Alinhamento de duas cadeias usando um transdutor adaptativo

(Novembro 2008)

J. Kinoshita e R. L. A. Rocha

***Resumo*— O algoritmo clássico que alinha duas cadeias (strings) s1 e s2 de tamanho m e n é seqüencial e da ordem de m x n. Nesse artigo propomos o uso de um transdutor adaptativo para resolver o mesmo problema que é baseado no algoritmo clássico, mas pode ser paralelizado porque funções adaptativas podem ser disparadas assim que prontas.**

***Palavras chave*— Autômatos adaptativos (*adaptative automata*), alinhamento de cadeias (*string alignment*), biologia computacional (*computational biology*).**

I. Introdução

Um algoritmo bastante utilizado ao se trabalhar com strings é o de se buscar uma string em um texto (ou uma substring em uma string). Um editor de texto simples deve prover ferramentas para se resolver esse problema e diversos algoritmos já haviam sido propostos na década de 70, porém devido ao aumento de textos em formatos eletrônicos que ocorrem na Web ou que representam seqüências de genes, surgiu um grande interesse em se resolver o problema de se buscar uma string em um texto sendo que a string poderia estar corrompida. Esse tipo de problema ocorre no plágio de uma música: é possível se identificar uma string (uma seqüência de notas musicais) mesmo que haja algumas distorções. Para se resolver esse tipo de problema, houve a necessidade de se usar uma função de distância entre duas strings e dentre várias possíveis, a função de distância de edição proposta por Levenshtein em 1965 ganhou notoriedade. A seguir explicamos a função de distância de edição, o algoritmo clássico que resolve essa função e nossa proposta de um algoritmo baseado em um dispositivo adaptativo.

II. Função Distância de Edição

Usando um editor podemos transformar uma string (a primeira string) em outra (a segunda string) realizando operações sobre cada caractere da primeira string. As operações sobre cada caractere podem ser apenas as 4 operações:

R: *replace* - troca um caractere por outro
I: *insert* - insere um caractere na segunda string. Como ênfase: não existem operações I sobre caracteres da primeira string.
D: *delete* - apaga o caractere da primeira string.
M: *match* - passa o caractere de entrada para a cadeia de saída.

Um exemplo de se editar "gato" e obter "gente" usando essas operações é a seguinte:

```
DDDDIIIII
gato
    gente
```

que significa: *delete* (D) 'g', *delete* 'a' , ou seja, apague todos os caracteres da primeira string e insira (I) todos os caracteres da segunda string.

Segundo Gusfield [1]:

A distância de edição entre duas strings é definida como o menor número de operações de edição: I, R, e D necessárias para transformar a primeira string na segunda. Para efeito de ênfase, observe que Ms não são contados.

Para medirmos uma distância entre as duas strings, associamos um número a cada operação: R-1,M-0,D-1,I-1. A função de distância consiste na soma de todos os valores referentes às operações R, M , D, e I. Como no exemplo que transforma "gato" em "gente" realizamos 4 Ds e 5 Is então a distância, segundo essa edição é de 9. Porém 9 não é o menor número de edições que transformam "gato" em "gente".

Uma outra forma de editar "gato" e obter "gente" é:

```
MRIMR
ga_to
gente
```

Nesse caso, a distância de edição é 3 que é a mínima distância de edição possível entre essas duas strings. As operações de edição são:

M(g) : *match* - mantém g.
R(a,e): *replace* - troque a por e.
I(n): *insert* - Insira n
etc.

A forma clássica de se descobrir a distância de edição é através da programação dinâmica [1]. Nesse algoritmo é montada uma tabela de n+1 linhas por m+1 colunas onde:

m = número de caracteres da primeira string
n = número de caracteres da segunda string.

Essa tabela é preenchida de forma a se obter o menor

J. Kinoshita, R. L. A. Rocha são professores do Departamento de Engnharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo. (emails: jorge.kinoshita@poli.usp.br; luis.rocha@poli.usp.br).

número de operações que transformam uma string em outra. No exemplo, a distância é 3, mas esse valor pode estar associado a uma outra edição:

```
MIRMR
g_ato
gente
```

Ou seja, podem existir várias edições que correspondem à distância mínima de edição.

Existem diversas aplicações para a função de distância de edição [2]:

- correção ortográfica
- reconhecimento de fala
- análise de DNA
- detecção de plágio.

Descobrir a distância de edição entre duas strings, corresponde a uma forma de alinhar as duas strings [1].

III. O ALGORITMO CLÁSSICO

Uma explicação mais detalhada do algoritmo pode ser vista em [1]. O algoritmo usa programação dinâmica e consiste em preencher uma tabela de (n+1)×(m+1) posições onde m é o número de caracteres da primeira string e n é o número de caracteres da segunda string. A tabela possui as linhas variando de zero a m e colunas de zero a n.

A string s1 corresponde a s1[1..m], ou seja, à substring que começa na posição 1 e termina na posição m e s2 corresponde a s2[1..n].

A célula [i,j] da tabela consiste na distância D(i,j) de edição entre a substring s1[1..i] e a substring s2[1..j].

A base para a programação dinâmica é feita preenchendo os valores das células:

D(0,j) corresponde ao menor número de operações que transformam a substring s1[1..0], a cadeia vazia, em s2[1..j]. O valor a ser colocado nessas células é j porque é necessário j Inserções para transformar o vazio em s2[1..j].

D(i,0) corresponde a transformar a s1[1..i] na cadeia vazio e para isso são necessários i Deletes sobre s1[1..i].

A distância D(i,j) entre s1[1..i] e s2[1..j] é calculada a partir do trabalho para se fazer o alinhamento de substrings com 1 caractere a menos (pois as operações são sempre sobre um caractere). D(i,j) pode ser:

- D(i,j-1) + 1: porque para transformar s1[1..i] em s2[1..j] já sabendo como transformar s1[1..i] em s2[1..j-1] basta uma operação (inserir o caractere s2[j] em s2[1..j-1])
- D(i-1,j) + 1: porque para transformar s1[1..i] em s2[1..j] já sabendo como transformar s1[1..i-1] em s2[1..j] basta uma operação (deletar o caractere s1[i]).
- D(i-1,j-1) + t(i,j) onde: se (s1[i] = s2[j]) então t(i,j) = 1, senão t(i,j) = 0. Isso corresponde à operação de Replace (trocar o caractere s1[i] por s2[j]) ou Match (repassar o caractere s1[i] para s2[j]).

O valor a ser preenchido em D(i,j) é o menor valor possível dentre os 3 valores.

A tabela vai sendo preenchida até que se obtém D(n,m).

Aplicando isso às strings "gato" e "gente" obtemos a tabela 1.

TABELA 1
APLICAÇÃO DO ALGORTIMO CLÁSSICO PARA O ALINHAMENTO DE "GATO" E "GENTE".

	0	1-g	2-a	3-t	4-o
0	0	1	2	3	4
1-g	1	0	1	2	3
2-e	2	1	1	2	3
3-n	3	2	2	2	3
4-t	4	3	3	2	3
5-e	5	4	4	3	3

O algoritmo pode ser feito de forma a se preencher uma linha de cada vez, à medida que chegam os caracteres de s2.

O número D(x,y) escolhido em cada célula (x,y) é o menor dentre 3 valores. Por exemplo, D(1,1) que corresponde a transformar a string "g" na string "g" poderia ser obtido de 3 formas:

- A partir de s1 = "g" e s2 = vazio D(0,1), para se obter "g" em s2 deveria haver uma inserção de "g" em s2. O valor de D(1,1) nesse caso seria de 1+ 1 = 2. Isso corresponderia a duas operações "Deletar g de s1 (daí D(0,1) =1) e depois inserir g em s2 (D(1,1) =2)".
- A partir de s1 = "" e s2 = "g", para se obter "g" em s2 deveria se apagar "g" de s1. Nesse caso, isso corresponderia a duas operações: "Inserir "g" em s2 (D(1,0)=1)e depois deletar g em s1".
- A partir de s1 ="" e s2 = "", para se obter g basta fazer Match, ou seja, repassar o "g" de s1 para s2 com peso zero.

Para se descobrir o alinhamento entre "gato" e "gente" basta observar quais foram as operações que levaram a D(n,m) no caso D(5,4) e ir traçando o caminho até D(0,0). Às vezes existe mais de um caminho para se chegar a uma célula. Por exemplo, para se chegar em D(3,2) pode-se ir por D(2,1) e D(2,2). Neste exemplo existem dois caminhos possíveis gerando a distância de edição 3 gerando os dois possíveis alinhamentos mostrados no item I.

IV. O TRANSDUTOR ADAPTATIVO APLICADO À DISTÂNCIA DE EDIÇÃO

Um autômato normal é composto de estados e transições. A cada transição está associado um caractere do alfabeto. Uma transição efetiva uma mudança na configuração do autômato por trocar de um estado a outro e consumir o caractere de entrada. Supondo que não exista uma transição capaz de consumir o caractere de entrada então a cadeia de entrada não é reconhecida.

Um autômato adaptativo é diferente. Ele pode estar preparado para se adaptar a situações não previstas e para isso ele utiliza uma função adaptativa. Caso não haja uma transição a ser disparada, o autômato adaptativo pode criar essa transição.

Um transdutor adaptativo é semelhante ao autômato adaptativo, mas gera uma saída ao se realizar a transição.

O algoritmo para o alinhamento pode ser reescrito usando transdutores adaptativos. A idéia é criar um transdutor que gere como resposta às operações de alinhamento para se obter s2 a partir de s1. O algoritmo vai funcionar de forma semelhante ao algoritmo clássico.

O transdutor vai ser representado por 2 elementos: estados e transições.

Ao se usar autômatos, na nomenclatura normal, não associamos valores a cada estado. Mas aqui, vamos associar a cada estado E, duas funções: a função de distância de edição d(E) e a função de pronto para disparar p(E).

O transdutor adaptativo começa com uma matriz de estados (n+1)×(m+1) e nenhuma transição entre os estados. Todos os estados estão inabilitados para o disparo a menos do estado (0,0). O estado (0,0) pede autorização para criar transições para seus estados vizinhos. À medida que isso ocorre, seus vizinhos se tornam habilitados a disparar transições e o processo continua até o estado (n,m) estiver habilitado ao disparo. Nessa situação, é possível se ter o alinhamento entre s1 e s2.

O transdutor adaptativo é composto de:

1) estados

Geralmente um estado é representado por um número, mas em nosso caso, vamos representar o estado por dois números correspondentes à célula da tabela usada no algoritmo clássico. Assim, teremos os estados de (0,0) a (n,m).

A cada estado associamos:

- função distância de edição

A "distância de edição", d(E) que corresponde ao menor número de operações de edição para transformar a string s1[1..j] na string s2[i..i]. A princípio ela é inicializada com o valor máximo i+j. O valor vai sendo reduzido à medida que transições são criadas para E. Ao final do algoritmo, os valores de d(E) correspondem a D(E) como no exemplo da tabela 1.

- função pronto

Um estado E pode ser destino de no máximo 3 transições e origem de no máximo 3 transições.

Um estado (i,j) pode criar no máximo 3 transições para os estados (i+1,j), (i,j+1) e (i+1,j+1). Se ele estiver na borda haverá menos estados vizinhos. Por exemplo, o estado (n,m) não possui vizinhos com quem se ligar. Da mesma forma, ele pode receber transições de no máximo 3 outros vizinhos.

O estado E só se tornará pronto para criar transições (transições onde E é origem) depois que o valor d(E) for o valor mínimo. Temos certeza que o valor é mínimo somente quando todos os outros estados tentaram criar transições a E. Assim, quando o transdutor adaptativo é criado, ele é inicializado com 3 (se não estiver na borda) e a cada tentativa de se criar uma transição, a função p(E) é executada.

A função "pronto", p(E) faz:

p(E) = p(E) -1;

Se p(E) =0 então, dispara o processo "criação de transições" explicado em 2.1 nesse mesmo item.

2) transição

Cada transição vai ser representada por:

(estado origem, estado destino, caractere consumido, saída).

Exemplo:

((0,0), (0,1), s1[1], D); transita do estado (0,0) para o estado (0,1), consome o caractere s1[1] e escreve D.

O caractere consumido varia dependendo da operação. A operação D consome caracteres de s1, a operação I, caracteres de s2 e R e M consomem caracteres de s1 e s2. No caso das operações R e M. No caso da operação R, vamos representar os dois caracteres consumidos como (s1[i], s2[j]) para o estado (i,j).

As transições para (i,j) provém somente de 3 estados outros estados possíveis e podem ser apenas de 4 tipos:

- ((i,j-1), (i,j), s1[j], D)
- ((i-1,j), (i,j), s2[i], I)
- ((i-1,j-1), (i,j), s1[j], M)
- ((i-1,j-1), (i,j), (s1[i],s2[j]), R)

A princípio, o transdutor começa sem nenhuma transição entre estados.

Em um autômato normal, uma marca (um token) passa de um estado para outro consumindo um caractere da cadeia de entrada. Nosso transdutor adaptativo funciona de forma diferente.

2.1) A criação de transições

Quando um estado (i,j) estiver pronto para criar transições (p(E) =0), ele fará:

p(i,j+1); isto é: avisa que houve uma tentativa para se criar uma transição para o estado (i,j+1).

Se (d(i,j+1) <= d(i,j) + 1) então crie a transição: ((i,j), (i,j+1), s1[j+1], D)

p(i+1,j);

Se (d(i+1,j) <= d(i,j) + 1) então crie a transição: ((i,j), (i+1,j), s2[i+1], I)

p(i+1,j+1);

Se d(i+1,j+1) <= d(i,j) + 1) então

    Se (s1[j+1] igual s2[i+1])) então

        crie a transição: ((i,j), (i+1,j+1), s1[j+1], M)

    senão

        crie a transição: ((i,j), (i+1,j+1), (s1[j+1],s2[i+1]), R)

Inicialização do transdutor

O transdutor é inicializado criando-se os estados de (0,0) a (n,m).

Para o estado (0,0), p(0,0) = 0; ou seja, este estado pode criar outras transições.

Para os estados (0,j) para j de 1 a m, p(0,j) = 1, d(0,j) = j
Para os estados (i,0) para i de 1 a n, p(i,0) = 1, d(i,0) = i
Para os demais estados (i,j), p(i,j) = 3, d(i,j) = i+j

Funcionamento do transdutor

O estado (0,0) executa o processo de criação de transições. Ele criará 3 transições: D, I e R ou M.
Quando essas transições são criadas, o estado (0,1) e o estado (1,0) estão prontos para criar outras transições. As transições sempre são criadas de forma a se reduzir o valor d(E).

Um exemplo

Aplicando esse transdutor adaptativo às strings "gato" e "gente" temos algo semelhante à tabela 1 onde cada célula (i,j) da tabela 1 corresponde ao estado (i,j) e d(i,j) corresponde ao valor da célula. Além disso, o autômato adaptativo teria gerado, entre outras, as transições na tabela 2.

TABELA 2
ALGUMAS TRANSIÇÕES GERADAS PELO TRANSDUTOR ADAPTATIVO NO ALINHAMENTO DE "GATO" E "GENTE"

estado origem	**estado destino**	**consome**	**escreve**
(0,0)	(0,1)	g	D
(0,0)	(1,1)	g	M
(1,1)	(2,1)	e	I
(2,1)	(3,2)	(a,n)	R
(3,2)	(4,3)	t	M
(4,3)	(5,4)	(o,e)	R
(2,2)	(3,3)	(t,n)	R

Para se obter as operações que transformam a string s1 em s2 basta observar o caminho que leva (0,0) a (n,m).
Dado que diversos estados podem ficar prontos para disparar transições, então se essas transições ocorressem em paralelo, esse algoritmo poderia rodar mais rápido do que o tradicional da seguinte forma: logo após a inicialização apenas o estado (0,0) está pronto (1 estado pronto) e pode tornar os estados (0,1) e (1,0) prontos para o disparo (2 estados prontos). Esses estados podem tornar os estados (2,0), (1,1) e (0,2) prontos (3 estados prontos); depois é a vez dos estados (3,0), (2,1), (1,2) e (0,3) (4 estados prontos) e assim por diante, ou seja, a tabela 1 é varrida pelas diagonais podendo processar os seguintes números de estados prontos em paralelo: 1, 2, 3, 4, 5, 5, 4, 3, 2 e 1. Se todos eles levassem uma unidade de tempo para processar e disparar então o algoritmo paralelo rodaria em 10 unidades de tempo, pois cada uma das 10 diagonais seria executada em uma unidade de tempo. A fim de generalizar essa idéia, observamos o seguinte: uma diagonal é formada pelas células (x,y) da tabela (n+1)x(m+1) tais que x+y = N. A primeira diagonal com a célula (0,0) corresponde ao número 0. A segunda diagonal às células (1,0) e (0,1) que somam 1 e assim por diante até que a última diagonal corresponde à soma m+n. Para que uma célula (x,y) qualquer no meio da tabela esteja pronta para o disparo ela depende das células (x-1,y) e (x,y-1) que estão na diagonal x+y-1 e também da célula (x-1,y-1) que está na diagonal x+y-2. Se todas as células das duas diagonais anteriores já tiverem executado seu trabalho então todas as células da diagonal x+y pode ser executada. A célula da diagonal 0 não depende de ninguém e as células da diagonal 1 dependem apenas da (0,0). Assim, por indução, temos que todas as outras diagonais podem ser resolvidas com base apenas nas diagonais anteriores, isto é, o trabalho pode ser realizado em paralelo seguindo pelas diagonais.

Comparando com o tradicional que é da ordem de (m+1) × (n+1), temos que aproveitando o máximo de paralelismo possível, o algoritmo executaria em um tempo da ordem de m+n (que corresponde ao número de diagonais que possui uma tabela (m+1)×(n+1)). Já o algoritmo apresentado em [4] é de ordem exponencial e, portanto, de desempenho pior que o tradicional.

V. CONCLUSÃO E CONSIDERAÇÕES FINAIS

Usando o transdutor adaptativo torna-se possível fazer o alinhamento entre duas strings de forma a se recuperar o caminho. O paralelismo desse algoritmo não é muito grande porque as células sendo geradas dependem das vizinhas, no entanto, é possível se trabalhar com autômatos adaptativos paralelos [3] de forma a se agilizar um pouco o algoritmo. O algoritmo aqui proposto permite um grau maior de paralelismo em relação ao tradicional porque os estados disparam assim que estiverem prontos. Como eles dependem dos estados vizinhos, se todos os estados disparassem assim que estivessem prontos então os estados (x,y) correspondentes a cada diagonal (ou seja, x+y=N) seriam disparados a cada vez para N de zero a m+n. Assim, em m+n disparos paralelos, o algoritmo seria resolvido, ao contrário do tradicional que seria resolvido em um tempo da ordem de m*n.

O transdutor adaptativo aqui proposto gera todas as soluções possíveis porque uma transição de E para V é criada desde que d(V) não aumente. Poderíamos pensar numa forma de se ter uma solução única. Nesse caso, uma transição de E para V é criada desde que d(V) seja reduzido. Existe um mecanismo de se retirar o não determinismo para autômatos adaptativos em [5], no entanto, escolher uma transição qualquer que leve ao d(V) mínimo já resolve o problema do alinhamento e por isso [5] não foi considerado nesse artigo.

A referência [4] trata do mesmo assunto: o alinhamento entre duas cadeias usando autômatos adaptativos. Aqui buscamos enfatizar o algoritmo clássico de alinhamento, entretanto sem enfocar o formalismo que pode ser visto em [4]. Porém [4] não utiliza programação dinâmica para o alinhamento, ou seja, não guarda os estados intermediários e por isso faz muito mais processamento que o necessário.

A forma como implementamos p(E) lembra redes de

Petri. Poderíamos pensar que o estado fica pronto para criar transições apenas quando todos os estados que poderiam se ligar a E através de transições estivessem com marcas. Porém o mecanismo é ainda diferente de redes de Petri porque no caso de redes de Petri, as transições e estados são estáticos.

A pesquisa em alinhamento de strings é bem vasta, sendo [1] uma referência básica sobre o assunto. Ela possui uma forte relação com autômatos. A pesquisa sobre o uso autômatos adaptativos em alinhamento de strings pode evoluir para englobar árvores de sufixos e outros mecanismos que tornem o alinhamento ainda mais rápido.

VI. AGRADECIMENTO

Ao professor João José Neto pelo apoio, críticas e sugestões.

REFERÊNCIAS

[1] Gusfield, Dan "Algorithms on Strings, Trees and Sequences"; Cambridge University Press, 1997

[2] Gonzalo Navarro. A guided tour to approximate string matching. ACM Computing Surveys, 33(1):31–88, 2001.

[3] ROCHA, R. L. A. ; GARANHANI, César Eduardo Cavani . Parallel adaptive finite state automata. In: XII Argentine Congress on Computer Science - CACIC 2006, 2006, Potrero de los Funes. Anales del CACIC 2006. Potrero de Los Funes: : Universidad Nacional de San Luis, 2006. v. 1. p. 1-12.

[4] RODRIGUES, E. S. C. ; RODRIGUES, F. A. ; ROCHA, R. L. A. . Automatos Adaptativos para Emparelhamento de Cadeias. In: Segundo Workshop de Tecnologia Adaptativa - WTA'2008, 2008, São Paulo. Memórias do Segundo Workshop de Tecnologia Adaptativa. São Paulo : Escola Politécnica da USP, 2008. v. 1. p. 27-30.

[5] ROCHA, R. L. A. ; JOSÉ NETO, João . An Adaptive Finite-State Automata Application to the problem of Reducing the Number of States in Approximate String Matching. In: XI Congreso Argentino de Ciencias de la Computación - CACIC 2005, 2005, Concordia. Proceedings of the CACIC'2005. Concordia : Universidad Nacional de Entre Ríos - Argentina, 2005. v. 1. p. 1-9.