Uma abordagem uniforme para problemas de comparação de seqüências

João Carlos Setubal e J. Rodolfo Suárez de Oliveira

Relatório Técnico IC–97-16

Outubro de 1997

Uma abordagem uniforme para problemas de comparação de seqüências

João Carlos Setubal*e J. Rodolfo Suárez de Oliveira†

Sumário

Apresentamos descrições sucintas de 7 problemas de comparação de seqüências de caracteres, que podem ser resolvidos por programação dinâmica. Alguns desses problemas são particularmente importantes na área de biologia computacional. Com base nessas descrições apresentamos uma formulação geral que captura de maneira uniforme todas as formulações específicas, permitindo assim a implementação de um único programa que resolve todos os problemas. Essa implementação foi realizada e se encontra disponível através da WWW. Descrevemos também um algoritmo associado simples que permite contar o número de soluções ótimas de um dado problema; tal recurso é particularmente útil em aplicações de biologia computacional.

Abstract

We present succinct descriptions of 7 problems on character sequence comparison solvable by dynamic programming. Some of these problems are particularly important in computational biology. Based on these descriptions we present a general formulation for all 7 problems, allowing thus the implementation of a simple program that is able to solve each of those problems, depending on the user's choice. This implementation was written and the corresponding program is available through the WWW. We also describe a simple associated algorithm that efficiently counts all optimal solutions of a given problem. Such a feature is useful in computational biology applications.

1 Introdução

Este trabalho trata de problemas envolvendo comparações de pares de seqüências de caracteres. Uma seqüência de caracteres, ou simplesmente uma seqüência, é uma sucessão finita de símbolos pertencentes a um certo conjunto de tamanho finito (o alfabeto). Assim, se $\Sigma = \{\texttt{A, C, G, T}\}$ é um alfabeto, `ACCG` e `CGAAT` são seqüências.

O objetivo do trabalho foi produzir um programa com interface pela WWW que resolve diversos problemas de comparação de pares de seqüências. Foram identificados sete problemas deste tipo, todos eles passíveis de solução pela técnica de programação dinâmica. Uma formulação uniforme para o tratamento desses problemas foi desenvolvida, aproveitando a

*Instituto de Computação, Universidade Estadual de Campinas, CP 6176, Campinas, SP, 13081–970. Pesquisa desenvolvida com suporte financeiro parcial do CNPq — Conselho Nacional de Desenvolvimento Científico e Tecnológico, sob projeto 351056/95-5.

†bolsista de iniciação científica da FAPESP, processo número 96/02858-5.

semelhança entre eles. Com base nesta formulação foi produzido um programa conforme o objetivo.

Na seção 2 apresentamos os problemas de comparação de seqüências enfocados e suas formulações. Na seção 3 mostramos a formulação uniforme para todos os problemas. Esta formulação foi implementada, incluindo a possibilidade de recuperação ou contagem de todas as soluções ótimas de um problema. Na seção 4 descrevemos o algoritmo de contagem de soluções e na seção 5 damos detalhes da implementação e da interface. Finalmente, a seção 6 traz a conclusão deste artigo.

2 Comparação de seqüências e programação dinâmica

Nesta seção apresentamos os problemas enfocados pelo trabalho. Todos eles são problemas bem conhecidos na literatura [1, 2, 3, 4], mas desconhecemos uma fonte que os descreva da maneira sistemática e conjunta dada abaixo.

2.1 Alinhamento Global

O problema do alinhamento global de duas seqüências consiste em se determinar quão parecidas são essas seqüências. Formalmente, dadas duas seqüências s e t sobre um alfabeto Σ, queremos posicionar s sobre t de modo a formar colunas com dois caracteres (x, y) em cada coluna (este posicionamento é o *alinhamento* de s e t). Vamos permitir que um caracter de s ou um de t seja alinhado com um caracter especial - $\notin \Sigma$ chamado de *espaço*. Tendo um alinhamento, a cada coluna i devemos associar uma pontuação $p_i(x, y)$. Chamamos p de *sistema de pontuação*. O *valor* de um alinhamento é dado por $\sum_i p_i(x, y)$.

Dado um sistema de pontuação, o problema do **alinhamento global** (**AG**) consiste em se determinar o alinhamento de valor máximo. Por exemplo, se $s =$ `ACGTTCGAAC` e $t =$ `ACTTCGAAG`, e se p é dado por

$$p(x, y) = \begin{cases} 1 & \text{se } x = y \\ -1 & \text{se } x \neq y, \;\; x, y \in \Sigma \\ -2 & \text{se } x = \text{- ou } y = \text{-}, \end{cases} \tag{1}$$

um alinhamento global de valor máximo é o seguinte:

```
ACGTTCGAAC
ACTT-CGAAG
```

Seu valor é $7 \times (+1) + 2 \times (-1) + 1 \times (-2) = 3$. O sistema de pontuação utilizado recompensa colunas e caracteres iguais, e penaliza colunas com caracteres diferentes, principalmente se um deles for um espaço. Este é um problema fundamental em diversos contextos, particularmente para comparação de seqüências de DNA ou de proteínas [4].

Para resolver esse problema utiliza-se a conhecida técnica de programação dinâmica [1]. Nesta técnica chega-se ao valor desejado através do preenchimento de uma matriz M de dimensão $n + 1$ por $m + 1$ onde $n = |s|$ e $m = |t|$. O alinhamento de valor máximo (ou

alinhamento ótimo) é dado pelo valor de $M[n,m]$, sendo que os valores intermediários são dados pela seguinte recorrência:

$$M[i,j] = \max \begin{cases} M[i-1,j-1] + p(x,y) \\ M[i-1,j] + v_3 \\ M[i,j-1] + v_4 \end{cases} \tag{2}$$

onde

$$p(x_i, y_j) = \begin{cases} v_1 \text{ se } x_i = y_j \\ v_2 \text{ se } x_i \neq y_j. \end{cases}$$

As constantes simbólicas v_k têm o seguinte significado: v_1 é a pontuação dada para caracteres iguais, v_2 é a pontuação dada para caracteres diferentes e v_3 e v_4 são pontuações dadas no caso de haver um espaço na coluna. Para o sistema de pontuação (1) acima esses valores são $v_1 = 1, v_2 = -1, v_3 = v_4 = -2$. (A razão de se definir duas constantes diferentes com o mesmo valor ficará clara na seção 3.)

A base da recorrência é dada pelos valores da primeira linha e primeira coluna:

$$M[i,0] = iv_3, \text{ para } 0 \leq i \leq n, \text{ e } M[0,j] = jv_4, \text{ para } 0 \leq j \leq m.$$

O algoritmo que acha o valor do alinhamento ótimo deve simplesmente preencher os valores da matriz obedecendo às regras da recorrência acima. A existência da matriz evita cálculos supérfluos (e esta é a essência da técnica de programação dinâmica), resultando numa complexidade de $\Theta(nm)$.

A obtenção do alinhamento propriamente dito se faz percorrendo o caminho que leva de $M[n,m]$ a $M[0,0]$, seguindo as células de M que efetivamente contribuíram para o valor de $M[n,m]$. Note que pode haver mais de um caminho, o que significa que pode haver mais de uma solução ótima para o problema. Na seção 4 é descrito um algoritmo que conta o número de soluções ótimas.

2.2 Outros problemas de comparação de seqüências

Nesta seção descrevemos os outros seis problemas de comparação de seqüências abordados.

Alinhamento Prefixo-Sufixo (APS): neste problema estamos interessados em conseguir um alinhamento entre o prefixo de uma seqüência e o sufixo de outra. Para descrever este problema precisamos definir formalmente as noções de prefixo e sufixo.

Se w é uma seqüência formada pela concatenação das seqüências u, x e v (ou seja, $w = uxv$), dizemos que u é um *prefixo* de w, x é um *fator* de w e v é um *sufixo* de w (u e v são também fatores de w).

Este problema é importante para a montagem de fragmentos de DNA [4]. Por exemplo: dadas as seqüências $s =$ `CTTGGATTCTCGG` e $t =$ `CAGCGTGGT` gostaríamos de obter o seguinte alinhamento:

```
---CTTGGATTCTCGG
CAGCGTGG-T------
```

Note que para obter tal alinhamento não devemos penalizar espaços nas extremidades. Isto é facilmente conseguido com uma inicialização conveniente da matriz de programação dinâmica M, e o valor do alinhamento ótimo é dado pela célula de maior valor da última linha de M. Os valores intermediários são dados pela mesma recorrência já vista, (2). A inicialização deve ser assim:

$$M[i,0] = iv_3, \text{ para } 0 \leq i \leq n, \text{ e } M[0,j] = 0, \text{ para } 0 \leq j \leq m.$$

Com essa inicialização da matriz obteremos o alinhamento ótimo entre um prefixo de s e um sufixo de t; se estivéssemos interessados no alinhamento ótimo entre um prefixo de t e um sufixo de s bastaria inicializar a primeira coluna da matriz com zeros e a primeira linha com valores jv_4. As constantes v_k têm o mesmo significado que no problema **AG**.

Alinhamento Local (AL): este problema é outra variação do problema **AG**, sendo o objetivo obter o melhor alinhamento de fatores de uma seqüência com fatores da outra seqüência. Por exemplo: dadas as seqüências $s =$ `TTCAGCACTTGGATTCTCGG` e $t =$ `AGCGTGG`, um possível alinhamento seria:

```
TTCAGCA-CTTGGATTCTCGG
------AGCGTGG--------
```

Este problema é importante na comparação de seqüências de proteínas, pois é comum que duas proteínas possuam certos trechos muito parecidos em meio a outros que são muito diferentes [4].

O valor do alinhamento ótimo é dado pelo maior valor presente em M após seu preenchimento. Isso significa que pode haver mais de um elemento na matriz com esse valor máximo, o que indicaria a presença de mais de um par de fatores cujo alinhamento é ótimo. Os valores da matriz são dados pela seguinte recorrência:

$$M[i,j] = \max \begin{cases} M[i-1,j-1] + p(x,y) \\ M[i-1,j] + v_3 \\ M[i,j-1] + v_4 \\ 0 \end{cases} \tag{3}$$

onde

$$p(x_i, y_j) = \begin{cases} v_1 \text{ se } x_i = y_j \\ v_2 \text{ se } x_i \neq y_j. \end{cases}$$

As constantes v_k têm o mesmo significado que no problema **AG** e a primeira coluna e a primeira linha devem ser inicializadas com zero. Note que embora tenha sido dada a formulação específica (3) para este problema, a única diferença dela para (2) é o acréscimo de um zero.

Subseqüência Comum Máxima (SUBMAX): uma subseqüência de uma seqüência dada é um subconjunto ordenado de elementos desta seqüência, não necessariamente consecutivos. Exemplo: `BCDB` é uma subseqüência de `ABCBDAB`. Dadas duas seqüências s e t, uma subseqüência comum é uma seqüência que é ao mesmo tempo subseqüência de s e de t. Exemplo: `BCA` é subseqüência comum às seqüências `ABCBDAB` e `BDCABA`. Uma Subseqüência Comum Máxima é aquela de maior comprimento entre todas as possíveis subseqüências comuns. Neste problema estamos interessados em encontrar a subseqüência comum máxima de duas seqüências dadas. O tamanho desta subseqüência é dado pelo valor de $M[n,m]$, sendo que os valores intermediários são dados pela seguinte recorrência:

$$M[i,j] = \begin{cases} M[i-1,j-1] + v_1 \text{ se } x_i = y_j \\ \max \left\{ \begin{array}{l} M[i-1,j] + v_3 \\ M[i,j-1] + v_4 \end{array} \right\} \text{ se } x_i \neq y_j \end{cases} \quad (4)$$

onde $v_1 = 1$, $v_3 = v_4 = 0$. A inicialização é dada por $M[i,0] = 0$ para $0 \leq i \leq n$, e $M[0,j] = 0$ para $0 \leq j \leq m$.

Os valores das constantes v_1, v_3 e v_4 não estão vinculados a nenhum sistema de pontuação, ou seja, terão sempre os valores acima para este problema. O motivo de se usar esta notação é que estas constantes são usadas para facilitar o tratamento homogêneo apresentado na Seção 3.

Superseqüência Comum Mínima (SUPMIN): se w é uma subseqüência de u, dizemos que u é uma superseqüência de w. Exemplo: `ABCBDAB` é uma superseqüência de `BCDB`. Dadas duas seqüências s e t, uma *superseqüência comum* é uma seqüência que é ao mesmo tempo superseqüência de s e de t. Exemplo: dadas as duas seqüências: `ABC` e `DCABEF`, uma superseqüência comum às duas é: `AAAAADCABECFDDD`. Uma Superseqüência Comum Mínima é aquela de menor comprimento entre todas as possíveis superseqüências comuns. Neste problema estamos interessados em encontrar a superseqüência comum mínima de duas seqüências dadas. O tamanho desta superseqüência é dado pelo valor de $M[n,m]$, sendo que os valores intermediários são dados pela seguinte recorrência:

$$M[i,j] = \begin{cases} M[i-1,j-1] + v_1 \text{ se } x_i = y_j \\ \min \left\{ \begin{array}{l} M[i-1,j] + v_3 \\ M[i,j-1] + v_4 \end{array} \right\} \text{ se } x_i \neq y_j \end{cases} \quad (5)$$

onde $v_1 = v_3 = v_4 = 1$. A inicialização é dada por $M[i,0] = i$ para $0 \leq i \leq n$, e $M[0,j] = j$ para $0 \leq j \leq m$. Novamente as constantes simbólicas foram utilizadas para facilitar o tratamento uniforme a ser visto abaixo.

Subcadeia Comum Máxima (SUBCAD): no problema **SUBMAX** foi enfatizado o fato de que uma subseqüência de uma seqüência não precisava ter seus elementos necessariamente consecutivos. Se impusermos a necessidade dos elementos serem consecutivos, passamos a chamar a subseqüência de subcadeia e os termos subcadeia comum e Subcadeia Comum Máxima são definidos de maneira análoga à que foi feita no problema **SUBMAX**. Neste problema estamos interessados em encontrar a subcadeia comum máxima de duas

seqüências s e t dadas. Exemplo: dadas as duas seqüências: `AEFJBCDK` e `GHBCDIFJ` a subcadeia comum máxima é `BCD`. Vale observar que este problema pode ser resolvido de forma mais eficiente, sem o uso da programação dinâmica (usando árvore de sufixos [2]), mas foi incluído por se tratar de um problema interessante envolvendo seqüências. O tamanho da subcadeia comum máxima é dado pelo maior valor presente em M após seu preenchimento. Os valores da matriz são dados pela seguinte recorrência:

$$M[i,j] = \begin{cases} M[i-1,j-1] + v_1 \text{ se } x_i = y_j \\ 0 \text{ se } x_i \neq y_j \end{cases} \tag{6}$$

sendo que $v_1 = 1$. A inicialização é dada por $M[i,0] = 0$ para $0 \leq i \leq n$, e $M[0,j] = 0$ para $0 \leq j \leq m$. Mais uma vez usamos uma constante simbólica v_1 tendo em vista o tratamento uniforme a ser visto na Seção 3.

Distância de Edição (DE): dadas duas seqüências s e t, o objetivo é efetuar certas operações sobre s para transformá-la em t. Estas operações são:

1. Substituição de um elemento por outro numa posição qualquer da seqüência.
2. Remoção de um elemento da seqüência.
3. Inserção de um elemento qualquer numa posição da seqüência.

Cada operação tem um custo não-negativo associado. O custo total é a soma dos custos de todas as operações que transformam s em t. Dá-se o nome de *Distância de Edição* ao menor custo total possível. Este valor é dado pelo valor de $M[n,m]$, sendo que os valores intermediários são dados pela seguinte recorrência:

$$M[i,j] = \min \begin{cases} M[i-1,j-1] + p(x,y) \\ M[i-1,j] + v_3 \\ M[i,j-1] + v_4 \end{cases} \tag{7}$$

onde

$$p(x_i, y_j) = \begin{cases} v_1 \text{ se } x_i = y_j \\ v_2 \text{ se } x_i \neq y_j, \end{cases}$$

sendo que $v_1 = 0$, $v_2 = v_3 = v_4 = 1$. A inicialização é dada por $M[i,0] = i$ para $0 \leq i \leq n$, e $M[0,j] = j$ para $0 \leq j \leq m$. As constantes têm o seguinte significado. v_1: custo de uma não-operação (substituição de elementos iguais); v_2: custo para substituir um elemento por outro; v_3: custo para remover um elemento x_i da seqüência s; v_4: custo para inserir um elemento y_j na seqüência s. Com os valores definidos acima estaremos minimizando o número total de operações sem fazer distinção entre elas. Para este problema, assim como ocorre nos problemas **AG**, **APS** e **AL**, pode-se definir um outro sistema de pontuação.

3 Tratamento uniforme

Nesta seção mostramos como todos os problemas apresentados na seção anterior podem ser resolvidos por uma mesma abordagem uniforme, o que constitui uma das principais contribuições deste trabalho.

As descrições dos problemas da seção anterior deixam claro que existe uma semelhança muito grande entre as formulações dos diversos problemas. As seguintes diferenças podem ser observadas: 1) em alguns problemas se usa a função max e em outros a função min; 2) a situação em que estas funções devem ser usadas difere: no caso dos problemas **SUBMAX** e **SUPMIN** essas funções são usadas apenas se $x_i = y_j$, enquanto que nos demais esta condição não é verificada; 3) a forma como a matriz é inicializada e os valores das constantes v_k também diferem de problema para problema; 4) o valor ótimo procurado está em diferentes células da matriz depois de preenchida. Apesar dessas diferenças, é possível encontrar uma formulação geral para todos os problemas.

Para obter uma formulação geral para os problemas **SUBMAX** e **SUPMIN**, por exemplo, basta trocar a função min por max na matriz M de **SUPMIN**, inicializá-la com valores negativos e mudar os sinais das constantes v_k. Analogamente, uma mesma formulação para os problemas **AG**, **APS** e **DE** é obtida trocando min por max em **DE**. Fazendo uma análise cuidadosa de todos os problemas, chega-se à seguinte formulação geral para a resolução dos 7 problemas:

$$M[i,j] = \max \begin{cases} M[i-1,j-1] + p(x,y) \\ M[i-1,j] + v_3 \\ M[i,j-1] + v_4 \\ v_5 \end{cases} \tag{8}$$

onde

$$p(x_i, y_j) = \begin{cases} v_1 \text{ se } x_i = y_j \\ v_2 \text{ se } x_i \neq y_j. \end{cases}$$

Note que as constantes v_k e a inicialização de M continuam dependendo do problema que está sendo resolvido. A seguir demonstramos que esta formulação está correta.

Teorema 3.1 *A formulação acima, com valores adequados para as constantes v_k e inicializações específicas, corretamente determina a solução de cada um dos problemas em consideração.*

Demonstração: Para resolver os problemas **AG** e **APS** defina $v_5 = -\infty$ e para o problema **AL** defina $v_5 = 0$. É imediato ver que a formulação geral (8) acima "simula" a formulação específica (2), que vale tanto para **AG** quanto para **APS**. Para simular o problema **DE** faça $v_5 = -\infty$ e tome v_2, v_3, v_4 e a inicialização da primeira coluna e da primeira linha com valores negativos para que a função min possa ser substituída pela função max. Para o problema **SUBMAX** defina $v_1 = 1$, $v_3 = v_4 = 0$ e $v_2 = v_5 = -\infty$. Dessa forma, se ocorrer que $x_i \neq y_j$, o valor da matriz sairá de $M[i-1,j] + v_3$ ou $M[i,j-1] + v_4$, concordando com a definição (4) de M em **SUBMAX**. Por outro lado, quando ocorrer

Tabela 1: Valores das constantes e forma de inicialização para cada problema.

problema	v_1	v_2	v_3	v_4	v_5	$M[i,0]$	$M[0,j]$
AG	1	-1	-2	-2	$-\infty$	iv_3	jv_3
APS	1	-1	-2	-2	$-\infty$	iv_3	0
AL	1	-1	-2	-2	0	0	0
SUBMAX	1	$-\infty$	0	0	$-\infty$	0	0
SUPMIN	-1	$-\infty$	-1	-1	$-\infty$	$-i$	$-j$
SUBCAD	1	$-\infty$	$-\infty$	$-\infty$	0	0	0
DE	0	-1	-1	-1	$-\infty$	$-i$	$-j$

$x_i = y_j$ a escolha sairá do máximo entre valores dependentes das três células adjacentes à célula com valor sendo calculado. Isso está de acordo com a definição (4) da matriz M de **SUBMAX**, apesar desta só considerar o valor da célula da diagonal na situação em que $x_i = y_j$. A justificativa para isso é o fato do valor da célula de cima e da esquerda diferirem da célula da diagonal em no máximo uma unidade, ou seja, sempre teremos $M[i-1,j] \leq M[i-1,j-1]+1$ e $M[i,j-1] \leq M[i-1,j-1]+1$. Isso ocorre porque a diferença de valores entre células vizinhas é no máximo 1, já que o valor de uma célula é o valor de uma vizinha acrescentando-se $v_3 = v_4 = 0$ ou $v_1 = 1$. Este mesmo raciocínio vale para o problema **SUPMIN**, realizando a devida troca de sinais. Para o problema **SUBCAD** defina $v_5 = 0$, $v_2 = v_3 = v_4 = -\infty$ e $v_1 = 1$ e uma simples inspeção mostra que assim obtemos a formulação específica (6) desse problema. □

Resumimos na Tabela 1 os valores das constantes v_k e a forma de inicialização da matriz M para cada problema.

4 Recuperação das soluções a partir da matriz de programação dinâmica

A partir da matriz M a recuperação da solução (por exemplo, o alinhamento ótimo em **AG**), se faz percorrendo os elementos da matriz segundo a fórmula de recorrência que gerou os valores das suas entradas. Para os problemas **AG**, **SUBMAX**, **SUPMIN** e **DE** a posição inicial para se começar a recuperação da solução é $M[n,m]$. Para o problema **APS** a recuperação deve ser iniciada a partir da célula de valor máximo da última coluna da matriz. Os problemas **AL** e **SUBCAD** têm suas soluções recuperadas iniciando-se a partir das células de maior valor da matriz. A posição final da recuperação em geral é $M[0,0]$, mas nos problemas **AL** e **SUBCAD** é a primeira célula encontrada com valor zero.

O algoritmo utilizado para a recuperação da solução ótima (válido para todos os 7 problemas) consiste na realização de uma busca em profundidade no grafo definido pela construção dos valores da matriz M. Nesse grafo cada célula da matriz que efetivamente contribuiu para o valor ótimo é um vértice; e existe aresta orientada do vértice $M[i,j]$ para o vértice $M[i-1,j-1]$ (ou $M[i-1,j]$ ou $M[i,j-1]$) se esta última célula contribuiu para o valor da primeira. Uma solução ótima é um caminho orientado da célula inicial (que varia

de problema para problema) para a célula final (idem). A realização de uma busca no grafo descrito para encontrar um caminho terá que, no pior caso, percorrer todas as células da matriz. Assim sendo seu tempo de execução é $O(nm)$.

O grafo descrito acima é acíclico. Isso permite modificar a busca em profundidade para que ela recupere *todas* as soluções ótimas de um dado problema ao invés de apenas uma. A modificação é a seguinte: Ao fazer o retrocesso, a busca desmarca todos os vértices visitados; isto permite que a partir de ramos (isto é trechos de um caminho) ainda não visitados outros ramos já visitados venham a ser visitados novamente.

Uma justificativa para capacitar o programa a recuperar todas as soluções ótimas vem da aplicação dos problemas aqui discutidos em biologia computacional. Em biologia molecular é muito comum a situação em que se sabe de antemão que um determinado problema tem apenas uma solução (por exemplo, o "alinhamento correto"). Porém, devido a erros na leitura de uma seqüência biológica como o DNA, podem existir várias soluções ótimas para um problema de alinhamento, sendo que apenas uma delas é a correta[1]. O biólogo molecular precisa então inspecionar todas as soluções para decidir (baseado em informações adicionais) qual delas é a correta.

Por outro lado, é também comum a situação em que o número de soluções ótimas é enorme. Isso ocorre, por exemplo, quando comparamos duas seqüências de DNA que tem pouco ou nada a ver uma com a outra. Nesse caso provavelmente o número de soluções ótimas é muito grande, e nenhuma delas é "boa". Assim sendo, é desejável incorporar ao programa a capacidade de contar eficientemente o *número* de soluções ótimas sem ter que recuperá-las uma por uma (o que levaria $O(Snm)$, onde S é o número de soluções ótimas). Tal número, juntamente com o valor ótimo da matriz, são dois indicadores importantes da qualidade da resposta. A seguir descrevemos um algoritmo que conta eficientemente o número de soluções ótimas.

O algoritmo é uma busca em largura modificada no mesmo grafo descrito acima. A modificação consiste em levar em conta o grau de um vértice no momento de inseri-lo na fila utilizada numa busca em largura. O grau de um vértice v é definido aqui como o número de arestas da forma (u, v) (ou seja, quantos vértices tem v como vizinho). Cada vértice só é inserido se o número de vezes que foi visitado (incluindo a visita corrente) é igual a seu grau. Um pseudo-código para o algoritmo é mostrado na Figura 1. Um argumento indutivo simples mostra que esse algoritmo está correto. Seu tempo de execução é proporcional ao número de células da matriz no pior caso, sendo portanto o mesmo do algoritmo que determina o valor da solução, isto é, $O(nm)$.

5 Implementação e interface

O programa que implementa os algoritmos descritos neste trabalho foi escrito em C++ usando o compilador GCC (GNU software) em ambiente Unix. O número de linhas do núcleo do programa (referente à inicialização das constantes v_i, inicialização e cálculo da

[1] A rigor, nesse caso devem ser incluídas também as soluções *sub-ótimas,* pois os erros de leitura mencionados podem fazer com que a solução correta não esteja entre as soluções ótimas. Não seria difícil adaptar o programa aqui descrito para recuperar também todas as soluções que estejam a um certo ϵ do valor ótimo.

```
Contadores (todos inicialmente zerados):
    c_v: número de caminhos que passam pelo vértice v
    δ_v: grau do vértice v
    x_v: número de visitas feitas ao vértice v
Num primeiro percurso do grafo, calculamos δ_v
O percurso se inicia na célula t (que depende do problema)
FilaInsere(fila, t)
enquanto fila não está vazia faça
    u ← FilaRemove(fila)
    Visita u
    para todo vizinho v de u faça
        δ_v ← δ_v + 1
        se v ainda não foi visitado então
            FilaInsere(fila, v)
No segundo percurso, contamos todos os caminhos
c_t ← 1
FilaInsere(fila, t)
enquanto fila não está vazia faça
    u ← FilaRemove(fila)
    para todo vizinho v de u faça
        x_v ← x_v + 1
        c_v ← c_v + c_u
        se δ_v = x_v então
            FilaInsere(fila, v)
devolve c_s onde s é o vértice final de todos os caminhos.
```

Figura 1: Algoritmo para contagem do número de soluções

matriz) é cerca de 150, o que mostra a economia do tratamento uniforme aqui proposto. A interface tem duas versões, uma para a execução através de linha de comando e outra através da WWW, utilizando algum *browser*. Esta última foi implementada usando o padrão CGI (Common Gateway Interface).

Os campos disponíveis para preenchimento na tela pelo usuário são os seguintes:

- as duas seqüências de entrada (máximo de 80 caracteres)
- problema a ser resolvido
- sistema de pontuação a ser usado nos problemas 1, 2, 3 ou 7 (defaults são fornecidos)
- forma de apresentação da solução:
  - apenas o número de soluções ótimas
  - apenas a matriz de programação dinâmica
  - todas as soluções ótimas

- a matriz de programação dinâmica e todas as soluções ótimas
- visualização gráfica da matriz de programação dinâmica

O programa está disponível em

`http://www.dcc.unicamp.br:8368/seq_problems.html`

6 Conclusão

Neste trabalho mostramos como é possível criar um mesmo algoritmo que resolve 7 problemas distintos de comparação de seqüências, partindo do fato de que todos eles podem ser resolvidos por programação dinâmica. Esse tratamento uniforme permitiu a criação de um programa elegante e conciso, disponível via WWW, que resolve os problemas descritos.

As extensões previstas do trabalho são as seguintes:

- Permitir leitura e gravação em arquivos (o que também permitirá a resolução de problemas de tamanho grande).
- Incluir outros problemas da literatura que se encaixem nesta formulação geral.

7 Agradecimentos

Gostaríamos de agradecer aos membros do grupo de pesquisas em biologia computacional do IC/UNICAMP por sugestões que melhoraram aspectos deste trabalho, e em particular a João Meidanis, que nos ajudou a tornar ainda mais compacta a formulação (8).

O trabalho foi possível graças a uma bolsa de iniciação científica concedida pela FAPESP ao segundo autor.

Referências

[1] T. H. Cormen, C. E. Leiserson and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[2] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, 1997.

[3] Udi Manber. *Introduction to Algorithms*. Addison-Wesley, 1989.

[4] J. Setubal e J. Meidanis. *Introduction to Computational Molecular Biology*. PWS, 1997.