

Metodologias Experimentais em Informática 2017/2018

Assignment 2

Experiment Design

André Clemêncio - 2013152406

Joel Pires - 2014195242

Pedro Andrade - 2014225147

Universidade de Coimbra
Departamento de Engenharia Informática
Mestrado em Engenharia Informática

Índice

Introdução	3
Abstract	4
Definição do Problema	4
Ambiente Experimental	6
Critérios de Selecção dos Programas	6
Programas Seleccionados	7
Programa 1 - Travelling Salesman	7
Programa 2 - Parser Calculadora	8
Scripts Desenvolvidos	9
Variáveis Estatísticas	10
Definição de Hipóteses	11
Conjunto de Testes	12
Resultados e Análise dos Testes	14
Travelling Salesman	14
Parser Calculadora	16
Análise e Teste de Hipóteses	19
Conclusão	21
Glossário	21
Referências	21
Anexo A	22
Anexo B	34

Introdução

Pretende-se com este trabalho **planear uma experiência de forma a que os dados obtidos possam ser analisados e conclusões possam ser retiradas**. A ideia para o projeto surgiu no âmbito do 2º trabalho para a unidade curricular de **Metodologias Experimentais em Informática da Universidade de Coimbra** no ano letivo de 2017/2018 e a cargo do Professor Henrique Madeira.

Dado que o design experimental ocorre em contexto académico, é naturalmente expectável uma simplificação de alguns aspetos do problema, embora **o enfrentemos com a minúcia e rigor que ele merece** com todos os passos que uma experiência de um sistema de software assim o exige.

Debrucemo-nos um pouco sobre o que está envolvido no **design experimental** de uma tarefa. O objetivo de se fazer este *design* experimental é descrever ou explicar a variação de informação em condições que são hipotetizadas para refletir a variação. De forma clara e concisa, uma experiência tem a finalidade de prever o resultado no caso de se introduzir uma alteração das pré-condições, designadas por **variáveis independentes** (ou **variáveis de entrada**). A alteração de uma ou mais destas variáveis é **hipotetizada** para resultar numa mudança em uma ou mais **variáveis dependentes** (ou **variáveis de saída**). Convém mencionar a existência de **variáveis de controle** que devem permanecer inalteráveis de forma a evitar que fatores externos influenciem os resultados. Está então envolvido no design experimental não só a seleção de variáveis dependentes, independentes e as de controle que acharmos adequadas, mas também o planeamento da entrega da experiência em condições estatisticamente ótimas dadas as restrições dos recursos disponíveis.

As **principais preocupações** no projeto experimental que vamos levar a cabo é o estabelecimento de **validade, confiabilidade e replicação**. Para isso teremos o cuidado de escolher cuidadosamente a variável independente, reduzindo o risco de erro de medição e garantindo que a documentação do método é suficientemente detalhada. Garantimos também a obtenção de níveis adequados de poder estatísticos e sensibilidade. Iremos, portanto, seguir todos as etapas envolvidas numa experiência, a saber:

- Definir o problema;
- Identificar as variáveis;
- Gerar hipóteses;
- Definir o **setup** experimental;
- Desenvolver as ferramentas e procedimentos necessários para realizar a experiência;
- Correr a experiência e extrair medições;
- Análisar os dados;
- Retirar as devidas conclusões.

Comecemos então pelo primeiro passo - a definição do nosso problema.

Abstract

O propósito deste trabalho é o de executar todos os passos necessários para avaliar a resposta ao seguinte problema:

Qual é a probabilidade de um conjunto de testes detetar falhas de software que ainda existem no código do programa?

O objetivo da experiência é então apurar a qualidade de um conjunto de testes em termos de probabilidade do conjunto de testes sob avaliação para detectar possíveis falhas no programa a ser testado. É expectável que um conjunto de casos de teste detectam todos os bugs residuais num determinado programa mas, infelizmente, muitas vezes não é isto que acontece.

Assim sendo o objetivo deste trabalho é planear e realizar uma experiência que nos permita:

- Apurar a probabilidade de detecção de bugs de um típico conjunto de testes desenvolvidos por programas reais.
- Formular uma hipótese e testar essa hipótese de forma a discutir as consequências do resultado de tal teste.

Definição Problema

Este trabalho será focado em teste de software. Um conjunto de casos de teste será usado para testar o software onde se irá comparar o output do programa (de acordo com determinados inputs recebidos) com o resultado expectável desse mesmo programa, mas livre de bugs.

Será então necessário definir um bom conjunto de casos de teste que assegure (tanto quanto possível) que os bugs no software serão detectados por um ou mais casos de teste.

Em relação ao teste de software, existem 2 grandes métodos para o fazer: Black box testing e White box testing. O Black-box testing testa a funcionalidade de uma aplicação sem olhar para o código do programa. Em oposição, o White-box testing tira vantagem do conhecimento prévio do código do programa de modo a assegurar que todo o código é testado adequadamente.

Foram seleccionados 2 programas diferentes para serem testados definindo posteriormente um conjunto de casos de teste para cada um deles. Foi usado o

método de Black-box testing para definir os casos de teste. A ideia principal é avaliar se um determinado caso de teste consegue detectar um bug que foi previamente injetado no programa comparando o output com aquele que seria inicialmente expectável. Foram então injetados bugs nos dois programas recorrendo a um software que os injeta automaticamente. A partir da utilização deste software foram gerados 100 patches para o primeiro programa (Travelling Salesman problem) e 200 patches no caso do segundo programa (Parser, onde é possível realizar inúmeras operações matemáticas).

Foi desenvolvido um pequeno script que nos permitiu automatizar todo o processo de teste dos dois programas. O script recorre a ficheiros com variados inputs e corre o programa em teste com cada um dos patches gerados fazendo a análise do output de seguida.

O objetivo deste assignment é então fazer um design experimental executando todos os seus passos, para que posteriormente se possa avaliar os resultados do seguinte problema: qual será a probabilidade de um conjunto de testes detetar falhas no software (bugs) que existam no código fonte de um programa.

Ambiente Experimental

Critérios de Seleção dos Programas

Para a experiência foram escolhidos dois programas. Antes de detalhar cada um destes programas gostaríamos apenas de explicar alguns dos fatores que foram levados em conta no critério de seleção destes programas:

- Conforme já referido os bugs são inseridos um de cada vez, gerando diferentes patches. Ora pretende-se então que os programas escolhidos possam **gerar um bom número de patches** de forma a que possamos ter dados suficientes para podermos tirar conclusões plausíveis e válidas (convenhamos que um bom número é para cima de 50 patches). Quantas mais patches forem geradas melhor.
- Ainda relacionado com o de cima, será abordada a característica do **número de linhas que os nossos programas devem ter**. Convenhamos que um bom número de linhas de 150 linhas para cima. É claro que houve o cuidado de que essas linhas fossem passíveis de lhe serem injetadas falhas. Assim, teve-se o cuidado de escolher programas cujo grande número de linhas fosse ocupado por comentários ou estruturas de dados.
- Se podemos escolher mais que um programa, então podemos escolher **diferentes dimensões com um diferente número de linhas/patches gerados** para que a nossa análise de dados seja ainda mais robusta e completa, em vez de escolher programas de tamanho similar.
- Foi dada preferência a **programas escritos na linguagem de C/C++**. Isto porque, conforme já referido anteriormente, foi usada uma ferramenta que injeta automaticamente bugs em programas escritos em C/C++. Se escolhêssemos programas escritos com outra linguagem de programação, o trabalho de investigar que tipo de bugs existem para essa linguagem e como injetá-los teria de ser feito de forma adicional para além do facto de que em nada adicionaria à problemática do trabalho em questão.
- Foram seleccionados programas cujo **output esperado coincide com o output efetivo do programa**. Por outras palavras, programas que sabemos que estão 100% corretos na resolução dos problemas a que se propõe resolver.
- Relacionado com o ponto anterior, os programas têm de estar **preparados para casos limites** (casos esses que vão ser também alvos de teste ao

longo da experiência). Com casos limites referimo-nos por exemplo a inserir como dados de entrada tipos de dados completamente diferentes dos esperados, ausência de inputs, quantidade de inputs inadequada, etc.

- Os programas têm de ser **passíveis de serem testados de forma a que todo o seu código seja percorrido**. Caso isto não acontecesse, haveria partes de código cuja utilidade seria nula e que não possuiriam absolutamente nenhum valor.
- Programas preparadas para **receber os inputs por meio de argumentos numa linha de comandos**. Esta questão trata-se de uma preferência nossa que visa facilitar mais tarde a automatização dos testes por meio de um script.
- Queremos que o **output do programa seja passível de ser comparado com o output desejado** de forma automática por meio de um script. Há programas em que esta tarefa de comparação se torna difícil de implementar por meio de código. Por exemplo, quando se trata de conversão de imagens ou compressão/descompressão de ficheiros.
- É imperativo seleccionar programas em que **saibamos calcular de forma relativamente fácil o output esperado**. Que sentido faz desenvolver casos de teste (neste caso black-box testing) se não temos a certeza do output que o programa deve ter?

Programas Seleccionados

Foram seleccionados dois programas que cumprem os critérios mencionados anteriormente, a saber:

Programa 1: Travelling Salesman

- **Fonte:** [Repositório do Github](#)
- **Nome do Ficheiro:** *trav_sales.cpp*
- **Linguagem de Programação:** C++
- **Descrição:** Este é um problema famoso. Este programa usa a técnica de branch-and-bound para tentar visitar todas as N cidades apenas uma vez e

retornar ao ponto de partida. Queremos saber não só o custo mínimo dessa operação, mas também o caminho que nós temos que tomar.

- **Número de Linhas:** 147 linhas
- **Número de Patches Geradas:** 100 patches
- **Inputs:** Primeiro insere-se a dimensão da nossa matriz (basta um número pois o número de linhas será igual ao número de colunas - típico de uma matriz de adjacência). Cada célula da matriz deve conter a distância em quilómetros entre cada par de cidades. Aqui está um exemplo:

```

5
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0

```

- **Outputs:** O programa deve gerar o custo mínimo da operação e o caminho seguido.

```

Minimum cost: 80
Path Taken: 0 1 3 2 0

```

Programa 2: Parser Calculadora

- **Fonte:** [Repositório do Github](#)
- **Nome do Ficheiro:** *parser.cpp*
- **Linguagem de Programação:** C++
- **Descrição:** Trata-se de um programa que funciona como parser de expressões matemáticas. Todas as expressões que envolvam operações que uma calculadora científica consegue processar, este programa resolve.
- **Número de Linhas:** 579 linhas
- **Número de Patches Geradas:** 200 patches
- **Inputs:** Qualquer expressão com operações de uma calculadora científica, incluindo operações bitwise.

$$(2 + 3) * 6$$
- **Outputs:** O programa devolve simplesmente o resultado do cálculo matemático:

Scripts desenvolvidos

Como referido anteriormente, para reduzir o tempo e esforço envolvido nas etapas da experimentação, foram desenvolvidos dois scripts (um para cada programa) com o objetivo de automatizar ao máximo a aplicação dos patches e recolha de resultados.

Esta secção visa explicar com maior detalhe o funcionamento dos dois scripts e a forma como foram implementados no contexto deste trabalho.

Depois de seleccionados os dois programas para este trabalho, uma primeira abordagem que seguimos foi uniformizar a forma como são passados os *inputs* e recolhidos os *outputs*. Optámos então por fornecer a cada programa ficheiros com os *inputs* (um ficheiro para cada caso de teste) e recolher os *outputs* do programa com cada um dos patches para um outro ficheiro de *output*, sendo que a quantidade de programas que falharam é mostrada no terminal no final da execução.

Esta uniformização permitiu-nos criar dois scripts com um funcionamento idêntico para os dois programas. Inicialmente é feita uma escolha do caso de teste, sendo carregado o ficheiro respectivo. De seguida são percorridos um a um todos os patches gerados anteriormente. Cada patch é inserido no código original, sendo depois necessário recompilar o programa e executá-lo. Na próxima fase o patch é removido e o output do programa com o bug é comparado com o output esperado. Esta ordem de intruções é realizada para cada um dos patches e, no final, é apresentado o valor do número de vezes em que um programa com bug apresenta o output esperado, isto é, o número de vezes em que o programa não detetou o bug e correu naturalmente.

Variáveis Estatísticas

Variáveis Independentes

Foram definidas **2 variáveis independentes** nesta experiência, a saber:

- **Tipos de Testes**
- **Tipos de Bugs**

Foram definidos conjuntos de testes diferentes para cada um dos programas em análise. Foram definidos 10 casos de teste diferentes para o programa do Travelling Salesman e 13 testes diferentes para o caso do Parser Calculadora. A descrição detalhada de cada um dos casos de testes encontra-se mais à frente neste documento. No entanto, é notório referir que foram testados casos limite (ausência de inputs, inputs cujo tipo de dados não fosse adequada, quantidade de inputs inadequada) e foram testados casos não limite que progressivamente atingem níveis de **complexidade** e **quantidades** de input cada vez maiores.

Relativamente aos tipos de Bugs dizem respeito àqueles que são injetados automaticamente pela ferramenta que adotamos e que dizem respeito especificamente à linguagem C/C++. Os bugs podem ser de 13 tipos:

- **MFC** - quando falta um operador na chamada de uma FUNção
- **MVIV** - quando falta um operador na inicialização de uma variável com um valor
- **MVAV** - quando falta um operador na atribuição de um valor a uma variável
- **MVAE** - quando falta um operador na atribuição de uma expressão a uma variável
- **MIA** - quando falta um operador em *if-statements*
- **MIFS** - quando falta IF nas linhas que engloba
- **MIEB** - quando há uma má colocação de if's e elses
- **MLAC** - quando falta uma expressão do tipo 'e' numa expressão lógica
- **MLOC** - quando falta uma expressão do tipo 'ou' numa expressão lógica
- **MLPA** - quando é perdida uma parte localizada do algoritmo
- **WVAV** - quando o valor atribuído a uma variável está errado
- **WPFV** - quando o operador de uma variável num parâmetro de uma função está errado
- **WAEP** - quando o operador aritmético de uma expressão num parâmetro de uma função está errado

Variáveis Dependentes

A variável dependente, isto é, aquela que neste caso varia consoante o tipo e a quantidade de teste é a **Percentagem de Bugs Detectados**.

Variáveis de Controlo

Existem fatores que poderiam afetar a nossa variável dependente, mas que nós vamos anulá-lo por manipulá-lo deliberadamente de forma a não interferir na relação entre variável independente e dependente. Nesta experiência poderemos indicar o tipo de linguagem de programação dos nossos programas como uma variável de controlo porque foram escolhidos programas que tivessem a mesma linguagem de programação, a saber, C++.

Hipóteses

Um teste de hipóteses é um procedimento estatístico que nos permite tomar uma determinada decisão recorrendo aos dados que foram obtidos numa experiência.

Estabelece-se então uma Hipótese Nula (H_0) que é a hipótese que inicialmente é assumida como sendo a verdadeira para a construção dos casos de teste. Posteriormente estabelece-se a Hipótese Alternativa (H_1) que é aquela que é assumida quando a hipótese nula falhou (não existe evidência estatística para provar H_0 como verdadeira).

De notar que será extremamente importante definir bem as duas hipóteses visto que caso não sejam assumidas de uma maneira correta o resultado obtido será também incorreto e consequentemente a informação será incoerente com a questão da análise experimental que estamos a desenvolver.

Vamos então definir um conjunto de duas hipóteses que são válidas para os dois programas:

- **H_0** - A probabilidade de um conjunto de testes conseguir detetar falhas no código fonte de um software é inferior a 0.8 com grau de confiança de 0.95
- **H_1** - A probabilidade de um conjunto de testes conseguir detetar falhas no código fonte de um software é igual ou superior a 0.8 com grau de confiança de 0.95

Conjuntos de Testes

Para cada programa foram, naturalmente, efetuados testes diferentes.

Para o programa do **Travelling Salesman**, foi efetuado o seguinte conjunto de testes:

Nº do Caso de Teste	Número de Inputs Diferentes	Tamanho das Matrizes de Adjacência
1	2	todas 2x2
2	2	todas 2x2
3	2	2x2 e 3x3, respetivamente
4	4	duas 2x2 e duas 3x3
5	6	uma 3x3 e as restantes 4x4
6	8	duas 3x3, três 4x4, duas 5x5 e uma 6x6
7	9	cinco 4x4, três 5x5 e uma 6x6
8	11	oito 5x5 e três 6x6
9	13	uma 5x5 e as restantes 6x6
10	15	cinco 5x5 e as restantes 6x6

Todos estes casos de teste estão disponibilizados no Anexo A.

Para o programa do **Parser Calculadora**, foram efetuados o seguinte conjunto de testes:

Nº do Caso de Teste	Número de Inputs Diferentes	Tipos de Operações
1	1	soma
2	6	somas e subtrações
3	6	somas, subtrações, multiplicações e divisões, com uso de parêntesis
4	10	todas as operações anteriores e usando operadores de comparação como o '<=' e '>='
5	13	todas as anteriores, combinado com a operação logarítmica
6	15	todas as anteriores, combinado com a operação de raiz quadrada
7	17	todas as anteriores usando também a operação de potência
8	20	todas as anteriores mais as operações trigonométricas de seno, cosseno e tangente
9	23	todas as anteriores mais atribuição de valores a variáveis 'a =42', 'b=5', 'c=30'
10	25	todas as anteriores mais operações bitwise ' ', '&&'
11	24	todas as anteriores mais conversão de float para inteiro, por exemplo 'int(1.2)
12	26	combinações de todas as anteriores
13	34	combinações de todas as anteriores mais uso de 'if's'

Todos estes casos de teste estão disponibilizados no Anexo B

Resultados e Análise dos Testes

Travelling Salesman

Para o caso do **Travelling Salesman** foram colocados os resultados dos testes na seguinte tabela:

Nº do Caso de Teste	Nº de Patches onde o Bug não é detectado	Nº de Patches onde o Bug é detectado	% de Patches onde o Bug não é detectado	% de Patches em que o Bug é detectado
1	36	64	36%	64%
2	31	69	31%	69%
3	25	75	25%	75%
4	23	72	23%	77%
5	15	85	15%	85%
6	15	85	15%	85%
7	12	88	12%	88%
8	11	89	11%	89%
9	11	89	11%	89%
10	0	100	0%	100%

Abaixo encontra-se o gráfico para que possamos visualizar melhor as percentagens de patches em que o Bug é detetado:

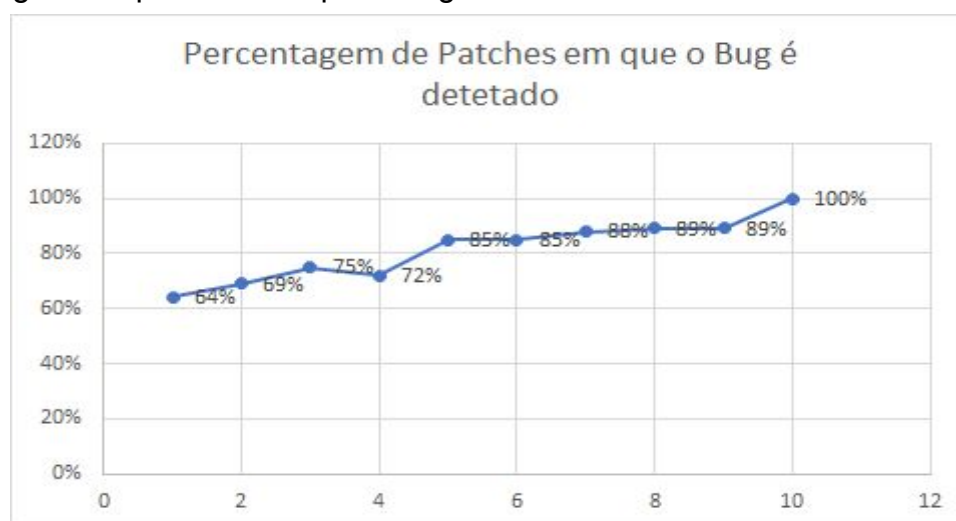


Fig 1. Percentagem de patches em que o bug é detetado por caso de teste

De acordo com o gráfico, a percentagem de patches em que o bug é detetado aumenta consoante o aumento de complexidade dos casos de teste. Verifica-se uma deteção total de todos os bugs no caso de teste nº 10. Embora seja estranho que haja uma deteção de 100% dos bugs, é explicável na medida em que estamos a utilizar um programa relativamente pequeno (157 linhas).

Desenvolvemos também uma tabela com os Tipos de Bugs que não foram detetados para cada Caso de Teste:

Caso de Teste	MIA	MIFS	MLPA	WPFV	Percentagem (MIA)	Percentagem (MIFS)	Percentagem (MLPA)	Percentagem (WPFV)
1	8	7	18	3	22%	19%	50%	9%
2	7	7	15	2	23%	23%	48%	6%
3	5	6	12	2	20%	24%	48%	8%
4	5	6	11	1	22%	26%	48%	4%
5	2	5	7	1	13%	33%	47%	7%
6	2	5	7	1	13%	33%	47%	7%
7	2	5	5	0	16%	42%	42%	0%
8	2	5	4	0	18%	45%	37%	0%
9	2	5	4	0	18%	45%	37%	0%
10	0	0	0	0	0%	0%	0%	0%

Colocando estas percentagens num gráfico, obtemos a seguinte figura:

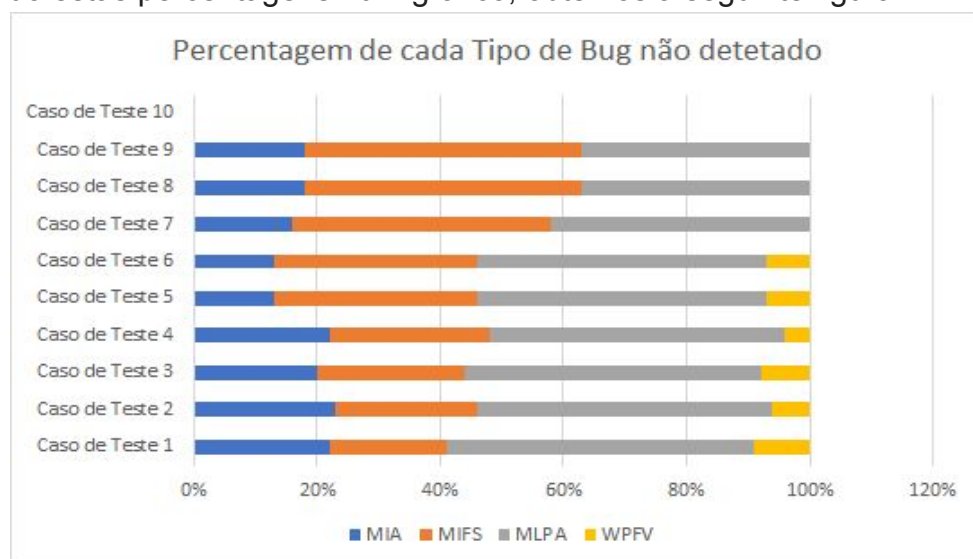


Fig 2. Percentagem de cada tipo de bug detetado em cada caso de teste

Pela observação deste gráfico conclui-se que o bug mais difícil de detetar pelos programas é o MLPA (quando é perdida uma parte localizada do algoritmo). Contudo, o bug mais fácil de detetar pelos nossos casos de teste é o WPFV (quando o operador de uma variável num parâmetro de uma função está errado).

Parser Calculadora

Para o caso do **Parser Calculadora** foram colocados os resultados dos testes na seguinte tabela:

Nº do Caso de Teste	Nº de Patches onde o Bug não é detetado	Nº de Patches onde o Bug é detetado	% de Patches onde o Bug não é detetado	% de Patches onde o Bug é detetado
1	199	1	99.5%	0.05%
2	181	19	90.5%	9.5%
3	179	21	89.5%	10.5%
4	150	50	75%	25%
5	130	70	65%	35%
6	119	81	59.5%	40.5%
7	108	92	54%	46%
8	95	105	47.5%	52.5%
9	72	128	36%	64%
10	67	133	33.5%	66.5%
11	60	140	30%	70%
12	60	140	30%	70%
13	59	141	29.5%	70.5%

Abaixo encontra-se o gráfico para que possamos visualizar melhor as percentagens de patches em que o Bug é detetado.

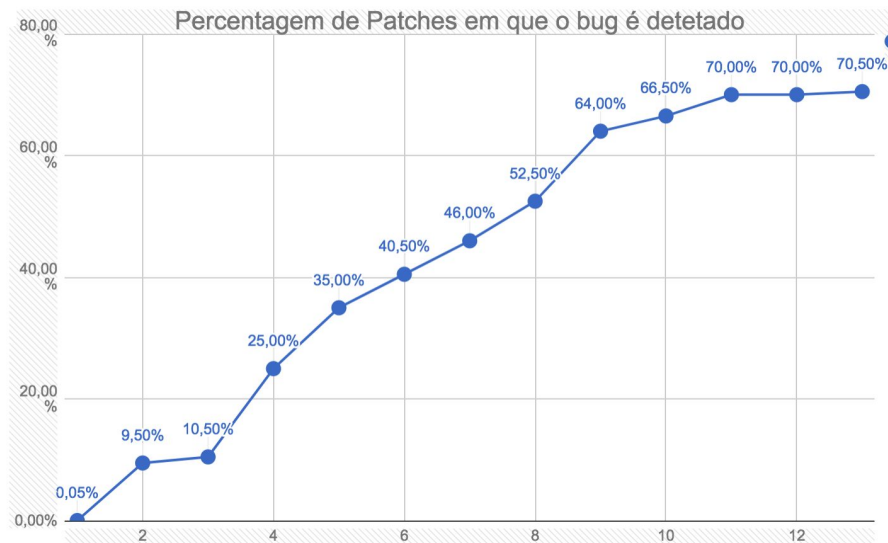


Fig 3. Percentagem de patches em que o bug é detetado por caso de teste

A percentagem de patches em que o bug é detetado aumenta consoante o aumento de complexidade dos casos de teste. No caso testado acima nunca chegamos a atingir um valor de 100% em termos de percentagem de deteção de bugs (ao contrário do primeiro caso). O que acontece é uma estabilização do valor da percentagem à medida que vamos subindo a complexidade dos casos de teste.

Tabela com os Tipos de Bugs que não foram detetados para cada Caso de Teste:

Caso de Teste	MIA	MIFS	MIEB	MLPA	Percentagem (MIA)	Percentagem (MIFS)	Percentagem (MLPA)	Percentagem (WPFV)
1	25	21	5	148	13%	11%	3%	73%
2	25	21	5	130	14%	12%	8%	66%
3	25	21	5	128	14%	12%	3%	71%
4	21	21	3	105	14%	14%	2%	70%
5	15	18	3	94	12%	14%	2%	72%
6	14	18	3	84	12%	15%	2%	71%
7	12	13	3	80	11%	12%	3%	74%
8	7	12	3	73	7%	13%	3%	77%

9	6	12	3	51	8%	17%	4%	71%
10	6	12	3	46	9%	18%	4%	69%
11	6	5	3	46	10%	8%	5%	77%
12	6	5	3	46	10%	8%	5%	77%
13	6	5	3	45	10%	8%	5%	76%

Colocando estas percentagens num gráfico, obtemos a seguinte figura:

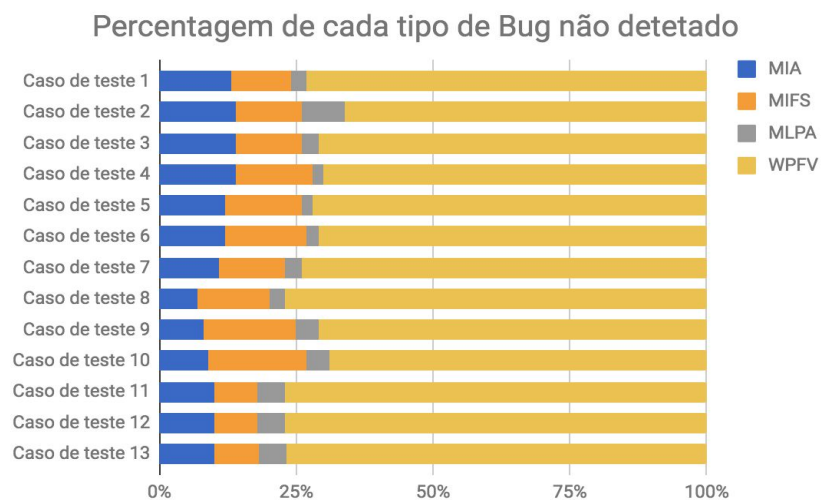


Fig 4. Percentagem de cada tipo de bug detetado em cada caso de teste.

Da observação deste gráfico verifica-se que o tipo de bug mais difícil de detetar pelo programas é o WPFV (quando o operador de uma variável num parâmetro de uma função está errado). Em contrapartida, aquele bug mais fácil de detetar pelos nossos casos de teste é o MLPA.

Análise e Teste de Hipóteses

Travelling Salesman

- **n = 10**

Dado que n é inferior a 30, vamos recorrer ao Teste T para verificar a veracidade das hipóteses, sabemos que:

- **grau de confiança = 0.95**
- **$\alpha = 0.05$**
- **df = n-1 = 9**
- **Média = 82.1%**
- **Desvio Padrão = $\sqrt{\frac{\sum |x - \bar{x}|^2}{n}} = 10.8$**

Indo à tabela T buscar o valor correspondente para este df e este alpha, temos que **t = 1.833**.

- **$t_c = \frac{\bar{x} - \mu}{s/\sqrt{n}} = (82.1 - 80)/(10.8/\sqrt{10}) = 0.61488$**

Conclusão: Recorrendo de novo à tabela T para ver aonde está este valor de t_c concluímos que $p > 0.25$. Como $t_c < t$ então não podemos rejeitar H_0 , isto é, não podemos provar que a probabilidade de um conjunto de testes conseguir detetar falhas no código fonte de um software não é inferior a 0.8.

Parser Calculadora

- $n = 13$

Dado que n é inferior a 30, vamos recorrer ao Teste T para verificar a veracidade das hipóteses, sabemos que:

- grau de confiança = 0.95
- $\alpha = 0.05$
- $df = n-1 = 12$
- Média = 43.1%
- Desvio Padrão = $\sqrt{\frac{\sum |x - \bar{x}|^2}{n}} = 25.37$

Indo à tabela T buscar o valor correspondente para este df e este α , temos que $t = 1.782$.

- $t_c = \frac{\bar{x} - \mu}{s / \sqrt{n}} = (43.1 - 80) / (25.37 / \sqrt{13}) = -5.24418$

Conclusão: Recorrendo de novo à tabela T para ver aonde está este valor de t_c concluímos que $p > 0.9995$. Como $t_c < t$ então não podemos rejeitar H_0 , isto é, não podemos provar que a probabilidade de um conjunto de testes conseguir detetar falhas no código fonte de um software não é inferior a 0.8.

Conclusão

Depois de toda esta análise podemos concluir que é um problema real o facto de se desenvolverem casos de teste que não testam a fundo o nosso programa, isto é, que não detectam bugs no código fonte. Claro que neste estudo apenas dois programas foram analisados. No entanto foi o suficiente para concluir que de forma alguma podemos provar que, por exemplo, a probabilidade de bugs serem detectados por um conjunto de testes de software é inferior a 80%.

Depois deste trabalho aprendemos a planear e desenhar uma experiência de forma a responder a uma questão. Desde a identificação de variáveis (dependentes e independentes), até à análise dos dados extraídos e teste de hipóteses.

Como futuros engenheiros, sabemos que o aprofundar dos conhecimentos no desenho de experiências é essencial para resolver e abordar diversos problemas e não somente dominar competências técnicas de programação

Glossário

- **DF** - Degree of Freedom
- **MFC** - Missing Function Call
- **MVIV** - Missing Variable Initialization with a Value
- **MVAV** - Missing Variable Assignment with a Value
- **MVAE** - Missing Variable Assignment with an Expression
- **MIA** - Missing If Around statements
- **MIFS** - Missing IF construct and surrounded Statements
- **MIEB** - Missing IF construct plus statements plus else before statements
- **MLAC** - Missing “and sub-expression” in logical expression used in branch condition
- **MLOC** - Missing “or sub-expression” in logical expression used in branch condition
- **MLPA** - Missing Localized Part of the Algorithm
- **WVAV** - Wrong Value Assigned to a Variable
- **WPFV** - Wrong variable in parameter of function Call
- **WAEP** - Wrong Arithmetic Expression in a function Parameter

Referências

- Slides de Metodologias Experimentais de Informática da Universidade de Coimbra
- Calculadora de Desvio Padrão: <http://www.calculator.net/standard-deviation-calculator.html>
- Software de Injeção de Bugs: <https://ucxception.dei.uc.pt/>

Anexo A

```
2  
-1 47  
26 44  
2  
13 -29  
42 0
```

Fig. 5: Caso de Teste 1 para o programa Travelling Salesman

```
2  
-1 47  
26 44  
2  
13 -29  
42 0
```

Fig. 6: Caso de Teste 2 para o programa Travelling Salesman

```
2  
-15 -31  
17 -9  
2  
16 26  
-5 41  
3  
16 26 -5  
41 -11 -9  
41 -16 -8
```

Fig. 7: Caso de Teste 3 para o programa Travelling Salesman

```

2
2 -12
13 30
3
2 -12 13
30 -50 -16
7 36 25
3
25 22 2
-2 28 -3
22 33 10
3
-38 20 -3
-12 3 42
-4 46 3

```

Fig. 8: Caso de Teste 4 para o programa Travelling Salesman

```

4
26 -29 11 -25
2 44 -41 -34
-39 15 -27 -9
38 -12 -25 31
4
25 -16 28 -20
14 4 -15 -35
-12 -50 -45 -12
-36 -40 1 -34
4
25 -16 28 -20
14 4 -15 -35
-12 -50 -45 -12
-36 -40 1 -34
4
25 -16 28 -20
14 4 -15 -35
-12 -50 -45 -12
-36 -40 1 -34
4
28 16 13 38
-27 42 -24 -32
28 23 3 -29
42 33 -49 7
3
-25 -33 4
35 24 10
26 29 -29

```

Fig. 9: Caso de Teste 5 para o programa Travelling Salesman

```

3
-24 20 -31
-22 -24 -23
19 30 -35
4
-24 20 -31 -22
-24 -23 19 30
-35 15 16 27
38 5 41 30
5
36 -31 -12 34 -19
-1 18 5 -15 37
1 -24 -3 7 -10
36 -8 -16 -46 16
-40 -20 19 33 -5
3
24 24 -30
34 -38 9
-9 -12 4
4
-44 26 -35 -33
-9 -30 39 28
36 42 46 17
17 -39 47 34
5
5 22 37 16 -7
-13 -16 33 -28 -49
33 -19 -39 -5 -45
-6 -24 -49 -7 38
0 -9 35 47 -37
4
-12 -20 33 -4
-50 26 -42 12
11 -11 -45 11
-44 37 -31 -34
6
-46 -37 -10 46 -45 29
29 -24 -19 13 -42 12
-9 20 -7 -39 24 37
18 -8 37 34 2 -47
30 -20 -36 37 -42 -15
-41 14 -2 -1 -38 3

```

Fig. 10: Caso de Teste 6 para o programa Travelling Salesman


```

4
28 -41 25 -34
-12 38 -16 10
-8 14 1 -33
40 -27 -20 -40
4
-3 1 36 -24
37 -15 16 23
-28 -31 2 -8
-32 -40 -5 -3
4
6 16 36 47
25 18 -22 39
43 46 4 -16
-20 -33 3 -45
4
6 16 36 47
25 18 -22 39
43 46 4 -16
-20 -33 3 -45
4
0 22 36 -15
-23 -46 11 31
33 -23 -33 -16
-41 -2 24 -4
5
-19 -31 -44 9 -9
-34 -20 -43 -7 -12
-45 -12 -15 22 23
44 48 -41 -44 27
12 7 -45 14 -50
5
-19 -31 -44 9 -9
-34 -20 -43 -7 -12
-45 -12 -15 22 23
44 48 -41 -44 27
12 7 -45 14 -50
5
30 48 -26 -11 -6
30 -26 -33 28 49
-21 26 28 -41 -33
-19 -42 39 37 39
-25 -40 43 36 -37
6
49 28 46 -23 -28 31
-32 -27 -16 -43 42 11
-36 -22 -17 -40 38 0
-40 41 -2 35 -27 8
5 -45 24 27 -11 40
16 40 18 -35 19 -8

```

Fig. 11: Caso de Teste 7 para o programa Travelling Salesman

```

5
-44 3 -2 28 -18
13 -10 44 -8 -12
-3 -3 -33 -25 35
-38 21 36 -34 -37
-36 6 43 7 35
5
42 -5 -11 0 27
42 24 -31 31 8
-2 42 -17 19 44
0 -3 6 -7 -19
-48 -21 -46 2 -18
5
42 -5 -11 0 27
42 24 -31 31 8
-2 42 -17 19 44
0 -3 6 -7 -19
-48 -21 -46 2 -18
5
23 16 -38 -22 -19
-36 -8 -24 -17 -1
35 27 0 -27 17
4 18 -46 -45 -19
-7 29 -6 20 39
5
23 16 -38 -22 -19
-36 -8 -24 -17 -1
35 27 0 -27 17
4 18 -46 -45 -19
-7 29 -6 20 39
5
-40 5 7 -7 8
-12 -5 43 43 -33
-38 -46 -3 18 -41
-23 -32 21 22 -23
0 5 -44 -13 48

```

```

5
9 49 -7 47 7
18 -33 -13 11 1
-30 35 0 -29 -2
-33 -29 -16 -27 33
-35 -13 -9 35 43
6
39 -41 32 -4 14 35
1 -20 -19 46 -6 17
-40 -12 19 10 4 -14
17 -34 -22 26 15 8
-42 26 -18 -17 23 34
-29 -35 43 -47 -37 8
6
39 -41 32 -4 14 35
1 -20 -19 46 -6 17
-40 -12 19 10 4 -14
17 -34 -22 26 15 8
-42 26 -18 -17 23 34
-29 -35 43 -47 -37 8
6
-40 36 45 46 -16 -49
3 21 18 -32 40 29
42 -11 -45 -40 36 -27
42 47 -15 -18 -35 47
1 23 8 -19 -9 0
-10 -47 -14 -14 2 21

```

Fig. 12: Caso de Teste 8 para o programa Travelling Salesman

```

5
30 20 2 10 16
38 -20 37 -25 -3
13 25 6 -2 35
-14 38 47 -21 -40
-44 26 24 40 -42
6
48 16 -17 16 -42 -40
48 24 -27 4 35 23
-28 35 7 23 -41 4
19 31 32 -42 -28 -23
35 48 43 29 -29 -10
9 -30 -43 -5 -12 17
6
-8 -29 8 8 11 39
-27 45 -17 39 33 38
41 5 -41 -20 9 -37
-6 -41 1 18 21 -49
-32 -5 4 -10 -2 44
42 40 -34 0 -1 27
6
-8 -29 8 8 11 39
-27 45 -17 39 33 38
41 5 -41 -20 9 -37
-6 -41 1 18 21 -49
-32 -5 4 -10 -2 44
42 40 -34 0 -1 27
6
20 7 -50 1 18 -26
-24 33 29 -48 -38 46
-20 -49 7 37 7 13
-37 5 -46 39 31 -29
-42 46 -22 -36 11 -40
-10 33 -30 -10 37 38

```

```

6
20 7 -50 1 18 -26
-24 33 29 -48 -38 46
-20 -49 7 37 7 13
-37 5 -46 39 31 -29
-42 46 -22 -36 11 -40
-10 33 -30 -10 37 38
6
-38 -41 -8 40 35 -45
33 -44 -12 24 -38 -20
3 -35 -27 -47 6 -20
-33 26 30 -5 -4 -18
30 -50 -39 28 1 -8
-14 13 -46 -20 3 39
6
-38 -41 -8 40 35 -45
33 -44 -12 24 -38 -20
3 -35 -27 -47 6 -20
-33 26 30 -5 -4 -18
30 -50 -39 28 1 -8
-14 13 -46 -20 3 39
6
-45 -48 14 -18 13 5
-46 -5 49 15 18 21
-22 0 34 -22 3 44
29 -27 19 -48 15 22
-33 28 1 32 -29 -27
7 28 -25 23 -40 38

```



```

6
-39 -34 18 13 34 -32
-31 -49 -38 24 -18 -40
39 19 -41 -14 38 -31
41 16 4 25 -48 44
-33 29 -28 17 41 37
-15 -48 6 5 -33 -10
6
-14 3 7 19 -43 -17
-8 24 44 -9 -39 -42
-39 -37 8 -15 33 -48
6 -46 -32 -50 0 48
-5 28 -33 16 5 19
-39 -7 -28 18 -37 32
6
-14 3 7 19 -43 -17
-8 24 44 -9 -39 -42
-39 -37 8 -15 33 -48
6 -46 -32 -50 0 48
-5 28 -33 16 5 19
-39 -7 -28 18 -37 32
6
36 -5 41 -30 22 5
27 -3 -45 0 -36 -13
-21 1 2 -11 -24 47
-41 -25 16 48 8 20
42 30 -20 -33 35 -4
-3 -26 -7 41 -6 -35
6
-38 -22 14 14 7 31
-50 43 43 -47 0 18
6 8 27 43 -8 -46
46 23 3 -42 27 -31
-35 23 12 -23 -43 -23
-1 22 6 15 -14 15

```

Fig 13: Caso de Teste 9 para o programa Travelling Salesman

```

5
4 11 44 12 -27
38 -35 48 18 -44
30 -27 -18 42 -36
-44 3 37 24 21
-6 -30 -3 16 -21
5
4 11 44 12 -27
38 -35 48 18 -44
30 -27 -18 42 -36
-44 3 37 24 21
-6 -30 -3 16 -21
5 Experiment Design
33 17 37 -19 -19
-8 42 22 23 -7
-45 -3 -44 -14 40
-3 24 38 43 41
-13 17 27 45 17
5
33 17 37 -19 -19
-8 42 22 23 -7
-45 -3 -44 -14 40
-3 24 38 43 41
-13 17 27 45 17
5
16 40 -11 25 -8
22 39 -3 -50 31
-24 -38 25 45 31
-9 13 30 -33 -19
11 -22 -41 -10 -6
6 (2 + 3) * 6
10 -30 41 21 -19 28
12 43 33 7 41 -15
42 13 23 -3 29 1
23 30 -21 -45 -22 5
48 38 -30 -11 19 25
-5 -21 45 39 3 28

```

```

6
-31 11 29 37 29 13
-26 45 47 29 12 5
3 21 30 -7 -24 6
-13 25 31 -9 -7 45
-12 -50 -46 -1 33 26
36 -48 -12 16 -9 -33
6 Programa 1: Travelling
-31 11 29 37 29 13
-26 45 47 29 12 5
3 21 30 -7 -24 6
-13 25 31 -9 -7 45
-12 -50 -46 -1 33 26
36 -48 -12 16 -9 -33

```

```

6
43 -46 6 -37 44 24
27 33 -17 42 -49 -41
44 5 43 -13 -16 -33
-39 44 30 -24 12 -32
-47 -27 -5 16 -32 -15
-41 -39 -11 -33 26 35
6 Definição Problema
43 -46 6 -37 44 24
27 33 -17 42 -49 -41
44 5 43 -13 -16 -33
-39 44 30 -24 12 -32
-47 -27 -5 16 -32 -15
-41 -39 -11 -33 26 35
6
43 -46 6 -37 44 24
27 33 -17 42 -49 -41
44 5 43 -13 -16 -33
-39 44 30 -24 12 -32
-47 -27 -5 16 -32 -15
-41 -39 -11 -33 26 35

```



```

6
-17 -46 41 23 -31 -17
-38 -17 -29 -13 4 -28
-43 -25 -49 -29 1 48
8 45 45 -16 15 -15
3 -41 -44 16 -27 35
-42 6 -9 2 -21 10
6
-17 -46 41 23 -31 -17
-38 -17 -29 -13 4 -28
-43 -25 -49 -29 1 48
8 45 45 -16 15 -15
3 -41 -44 16 -27 35
-42 6 -9 2 -21 10
6
1 -24 -10 -35 -46 -32
-14 38 -22 -10 -1 -17
-26 25 -30 -28 -4 17
-49 -38 37 -4 -10 44
11 44 -7 -22 14 -9
-46 -35 18 46 -20 -26
6
1 -24 -10 -35 -46 -32
-14 38 -22 -10 -1 -17
-26 25 -30 -28 -4 17
-49 -38 37 -4 -10 44
11 44 -7 -22 14 -9
-46 -35 18 46 -20 -26

```

Fig. 14: Caso de Teste 10 para o programa Travelling Salesman

Anexo B



Fig. 15: Caso de Teste 1 para o programa Parser Calculadora

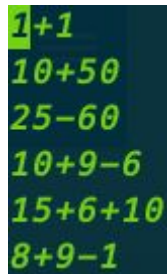


Fig. 16: Caso de Teste 2 para o programa Parser Calculadora

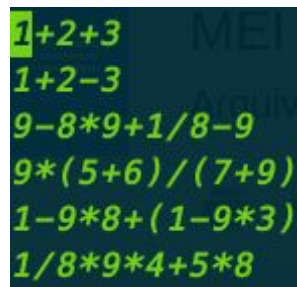


Fig. 17: Caso de Teste 3 para o programa Parser Calculadora

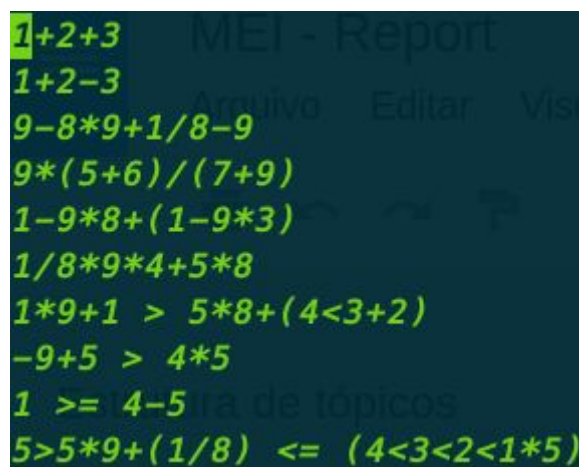


Fig. 18: Caso de Teste 4 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))

```

Fig. 19: Caso de Teste 5 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))

```

Fig. 20: Caso de Teste 6 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))

```

Fig. 21: Caso de Teste 7 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))
sin(5<0)
cos(4>1) + sin(4>1)
tan(log10(5)) + cos(sin(5))

```

Fig. 22: Caso de Teste 8 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))
sin(5<0)
cos(4>1) + sin(4>1)
tan(log10(5)) + cos(sin(5))
a=42, a/=7
b=10, b-=5
c=log10(5*10)

```

Fig. 23: Caso de Teste 9 para o programa Parser Calculadora


```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))
sin(5<0)
cos(4>1) + sin(4>1)
tan(log10(5)) + cos(sin(5))
a=42, a/=7
b=10, b-=5
c=log10(5*10)
a+5 > b-9 || a==5
c!=1 && b=20 || a>=0

```

Fig. 24: Caso de Teste 10 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))
sin(5<0)
cos(4>1) + sin(4>1)
tan(log10(5)) + cos(sin(5))
a=42, a/=7
b=10, b-=5
c=log10(5*10)
int(1.2)

```

Fig. 25: Caso de Teste 11 para o programa Parser Calculadora

```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 4-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))
sin(5<0)
cos(4>1) + sin(4>1)
tan(log10(5)) + cos(sin(5))
a=42, a/=7
b=10, b-=5
c=log10(5*10)
int(1.2)
if (a > 5, 22, 33)
if (a+5 == int(-1.95), log10(10), -5)

```

Fig. 26: Caso de Teste 12 para o programa Parser Calculadora


```

1+2+3
1+2-3
9-8*9+1/8-9
9*(5+6)/(7+9)
1-9*8+(1-9*3)
1/8*9*4+5*8
1*9+1 > 5*8+(4<3+2)
-9+5 > 4*5
1 >= 14-5
5>5*9+(1/8) <= (4<3<2<1*5)
log10 (4*5-(5+6))
log10 (1)*log10(9/2+5>1*3)
1*5 > (log10 (5--9))
sqrt (4+5*6)
sqrt((15>5)+8*3) * 5+log10( sqrt(9))
pow(13*8,5)-pow(15*2,(1>5)+6)
5+log10(pow(2,3))
sin(5<0)
cos(4>1) + sin(4>1)
tan(log10(5)) + cos(sin(5))
a=42, a/=7
b=10, b-=5
c=log10(5*10)
int(1.2)
a=42, a/=7
b=10, b-=5
c=log10(5*10)
if(a<pow(2*sqrt(log10(a>5*9)),2), int(9.1), b)
d=a*c+0/8
d+=if(a==6,22,23)
int(log10(10+d*c-a)+ (a>b || c!= 0)+int(2.3))
a <= b || c!=0
a == 10 - sin(tan(cos(log10(sqrt(5)))))
e=a+b+c-d/5+b

```

Fig. 27: Caso de Teste 13 para o programa Parser Calculadora