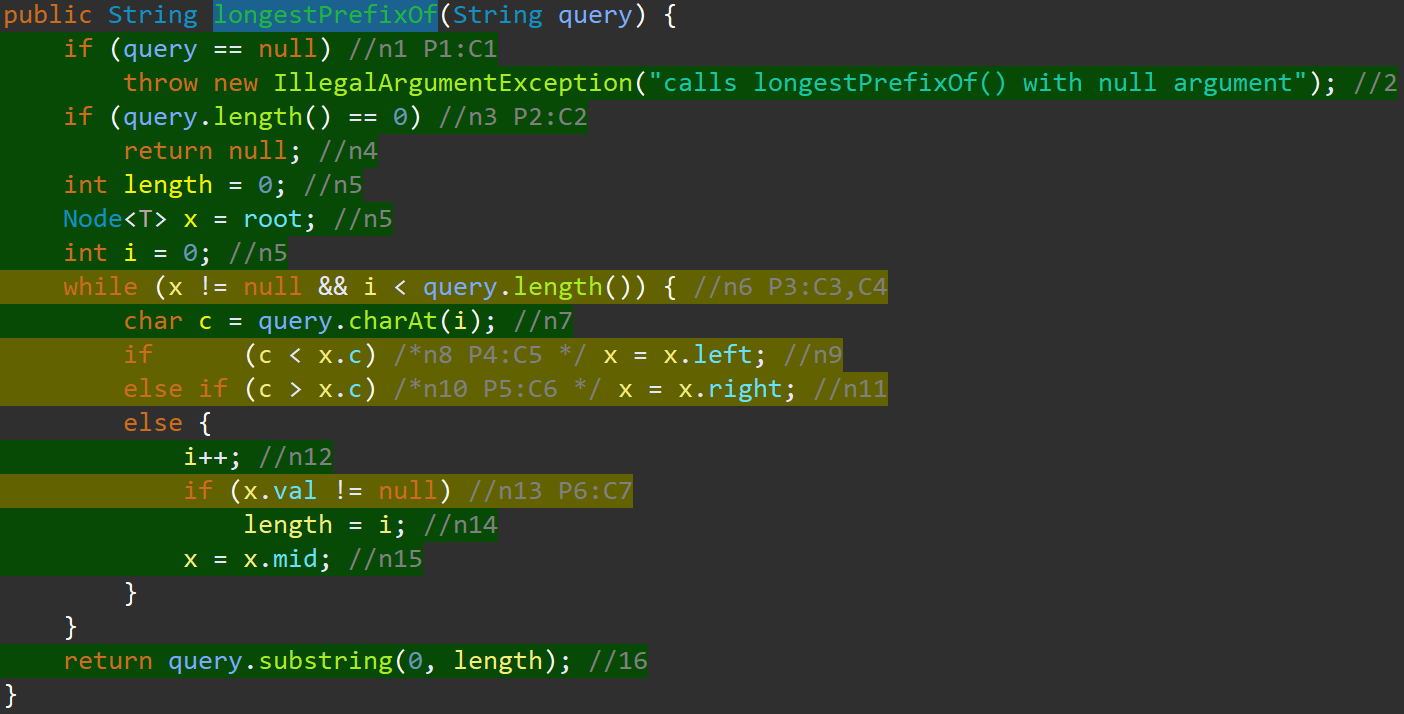
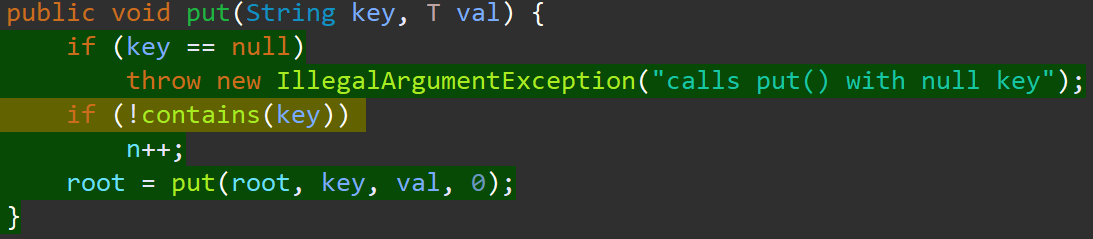
# Relatório de VVS01

Gabriel Henriques - 58182

## Line Coverage

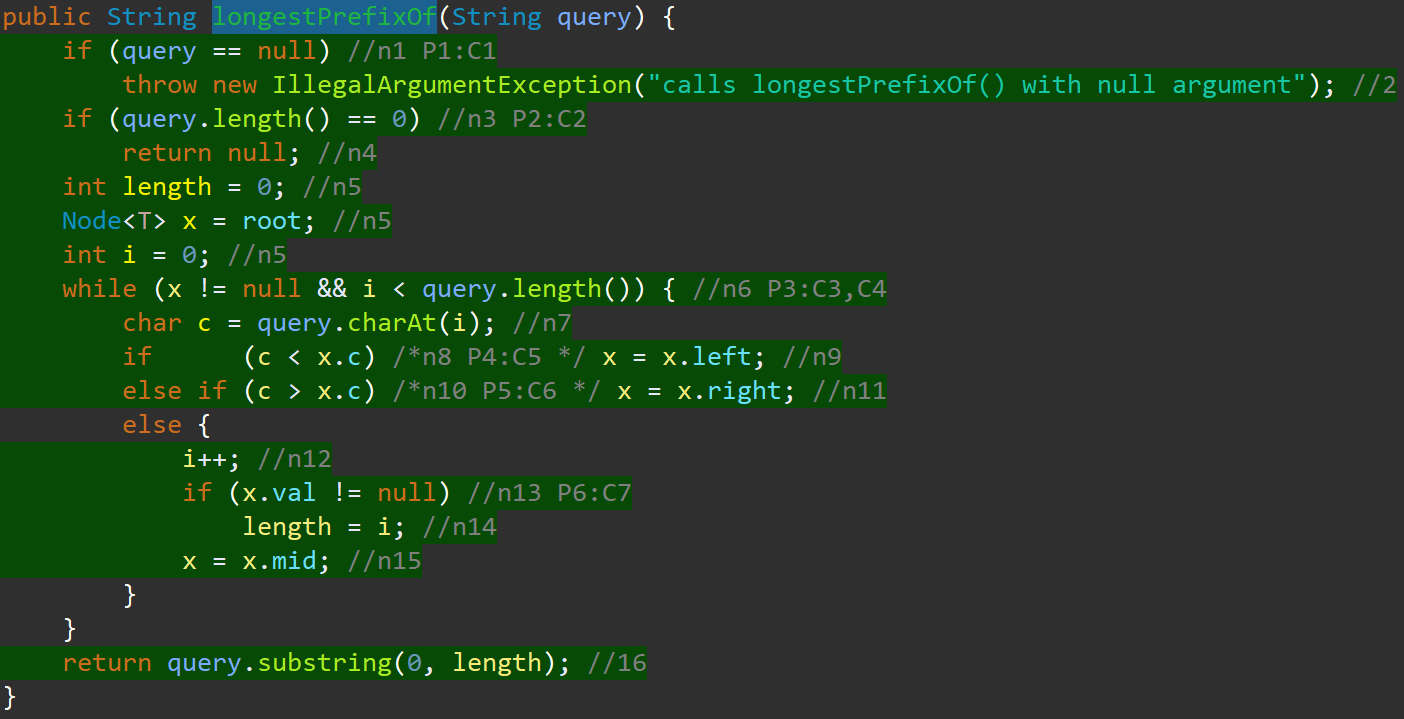
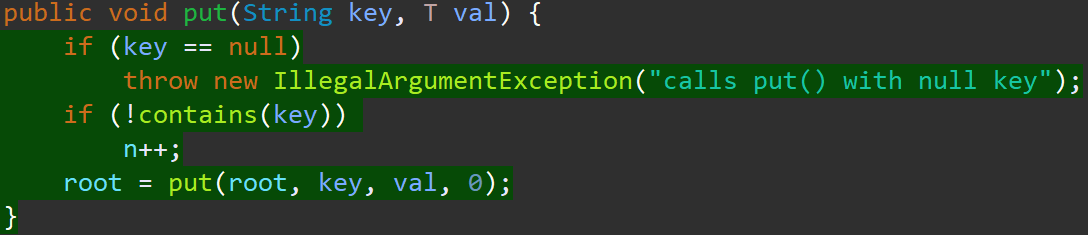
O *Line Coverage* foi atingido em todos os métodos públicos ao fazer o mínimo de testes necessários que passassem pelo menos uma vez por cada linha de código. Todos os métodos obtiveram coverage verde com exceção do *longestPrefixOf* e *put*, pois não forem feitos testes suficientes para que passassem em todas branches.



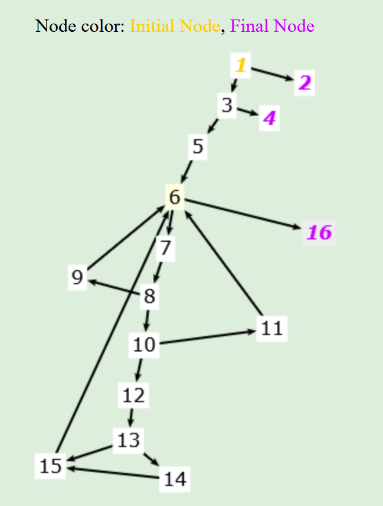


## Branch Coverage

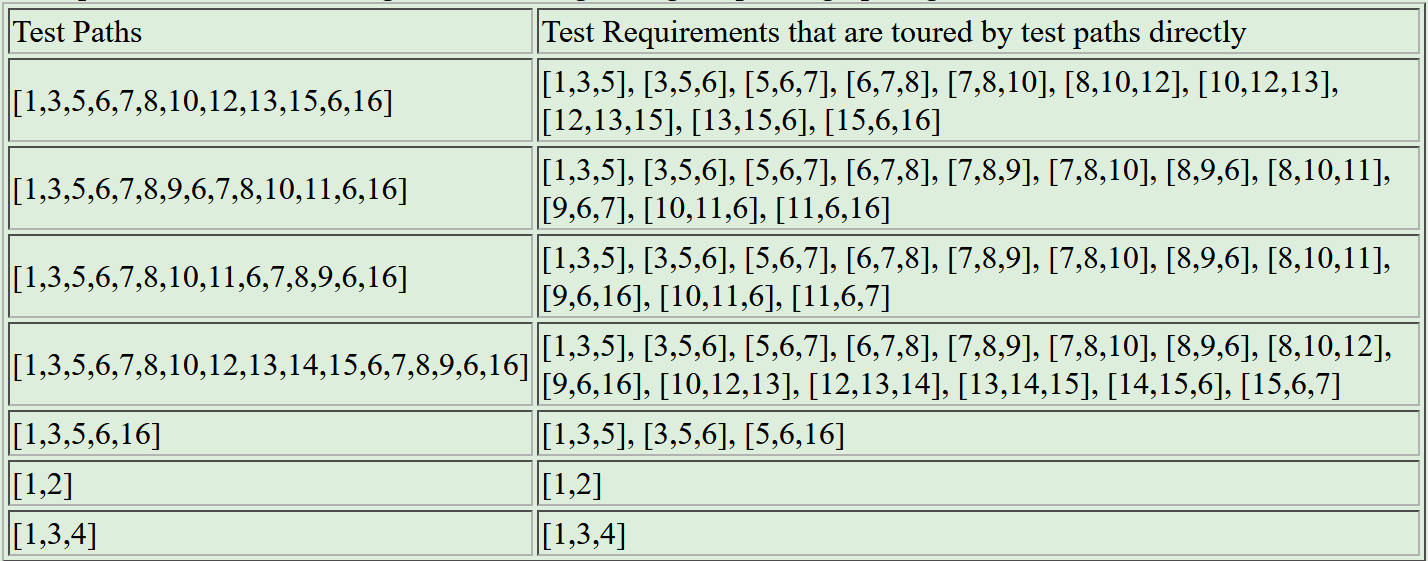
No *Branch* *Coverage*, foram feitos todos os testes já escritos no *Line* *Coverage*, com a adição de mais 3 testes no *longestPrefixOf* e mais 1 teste no *put*, de modo a satisfazer a coverage que antes aparecia em amarelo, por não serem testadas todas as branches dos métodos. Demontrou-se então que o *Branch* *Coverage* testa mais exaustivamente o sistema, sem necessitar de uma maior análise em relação ao *Line* *Coverage*.

## Edge-Pair Coverage

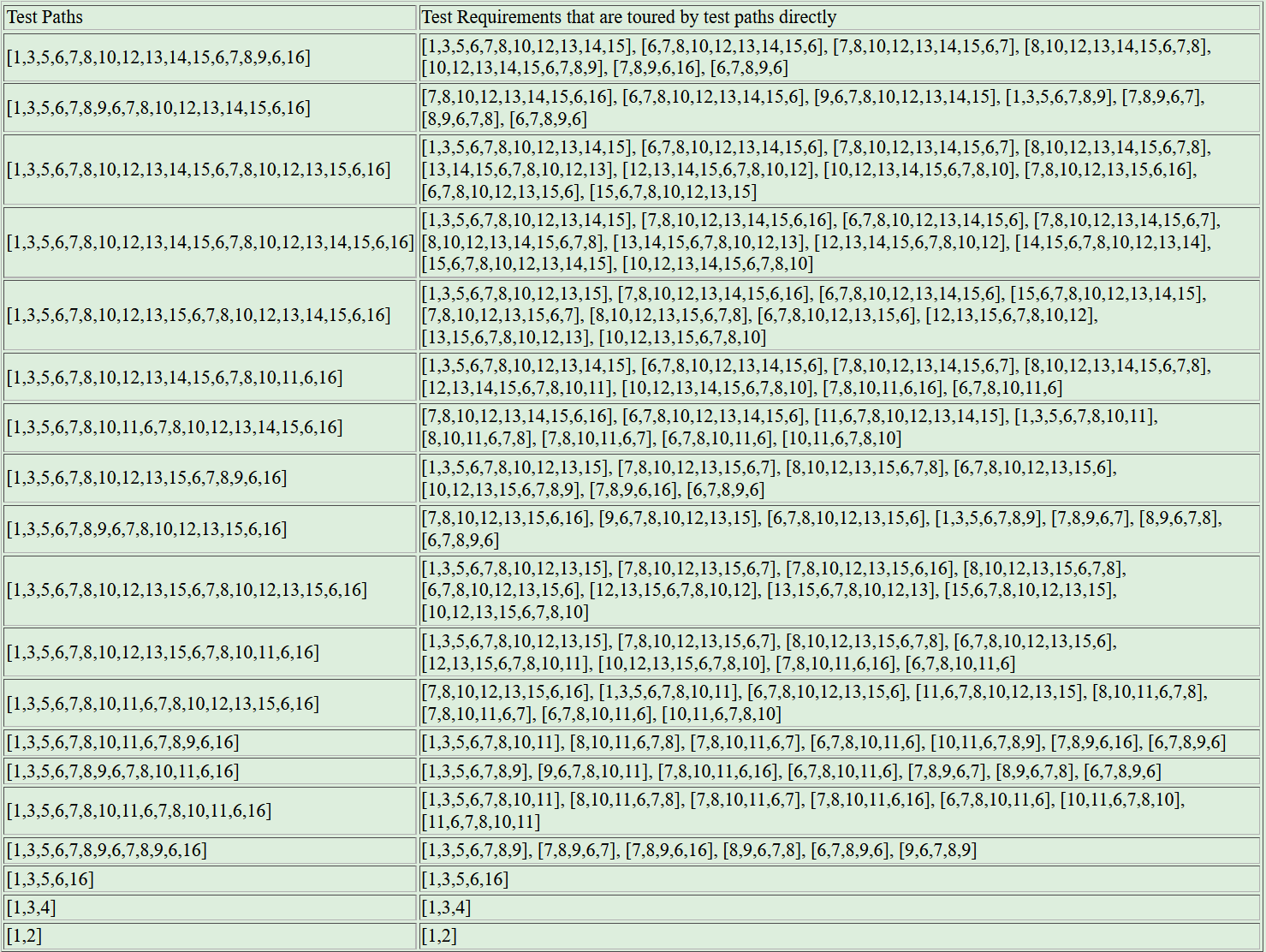
Para satisfazer o *Edge* *Pair* *Coverage*, foi necessário fazer o grafo do método *longestPrefixOf* tal como representado na seguinte imagem:

Após a criação do grafo, foi inserido no site <https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage> os seus edges, initial node e final nodes, e gerado pelo Algoritmo 2 os paths a percorrer pelos testes de modo a satisfazer o *Edge* *Pair* *Coverage:*



## Prime Path Coverage

Para o Prime Path Coverage do método *longestPrefixOf,* foi feito o mesmo procedimento, mas foi escolhido o Algoritmo 1 que gerou mais paths, porém mais curtos, reduzindo assim a complexidade na escrita dos testes:

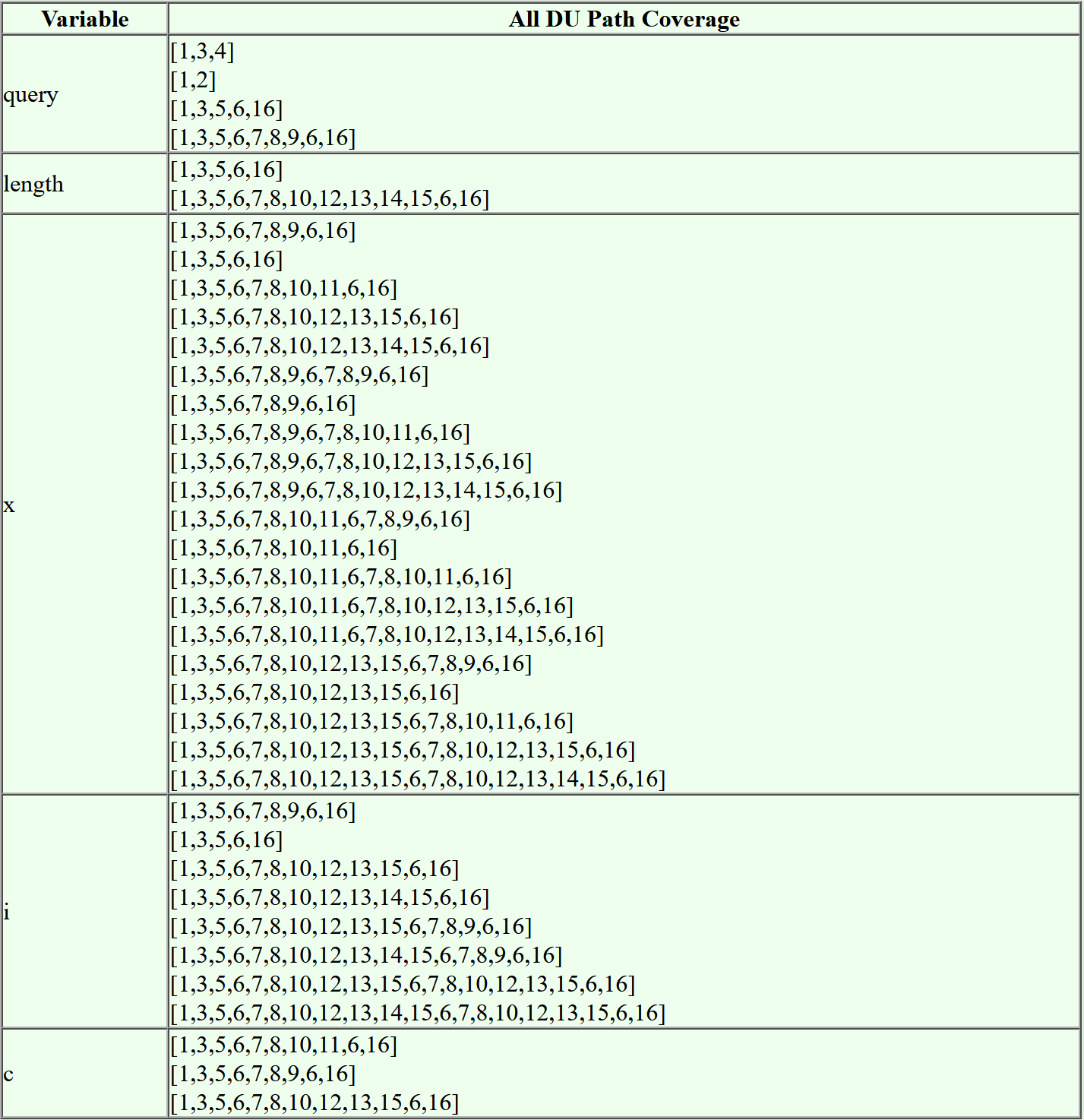


## All-Du-Paths Coverage

No *All-Du-Paths* *Coverage* para o *longestPrefixOf*, voltou-se a usar o mesmo grafo já criado anteriormente, mas desta vez foi também analisado em que *nodes* e *edges* cada variável é definida e usada, obtendo a seguinte tabela:

|  |  |  |
| --- | --- | --- |
| Nodes & Edges: I | def(I) | use(I) |
| 1 | {query} |  |
| (1,2), (1,3) |  | {query} |
| 3 |  |  |
| (3,4), (3,5) |  | {query} |
| 5 | {len, x, i} |  |
| (5,6) |  |  |
| 6 |  |  |
| (6,7), (6,16) |  | {x, i, query} |
| 7 | {c} | {query, i} |
| (7,8) |  |  |
| 8 |  |  |
| (8,9), (8,10) |  | {c, x} |
| 9 | {x} | {x} |
| (9,6) |  |  |
| 10 |  |  |
| (10,11), (10,12) |  | {c, x} |
| 11 | {x} | {x} |
| (11,6) |  |  |
| 12 | {i} | {i} |
| (12,13) |  |  |
| 13 |  |  |
| (13,14), (13,15) |  | {x} |
| 14 | {len} | {i} |
| (14,15) |  |  |
| 15 | {x} | {x} |
| (15,6) |  |  |
| 16 |  | {query, len} |

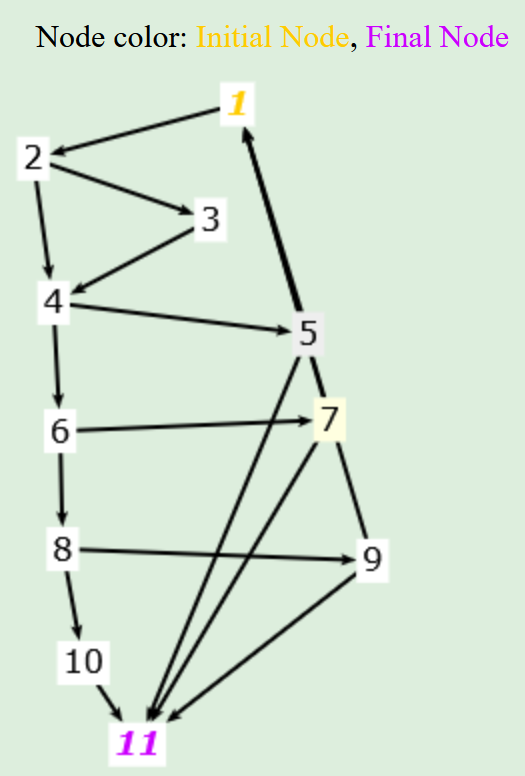
Posteriormente, foi inserido no site <https://cs.gmu.edu:8443/offutt/coverage/DFGraphCoverage> as edges, initial node, final nodes, defs e uses das variáveis e gerado os test paths para cada variável:



Algumas variáveis tinham test path iguais, fazendo com que um mesmo path conseguisse satisfazer diversos requisitos, então não foi necessário escrever tantos testes unitários quantos os gerados.

## All-Coupling-Use-Paths Coverage

Para satisfazer o *All-Coupling-Use-Paths*, foi necessário analisar o método *put* e criar uma representação dele em grafo. As edges dos nodes 5,7 e 9 para o node 1 representam as chamadas recursivas.



Com o grafo feito, foi necessário analisar em que nodes são feitas as últimas definições das variáveis antes das chamadas recursivas, e também em que nodes da função chamada são primeiramente usadas as variáveis passadas, obtendo a seguinte tabela:

|  |  |
| --- | --- |
| Last-def | First-use |
| key: {1} | key: {i} |
| val: {1} | val: {v, vii, ix, x} |
| x: {1,3} | x: {ii} |
| d: {1,9} | d: {i} |
| x' : {v, vii, ix, x} | x' : {11} |

Com estas informações, foi então possível verificar que paths são necessários os testes percorrerem de modo a satisfazer a coverage *All-Coupling-Use-Paths*:

Paths:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| key | key {1} -> key {i} |  |  |  |
| val | val {1} -> val {v} | val {1} -> val {vii} | val {1} -> val {ix} | val {1} -> val {x} |
| x | x {1} -> x {ii} | x {3} -> x {ii} |  |  |
| d | d{1} -> d{i} | d{9} -> d{i} |  |  |
| x' | x’{v} -> x’{11} | x’{vii} -> x’{11} | x’{ix} -> x’{11} | x’{x} -> x’{11} |

## Logic-based Coverage

Como o método *longestPrefixOf* tem predicados simples, onde apenas um deles tem mais que uma cláusula, foi decidido não complicar demasiado e escolher um dos critérios básicos. Neste caso, foi escolhido o *Combinatorial* *Coverage* por ser o mais abrangente sem complicar demasiado a realização dos testes.

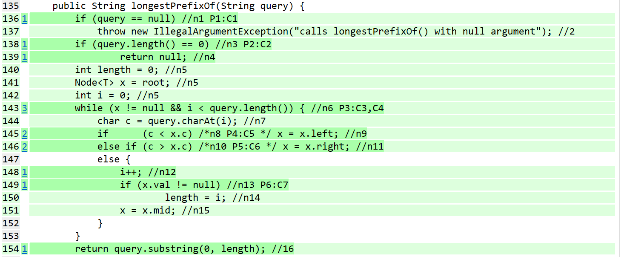
## Base Choice Coverage

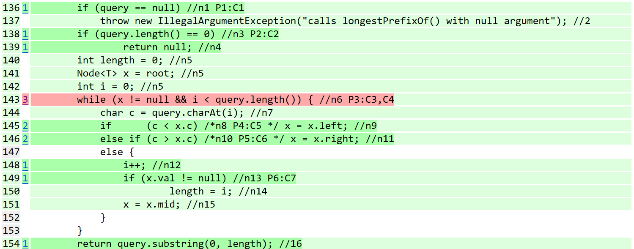
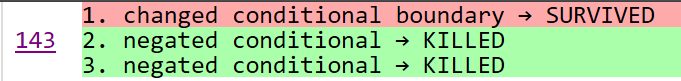
No *Base* *Choice* *Coverage*, cada uma das características foi subdividida binariamente, com exceção da última que foi ternariamente. Foi escolhido um *Base* *Choice* de acordo com o caso que se achou mais comum:

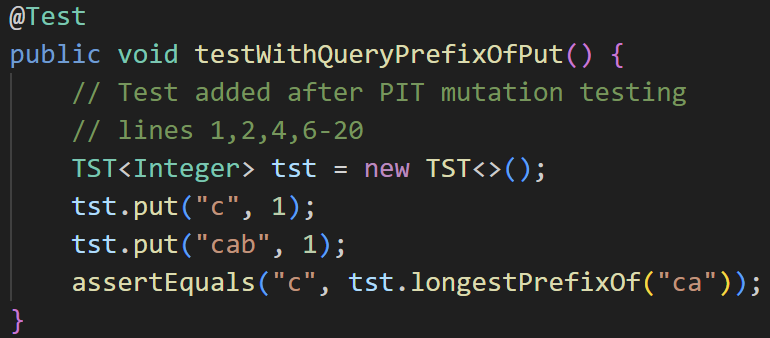
1. A nova *key* não estar ainda na *Trie*
2. Ainda não existir um prefixo da nova *key* na *Trie*
3. A *Trie* não estar vazia
4. A nova *key* ser uma *key* típica lexicograficamente

Foi feito um teste para a *Base* *Choice* e também testes para cada alteração de característica da *Base* *Choice*.

## PIT Mutation Coverage

Ao correr as mutações do programa através do PIT, foi possível em (quase) todas as *Coverage* *Criteria* do *longestPrefixOf* matar os mutantes gerados:

Com exceção da *Line* *Coverage* que não foi capaz de matar o seguinte mutante:

Este mutante poderia ter sido detetado e morto com a adição do seguinte teste:

# JUnit QuickCheck

Para realizar os testes de QuickCheck, foi necessário acrescentar três métodos ao sistema: *equals*(), *delete*() e *clone*(). Foi também preciso criar três classes de geradores: *TrieGenerator*, *KeyGenerator* e *KeyListGenerator*.

Para testar a propriedade “The order of insertion of different keys does not change the final tree value” foi necessário receber uma *Trie*, uma lista de *Keys*, e um valor gerado aleatoriamente. Foi inicialmente clonada a Trie recebida e de seguida foram adicionadas as *Keys* à *Trie* original. Depois a ordem das *Keys* foi baralhada e foram adicionadas à *Trie* clonada. Por fim, comparou-se a *Trie* original com a *Trie* clonada.

Para testar a propriedade “If you remove all keys from a tree, the tree must be empty” foi necessário receber uma *Trie* gerada aleatoriamente. Foram removidas todas as *Keys* presentes na *Trie*, e de seguida comparou-se a *Trie* com uma *Trie* iniciada vazia, e também se verificou que o seu *size* é 0.

Para testar a propriedade “Given a tree, inserting and then removing the same key value will not change its initial value” foi necessário receber uma *Trie*, uma *Key*, e um valor gerado aleatoriamente. Foi inicialmente clonada a *Trie* recebida, e de seguida foi adicionada a *Key* com o dado valor, e removida. Por fim comparou-se a *Trie* inicial (clonada) com a *Trie* final. Esta **propriedade não é verdadeira** para *Tries* que já contenham o valor a ser adicionado/removido, pois, após a adição/remoção, a *Trie* inicial seria diferente da final.

Para testar a propriedade “Selecting a stricter prefix keysWithPrefix returns a strict subset result” foi necessário receber uma *Trie*, uma lista de *Keys*, e um valor gerado aleatoriamente. Foram inicialmente buscadas todas as *Keys* da *Trie*, e uma delas foi selecionada aleatoriamente para ser a *Key* base. De seguida, foram adicionadas à *Trie* novas *Keys* aleatórias todas com o prefixo da *Key* base e guardou-se o *keysWithPrefix(base\_key).* Por fim, foi verificado que com todos os prefixos possíveis da *Key* base (removendo caracter a caracter) obtém-se sempre um subconjunto de *keysWithPrefix(base\_key).*

# Conclusão:

Executando todos os testes, o sistema ficou com uma coverage total de 80,2%. Faltou testar o método privado *collect* e os métodos, posteriormente adicionados, *equals* e *delete*. O método privado *get* tem uma linha de código impossível de alcançar, pois é relacionada com um check já feito previamente no seu método público.

