# 6.
# Experimentos

Este capítulo apresentará experimentos realizados sobre a ferramenta *SDiff*. Inicialmente faremos a comparação dos resultados da ferramenta *SDiff* com os resultados obtidos por uma ferramenta tradicional. Em seguida, faremos algumas observações sobre experimentos realizados com arquivos fonte de aplicações reais. Finalmente, apresentaremos algumas estatísticas sobre a execução da ferramenta *SDiff*.

## 6.1
## Comparação dos resultados com os de uma ferramenta tradicional

Em cada uma das subseções seguintes será discutido um caso de evolução de um trecho de código C++, cujas versões inicial e final foram comparadas pela ferramenta *SDiff* e por uma ferramenta tradicional de comparação textual chamada *KDiff3* [EIBL, 2003], comumente usada em ambientes Unix. Para cada caso explicaremos brevemente a evolução, apresentaremos o resultado de ambas as ferramentas e discutiremos os resultados.

### 6.1.1 Inserção e remoção simples

O primeiro experimento é trivial, mostrando que a ferramenta identifica inserções e remoções como outra qualquer.

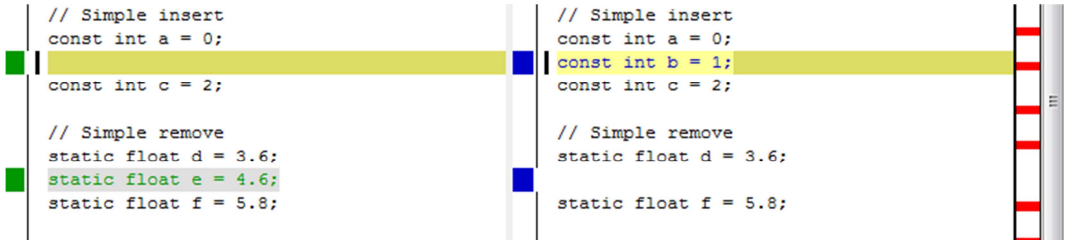


**Figura 31 – Exemplo de inserção e remoção mostrado na aplicação *KDiff3*.**


**Figura 32 – Exemplo de inserção e remoção mostrado na aplicação *SDiff*.**

Como se pode ver nas Figuras 31 e 32 as ferramentas identificaram com a mesma precisão a inserção e a remoção de declarações idênticas.

### 6.1.2 Modificação

Este experimento avaliará a capacidade dos mecanismos de comparação de identificar trechos modificados em elementos. O exemplo consiste na modificação de duas *strings* utilizadas como parâmetro da função *printf*. A primeira string trocará um trecho por outro, e a segunda fará a inserção e remoção de trechos.

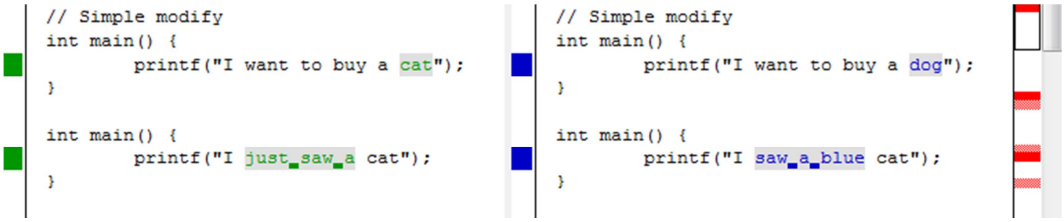


**Figura 33 – Exemplos de modificação mostrado na aplicação *KDiff3*.**


**Figura 34 – Exemplos de modificação mostrado na aplicação *SDiff*.**

Como pôde ser observado, ambos os mecanismos de comparação foram capazes de identificar corretamente a alteração da primeira *string*. Porém, na segunda, a ferramenta *KDiff3* não identificou corretamente as alterações, tentando inclusive unificá-las em uma só. A ferramenta *SDiff* identificou corretamente as operações de inserção e remoção, indicando com precisão cada diferença. Uma observação importante é que em ambos os casos o mecanismo de comparação

utilizado foi o textual, porém ainda assim a ferramenta *SDiff* encontrou um resultado melhor.

Além dos resultados da comparação textual, a ferramenta *SDiff* marcou os elementos sintáticos que contém as diferenças identificadas, ou seja, indicou que o elemento que representa o primeiro e único parâmetro da função *printf* foi modificado. Este recurso torna mais confortável a leitura pois o usuário identifica primeiro o elemento que foi alterado, e somente depois o que mudou nele.

### 6.1.3 Movimentação

Este experimento avaliará a capacidade dos mecanismos de comparação de identificar trechos de código reorganizados dentro da sua estrutura. O exemplo consiste em duas estruturas *enum* em que um dos seus elementos é trocado de posição. A diferença entre as duas estruturas é que na segunda o elemento, além de ser trocado de posição, é ligeiramente modificado.

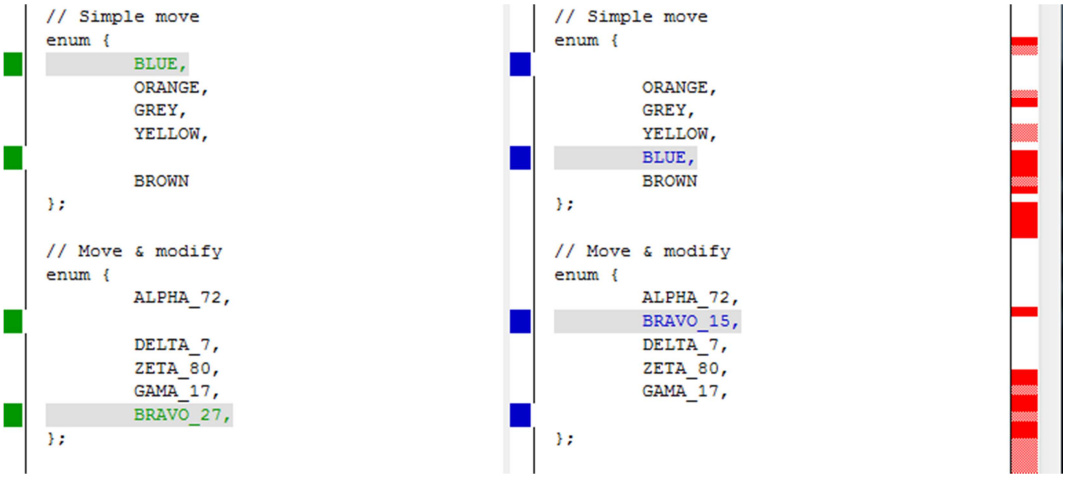


**Figura 35 – Exemplos de movimentação mostrado na aplicação *KDiff3*.**

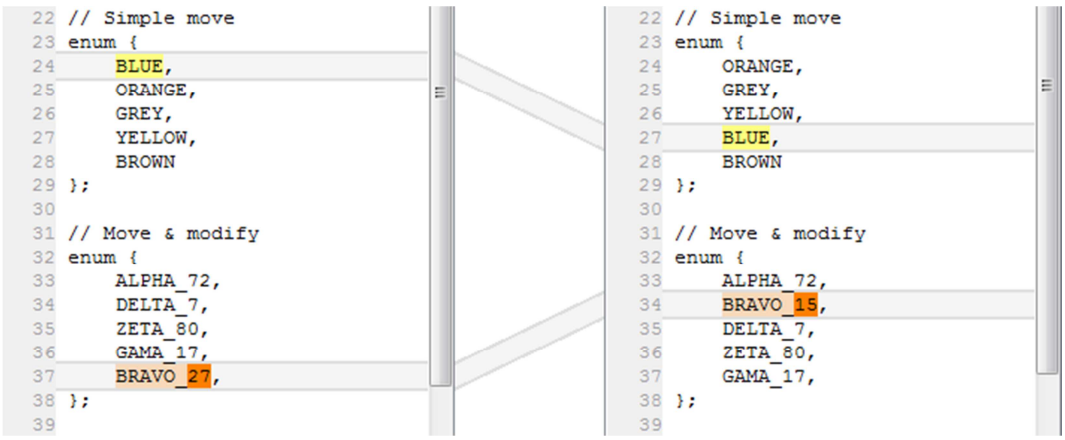


**Figura 36 – Exemplos de movimentação mostrado na aplicação *SDiff*.**

Como se pode observar a ferramenta *KDiff3* indicou uma inserção e uma remoção para ambos os casos. É um resultado pouco preciso expressar uma movimentação desta forma, pois mesmo que computacionalmente produza o mesmo resultado, não condiz com a semântica esperada pelo usuário.

Contudo, a ferramenta *SDiff* identificou que os elementos eram equivalentes e apresentou uma única operação de movimentação, para cada caso. Deve-se dar mais importância ao segundo caso, em que mesmo com uma pequena alteração no elemento, a ferramenta foi capaz de identificar a equivalência.

### 6.1.4 Alteração no contexto

Este experimento avaliará a capacidade dos mecanismos de comparação de identificar trechos de código cujo contexto externo tenha sido modificado. O exemplo consiste em duas classes em que um dos seus membros troca de visibilidade. O segundo caso explora este contexto combinado com uma modificação na estrutura, a fim de mostrar visualmente como este resultado é apresentado. E, em ambos os casos promovemos a movimentação de um atributo mantendo sua visibilidade, a fim de comparar os tipos de diferenças.

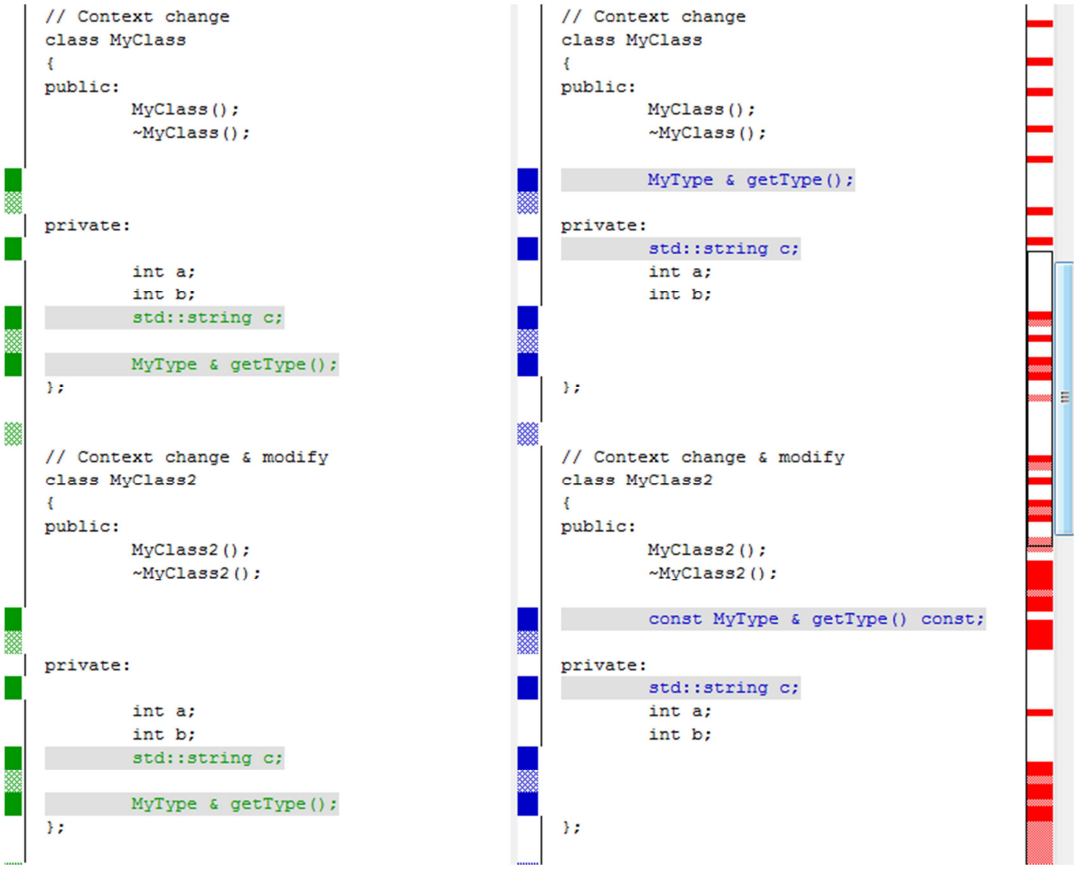


**Figura 37 – Exemplos de alteração de propriedades contextuais mostrado na aplicação *KDiff3*.**

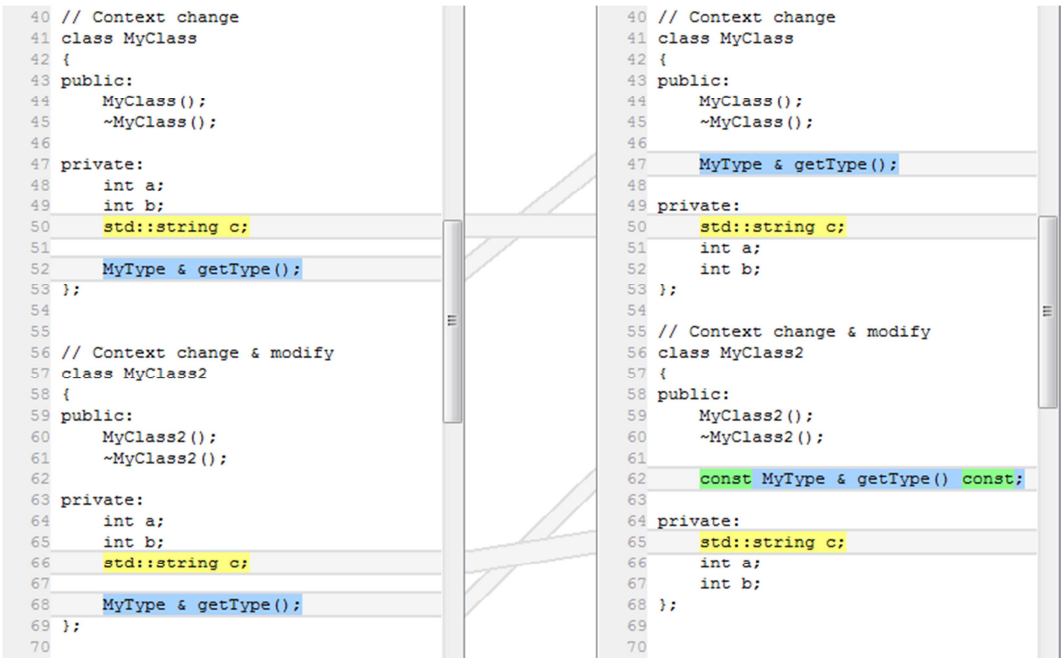


**Figura 38 – Exemplos de alteração de propriedades contextuais mostrado na aplicação *SDiff*.**

Como se pôde observar a ferramenta *KDiff3* novamente indicou ambas as operações como inserções e remoções em vez de identificar para cada caso: (1) uma movimentação combinada com uma troca de visibilidade (método *getType*); (2) uma movimentação simples (atributo *c*). Naturalmente, devido a limitações do mecanismo de comparação textual, esta ferramenta também não notificou que o método *getType* tornou-se publico.

A ferramenta *SDiff* identificou corretamente ambas as diferenças e marcou a operação de movimentação com troca de contexto de uma cor diferente, a fim de expressar claramente que não foi uma simples movimentação, ocorreu também uma troca no contexto externo do elemento.

O segundo caso aproveita o exemplo para mostrar que a ferramenta *SDiff* é capaz de identificar movimentações com troca de contexto em que o elemento também tenha sido alterado. E, além de exibir com precisão as propriedades da movimentação com troca de visibilidade, também exibe com precisão as operações de inserção realizadas sobre este elemento.

### 6.1.5 Reformatação

Este experimento demonstrará uma das principais vantagens da comparação sintática: a capacidade de ignorar a formatação do código por comparar apenas a estrutura sintática dos trechos de código. Neste exemplo serão apresentados dois casos em que uma declaração sofreu reformatação, sendo o primeiro sem nenhuma alteração, e o segundo com o tipo de um dos parâmetros modificado.

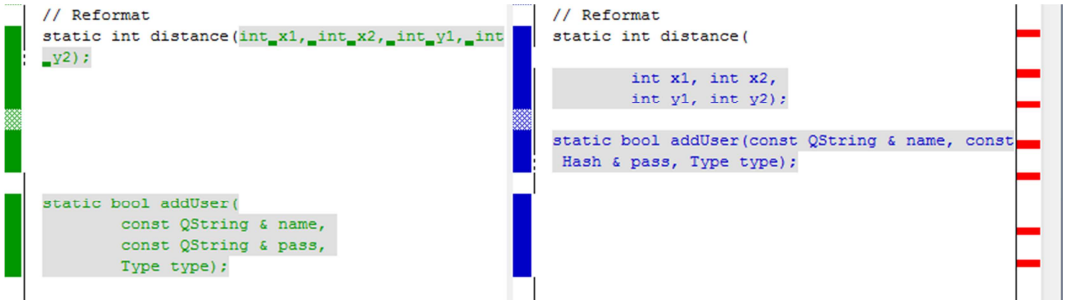


**Figura 39 – Exemplos de reformatação mostrado na aplicação *KDiff3*.**


**Figura 40 – Exemplos de reformatação mostrado na aplicação *SDiff*.**

Conforme esperado, a ferramenta de comparação textual informa que ambos os trechos de código foram totalmente alterados, dificultando a identificação das alterações sintáticas.

A ferramenta *SDiff* ignora a reformatação em ambos os casos, identificando apenas a modificação do tipo do parâmetro *pass* no segundo caso. Ou seja, ela foi capaz de identificar com precisão que a única alteração relevante para o usuário tinha sido a troca do tipo do parâmetro.

### 6.1.6 Comparação de elementos com tipos diferentes

Este experimento demonstrará que a comparação sintática respeita o tipo dos elementos, impedindo que elementos de tipos diferentes sejam comparados. O exemplo consiste na remoção de um atributo seguido da inserção de um método, porem ambos com um alto grau de similaridade textual.

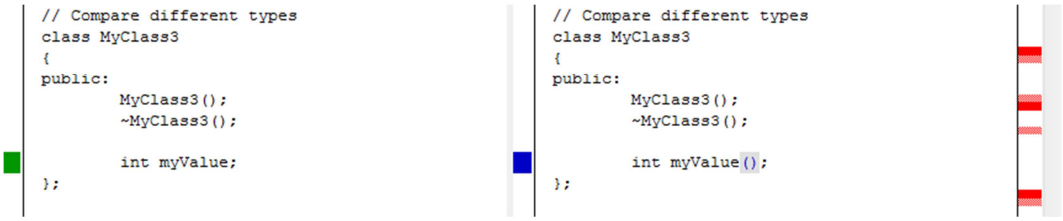


**Figura 41 – Exemplo de comparação de elementos sintáticos com tipos diferentes mostrado na aplicação *KDiff3*.**

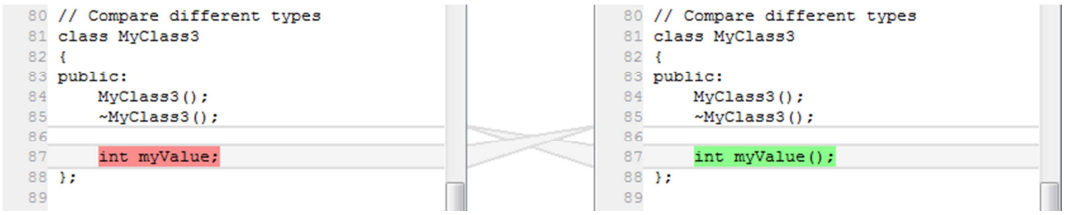


**Figura 42 – Exemplo de comparação de elementos sintáticos com tipos diferentes mostrado na aplicação *SDiff*.**

Conforme esperado, a ferramenta *KDiff3* identificou que um par de parêntesis havia sido inserido. Contudo, a ferramenta *SDiff* não considerou que a inserção seguida de remoção representava uma modificação, pois os elementos comparados possuíam tipos diferentes. Logo, o resultado apresentado pela ferramenta *SDiff* foi a remoção de um atributo e a inserção de um novo método.

### 6.1.7 Extração de código

Este experimento demonstrará que o mecanismo de comparação sintática respeita os limites dos elementos envolvidos na comparação. O exemplo consiste em um trecho de código onde foi aplicada uma operação de extração de código para um novo método.

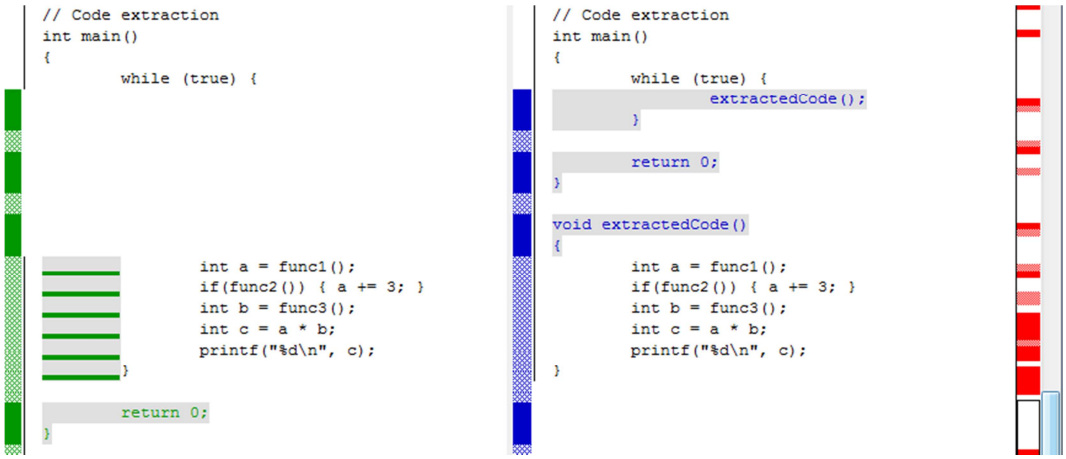


**Figura 43 – Exemplo de extração de código para um novo método mostrado na aplicação *KDiff3*.**

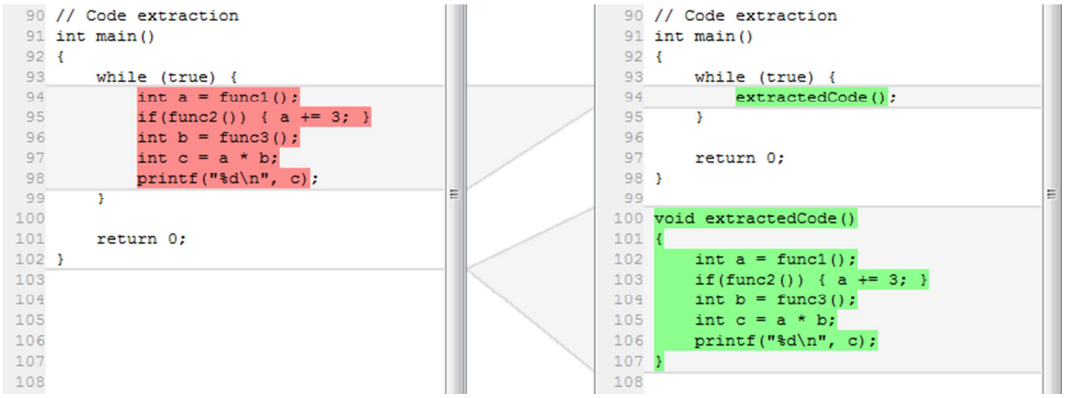


**Figura 44 – Exemplo de extração de código para um novo método mostrado na aplicação *SDiff*.**

Conforme esperado, a ferramenta *KDiff3* indicou a menor distância de edição textual, desrespeitando os limites sintáticos dos elementos. Este resultado é pouco legível por um usuário final, por não expressar a semântica associada as operações lógicas realizadas durante a evolução do trecho de código.

Entretanto, a ferramenta *SDiff* identificou com a melhor precisão possível as operações realizadas, indicando que o trecho de código extraído foi substituído por uma chamada de método, e, um novo método foi inserido ao final do trecho.

## 6.2
## Experimentos em aplicações reais

Durante os experimentos aplicamos a ferramenta *SDiff* sobre versões consecutivas de arquivos fonte de aplicações reais, e, conforme esperado, as diferenças foram identificadas com um alto nível de precisão, comparando-as com as diferenças resultantes das ferramentas tradicionais. Nestes experimentos utilizamos módulos com um número de linhas de código consideravelmente grande (de 1 a 5 KLOC), a fim de verificar a robustez da ferramenta criada. Outra característica relevante destes módulos é a presença de símbolos não definidos por eles, ou seja, a existência de dependências não resolvidas, comprovando que a ferramenta é capaz de comparar separadamente cada arquivo de código de uma aplicação.

Porém, observamos também duas limitações na implementação. A primeira corresponde ao formato dos blocos de comentário: durante o desenvolvimento da aplicação os exemplos foram escritos utilizando comentários multi-linha (Figura 45), porém, ao experimentar comparar um código utilizando blocos formados por comentários simples (Figura 46), em que apenas ocorreu a movimentação de uma declaração com seu respectivo comentário, o resultado foi confuso, pois o algoritmo de comparação considerou cada linha do comentário como um elemento separado, e, conforme esperado, acusou modificações em somente parte das linhas do comentário, mantendo o restante inalterado. Apesar deste resultado ser logicamente correto, não corresponde a semântica das alterações.

```
1  /*********************************************
2  *
3  * @brief       Realiza a soma de dois inteiros
4  *
5  * @param a     Primeiro inteiro
6  * @param b     Segundo inteiro
7  *
8  * @return      Resultado da soma
9  *
10 *********************************************/
11
```

**Figura 45 – Exemplo de bloco formado por comentário multi-linha.**

```
1  ////////////////////////////////////////////////
2  ///
3  /// @brief     Realiza a soma de dois inteiros
4  ///
5  /// @param a   Primeiro inteiro
6  /// @param b   Segundo inteiro
7  ///
8  /// @return    Resultado da soma
9  ///
10 ////////////////////////////////////////////////
11
```

**Figura 46 – Exemplo de bloco formado por comentário simples.**

Este comportamento pode ser explicado analisando a estrutura canônica resultante: cada bloco de comentário, formado por comentários simples, é representado como uma sequência de elementos dissociados, porém, semanticamente, deveriam ser considerados um único elemento sintático. Uma possível solução para este problema é agrupar comentários consecutivos em um novo tipo de elemento (*CommentBlock*). Aproveitando, esta solução pode ser incrementada analisando e extraindo a estrutura sintática de cada bloco de comentário, quando estes forem escritos utilizando um padrão como Javadoc [ORACLE, 2010] ou Doxygen [HEESCH, 1997]. Para tanto, é necessário criar suporte a documentos escritos em mais de uma sintaxe. Em suma, este é um trabalho futuro bastante relevante, pois existem tipos de documentos que são comumente escritos em mais de uma linguagem, como por exemplo, arquivos fonte de aplicações web, onde é comum misturar linguagens como HTML e Javascript com linguagens de programação Java, C#, PHP, etc.

A segunda limitação observada, especificamente no suporte a linguagem C++, é a utilização de macros que modificam a sintaxe da linguagem. O interpretador implementado classifica macros como chamadas de função, porém, macros cuja utilização não respeite a sintaxe da linguagem, como por exemplo, a macro *foreach* fornecida pela ferramenta Qt, impossibilita que o código seja interpretado. Para eliminar esta limitação seria necessário conhecer todas as macros definidas, porém, isso implica em conhecer símbolos externos, o que vai contra um dos objetivos deste trabalho. Logo, para esta ferramenta especificamente, é preferível manter a limitação, pois esta afeta uma porção pequena de código e pode ser resolvida adotando um padrão de criação de macros que não modifiquem a sintaxe da linguagem. Em uma evolução da ferramenta, acoplando-a a uma IDE, é possível criar um mecanismo que faça um pré-

processamento em todos os arquivos do projeto, com o objetivo de cadastrar as macros encontradas.

## 6.3 Estatísticas da execução da ferramenta SDiff

Esta seção discutirá algumas medições realizadas sobre a execução do algoritmo de comparação. Aplicamos a ferramenta *SDiff* a diferentes instâncias, consistindo cada uma em um par de arquivos A e B, representando diferentes versões de um mesmo documento. O formato de escrita destes documentos segue o mesmo padrão de codificação, a fim de evitar que este influencie nas medições. Estas instâncias foram dispostas em ordem crescente de tamanho para facilitar a avaliação do comportamento do algoritmo. Na tabela 4 apresentamos, para cada arquivo de uma instância, o seu número de linhas, o número de linhas do texto XML resultante da sua conversão para a forma canônica, e, o número de elementos em cada árvore sintática construída.



| | Número de linhas no documento | | Número de linhas no XML | | Número de elementos na árvore sintática | |
|---|---|---|---|---|---|---|
| | A | B | A | B | A | B |
| **1** | 43 | 58 | 135 | 152 | 101 | 115 |
| **2** | 287 | 298 | 2.461 | 2.409 | 1.556 | 1.522 |
| **3** | 411 | 407 | 3.297 | 2.962 | 2.122 | 1.928 |
| **4** | 663 | 712 | 5.197 | 5.210 | 3.374 | 3.403 |
| **5** | 1.057 | 1.074 | 9.061 | 9.161 | 5.774 | 5.832 |
| **6** | 1.683 | 1.723 | 12.157 | 12.762 | 7.777 | 8.143 |
| **7** | 2.019 | 2.201 | 12.714 | 14.139 | 8.216 | 9.138 |
| **8** | 2.543 | 2.587 | 15.228 | 16.135 | 9.878 | 10.424 |
| **9** | 3.214 | 3.313 | 20.302 | 22.285 | 13.155 | 14.402 |
| **10** | 4.126 | 4.117 | 26.729 | 27.182 | 17.359 | 17.582 |

**Tabela 4 – Estatísticas relacionadas ao tamanho de cada instância.**

A partir desta tabela criamos o gráfico da Figura 47, que apresenta a dependência linear entre o número de linhas de código no documento (tamanho da instância) e o número de linhas no XML resultante.

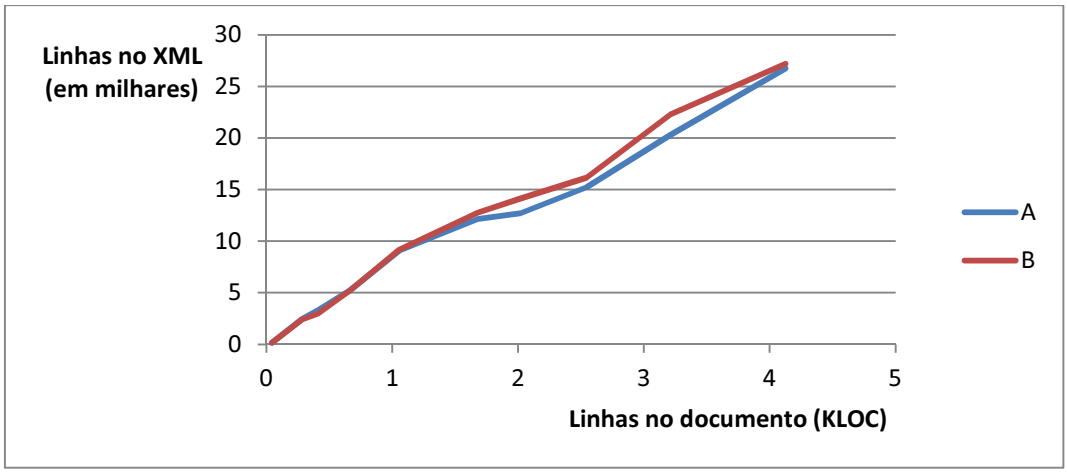


**Figura 47 – Número de linhas na representação XML versus Número de linhas no documento.**

Em seguida, criamos o gráfico da Figura 48, que também apresenta uma dependência linear entre o tamanho da instância e o número total de elementos sintáticos criados.

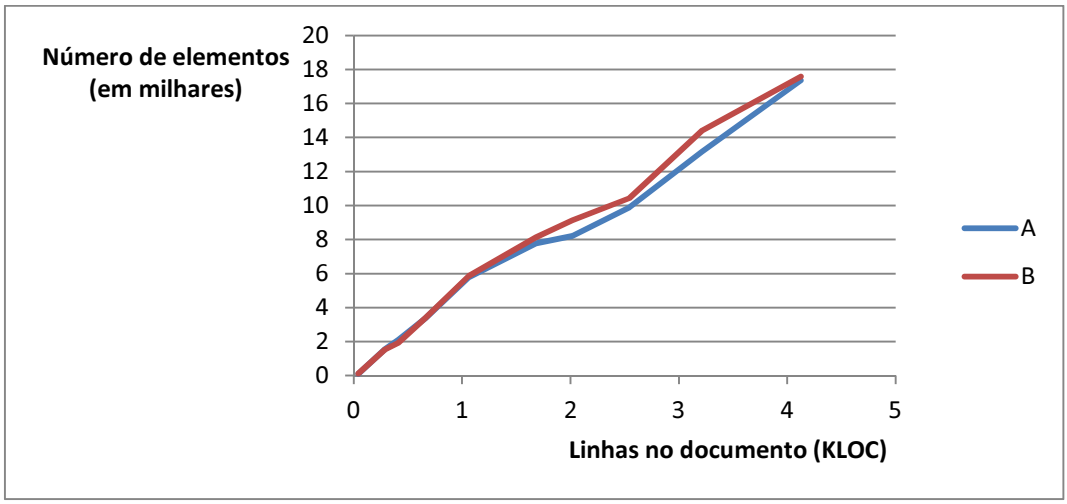


**Figura 48 – Número total de elementos sintáticos versus Número de linhas no documento.**

Outra medição computada foi o número de comparações realizadas, e, o número de comparações descartadas, devido a diferença entre os tipos dos elementos envolvidos. Estes resultados são apresentados na Tabela 5.

| | Número de comparações | | | |
|---|---|---|---|---|
| | **Realizadas** | **Descartadas** | **Realizadas / Descartadas** | **Todos de A X Todos de B** |
| **1** | 373 | 722 | 0.52 | 11.615 |
| **2** | 3273 | 6.685 | 0.49 | 2.368.232 |
| **3** | 14.844 | 26.575 | 0.56 | 4.091.216 |
| **4** | 80.427 | 112.283 | 0.72 | 11.481.722 |
| **5** | 164.618 | 232.975 | 0.71 | 33.673.968 |
| **6** | 301.232 | 478.908 | 0.63 | 63.328.111 |
| **7** | 494.442 | 911.310 | 0.54 | 75.077.808 |
| **8** | 761.368 | 1.524.932 | 0.50 | 102.968.272 |
| **9** | 1.242.894 | 2.624.442 | 0.47 | 189.458.310 |
| **10** | 2.195.522 | 4.615.683 | 0.48 | 305.205.938 |

**Tabela 5 – Estatísticas relacionadas ao número de comparações em cada instância.**

Conforme esperado, o número de comparações descartadas é relativamente alto quando comparado ao número de comparações realizadas. Comprovamos que a complexidade do nosso algoritmo é reduzida ao verificar que a média das razões apresentada na coluna 3 desta tabela é exatamente o valor 0.56, com desvio padrão de 0.09, ou seja, para cada comparação realizada, em média, duas são descartadas, inclusive evitando a recursão em suas sub-árvores.

A quarta coluna desta tabela apresenta o número de comparações que seriam realizadas caso todos os elementos de uma árvore fossem comparados com todos os elementos da outra árvore. Esta informação é apresentada somente para fazer sua comparação com o resultado da nossa ferramenta, onde é levado em consideração o tipo e a disposição dos elementos na estrutura sintática. O resultado desta comparação é apresentado no gráfico da Figura 49, onde comprovamos que a comparação respeitando as propriedades sintáticas produz um número de comparações significativamente menor.

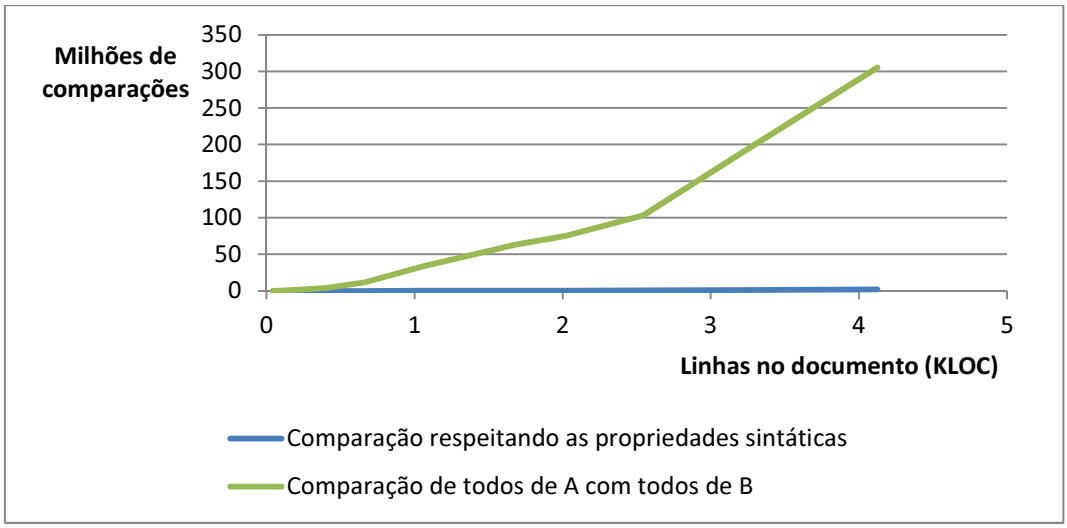


**Figura 49 – Número de comparações versus Número de linhas no documento.**

Finalmente, apresentamos na Tabela 6 as medições relacionadas ao tempo de execução do algoritmo sobre as instâncias. Nesta tabela apresentamos o tempo de carga de cada um dos documentos, ou seja, o custo para transformá-los na sua representação canônica. A terceira coluna apresenta a informação mais relevante, o tempo gasto para comparar as árvores sintáticas de cada instância, e, na quarta coluna, apresentamos o tempo total gasto para realizar a comparação dos arquivos A e B.

| | Tempo | | | |
|---|---|---|---|---|
| | **Carga de A (ms)** | **Carga de B (ms)** | **Comparação (ms)** | **Total (s)** |
| **1** | 12 | 12 | 2 | 0.03 |
| **2** | 174 | 167 | 21 | 0.36 |
| **3** | 228 | 204 | 71 | 0.5 |
| **4** | 359 | 356 | 397 | 1.11 |
| **5** | 622 | 632 | 518 | 1.77 |
| **6** | 826 | 869 | 1008 | 2.7 |
| **7** | 805 | 927 | 1749 | 3.48 |
| **8** | 938 | 1009 | 2249 | 4.196 |
| **9** | 1310 | 1465 | 4243 | 7.02 |
| **10** | 1716 | 1802 | 8076 | 11.59 |

**Tabela 6 – Apresenta os tempos de execução para cada instância.**

A fim de visualizar com clareza o comportamento dos valores apresentados nesta tabela, criamos o gráfico da Figura 50, em que relacionamos os tempos de

carga dos arquivos A e B e o tempo de comparação das suas respectivas árvores sintáticas com o número de linhas de cada instância.

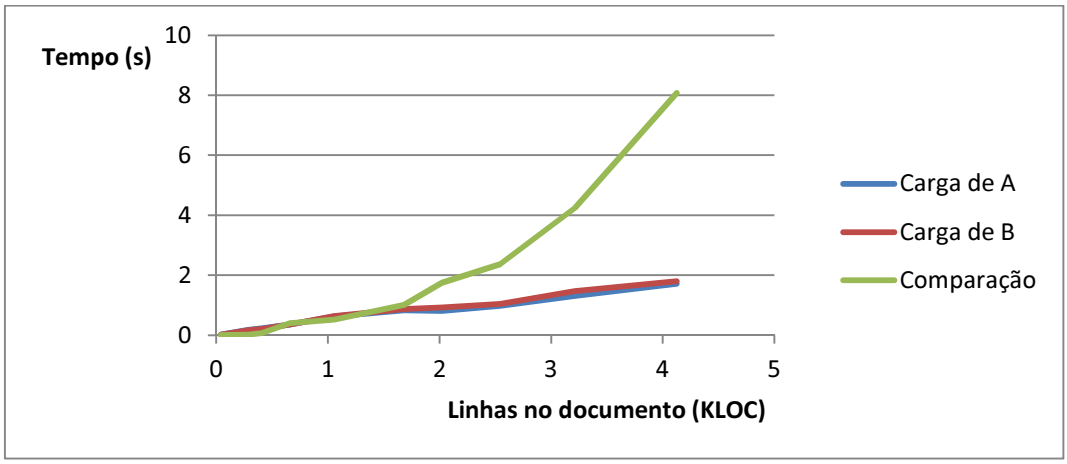


**Figura 50 – Tempo de execução versus Número de linhas no documento.**

Concluímos a partir deste gráfico que os tempos de carga apresentam uma dependência linear com o tamanho da instância, contudo, o tempo de comparação apresenta uma dependência não-linear. Entretanto, considerando que as boas práticas de codificação sugerem a criação de módulos pequenos, temos que este resultado é adequado para o contexto da ferramenta criada, permitindo que arquivos contendo até 2 KLOC sejam comparados em menos de 2 segundos. Além disso, consideramos como um trabalho futuro otimizar este algoritmo, buscando soluções para reduzir ainda mais a sua complexidade.