário e: com duas variáveis globais inicializadas com 1, atribuídas às variáveis locais tipo ponteiro para endereço de memória, que são enviadas por referência a um procedimento que faz comparação e impressão. Esta versão não apresenta erro.

```
#include <iostream>
int a=1,b=1;
void processo(int *x,
int *y){
 int max;
 while (*x>0&&*y>0){
  std::cin>>*x>>*y;
  if (*x>*y){
      max=b;}
  else {
      max=a;}
  std::cout <<"max:
"<<max<<"\n";}
}
int main(){
  int *x,*y;
  x=&a;y=&b;
  processo(x,y);
}
```

(a)

```
#include <iostream>
int a=1,b=1;
void processo(int x,
int y){
int max;
while (x>0&&y>0){
  std::cin>>x>>y;
  if (x>y){
      max=b;}
   else {
      max=a;}
   std::cout<<"max:
"<<max<<"\n";}
}
int main(){
  int x,y;
  x=a;y=b;
  processo(x,y);
}
```

(b)

```
#include <iostream>
int a=0,b=0;
void processo(int *x,
int *y){
 int max;
 while (*x>0&&*y>0){
 std::cin>>*x>>*y;
 if (*x>*y){
      max=b;}
    else {
      max=a;}
   std::cout<<"max:
"<< max<<"\n";}
}
int main(){
  int *x,*y;
  x=&a;y=&b;
processo(x,y);
}
```

(c)

```
#include <iostream>
int a=0,b=0;
void processo(int x, int y){
 int max;
 while (x>0&&y>0){
  std::cin>>x>>y;
  if (x>y){
      max=b;}
  else {
      max=a;}
  std::cout <<"max: "<<max<<"\n";}
}
int main(){
  int x,y;
  x=a;y=b;
  processo(x,y);
}
```

(d)

```
#include <iostream>
int a=1,b=1;
void processo(int *x, int *y){
int max;
while (*x>0&&*y>0){
  std::cin>>*x>>*y;
  if (*x>*y){
      max=a;}
   else {
      max=b;}
   std::cout<<"max: "<<max<<"\n";}
}
int main(){
  int *x,*y;
  x=&a;y=&b;
  processo(x,y);
}
```

(e)

Figura 2. Programas apresentados nos cenários: (a) inicializado com 1, passagem por referência, erro semântico; (b) inicializado com 1, passagem por valor, erro semântico; (c) inicializado por 0, passagem por referência, erro semântico; (d) inicializado por 0, passagem por valor, erro semântico; e, (e) inicializado por 1, passagem por referência, sem erro.

Inicialmente foi calculada a métrica da complexidade ciclomática para o grafo produzido, aplicou-se as equações eq. 1 e eq. 2 para o programa genérico, considerando a recomendação didática mencionada. Em seguida, foram executados os programas utilizando-se a notação do fluxo de controle lógico (Pressman, 2016) para os cinco cenários considerados, empregou-se o critério de Todos os Caminhos (Maldonado, 1991), que exige que todos os fluxos possíveis do programa sejam executados ao menos uma vez.

Para automatizar os procedimentos, mostrar didaticamente as métricas do teste estrutural e fazer a comparação dos resultados da execução dos programas, foi desenvolvida uma ferramenta educacional para ilustrar o *parsing* em programas nas linguagens C, C++ e derivadas, calcular a complexidade ciclomática, projetar o grafo correspondente e indicar os caminhos independentes, na forma das indicações didáticas em Sommerville (2008) e Pressman (2016).

A ferramenta foi desenvolvida com a IDE Rad Studio, na linguagem C++Builder (Embarcadero, 2017), com repositório de dados em documentos XML para os atributos: projeto de teste de software, código do programa, cálculo da complexidade ciclomática, matriz de adjacências e caminhos independentes.

O trabalho da ferramenta inicia no reconhecimento do código e composição de um *token* específico para representar as estruturas de controle dos fluxos lógicos e seus escopos. Depois, decompor em unidades estruturais para serem organizadas numa árvore com hierarquia para empilhar e possibilitar posterior processamento de reconhecimento dessas estruturas lógicas, para indicação dos predicativos.

Em seguida, as operações numa pilha possibilitam a contagem de vértices e arestas de acordo com os escopos dos possíveis fluxos nas estruturas lógicas de controle, que junto com as operações sequenciais, indicam os caminhos independentes no grafo.

Para em seguida a formação de uma matriz de adjacência A para representar o grafo orientado do programa, sem pesos nas arestas. Em que as entradas E_{ij} da matriz A registram binariamente 1, se V_i e V_j são adjacentes, caso contrário registram 0. A matriz de adjacência também é usada para aplicação dos cálculos de complexidade ciclomática.

3. Resultados

O resultado do cálculo da complexidade ciclomática $V(G)$ do grafo de fluxo na Figura 1(b) é igual a 3, pois são apresentados 2 nós predicados mais 1 constante ($V(G) = 2 + 1$), como também, é definido por 9 arestas, menos 8 nós, mais 2 constantes ($V(G) = 9 - 8 + 2$).

Isso pressupõe que existem 3 caminhos independentes para formação de casos de teste no limite superior. Os caminhos independentes resultantes do grafo, pelos dados genéricos, na forma didática descrita anteriormente, aqui denominado métrica tradicional, são:

A) 1,2,3,4,5,6,7,8,10,4,11
B) 1,2,3,4,5,6,7,9,10,4,11
C) 1,2,3,4,11

Considerando os cenários de dados *a, b, c, d* e *e*, descritos anteriormente, mostrados no Quadro 1., foram executados os programas com entrada de dados variados presumindo-se todas as possibilidades de execução, considerando os caminhos completos.

(a) variáveis inicializadas com 1, passagem por referência, erro semântico										
Ordem	a	b	while(*x>0&&*y>0)	cin>>*x;	cin>>*y;	If(*x>*y)	max =b;	max =a;	cout >> max;	Caminho
1.1	{1}	{1)	V	(2)	(8)	F	-	{2}	2	1,2,3,4,5,6,7,9,10,4,5, 6,7,9,10,4,11
1.2	{2}	{8}	V	(0)	(0)	F	-	{0}	0	
2.1	{1}	{1}	V	(8)	(2)	V	{2}	-	2	1,2,3,4,5,6,7,8,10,4,5, 6,7,9,10,4,11
2.2	{8}	{2}	V	(0)	(0)	F	-	{0}	0	
3.	{1}	{1}	V	(0)	(0)	F	-	{0}	0	1,2,3,4,5,6,7,9,10,4,11
4.	{1}	{1}	V	(-2)	(-8)	V	{-8}	-	-8	1,2,3,4,5,6,7,8,10,4,11
(b) variáveis inicializadas com 1, passagem por valor, erro semântico										
Ordem	a	b	while(x>0&&y>0)	cin>>x;	cin>>y;	If(x>y)	max =b;	max =a;	cout >> max;	Caminho
1.1	{1}	{1)	V	(2)	(8)	F	-	{1}	1	1,2,3,4,5,6,7,9,10,4,5, 6,7,9,10,11
1.2	{1}	{1}	V	(0)	(0)	F	-	{0}	1	
2.1	{1}	{1}	V	(8)	(2)	V	{1}	-	1	1,2,3,4,5,6,7,8,10,4,5, 6,7,9,10,11
2.2	{1}	{1}	V	(0)	(0)	F	-	{1}	1	
3.	{1}	{1}	V	(0)	(0)	F	-	{1}	1	1,2,3,4,5,6,7,9,10,4,11
4.	{1}	{1}	V	(-2)	(-8)	V	{1}	-	1	1,2,3,4,5,6,7,8,10,4,11
(c) variáveis inicializadas com 0, passagem por referência, erro semântico										
Ordem	a	b	while(*x>0&&*y>0)	cin>>*x;	cin>>*y;	If(*x>*y)	max =a;	max =b;	cout >> max;	Caminho
1.	{0}	{0}	F	-	-	-	-	-	-	1,2,3,4,11
(d) variáveis inicializadas com 0, passagem por valor, erro semântico										
Ordem	a	b	while(x>0&&y>0)	cin>>x;	cin>>y;	If(x>y)	max =a;	max =b;	cout >> max;	Caminho
1.	{0}	{0}	F	-	-	-	-	-	-	1,2,3,4,11
(e) variáveis inicializadas com 1, passagem por referência, sem erro										
Ordem	a	b	while(*x>0&&*y>0)	cin>>*x;	cin>>*y;	If(*x>*y)	max =a;	max =b;	cout >> max;	Caminho
1.1	{1}	{1}	V	(2)	(8)	F	-	{8}	8	1,2,3,4,5,6,7,9,10,4,5, 6,7,9,10,4,11
1.2	{2}	{8}	V	(0)	(0)	F	-	{0}	0	
2.1	{1}	{1}	V	(8)	(2)	V	{8}	-	8	1,2,3,4,5,6,7,8,10,4,5, 6,7,9,10,11
2.2	{8}	{2}	V	(0)	(0)	F	-	{0}	0	
3.	{1}	{1}	V	(0)	(0)	F	-	{0}	0	1,2,3,4,5,6,7,9,10,4,11
4.	{1}	{1}	V	(-2)	(-8)	V	{-2}	-	-2	1,2,3,4,5,6,7,8,10,4,11

Quadro 1. Teste de mesa para os cenários de dados *a, b, c, d* e *e*, com respectivos caminhos independentes.

Para o **cenário (a)**, ordem 1., as variáveis globais *a* e *b* inicializadas com 1 induziram a entrada no escopo da estrutura repetitiva, para novas entradas de x=2 e y=8 que forçaram novo laço repetitivo para as entradas de x=0 e y=0, quando houve completude do ciclo do caminho e posterior parada regular do programa. Em razão do referenciamento por ponteiros a cada laço os valores nas variáveis globais *a* e *b* são alterados.

O mesmo comportamento para a ordem 2. Na ordem 3. (entradas com 0) e 4. (entradas com valores negativos), nota-se o mesmo comportamento de execução da estrutura de repetição e posterior saída do programa com valores inconsistentes. As saídas foram resultadas inconsistentes por causa do erro semântico apresentado na estrutura de seleção composta. Observa-se indicação de 4 caminhos independentes, conflitando-se com 3 da abordagem tradicional.

Para o **cenário (b)**, ordem 1., as variáveis globais *a* e *b,* também inicializadas com 1 forçaram a entrada na estrutura repetitiva. As entradas de x=2 e y=8 induziram novo laço de repetição para recebimento das entradas de x=0 e y=0 no propósito da completude do ciclo do caminho e posterior parada do programa. Para as ordens 2., 3. e 4. os comportamentos foram os mesmos do cenário (a).

Em todas as ordens, as variáveis sem referenciamento impediram a alteração de valor nas variáveis globais, forçando a saída do programa sempre inconsistente em 1, nos mesmos valores inicializados em *a* e *b*, embora na estrutura de seleção houvesse um erro semântico. Observa-se a indicação dos mesmos caminhos do cenário (a) e conflito com a métrica da abordagem tradicional, com saídas diferentes e também inconsistentes.

Nos **cenários (c) e (d)** com variáveis globais inicializadas com 0, não importando se uso de variáveis referenciadas ou por objeto de valor, o caminho não entra na estrutura repetitiva, forçando finalização do programa, sem resultar saída de valores. Observou-se que o erro na saída está em razão da não execução do escopo das estruturas de controle do fluxo, embora houvesse um erro semântico no desvio condicional. Esses cenários indicaram discrepância entre a quantidade de caminhos independentes (apenas 1) com a quantidade de caminhos independentes da abordagem tradicional (3).

No **cenário (e)**, foram empregadas variáveis globais inicializadas com 1, o uso de variáveis referenciadas, as mesmas entradas de dados dos cenários *a* e *b*, resultando na mesma quantidade e nos mesmos caminhos independentes, conflitando com a métrica tradicional. Porém, sem erro semântico na estrutura de seleção composta. Em todas as ordens o comportamento do programa foi o mesmo para os cenários *a* e *b* e o resultado na saída do programa foi correto, conforme previsto.

Foi aplicada a técnica *Crosstab* do pacote estatístico SPSS 18.0 (Pallant, 2013), para analisar a homogeneidade entre as variáveis independentes – Abordagem do Caminho Tradicional e Abordagem do Caminho Calculado, de acordo com os cenários *a, b, c, d* e *e*, objetivando conhecer a quantidade de ocorrências idênticas. O resultado está mostrado no Quadro 2., onde nas linhas estão os parâmetros que resultaram no caminho calculado respectivo, e nas colunas os caminhos pela abordagem tradicional. Um valor na intersecção entre linha e coluna indica homogeneidade ou a falta de correlação entre os resultados dos métodos e, a falta de valor indica discrepância entre os métodos e possível não alcance do limite superior pelo método dos caminhos independentes.

Saída certa	Erro semantico	Inicializacao	Tipo Dado	while	Entrada	if	**Abordagem do Caminho Calculado**	**Abordagem do Caminho Tradicional** (A) 1,2,3,4,5,6,7,8,10,4,11	(B) 1,2,3,4,5,6,7,9,10,4,11	(C) 1,2,3,4,11	(D) Sem correlação	Total
Não	Sim	1	Referência	V	Num. Negativo	V	(A) 1,2,3,4,5,6,7,8,10,4,11	1				1
				V	Zero	F	(B) 1,2,3,4,5,6,7,9,10,4,11		1			1
					[1]Num. Positivo	V	(D) 1,2,3,4,5,6,7,9,10,4,5,6,7,9,10,4,11				1	1
							(D) 1,2,3,4,5,6,7,8,10,4,5,6,7,9,10				1	1
			Valor	V	Num. Negativo	V	(A) 1,2,3,4,5,6,7,8,10,4,11	1				1
					Zero	F	(B) 1,2,3,4,5,6,7,9,10,4,11		1			1
					[1]Num. Positivo	V	(D) 1,2,3,4,5,6,7,9,10,4,5,6,7,9,10,4,11				1	1
							(D) 1,2,3,4,5,6,7,8,10,4,5,6,7,9,10				1	1
Sim	Não	1	Referência	V	Num. Negativo	V	(A) 1,2,3,4,5,6,7,8,10,4,11	1				1
					Zero	F	(B) 1,2,3,4,5,6,7,9,10,4,11		1			1
					[1]Num. Positivo	V	(D) 1,2,3,4,5,6,7,9,10,4,5,6,7,9,10				1	1
							(D) 1,2,3,4,5,6,7,8,10,4,5,6,7,9,10,11				1	1
	Sim	0	Referência	F	Nulo	*	(C) 1,2,3,4,11			1		1
			Valor	F	Nulo	*	(C) 1,2,3,4,11			1		1
[1]considera laços de repetição							Total	3	3	2	6	14

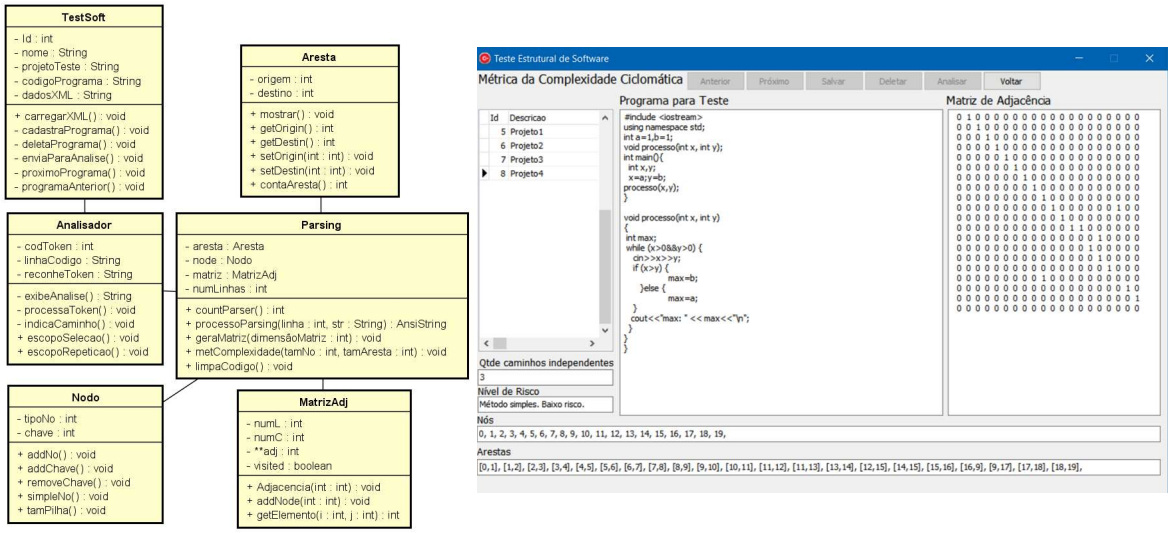
Quadro 2. Correlação entre abordagens pela técnica *Crosstab*.

Observa-se o mesmo comportamento com os cenários *a, b* e *e* com as mesmas ocorrências de caminhos independentes, não importando o cenário. Os novos caminhos independentes "sem correlação" entre as abordagens ocorreram, sobretudo, em razão da inicialização das variáveis globais que forçaram novos laços de repetição.

Para se chegar a esses resultados, o uso da ferramenta desenvolvida foi crucial. Porém, neste estágio inicial a ferramenta ainda é considerada um protótipo em evolução para aplicação plena como instrumento didático, pois somente dispõe de visualização do grafo do programa e dos caminhos executáveis em caracteres e na forma de matriz,

deixando de apresentar os fluxos e caminhos graficamente, o que seria melhor para o processo ensino-aprendizado.

Observa-se na Figura 3(a) o diagrama de classe da ferramenta, com as principais interfaces integradas. A classe principal *TestSoft* carrega o arquivo de marcação em XML onde os dados dos projetos estão em persistência, esta tem recurso para inserir, atualizar e consultar projetos para teste, que invoca objeto da classe *Analisador* para reconhecer através de um *token* as estruturas lógicas e respectivos escopos de validação e indicar a sequência do grafo do fluxo com auxílio de uma pilha. A classe *Parsing* também processa as estruturas analisadas, registra os nós (classe *Nodo*) e arestas (classe *Aresta*) através de pilha e reconhece o grafo do fluxo de controle para registrar na matriz de adjacência (classe *MatrizAdj*), calcular a métrica da complexidade ciclomática e possibilitar a indicação dos possíveis caminhos independentes.



(a) (b)

Figura 3. (a) Diagrama de classe da ferramenta de teste de software e (b) tela de entrada de dados e métrica para complexidade ciclomática.

A interface de usuário na Figura 3(b) foi construída para mostrar didaticamente o processo da métrica do teste estrutural. Possui recursos para carregar, editar e salvar os projetos de métricas no documento XML, ilustrar o processo de contagem das estruturas lógicas do programa e mostrar o resultado da métrica da complexidade ciclomática no limite superior, a análise do risco de erros utilizando a tabela de probabilidade de McCabe (1976), os nós e arestas do grafo do fluxo de controle e a matriz de adjacência.

4. Considerações finais

Este artigo apresenta uma discussão sobre o conteúdo didático que menciona a métrica da complexidade ciclomática, como indicador para a quantidade de caminhos independentes no limite superior e, não considera, parâmetros de inicialização de variáveis globais e referenciamento para memória.

Para tanto, foram feitos ensaios com um programa e cinco cenários com parâmetros diversos objetivando mostrar possibilidades de diferentes contagens para

caminhos independentes conforme entradas, daquela indicada pela complexidade ciclomática tradicional. Além da apresentação de um protótipo de uma ferramenta didática para ilustrar o processo do teste estrutural em cursos de engenharia de software.

O resultado mostrou diferença na quantidade de caminhos independentes em razão do emprego da inicialização de variáveis globais, o que possibilita presumir erros na indicação de casos de teste de software. Essa situação não é mencionada na literatura didática, podendo levar a erros de interpretação no processo ensino-aprendizagem.

A ferramenta desenvolvida mostrou viabilidade de aplicação para a finalidade proposta nas duas abordagens e diversos cenários de dados, pois em todos ensaios os grafos calculados e contagem de caminhos resultaram em números equivalentes entre si conforme as equações empregadas.

Para o futuro pretende-se avançar para mostrar graficamente o grafo do fluxo de controle e os caminhos independentes gerados. Pretende-se também dispor de licença do software do tipo *GPL – General Public License,* quando o processo tramitar junto ao licenciador, com o propósito da disseminação da prática e auxílio para interessados que dispõem de poucos recursos ou pouco conhecimento para tal avaliação.

Referências

Beizer, Boris. *Software testing techniques*. Dreamtech Press, 2003.

Clarke, L, Podgurski, A., Richardson, D.J and Zeil S.J., "A Comparison of Data Flow Path Selection Criteria", in Proc. of the 8th int'l Conf. on Software Engineerin, Ago 1985, pp. 244-251.

Embarcadero Technologies. RX Rad Studio. https://www.embarcadero.com/br/products/rad-studio. Disponível: ago/2017, Acessado: ago/2017.

Frankl, Phyllis G. *Use of data-flow information for the selection and evaluation of software test data*. New York Univ., NY (USA), 1987.

Herman, P.M. "A Data Flow Analysis Approach to Program Testing", The Australian Computer Journal, Vol. 8. No. 3, Nov. 1976, pp. 92-96.

Howden, William E. "Reliability of the path analysis testing strategy." *IEEE Transactions on Software Engineering* 3 (1976): 208-215.

Laski, Janusz W., and Bogdan Korel. "A data flow oriented program testing strategy." IEEE Transactions on Software Engineering 3 (1983): 347-354.

Maldonado, J. Carlos. "Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados", Tese de Doutorado, DCA/UNICAMP, Campinas - SP, 1991.

McCabe, Thomas J. "A complexity measure." *IEEE Transactions on software Engineering* 4 (1976): 308-320.

Pallant, Julie. *SPSS survival manual*. McGraw-Hill Education (UK), 2013.

Pressman, Roger S. *Engenharia de software*. 8ª edição. Sao Paulo: Makron books, 2016.

Rapps, Sandra, and Elaine J. Weyuker. "Data flow analysis techniques for test data selection." *Proceedings of the 6th international conference on Software engineering*. IEEE Computer Society Press, 1985.

Sommerville, Ian. "Engenharia de Software, 8ª edição, Tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edilson de Andrade Barbosa." *São Paulo: Pearson Addison-Wesley* 22 (2007): 103.

Weyuker, E.J., "On Testing Complexity of Data Flow Criteria for Test Data Selection", Information Processing Letters, vol. 19, n. 2, Aug. 1984, pp. 103-109