UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

2005.2

Algoritmo de Comparação de Strings para Integração de Esquemas de Dados

---

TRABALHO DE GRADUAÇÃO

Aluno – Flavio Melo Gondim (fmg@cin.ufpe.br)
Orientadora – Ana Carolina Salgado (acs@cin.ufpe.br)

Recife, Fevereiro de 2006.

Resumo

Este trabalho tem como objetivo avaliar diferentes métodos de comparação de strings determinando qual algoritmo é mais adequado para comparar descritores de elementos de bancos de dados (entidades, atributos, relacionamentos) escritos em português. Os algoritmos avaliados produzem um escore que é um número decimal variando entre 0 e 1, e se refere à similaridade entre duas strings. Este trabalho terá como foco a comparação da qualidade dos escores produzidos. Este trabalho de graduação também se propõe a efetuar alterações nos algoritmos existentes para melhorar a qualidade dos escores produzidos. Depois de criado, este novo algoritmo será integrado ao "Sistema de Enriquecimento Léxico de Esquemas para um Sistema de Integração de Dados", que é um subsistema responsável pela associação lingüística de esquemas no Sistema Integra.

Abstract

This work has as objective to evaluate different methods of comparison of strings determining which algorithm could better be adjusted to compare elements that describe the databases (entities, attributes, and relationships) written in Portuguese. The evaluated algorithms produce a score that is a decimal number varying between 0 and 1, which relates the similarity between two strings. This work will have as focus the comparison of the quality of the produced scores. This graduation work also considers to effect alterations in the existing algorithms to improve the quality of the produced scores. This new algorithm will be integrated to the "System of Lexicon Enrichment of Schemas for a Data Integration System", that is a subsystem responsible for the linguistic association of projects in a System called Integra.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Com o aumento da utilização da WEB tornou-se possível o acesso a bases de dados remotas. Mas este fato torna-se não trivial quando se tenta integrar várias bases de dados heterogêneas. O Sistema Integra é uma proposta de como solucionar este problema, pois ele propõe um mecanismo de integração de bases autônomas e heterogêneas, de forma transparente para o usuário.

O Integra propõe um sistema de integração de dados que adota a abordagem da Visão Global, onde cada elemento do esquema global único é representado como uma visão sobre as fontes de dados. O diferencial do Integra é que além de prover o acesso integrado a dados distribuídos e heterogêneos, ele também oferece soluções para os problemas relacionados à geração e à manutenção de consultas submetidas ao esquema global único [LÓSCIO 03].

Na Figura 1, temos a arquitetura do Sistema Integra, que permite a execução de consultas a Bases de Dados Heterogêneas. Esta arquitetura está dividida em quatro ambientes: Ambiente Comum; Ambiente de Integração de Dados; Ambiente de Geração e Manutenção do Mediador; e Ambiente do Usuário.

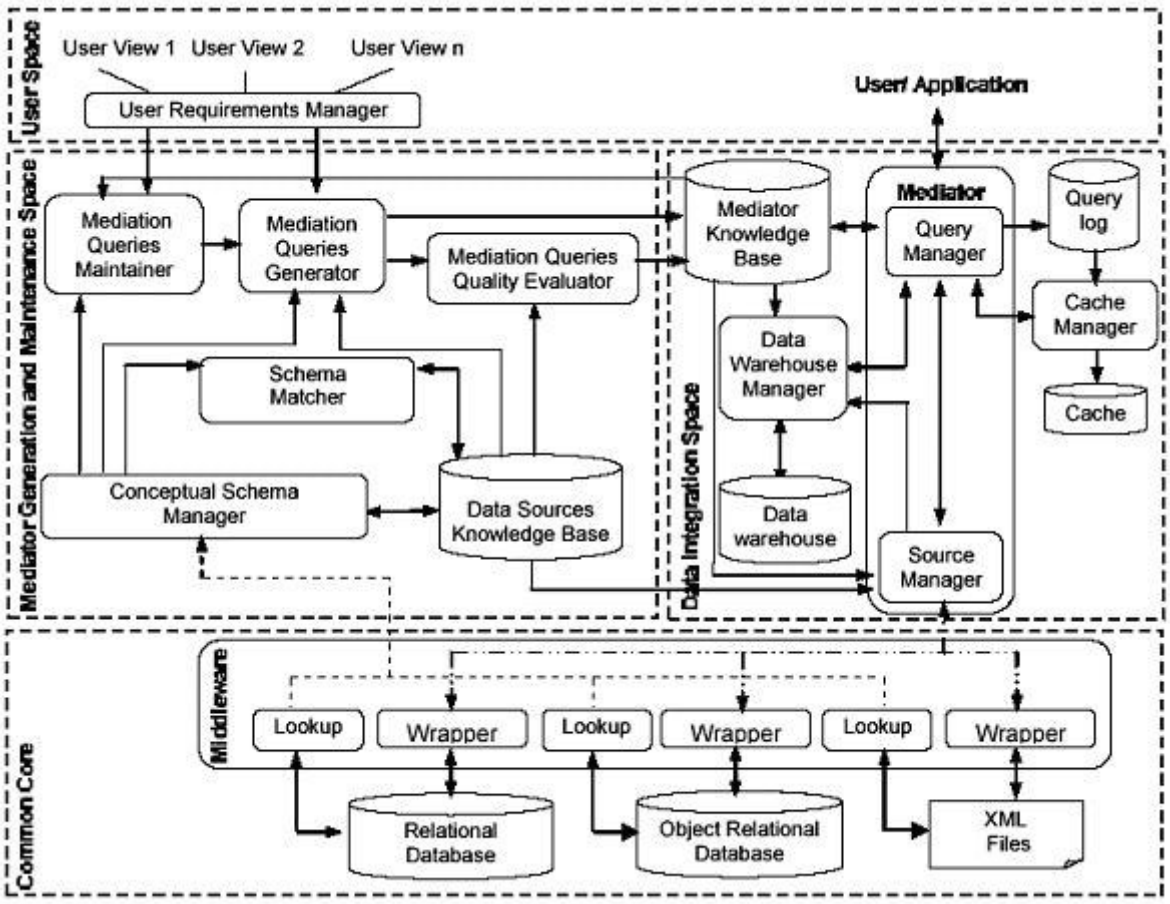

Figura 1 – Arquitetura do Sistema Integra [LÓSCIO 03]

No Ambiente de Geração e Manutenção do Mediador temos o Associador de Esquemas (*Schema Matcher*) que compara descritores de entidades, atributos e relacionamentos (extraídos dos esquemas das fontes de dados) com conceitos da ontologia de domínio. Para está comparação é necessária a utilização de um algoritmo de comparação de Strings (String *Match*). O objetivo deste trabalho será o de determinar qual algoritmo deverá ser utilizado para esta comparação.

Neste trabalho realizamos um estudo sobre diferentes métodos de comparação de strings determinando qual algoritmo é mais adequado para comparar descritores de elementos de bancos de dados (entidades, atributos, relacionamentos) escritos em português. O foco desta comparação é a qualidade dos escores produzidos.

Este trabalho também se propõe a efetuar alterações nos algoritmos existentes para melhorar a qualidade dos escores produzidos.

Este documento está organizado da seguinte forma:

- O capítulo 1 apresentou a introdução e a proposta do trabalho;
- O capítulo 2 detalha o contexto onde este trabalho está inserido;
- O capítulo 3 apresenta e avalia os algoritmos presentes na literatura;
- O capítulo 4 apresenta uma comparação dos escores produzidos pelos algoritmos de alinhamento local;
- O capítulo 5 apresenta o algoritmo de alinhamento inicial que foi criado neste trabalho;
- O capítulo 6 evidencia as conclusões obtidas neste trabalho.

2. Integração de Esquemas de Dados

Este capítulo analisa o modelo de Integração de Esquemas que é proposto pelo Integra, e em seguida descreve o processo de enriquecimento léxico que passará a utilizar o algoritmo de comparação de strings proposto por este trabalho.

Por fim, é discutida a problemática da integração semântica e são apresentadas algumas modificações, que têm como objetivo solucionar estes problemas no Sistema de Integração de Dados.

2.1 Integração de Esquemas no Integra

Belian [BELIAN 05], propôs um processo de geração de um esquema de mediação semanticamente alinhado para o sistema Integra. Isto significa que este processo deverá gerar as entidades, atributos e relacionamentos de mediação, bem como as assertivas de correspondência entre elementos dos esquemas das fontes de dados e de mediação com os conflitos semânticos corretamente resolvidos.

A proposta para geração do esquema de mediação se baseia em um processo de comparação lingüística dos identificadores dos elementos dos esquemas das fontes de dados com conceitos da ontologia de domínio [Belian 05].

Na figura 2, temos a arquitetura do Sistema Integra, estendida no ambiente de geração e manutenção do mediador. Esta extensão ocorreu, pois se fez necessária a criação de módulos responsáveis pelo tratamento semântico utilizado nas 3 fases de geração do esquema de mediação: Associador lingüístico de esquemas; Unificador semânticos de conceitos; e Gerador do esquema de mediação.

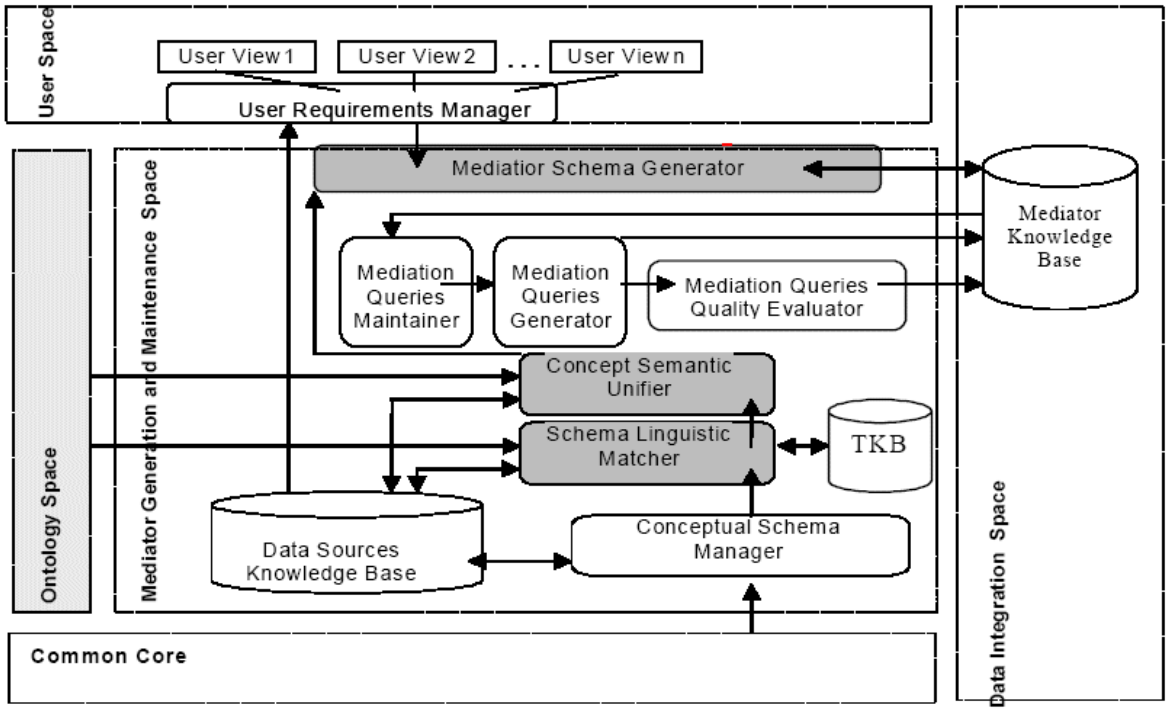

Figura 2 – Arquitetura estendida para geração do mediador com resolução semântica [BELIAN 05]

O Associador Lingüístico de Esquemas (*Schema linguistic matcher*) é o módulo responsável por um processo de casamento lingüístico, que compara identificadores de entidades, atributos e relacionamentos (extraídos dos esquemas das fontes de dados) com conceitos da ontologia de domínio. Ao final do processo os esquemas das fontes de dados estarão enriquecidos com termos ontológicos que correspondem sintaticamente aos identificadores dos elementos das fontes de dados [BELIAN 05].

O Associador Lingüístico de Esquemas realiza um processo de casamento lingüístico. Para isto é necessária a utilização de um algoritmo de comparação de Strings (String *Match*) e o objetivo deste trabalho é identificar qual algoritmo deverá ser utilizado nesta fase.

O Associador Lingüístico de Esquemas atualiza (enriquece) os esquemas das fontes de dados com informações léxicas extraídas da ontologia de domínio. Estes esquemas serão utilizados na etapa seguinte que é descrita no próximo parágrafo.

O Unificador Semântico de Conceitos é responsável pela unificação de conceitos semanticamente similares encontrados nos esquemas das fontes de dados e pela organização destes conceitos no que chamamos de "Coleção de grupos semânticos similares" [BELIAN 05].

O Gerador do esquema de mediação gera o esquema do mediador. Ele utiliza a coleção de grupos semânticos para gerar as entidades, atributos e relacionamentos que deverão fazer parte do esquema de mediação, e conseqüentemente, finaliza a geração das assertivas de correspondência entre o esquema de mediação e as fontes de dados.

2.2 Descrição do Processo de Enriquecimento Léxico

Na integração de Esquemas de dados é recebido um esquema conceitual, representado através de um arquivo XML que descreve as tabelas presentes no esquema, e para cada tabela são descritos seus atributos. Este XML será enriquecido lexicamente com novas tags que farão uma relação entre o termo presente no esquema com o termo mais similar encontrado na ontologia.

Para fazer a busca pelo termo mais próximo a um termo presente em um esquema conceitual, é necessário comparar cada termo do esquema com os termos da ontologia. Depois disto, é atribuído um escore, representando o grau de similaridade entre este dois termos, este escore varia de 0 a 1.

Como já foi dito anteriormente, este processo de enriquecimento é realizado de forma semi-automática, pois pode ser necessária uma interação com o usuário do sistema, no caso de não se encontrar na ontologia uma string similar a string presente no esquema. A nova string inserida será armazenada em uma Base de Conhecimento, e quando houver uma nova ocorrência desta string, o *match* passará a ser automático. Em alguns casos também será preciso que o usuário selecione manualmente (em uma tabela) a string da ontologia que ele considera mais similar a uma string do esquema.

Existem algumas variáveis utilizadas para auxiliar este processo:

- similaridadeMatchAutomatico – para que haja um *match* automaticamente, sem consulta ao usuário, é necessário que o escore obtido ultrapasse o valor definido por esta variável;
- similaridadeConsideravel – para que duas strings possam ser consideradas similares, podendo ser apresentadas para o usuário, é necessário que o escore obtido ultrapasse o valor definido por esta variável;
- precisao – para que um *match* ocorra automaticamente, não pode existir uma segunda string cuja similaridade seja muito próxima a da primeira string (a de maior similaridade), este valor então é uma espécie de distância mínima que deve existir entre os valores das duas maiores similaridades.
- quantidadeMaxima – quantidade máxima de strings similares que poderão ser apresentadas para o usuário, estas strings estão ordenadas pelo grau de similaridade, portanto as mais similares é que serão apresentadas.

No processo de seleção de um termo similar, a primeira verificação feita é se houve algum termo similar à string do Esquema Global. O pseudo-código referente a este processo é apresentado na Figura 3.

```
IF não houver nenhum termo semelhante
THEN é solicitado que o usuário insira um novo termo, que será
     inserido tanto no XML enriquecido quanto na Base de
     Conhecimento
ELSE
     IF quantidade de termos semelhantes for igual a um
     THEN
          IF similaridade for maior que
               similaridadeMatchAutomatico
          THEN é realizado um match automático
          ELSE é solicitado que o usuário selecione este termo ou
               que insira um novo termo
     ELSE
          IF   similaridade do termo mais similar for maior que
               similaridadeMatchAutomatico
               AND distancia entre similaridades do primeiro e
               segundo termo mais similares for maior do que a
               precisao
          THEN é realizado um match automático
          ELSE os termos mais similares são exibidos na tela e é
               solicitado que o usuário escolha um destes termos ou
               que crie um novo termo
```

Figura 3 – Pseudo-código do Processo de Enriquecimento Léxico

Na Figura 4, podemos observar o formato padrão de um esquema enriquecido onde para cada termo do XML é inserido um conjunto de tags denominadas por SYNTACTIC_MATCHING. Esta tag contém o termo que foi considerado mais similar ao termo do XML em questão, além de conter

a descrição do termo que foi selecionado, a fonte de onde este termo foi extraído, e a similaridade entre o termo encontrado e o termo que originou esta busca.

```
<ENTITY name="PACIENTE">
<SYNTACTIC_MATCHING>
   <ASSIGNED_TERM>pacientes</ASSIGNED_TERM>
   <DESCRIPTION>individuals participating in the health care system
   for the purpose of receiving therapeutic, diagn</DESCRIPTION>
   <SOURCE>UMLS</SOURCE>
   <STRING_PROXIMITY>0,952070</STRING_PROXIMITY>
</SYNTACTIC_MATCHING>
<ATTRIBUTE name="NOME" type="xs:string" nullable="false" />
<SYNTACTIC_MATCHING>
   <ASSIGNED_TERM>nomes</ASSIGNED_TERM>
   <DESCRIPTION>personal names, given or surname, as cultural
   characteristics, as ethnological or religious
   pattern</DESCRIPTION>
   <SOURCE>UMLS</SOURCE>
   <STRING_PROXIMITY>0,911111</STRING_PROXIMITY>
</SYNTACTIC_MATCHING>
```

Figura 4 – Esquema Enriquecido

2.3 Problemática da Integração Semântica

Uma palavra pode ser interpretada de duas formas diferentes, pelo significante e pelo significado.

> "**Significante** é a parte física da palavra, constituída na fala pelos fonemas e, na escrita, pelas letras.
> **Significado** é o sentido, a informação que a palavra carrega e faz nascer na mente do ouvinte ou leitor uma imagem, uma idéia." [MESQUITA and MARTOS 91].

Na integração de dados podemos considerar que a Integração **Sintática** é uma forma de integração que leva em consideração apenas o significante da palavra. Já a Integração **Semântica** leva em consideração o significado da palavra, podendo identificar similaridades semânticas entre duas palavras de grafia diferente.

Belian [BELIAN 05], descreve vários problemas que ocorrem quando se tenta representar dados semanticamente similares através de esquemas divergentes. O estudo destes conflitos é bastante importante para a integração de esquemas de dados de fontes heterogêneas.

Conflitos entre nomes de entidades. As entidades semanticamente relacionadas podem ser representadas com nomes diferentes. As entidades "estudante" e "aluno" são exemplos deste tipo de conflito. Estas entidades se relacionam através da sinonímia e são consideradas semanticamente equivalentes.

Conflitos entre esquemas de entidades representadas de forma heterogênea. Algumas entidades semanticamente similares podem ser representadas nos bancos de dados através de esquemas distintos:

A entidade "agente-saude" pode ser um médico se tipo_agente = medico. Para a entidade "medico" pode ser assumido que ela apresenta um atributo tipo = medico como valor default.

- *medico1 (num_conselho, nome, especialidade)*
- *agente-saude2 (num_conselho, nome, tipo_agente)*

Conflitos entre atributos semanticamente semelhantes com conflitos de nomes. Este tipo de conflito acontece quando dois atributos são semanticamente semelhantes e apresentam nomes diferentes (sinônimos). Por outro lado, atributos com mesmo nome podem não ter qualquer relação semântica (homônimos).

Atributos sinônimos (num_conselho e CRM):

- *medico1 (num_conselho, nome, especialidade)*
- *doutor2 (CRM, nome)*

Atributos antônimos (num_conselho em medico1 e enfermeiro2):

- *medico1 (num_conselho, nome, especialidade)*
- *enfermeiro2 (num_conselho, nome)*

Conflitos de generalização. Este tipo de conflito acontece quando duas entidades são representadas em dois bancos de dados em níveis diferentes de abstração (generalização).

Como exemplo, podem ser apresentadas as entidades, onde a entidade "agente-saude" representa os dados em um nível de abstração mais geral.

- *medico1 (num_conselho, nome, especialidade)*
- *agente-saude2 (num_conselho, nome, tipo_agente)*

Conflitos de agregação. Este tipo de conflito acontece quando algumas entidades de um banco de dados correspondem a uma representação em forma de agregação em outro banco de dados. No exemplo abaixo, a entidade "interrog_sintomatologico" corresponde a um conjunto de "sintomas" entidade do banco de dados S1.

- *sintoma1 (codigo, nome, sistema)*
- *interrog_sintomatologico2 (codigo, nome, data sistema)*

O Sistema de Integração de Dados, que foi descrito na seção 2.2, leva em consideração apenas a análise sintática, verificando apenas se a escrita de duas palavras é semelhante. Isto torna o sistema incompleto, pois, podem ocorrer diversos conflitos semânticos.

Para impedir que tais conflitos aconteçam, deve-se utilizar uma Ontologia que trate da Similaridade Semântica. A similaridade semântica quantifica o grau de similaridade semântica entre dois objetos do mundo real. A escala de similaridade possui 5 níveis, variando do maior nível que

é a Equivalência Semântica, até o menor que é a Incompatibilidade Semântica, estes dois níveis serão apresentados a seguir.

Equivalência semântica. Consiste na medida que indica maior similaridade semântica entre dois objetos. Dois objetos são semanticamente equivalentes quando representam a mesma entidade ou conceito do mundo real. Os objetos estudante e aluno, por exemplo, são considerados semanticamente equivalentes [BELIAN 05].

Incompatibilidade semântica. Esta medida indica completa dissociação semântica. Neste caso, dois objetos que são semanticamente incompatíveis não se relacionam em nenhum contexto.

A escala anterior indica uma classificação que permite a sistemas de integração avaliar se determinadas entidades das fontes de dados são semanticamente similares e como estas devem ser integradas pelo sistema considerando o grau de similaridade apresentado [BELIAN 05].

O algoritmo de comparação de strings que é discutido neste trabalho leva em consideração apenas a similaridade sintática. Para que o processo de Enriquecimento Léxico visto na seção anterior fique completo, é preciso acrescentar o tratamento semântico para que também seja levada em consideração a similaridade semântica entre duas strings.

2.4 Requisitos da Aplicação

Este trabalho visa encontrar um algoritmo que seja adequado à comparação de duas strings que tenham sido pré-processadas, recebendo um tratamento chamado de normalização.

Este tratamento é necessário, pois os elementos dos esquemas correspondem a entidades, atributos e relacionamentos dos esquemas das fontes de dados locais que geralmente são definidos de forma "livre" nos bancos de dados. Os identificadores destes elementos usualmente são formados com hífens, abreviações e acrônimos, o que impede a correta identificação destes nas ontologias e vocabulários do domínio.

As strings que serão comparadas passam inicialmente por um processo de normalização, que trata hífens, gênero, número, palavras compostas e procura eliminar prefixos e sufixos que não contribuem para a identificação do elemento [BELIAN 05].

Esta comparação de strings servirá para determinar um grau de similaridade entre duas strings, este grau irá variar entre [0 - 1]. Isto será útil para determinar o quão próxima uma string pertencente a um esquema local de uma fonte de dados está de uma string pertencente a uma Ontologia de Domínio.

3. Algoritmos Existentes

Neste capitulo serão apresentados os algoritmos de comparação de strings mais comuns na literatura. Os algoritmos comparados recebem como parâmetros duas Strings e retornam como resultado um número decimal variando entre 0 e 1.

O valor 0 representa que não há nenhuma relação entre as duas strings e 1 representa que as strings são exatamente idênticas. Para os demais resultados quanto mais próximo de 1 mais as strings são próximas uma das outras, e quanto mais próximo de 0 mas as strings se distanciam.

Para a notação de eficiência será considerado que as strings recebidas como parâmetro serão x e y, os tamanhos (número de caracteres) destas duas strings serão respectivamente |x| e |y|.

Antes da descrição destes algoritmos, são mostradas na Figura 5 as diversas maneiras de se alinhar duas strings. O Tipo de Alinhamento pode ser Global (alinha as duas string como um todo), Semi-Global (ignora espaços nos extremos das seqüências) e Local (ignora espaços e *mismatchs* nos extremos das seqüências).

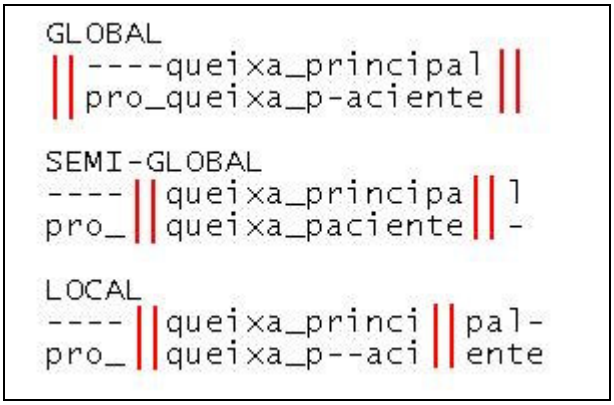

Figura 5 – Tipos de Alinhamento

3.1 Descrição dos Algoritmos

Os algoritmos que serão mostrados aqui são em sua maioria uma derivação da forma de implementar o primeiro algoritmo apresentado, o de Levenshtein [NAVARRO 01]. Estas derivações têm o objetivo de melhorar a eficiência ou a qualidade dos escores obtidos.

Serão apresentados ao todo 10 algoritmos de comparação de string, e em alguns casos será dado um exemplo de como estes algoritmos funcionam.

3.1.1 Levenshtein Metric

O algoritmo de Levenshtein ou *edit distance* (distância de edição) leva o nome de seu autor. Este foi um dos primeiros algoritmos de comparação de String e até hoje ainda é um dos mais utilizados.

Este algoritmo permite inserções, remoções e substituições. E pode ser parafraseado como "o menor número de inserções, remoções e substituições para igualar duas strings" [NAVARRO 01].

São atribuídos escores diferentes para cada operação possível: *match* (casamento, igualdade dos caracteres); *mismatchs* (substituições); inserções, remoções. Todas as possibilidades são avaliadas para se chegar ao maior escore.

Esta é a definição da função de Levenshtein:

M(i,j) = Max **{** M(i - 1, j) - 1, *//inserção*
M(i - 1, j -1) + p (i , j), *//match ou substituição*
M(i , j - 1) - 1 **}** *//remoção*

Onde p (i , j) = +2 se Xi = Yj *//match*
-1 se Xi ≠ Yj *//substituição*

M é a matriz de Levenshtein, e M(0,0) = 0. A função p(i,j) é utilizada para determinar se houve igualdade entre os termos comparados (*match*) ou não (substituição). X e Y são as strings que estão sendo comparadas, "i" e "j" são respectivamente as posições dos caracteres destas duas strings. Na Tabela 1 temos um exemplo, comparamos "medico" com "biomedicina". Podemos observar que o escore obtido (último escore da matriz) neste caso foi 4.

Tabela 1 – Exemplo de Matriz de Levenshtein

	ε	**b**	**i**	**o**	**m**	**e**	**d**	**i**	**c**	**i**	**n**	**a**
ε	**0**	**-1**	**-2**	**-3**	-4	-5	-6	-7	-8	-9	-10	-11
m	-1	-1	-2	-3	**-1**	-2	-3	-4	-5	-6	-7	-8
e	-2	-2	-2	-3	-2	**1**	0	-1	-2	-3	-4	-5
d	-3	-3	-3	-3	-3	0	**3**	2	1	0	-1	-2
i	-4	-4	-1	-2	-3	-1	2	**5**	4	3	2	1
c	-5	-5	-2	-2	-3	-2	1	4	**7**	6	5	4
o	-6	-6	-3	0	-1	-2	0	3	6	**6**	**5**	**4**

Este é o alinhamento obtido:

- - - m e d i c - - o
b i o m e d i c i n a

Quanto ao desempenho, o tempo gasto por este algoritmo é da ordem de *O*(|x| |y|), enquanto o espaço requerido é de apenas *O*(mínimo(|x| |y|)), onde |x| e |y| são os tamanhos das strings x e y respectivamente.

Apesar deste algoritmo não ser muito eficiente, ele é um dos mais flexíveis, pois é adaptável a diferentes funções de distância, já que os pesos da função vista acima podem ser modificados em função da aplicação.

Um grande problema deste algoritmo para a nossa aplicação é que ele alinha strings globalmente, podendo atribuir baixos escores para duas strings com sufixos semelhantes como é o caso, por exemplo, de "telephone" e "phone".

3.1.2 Smith Waterman

Este algoritmo é bastante semelhante ao Levenshtein, visto anteriormente, com a exceção de pequenas alterações. Ele alinha duas substrings e não duas strings como o de Levenshtein.

Neste algoritmo qualquer escore negativo é substituído por zero e o escore do alinhamento é o melhor escore dentre todos. Isto possibilita que nem o começo nem o fim das duas strings precisem estar alinhados [SMITE and WATERMAN 80].

Esta é a definição da função de Smith-Waterman:

M(i,j) = Max **{** M(i - 1, j) - 1, *//inserção*
M(i - 1, j -1) + p (i , j), *//match ou substituição*
M(i , j - 1) – 1, *//remoção*
0 **}** *//alinhamento vazio*

M é a matriz de Smith-Waterman, e M(0,0) = 0. A função p(i,j) é calculada da mesma forma que a vista anteriormente. Podemos observar que nesta função o alinhamento nunca será negativo, já que na função acima sempre se pode obter zero como resultado.

O alinhamento a ser escolhido será o de maior escore em qualquer posição da matriz. Na Tabela 2 temos um exemplo, comparamos "medico" com "biomedicina". Podemos observar que o escore obtido (maior escore da matriz) foi 10.

Tabela 2 – Exemplo de Matriz de Smith-Waterman

	ε	**b**	**i**	**o**	**m**	**e**	**d**	**i**	**c**	**i**	**n**	**a**
ε	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0	0
m	0	0	0	0	**2**	1	0	0	0	0	0	0
e	0	0	0	0	1	**4**	3	2	1	0	0	0
d	0	0	0	0	0	3	**6**	5	4	3	2	1
i	0	0	2	1	0	2	5	**8**	7	6	5	4
c	0	0	1	1	0	1	4	7	**10**	9	8	7
o	0	0	0	3	2	1	3	6	9	9	8	7

Este é o alinhamento obtido:

\- - - m e d i c o - -
b i o m e d i c i *n a*

O seu desempenho é semelhante ao do Levenshtein, e seu alinhamento é local o que possibilita duas strings com sufixos semelhantes como é o caso, por exemplo, de "telephone" e "phone" possuam um bom escore de alinhamento.

3.1.3 Stochastic Model

Este algoritmo também é semelhante ao de Levenshtein, pois igualmente se baseia em inserções, remoções e substituições para atribuir escores a uma matriz. Escores estes que são atribuídos através de programação dinâmica [RISTAD and YANILOS 96].

Este algoritmo é muito eficiente para aprender os custos de edição de um corpo de exemplos, ou seja ele adapta a maneira como ele atribui os escores dependendo do corpo de exemplos que este algoritmo recebe. E tudo isto é feito de maneira automática.

O seu desempenho é semelhante ao do algoritmo de Levenshtein.

A grande vantagem deste algoritmo, segundo os seus autores, é que ele apresenta apenas ¼ (um quarto) da taxa de erro do algoritmo de Levenshtein. Este algoritmo pode ser aplicado a qualquer problema de classificação de strings que possa ser resolvido usando funções de similaridade, embora tenha sido testado apenas para o aprendizado de palavras em conversações.

Ele alinha strings globalmente, podendo atribuir baixos escores para duas strings com sufixos semelhantes como é o caso, por exemplo, de "telephone" e "phone". Mas é possível adaptar este algoritmo para que ele passe a alinhar strings localmente. A grande vantagem de utilizar este algoritmo na nossa aplicação seria a taxa de erro bem menor do que a encontrada nos algoritmos derivados do Levenshtein.

3.1.4 Jaro Metric

Este algoritmo se parece um pouco com o de Levenshtein, pois ele também utiliza uma matriz, embora esta só seja utilizada no modelo conceitual e não precise ser implementada na prática, já que não é necessário gravar todos os dados da matriz.

Ele se baseia no número de caracteres comuns entre duas strings, e na semelhança da ordem na qual estas duas cadeias de caracteres se apresentam.

Seja ***s = a1 … ak*** e ***t = b1 … bL***, um caractere ***ai*** será considerado "em comum" com ***t*** se existir um caractere ***bj = ai***, e o valor de ***i*** não pode diferir do valor de ***j*** mais do que H, onde ***H = mínimo(\|s\| \|t\|)/2***, onde |s| e |t| são os tamanhos das strings s e t respectivamente.

Seja ***s' = a'1 … a'k*** os caracteres em s que são comuns com os de t (na mesma ordem que aparecem em s), e seja ***t' = b'1 … b'L***, análogo a ***s'***.

Uma transposição ocorrerá quando um caractere em ***s'*** for diferente de um caractere que ocupa a sua mesma posição em ***t'***, ou seja, quando ***a'i ≠ b'i***. ***Ts',t'*** é uma variável que representa a metade (dividisão por dois) do número de transposições entre ***s'*** e ***t'*** [BILENKO et al 03].

Portanto a métrica de Jaro será:

$$Jaro(s,t) = \frac{1}{3}\left(\frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s'| - Ts',t'}{|s'|}\right)$$

Na Tabela 3 temos um exemplo, comparamos "data_inicio_caso" com "dataincio".

Tabela 3 – Exemplo de Matriz de Jaro

	d	**a**	**t**	**a**	**_**	**i**	**n**	**i**	**c**	**i**	**o**	**_**	**c**	**a**	**s**	**o**
d	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
o	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0

Este é o alinhamento obtido:

d a t a i n *i* *c* o
d a t a i n *c* *i* o

Obtemos: |s| = 16; |t| =9; |s'| = 9; |t'| = 9;
#transposições = 2; Ts't' = 2/2 = 1.

Aplicando a fórmula vista acima:
Jaro = (1/3)*[(9/16)+(9/9)+((9-1)/9)].
Jaro = (1/3)*(353/144) = (353/432); Jaro = 0,817.

O algoritmo de Jaro possui desempenho semelhante ao do Levenshtein, e seu alinhamento é global, mas neste caso o escore dado aos alinhamentos locais não é muito ruim. Isto possibilita que ele dê bons escores para os dois tipos de alinhamento. Este algoritmo foi desenvolvido para o match de strings pequenas, que é exatamente o nosso caso.

3.1.5 Hamming Distance

Este algoritmo, na sua versão simplificada, permite apenas substituições com custo 1. Na literatura este problema de busca é chamado de "*string matching with k mismatches*" [NAVARRO 01].

Outra definição [CHAPMAN 05] é: número de bits que diferem entre duas strings binárias, ou número de bits que precisam ser modificados para transformar uma string idêntica à outra.

Por exemplo: 1 0 0 *1* 1 *0* *1* *0*
1 0 0 *0* 1 *1* *0* *1*

As duas cadeias de bits acima têm uma *hamming distance* de 4 bits, pois 4 bits são diferentes.

Este algoritmo é bastante eficiente, mas não é aplicável ao nosso problema, já que este permite apenas substituições não permitindo nem inserções, nem remoções. Isto inviabilizaria comparações, por exemplo, de "datanascimento" com "data-nascimento".

3.1.6 Soundex Distance Metric

Este método reduz cada String a um "*Soundex Code*" (Código Soundex) que é formado por uma letra e três dígitos. E declara similar todas as strings que possuem o mesmo código [HALL and DOWLING 80].

O objetivo deste método é transformar um nome em um código de 4 dígitos de forma tal que sons similares possuam estes 4 caracteres. O primeiro caractere é a primeira letra do nome, as letras seguintes são substituídas pelos seguintes números:

0) A, E, I, O, U, H, W, Y
1) B, P, F, V
2) C, S, K, G, J, Q, X, Z
3) D,T
4) L
5) M, N
6) R

Em seguida os zeros são removidos; as repetições do mesmo dígito são reduzidas a um único dígito; e, por fim, o código é truncado a apenas uma letra e 3 dígitos.

Por exemplo, "Dickson" e "Dixon" recebem o mesmo código D25, o que mostra a sua similaridade.

Este algoritmo tem um bom desempenho, pois é executado com tempo linear, mais ele só deve ser aplicado para verificar erros fonéticos, pois ele pode ter grandes erros quando usado para outra aplicação. Outro problema é que este foi desenvolvido para a língua inglesa, para ser utilizado na Língua Portuguesa seria necessária uma adaptação.

3.1.7 Covington's distance function

Este método também é utilizado para realizar comparações que levam em conta se o termo comparado é vogal ou consoante. É uma espécie de comparação fonética bruta. Ele atribui diferentes pesos para as substituições de pares de segmentos, e possui custos de *indel* (inserção ou remoção) independentes do contexto [KONDRAK 03].

Foi construída uma função de distância baseada na tentativa e erro. Esta função é bastante simples e identifica apenas 3 tipos de segmentos: consoante, vogais e espaços.

Como as vogais são mais voláteis que as consoantes, se dá preferência à comparação entre consoantes. *Indels* recebem uma penalidade de 50 para abrir o *gap* e passam a receber 40 depois de aberto. Na tabela 4 temos os diferentes termos (*matchs* & *mismatchs*) e suas respectivas penalidades:

Tabela 4 – Penalidades da função de distância de Covington

Termo	Penalidade
Consoantes idênticas ou espaços	0
Vogais idênticas	5
Vogais Diferentes	30
Consoantes diferentes	60
Termos diferentes	100

Através dos valores (penalidades) descritos acima pode-se comparar duas strings de diversas formas para se encontrar o menor valor, que será a distância de Covington entre estas duas strings. Este valor pode então ser utilizado para determinar a similaridade entre estas strings.

Exemplo:

p	a	c	i	e	n	t	e
p	a	t	i	e	n	t	-
0	5	60	5	5	0	0	50

No exemplo acima comparamos a string "paciente" com "patient" e obtemos a distancia de Covington (soma das penalidades) de 125. Como o tamanho da maior string é de 8 caracteres, o pior escore poderia ser 800. Portanto, a semelança será (800-125)/800 = 0,844.

O seu desempenho é muito ruim, pois ele utiliza a técnica de construir uma árvore usando o *depth-first search*, o tempo gasto é muito grande, pois o número de possíveis alinhamentos testados é de *3 elevado a | x |*, e o espaço requerido é gigantesco pois são gerados diversos nós na árvore, x é o tamanho da string.

Este algoritmo é muito bom para a nossa aplicação, pois minimiza a degradação do escore quando erros fonéticos são gerados, mas ele precisaria ser implementado em forma de programação dinâmica para melhorar a sua eficiência.

3.1.8 Autômato Não Determinístico (NFA)

Este algoritmo se parece com o do Levenshtein, só que ao invés de se construir uma matriz é construído um autômato não deterministico que é uma espécie de um grafo com transições para as strings subseqüentes, como pode ser visto na Figura 6.

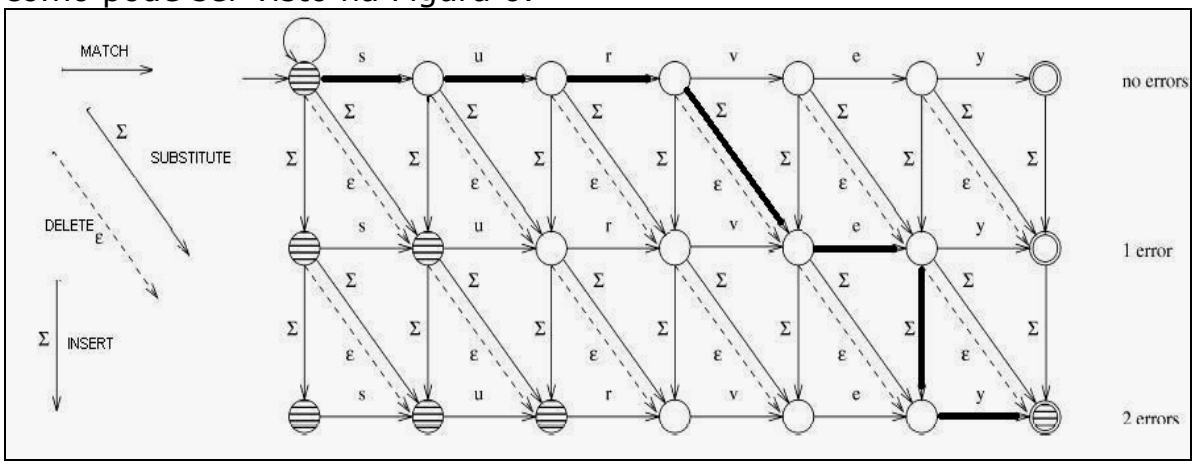

Figura 6 – NFA, comparando "survey" com "surgery" [NAVARRO 01]

Uma aresta horizontal representa o casamento de caracteres, arestas verticais inserem um caractere, arestas diagonais sólidas substituem um caractere e arestas diagonais pontilhadas removem um caractere. Pelo número de linhas existentes é possível descobrir o número de erros [NAVARRO 01].

Quanto ao desempenho, o tempo do pior caso deste algoritmo é da ordem de $O(|y|)$, enquanto o espaço requerido varia exponencialmente em função de |x| e k, onde |x| e |y| são os tamanhos das strings x e y respectivamente, e k é o número de erros.

3.1.9 BLAST

Quando comparamos seqüências reais é possível ter vários bons alinhamentos ou nenhum. O que se deseja é ver o resultado de todos os alinhamentos estatisticamente significativos, isto é o que o BLAST faz. Entretanto, diferentemente do algoritmo Smith-Waterman, o BLAST não explora todo o espaço de pesquisa entre duas seqüências [CRUZ 03].

Como exatamente o BLAST faz para encontrar similaridade sem explorar todo o espaço de pesquisa? Ele usa três regras para refinar seqüencialmente os potenciais *high scoring pairs* (HSP). Estas regras heurísticas, conhecidas como semea/dura, extensão e avaliação, a partir de um refinamento gradual que permite ao BLAST amostrar todo o espaço de pesquisa sem perder tempo em regiões sem similaridade.

A minimização do espaço de pesquisa é a chave para sua velocidade, mas às custas de perda na sensibilidade, pois na tentativa de melhorar o desempenho do algoritmo pode ser que *matchs* passem despercebidos. O acerto entre velocidade *versus* sensibilidade é a chave na elaboração das pesquisas com o BLAST.

Como podemos observar o BLAST tem um desempenho superior ao do Smith-Waterman, mas isto causa uma pequena perda de sensibilidade que pode gerar erros, como na nossa aplicação o desempenho não é algo tão importante é preferível a utilização do Smith-Waterman ou de outro algoritmo que não tenha perda de sensibilidade.

3.1.10 Q-gram

Um q-gram é o conjunto de todas as substrings que podem ser geradas a partir de uma determinada string e "q" representa o tamanho destas substrings.

Exemplo (q-gram gerado para a string "paciente" com q = 3):
{##p, #pa, pac, aci, cie, ien, ent, nte, te$, e$$}

Este algoritmo foi utilizado inicialmente como técnica de filtragem, cujo objetivo é descartar áreas onde não pode haver *matching* (casamento de palavras) [FOSTER 03].

Entretanto esta técnica pode ser aplicada de forma diferente para identificar seqüências de texto que possuam palavras em comum. Se tivermos as strings A e B podemos gerar os q-grams de A e B e depois contamos o número de q-grams idênticos. Então é possível encontrar a "*q-ram distance*":

| q-ram distance | =
| tamanho do maior q-grams | - | número de q-grams em comum |

Exemplo, comparando "paciente" com "patient":

q-gram paciente	{##p, #pa, pac, aci, cie, ien, ent, nte, te$, e$$}
q-gram patient	{##p, #pa, pat, ati, tie, ent, nt$, t$$}
q-gams em comum	**{##p, #pa, ent}**

Neste exemplo temos apenas 3 q-grams em comum, o que é um valor muito baixo, pois as strings comparadas são semelhantes, e a maior string possui 10 qgrams.

Aplicando a fórmula vista acima obtemos:

| q-ram distance | = 10 – 3 = 7.

Isto representa uma similaridade de 3/10 = 0,3 que neste caso é muito baixa, já que as strings comparadas são muito semelhantes.

Sejam |x| e |y| os tamanhos das strings x e y, que serão comparadas. O tempo gasto por este algoritmo é melhor do que o da programação dinâmica, pois como cada q-gram de x é comparado apenas com uma certa quantidade dos q-gram de y então o tempo gasto é da ordem de O(mínimo(|x| |y|)) e a quantidade de espaço requerida é da ordem de O(|x| + |y|).

Este algoritmo é mais eficiente do que o de programação dinâmica, mas a qualidade do escore gerado não é tão boa, portanto não deve ser utilizado na nossa aplicação.

3.2 Bit-Parallelism

O Bit-Parallelism não é um algoritmo, é sim uma técnica visa explorar o paralelismo dos computadores quando estes trabalham com bits. A idéia básica é implementar outro algoritmo de forma paralela utilizando bits.

Existem dois algoritmos que são mais comumente implementados desta forma: autômatos não determinísticos; e programação dinâmica de matrizes (Levenshtein).

Seja w o tamanho de uma *computer word* que pode ser de 32 ou 64 bits. Podemos utilizar uma máscara para representar diversas palavras utilizando uma única *computer word* isto possibilita que sejam feitos *matchs* em conjuntos de caracteres em vez de ser feito em um único caractere.

Esta técnica melhora a eficiência dos algoritmos de Levenshtein e NFA, e os valores dos escores obtidos nas comparações não são modificados. Ela poderia ser utilizada na nossa aplicação para que houvesse um ganho na eficiência.

3.3 Comparação da Eficiência

Na figura 7 temos uma comparação da eficiência dos principais algoritmos apresentados neste trabalho. A primeira figura mostra os resultados para m = 10 e a segunda para m =30, onde m é o tamanho das palavras comparadas. Esta comparação foi feita com linguagem natural em inglês.

Esta comparação foi feita com strings que possuem um número limitado de caracteres, no máximo 30, pois nas aplicações em geral o número de caracteres comparado não ultrapassa este valor.

Em ambos gráficos da figura 7, k representa o número de erros (que varia de 0 a m) ao longo do eixo horizontal. O tempo obtido é representado por t no eixo vertical.

Quanto às siglas contidas na figura: CFT e CLP são o algoritmo de Levenshtein melhorados no caso médio (em relação ao tempo); DFA e RUS são duas maneiras distintas de implementar o Autômato Não Determinístico (NFA); BPR e BPD são implementações do NFA utilizando bit-parallelism; e BPM é uma implementação de programação dinâmica (Levenshtein) utilizando bit-parallelism.

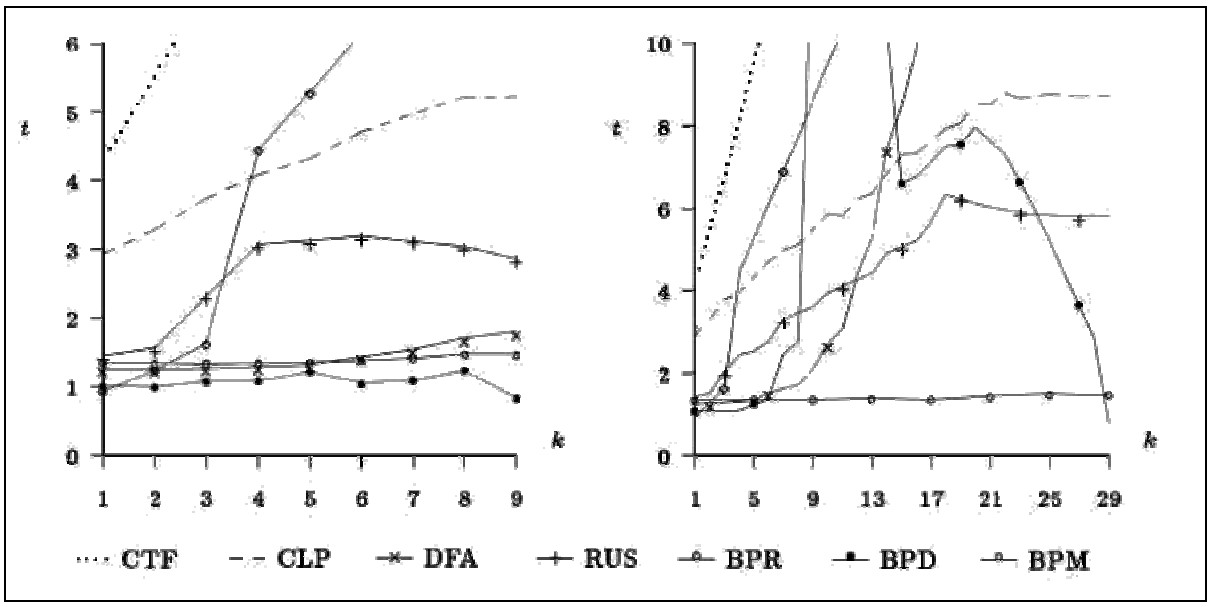

Figura 7 – Comparação da Eficiência dos Algoritmos [NAVARRO 01]

Podemos observar que os algoritmos de programação dinâmica (CFT e CLP) não têm um desempenho muito bom. Já os demais possuem um desempenho parecido para uma quantidade de erros (k) pequena.

O algoritmo de programação dinâmica que utiliza bit-parallelism (BPM) é o que apresenta desempenho mais regular e o que se sai melhor para strings grandes e com alta taxa de erro. Sua utilização poderia melhorar o desempenho da nossa aplicação.

4. Alinhamento Local

Neste capitulo mostraremos os resultados das Comparações de Strings que foram feitas, através de diversos algoritmos. Os algoritmos foram comparados através de um Programa Escrito em Java. E tanto as Strings utilizadas como entrada quanto os resultados da comparação foram respectivamente lidos e escritos em arquivos.

Neste tipo de comparação as duas strings são tratadas como duas cadeias de caracteres. Portanto, palavras simples e compostas são tratadas da mesma forma, não havendo comparações especificas com as strings que formam as palavras compostas. Por exemplo, ao se comparar "apresenta evolucao" com outra string ambas serão consideradas uma única string não havendo comparação isolada com as substrings "apresenta" e "evolucao".

O alinhamento local permite que as duas substrings que possuam maior similaridade sejam alinhadas, descartando as partes não similares das strings originais. Isto permite que as strings "telephone" e "phone" possuam um bom alinhamento.

4.1 Comparação da Produção de Escores

O arquivo assim obtido foi utilizado para a criação da Tabela 5, que contém os escores obtidos pelos algoritmos, para cada dupla de strings comparadas. Esta tabela está ordenada em função da média dos escores dos algoritmos, e seus dados serão utilizados para a criação de diversos gráficos que veremos a seguir. Na ultima coluna temos os escores do algoritmo selecionado, que será explicado na seção 4.1.5.

Tabela 5 – Escores Obtidos na Comparação de Strings

nº	strings	lev	smi	convdis	sdx	qgram	jaro	jarowin	monge	TFIDF	slimTFI	jaroWT	MEDIA	*SMI+JA*
1	(agente de saude,caso)	0.0000	0.1053	0.1474	0.5556	0.0000	0.0000	0.0000	0.4000	0.0000	0.3801	0.5421	***0.1937***	***0.0526***
2	(paciente,caso)	0.0000	0.1667	0.2333	0.6667	0.0000	0.4167	0.4167	0.4000	0.0000	0.3750	0.4167	***0.2811***	***0.2917***
3	(paciente,medico)	0.0000	0.1429	0.2571	0.6944	0.0000	0.5139	0.5139	0.3667	0.0000	0.5000	0.5139	***0.3184***	***0.3284***
4	(caso clinico,paciente)	0.0000	0.2000	0.1800	0.6667	0.0000	0.5278	0.5278	0.2750	0.0000	0.6503	0.7197	***0.3407***	***0.3639***
5	(agente de saude,medico)	0.0000	0.1429	0.1714	0.5556	0.0000	0.4889	0.4889	0.3667	0.0000	0.7409	0.8468	***0.3456***	***0.3159***
6	(caso clinico,medico)	0.0000	0.3333	0.3333	0.5556	0.2727	0.4722	0.4722	0.5000	0.0000	0.7239	0.4546	***0.3744***	***0.4028***
7	(inicio caso,dataincio)	0.0000	0.4500	0.3700	0.6667	0.1667	0.6263	0.6263	0.4444	0.0000	0.9723	0.5631	***0.4441***	***0.5381***
8	(agente de saude,paciente)	0.0000	0.3478	0.3304	0.7778	0.2222	0.6528	0.6528	0.5000	0.0000	0.8612	1.0000	***0.4859***	***0.5003***
9	(queixa principal,pro queixa paciente)	0.2571	0.4857	0.4514	0.6944	0.3077	0.7278	0.7278	0.5375	0.4082	0.7068	0.6955	***0.5455***	***0.6067***
10	(evolucao caso,apres evolucao)	0.1852	0.5926	0.5556	0.6667	0.4516	0.7258	0.7258	0.6154	0.5000	0.8125	0.8167	***0.6043***	***0.6592***
11	(survey,surgery)	0.6154	0.6154	0.5538	0.8444	0.4706	0.8492	0.8944	0.5667	0.0000	0.8917	0.8944	***0.6542***	***0.7323***
12	(caso clinico,caso)	0.0000	0.5000	0.4750	0.7667	0.5000	0.7778	0.8667	1.0000	0.7071	1.0000	1.0000	***0.6903***	***0.6389***
13	(queixa prin,pro queixa prim)	0.5769	0.7692	0.7308	0.6944	0.5333	0.7253	0.7253	0.9636	0.4082	0.7451	0.7689	***0.6946***	***0.7472***
14	(crm ag,crm)	0.3333	0.6667	0.6667	0.9333	0.4615	0.8333	0.8833	1.0000	0.7071	0.7071	0.7071	***0.7181***	***0.7500***
15	(telefone ag,telefone1)	0.6500	0.8000	0.7600	0.9444	0.6667	0.8721	0.9232	0.8889	0.5000	0.5000	0.5000	***0.7278***	***0.8360***
16	(queixa pac,queixa prin)	0.6190	0.7619	0.7238	0.8667	0.6400	0.8424	0.9055	0.8000	0.5000	0.6812	0.7875	***0.7389***	***0.8022***
17	(nome ag,nome)	0.4545	0.7273	0.6909	0.9222	0.5333	0.8571	0.9143	1.0000	0.7071	0.7071	0.7071	***0.7474***	***0.7922***
18	(email ag,email)	0.5385	0.7692	0.7231	0.9333	0.5882	0.8750	0.9250	1.0000	0.7071	0.9546	1.0000	***0.8195***	***0.8221***
19	(uf crm ag,uf crm)	0.6000	0.8000	0.7867	1.0000	0.6316	0.8889	0.9333	1.0000	0.8165	0.8165	0.8165	***0.8264***	***0.8444***
20	(paciente,paciente)	1.0000	1.0000	0.9500	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	***0.9955***	***1.0000***
	MEDIA	**0.2915**	**0.5188**	**0.5045**	**0.7703**	**0.3723**	**0.6837**	**0.7062**	**0.6812**	**0.3481**	**0.7363**	**0.7375**	***0.5773***	***0.6012***

Os algoritmos comparados são de 3 fontes:

- **Implementação Própria** (3 algoritmos de edição de distância)
- **Biblioteca SimMetrics**

String Similarity Metrics for Information Integration – biblioteca java Open Source para medida de similaridade entre strings.
Disponível em: http://sourceforge.net/projects/simmetrics/

- **Biblioteca Secondstring**

A Open Source Java Toolkit para algoritmos de aproximação de string (técnicas de string Match).
Disponível em: http://secondstring.sourceforge.net

Estes foram os algoritmos comparados:

- Implementação Realizada:
  - Levenshtein Metric
  - Smith Waterman
  - Convigton Distance
- Biblioteca SimMetrics
  - Soundex
  - QGram Distance
- Biblioteca Secondstring
  - Jaro
  - Jaro Winkler
  - Monge Elkan
  - TFIDF
  - Slim TFIDF
  - Jaro Winkler TFIDF
- Algoritmo Selecionado
  - Smith Waterman + Jaro

4.1.1 Avaliação Geral

Neste primeiro gráfico (Figura 8) temos uma comparação de todos os algoritmos descritos acima. Nas seções 4.2, 4.3 e 4.4 será feita uma comparação em separado dos algoritmos das 3 fontes de dados vistas acima.

A partir de agora, o nome **média** irá representar a **linha média** que está presente em todos os gráficos (neste caso em amarelo). Ela será considerada como o escore ideal, e os algoritmos serão considerados com boa produção de escores quando os seus valores conseguirem acompanhar os valores desta linha.

Quando se calcula o escore através de um único algoritmo é possível que o valor atribuído esteja distorcido, mas quando se considera que a média será o escore ideal estas distorções são corrigidas. Pois as distorções quando um algoritmo atribui um escore muito alto para a comparação passa a ser corrigida por outra distorção onde um escore muito baixo é atribuído, e vice-versa. Para isto é importante que uma grande quantidade de algoritmos seja utilizada para calcular a média, pois quanto maior for à quantidade de algoritmos maior será a interação entre os escores dos algoritmos e menor será a distorção.

A estratégia da média tem como objetivo evitar que hajam distorções. Ao se comparar, por exemplo, "paciente" e "medico" o escore não deve ser muito alto já que estas strings não possuem nem semelhança sintática nem semântica.

Já ao se comparar strings como "queixa pac" e "queixa prin" o escore deverá ser relativamente alto, pois estas strings apresentam tanto semelhança sintática quanto semântica. Os algoritmos fazem apenas comparações sintáticas, mas o objetivo é que semelhanças sintáticas reflitam em semelhanças semânticas.

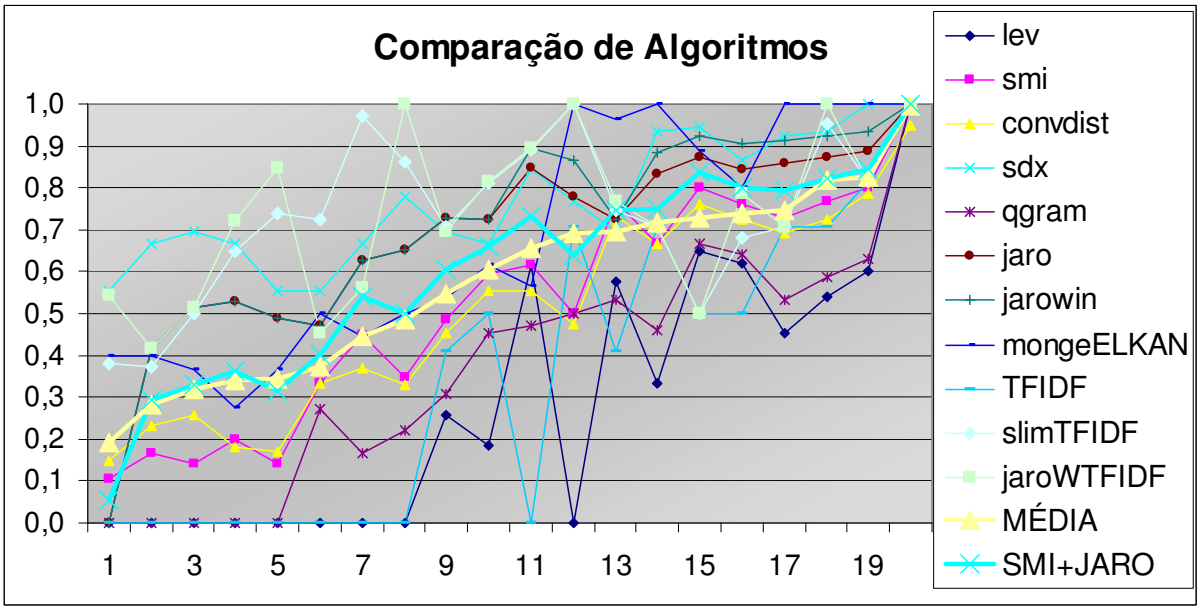

Figura 8 – Gráfico Comparando produção de Escores de todos os Algoritmos

Na Figura 9 temos uma comparação entre as médias dos escores vistos no gráfico acima. Podemos ver que o algoritmo que mais se aproximou da média foi a combinação do "Smith Waterman + Jaro".

Dentre os que mais se distanciaram estão o Soundex que ficou muito acima da média e os algoritmos Qgram e TFIDF que ficaram muito abaixo do valor da média.

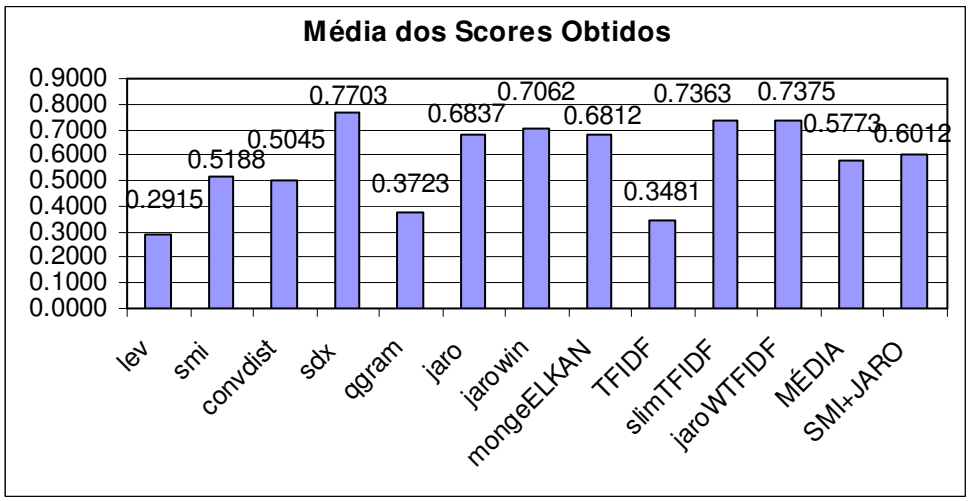

Figura 9 – Media dos Escores Obtidos

4.1.2 Algoritmos Implementados

No gráfico da Figura 10 temos um gráfico do desempenho dos algoritmos que foram implementados neste trabalho. Os algoritmos de Smith Waterman e Convigton Distance tiveram um comportamento muito próximo ao da linha média, mas ficaram um pouco abaixo do valor que deveriam ter.

O Convigton Distance foi o único que não apresentou o valor 1,0000 para strings idênticas, pois ele não considera o casamento de duas vogais como um match exato.

O algoritmo Levenshtein Metric não acompanhou a média, pois ora ficava acima do seu valor e ora ficava abaixo. Isto ocorreu em parte por que seu alinhamento é global enquanto os alinhamentos dos outros dois algoritmos vistos aqui são locais.

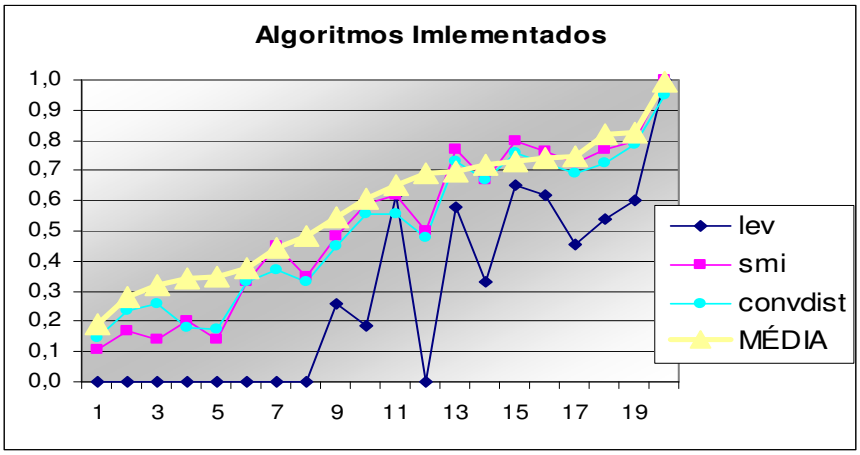

Figura 10 – Gráfico do Escores dos Algoritmos Implementados

4.1.3 Biblioteca SimMetrics

No gráfico da Figura 11, temos um gráfico do desempenho dos algoritmos da Biblioteca SimMetrics. Os dois algoritmos comparados aqui não tiveram bom desempenho, o ideal seria que o valor médio destes dois algoritmos se aproximasse do valor da linha média.

O Algoritmo Soundex não ficou muito estável, variando bastante. E, além disso, o seu valor médio foi muito maior que o da média.

O Qgram teve um comportamento um pouco melhor que o do Soundex, mas mesmo assim não acompanhou a linha média muito bem e o seu valor médio foi muito baixo.

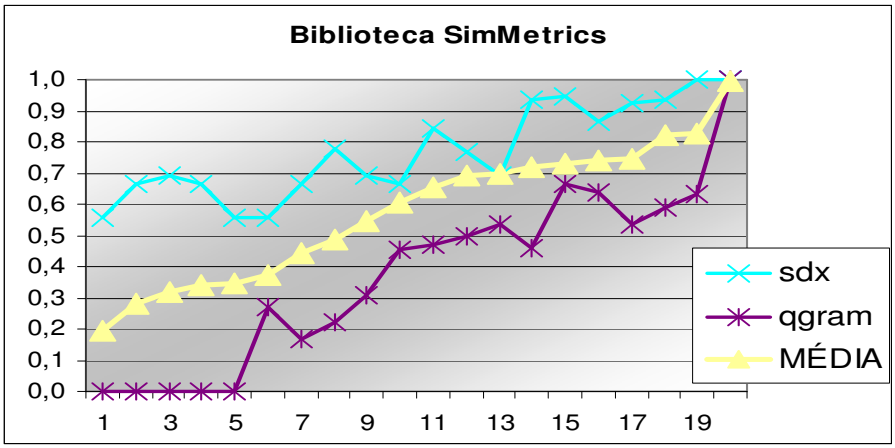

Figura 11 – Gráfico do Escores da Biblioteca SimMetrics

4.1.4 Biblioteca Secondstring

No gráfico da Figura 12 temos um gráfico do desempenho dos algoritmos da Biblioteca Secondstring. Os algoritmos Jaro e Jaro Winkler foram os que produziram os melhores escores, ambos ficaram próximos da linha média, embora com valores um pouco acima. E o Jaro se apresentou melhor que o Jaro Winkler, pois seus valores foram mais realistas, aproximando-se mais da linha média.

O algoritmo Monge Elkan é praticamente idêntico à linha média, para strings com pouca similaridade, mas quando a similaridade aumenta o seu valor se aproxima de 1,000 ficando muito distante do valor da média.

O algoritmo TFIDF *term frequency - inverse document frequency* tem como objetivo comparar strings compostas, ou seja, strings formadas por tokens (sub-strings). E para ele a ordem destes tokens não tem importância, o importante é que os tokens sejam exatamente iguais.

Mas como o nosso objetivo não é ter tokens exatamente iguais e a ordem é importante, o algoritmo TFIDF foi o que obteve os piores resultados. Por exemplo, deve-se considerar que tokens como "cod" e "codigo" são semelhantes, e deve-se considerar a ordem dos tokens como no caso de "queixa pac" e "queixa prin". Estes dois tipos de verificações são ignorados no TFIDF e por isto ele não obteve bons resultados.

Os algoritmos Slim TFIDF e Jaro Winkler TFIDF são algoritmos híbridos que combinam funções de edição de distância com o TFIDF (que é baseado em tokens). Mas estes algoritmos também não tiveram bons resultados, pois não acompanharam a linha média, já que seus valores variaram bastante. E os resultados foram ainda piores para strings que deveriam ter baixa similaridade e acabaram tendo similaridade.

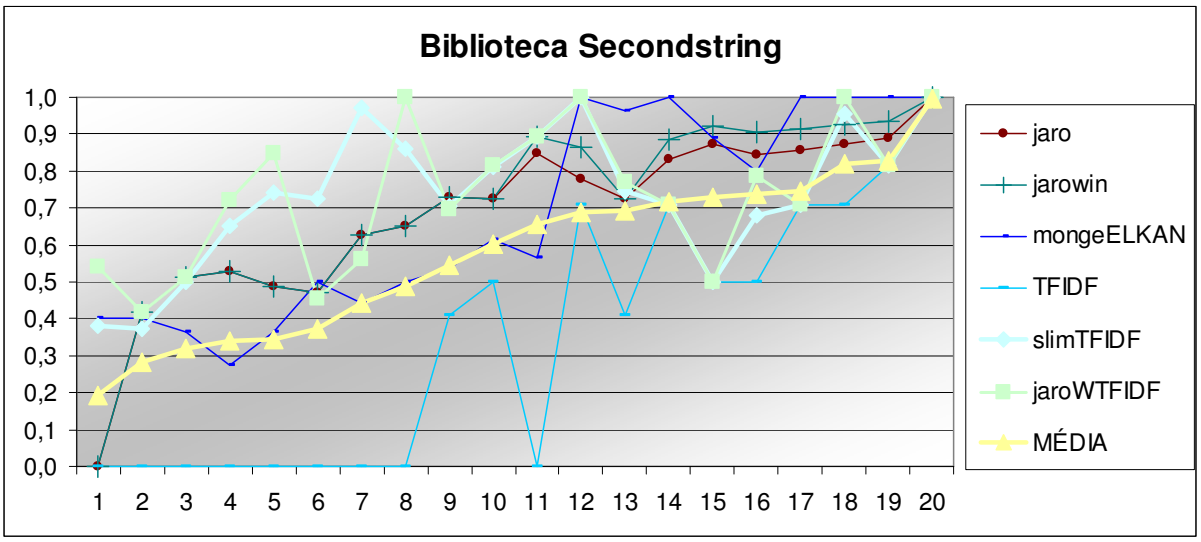

Figura 12 – Gráfico do Escores da Biblioteca Secondstring

4.1.5 Algoritmos Selecionados

No gráfico da Figura 13 estão dois algoritmos, Smith Waterman e Jaro, que tiveram melhor desempenho (algoritmos que mais se aproximaram da linha média).

Foi observado que o Smith Waterman ficava um pouco abaixo da linha média e que o Jaro ficava um pouco acima desta linha. Portanto optou-se por fazer a média do Smith Waterman com o Jaro e como podemos ver a linha criada é quase idêntica à linha média, com exceção de poucos casos onde a linha média teve sua posição demasiadamente alterada por valores incoerentes de outros algoritmos.

Como foi visto na seção 3.1.4, o Jaro Metric se baseia no número de caracteres comuns entre duas strings, e na semelhança da ordem na qual estas duas cadeias de caracteres se apresentam. Em geral obtém-se bons resultados com este algoritmo, mas pode haver distorções ao se comparar, por exemplo, "paciente" e "medico" que possuem vários caracteres em comum: "c,e,i,o". O escore desta comparação é de 0,514, o que representa uma grande distorção, já que estas strings não representam nem semelhança sintática nem semântica.

Já o Smith Waterman (visto na seção 3.1.2), passa a ter um escore muito baixo quando o tamanho das duas strings é muito diferente, já que o seu escore se baseia no número de caracteres em comum e na mesma ordem, dividido pelo tamanho médio das strings. Por exemplo, ao se comparar "caso" e "caso clinico", obtemos uma similaridade de 0,50 que é um escore muito baixo, pois neste caso uma das strings é exatamente o primeiro token da outra string.

Com a junção (média aritmética) destes dois algoritmos, um algoritmo passou a aumentar o escore da comparação no mesmo ponto onde o outro algoritmo decrementa este escore. Isto impede que hajam distorções tanto para valores acima quanto para valores abaixo do ideal.

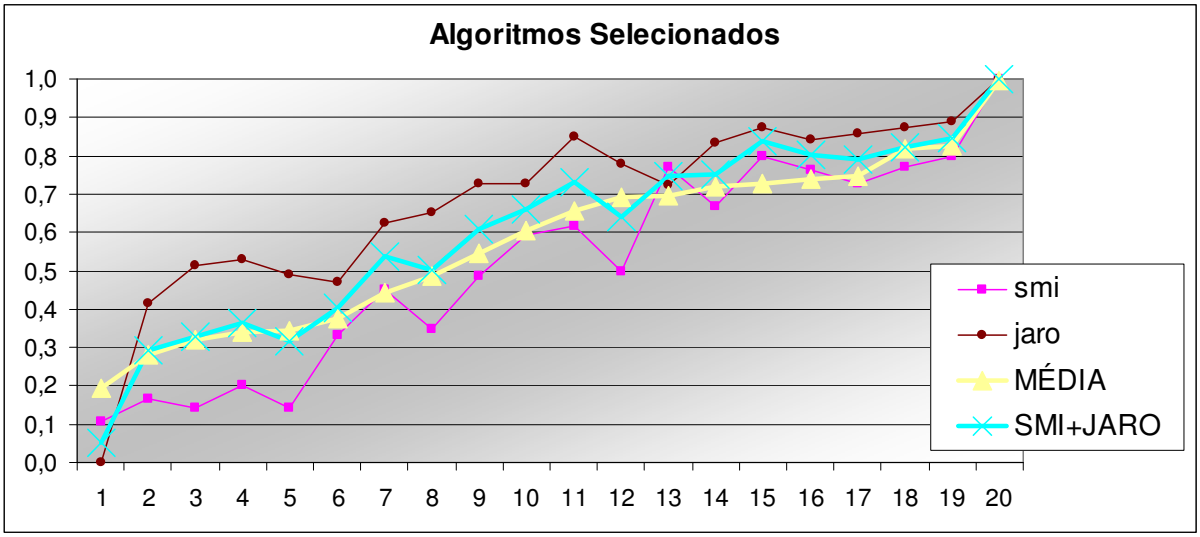

Figura 13 – Grãfico do Escores dos Algoritmos Selecionados

4.2 Tabela Comparativa

A tabela 6 resume as características[1] dos algoritmos vistos anteriormente. As colunas referem-se respectivamente a:

- Qualidade da produção de escores, baseada nos dados obtidos neste Capítulo;
- Define se os algoritmos são aplicáveis à comparação da similaridade entre strings (pois eles podem ser aplicáveis apenas à comparação de DNA, fonética, ou reconhecimento da fala);
- Tipo de Alinhamento que pode ser Global, Semi-Global e Local (descritos no início do capítulo 2). O tipo de alinhamento pode ser transformado na maioria dos casos;
- Aplicável ao nosso problema são os algoritmos com produção de escore razoável, aplicáveis à lingüística e com alinhamento Local ou adaptável para este tipo de alinhamento;

Tabela 6 – Comparação dos Algoritmos

Nª	Algoritmo	Produção de Escores	Aplicável a Lingüística	Tipo de Alinhamento	Aplicável ao Nosso Problema
1	**Levenshtein**	Regular	SIM	Global	NÃO
2	**Smith Waterman**	Muito Boa	SIM	Local	SIM
3	**Alinhamento Inicial**	Muito Boa	SIM	Inicial	SIM
4	**Stochastic Model**	Boa	SIM	Global	SIM[2]
5	**Jaro Metric**	Muito Boa	SIM	Global	SIM
6	**Hamming distance**	Ruim	SIM	Global	NÃO
7	**Soundex**	Ruim	NÃO	Global	NÃO
8	**Covington's distance**	Muito Boa	SIM	Global	SIM²
9	**NFA**	Boa	SIM	Semi - Global	SIM²
10	**BLAST**	Boa	SIM	Local	SIM
11	**q-gram**	Ruim	SIM	Global	SIM²
12	**Bit-Parallelism[3]**	-	SIM	-	SIM

[1] A eficiência dos algoritmos não está sendo avaliada aqui, pois estes serão utilizados para comparar strings pequenas que não exigem um grande tempo computacional, e mesmo assim o processo de comparação será realizado em uma única vez.
[2] Estes algoritmos devem ser adaptados para produzirem alinhamento local;
[3] Bit-Parallelism não é exatamente um algoritmo, na verdade é uma técnica que pode ser utilizada para melhorar o desempenho de outros algoritmos.

5. Alinhamento Inicial

Como nosso objetivo é que o escore tenha valores mais altos quando o inicio de duas strings for semelhante, foi preciso criar um novo algoritmo, pois não existia nenhum algoritmo com esta finalidade. Como já foi dito o objetivo do alinhamento inicial é o de dar relevância à similaridade dos caracteres iniciais das duas strings comparadas, por exemplo "inicio" e "ini" tem caracteres iniciais semelhantes. Isto indica que, apesar de o número de caracteres diferir bastante, estas string são bem similares.

A palavra token é utilizada para representar o número de sub-strings presentes em uma string. Por exemplo, podemos ver que "apresenta evolucao" possui dois tokens.

O objetivo do alinhamento inicial é comparar strings com um único token. Quando for necessário comparar strings com mais de um token, estes tokens devem ser separados e as comparações devem ser feitas isoladamente.

O código completo deste algoritmo, que foi implementado na linguagem Java, pode ser encontrado no Apêndice A.

5.1 Descrição do Alinhamento Inicial

Este algoritmo também é semelhante ao Levenshtein que foi descrito na seção 3.1.1. Ele alinha o começo de duas strings como o de Levenshtein, mas ignora baixos escores no final da comparação.

Neste algoritmo procedemos da mesma forma que no Levenshtein, mas consideramos como escore selecionado o maior escore obtido em toda a matriz, e não o ultimo escore.

Sua diferença em relação ao algoritmo de Smith-Waterman é que neste algoritmo, não substituímos, por zero, os escores negativos.

A função, algoritmo propriamente dito, do alinhamento inicial é semelhante à de Levenshtein, mas como já foi dito o escore do alinhamento será o maior escore da matriz.

O alinhamento a ser escolhido será o de maior escore em qualquer posição da matriz. Na Tabela 7 temos um exemplo, comparamos "medico" com "biomedicina". Podemos observar que o melhor escore obtido foi 7.

Tabela 7 – Exemplo de Matriz de Alinhamento Inicial

	ε	**b**	**i**	**o**	**m**	**e**	**d**	**i**	**c**	**i**	**n**	**a**
ε	**0**	**-1**	**-2**	**-3**	-4	-5	-6	-7	-8	-9	-10	-11
m	-1	-1	-2	-3	**-1**	-2	-3	-4	-5	-6	-7	-8
e	-2	-2	-2	-3	-2	**1**	0	-1	-2	-3	-4	-5
d	-3	-3	-3	-3	-3	0	**3**	2	1	0	-1	-2
i	-4	-4	-1	-2	-3	-1	2	**5**	4	3	2	1
c	-5	-5	-2	-2	-3	-2	1	4	**7**	6	5	4
o	-6	-6	-3	0	-1	-2	0	3	6	6	5	4

Este é o alinhamento obtido:

```
- - - m e d i c - - o
b i o m e d i c i n a
```

O seu desempenho é semelhante ao do Levenshtein, e seu alinhamento é inicial, o que impede que duas strings que tenham apenas caracteres semelhantes no meio da string, como é o caso, por exemplo, de "bairro" e "cirrose" possuam um bom escore de alinhamento.

Outra vantagem deste algoritmo é que ele possibilita que strings como "telefone" e "tele" possuam um bom escore de alinhamento, ou seja, strings com o mesmo prefixo, e strings que sejam uma espécie de abreviação de outra string sempre terão um bom escore quando comparados à string original.

5.2 Comparação do Algoritmo Inicial com os Existentes

A Tabela 8 contém os escores obtidos por diversos algoritmos incluindo o de alinhamento inicial, para cada dupla de strings comparadas. Esta tabela está ordenada em função do escore obtido pelo algoritmo de alinhamento inicial, e seus dados serão utilizados para a criação dos gráficos que veremos a seguir.

Como podemos ver estamos comparando apenas strings compostas por um único token (strings simples), pois este é o intuito de nossa aplicação, e mesmo assim este algoritmo não daria bons escores para strings com mais de um token.

Na tabela 8 podemos ver que este algoritmo (**alinhamento inicial**) cumpriu o seu objetivo, que é o de não dar escores muito altos para strings que precisem fazer vários deslocamentos para ficarem alinhadas. Este é o caso da comparação de "bairro" com "cirrose" (linha 13).

Ao mesmo tempo este algoritmo manteve os bons escores quando comparou strings com os mesmos prefixos, como é o caso de "tele" e "telefone" (linha 21), e "num" e "numero" (linha 22).

No entanto este algoritmo pode gerar alguns problemas quando comparar strings que tenham apenas sufixos em comum, pois ele dará baixos escores a strings com grande similaridade como é o caso de "fone" e "telefone" (linha 9),e "inicio" e "dataincio" (linha 10).

Tabela 8 – Escores Obtidos na Comparação de Alinhamento Inicial

nº	strings	ini	lev	smi	jaro	smi+jar	MEDIA
1	(caso,medico)	**0,0000**	*0,0000*	0,2000	0,0000	0,1000	***0,0750***
2	(agente,medico)	**0,0000**	*0,0000*	0,1667	0,4444	0,3056	***0,2292***
3	(paciente,medico)	**0,0000**	*0,0000*	0,1429	0,5139	0,3284	***0,2463***
4	(caso,paciente)	**0,0833**	*0,0000*	0,1667	0,4167	0,2917	***0,2188***
5	(paciente,caso)	**0,0833**	*0,0000*	0,1667	0,4167	0,2917	***0,2188***
6	(agente,caso)	**0,1000**	*0,0000*	0,2000	0,4722	0,3361	***0,2521***
7	(principal,paciente)	**0,1765**	*0,0000*	0,2353	0,5648	0,4001	***0,3000***
8	(id,identificador)	**0,2667**	*0,0000*	0,2667	0,7179	0,4923	***0,3692***
9	(fone,telefone)	**0,3333**	*0,3333*	0,6667	0,6667	0,6667	***0,5833***
10	(inicio,dataincio)	**0,3333**	*0,3333*	0,6000	0,7963	0,6981	***0,6069***
11	(status,estado)	**0,4167**	*0,1667*	0,5000	0,6667	0,5833	***0,4792***
12	(numreginst,numero)	**0,4375**	*0,1250*	0,4375	0,7111	0,5743	***0,4620***
13	(bairro,cirrose)	**0,4615**	*0,3077*	0,6154	0,7460	0,6807	***0,5875***
14	(ag,agente)	**0,5000**	*0,0000*	0,5000	0,7778	0,6389	***0,4792***
15	(agente,paciente)	**0,5000**	*0,5000*	0,5714	0,8194	0,6954	***0,6466***
16	(agente,paciente)	**0,5000**	*0,5000*	0,5714	0,8194	0,6954	***0,6466***
17	(medico,etico)	**0,5455**	*0,5455*	0,6364	0,8222	0,7293	***0,6833***
18	(survey,surgery)	**0,6154**	*0,6154*	0,6154	0,8492	0,7323	***0,7031***
19	(doctor,doc)	**0,6667**	*0,3333*	0,6667	0,8333	0,7500	***0,6458***
20	(cod,codigo)	**0,6667**	*0,3333*	0,6667	0,8333	0,7500	***0,6458***
21	(tele,telefone)	**0,6667**	*0,3333*	0,6667	0,8333	0,7500	***0,6458***
22	(num,numero)	**0,6667**	*0,3333*	0,6667	0,8333	0,7500	***0,6458***
23	(prin,prim)	**0,7500**	*0,6250*	0,7500	0,8333	0,7917	***0,7500***
24	(telefone,telefone1)	**0,9412**	*0,8824*	0,9412	0,9630	0,9521	***0,9346***
25	(paciente,paciente)	**1,0000**	*1,0000*	1,0000	1,0000	1,0000	***1,0000***
	MEDIA	**0,4284**	**0,2907**	**0,5047**	**0,6940**	**0,5994**	**0,5222**

No gráfico da Figura 14, temos uma comparação entre as médias dos escores vistos na Tabela 8. Podemos ver que o algoritmo de alinhamento inicial teve escore maior do que o algoritmo de Levenshtein, mas menor do que os outros algoritmos. Este comportamento é exatamente o esperado e será melhor explicado a seguir.

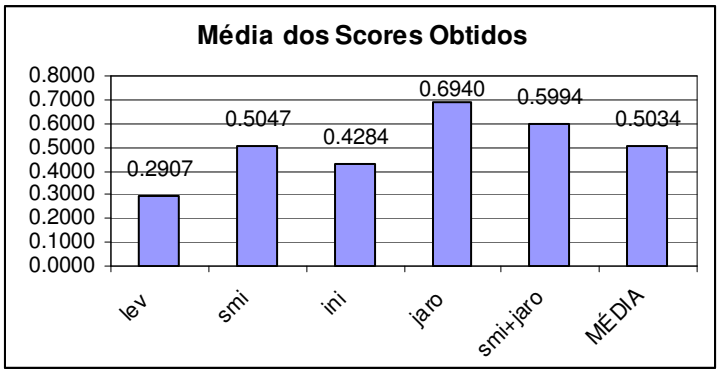

Figura 14 – Média dos Escores Obtidos no Alinhamento Inicial

Na Figura 15 também podemos ver que o algoritmo de **alinhamento inicial** cumpriu o comportamento esperado. O seu escore foi maior do que o obtido no algoritmo de Levenshtein, porque o alinhamento inicial ocorre da mesma forma que o de Levenshtein, mas no alinhamento inicial o escore selecionado é o maior escore presente na

Matriz gerada, portanto seu escore sempre será maior ou igual ao escore do Levenshtein.

No algoritmo de Smith Waterman os escores menores do que zero são substituídos por zero (na Matriz gerada), isto permite que o alinhamento não precise começar no início das strings. Portanto, o escore deste algoritmo sempre será maior ou igual ao escore do algoritmo inicial, pois no inicial não se pode substituir valores negativos por zero.

O algoritmo de Jaro se baseia no número de caracteres comuns entre duas strings, e na semelhança da ordem na qual estas duas cadeias de caracteres se apresentam, não se preocupando se os caracteres estão próximos uns dos outros. Isto explica os altíssimos escores dados por este algoritmo no gráfico da Figura 15.

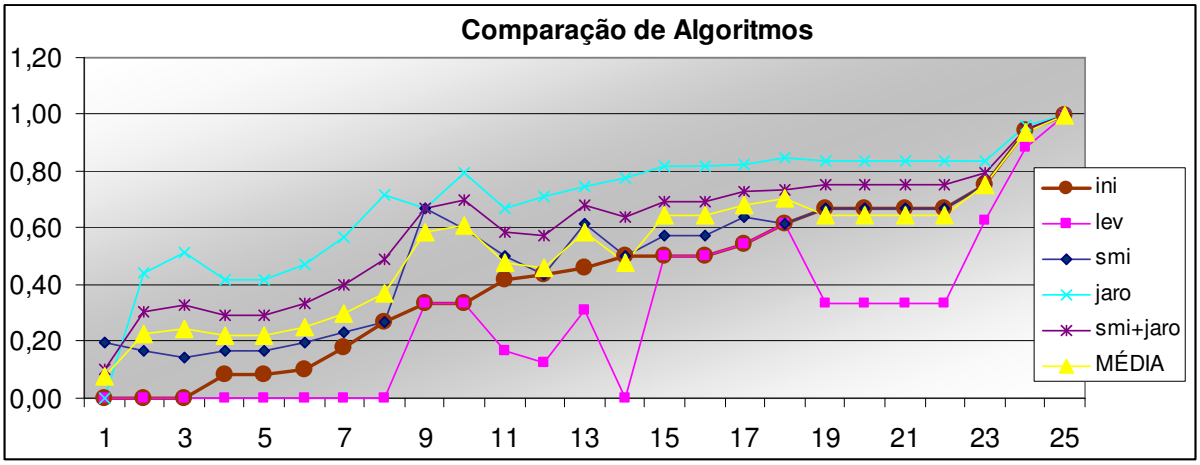

Figura 15 – Gráfico Comparando Escores para Alinhamento Inicial

5.3 Derivações do Algoritmo Inicial

O algoritmo Inicial foi o que obteve melhores resultados nos testes descritos na seção 5.2. Nesta seção descrevemos as alterações que foram realizadas com o intuito de melhorar ainda mais a qualidade do escore produzido por este algoritmo.

Foram feitas diversas tentativas para que os escores se baseassem ainda mais nas semelhanças entre os caracteres iniciais de duas strings. Nesta seção serão apresentadas as tentativas realizadas e depois serão apresentados os resultados obtidos.

Testes realizados para modificar os escores dos três caracteres iniciais das strings:

1. Atualmente quando ocorre um *match* entre dois caracteres é atribuído um valor 2 a este *match*; quando os caracteres diferem é atribuído o valor negativo -1.

Com esta modificação o **Peso** do *match* dos três primeiros caracteres que atualmente é de 2-2-2, passa a ser 5-4-3;

2. Adicionar pontuações considerando as duas strings comparadas em seu estado bruto – comparação **Incondicional**. Normalmente o

escore varia de 0 a 1, mas com esta modificação são adicionados os valores 0.10, 0.06 e 0.04 respectivamente, no caso de os três primeiros caracteres das duas strings serem semelhantes. Neste caso o escore passa a variar de 0 a 1.2 e se torna necessário normalizar o seu valor para que ele volte a variar entre 0 e 1;

3. É semelhante à modificação 2, só que neste caso são comparadas as duas strings depois de alinhadas – comparação **Condicional**, ou seja depois que recebem hífens "–" para separar as partes semelhantes das não semelhantes, como pode ser visto na Figura 16.

Abaixo temos um exemplo de como é feito o incremento do escore para as modificações 2 e 3, mas especificamente para a modificação 3. Neste caso o escore é incrementado devido a *matchs* no 1º e 3º caracteres:

q	u	e	i	x	a	_	p	*r*	*i*	*n*
q	-	e	i	x	a	_	p	*a*	*c*	-
+0.10	0	+0.04								

Outra modificação que pode ser feita é quanto a Dependência, na hora de se continuar a incrementar os escores:

a. Dependente – só incrementa o escore do próximo caractere caso o caractere atual tenha sido incrementado. No exemplo visto acima, os caracteres da segunda posição "u" e "–" diferem, portanto não haveria incremento devido ao *match* dos caracteres da terceira posição "e" e "e", mesmo sendo semelhantes;

b. Independente – ignora se houve *match* ou *mismatch* na comparação anterior. Neste caso haveria incremento devido à semelhança dos caracteres da terceira posição "e" e "e".

Na tabela 9 podemos ver os resultados obtidos com as derivações do algoritmo de alinhamento inicial. A seguir serão feitas considerações sobre os resultados obtidos com as diversas modificações propostas nesta seção.

Tabela 9 – Escores das Derivações do Alinhamento Inicial

			Independente			Dependente			
nº	strings	Inicial	Peso	Incondi	Condic	Peso	Incondi	Condic	MEDIA
1	(paciente,medico)	*0,0000*	0,0000	0,0000	0,0000	0,0000	0,0000	**0,0000**	*0,0000*
2	(paciente,caso)	*0,0833*	0,1667	0,1194	0,1194	0,0556	0,0694	**0,0694**	*0,0976*
3	(agente,caso)	*0,1000*	0,0667	0,0833	0,0833	0,0667	0,0833	**0,0833**	*0,0810*
4	(principal,paciente)	*0,1765*	0,2353	0,2304	0,2304	0,2353	0,2304	**0,2304**	*0,2241*
5	(id,identificador)	*0,2500*	0,3750	0,3417	0,3417	0,3750	0,3417	**0,3417**	*0,3381*
6	(fone,telefone)	*0,3333*	0,2222	0,2778	0,2778	0,2222	0,2778	**0,2778**	*0,2698*
7	(email,edema)	*0,4000*	0,4667	0,4167	0,3333	0,4667	0,4167	**0,3333**	*0,4048*
8	(complemento,coma)	*0,4000*	0,5333	0,5000	0,5000	0,5333	0,5000	**0,5000**	*0,4952*
9	(status,estado)	*0,4167*	0,2778	0,3472	0,3472	0,2778	0,3472	**0,3472**	*0,3373*
10	(caso,basofilo)	*0,4167*	0,4444	0,4306	0,4306	0,2778	0,3472	**0,3472**	*0,3849*
11	(numreginst,numero)	*0,4375*	0,5417	0,5313	0,5313	0,5417	0,5313	**0,5313**	*0,5208*
12	(ag,agente)	*0,4444*	0,6667	0,5037	0,5037	0,6667	0,5037	**0,5037**	*0,5418*
13	(bairro,cirrose)	*0,4615*	0,3077	0,3846	0,3846	0,3077	0,3846	**0,3846**	*0,3736*
14	(nascimento,pacientes)	*0,4737*	0,3860	0,4447	0,4447	0,3158	0,3947	**0,3947**	*0,4078*
15	(agente,paciente)	*0,5000*	0,3333	0,4167	0,4167	0,3333	0,4167	**0,4167**	*0,4048*
16	(nome,numero)	*0,5000*	0,6000	0,5333	0,5333	0,5333	0,5000	**0,5000**	*0,5286*
17	(inicio,clinico)	*0,5385*	0,3590	0,4821	0,4487	0,3590	0,4487	**0,4487**	*0,4407*
18	(medico,etico)	*0,5455*	0,3636	0,4545	0,4545	0,3636	0,4545	**0,4545**	*0,4416*
19	(paciente,paracentese)	*0,5789*	0,5614	0,6158	0,5658	0,5614	0,6158	**0,5658**	*0,5807*
20	(tele,telefone)	*0,6667*	0,7778	0,7222	0,7222	0,7778	0,7222	**0,7222**	*0,7302*
21	(doctor,doc)	*0,6667*	0,8889	0,7222	0,7222	0,8889	0,7222	**0,7222**	*0,7619*
22	(medico,medicina)	*0,7143*	0,7619	0,7619	0,7619	0,7619	0,7619	**0,7619**	*0,7551*
23	(prin,prim)	*0,7500*	1,0000	0,7917	0,7917	1,0000	0,7917	**0,7917**	*0,8452*
24	(telefone,telefone1)	*0,9412*	0,8627	0,9510	0,9510	0,8627	0,9510	**0,9510**	*0,9244*
25	(paciente,paciente)	*1,0000*	0,9167	1,0000	1,0000	0,9167	1,0000	**1,0000**	*0,9762*
	MEDIA	**0,4718**	**0,4846**	**0,4825**	**0,4758**	**0,4680**	**0,4725**	**0,4672**	0,4746

A modificação 1 (baseada na modifcação do Peso do *match*) geralmente teve bons resultados, mas em alguns casos foi gerada uma alta pontuação para strings curtas e com relativa similaridade, como é o caso da comparação de "prin" e "prim" que recebeu escore 1,00.

Comparando as modificações 2 e 3. Quando as strings "email" e "edema" são comparadas através da modificação 2, o algoritmo percebe que o primeiro caractere das duas strings "e" é semelhante e em função disto aumenta o escore da comparação. Através da modificação 3, como pode ser visto na Figura 16, podemos ver que as duas strings não estão alinhadas e com isto o escore não é incrementado.

Neste caso o escore realmente não deveria ter sido incrementado, pois as strings comparadas não possuem nem semelhança sintática nem semântica. Esta mesma consideração também é válida para a comparação entre "paciente" e "paracentese".

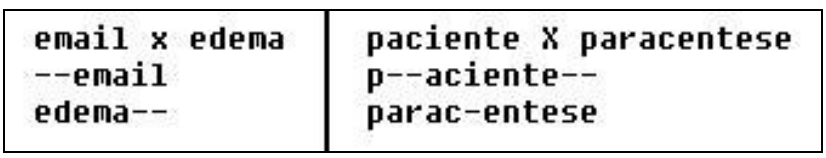

Figura 16 – Exemplos de Alinhamento Inicial Condicional

Quanto a Dependência na hora de se continuar a incrementar o escore, dependendo se o escore anterior era semelhante, foi observado que a qualidade dos escores obtidos com dependência foi melhor do que os dos independentes.

Os escores de diversas comparações tiveram seus valores decrementados quando se utilizou a dependência, e em todos os casos esta diminuição do escore se mostrou correta, pois as strings não eram semelhantes, exemplos (os caracteres semelhantes estão em negrito): "c**as**o" e "b**as**ofilo"; "**n**o**m**e" e "**n**u**m**ero".

Como pode ser visto, a melhor opção de algoritmo é o que utiliza a modificação 3 Comparação Condicional, baseada no alinhamento, e com dependência. No final desta seção serão mostrados alguns resultados que comprovam que esta é a melhor opção.

No gráfico da Figura 17, temos uma comparação entre as médias dos escores vistos na Tabela 9. Podemos ver que os escores atribuídos não variam muito porque todos os algoritmos são uma derivação do algoritmo inicial e têm como objetivo melhorar pequenos detalhes deste algoritmo. Este comportamento é exatamente o esperado, pois as alterações efetuadas neste algoritmo devem ter resultados sutis.

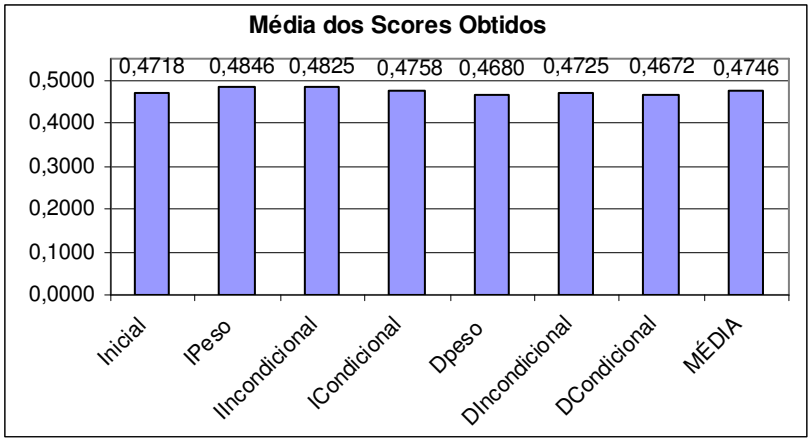

Figura 17 – Média dos Escores das Derivações do Alinhamento Inicial

Na Figura 18 também podemos ver que os algoritmos que sofreram modificação no peso (1) não foram muito constantes, ficando com seus escores distantes dos escores do alinhamento inicial. Isto não é bom, pois mostra que podem ser geradas avaliações erradas dos escores.

Já os algoritmos que sofreram as modificações 1 e 2 (Incondicional e Condicional respectivamente), tiveram um comportamento estável tendo escores próximos dos escores do algoritmo inicial. E este é o comportamento desejado.

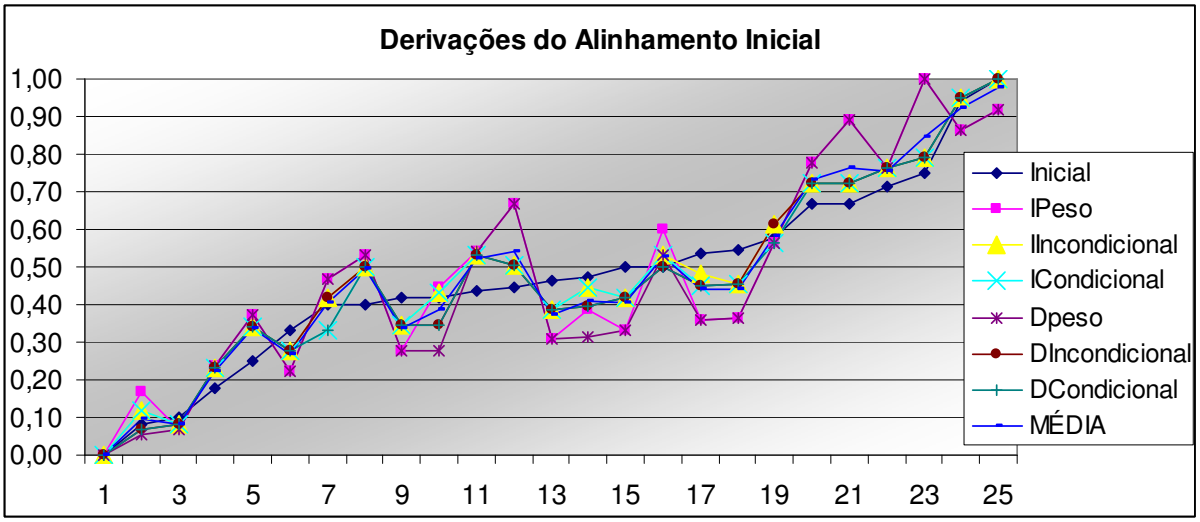

Figura 18 – Gráfico dos Escores das Derivações do Alinhamento Inicial

Como vimos, a melhor opção de algoritmo é o que utiliza a modificação 3 (Comparação Condicional), baseada no alinhamento, e com dependência. A seguir serão mostrados alguns exemplos que comprovam que esta é a melhor opção de algoritmo.

Strings semelhantes como "medico" e "medicina"; "tele" e "telefone" tiveram seu escore incrementado em aproximadamente 0.05.

Já strings que não apresentam semelhança como é o caso de "inicio" e "clinico"; "medico" e "ético", tiveram seu escore decrementado em 0.10.

Isto mostra que o algoritmo escolhido é útil tanto para aumentar o escore de strings semelhantes quanto para diminuir o escore de strings sem semelhança.

O único equívoco observado na geração deste algoritmo foi na comparação de "complemento" e "coma" que são duas strings não semelhantes, mas com os três primeiros caracteres idênticos. Neste caso o escore subiu de 0.40 para 0.50, mas isto não é um grande problema, pois 0.50 não é um escore tão alto a ponto de indicar que estas duas strings são semelhantes.

6. Considerações Finais

Os dois algoritmos que obtiveram melhores resultados para o Alinhamento Local (melhor produção de escores, vista na seção 4.1) foram o Smith Waterman (seção 2.2) e o Jaro (seção 2.5). E como foi visto na seção 4.1.5, a média aritmética destes dois algoritmos produz um resultado muito bom, pois os escores obtidos são quase idênticos aos escores esperados.

Esta junção, no entanto piora o tempo de processamento, pois o tempo para se obter o escore passa a ser o dobro de um tempo que já não é muito bom. Mas como o objetivo da nossa aplicação é apenas inserir *tags xml* em um Esquema, e isto não será feito com uma grande freqüência, o requisito eficiência não é muito importante neste caso.

Quanto aos algoritmos fonéticos, o Soundex não é adequado para a língua portuguesa e seria necessária uma adaptação, bastante complexa, para utilizá-lo. Vale ressaltar que o Soundex até mesmo quando utilizado no inglês pode gerar grandes distorções.

O algoritmo de Covington poderia ser utilizado, pois ele geraria escores melhores do que o gerado pelo de Smith-Waterman, isto assumindo que as strings comparadas possuam fonética correta e erros de grafia. No entanto, a implementação deste algoritmo teria de ser melhorada, sendo necessária uma implementação mais complexa. Mas os ganhos seriam pequenos, visto que os erros de grafia, se ocorrerem, serão pequenos.

Quando for preciso fazer comparações de strings formadas por um único token, propomos a utilização do algoritmo de Alinhamento Inicial que foi o algoritmo que obteve melhores resultados (melhor produção de escores, visto no Capítulo 5) para este tipo de comparação, pois este algoritmo foi desenvolvido especificamente para este caso.

Concluindo, deve-se, utilizar o algoritmo "Smith Waterman + Jaro", para comparação de strings com mais de um token, devido a sua ótima qualidade na produção de escores. Quando se desejar comparar strings com um único token, e com prefixo semelhante, deve-se utilizar o algoritmo Alinhamento Inicial.

Uma sugestão para trabalhos futuros é a melhoria do tempo do algoritmo que pode ser feita de duas formas:

Se for necessária a melhora do tempo, pode-se utilizar o NFA, que é um algoritmo muito eficiente, e possui um alinhamento semi-global. Se adaptado para construir alinhamentos locais, poderia proporcionar o mesmo escore dado pelo algoritmo do Smith-Waterman em um tempo bem menor.

Outra solução para melhorar o tempo seria utilizando a técnica de bit-parallelism que foi desenvolvida para matrizes de programação dinâmica. Esta técnica pode ser adaptada para implementar o Smith Waterman e o Jaro. Como pôde ser observado na Seção 3.3, o tempo para executar a programação dinâmica tradicional cresce drasticamente com a quantidade de erros encontrados na comparação.

Referências Bibliográficas

[BELIAN 05] BELIAN, Rosalie Barreto. Semantic-based Information Integration. Recife, 2005. Tese de Doutorado. Universidade Federal de Pernambuco. 2004.

[BILENKO et al 03] BILENKO, Mikhail; MOONEY Raymond; COHEN, William; RAVIKUMAR Pradeep; and FIENBERG Stephen. Adaptive Name Matching in Information Integration. USA, 2003. Published by the IEEE Computer Society 1094-716/03 SEPTEMBER/OCTOBER 2003.

[CHAPMAN 05] CHAPMAN, Sam. String Similarity Metrics for Information Integration. Sheffield, UK, 2005.
Disponível em: < http://www.dcs.shef.ac.uk/~sam/stringmetrics.html >.
Acesso em: 1 janeiro 2006.

[CRUZ 03] CRUZ, Por Leonardo M.. BLAST Teoria e Prática. Curitiba, 2003. Disponível em: < http://www.bionet.pucpr.br/tutoriais/apostila_blast.pdf >. Acesso em: 26 abril 2005.

[FOSTER 03] FOSTER JR., Blair. Comparative q-gram Analysis of Gene Promoter Regions. New Brunswick, 2003.
Disponível em: < http://www.cs.unb.ca/undergrad/html/200304_Blair.Foster.pdf >. Acesso em: 1 janeiro 2006.

[HALL and DOWLING 80] HALL, Patrick A. V. and DOWLING, Geoff R.. Approximate String Matching. England, Computing Surveys, Vol. 12, No. 4, December 1980.

[KONDRAK 03] KONDRAK, Grzegorz. Phonetic alignment and similarity. Edmonton, Canada, 2003. Disponível em:
< http://www.cs.ualberta.ca/~kondrak/papers/chum.pdf >.
Acesso em: 1 janeiro 2006.

[LÓSCIO 03] LÓSCIO, B. Managing the Evolution of XML-based Mediation Queries. PHD Thesis, Federal University of Pernambuco, Brazil, 2003.

[MESQUITA and MARTOS 91] MESQUITA, Roberto M. and MARTOS, Cloder R.. Gramática Pedagógica. São Paulo, Editora Saraiva, 1991.

[NAVARRO 01] NAVARRO, Gonzalo. A Guided Tour to Approximate String Matching. University of Chile. ACM Computing Surveys, Vol. 33, No. 1, March 2001, pp. 31-88.

[PARREIRAS 04] PARREIRAS, Fernando. Ontologias e integração de sistemas corporativos. Brazil, 2004.
Disponível em: < http://www.webinsider.com.br/ > Acesso em: 1 janeiro 2006.

[RISTAD and YANILOS 96] RISTAD, Eric Sven and YANILOS, Peter N.. Learning String Edit Distance. Research Report CS-TR-532-96, October 1996.

[SMITE and WATERMAN 81] SMITE, T. F. and WATERMAN M. S.. Identification of Common Molecular Subsequences. USA, 1980 Academic Press Inc. (London) Ltd. Reprinted from J . Mol. Biol. (1981) 147, 195-197.

Apêndice A

A.1 Código do Alinhamento Inicial

```
package algoritmos;
//classe utilizada para normalizar as Strings
import util.Filtro;
/**
 * <p>Title: Ontologia Aplicada a Saúde</p>
 * <p>Description: Enriquecimento Léxico de Esquemas para um
Sistema de
 * Integração de Dados Alicado a Saúde</p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: CIn UFPE</p>
 * @author Flavio Melo GONDIM
 * @version 1.0
 */

/**
 * A classe Levenshtein Inicial realiza um alinhamento global
entre duas string
 * através de programação dinâmica (matrizes).<br>
 * É encontrado então o melhor score do alinhamento que serve
tanto para
 * atribuir um rate ao alinahemnto como para mostrar o melhor
alinhamento
 * sobre a forma de string
 */
public class LevenshteinInicial {

 public static final int TIPO_NORMAL = 0;
 public static final int TIPO_SCORE = 1;
 public static final int TIPO_PONTUACAO_INCONDICIONAL = 2;
 public static final int TIPO_PONTUACAO_ALINHAMENTO = 3;

 public static final int DEP_DEPENDENTE = 1;
 public static final int DEP_INDEPENDENTE = 2;

 private static final int pesoMatch = 2;
 private static final int pesoMisMatch = -1;
 private static final int pesoIndel = -1;

 private int tipo;           //tipo de alinhamento
 private int dependencia;    //tipo de dependencia

 private int[][] M;          //matriz onde os pesos dos scores
                            // são armazenados
 private int peso;           //peso da comparação atual
 private int temp;

 private int m;              //tamanho da string 1 (eixo x)
 private int n;              //tamanho da string 2 (eixo y)
```

```
 private int maior;          //maior score obtido na matriz
 private int i_l;            //posição do maior score no eixo x
 private int j_l;            //posição do maior score no eixo y

 private char[] s;           //caracteres da 1º string
 private char[] t;           //caracteres da 2º string

 //1º string com caracteres do alinhamento
 private char[] align_s;
 //2º string com caracteres do alinhamento
 private char[] align_t;

/**
  * Construtor da Classe Levenshtein Inicial.
  * Seta o tipo de Incremento dos 3 primeiros caracteres e
seta a dependencia.
  * @param tipo tipo de incremento
  * @param dependencia dependente ou independente
  */
 public LevenshteinInicial(int tipo, int dependencia) {
  this.tipo = tipo;
  this.dependencia = dependencia;
 }

 /**
  * Este método retorna um float que indica o rate (grau de
similaridade entre duas string)<br>
  * O rate = (maior score)/(soma do tamanho das duas strings),
escala [0,1]
  *
  * @param String uma das strings que será comparada
  * @param String a outra string que será comparada
  * @return float grau de similaridade entre as strings
  */
 public synchronized float comparar(String str1, String str2)
{
  if(str1==null || str2==null)   {
   return 0;
  }

  //lê as strings
  str1 = Filtro.filtrar(str1);
  str2 = Filtro.filtrar(str2);

  //aqui é feito o processamento real
  this.processar(str1,str2);

  //Consider-se que as Strings tem tamanho mínimo de 3
  //caracteres
  float retorno = ( (float)this.maior /
   (Math.max(this.m,3) + Math.max(this.n,3)) );
```

```
//MODIFICAÇÕES PARA O TIPO_PONTUACAO_ALINHAMENTO
if (tipo == TIPO_PONTUACAO_ALINHAMENTO) {
 this.align();
 str1 = this.getAlign(this.align_s);
 str2 = this.getAlign(this.align_t);
}//FIM MODIFICAÇÃO

//MODIFICAÇÕES PARA O TIPO_PONTUACAO_ALINHAMENTO
// e TIPO_PONTUACAO_INCONDICIONAL
if (tipo == TIPO_PONTUACAO_ALINHAMENTO ||
 tipo == TIPO_PONTUACAO_INCONDICIONAL) {
 boolean matchAnterior = false;
 boolean valido = true;
 if (str1.length()>=1 && str2.length()>=1) {
  if (str1.charAt(0)==str2.charAt(0)) {
   retorno += 0.1;
   matchAnterior = true;
  }
  else if (str1.charAt(0)=='-' || str2.charAt(0) =='-') {
   valido = false;
  }
 }
 if (valido && str1.length()>=2 && str2.length()>=2) {
  if (str1.charAt(1) == str2.charAt(1) &&
   (dependencia == DEP_INDEPENDENTE || matchAnterior)) {
   retorno += 0.06;
   matchAnterior = true;
  }
  else if (str1.charAt(1) == '-' || str2.charAt(1) == '-') {
   valido = false;
  }
  else {
   matchAnterior = false;
  }
 }
 if (valido && str1.length()>=3 && str2.length()>=3 &&
  str1.charAt(2)==str2.charAt(2) &&
  (dependencia == DEP_INDEPENDENTE || matchAnterior) ) {
  retorno += 0.04;
 }
 //normalização da escala [0,1.2] para [0,1]
 retorno = retorno/1.2f;
}//FIM MODIFICAÇÃO

 //MODIFICAÇÕES PARA O TIPO_SCORE
else if (tipo == TIPO_SCORE) {
 //o retonno é colocado na escal [0,1]
 retorno = retorno/1.5f;

 if (retorno > 1) {
  retorno = 1;
 }
}
```

```
  if(retorno<0)  {
   retorno=0;
  }

  return retorno;
 }

 /**
  * Este método retorna o alinhamento local entre duas strings
  * remoções e inserções são representadas por (-)
  *
  * @return String alinhamento entre as duas últimas strings
    processadas
  */
 public synchronized String getAlinhamento() {
  this.align();
  String retorno = this.getAlign(this.align_s);
  retorno = retorno + "\n" +this.getAlign(this.align_t);
  return retorno;
 }

 /**
  * Este método executa o processamento comum aos métodos
  * comparar e alinhar
  *
  * @param String uma das strings que será comparada
  * @param String a outra string que será comparada
  */
 private synchronized void processar(String str1, String str2)
{

  //inverte a posicao das strings
  char[] temp = str1.toCharArray();
  this.s = new char[temp.length + 1];
  for (int i = 0; i < temp.length; i++) {
   s[i + 1] = temp[i];
  }

  temp = str2.toCharArray();
  this.t = new char[temp.length + 1];
  for (int i = 0; i < temp.length; i++) {
   t[i + 1] = temp[i];
  }

  //Faz o alinhamento
  this.montarMariz();
 }
```

```
 /**
  * Método que constroi a matriz de alinhamento utilizando
para isso a programação dinamica
  */
 //ALINHAMENTO
 public void montarMariz()  {
  this.M = new int[s.length][t.length];
  this.m = s.length-1;
  this.n = t.length-1;
  this.maior=0;
  this.i_l=0;
  this.j_l=0;
  for(int i=0; i<=m ; i++) {
   M[i][0] = i*(pesoIndel);
  }
  for(int j=0; j<=n; j++) {
   M[0][j] = j*(pesoIndel);
  }
  int pesoMatchAtual = 0;

  for(int i=1; i<=m ; i++) {
   for(int j=1; j<=n ; j++) {
    pesoMatchAtual = pesoMatch;
    //MODIFICAÇÕES PARA O TIPO_SCORE
    if(tipo == TIPO_SCORE) {
     if(i<=3 && j<=3 && i==j) {
      if (i == 1) {
       //pesoMatch + pesoMatch + pesoMatch + pesoMatch;
       pesoMatchAtual = 5;
      }
      else if (i == 2) {
       //pesoMatch + pesoMatch + pesoMatch;
       pesoMatchAtual = 4;
      }
      else {
       //pesoMatch + pesoMatch;
       pesoMatchAtual = 3;
      }
      if (dependencia == DEP_DEPENDENTE) {
       if (i==2) {
        if (M[i - 1][j - 1] != 5) {
         pesoMatchAtual = pesoMatch;
        }
       }
       if (i==3) {
        if (M[i - 1][j - 1] != (5+4) ) {
         pesoMatchAtual = pesoMatch;
        }
       }
      }
     }
    }//FIM MODIFICAÇÃO
```

```
    if ( s[i] == t[j] ) {
     peso = pesoMatchAtual;
    }
    else {
     peso = pesoMisMatch;
    }

    temp = Math.max(M[i][j-1] +pesoIndel, M[i-1][j] +
pesoIndel);
    M[i][j] = Math.max(M[i-1][j-1] + peso, temp);

    if(M[i][j] > maior) {
     // este valor será o maior
     maior = M[i][j];
     //seta a posição do maior
     this.i_l=i;
     this.j_l=j;
    }
   }
  }
 }


 /**
  * Este método transforma charArrays em strings invertendo a
posição
  * dos caracteres do array
  *
  * @param char[] a ser transformado
  * @return String obtida
  */

 private String getAlign(char align[]) {
  String ali = "";
  for (int i = align.length - 1; i >= 0; i--) {
   if (align[i] != ' ') {
    ali = ali + String.valueOf(align[i]);
   }
  }
  return ali;
 }

 /**
  * Este método seleciona o melhor caminho a ser percorrido na
matriz gerando charArrays que são as strings originais
enriquecidas com (-) referentes a inserções e remoções
  */
 private void align()  {
  int i = this.i_l;
  int j = this.j_l;
  int len = 0;
  this.align_s = new char[m + n + 3];
  this.align_t = new char[m + n + 3];
```

```
for (int z = 0; z < (m + n + 3); z++) {
 align_s[z] = ' ';
 align_t[z] = ' ';
}

//maior distancia entre o maior score e o final da matriz
int mn = Math.max(this.m - this.i_l, this.n - this.j_l);

//enquanto não se chega na posição do maior score são
//colocados hifens  ou o caractere desta posição
while (mn > 0) {
 len++;
 if (mn + i_l <= m) {
  align_s[len] = s[mn + i_l];
 }
 else {
  align_s[len] = '-';
 }
 if (mn + j_l <= n) {
  align_t[len] = t[mn + j_l];
 }
 else {
  align_t[len] = '-';
 }
 mn--;
}

//vesifica-se se ocorreu (Match ou Mismatch) ou indel
while (i > 0 && j > 0) {
 if ( s[i] == t[j] ) {
  peso = pesoMatch;
 }
 else {
  peso = pesoMisMatch;
 }
 //Match ou Mismatch
 if (M[i][j] == M[i - 1][j - 1] + peso) {
  len++;
  align_s[len] = s[i];
  align_t[len] = t[j];
  i--;
  j--;
 }
 //indels
 else if (M[i][j] == M[i - 1][j] + pesoIndel) {
  len++;
  align_s[len] = s[i];
  align_t[len] = '-';
  i--;
 }
```

```
    //indels
    else if (M[i][j] == M[i][j - 1] + pesoIndel) {
     len++;
     align_s[len] = '-';
     align_t[len] = t[j];
     j--;
    }
   }

   //os carateres que ficaram desalinhados no começo da string
   //são alinhados com hifens
   while (i > 0 || j > 0) {
    len++;
    if (i > 0) {
     align_s[len] = s[i];
    }
    else {
     align_s[len] = '-';
    }
    if (j > 0) {
     align_t[len] = t[j];
    }
    else {
     align_t[len] = '-';
    }
    i--;
    j--;
   }
  }
}//fim da classe
```