Métodos eficientes para a detecção de padrões exactos (*pattern-matching*) em sequências biológicas

Sérgio Deusdado, CIMO – Centro de Investigação de Montanha, ESA do Instituto Politécnico de Bragança, 5300 Bragança, Portugal

Resumo Os algoritmos de detecção de padrões (*pattern-matching*), sejam exactos ou aproximados, são fundamentais na maioria das aplicações orientadas à análise de sequências biológicas. Nesta comunicação apresenta-se um novo algoritmo, denominado DC, desenvolvido para a especificidade do *pattern-matching* exacto, bem como uma análise comparativa do seu desempenho. Conclui-se que o desempenho do novo algoritmo supera, em média, o dos seus concorrentes, atribuindo-se o ganho de eficiência, sobretudo, à introdução de uma nova regra de filtragem denominada regra de compatibilidade.

1 Introdução

Basicamente um algoritmo de pesquisa de padrões exactos procura encontrar a totalidade das instâncias de uma subsequência padrão *p*, de tamanho *m* numa sequência *x* de tamanho *n*, sendo $n \geq m$. As sequências *m* e *n* são produtos de um alfabeto finito, de dimensão σ. A informação biológica inclui a informação das cadeias peptídicas que formam as proteínas. As proteínas resultam de uma sucessão de aminoácidos que, por sua vez, resultaram de codões que os codificaram. Um codão corresponde a 3 bases consecutivas, e com 3 bases (do alfabeto quaternário do ADN) podem formar-se 64 diferentes combinações. Devido à existência da redundância introduzida pelos codões sinónimos apenas 20 diferentes aminoácidos são expressados pelo nosso ADN. Assim, o alfabeto das proteínas tem dimensão $\sigma=20$, correspondendo a $\Sigma=\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

Os algoritmos de detecção de padrões (*pattern-matching*), sejam exactos ou aproximados, são fundamentais na maioria das aplicações orientadas à análise de sequências, a saber, na segmentação de sequências, no alinhamento, na predição de genes, na análise filogenética, etc. O desempenho desses algoritmos basilares condiciona o desempenho dessas aplicações, pelo que o avanço nesses algoritmos de *pattern-matching* se reveste de grande importância. Nesta comunicação apresenta-se um novo algoritmo, denominado DC, desenvolvido para a especificidade do *pattern-matching* exacto, bem como uma análise comparativa do seu desempenho.

2 Descrição e funcionamento do novo algoritmo DC

O algoritmo DC (Deusdado e Carvalho) [1] é baseado em caracteres individualmente considerados, em 1-gramas portanto. Se pensarmos em proteínas é baseado em aminoácidos, se considerarmos a linguagem natural é baseado em caracteres: letras, números, símbolos de pontuação, etc. Esta propriedade confere-lhe elevada flexibilidade pois pode facilmente adaptar-se para funcionar com *n*-gramas maiores e assim aumentar o seu potencial em face de pesquisas em "textos" baseados em alfabetos de dimensão reduzida ou muito reduzida, tipicamente com $\sigma<8$.

A estratégia de pesquisa assenta no conceito do caracter central de uma janela de pesquisa como referência para a primeira filtragem da possibilidade de ocorrências. Assim, o caracter central da janela de pesquisa marca o eixo usado para todos os alinhamentos, sendo que o tamanho da janela de pesquisa corresponde a $2m-1$ caracteres. Vejam-se os casos extremos (ver Tabela 2.1) para um padrão *p* com $m=10$ e os seus alinhamentos extremos numa janela de tamanho $2m-1=19$.

									cc									
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	2	3	4	5	6	7	8	9	10									
									1	2	3	4	5	6	7	8	9	10

Tabela 2.1 – Casos extremos de alinhamento de um padrão dentro da janela de pesquisa no algoritmo DC.

A estratégia de pesquisa do algoritmo DC executa um ciclo de avanços antes de se centrar na janela de pesquisa. Esse ciclo de avanços termina quando a regra de avanços extra, baseada na heurística do mau caracter do algoritmo BMH [2], encontrar no texto um caracter igual a *p[m]*, ou seja igual ao último caracter do padrão, o que equivale a dizer quando o avanço proporcionado por dita regra for nulo. Assim, pode utilizar-se um ciclo de avanços independente enquanto não for encontrado um caracter igual a *p[m]* simplesmente percorrendo a sequência *x,* analisando inicialmente o caracter central da primeira janela em $cc_{1.1}=x[m]$, depois o caracter em $cc_{1.2}=m+f(x[cc_{1.1}])$, e assim sucessivamente, sendo *f*, a função de avanço suplementar. O definitivo cc_1 é determinado quando a função de avanço devolver um avanço nulo. Tendo por exemplo o padrão *p=”AFFKRQYYER”* e a resultante tabela de avanços (Tabela 2.2), podemos afirmar que o algoritmo DC, na pesquisa do padrão *p*, só estabelecerá a primeira janela quando através dos sucessivos saltos proporcionados pela tabela de avanços encontrar em *x* o caracter “*R*”.

A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
9	10	10	1	7	10	10	10	6	10	10	10	10	4	0	10	10	10	10	2

Tabela 2.2 – Tabela de avanços do padrão ”AFFKRQYYER” baseada na heurística do mau caracter.

Encontrada a primeira janela de pesquisa, sendo sempre certo que $x[cc_n]=p[m]$, haverá tantos alinhamentos a testar quantas as ocorrência do caracter *p[m]* no padrão. No caso em análise há apenas duas ocorrências de “R” no padrão *p*, mas no caso limite, que corresponde ao padrão uniforme, poderia haver até *m* alinhamentos. Esses possíveis alinhamentos são classificados em compatíveis ou incompatíveis, aplicando a regra da compatibilidade. A regra de compatibilidade atende ao caracter precedente do caracter central, portanto $x[cc_n-1]$. Assim, para todos os alinhamentos possíveis, que resultam das ocorrências do caracter *p[m]* no padrão, é analisado o caracter antecedente dessas ocorrências para definir que caracteres são compatíveis com o alinhamento.

O algoritmo DC assenta em duas fases, a fase de pré-processamento e a fase de processamento ou pesquisa. A fase de pré-processamento é orientada ao padrão, mais concretamente ao estudo da composição do padrão bem como das condições de compatibilidade para cada alinhamento passível de resultar numa descoberta do padrão. Na fase seguinte, que corresponde à pesquisa de réplicas do padrão percorrendo o texto através do avanço iterativo das janelas de pesquisa, o conhecimento adquirido no pré-processamento é fundamental. Sobretudo, no sentido de estabelecer mecanismos simples e eficazes para aumentar a selectividade com que se lida com os *m* alinhamentos que podem existir dentro de uma janela de pesquisa.

Antecedendo e sucedendo a cada estabelecimento da janela de pesquisa ocorre um ciclo de avanços pela sequência até nova ocorrência do caracter em *p[m]*. De referir que o DC incorpora também uma componente de avanço fixa exercida após o término das verificações em cada janela estabelecida, essa componente fixa corresponde a *m* caracteres. Consequentemente, o primeiro avanço na procura da nova janela incorpora a componente fixa igual a *m*, à qual se segue o, já aludido, ciclo de avanços

suplementares. Como súmula desta descrição mais informal segue-se a apresentação do fluxograma do DC, patente na Figura 2.1.

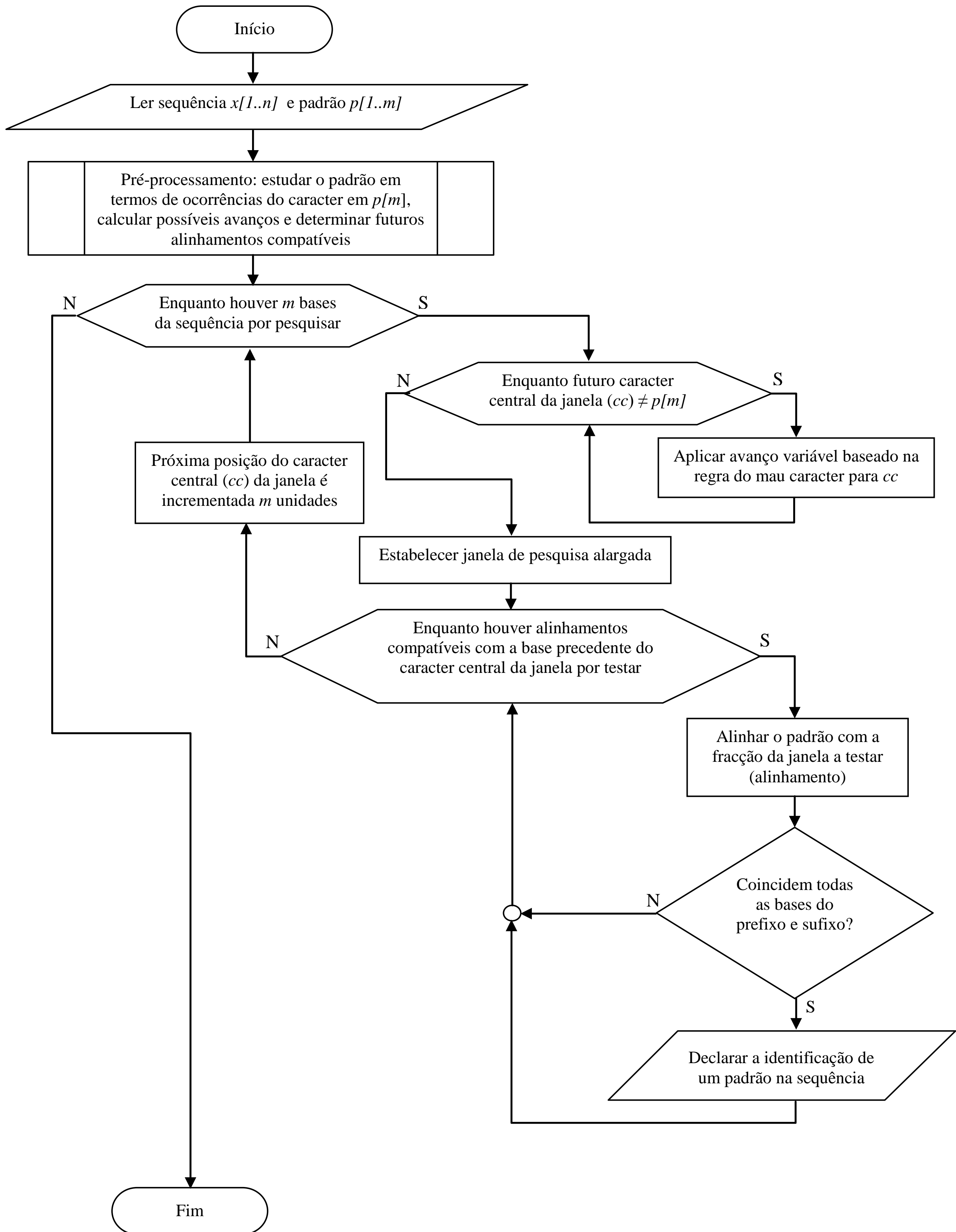

Figura 2.1 – O fluxograma do DC.

Numa abordagem lata, pode-se dizer que o DC é um algoritmo baseado em heurísticas, apresentando resultados que demonstram flexibilidade e eficiência em qualquer tipo de alfabeto. Segue-se uma análise detalhada das diferentes fases do novo algoritmo.

2.1 Fase de pré-processamento

No pré-processamento do padrão estão incluídas as seguintes computações:
- a análise das ocorrências dos caracteres *p[m]* no padrão;
- o cálculo da tabela de compatibilidades dos alinhamentos das ocorrências de *p[m]* no padrão;
- o cálculo da tabela de avanços suplementares.

2.1.1 Análise das ocorrências dos caracteres *p[m]* no padrão

A tabela de estudo das ocorrências dos caracteres *p[m]* no padrão é muito simples. Trata-se de um vector que armazena o número total de ocorrências dos caracteres *p[m]* na primeira célula, guardando nas restantes a posição em que aparecem. Retomando o padrão *p="AFFKRQYYER"*, onde o caracter em *p[m]* ocorre na 5ª e 10ª posições, apresenta-se o resultado do estudo na Tabela 2.3.

1	2	3	4	…
2	5	10	0	…

Tabela 2.3 – Tabela de estudo das ocorrências de *p[m]* no padrão.

2.1.2 Cálculo da tabela de compatibilidades

A tabela de compatibilidades incide apenas sobre os alinhamentos das ocorrências de *p[m]* em *p*, pois daí resultarão os alinhamentos a, eventualmente, testar na futura fase de pesquisa. A Tabela 2.4 mostra esses alinhamentos e permite-nos observar que, atendendo apenas ao caracter precedente do futuro caracter central (*cc-1*), "K" e "E" são as únicas compatibilidades a registar, qualquer outro caracter é incompatível.

								cc-1	cc					
					A	F	F	K	R	Q	Y	Y	E	R
A	F	F	K	R	Q	Y	Y	E	R					

Tabela 2.4 – Alinhamentos do padrão pelas ocorrências do caracter em *p[m]*.

A Tabela 2.5 encerra as compatibilidades observadas, na coluna correspondente ao caracter compatível guardam-se os alinhamentos compatíveis. Caso haja alinhamentos múltiplos para um mesmo *cc-1*, a ordem de registo corresponde à ordem inversa de aparição no padrão. Esta inversão justifica-se pelo facto de que na altura de testar os alinhamentos estes são testados pela ordem seguida por esta tabela. Note-se que quanto mais à esquerda se situar o alinhamento do padrão, menor será a posição onde ocorre na sequência, desta forma garante-se que se descobrem as instâncias do padrão na ordem de aparição correcta.

p[m]="R"

A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
0	0	0	10	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

Tabela 2.5 – Tabela de compatibilidades.

2.1.3 Tabela de avanços suplementares

A tabela de avanços suplementares já foi abordada e representada na Tabela 2.2. A mesma é utilizada para percorrer celeradamente as regiões da sequência a pesquisar que não apresentam ocorrências de *p[m]*. Esse percurso acontece continuadamente a partir do caracter cc_n+m, já que *m* é o avanço fixo após cada iteração. Os avanços suplementares são fornecidos inicialmente por $f(x[cc_n+m])$, sendo os subsequentes obtidos pelos incrementos >0 provenientes da função *f* para o novo caracter de destino, a função *f (c)* devolve de avanço suplementar para o caracter *c*.

2.2 Fase de pesquisa ou processamento

A fase de processamento principia com um ciclo de avanços que se inicia no caracter *x[m]*, assim, *x[m]* será o primeiro candidato a caracter central cc_1, que se referencia como $cc_{1.1}$. Verifica-se se $f(x[cc_{1.1}])>0$. Em caso afirmativo, a $cc_{1.1}$ incrementa-se $f(x[cc_{1.1}])$ para obter $cc_{1.2}$. Repete-se o processo até que $f(x[cc_{1.n}])=0$. Atingida esta condição obtém-se cc_1. A primeira janela de pesquisa é estabelecida com o seu caracter central a corresponder a $cc_1=p[m]$. O passo seguinte é verificar se há alinhamentos compatíveis a testar, para tal recorre-se à tabela de compatibilidade criada na fase de pré-processamento e caso haja alinhamentos a testar iniciam-se as comparações caracter a caracter para o primeiro alinhamento compatível. As verificações interrompem-se se suceder uma falha, ou então resultam numa ocorrência do padrão. As verificações não incorrem em redundância pois os caracteres *cc* e *cc-1* já foram anteriormente testados, assim testam-se apenas os caracteres sobrantes presentes no prefixo e/ou sufixo, sempre da esquerda para a direita.

Para clarificar todos os passos da fase de pesquisa do algoritmo DC, fornece-se, na Figura 2.2, uma proposta de implementação em linguagem C dessa fase.

```c
void DC_Search(char *texto, long n, char *padrao, unsigned int m)
{
 int b,cc,na,precedente,ia,iap,j,prefixo;
 char c;

 b=padrao[m-1]; //Último caracter no padrão
 cc=m-1; // Primeiro caracter central cc
 while (((c=texto[cc])!=b) && (cc<=n)) cc+=xshift[c];  // Primeiro ciclo de avanços
 while (cc<=n)
   {
     precedente=texto[cc-1];
     ia=1;
     while ((na=compatibilidade[precedente][ia])>0)
         {
            prefixo=na-2; // Tamanho do prefixo
            iap=cc-prefixo-1; // Posição para alinhamento do padrão
            j=0;
            while ((j<prefixo) && (texto[iap+j]==padrao[j])) j++; // Teste do prefixo
            if (j==prefixo)
              {
               j+=2; //cc e cc-1 já foram testados
               while ((j<m) && (texto[iap+j]==padrao[j])) j++;   // Teste do sufixo
               if (j>=m) printf ("\nOcorrencia na posicao %d.", iap);
              }
            ia ++;  // Segue para o próximo alinhamento compatível
         }
     cc+=m;  // Avanço fixo
     while (((c=texto[cc])!=b) && (cc<=n)) cc+=xshift[c]; // Ciclo de avanços extra
   }
}
```

Figura 2.2 – Proposta de implementação da fase de pesquisa do algoritmo DC.

2.3 Análise de complexidade

A análise de complexidade do algoritmo DC pode ser vista nas duas fases do algoritmo. Na fase de pré-processamento a complexidade temporal e espacial é repartida pelas seguintes necessidades computacionais:

- Analisar as ocorrências de *p[m]* que compõem o padrão;
Complexidade temporal associada: $O\ (2m)$
Complexidade espacial associada: $O\ (m+1)$

- Calcular a tabela de compatibilidades;
Complexidade temporal associada: $O\ (m)$
Complexidade espacial associada: $O\ (\sigma m)$

- Calcular a tabela de avanços suplementares.
Complexidade temporal associada: $O\ (\sigma+m)$
Complexidade espacial associada: $O\ (\sigma)$

Na fase de processamento a análise de complexidade que importa considerar é a temporal. Começando por analisar o melhor caso, que corresponderá a pesquisar toda a sequência aplicando apenas o primeiro ciclo de avanços, portanto sem encontrar qualquer ocorrência de *p[m]* e, sempre com avanços máximos de *m*. Assim, podemos considerar o melhor caso como $O\ (n/m)$.

No outro extremo, o pior caso ocorre quando o algoritmo terá de estabelecer janelas a cada *m* caracteres, usando apenas o avanço fixo e nunca recorrendo a qualquer avanço suplementar. Concomitantemente, por cada janela teria de testar *m* alinhamentos (o máximo) e por cada alinhamento verificar *m-2* caracteres (os caracteres *cc* e *cc-1* são verificado apenas uma vez em paralelo para todos os alinhamentos). Em suma trata-se de uma sequência com repetições do mesmo símbolo e o padrão é uma subsequência dessa sequência. Com estes considerandos, pode inferir-se que a complexidade temporal no pior caso se cifra em $O\ ((n/m)(2+m(m-2)))$. Simplificando, temos genericamente como pior caso, complexidade temporal $O\ (nm)$.

No caso médio, por análise experimental, verificamos que se trata de um algoritmo sublinear, com complexidade média inferior a $O(n)$, isto dado o comportamento em curva logarítmica descendente que é evidenciado nos gráficos de desempenho. A determinação/prova teórica do caso médio é matéria complexa e constituirá certamente um interessante problema em aberto que ficará como desenvolvimento futuro.

3 Resultados experimentais e comparações

Para testar o desempenho do algoritmo DC escolheram-se os algoritmos de referência em pesquisa de padrões exactos para alfabetos com $\sigma \geq 20$. Isto porque se pretendeu optimizar o DC para pesquisas em proteínas e linguagem natural. Assim, decidiu-se incluir nas comparações, pelo seu desempenho, o algoritmo BMH [2], por ser o mais rápido dos clássicos, o FJS [3] de 2005, que foi publicado como o mais rápido para linguagem natural e os algoritmo SBNDM [4] e WML [5], por serem considerados como dos mais rápidos algoritmos para alfabetos moderados mas permanecendo competitivos em todos os alfabetos. Do algoritmo WML usou-se a versão 2-gramas considerando-a a versão mínima do WML, pois sendo um algoritmo baseado em *hashing* de *q*-gramas descendente do Shift-Or [6], perderia a sua essência se reduzido a

um algoritmo orientado ao caracter como os demais. Quer o BMH, quer o SBNDM são algoritmos bem conhecidos e referenciais recorrentes na literatura para a avaliação de desempenho de novos algoritmos.

O FJS é um algoritmo baseado em heurísticas, que reúne numa solução híbrida, ideias dos algoritmos de KMP [7], BM [8] e Sunday [9] na tentativa de aproveitar os pontos fortes de cada um deles. O resultado foi um algoritmo muito competitivo, dominante na data de apresentação, segundo os autores, para alfabetos com $\sigma \geq 8$.

As implementações usadas são genuínas, codificadas em linguagem C e compiladas, usando máxima optimização –O3, recorrendo ao compilador *gcc* - versão 3.4.2. A implementação do algoritmo de BMH foi adaptada de [10], a implementação dos algoritmos SBNDM e WML foi gentilmente fornecida pelos seus autores e a implementação do algoritmo FJS é providenciada pelos autores em [3].

Como plataforma dos testes utilizou-se um sistema computacional com base num processador Intel Pentium IV 3.4GHz, 8KB L1 + 512KB L2 de cache e 1GB de DDR-RAM, sobre Windows XP Professional SP2 OS. Os tempos foram colectados com recurso a uma função incluída num biblioteca do sistema operativo, referida como *timeGetTime()*, que fornece medições instantâneas do tempo em milissegundos.

As sequências usadas foram uma colectânea de proteomas obtidos da base de dados pública Integr8[1], que inclui os proteomas das espécies *Homo Sapiens, C. Elegans, A. Thaliana e Mouse Musculus*, cujo conjunto perfaz uma sequência com cerca de 50MB. Já para os testes em linguagem natural usou-se uma compilação de 37 livros electrónicos, contendo textos de diversos autores da literatura europeia obtidos a partir da base de dados do Projecto Gutenberg[2], num total de cerca de 50MB de informação. Quanto aos padrões, para o caso das pesquisas em sequências de aminoácidos foram criados aleatoriamente 100 padrões por cada uma das categorias de comprimentos de padrão testados, a saber, *m=2, 4, 8, 16, 32, 64, 128*. Para o caso da linguagem natural foram usadas palavras ou frases, ou excertos das mesmas retiradas ao acaso do texto. Os algoritmos em contenda efectuaram as pesquisas para todos os padrões considerados, e para cada categoria foi recolhido o tempo médio de execução (T.E.) da pesquisa em milissegundos. Os resultados estão patentes nas Tabela 3.1 e Tabela 3.2.

	DC		BMH		FJS		SBNDM		WML2	
m	T.E. (ms)	Rank	T.E. (ms)	Rank	T.E. (ms)	Rank	T.E. (ms)	Rank	T.E. (ms)	Rank
2	**173**	**1**	271	4	174	2	177	3	672	5
4	**111**	**1**	149	4	120	2	132	3	235	5
8	**67**	**1**	86	3	77	2	96	4	111	5
16	**44**	**1**	56	3	52	2	61	5	60	4
32	32	2	42	5	40	4	**31**	**1**	37	3
64	**26**	**1**	33	4	32	3	$>\omega$	?	27	2
128	**22**	**1**	31	4	30	3	$>\omega$	?	**22**	**1**

Tabela 3.1 – Resultados de desempenho comparado do algoritmo DC com os concorrentes para diversas categorias de padrões pesquisando sequências de aminoácidos (σ=*20*).

[1] www.ebi.ac.uk/integr8/
[2] www.gutenberg.org

	DC		BMH		FJS		SBNDM		WML2	
m	T.E. (ms)	Rank	T.E. (ms)	Rank	T.E. (ms)	Rank	T.E. (ms)	Rank	T.E. (ms)	Rank
2	160	3	237	4	149	2	**144**	**1**	586	5
4	**94**	**1**	127	4	95	2	101	3	197	5
8	**58**	**1**	75	3	65	2	84	4	96	5
16	**44**	**1**	55	3	52	2	64	4	52	2
32	**31**	**1**	39	4	39	4	37	3	32	2
64	25	2	32	3	29	2	$>\omega$	?	**24**	**1**
128	**17**	**1**	21	4	19	3	$>\omega$	?	**17**	**1**

Tabela 3.2 - Resultados de desempenho comparado do algoritmo DC com os concorrentes para diversas categorias de padrões pesquisando linguagem natural ($\sigma=256$).

4 Discussão e conclusões

Pela apreciação das tabelas de resultados fica claro o domínio do algoritmo DC na pesquisa de proteínas e também nas pesquisas em linguagem natural, a sua vantagem é, em média, de cerca de 15% para o melhor concorrente, o WML2, que, se sublinha tratar-se de um algoritmo baseado em 2-gramas. O WML2 destaca-se nos padrões de maior dimensão mas, com uma flexibilidade reduzida pois deprecia-se na pesquisa padrões curtos. O SBNDM revela-se mais competitivo nos alfabetos de menor dimensão, em alfabetos maiores consegue equiparar-se ao BMH e perdendo claramente para o WML2. O FJS demonstra alguma consistência nos padrões mais curtos apresentando-se, no entanto pouco flexível quando os padrões aumentam de dimensão.

O algoritmo DC posiciona-se como algoritmo dominante para alfabetos em análise com $\sigma = 20$ (amino-ácidos) e $\sigma = 256$ (ASCII). O novo algoritmo é um modelo de eficiência e flexibilidade, adaptando-se com sucesso a qualquer alfabeto, dimensão ou composição de padrão.

O algoritmo BMH é um algoritmo altamente eficiente em pesquisas de padrões exactos em linguagem natural. Ocasionalmente surgiram algoritmos (tal como o FJS) que melhoraram marginalmente o desempenho patenteado pelo algoritmo BMH, porém nenhum algoritmo baseado em heurísticas conseguiu destacar-se significativamente da referência do BMH como agora acontece, com o novo algoritmo aqui apresentado, o algoritmo DC.

O algoritmo SBNDM é de outra categoria, baseado numa abordagem sustentada pelo *bit parallelism*, potenciada pelas rápidas operações de *bitwise*. Contudo, apresenta limitações no tamanho do padrão a pesquisar, estando limitado ao tamanho do número de bits da arquitectura do computador em que é executado, normalmente 32 ou 64 bits, que corresponderão a outros tantos caracteres. É possível contornar esta limitação segmentando o padrão e pesquisar cada segmento por separado, sendo que esta metodologia introduz *overhead* suplementar ao algoritmo pelo que a versão original preferiu ignorar os padrões maiores que o comprimento da palavra suportado pela arquitectura. O SBNDM é mais competitivo em alfabetos curtos, para $\sigma \geq 20$ apenas se assemelha ao DC em termos de desempenho quando o tamanho do padrão se situa em torno dos 32 caracteres, o que corresponde à maximização do aproveitamento do potencial do *bit parallelism*.

Referências

[1] S. Deusdado and P. Carvalho, "Efficient exact pattern-matching in proteomic sequences," in *Lecture Notes in Computer Science - Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living - IWPACBB'2009*, vol. 5518: Springer Berlin / Heidelberg, pp. 1178-1186, 2009.

[2] R. Horspool, "Practical fast searching in strings," *Software-Practice and Experience*, vol. 10, pp. 501-506, 1980.

[3] F. Franek, C. G. Jennings, and W. F. Smith, "A simple fast hybrid pattern-matching algorithm," *Lecture Notes in Computer Science, CPM2005*, vol. 3537, pp. 288-297, 2005.

[4] H. Peltola and J. Tarhio, "Alternative algorithms for bit-parallel string matching," *Lecture Notes in Computer Science, Proc. SPIRE '03*, vol. 2857, pp. 80-94, 2003.

[5] T. Lecroq, "Fast exact string matching algorithms," *Information Processing Letters*, vol. 102, pp. 229-235, 2007.

[6] R. A. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Commun. ACM*, vol. 35, pp. 74-82, 1992.

[7] D. Knuth, J. Morris, and V. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, pp. 323-350, 1977.

[8] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 762-772, 1977.

[9] D. Sunday, "A Very Fast Substring Search Algorithm," *Commun. of the ACM*, vol. 33, pp. 133-142, 1990.

[10] T. Lecroq, "Experimental results on string matching algorithms," *Software Practice & Experience*, vol. 25, pp. 727-765, 1995.