UNIVERSIDADE D COIMBRA

Qualidade e Confiabilidade de Software

Trabalho Prático #2

Mestrado em Segurança Informática/Mestrado em Engenharia Informática

2020/2021

Trabalho realizado por:

Carolina de Castilho Godinho – 2017247087

Maria Paula de Alencar Viegas – 2017125592

Maria Carolina Jordão Pereira Gonçalves – 2017247573

Índice

1. Introdução

Neste trabalho temos como objetivo desenvolver um plano de teste de software e realizar uma campanha de testes para um código de software selecionado por nós. Para tal, foi necessário identificar e selecionar as abordagens de teste mais adequadas consoante o nosso software. Para além disso, o foco deste trabalho são as abordagens de teste dinâmicos, sendo que os testes estáticos estão fora do escopo do trabalho.

O software selecionado é o algoritmo Convex Hull (Algoritmo de Jarvi) desenvolvido em Python que se encontra em anexo a este documento. Para o plano de teste temos duas abordagens principais: *White Box Testing* e *Black Box Testing*.

Dentro do *White Box Testing* iremos realizar o *Control Flow* e o *Data Flow* e os respetivos casos de teste e resultados dos mesmos.

Para o *Black Box Testing* será feito o *Equivalence Class Partitioning* e *Boundary Values* e apresentados os testes realizados para o mesmo.

No final de obter todos os resultados, estes irão ser analisados e iremos tomar uma conclusão final sobre o software selecionado.

2. Problemas de risco do software

O algoritmo do Convex Hull, é um algoritmo bastante conhecido e bastante usado. Por esse mesmo motivo, leva-nos a acreditar que em princípio este não apresenta riscos demasiado elevados com uma pequena análise.

A complexidade do mesmo não é elevada, pois não inclui bibliotecas externas, nem usa aplicações externas. O próprio código encontra-se bem comentado, com uma explicação bem descritiva do procedimento de cada função deste algoritmo.

Neste algoritmo, que pode ser utilizado em diferentes aplicações ou ter diferentes cenários de uso, temos de garantir que o input que é fornecido nas diferentes funções é o correto e esperado, dado que qualquer má formatação deste, pode corromper o algoritmo ou criar algum problema numa aplicação que este esteja incluído.

Mesmo após o teste deste código/algoritmo, poderão existir bugs que nunca iremos encontrar, contudo, com os planos de teste que iremos apresentar tentaremos descobrir riscos de software que influenciem o resultado final do algoritmo ou o próprio fluxo do mesmo.

3. Itens e recursos a ser testados

Para o algoritmo escolhido, temos três funções para serem testadas: a função principal *convexHull(points, n)* e as funções que este chama, *orientation(p, q, r)* e *Left_index(points)*. Testámos não só o funcionamento dessas funções, mas também as suas variáveis e respetivas aplicações. Também testámos os parâmetros de input a serem enviados para as funções.

O algoritmo tem uma classe *Point* que guarda um valor *x* e um valor *y* associados à mesma. A classe pretende representar um ponto num sistema de coordenadas cartesiano de posição (x,y).

A função *Left_index(points)* recebe uma lista de pontos *Point* e retorna a posição na lista do ponto *Point* mais à esquerda. Já a função *orientation(p, q, r)* recebe três pontos e retorna indicadores para a orientação desses três pontos (1 se for sentido horário, 2 se for sentido anti-horário, 0 se forem colineares).

Finalmente, a função *convexHull(points, n)* recebe uma lista de pontos e um inteiro n com a quantidade de pontos na lista. Esta exige que a lista tenha no mínimo três pontos e, consequentemente, n seja no mínimo três também. Esta funciona escolhendo o ponto com o menor ângulo, guardando-o, e repetindo esse processo para esse ponto. O resultado dessa função é um *print* do menor conjunto de pontos *Point* que formam um fecho convexo. Vale realçar que alteramos o final da função *convexHull* para os testes de *Data Flow* para que esta retorne os pontos em vez do print do resultado.

4. Itens e recursos a não ser testados

A totalidade dos planos de teste definidos cobrem todas as funções do algoritmo, visto que este apenas apresenta 3 funções.

Contudo cada plano de teste não cobre as três funções. Ou seja, para os testes de *White Box*, no caso do *Control Flow* apenas iremos testar a função *Left_index*. Para o caso dos testes do *Data Flow*, apenas a função de *convex_hull* irá ser testada, visto que era a função que apresenta e gere mais variáveis.

Finalmente, para os testes de Black Box, mais especificamente os testes de Equivalência Válida e Inválida de classes, todas as funções são testadas porque todas recebem dados de entrada.

Para os testes de Limite de Valor, apenas a função *convex_hull* irá ser testada, e apenas iremos focar na variável de input *n*, dado que este é o único inteiro que possui restrições. Este n representa o número de pontos que são fornecidos e por isso mesmo não pode ser negativo. Todas as outras variáveis de input e outras funções, são passados pontos (classe *Point*), que possui dois inteiros. Um número inteiro para as coordenadas de x e outro número inteiro para as coordenadas de y. Assim, estes valores podem ser negativos, positivos, pequenos, elevados, etc.

5. Abordagem de teste

Nos nossos testes, usamos a *framework pytest* para facilitar e automatizar a validação dos mesmos. Para cada função do programa a ser avaliado - *Left_index(points)*, *orientation(p,q,r)* e *convexHull(points, n)* - usamos *raises* para verificar os erros esperados dos testes que falham e também *assertions* para verificar as métricas finais.

A análise dos casos de teste é feita por meio de representações de partes do código-fonte em tabelas, grafos e esquemas consequentes aos tipos de testes que julgamos necessários para cada função.

Apesar da linguagem Python ser normalmente muito livre com o tipo de uma variável, observámos que os inputs do algoritmo escolhido são muito restritos. Isto porque a maioria das variáveis segue uma classe *Point(x, y)* declarada no próprio código que representa um ponto de coordenadas cartesianas. Nestes casos, conseguimos testar variações de x e y porém tivemos que respeitar a classe *Point(x,y).*

5.1. White Box Testing

O *White Box Testing* é um método de teste de software que testa a estrutura interna com conhecimento prévio da mesma. Quem efetua os testes, fá-lo de modo que determinados dados de entradas percorram certos caminhos do código e verifica se os dados de saída são os esperados.

5.1.1. Control flow

```
11  def Left_index(points):
12
13      '''
14      Finding the left most point
15      '''
16      minn = 0 1
17      for i in range(1,len(points)): 2
18          if points[i].x < points[minn].x: 3
19              minn = i 4
20          elif points[i].x == points[minn].x: 5
21              if points[i].y > points[minn].y: 6
22                  minn = i 7
23      return minn 9
                    8
```

Figura 2 - Função Left_index numerada para o Control Flow

Este tipo de teste de software usa o fluxo de controlo do programa como modelo. Esta é uma estratégia de teste estrutural. Neste teste é necessária que a equipa de teste conheça toda a estrutura, design, código e implementação do software previamente.

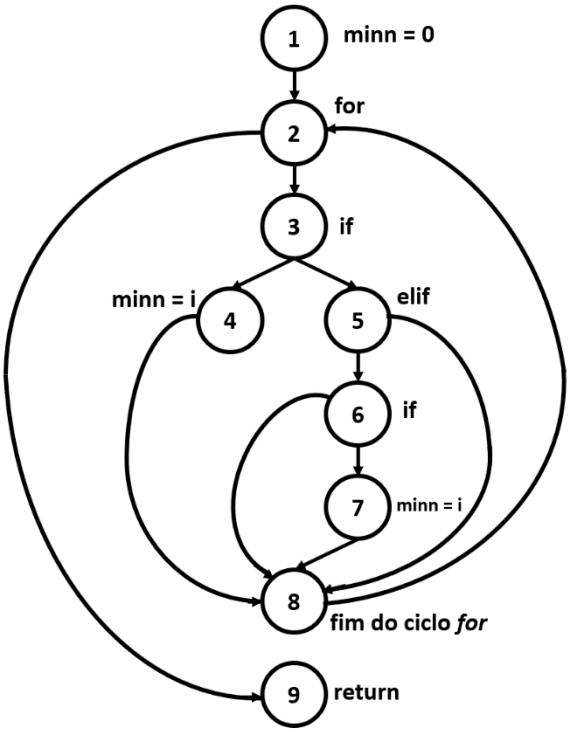

Figura 1 - Control Fow grafo para a função Left_index

Caminhos independentes:

Caminho 1: (1,2,9)

- Este caminho verifica, mas não entra no *for* e vai diretamente para o *return.*

Caminho 2: (1,2,3,4,8,2,9)

- Este caminho entra *for*, entra no *if*, faz os procedimentos do *if,* chega ao fim ciclo *for*, verifica de novo o *for* e vai para o *return.*

Caminho 3: (1,2,3,5,8,2,9)

- Este caminho entra no *for*, verifica o *if* mas não entra, verifica de seguida o *elif* e também não entra, chega ao fim ciclo *for*, verifica a condição do *for*, que não é cumprida e vai para o *return.*

Caminho 4: (1,2,3,5,6,8,2,9)

- Este caminho entra no *for*, verifica o *if* mas não entra, verifica de seguida o *elif*, verifica o if mas não entra, chega ao fim do ciclo *for*, verifica a condição do *for*, não entra no *for* e vai para o *return.*

Caminho 5: (1,2,3,5,6,7,8,2,9)

- Este caminho entra no *for*, verifica o *if* mas não entra, verifica de seguida o *elif*, entra no *elif*, entra no *if*, faz os procedimentos do *if*, chega ao fim do ciclo *for*, verifica a condição do *for*, não entra no *for* e vai para o *return.*

5.1.2. Data flow

Este teste é um teste do tipo estrutural e é usado para encontrar os caminhos de teste de um programa de acordo com os locais de definições e usos de variáveis no programa. Este teste foca-se nas declarações em que as variáveis recebem os valores e onde esses valores são usados e referenciados.

Nas Figuras 3 e 4 que se seguem abaixo podemos ver o código escolhido para a utilização do Data Flow testing e o respetivo grafo de Data Flow. O critério que vai ser utilizado no data flow testing é o ADUP.

```python
def convexHull(points, n):

    # There must be at least 3 points
    if n < 3:
        return

    # Find the leftmost point
    l = Left_index(points)

    hull = []

    '''
    Start from leftmost point, keep moving counterclockwise
    until reach the start point again. This loop runs O(h)
    times where h is number of points in result or output.
    '''
    p = l
    q = 0
    while(True):

        # Add current point to result
        hull.append(p)

        '''
        Search for a point 'q' such that orientation(p, q,
        x) is counterclockwise for all points 'x'. The idea
        is to keep track of last visited most counterclock-
        wise point in q. If any point 'i' is more counterclock-
        wise than q, then update q.
        '''
        q = (p + 1) % n

        for i in range(n):

            # If i is more counterclockwise
            # than current q, then update q
            if(orientation(points[p],points[i], points[q]) == 2):
                q = i

        '''
        Now q is the most counterclockwise with respect to p
        Set p as q for next iteration, so that q is added to
        result 'hull'
        '''
        p = q

        if(p == l):
            break

    for each in hull:
        print(points[each].x, points[each].y)
```

Figura 3 - Código para o Data Flow

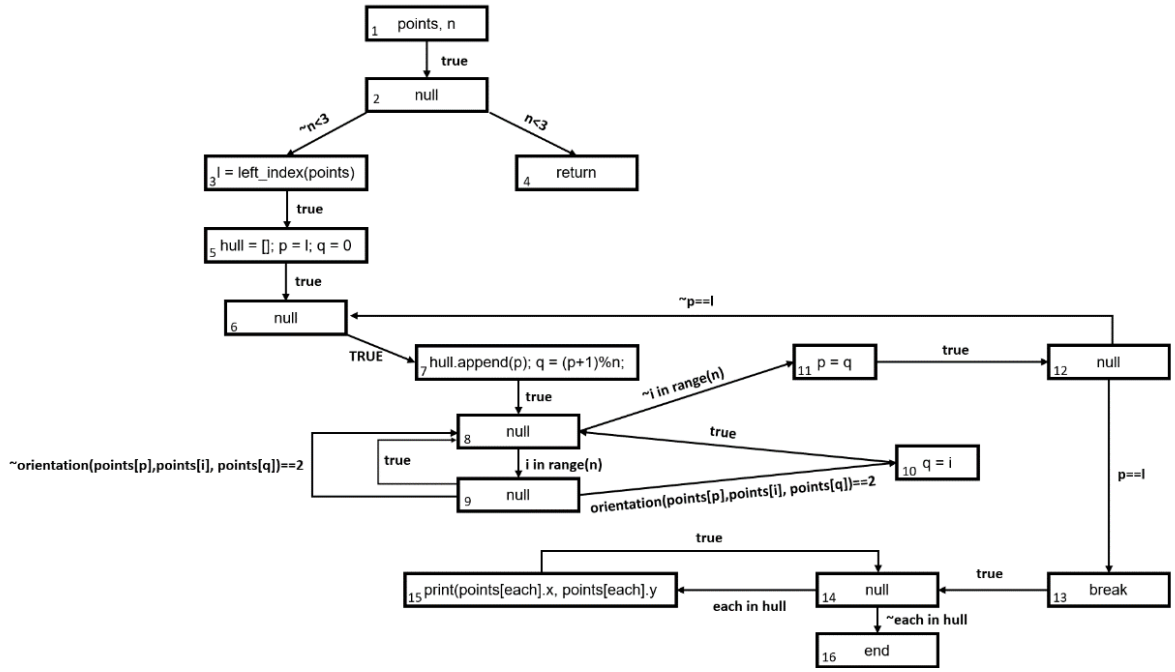

Figura 4 - Data Flow grafo

De seguida, apresentamos a tabela auxiliar para a categorização das variáveis para posteriormente selecionar os caminhos de cada variável.

Código	def	c-use	p-use
def convexHull(points, n):	points, n		
if n < 3			n
return;			
l = left_index(points)	l	points	
hull = []	hull		
p = l	P	l	
q = 0	q		
while(True):			
hull.append(p)	hull	p , hull	
q = (p + 1) % n	q	p , n	
for i in range(n):	i	l,n	n
if(orientation(points[p], points[i], points[q]) == 2):			points, p,q, i
q = i	q	i	
p = q	p	q	
if(p == l):			p,l
break			
for each in hull:	each	each, hull	hull
print(points[each].x, points[each].y)		points	

Tabela 1 - Tabela Auxiliar do Data Flow

Após a tabela feita traçamos os caminhos de cada variável traçando-os no Data Flow. Descrevemos a numeração dos caminhos e as condições que seguem para os mesmos acontecerem.

5.1.2.1. Caminhos por variável

Variável points

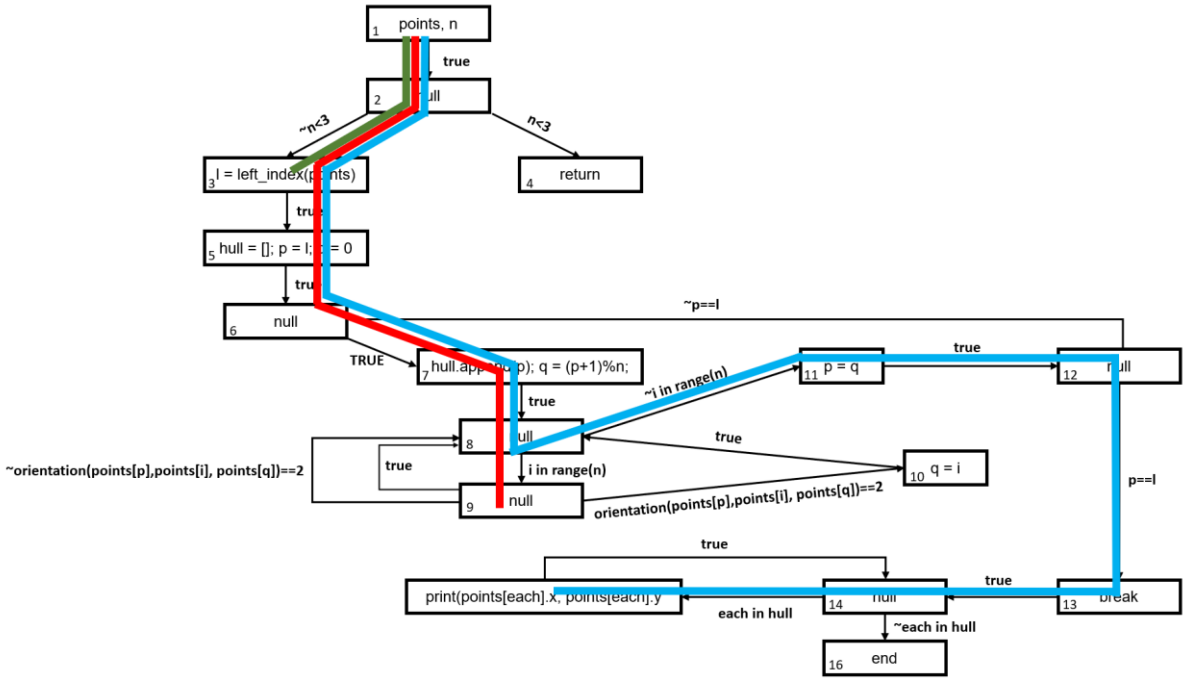

Figura 5 - Caminhos da variável points

Caminho 1: (1,2,3)
Condições:

- Condição em 2 ser falsa

Caminho 2: (1,2,3,5,6,7,8,9);
Condições:

- Condição em 2 ser falsa
- Em 8 entrar no *for*

Caminho 3: (1,2,3,5,6,7,11,12,13,14,15)
Condições:

- Condição em 2 ser falsa
- Em 9 ser *true*
- Em 14 entrar no *for*

Variável n:

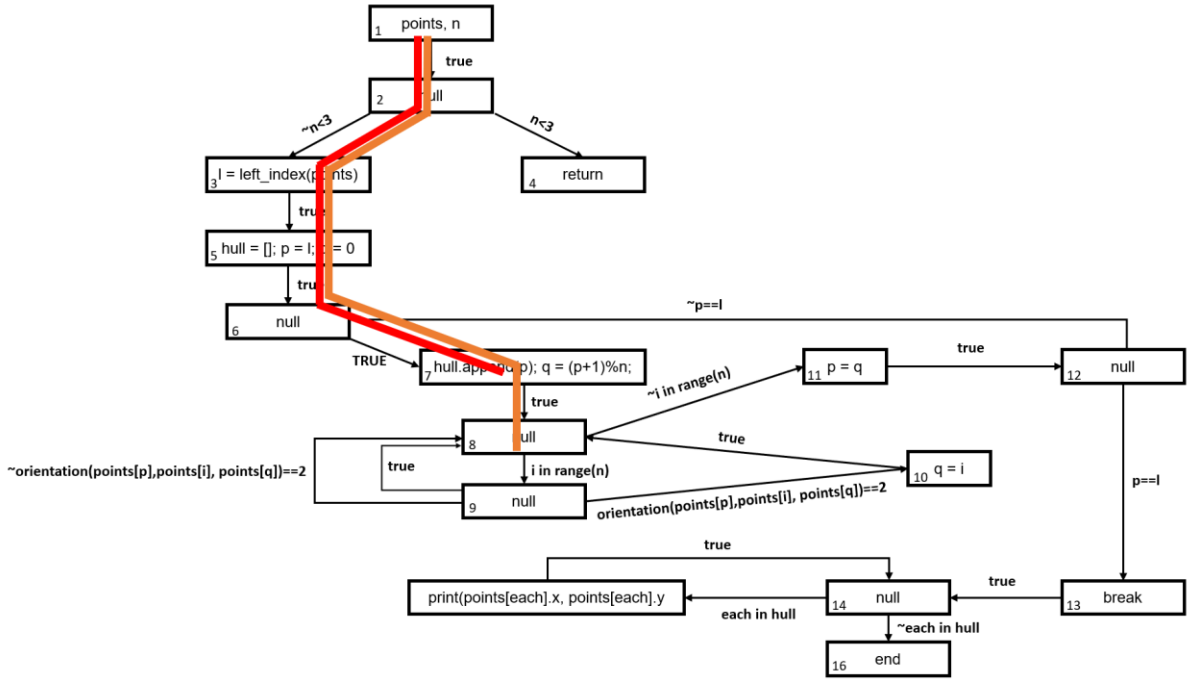

Figura 5 - Caminhos da variável n

Caminho 1: (1,2)
Condições: Não precisa

Caminho 2: (1,2,3,5,6,7)
Condições:

- Condição em 2 tem de ser falsa

Caminho 3: (1,2,3,5,6,7,8)
Condições:

- Condição em 2 tem de ser falsa

Variável l:

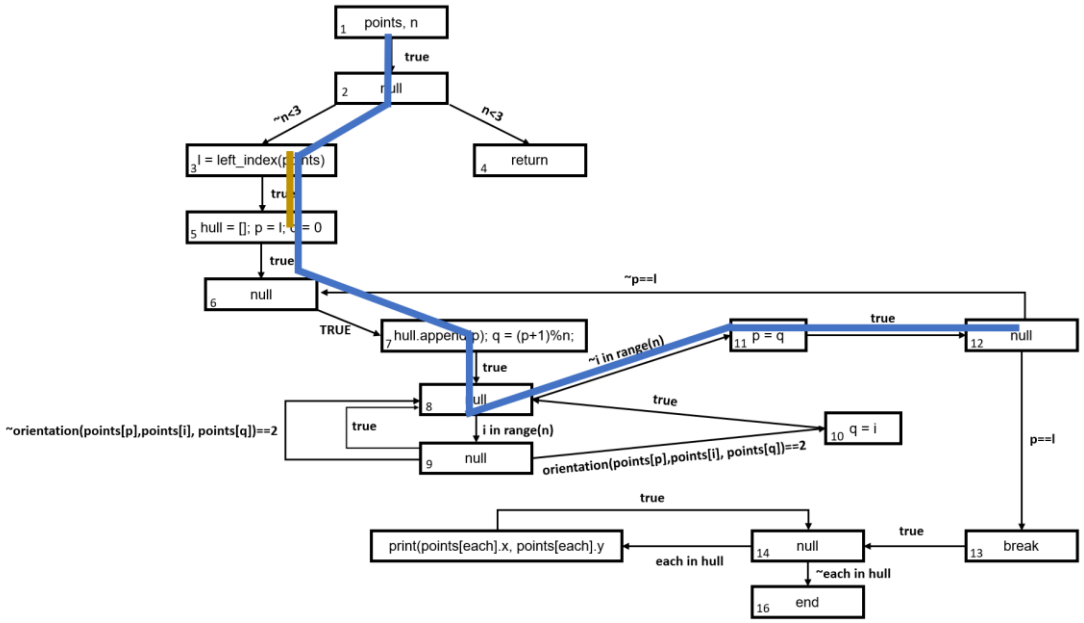

Figura 7 - Caminhos da variável l

Caminho 1: (3,5)
Condições: Não precisa

Caminho 2: (3,5,7,11,12)
Condições: Não precisa

Variável hull:

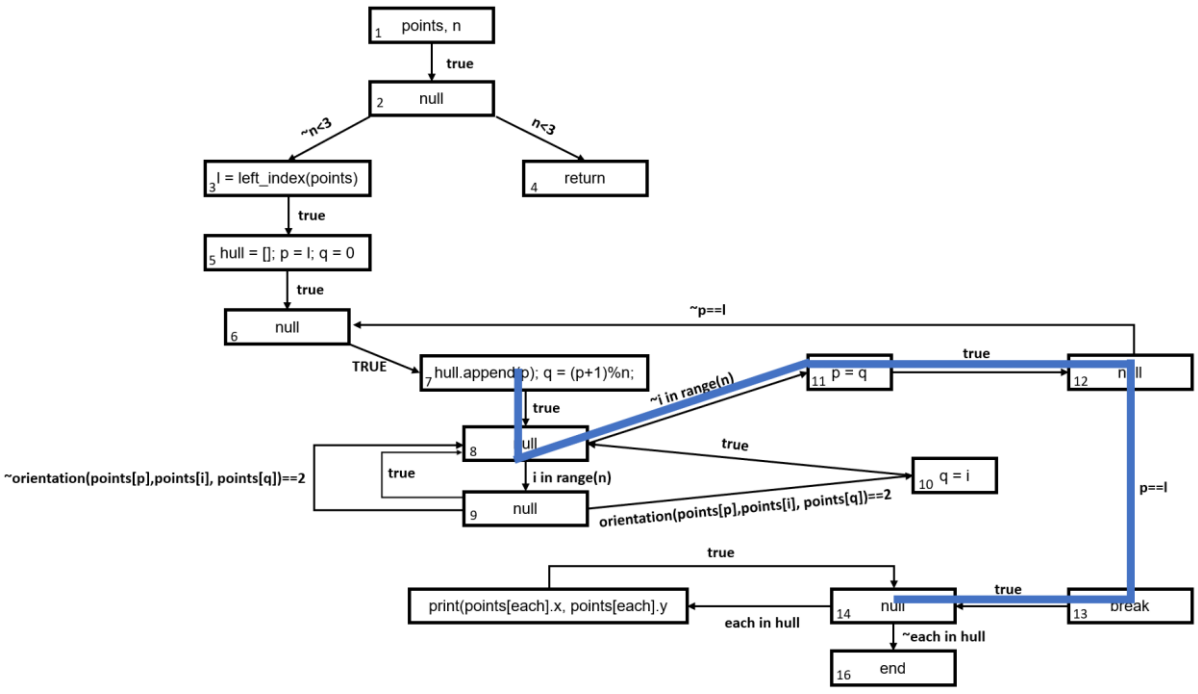

Figura 8 - Caminhos da variável hull

Caminho 1: (5,6 7,11,12,13,14)
Condições:

- 12 tem de ser *true*

Variável p:

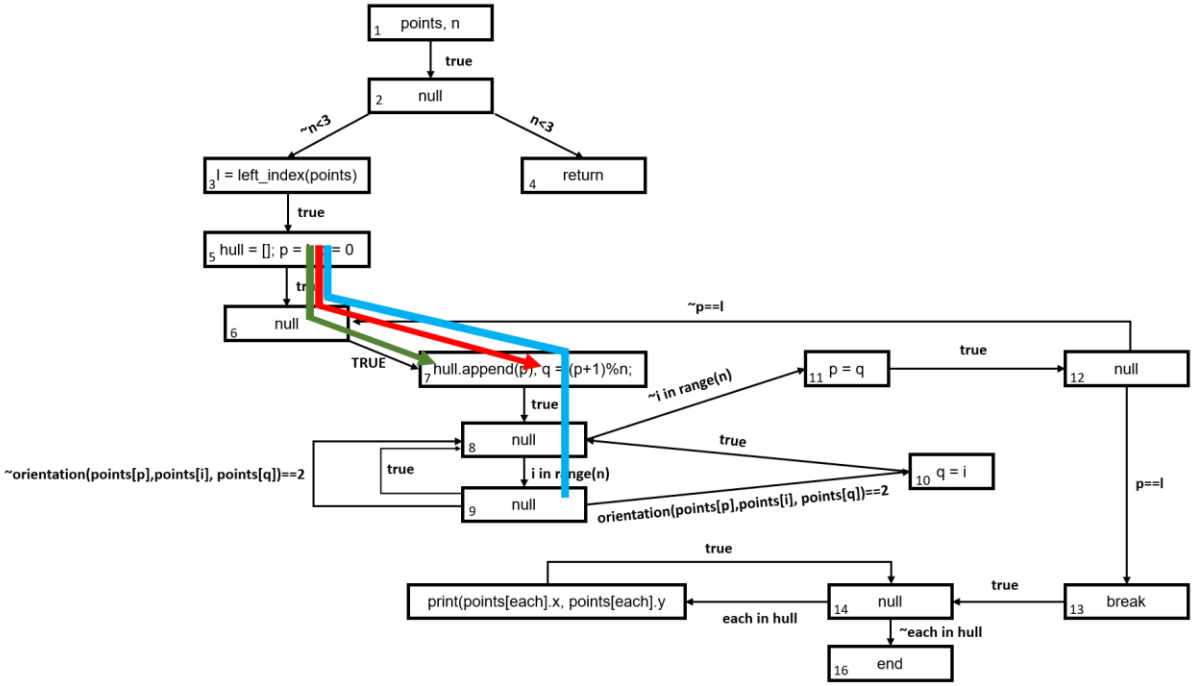

Figura 9 - Caminhos da variável p

Caminho 1: (5,6,7) [1ª declaração neste nó]
Condições: Não precisa

Caminho 2: (5, 6,7) [2ª declaração neste nó]
Condições: Não precisa

Caminho 3: (5,6,7,8,9)
Condições:

- No nó 8 tem de entrar no for

Caminho 4: (11,12)
Condições: Não precisa

Variável q:

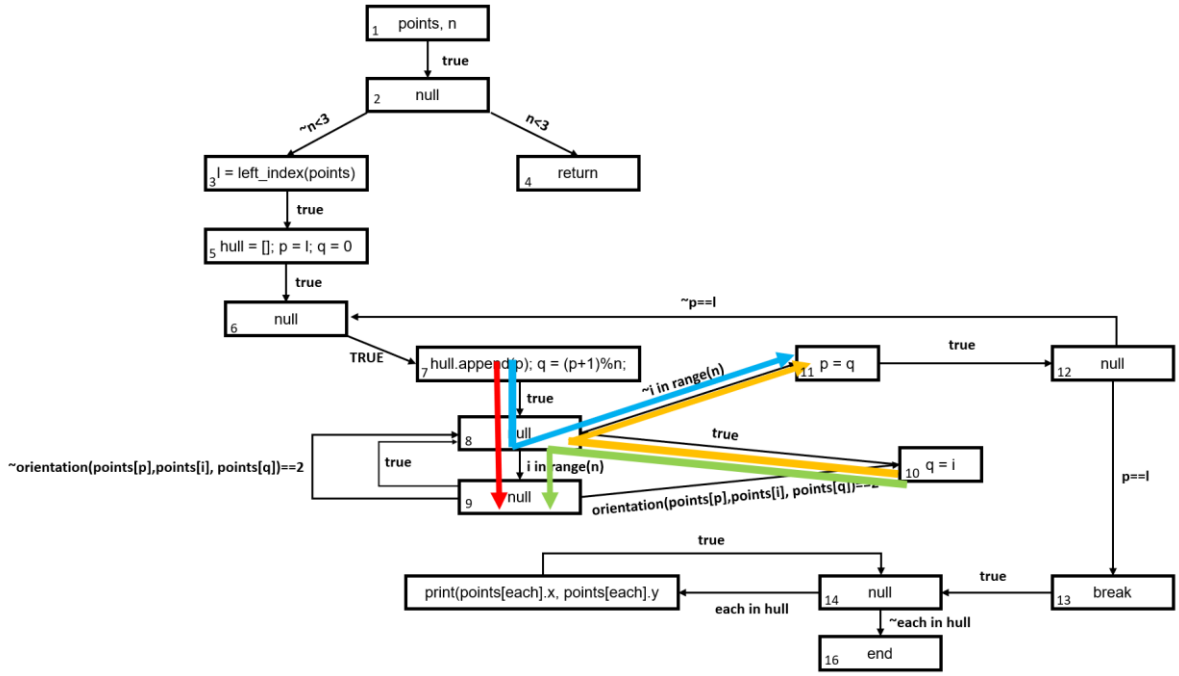

Figura 10 - Caminhos da variável q

Caminho 1: (5,6,7,8,9)
Condições:

- No nó 8 tem de entrar no *for*

Caminho 2: (5,6,7,8,11)
Condições:

- Não entra em 8 ou em 9 é falso

Caminho 3: (10,8,9)
Condições:

- Em 8 voltar a entrar no *for*

Caminho 4: (10,8,11)
Condições:

- Em 8 não entra no *for*

Variável i:

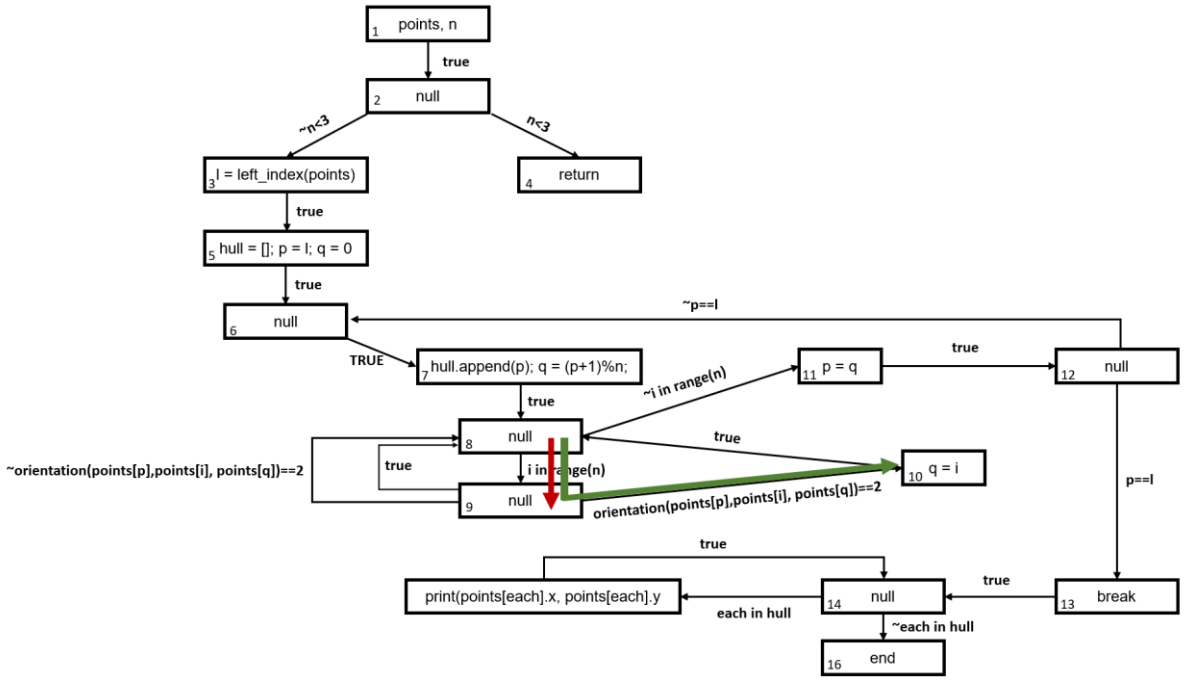

Figura 11 - Caminhos da variável i

Caminho 1: (8,9)
Condições: Não precisa

Caminho 2: (8,9,10)
Condições:

- Em 9 tem de ser true

Variável each:

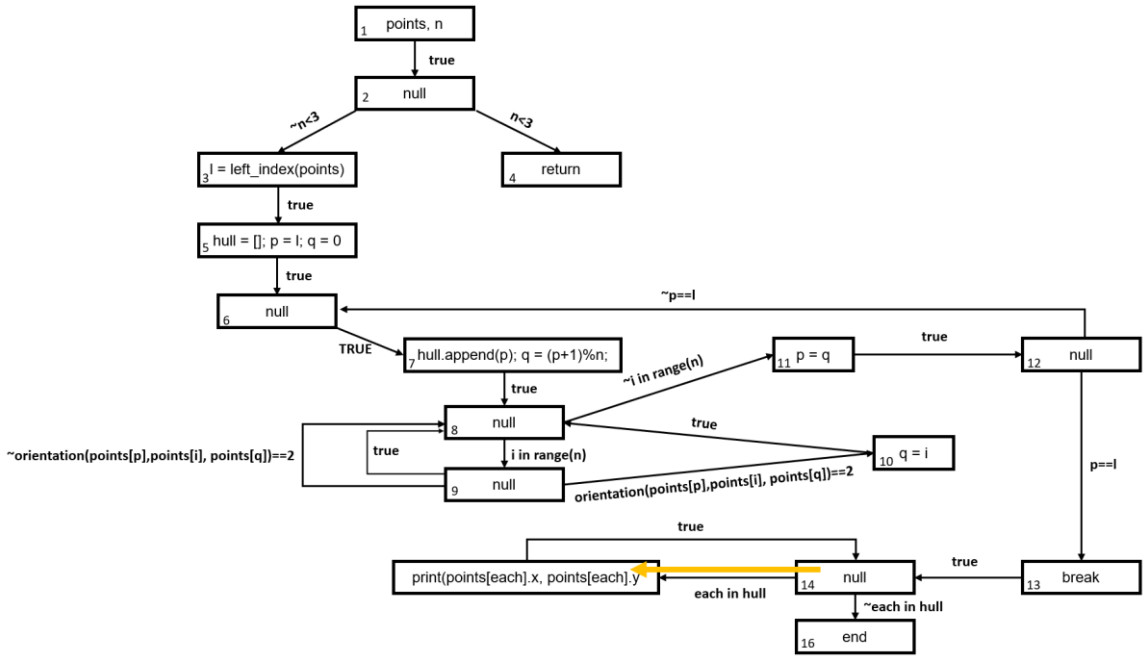

Figura 12 - Caminhos da variável each

Caminho 1: (14,15)
Condições: Não precisa

5.2. Black Box Testing

Este é um método de teste de software que examina a funcionalidade de uma aplicação sem ter em consideração as suas estruturas internas e as funcionalidades. Ou seja, este teste requer que não exista conhecimento prévio sobre o funcionamento do software.

5.2.1. Equivalence Class Partitioning

Para esta técnica as unidades de teste de dados de entrada são divididas em partições equivalentes. Estas partições podem ser usadas para derivar casos de teste. Isto faz com que se poupe tempo, devido ao pequeno número de testes. Este teste divide os dados de entrada do software em diferentes classes de dados de equivalência.

Função	Input	Válido	Inválido
Left_index(points)	*points*, Lista de *Point(x,y)*, values *x* e *y*	Lista de Point(Integer, Integer), inteiros $]-\infty,+\infty[$	Lista de inteiros, inteiros $]-\infty,+\infty[$
		Lista de Point(float, float), float $]-\infty,+\infty[$	Lista de Point(string, string), String numérica negativa
		Lista de Point(string, string), String numérica positiva ou nula	Lista (boolean, boolean)
		Lista de Point(boolean, boolean)	Lista de String não numéricas
		empty	inteiros $]-\infty,+\infty[$
		-	boolean
		-	Lista de tuplos numéricos
convexHull (points, n)	*points*, Lista de *Point(x,y)*, values *x* e *y*	Lista de Point(Integer, Integer), inteiros $]-\infty,+\infty[$	Lista de inteiros, inteiros $]-\infty,+\infty[$
		Lista de Point(float, float), float $]-\infty,+\infty[$	Lista de Point(String, String)
		Lista de Point(boolean, boolean)	Lista (boolean, boolean)
		-	Lista de String
		-	inteiros $]-\infty,+\infty[$
		-	Boolean
		-	Lista de tuplos numéricos
	n, valores inteiros, $n \geq 0$	inteiros $[0,+\infty[$	Sting
		-	Float
		-	Integer Negativo
orientation (p, q, r)	p, q e r, Point(x,y), values x e y	Point(Integer, Integer), inteiros $]-\infty,+\infty[$	Point(String, String)
		Point(float, float), float $]-\infty,+\infty[$	Listas (tuplos, inteiros, floats, strings)
		Point(boolean, boolean)	tuplos
			Inteiros ou floats $]-\infty,+\infty[$

Tabela 2 - Input Partitioning Classes

5.2.2. Boundary Values

Esta é uma técnica de teste de software onde os testes são projetados para incluir valores limites representativos dentro de um intervalo. Como temos um conjunto de vetores de teste para o sistema, pode ser definida uma topologia dentro desse conjunto.

Para o nosso código, a única função que pode apresentar variáveis com limites é a variável n na função *convexHull(points, n)*, visto que este deve ser um inteiro positivo. Essa variável guarda a quantidade de pontos passados em *points*, e portanto, ou não existe pontos ($n = 0$), ou existe uma quantidade positiva de pontos ($n > 0$).

Aproveitamos então essa restrição para fazer um teste de *Boundary Values* do Black-Box testing.

Input	Boundary Values
n , integer value n >= 0	-1, 0,1,2,3, …

6. Critérios de aprovação/reprovação dos itens

Para os critérios de aprovação/reprovação no *White* e *Black Box Testing* temos o output expetável que foi feito através da lógica do código, dos caminhos e dos grafos. Como auxílio também usamos a framework Pytest da linguagem de programação Python para fazer as asserções dos casos de teste entre o output gerado e o output esperado.

Caso o resultado expetável e o que a função devolver for o mesmo então o caso de teste é aprovado, caso contrário, é reprovado.

Para os diferentes grupos de casos de teste que iremos fazer, estes serão avaliados de forma independente, visto que nem todos os casos de teste testam as mesmas funções. Assim, ao descobrir falhas em algum caso de teste, é mais fácil identificar o problema de forma a corrigir o mesmo.

Assim, quando existe uma falha nas funções de *Left_index* ou *orientations* este é considerado um erro de severidade baixa. Contudo, quando ocorre um erro na função *convex_hull,* é considerado um erro de severidade alta, pois envolve todos as funções deste algoritmo. Há exceções desta alta severidade em caso de falta de alertas que ajudem o utilizador, neste caso tem severidade baixa. Apenas se forem falhas de funcionamento que elas são consideradas de severidade alta.

7. Entregáveis de teste

Escolhemos apresentar cada um de nossos testes em tabelas de casos de teste como as fornecidas pelo professar em exercícios. Temos também pequenos exemplos do código de teste e validação que usamos. Finalmente, temos uma tabela final apenas a dizer se o respetivo caso de teste passou ou falhou.

Para conseguir realizar os casos de teste, a função convex_hull foi modificada de modo a devolver os pontos finais pertencentes ao convex_hull, em vez de apenas os apresentar.

```
return_points=[]
# Print Result
for each in hull:
    return_points.append(points[each])
    print(points[each].x, points[each].y)

return return_points
```

Figura 6 - Mudança efetuada no código, função convex_hull

7.1. White box Testing

7.1.1. Control flow

Paths	Test Cases		Observações
	Input	Expected Outcome	
Path 1	points = [Point(5,9)]	minn = 0	Não entre no ciclo for
Path 2	points = [Point (6,5), Point (3,6)]	minn = 1	1 vez ciclo for
Path 3	points = [Point (4,7), Point (5,7)]	minn= 0	1 vez ciclo for
Path 4	points = [Point (5,4), Point (5,1)]	minn= 0	1 vez ciclo for
Path 5	points = [Point (6,3), Point (6,7)]	minn: 1	1 vez ciclo for
Path 2, 5	points = [Point (6,5), Point (3,6), Point (3,7)]	minn = 2	2 vezes ciclo for
Path 3,2	points = [Point (4,7), Point (5,7), Point (3,5)]	minn = 2	2 vezes ciclo for
Path 4,3	points = [Point (4,7), Point (4,6), Point (5,9)]	minn = 0	2 vezes ciclo for
Path 2,4	points = [Point (7,9), Point (5,8), Point (5,6)]	minn = 1	2 vezes ciclo for
Path 5,2	points = [Point (3,4), Point (3,5), Point (2,9)]	minn = 2	2 vezes ciclo for

Tabela 3- Control flow table

De seguida, iremos apresentar o formato dos casos de teste para o *Control Flow Testing*. Não irá ser apresentado todo o código fonte dos testes, visto que iria ocupar muito espaço, mas irão ser aapresetnados pequenos exemplos de forma a ter uma ideia do processo.

```
def test_case_1_control_flow():
    points = []
    points.append(Point(5, 9))

    assert convex_hull.Left_index(points) == 0

def test_case_2_control_flow():
    points = []
    points.append(Point(6, 5))
    points.append(Point(3, 6))

    assert convex_hull.Left_index(points) == 1
```

Figura 7 - Exemplo código test case control flow

A imagem acima, apresenta os dois primeiros casos de teste da tabela. Como podemos ver, é preparado a lista de points, com objetos da classe Points, e posteriormente, é feito um *assert*, entre o resultado devolvido pela função e o *expected outcome*.

A tabela seguinte mostra os casos de teste efetuados e se estes passaram ou falharam.

Teste Case	Pass/Fail
1	Pass
2	Pass
3	Pass
4	Pass
5	Pass
6	Pass
7	Pass
8	Pass
9	Pass
10	Pass

Tabela 4- Resultados test case

7.1.2. Data flow

Variáveis	Paths	Test Cases		Condições
		Input	Expected Outcome	
points	Path 1,2,3	N=3; points = [Point (6,8) , (3,4) ,(8,5)]	points = [Point (3,4), Point (8,5), Point (6,8)]	Condição Nó 2 tem de ser true, 8 entrar no for, 9 ser true e 14 entrar no for
n	Path 1,2,3	N = 4; points = [Point (3,5), Point (6,8) , Point (10,5), Point (4,7)]	points = [Point (3,5), Point (10,5), Point (6,8), Point (4,7)]	Condição nó 2 tem de ser falsa
l	Path 1,2	N = 4; points = [Point (3,5), Point (6,8) , Point (10,5), Point (6,6)]	points = [Point (3,5), Point (10,5), Point (6,8)]	Não requer
hull	Path 1	N = 4; points [Point (3,6), Point (4,8) , Point (10,5), Point (6,7)]	points = [Point (3,6), Point (10,5), Point (6,7), Point (4,8)]	Condição nó 12 tem de ser true
p	Path 1,2,3,4	N = 5; points = [Point (3,6), Point (4,8) , Point (10,5), Point (6,7), Point (6,8)]	points = [Point (3,6), Point (10,5), Point (6,8), Point (4,8)]	Condição nó 8 tem de entrar no for
q	Path 1,2,3,4	N = 4; points = [Point (4,8), Point (10,5), Point (6,7), Point (6,8)]	points = [Point (4,8), Point (10,5), Point (6,8)]	Condição nó 8 entrar no for, entrar no 9, voltar a entrar em 8, não fazer 9, não entra mais em 8

i	Path 1,2	N = 4; points =[Point (6,8), Point (1,6), Point (4,8), Point (6,7)]	points = [Point (1,6), Point (6,7), Point (6,8), Point (4,8)]	Condição 9 tem de ser true
each	Path 1	N = 3; points = [Point (6,8), Point (4,8), Point (6,7)]	points = [Point (4,8), Point (6,7), Point (6,8)]	Não precisa

Tabela 5 - Tabela data flow

Para cada variável foi criado um test case, como mostra a tabela. Do mesmo modo que os casos de teste realizados para o *Control Flow* foram codificados, para o *Data Flow* o processo foi muito semelhante. Na figura abaixo, que mostra os dois primeiros casos de teste da tabela, é na mesma declarado o array de classe de points de Input e declarada a lista de classes de points esperada. No final é feita uma asserção entre a lista de points e a lista de points esperados, comparando cada elemento x e y das duas listas.

```
def test_case_variavel_points_data_flow():

    points = []
    points.append(Point(6, 8))
    points.append(Point(3, 4))
    points.append(Point(8, 5))

    expected_points =[]
    expected_points.append(Point(3, 4))
    expected_points.append(Point(8, 5))
    expected_points.append(Point(6, 8))

    result = convex_hull.convexHull(points,3)

    assert result[0].x == expected_points[0].x and result[0].y == expected_points[0].y and result[1].x == expected_points[1].x and result[1].y == expected_points[1].y and result[2].x == expected_points[2].x and result[2].y == expected_points[2].y

def test_case_variavel_n_data_flow():
    points = []
    points.append(Point(3, 5))
    points.append(Point(6, 8))
    points.append(Point(10, 5))
    points.append(Point(4, 7))

    expected_points =[]
    expected_points.append(Point(3, 5))
    expected_points.append(Point(10, 5))
    expected_points.append(Point(6, 8))
    expected_points.append(Point(4, 7))

    result = convex_hull.convexHull(points,4)

    assert result[0].x == expected_points[0].x and result[0].y == expected_points[0].y and result[1].x == expected_points[1].x and result[1].y == expected_points[1].y and result[2].x == expected_points[2].x and result[2].y == expected_points[2].y and result[3].x == expected_points[3].x and result[3]
```

Figura 8- Exemplo Código Test Case data flow

No final, foi registado quais os casos de teste que passaram com sucesso e quais falharam. A tabela abaixo resume as conclusões recolhidas destes testes.

Teste Case	Pass/Fail
1	Pass
2	Pass
3	Pass
4	Pass
5	Pass
6	Pass
7	Pass
8	Pass

Tabela 6- Resultados Test Case

7.2. Black Box Testing

7.2.1. Equivalence class Partitioning

Funções	Válido ou Inválido	Test Cases	
		Input	Expected Outcome
		points = [Point (5,9)]	minn = 0
		points = [Point (4,7), Point (5,7) , Point (3,5)]	minn = 2
		points = [Point (float(4.00), float(7.00)), Point (float(5.00), float(7.00)), Point (float(3.00), float(5.00))]	minn= 2
		points = [Point (“4”,”7”), Point (“5”,”7”) , Point (“3”,”5”)]	minn= 2

Left_index(points)	Válido	points = [Point (float(4.0000001), float(7. 0000001)), Point (float(5. 0000001), float(7. 0000001)), Point (float(3. 0000001), float(5. 0000001))]	minn: 2
		points = [Point (-4,-7), Point (-5,-7) , Point (-3,-5)]	minn = 1
		points = [Point (float(-4.00), float(-7.00)), Point (float(-5.00), float(-7.00)), Point (float(-3.00), float(-5.00))]	minn = 1
		points = [Point (float(4.0000001),"7.00"), Point (float(5.00), 7) , Point (3.0,"5.00")]	minn = 2
		points = [Point (6,8) , Point (3,4) , Point (8,5)]	minn = 1
		points = [Point (True,False), Point (False,False) , Point (True,False)]	minn = 1
		points = [Point (3+1,5+2), Point (3+2,5+2) , Point (2+1,4+1)]	minn = 2
		points = [ ]	minn = 0
	Inválido	points = [Point (float(4.0000001),"7.00"), Point (float(5.00), 7) , Point ("3.0","5.00")]	Type Error: cant compare string with float
		points = [Point ("-4","-7"), Point ("-5","-7") , Point ("-3","-5")]	Assertion: value diferente, cant convert "-" into a negative value and give the correct result
		points = [("abc","too"), ("ma","do") , Pint ("ti","ai")]	AttributeError : input não é uma lista de Classe Points
		points = [(6,8), (4,8), (6,7)]	AttributeError : input não é uma lista de Classe Points
		points = [(True,False), (False,False) , (True,False)]	AttributeError : input não é uma lista de Classe Points
		points = True	TypeError : boolean não tem len()
		points = "abc"	AttributeError : input não é uma lista de Classe Points
		points = {(6,5),(7,8)}	TypeError: objeto set não é subscriptable
orientation(p,q,r)	Válido	p=Point(4, 7), q=Point(5, 7), r=Point(3, 5)	1
		p=Point(3, 5), q=Point(5, 7), r=Point(4, 7)	2
		p=Point(3, 5), q=Point(3, 6), r=Point(3, 7)	0
		p= Point(float(4.00), float(7.00)), q= Point(float(5.00), float(7.00)), r=Point(float(3.00), float(5.00))	1
		p=Point(float(4.000000001), float(7.000000001)), r=Point(float(5.000000001), float(7.000000001)), q=Point(float(3.000000001), float(5.000000001))	2
		p=Point(-4, -7), r=Point(-5,-7), q=Point(-3, -5)	2
		p=Point(float(-4.00), float(-7.00)), r=Point(float(-5.00), float(-7.00)), q=Point(float(-3.00), float(-5.00))	2
		p=Point(float(4.000000001), float(7.00)), r=Point(float(5.00), 7), q=Point(3.0, float(5.00))	2
		p=Point(True, True), r=Point(False, False), q=Point(True, False)	1
		p=Point(3+1, 5+2), r=Point(3+2, 5+2), q=Point(2+1, 4+1)	2
	Inválido	p=Point("4", "7"), q=Point("5", "7"), r=Point("3", "5")	TypeError: não consegue fazer as operações
		p=Point(float(4.000000001), "7"), r=Point(float(5.00), 7), q=Point(3.0, "3")	TypeError: não consegue comparar string com float

		p=(3, 5), q=(3, 6), r=(3, 7)	AttributeError: inputs não são da Classe Points
		p=[3, 5], q=[3, 6], r=[3, 7]	AttributeError: inputs não são uma lista de Classe Points
		p=4, r=7, q=8	AttributeError: inputs não são da Classe Points
		p=(True, True), r=(False, False), q=(True, False)	AttributeError: inputs não são da Classe Points
convexHull(points, n)	Válido	points= [Point(5, 9)] n=1	None
		points= [Point(4, 7), Point(5, 7), Point(3, 5)] n=3	points= [Point(3, 5), Point(5, 7), Point(4, 7)]
		points= [Point(float(4.00), float(7.00)), Point(float(5.00), float(7.00)), Point(float(3.00), float(5.00))] n=3	points= [Point(3, 5), Point(5, 7), Point(4, 7)]
		points= [Point(float(4.000000001), float(7.000000001)), Point(float(5.000000001), float(7.000000001)), Point(float(3.000000001), float(5.000000001))] n=3	points= [Point(float(3.000000001), float(5.000000001)), Point(float(5.000000001), float(7.000000001)), Point(float(4.000000001), float(7.000000001))]
		points= [Point(-4, -7), Point(-5,- 7), Point(-3, -5)] n=3	points= [Point(-5,- 7), Point(-4, -7), Point(-3, -5)]
		points= [Point(float(-4.00), float(-7.00)), Point(float(-5.00), float(-7.00)), Point(float(-3.00), float(-5.00))] n=3	points= [Point(float(-5.00), float(-7.00)), Point(float(-4.00), float(-7.00)), Point(float(-3.00), float(-5.00)))]
		points= [Point(float(4.000000001), float(7.00)), Point(float(5.00), 7), Point(3,float(5.00))] n=3	points= [Point(3, float(5.00)), Point(float(5.00), 7), Point(float(4.000000001), float(7.00))]
		points= [Point(True, True), Point(False, False), Point(True, False)] n=3	points= [Point(False, False), Point(True, False), Point(True, True)]
		points= [Point(3+1, 5+2), Point(3+2, 5+2), Point(2+1,4+1)] n=3	points= [Point(3, 5), Point(5, 7), Point(4, 7)]
		points = [] n=0	None
	Inválido	points = [Point(4, 7), Point(5, 7), Point(3, 5)] n = “3”	Type Error: não consegue comparara string n com integer
		points = [Point(4, 7), Point(5, 7), Point(3, 5)] n = 3.0	TyepError: float n não consegue ser interpretado como integer
		points = [Point(float(4.000000001), "7.00"), Point(float(5.00), 7), Point("3.0", "5.00")] n = 3	TyepError: não consegue comprar string com float
		points = Point("-4", "-7"), Point("-5", "-7"), Point("-3", "-5")] n = 3	TyepError: não consegue fazer operações com string
		points = (4, 7), (5, 7), (3, 5)] n = 3	AttributeError: lista de points não é lista com classe de points
		points = ("abc", "too"), ("ma", "do"), ("ti", "ai")] n = 3	AttributeError: lista de points não é lista com classe de points
		points = (True, True), (False, False), (True, False)] n = 3	AttributeError: lista de points não é lista com classe de points

Tabela 7- Tabela de equivalence class partitioning

As duas imagens que se seguem, representam exemplos de casos de teste para as classes válidas dos dados de entrada do *Left_index*, e para as classes inválidas dos dados de entrada da função *orientation*. Do lado das classes válidas, o processo é similar ao que descrevemos anteriormente, preparando os dados de entrada e o resultado esperado, para no final fazer uma comparação na asserção entre o resultado devolvido e o resultado esperado.

Os casos de teste para as classes inválidas, normalmente é verificado se a função ao ser chamada, levanta erros ou exceções como esperado.

```python
def test_left_index_valid_classes():
    points = []
    points.append(Point(5, 9))

    assert convex_hull.Left_index(points) == 0

    points = []
    points.append(Point(4, 7))
    points.append(Point(5, 7))
    points.append(Point(3, 5))

    assert convex_hull.Left_index(points) == 2

    points = []
    points.append(Point(float(4.00), float(7.00)))
    points.append(Point(float(5.00), float(7.00)))
    points.append(Point(float(3.00), float(5.00)))

    assert convex_hull.Left_index(points) == 2
```

```python
def test_orientation_invalid_classes():

    p=Point("4", "7")
    q=Point("5", "7")
    r=Point("3", "5")
    with raises(TypeError):
        assert convex_hull.orientation(p,q,r) == 1


    p=Point(float(4.000000001), "7")
    r=Point(float(5.00), 7)
    q=Point(3.0, "3")
    with raises(TypeError):
        assert convex_hull.orientation(p,q,r) == 2


    p=(3, 5)
    q=(3, 6)
    r=(3, 7)
    with raises(AttributeError):
        assert convex_hull.orientation(p,q,r) == 0
```

Figura 9 - equivalence class partitioning test case

A tabela seguinte indica o sucesso ou não, para cada função e classe, dos casos de teste.

Funções	Válido/Inválido	Teste Case	Pass/Fail
Left_index(points)	Válido	1	Pass
		2	Pass
		3	Pass
		4	Pass
		5	Pass
		6	Pass
		7	Pass
		8	Pass
		9	Pass
		10	Pass
		11	Pass
		12	Pass
	Inválido	1	Pass
		2	Pass
		3	Pass
		4	Pass
		5	Pass
		6	Pass
		7	Pass
		8	Pass
orientation(p,q,r)	Válido	1	Pass
		2	Pass
		3	Pass
		4	Pass
		5	Pass
		6	Pass
		7	Pass
		8	Pass
		9	Pass
		10	Pass

	Inválido	1	Pass
		2	Pass
		3	Pass
		4	Pass
		5	Pass
		6	Pass
convexHull(points, n)	Válido	1	Pass
		2	Pass
		3	Pass
		4	Pass
		5	Pass
		6	Pass
		7	Pass
		8	Pass
		9	Pass
		10	Pass
	Inválido	1	Pass
		2	Pass
		3	Pass
		4	Pass
		5	Pass
		6	Pass
		7	Pass

Tabela 8 - Resultados Test Case

7.2.2. Boundary values

Test Cases	
Input	Expected Outcome
Lista de points, N=0	None
Lista de points, N=1	None
Lista de points, N=3	Uma lista de Points
Lista de points, N=-1	Erro a indicar n negativo

Tabela 9- Tabela Boundary Values

Para os casos de teste do *Boundary Values* da variável *n* é apresentada a imagem abaixo, que indica o formato dos casos de teste.

```
def test_boundary_convex_hull():

    points = []
    assert convex_hull.convexHull(points,0) == None

    points = []
    points.append(Point(5, 9))

    assert convex_hull.convexHull(points,1) == None

    points = []
    points.append(Point(4, 7))
    points.append(Point(5, 7))
    points.append(Point(3, 5))

    expected_points =[]
    expected_points.append(Point(3, 5))
    expected_points.append(Point(5, 7))
    expected_points.append(Point(4, 7))

    result = convex_hull.convexHull(points,3)

    assert result[0].x == expected_points[0].x and result[0].y == expected_points[0].y and result[1].x == expected_points[1].x and result[1].y == expected_points[1].y and result[2].x == expected_points[2].x and result[2].y == expected_points[2].y


    points = []
    points.append(Point(5, 9))
```

Figura 10- Código exemplo boundary values test case

Esta tabela indica se os casos de teste para os *Boundary Values* da variável *n* passaram ou não.

Teste Case	Pass/Fail
1	Pass
2	Pass
3	Pass
4	Fail

Tabela 10- Resultados Test Case

O caso teste 4 falhou porque a variável a ser analisada guarda a quantidade de pontos *Point(x,y)* na lista *points* e, portanto, nunca poderá ser negativa. O programa deveria avisar que é um cenário impossível para o utilizador que pode desconhecer o código.

8. Necessidades ambientais

Para realizar estes testes foi apenas necessário ter o Python e a *framework* Pytest instalada.

9. Recursos humanos e responsabilidades

Todos os elementos que realizaram os testes possuem licenciatura em engenharia informática.

Devido à pandemia atual, as responsabilidades foram divididas da seguinte maneira: enquanto duas pessoas foram descobrindo e executando a parte técnica, a terceira pessoa ia registando os passos, documentando os testes, e criando os diferentes grafos que permitisse completar a parte técnica dos testes.

10.Relatório de conclusão de teste

O trabalho que nos foi proposto tinha como intenção avaliar a qualidade e a confiabilidade de um simples código para que desenvolvêssemos habilidades de teste de software dinâmico com carácter exploratório. Fizemos testes *White Box* e *Black Box* num algoritmo de Jarvi para calcular o fecho-convexo.

Após os nossos meticulosos testes e análises e ainda considerando os nossos critérios de avaliação, concluímos que o código avaliado é seguro, visto que sucedeu em testes de *Control* e *Data Flow*, assim mesmo como testes de *Equivalent Class Partitioning*. Porém, mostrou falta suporte para os utilizadores não familiarizados com o algoritmo ao realizar testes com *Boundary Values*.

Com o erro obtido nos testes de *Boundary Values*, na função de *convex_hull*, este poderia ser considerado um erro de severidade alta como mencionámos na secção 6. Contudo, também mencionámos que erros relacionados com falta de alertas para utilizadores relativamente a dados de entrada mal formulados e que não afetem a funcionalidade do programa, visto que estes têm proteção, são apenas considerados erros de severidade baixa.