Ampliando o apoio à comparação de artefatos via aplicação recursiva do algoritmo de LCS

Fernanda F. Silva, Felipe V. Duval, Leonardo G. P. Murta

Instituto de Computação – Universidade Federal Fluminense (UFF)
CEP: 24210-240 – Niterói – RJ - Brasil

{ffloriano,fduval,leomurta}@ic.uff.br

Abstract. *Software Development Processes have become increasingly complex, with the use of distributed applications and collaborative work. Thus, it is essential to improve the concurrency control support in these scenarios, providing better ways to compare different user contributions. This paper presents an approach that aims at supporting this by obtaining the differences between files, drawing not only the additions and removals made in different versions of an artifact, but also an indicating moves. Moreover, the proposed approach allows changing the granularity, minimum movement size, and recursion levels.*

Resumo. *Processos de desenvolvimento de software têm se tornado cada vez mais complexos, com o uso de aplicações distribuídas e trabalho colaborativo. Com isso, torna-se indispensável o aperfeiçoamento do apoio ao controle de concorrência nestes cenários, proporcionando melhores maneiras de comparar diferentes contribuições dos usuários. Este trabalho apresenta uma abordagem que visa este apoio obtendo as diferenças entre os arquivos, extraindo não somente as adições e exclusões realizadas em versões diferentes de um artefato, como também a indicação de movimentações. Além disso, a abordagem proposta permite alterações de granularidades, tamanho mínimo das movimentações e níveis de recursão.*

Palavras-chave: DIFF, LCS, Programação Dinâmica

1. Introdução

Sistemas grandes e complexos, aplicações distribuídas e trabalho colaborativo compõem cenários comuns nos dias de hoje. Com isso, diante de mudanças e melhorias impreteríveis, torna-se cada vez mais necessária a utilização de técnicas apropriadas para controle e gestão de artefatos. A manutenção e evolução de software buscam a preservação da integridade destes artefatos, levando em consideração essas novas tendências e práticas [Bennett e Rajlich 2000].

Diante da necessidade de detecção de divergências entre artefatos devido a edições em paralelo por diferentes desenvolvedores, surgiram algoritmos e ferramentas para esse fim. É possível citar, dentre outras, *diff2* [J. W. Hunt 1976], *diff3* [U.Asklund 1994], *diff sintático* [Buffenbarger 1995] e *diff semântico* [Berzins 1994]. Contudo, a técnica de *diff* aplica tradicionalmente o algoritmo de maior subsequência comum, do inglês, Longest Commom Subsequence (LCS) [Cormem , Leiserson e Stein 1998], que detecta adições e exclusões entre os artefatos sendo comparados, ignorando possíveis movimentações.

Para exemplificar o funcionamento geral do algoritmo LCS, e consequentemente o funcionamento geral da grande maioria dos algoritmos de *diff*, suponha que a sua execução seja representada por: *LCS [V_1(...), V_2(...)],* onde as sequências V_1 e V_2 representam versões distintas de um mesmo artefato ou mesmo versões de artefatos diferentes, com seus conteúdos descritos entre parênteses. O resultado da execução do algoritmo fornece a maior sequência, contínua ou não, que permeia ambas as versões passadas como argumento. Desta forma, considerando *V_1 (A, B, C, D, E, F)* e *V_2 (E, C, H, D, I, F, A, B, G)*, é possível identificar visualmente que a maior sequência comum é *(C, D, F)*, como representado a seguir:

LCS [V_1 (A, B, C, D, E, F), V_2 (E, C, H, D, I, F, A, B, G)] = (C, D, F)

Na utilização convencional do algoritmo LCS, os conteúdos *V_1 (A, B, E)* e *V_2 (E, H, I, A, B, G)* são apontados como divergências entre as versões, na forma de adições e exclusões, porém tanto (A, B) quanto (E) poderiam ser tratados como movimentações, ou seja, mudanças no posicionamento do conteúdo apresentado.

Como é possível observar, movimentações podem ser consideradas como exclusões seguidas de adições. Contudo, a vantagem de considerar explicitamente os casos de movimentações é evitar ciclos de revisão desnecessários em trechos já estáveis do artefato e permitir uma identificação mais precisa de quem foi de fato o responsável por modificações em partes do artefato. Ou seja, o conteúdo movido não foi feito por quem moveu, mas sim por quem adicionou. A mistura desses conceitos leva a uma imprecisão na identificação da autoria das partes dos artefatos.

Desta forma, o objetivo deste artigo é apresentar uma nova abordagem para detecção de diferenças considerando movimentações entre duas versões de um artefato de software. Para isto, é aplicada a técnica de *diff2 textual*, que aumenta a abrangência da abordagem para diferentes linguagens de programação, apoiada na aplicação recursiva do algoritmo LCS. Além disso, permite alterações de granularidades, tamanho mínimo das movimentações e níveis de recursão.

O restante do artigo está organizado em quatro seções. Na Seção 2 são apresentados conceitos básicos sobre algoritmos de *diff* e programação dinâmica. A Seção 3 descreve a abordagem proposta para detecção de movimentações com o uso de LCS recursivo. A Seção 4 exemplifica a utilização da abordagem. A seção 5 apresenta trabalhos relacionados. Finalmente, a Seção 6 apresenta as considerações finais e trabalhos futuros.

2. Algoritmos de *diff*

O funcionamento de algoritmos de *diff2 textual* consiste numa execução do algoritmo LCS para a identificação da maior sequência comum, seguida da identificação das adições necessárias para transformar essa sequência comum em cada um dos arquivos passados por argumento. Desta forma, o LCS é considerado o passo mais importante de um algoritmo de *diff*.

Um possível algoritmo (ingênuo) para LCS consiste em, supondo dois arquivos V_1 e V_2, com tamanhos n e m, respectivamente, obter todas as subseqüências do arquivo V_1 e verificar se cada subsequência existe em V_2. Porém esta solução apresenta uma complexidade assintótica exponencial na ordem de $O(2^n \times m)$ onde 2^n representa o

número de subseqüências diferentes em V_1, e O(*m*) representa o tempo requerido para verificação da existência da seqüência em V_2.

Com o propósito de melhorar o desempenho, foi introduzida a programação dinâmica no algoritmo de LCS. De forma sucinta, o algoritmo decompõe seu problema em problemas menores, com soluções conhecidas e pequenas, de modo que a junção destes resultados irá compor a solução final [Cormem , Leiserson e Stein 1998]. A estrutura do algoritmo é baseada na Fórmula 1.

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases} \qquad \textbf{Fórmula 1}$$

Onde *c[i, j]* representa o tamanho da subseqüência comum encontrada, as letras *i* e *j* indicam os números dos elementos avaliados para cada arquivo e x_i , y_j representam os conteúdos das versões avaliadas. Esta fórmula recursiva gera complexidades assintóticas reduzidas em relação à originada pelo algoritmo de força bruta: O(n × m).

Retornando ao exemplo apresentado na introdução, a execução dessa versão do algoritmo baseada em programação dinâmica pode ser visualizada através da Tabela 1, onde cada célula representa o cálculo do LCS para as sequências de tamanho *i* (linha) e *j* (coluna), como descrito na Fórmula 1, onde o número na célula representa o valor de *c[i, j]* e a seta representa o caminho para a identificação dos elementos pertencentes à maior subsequência comum. Desta forma, caso os valores de x_i e y_j sejam iguais, uma seta na direção diagonal é utilizada. Caso contrário, a seta é direcionada para *max (c[i, j-1],c[i-1, j])*. Por fim, caso os valores de *c[i, j-1]* e c*[i-1, j]* sejam iguais, duas setas são utilizadas, na horizontal e na vertical. Após a montagem da tabela, é possível percorrê-la através das setas, obtendo assim a maior sequência comum entre os arquivos.

Tabela 1. Tabela gerada pela aplicação da programação dinâmica ao LCS

V_1 → / V_2 ↓	Ø	A	B	C	D	E	F
Ø	0	0	0	0	0	0	0
E	0	0	0	0	0	↖1	←1
C	0	0	0	↖1	←1	←1↑	←1↑
H	0	0	0	↑1	←1↑	←1↑	←1↑
D	0	0	0	↑1	↖2	←2	←2
I	0	0	0	↑1	↑2	←2↑	←2↑
F	0	0	0	↑1	↑2	←2↑	↖3
A	0	↖1	←1	←1↑	↑2	←2↑	↑3
B	0	↑1	↖2	←2	←2↑	←2↑	↑3
G	0	↑1	↑2	←2↑	←2↑	←2↑	↑3

3. LCS Recursivo para detecção de movimentações

A proposta deste trabalho é apresentar uma abordagem para detecção de movimentações em arquivos de texto, através da aplicação recursiva do algoritmo de LCS. No caso de

comparação entre códigos fonte, esta abordagem considera inclusive comentários e espaços em branco, visando à generalização da solução apresentada. Ou seja, apesar de diffs semânticos serem em geral mais poderosos, os diffs textuais têm um papel importante exatamente por serem indiferentes à sintaxe do artefato. Com isso, é possível a sua aplicação em qualquer artefato textual, o que engloba a grande maioria das linguagens de programação.

Programas conhecidos, como o *GNU diff*, já fazem uso do LCS. Contudo, na sua implementação o LCS é utilizado uma única vez, para detectar a maior subsequência comum e todo o restante é identificado como adicionado ou removido. A proposta deste artigo consiste em, ao final da primeira execução, retirar a seqüência obtida e re-executar o LCS com os dados restantes. Desta maneira é possível obter a próxima subsequência máxima comum. Essa subsequência representa, na verdade, um bloco de linhas que foi movido. Como pode ser observada, a recursão pode ser feita diversas vezes, sempre com o intuito de identificar outras subsequências menores que foram movidas. Por conseguinte, as execuções ocorrerão de acordo com os parâmetros estabelecidos pelo usuário, respeitando limites e domínios definidos para o sistema.O número de execuções do algoritmo é chamado de nível de recursão, e poderá receber três valores, sendo eles: *Primeiro Nível*, *Segundo Nível* e *Nível Indeterminado*. Ao designar a primeira opção, apenas uma execução será realizada. Com isso, serão detectadas apenas as divergências no código (i.e., comportamento semelhante ao *GNU diff*). Ao considerar a segunda opção, duas execuções serão realizadas e será detectada a maior subsequência comum de movimentações através do algoritmo. No caso de nível indeterminado, o número de execuções será definido de acordo com o número de subsequências do arquivo, ou seja, o algoritmo é interrompido quando a LCS obtida é nula.

Outro parâmetro referente à solução proposta diz respeito à granularidade. Um aspecto chave na execução do LCS é determinar a unidade de comparação a ser utilizada [Murta, Dantas, Lopes e Werner 2007]. Caso a unidade de comparação seja linha, o algoritmo tende a ter um melhor desempenho, pois o número total de elementos é reduzido, mas o resultado é em termos de linhas adicionadas, excluídas ou movidas. Por outro lado, caso a unidade de comparação seja palavra, o algoritmo será capaz de analisar individualmente cada palavra do arquivo, e indicar adições, exclusões e movimentações nesse grão. As opções para o parâmetro granularidade são *linha*, *palavra* e *letra.*

Além disso, é possível informar, através do parâmetro denominado Movimentos, o tamanho mínimo da subsequência a ser definida como um movimento, sempre considerando a granularidade escolhida. Desta forma, o algoritmo só irá considerar movimentações que sejam de tamanho maior ou igual ao estipulado neste parâmetro. Esse parâmetro é de especial interesse quando a granularidade é configurada para palavra ou letra.

Após a indicação dos parâmetros necessários, torna-se possível a execução do algoritmo para a detecção das diferenças e movimentações, de forma que as movimentações e alterações sejam identificadas e posteriormente demarcadas com cores distintas na interface gráfica.

O algoritmo implementado como base nesta abordagem faz uso de uma estrutura bi-dimensional referente à programação dinâmica completa. A melhor utilização desta

estrutura e a diminuição da complexidade assintótica do algoritmo são fatores relevantes para melhoria do desempenho do algoritmo.

A Fórmula 2 representa a recursão utilizada para verificação da complexidade assintótica, T [α], do algoritmo em questão. Onde α é o número de execuções realizadas para obtenção do resultado esperado e $O(n \times m)$ representa a complexidade assintótica do algoritmo de LCS.

$$T[\alpha] = \begin{cases} 0 & \text{se } \alpha = 0 \\ O(n \times m) & \text{se } \alpha = 1 \\ O(\alpha \times n \times m) & \text{se } \alpha > 1 \end{cases}$$

Fórmula 2

A Figura 1 exibe o passo-a-passo da execução da solução proposta, com Granularidade de Linha selecionada. Os dados de entrada, considerados pelo algoritmo, são indicados em negrito durante cada execução. Após a detecção da maior subsequência comum nula, o algoritmo é interrompido. Ao final, são exibidos os dados divergentes em negrito e os dados movimentados marcados com transparência na figura.

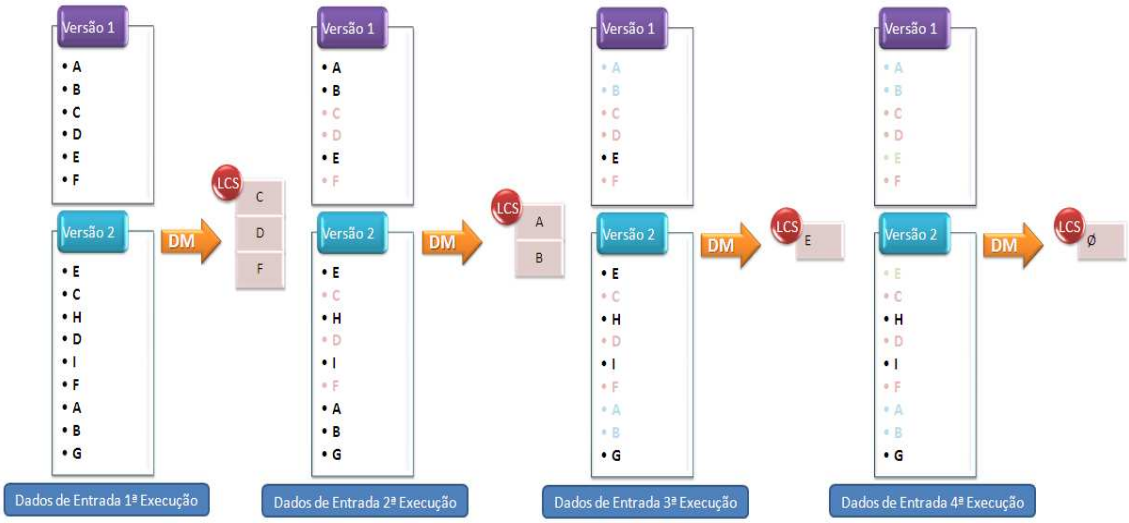

Figura 1. Passo-a-passo da execução do algoritmo proposto

4. Exemplos de Execução do Algoritmo

Uma ferramenta que implementa o algoritmo proposto foi construída. Desta forma, após a definição dos parâmetros, e execução do algoritmo, a ferramenta exibe as divergências entre as versões do artefato em questão. Além disso, as linhas movimentadas são grifadas com cores distintas, permitindo assim a visualização da movimentação realizada, conforme Figura 3.

Para permitir a visualização de como seria a execução convencional, sem identificação de movimentação, a Figura 2 exibe o resultado da execução com o parâmetro Primeiro Nível escolhido.

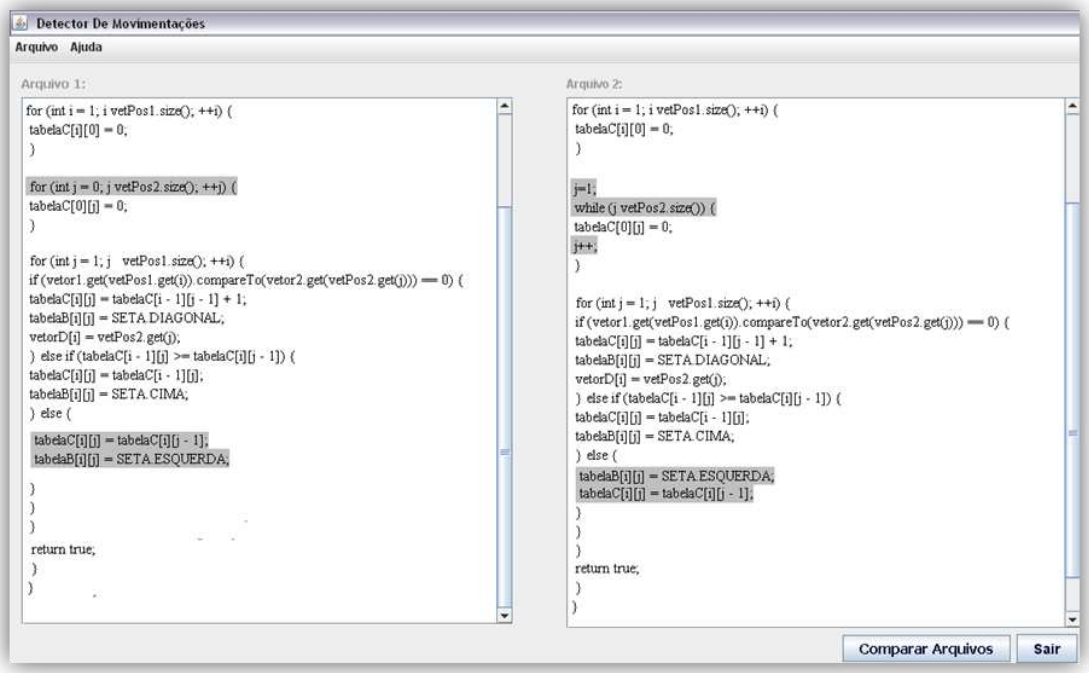

Figura 2. Execução do Algoritmo – Primeiro Nível

Os parâmetros Granularidade de Linha, Movimentos = 1 e Nível Indeterminado foram selecionados para a execução representada na Figura 3. As linhas sublinhadas na cor cinza indicam as diferenças entre os códigos, e as demais cores indicam as movimentações.

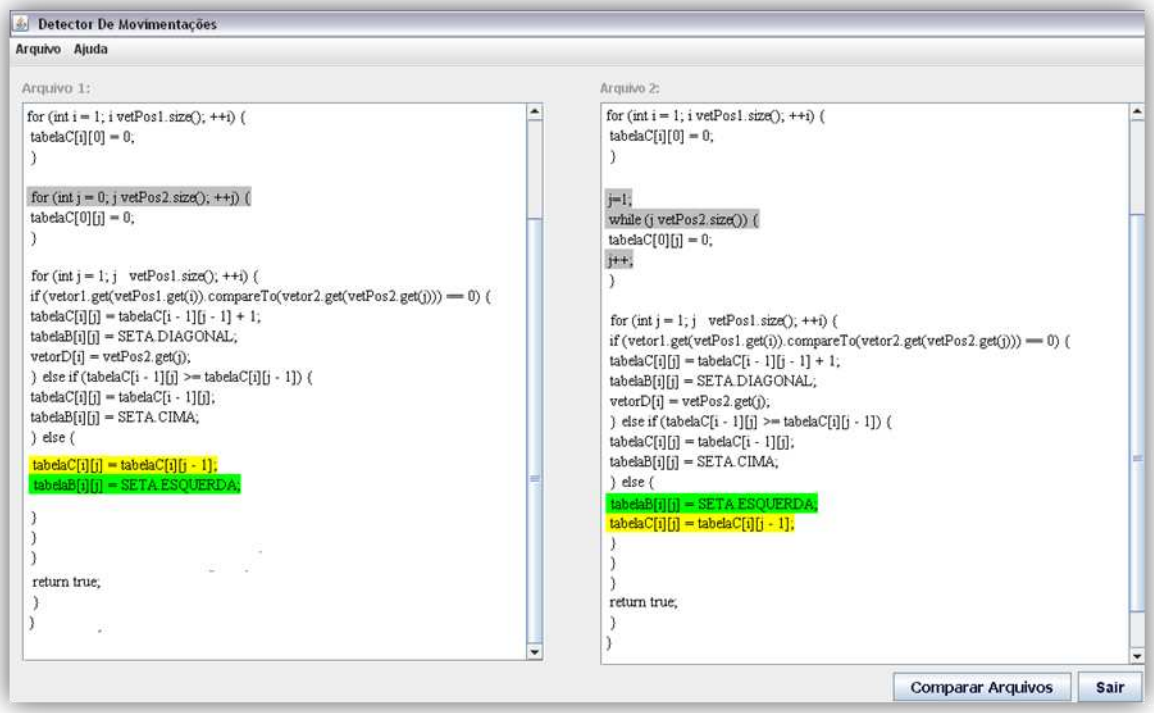

Figura 3. Execução do Algoritmo – Nível Indeterminado

5. Trabalhos Relacionados

A Tabela 2 realiza uma breve comparação de recursos entre algumas técnicas citadas anteriormente, de acordo com características como definição de granularidades, tipos de comparação, tipos de *diff* e realização de merge. Onde **DM** refere-se ao Detector de Movimentações, descrito neste trabalho.

Tabela 2. Comparação de Recursos

Nome	Δ ≠ Linha	Comparação de Diretórios	Movimentação de Linhas	Comparação 3-way	Merge
DIFF	✗	✓		✗	
DIFF 3	✗	✗		✓	
DM	✓	✗	✓	✗	
WinDiff	✓	✓		✗	
WinMerge	✓	✓	✓	✗	✓

É importante notar que apesar do WinMerge também oferecer recursos de detecção de modificação, o seu algoritmo não está explícito e não permite ajustes por parte do usuário. No caso da abordagem proposta neste artigo, o algoritmo é público e com complexidade proporcional ao LCS, que é alvo de diversas pesquisas para melhoria de desempenho.

Muitos estudos estão sendo realizados visando principalmente à redução da complexidade assintótica do algoritmo de LCS, obtendo melhorias consideráveis de desempenho. Como exemplo, [Iliopoulos e Rahman 2007] sugerem uma nova abordagem de forma que o custo para esta solução seria O(Rlog(logn)), onde R é o total de pares ordenados de posições com que as duas strings combinam. Outra abordagem é citada em [Yang, Huang e Chao 2005], onde é proposto um algoritmo na ordem O(m × n) considerando tempo e espaço.

6. Conclusão

Este artigo apresentou uma abordagem para detecção de movimentações e modificações entre diferentes versões de um mesmo artefato de software ou entre diferentes artefatos. Essa abordagem é baseada na execução recursiva do algoritmo LCS e foi implementada utilizando a linguagem Java. A contribuição principal dessa abordagem concentra-se na detecção de movimentações, que permite que o desenvolvedor possa focar no que de fato foi modificado entre duas versões de um mesmo artefato, contribuindo para atividades como inspeção e compreensão de modificações passadas.

Contudo, outra contribuição importante deste trabalho é a parametrização da granularidade da comparação, da quantidade mínima de movimentações e do nível de recursão a ser adotado. Como sequência deste trabalho, uma revisão sistemática está sendo realizada, com o intuito de identificar as técnicas existentes de *diff* e viabilizar um posterior levantamento de características positivas e negativas dessas técnicas.

Dentre as principais limitações podemos citar a ausência de interação com interface gráfica para as granularidades de *palavra* e *letra*. Para esses casos, são gerados arquivos

de saída com o resultado da execução do algoritmo. Além disso, alguns trabalhos futuros são vislumbrados, dentre eles: o armazenamento completo da estrutura bi-dimensional utilizada pela aplicação da técnica de programação dinâmica, de forma a considerar apenas as posições referentes à indicação da maior subsequência comum; a ampliação do escopo de recursos a serem considerados, como por exemplo, desacoplando o algoritmo da GUI; a aplicação de técnicas para melhorias de desempenho no algoritmo de LCS; e o aumento do domínio das granularidades a serem contempladas, permitindo, por exemplo, a comparação entre diretórios.

Referências

[Bennett e Rajlich 2000] Bennett, K. H. e Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. *ACM*, pages 73–87.

[Berzins 1994] Berzins, V. (1994). Software merge: Semantics of combining changes to programs. In *ACM. Trans. Programming Languages and Systems*, volume 16, pages 1875–1903.

[Buffenbarger 1995] Buffenbarger, J. (1995). Syntactic software merging. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5, J.Estublier, ed*, pages 153–172.

[Estublier 2001] Estublier, J. (2001). Objects control for software configuration management. pages 359–373. SpringerVerlag.

[Horwitz 1990]Horwitz, S. (1990). Identifying the semantic and textual differences between two versions of a program. In *Proc. ACM SIPLAN*, pages 234–245.

[Yang, Huang e Chao 2005] hsuan Yang, I., pin Huang, C., and mao Chao, K. (2005). A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93:249–253.

[Iliopoulos e Rahman 2007] Iliopoulos, C. S. and Rahman, M. S.(2007) A new efficient algorithm for computing the longest common subsequence. *Springer Berlin / Heidelberg*, 4508:82–90.

[J. W. Hunt 1976] Hunt, J. W. , M. D. M. (1976). An algorithm for differential file comparison. *Tecnical Report 41, ATT Bell Laboratories Inc.*

[Murta, Dantas, Lopes e Werner 2007] Murta, L., Oliveira H., Dantas C., Lopes L. G. and Werner C. (2007). Odyssy-SCM: An integrated software configuration management infrastructure for UML models. *Science of Computer Programming. Volume 65, Issue 3, Pages 249-274.*

[Mens 2002] Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng*, 28(5):449–462.

[Narao Nakatsu e Yajima 2008] Narao Nakatsu, Y. K. e Yajima, S. (2008). A longest common subsequence algorithm suitable for similar text strings. *In Proc. of 19th MFCS, number 841 in LNCS*, volume 18, pages 171–179. Springer.

[Paterson e Dancik 1994] Paterson, M. e Dancik, V. (1994). Longest common subsequences. *In Proc. of 19th MFCS, number 841 in LNCS*, pages 127–142. Springer.

[Cormem , Leiserson e Stein 1998] Cormem, T.,C. Leiserson, R. R. e Stein, C. (1998). *Algoritmos Teoria e Prática.* Editora Campus.

[Tichy 1988] Tichy, W. F. (1988). Tools for software configuration management. *In Proc. Int'l Workshop Software Version an Configuration Control*, pages 1–20.

[U.Asklund 1994] U.Asklund (1994). Identifying conflicts during structural merge. In *Proc. Nordic Workshop Programming Environment Research*, pages 231–242.