**Report**

# Line/Branch Coverage

Como Branch Coverage “subsumes” Line Coverage, os testes foram feitos apenas com o objetivo de cumprir os requisitos do Branch Coverage, visto que ao cumprir os requisitos deste critério então também se estão a cumprir os requisitos do Line Coverage.

Não são possíveis cobrir todos os branches do método privado isLeaf chamado pelo método contains. O branch em que se empty é true (!empty é false)

É impossível cobrir todos os branches do método equalTrees. O branch “one == other” já é coberto no método equals e que caso aconteça, o método equalTrees não é chamado. O branch e “one == null || other == null” também pois, o one aqui seria o this, que não poderia ser null se não ao tentar chamar o método equals seria lançada uma NullPointerException, e o other também nunca poderia ser null pois a condição “other instanceof NTree” não se verificaria e o método equalTrees não seria chamado.

É também impossível cobrir todos os branches do método proposePosition. No if “if (children[i].data.compareTo(elem)<0)” é impossível cobrir o branch em que esta condição não se verifica, isto porque todas as outras condições (ser igual ou maior do que 0) já foram verificadas em partes anteriores do código. Portanto para obter cobertura total, o código teria de ser escrito com if, else if e else, visto que no fim não há outra alternativa a “children[i].data.compareTo(elem)<0”.

Com este critério foi possível descobrir que o construtor do iterador estava mal implementado. A raiz era pushed para a stack sem verificar se a tree é emtpy. Isto fazia com que caso a tree fosse vazia, e não houvesse elementos para iterar, como a stack não estava vazia, a função hasNext retornava true. Este erro foi encontrado no teste testArrayNTreeEquals2EmptyNTrees da classe ArrayNTreeLineBranchCoverageTest.

# Edge-Pair / Prime Path Coverage

Como o método equals é apenas um return, o seu grafo de controlo de fluxo seriam apenas dois nós. Portanto, para testar este método, foi feito o grafo para o método privado equalTrees, que é onde estão as instruções mais relevantes para a comparação de igualdade entre duas NTrees. Apresenta-se, então, o grafo para o método equalsTrees:

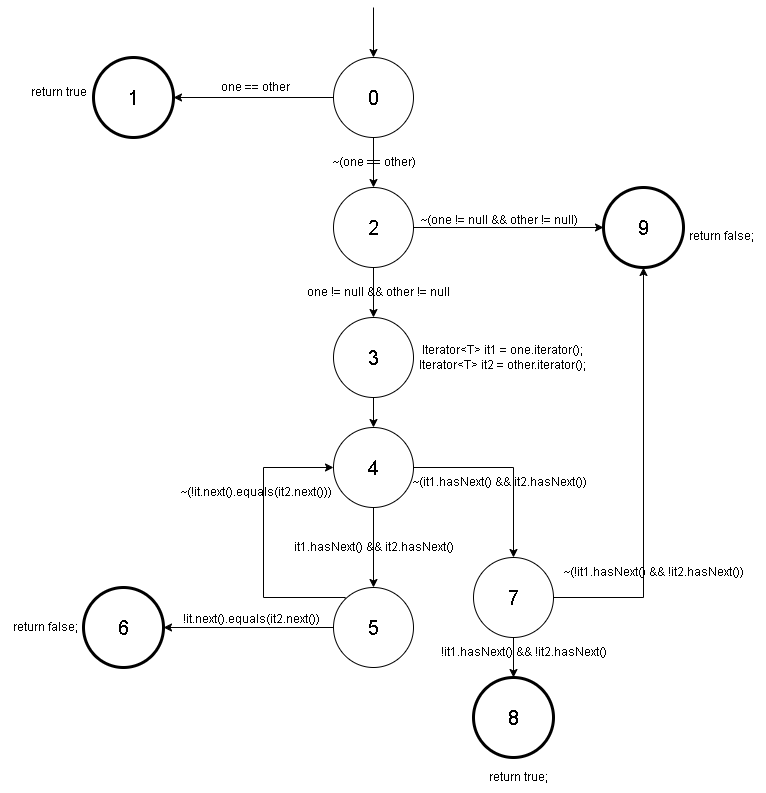


Figure 1 - Grafo de Controlo de Fluxo de equalTrees

Sabendo que este é um método privado chamado pelo método equals, passar por [0,1] é impossível este arco corresponde a verificação de um predicado que é verificado anteriormente (no método equals), e que caso se verifique o método equalTrees não é chamado. O mesmo acontece com o caminho [0,2,9] também porque se one não pode ser null porque é o this e o other também não porque faz se o instanceof NTree e null não é, logo não passa para o equalTrees. Apesar disto, é possível na mesma fazer testes que se assemelham ao comportamento destes caminhos. Portanto na tabela que representa o conjunto de testes, são apresentados testes como “best-effort” para cumprir estes requisitos.

## 2.1. Edge-Pair Coverage

TR(EPC) = { [0,1], [0,2,9], **[0,2,3]**, **[2,3,4]**, **[3,4,5]**, **[3,4,7]**, **[4,5,4]**, **[4,5,6]**, **[4,7,8]**, **[4,7,9]**, **[5,4,5]**, **[5,4,7]** }

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do teste | Valores  (this,other) | Resultado esperado | Caminho | Requerimentos cobertos |
| testArrayNTreeSameRefTrees | Mesma ref | true | [0,1] | [0,1] |
| testArrayNTreeOtherNull | ([],null) | false | [0,2,9] | [0,2,9] |
| testArrayNTreeEqualsFirstChildDiff | ([1:[2]], [1:[3]]) | false | [0,2,3,4,5,4,5,6] | [0,2,3] [2,3,4] [3,4,5] [4,5,4] [5,4,5] [4,5,6] |
| testArrayNTreeEqualsEmptyTrees | ([],[]) | true | [0,2,3,4,7,8] | [0,2,3] [2,3,4] [3,4,7] [4,7,8] |
| testArrayNTreeEqualsSingleEqualRoots | ([1],[1]) | true | [0,2,3,4,5,4,7,8] | [0,2,3] [2,3,4]  [3,4,5] [4,5,4]  [5,4,7] [4,7,8] |
| testArrayNTreeEqualsOneEmptyTree | ([1], []) | false | [0,2,3,4,7,9] | [0,2,3] [2,3,4]  [3,4,7] [4,7,9] |

Table 1 - Conjunto de testes para os requisitos de EPC

## 2.2 Prime Path Coverage

Comecemos por determinar os prime paths (marcados a negrito):

[0] -> **[0,1]!** [0,2] -> [0,2,3] **[0,2,9]!** -> [0,2,3,4] -> [0,2,3,4,5] [0,2,3,4,7] -> **[0,2,3,4,5,6]! [0,2,3,4,7,8]! [0,2,3,4,7,9]!**

[1]!

[2] -> [2,9]2,3] -> [2,3,4] -> [2,3,4,5] [2,3,4,7] -> [2,3,4,5,6]! [2,3,4,7,8]! [2,3,4,7,9]!

[3] -> [3,4] -> [3,4,5] [3,4,7] -> [3,4,5,6]! [3,4,7,8]! [3,4,7,9]!

[4] -> [4,5] [4,7] -> **[4,5,4]\*** [4,5,6]! [4,7,8]! [4,7,9]!

[5] -> [5,4] [5,6]! -> **[5,4,5]\*** [5,4,7] -> **[5,4,7,8]!** **[5,4,7,9]!**

[6]!

[7] -> [7,8]! [7,9]!

[8]!

[9]!

De novo, há caminhos impossíveis (marcados a vermelho) e são feitos testes “best-effort” para cobrir esses caminhos.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do teste | Valores  (this,other) | Resultado esperado | Caminho | Requerimentos cobertos |
| testArrayNTreeSameRefTrees | Mesma ref | true | [0,1] | [0,1] |
| testArrayNTreeOtherNull | ([],null) | false | [0,2,9] | [0,2,9] |
| testArrayNTreeEquals  RootDifferent | ([1],[2]) | false | [0,2,3,4,5,6] | [0,2,3,4,5,6] |
| testArrayNTreeEquals  EmptyTrees | ([],[]) | true | [0,2,3,4,7,8] | [0,2,3,4,7,8] |
| testArrayNTreeEquals  OneEmptyTree | ([1], []) | false | [0,2,3,4,7,9] | [0,2,3,4,7,9] |
| testArrayNTreeEquals  2EqualTreesWith2Elems | ([1:[2]], [1:[2]]) | true | [0,2,3,4,5,4,5,4,7,8] | [4,5,4] [5,4,5] [5,4,7,8] |
| testArrayNTreeEquals  EqualRootsOnlyRootTreeAndRootWithChildTree | ([1:[2]], [1]) | false | [0,2,3,4,5,4,7,9] | [4,5,4] [5,4,7,9] |

Table 2 - Conjunto de testes para os requisitos de PPC

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

# All-Coupling-Use Coverage

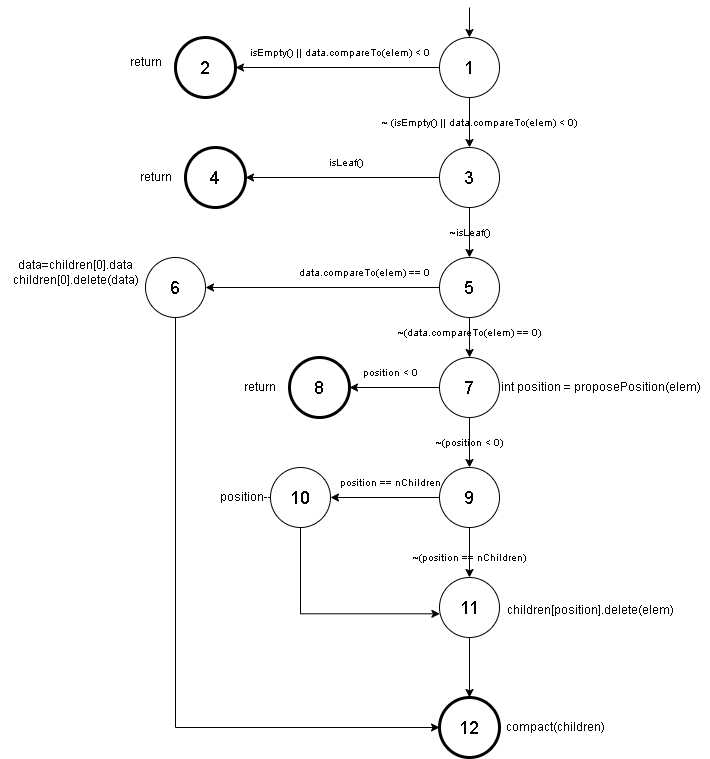
A função delete, invoca os métodos privados isLeaf, isEmpty proposePosition e compact. isLeaf e isEmpty são só returns, e são logo chamados no início da função delete, portanto, ao testar os dois métodos privados mais complexos (proposePosition e compact), também está se a cumprir os requisitos para os métodos privados isLeaf e isEmpty, daí não ser mostrado aqui explicitamente o seu processo de testagem com este critério. Apresenta-se, então, o grafo do método delete:

Figure 2 - Grafo de Controlo de Fluxo para delete

## 3.1 proposePosition

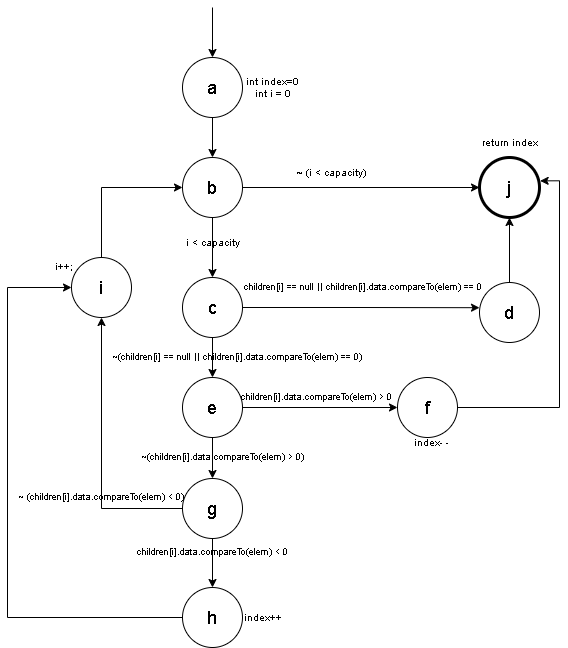
 Para o método privado proposePositions, chamado pelo método delete, há o seguinte grafo:

Figure 3 - Grafo de Controlo de Fluxo de proposePosition

Observando os grafos das Figuras 2 e 3, é possível construir a seguinte tabela:

|  |  |
| --- | --- |
| Last-def | First-use |
| elem: {1} | elem: {d,e} |
| index: {f,h} | position: {8,9} |

Table 3 - Parameter e return value coupling entre delete e proposePosition

Através da tabela, é possível extrair os seguintes pares (last-def -> first-use):

a) (delete, elem, (1) line 204) -> (proposePosition, elem, (d & e) line 268)

b) (proposePosition, (f) index, line 273) -> (delete, position, (8 & 9) line 225)

c)(proposePosition, (h) index, line 278) -> (delete, position, (8 & 9) line 225)

As variáveis de parâmetro e retorno, nos first-uses são arcos, logo são responsáveis os nos finais dos arcos. Isto resulta em pares “diferentes” terem o mesmo fluxo de execução.

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do teste | Valores  (ntree;2) | Resultado Esperado | Requerimentos cobertos |
| testArrayNTreeDeleteFirstChild | ([1:[2]];2) | [1] | a) |
| testArrayNTreeDeleteElemSmallerThanFirstChild | ([1:[3]];2) | [1:[3]] | b) |
| testArrayNTreeDeleteElemGraterThanFirstChild | ([1:[2]];3) | [1:[2]] | c) |

Table 4 - Conjunto de testes para os requisitos de ACUC entre delete e proposePosition

\*Para a representação das árvores foi usada a representação toString dos objetos ArrayNTree

## 3.2 compact

Para o método privado compact, chamado pelo método delete, há o seguinte grafo:

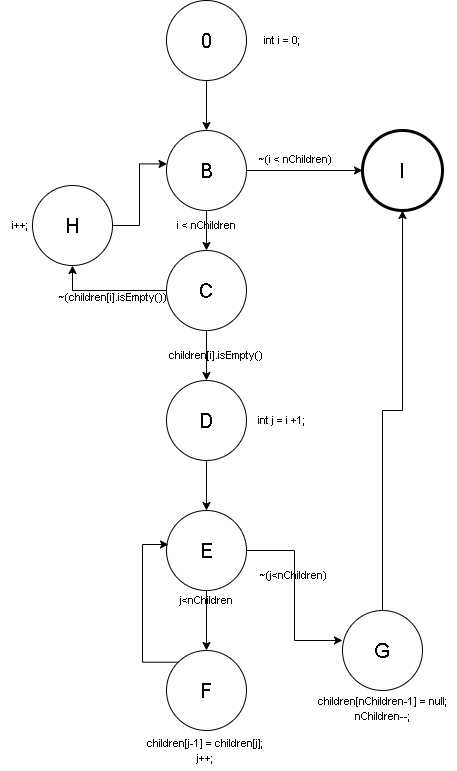


Figure 4 - Grafo de Controlo de Fluxo de compact

Para o método compact, não há return value coupling porque é um método void, portanto a sua tabela apenas irá conter entradas relativas a parameter coupling. Portanto, observando os grafos das Figuras 2 e 4, é possível construir a seguinte tabela:

|  |  |
| --- | --- |
| Last-def | First-use |
| children: {6,11} | chidren: {D,H} |

Table 5 - Parameter coupling entre delete e compact

Através da tabela, é possível extrair os seguintes pares (last-def -> first-use):

1. (delete, children, (6) line 221) -> (compact, children, (D & H) line 243)
2. (delete, children, (11) line 230) -> (compact, children, (D & H) line 243)

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do teste | Valores  (ntree;e) | Resultado esperado | Requerimentos cobertos |
| testArrayNTreeDeleteRootWithChildren | ([1:[2] [3]];1) | [2:[3]] | a) |
| testArrayNTreeDeleteGreatestElement | ([1: [2] [3]], 3) | [1: [2]] | b) |

Table 6 - Conjunto de testes para os requisitos de ACUC entre delete e compact

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

# Logic Coverage

Comecemos por identificar os predicados, clausulas e as determinações de predicado para cada cláusula:

* p1: a || b && c
  + a: this == other
    - d(a): ~b || ~c
  + b: other isntanceof NTree
    - d(b): ~a && c
  + c: equalsTrees(this, (NTree<T>) other)
    - d(c): ~a && b
* p2: d
  + d: one == other
    - d(d):true
* p3: e && f
  + e: one != null
    - d(e): f
  + f: other != null
    - d(f): e
* p4: g && h
  + g: it1.hasNext()
    - d(g): h
  + h: it2.hasNext()
    - d(h): g
* p5: !i
  + i: it1.next().equals(it2.next()))
    - d(i): true
* p6: !j && !k
  + j: it1.hasNext()
    - d(j): ~k
  + k: it2.hasNext()
    - d(k): ~j

Com estes dados, é possível extrair os requerimentos para o critério Correlated Active Clause. Este foi o critério escolhido pois com este, estão incluídos os critérios Predicate e Clause Coverage, o que demonstra que este é um critério bastante geral, que fornecerá uma boa cobertura para o conjunto de testes, e produz menos casos infazíveis como o RACC.

TR(CACC):{

**(1) a && (~b || ~c) && (a || b && c)**, (2) ~a && (~b || ~c) && ~(a || b && c),

(3) b && (~a && c) && (a || b && c), **(4) ~b && (~a && c) && ~(a || b && c)**,

(5) c && (~a && b) && (a || b && c), (6) ~c && (~a && b) && ~(a || b && c),

**(7) d && d**, (8) ~d && ~d,

(9) e && f && (e && f), **(10) ~e && f && ~(e && f)**,

(11) f && e && (e && f), **(12) ~f && e && ~(e && f)**,

(13) g && h && (g && h), (14) ~g && h && ~(g && h),

(15) h && g && (g && h), (16) ~h && g && ~(g && h),

(17) i && i, (18) ~i && ~i,

(18) j && ~k && ~(~j && ~k), (19) ~j && ~k && (~j && ~k),

(20) k && ~j && ~(~j && ~k), (21) ~k && ~j && (~j && ~k),

}

A vermelho estão os requisitos para os quais não existe nenhuma combinação de cláusulas que satisfaça o requisito e que sejam exequíveis devido à lógica do programa. No caso de (1), apenas três combinações a satisfazem: a b ~c, a ~b c e a ~b ~c. Nenhuma destas combinações é exequível no código, porque com a e b a true, então c também tem de o ser visto que a usar o equalsTrees em trees que são a mesma referência tem de ser true, e com a true então b também tem de ser se não seria impossível ser chamado o equals da classe NTree. Para o caso de (4), só com uma combinação é que é satisfeito: ~a ~b c. Esta combinação não é exequível pois se b é falso então c também não pode ser visto que o método equalsTrees nunca daria true com objeto que não são NTrees.

A laranja estão os requisitos que não são possíveis cobrir, pois o predicado já havia sido verificado anteriormente, e método equalsTrees não teria sido chamada caso estes se verificassem. Estes apenas aparecem para cláusulas de predicados relativos ao método privado equalsTrees, em que há predicados que já foram verificados anteriormente pelo método equals.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do teste | Valores  (this;other) | Resultado esperado | Predicado e Valores das clausulas | Requerimentos cobertos |
| testArrayNTreeEqualsOtherNotNTrees | ([],1) | false | p1: ~a ~b ~c | (2) ~a && (~b || ~c) && ~(a || b && c) |
| testArrayNTreeEqualsDiffRefEqualNTrees | ([1:[2,3]],  [1:[2,3]]) | true | p1: ~a b c  p2: ~d  p3: e f  p4: g h  p5: i  p4: ~g ~h  p6: ~j ~k | (3) b && (~a && c) && (a || b && c)  (5) c && (~a && b) && (a || b && c)  (8) ~d && ~d  (9) e && f && (e && f)  (11) f && e && (e && f)  (13) g && h && (g && h)  (15) h && g && (g && h)  (17) i && i  (19) ~j && ~k && (~j && ~k)  (21) ~k && ~j && (~j && ~k) |
| testArrayNTreeEqualsDiffRoots | ([1],[2]) | false | p1: ~a b ~c  p2: ~d  p3: e f  p4: g h  p5: ~i | (6) ~c && (~a && b) && ~(a || b && c)  (8) ~d && ~d  (9) e && f && (e && f)  (11) f && e && (e && f)  (13) g && h && (g && h)  (15) h && g && (g && h)  (18) ~i && ~i |
| testArrayNTreeEqualsThisEmpty | ([],[1]) | false | p4: ~g h  p6: ~j k | (14) ~g && h && ~(g && h)  (20) k && ~j && ~(~j && ~k) |
| testArrayNTreeEquals  OtherEmpty | ([1],[]) | false | p4: g ~h  p6: j ~k | (16) ~h && g && ~(g && h)  (18) j && ~k && ~(~j && ~k) |

Table 7 - Conjunto de testes para os requisitos de CACC

Na escolha dos valores para os predicados foi necessária alguma atenção, pois nem todas as combinações que cobrem requisitos eram exequíveis. Um exemplo deste caso seria para p1 com cláusulas a b e ~c que cobre o requisito a && (~b || ~c) && (a || b && c). Esta combinações de cláusulas é impossível porque caso a e b sejam verdadeiras então c também tem de ser.

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

\* Todos os resultados foram obtidos com a ferramenta: *Logic Coverage Web Application*

# Base Choice Coverage

Seguindo as seguintes características:

1. Tree 1 is empty – [T1Empty, T1NotEmpty]

2. Tree 2 is empty – [T2Empty, T2NotEmpty]

3. Tree 2 is null - [T2Null, T2NotNull]

4. Tree 1 intersection of Tree 2 is empty/full/partial – [E,F,P]

Um bom caso base seria aquele que seria o mais comum num ambiente de execução normal do sistema, portanto sugere-se: (T1NotEmpty, T2NotEmpty, T2NotNull, E). Para a característica 4, o valor que aparenta ser o mais comum seria o P (partial intersection), o problema deste valor é que seriam gerados muitos requisitos impossíveis (com T1 ou T2 empty interseção parcial é impossível). Portanto, a partir do caso base escolhido, os requerimentos para testar são:

TR(BCC): {

(1) (T1NotEmpty, T2NotEmpty, T2NotNull, E), (2) (**T1Empty**, T2NotEmpty, T2NotNull, E),

(3) (T1NotEmpty, **T2Empty**, T2NotNull, E), (4) (T1NotEmpty, T2NotEmpty, **T2Null**, E),

(5) (T1NotEmpty, T2NotEmpty, T2NotNull, **F**),(6) (T1NotEmpty, T2NotEmpty, T2NotNull, **P**)

}

Em cada requisito há uma característica a negrito, que indica a que foi alterada, de modo a seguir o Base Choice Coverage.

O requisito (4) é impossível pois não há como T2 ter valores e ser null.

|  |  |  |  |
| --- | --- | --- | --- |
| Nome do teste | Valores  (this;other) | Resultado Esperado | Requerimentos cobertos |
| testArrayNTreeEqualsDiffNotNullTrees | ([1:[2]];[5:[8][11]]) | false | (1) |
| testArrayNTreeEqualsThisEmpty | ([],[1:[3]]) | false | (2) |
| testArrayNTreeEqualsOtherEmpty | ([2:[5] [6]];[]) | false | (3) |
| testArrayNTreeEqualsThisEqualsOther | ([3:[4] [5]]; [3:[4] [5]]) | true | (5) |
| testArrayNTreeEqualsThisSubsetOther | ([1:[3]]; [1:[3] [4:[5] [6]]] | false | (6) |

Table 8 - Conjunto de testes para os requisitos de BCC

\*Para a representação dos valores foi usada a representação toString dos objetos ArrayNTree

# Verificação dos conjuntos de testes com program mutation (PIT)

Neste tópico vão ser avaliados os conjuntos de testes elaborados para o método equals, extraídos pelos requisitos obtidos com os diferentes critérios utilizados. **Os métodos equals e equalsTrees estão implementados entre as linhas 290 e 316**, portanto serão procuradas mutações que tenham sobrevivido neste intervalo de linhas.

É relevante notar que em todos os relatórios aparece na linha 301 um aviso de NO\_COVERAGE. O motivo disto acontecer já foi explicado anteriormente. Resumidamente o predicado verificado na linha 301 é verificado anteriormente pelo método equals, e caso este predicado se verifique no método equals, o método equalsTrees não é chamado.

Depois do tratamento das mutações no método equals, também foi feita uma exploração de mutações em outros métodos. Para “matar” estas outras mutações, por vezes foi necessário acrescentar testes, com o mesmo objetivo dos que já existiam, mas feito de forma diferente, ou então tornar a lógica dos já existentes mais complexa (explorando diferentes ordens de inserir elementos). Apesar de ser apenas pedido para se fazer este processo para o equals, extender esta tarefa de modo a “matar” o máximo de mutações possível, tornando os conjuntos de testes mais fortes, apenas apresenta benefícios, daí ter sido realizado este processo extra.

## 6.1 Branch/Line Coverage

Correndo PIT mutation test na classe ArrayNTreeLineBranchCoverageTest e observando, para as linhas relevantes aos métodos equals e equalsTrees, tem-se a seguinte imagem:

![A screenshot of a computer code

Description automatically generated

Figure 5 - Estado das mutações no método equals, para o conjunto de testes de Branch e Line Coverage

Nesta imagem, vê-se que duas mutações sobreviveram. Estas mutações são relativas ao predicado do while: “while(it1.hasNext() && it2.hasNext())”, em que substituiu ambas as clásulas (não no mesmo teste), por false e o resultado foi o esperado na mesma. Isto indica que é preciso reforçar os testes existentes ou adicionar novos.

O problema, que fazia estas mutações passarem despercebidas, estava no facto de sempre que aquela condição do while era relevante, no contexto dos testes, o resultado ia sempre dar falso. Portanto acrescentou-se um teste com duas árvores iguais em que aquele while precisava ser true para se poderem comparar elementos da árvore e verificar que quando alguma não tem elementos, então ambas não têm. Depois desta adição, ambas as mutações foram “mortas”.

De forma a garantir ter um conjunto testes o mais forte possível, também foram “mortas” outras mutações relativas a outros métodos. No conjunto de testes ArrayNTreeLineBranchCoverage, estão comentários a explicar mais especificamente o que foi alterado, e que mutação essa alteração estava a “matar”. No método contains havia mutações relativas ao último return, que não era explorado suficientemente bem com os testes presentes, tendo-se então alterado os testes (chamando o método em arvores com maior profundidade e procurar em níveis mais profundos) de forma a alcançar esse return final, para que este tenha impacto no resultado do método. No método insert, em vez de se colocar sempre os elementos de forma ordenada, foram se baralhando, explorando casos em que há troca da root e shifts nos filhos de um nó. É relevante notar que não foi possível “matar” todas as mutações. Sobreviveram mutações relativas a “conditional boundary changed” em predicados faziam comparações de grandeza estrita. Estas são impossíveis de “matar” porque a mutação era fazer com que a comparação de grandeza não fosse estrita, e os casos de igualdade já eram testados em predicados anteriores e passavam para outro fluxo de execução caso se verificassem.

## Edge-Pair / Prime Path Coverage

Correndo PIT mutation test na classe ArrayNTreeEdgePairPrimePathCoverageTest e observando, para as linhas relevantes aos métodos equals e equalsTrees, tem-se informação relativa às mutações na seguinte imagem:

A screenshot of a computer

Description automatically generatedCom este relatório verifica-se que, com o conjunto de testes gerado para o cumprir os critérios Edge-Pair e Prime Paths, todas as mutações relativas ao método equals foram “mortas”.

Figure 6 - Estado das mutações no método equals, para o conjunto de testes de Edge-Pair e Prime Path Coverage

De forma a garantir ter um conjunto testes o mais forte possível, também foram “mortas” outras mutações relativas a outros métodos. No conjunto de testes ArrayNTreeEdgePairPrimePath, estão comentários a explicar mais especificamente o que foi alterado, e que mutação essa alteração estava a “matar”. Foram “mortas” mutações no insert relativas à troca da root e também relativas ao respeito do invariante. Esta última gerou uma solução peculiar. Uma da mutação era negar o predicado para se dar a troca da root (linha 156). Ora o conjunto de testes é apenas para equals, porntato asserts só com o equals não iriam matar esta mutação, pois as àrvores continuavam iguais, o invariante é que era violado. Portanto para “matar” esta mutação foi acrescentado um assert para verificar o valor mínimo da arvore, pois com o invariante violado, os algoritmos de procura retornam o resultado errado. De novo, neste test set surigram as mesmas mutações “unkillable”, relativas a “conditional boundary changed”

## Logic Coverage

Correndo PIT mutation test na classe ArrayNTreeLogicCoverageTest e observando, para as linhas relevantes aos métodos equals e equalsTrees, tem-se informação relativa às mutações na seguinte imagem:

A screenshot of a computer

Description automatically generated

Figure 7- Estado das mutações no método equals, para o conjunto de testes de Logic Coverage

Com este relatório verifica-se que, com o conjunto de testes gerado para o cumprir o critério CACC para Logic Coverage, todas as mutações relativas ao método equals foram “mortas”.

De forma a garantir ter um conjunto testes o mais forte possível, também foram “mortas” outras mutações relativas a outros métodos. Novamente foram encontradas mutações “unkillable” relacionadas com “conditional boundary changed” e foram mortas mutações relativas ao shift da children de nós, troca de roots e da violação do invariante. Porém, neste conjunto de testes surgiu novo tipo de mutação, um relacionado à ocultação de chamadas a métodos.

## Base Choice Coverage

A screenshot of a computer code

Description automatically generatedCorrendo PIT mutation test na classe ArrayNTreeBaseChoiceCoverageTest e observando, para as linhas relevantes aos métodos equals e equalsTrees, tem-se informação relativa às mutações na seguinte imagem:

Figure 8- Estado das mutações no método equals, para o conjunto de testes de Base Choice Coverage

Com este relatório verifica-se que, com o conjunto de testes gerado para o cumprir o critério Base Choice, todas as mutações relativas ao método equals foram “mortas”.

De forma a garantir ter um conjunto testes o mais forte possível, também foram “mortas” outras mutações relativas a outros métodos. Novamente foram encontradas mutações “unkillable” relacionadas com “conditional boundary changed” e foram mortas mutações relativas ao shift da children de nós, troca de roots, ocultação de chamadas a outros métodos e da violação do invariante. Porém, neste conjunto de testes também surgiu um novo tipo de mutação, um relacionado à troca de incremento para decremento de variáveis.

# Property based testing

Para testar as diferentes propriedades foi necessária a implementação de um gerador aleatório de ArrayNTrees de integers. O gerador está implementado na classe ArrayNTreeGenerator. Este gerador tem a possibilidade de ser usado nas propriedades com a anotação InRange, para se poder ter um controlo mais fino sobre o número de elementos na árvore.

Depois de implementado o gerador de árvores, fazer este tipo de testing é bastante simples, sendo apenas necessário testar as propriedades pedidas com as árvores geradas. O único ponto mais interessante de se mencionar é o facto de ter sido usado a class Random para aumentar ainda mais a aleatoriedade dos testes executados. Na propriedade 3 é se pedido para acrescentar um elemento e removê-lo, então para aumentar a aleatoriedade dos testes executados, este elemento em vez de ser um valor igual para todos os testes, é um valor gerado pela class Random. O mesmo acontece na propriedade 5, mas também é gerado outro valor aleatório para as vezes em que o elemento é inserido na árvore.