Valoració d'eficiència de diferents algorismes de cerca

Miquel Perello Nieto

7 de març de 2012

Resum

La finalitat d'aquesta pràctica es comprovar si avui en dia continua essent valida la gràfica dels algorismes mes eficients presentada a classe. Per tal motiu la feina pel proper dia serà avaluar, per a tres alfabet de mida m-1,m i m+1 caràcters (en aquest cas m = 3), els algorismes mes eficients (en temps d'execució) per a patrons de longitud k ($1 \leq k \leq \infty$). Els algorismes que tindrem en compte són els de Força bruta, Horspool, BNDM, BOM (tota la informació dels diferents algoritmes es pot veure a [1]).

A part es podrà observar en la pràctica els resultats de l'algoritme Karp-Rabin, el qual només tindrà caràcter informatiu, sense entrar en el calcul de resultats.

1 Introducció

Durant aquesta pràctica voldrem veure si els resultats esperats amb la gràfica 1 es correspon amb l'execució pràctica en un ordinador amb processador de 64 bits. Però primerament veurem els diferents algoritmes que utilitzarem en la pràctica, per veure quins son els seus costos teòrics, quines característiques te i una implementació de cadascun en C.

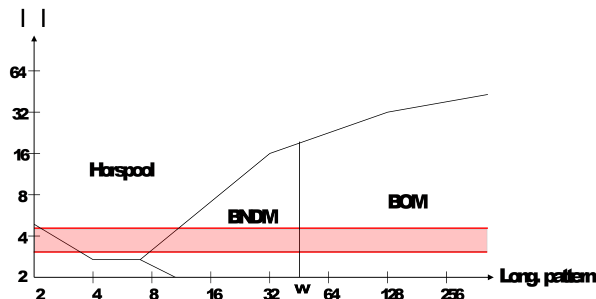

Figura 1: Gràfica de resultats que volem validar, amb marca en el rang dels nostres alfabets

1.1 Bruteforce

L'algoritme de força bruta consisteix en comprovar per totes les posicions del text entre 0 i n-m, si el principi del patró coincideix o no amb el text. Després de cada comprovació es shifta el patró una posició a la dreta.

Aquest algoritme no requereix cap fase de preprocés, i un espai extra constant adicionalment al patró i al text.

Durant la fase de cerca la comparació dels caràcters pot ser en qualsevol ordre. La complexitat temporal d'aquesta fase es O(mn). I el nombre de comparacions esperat es de 2n.

1.1.1 Característiques

- No hi ha fase de preprocés;
- Espai extra necessari constant;
- Sempre mou la finestra una posició a la dreta;
- La comparació es pot efectuar en qualsevol ordre;
- Fase de cerca en temps O(mn);
- Comparacions de text esperat de 2n;

1.1.2 Codi C

Codi C 1: `codi Bruteforce`

```
1  void BF(char *x, int m, char *y,
       int n) {
2     int i, j;
3
4     /* Searching */
```

```
5     for (j = 0; j <= n - m; ++j)
         {
6        for (i = 0; i < m && x[i]
            == y[i + j]; ++i);
7        if (i >= m)
8           OUTPUT(j);
9     }
10 }
```

1.2 Horspool

El shift per caràcter incorrecte de l'algoritme de Boyer-Moore no és molt eficient per alfabets petits, però quan els alfabets són grans comparats amb la longitud del patró és comporta molt bé.

Utilitzant-lo sol es produeix un algoritme molt eficient en la pràctica. El Horspool proposa utilitzar només el shift de caràcter incorrecte del caràcter més la la dreta de la finestra per calcular els shifts de l'algoritme de Boyer-Moore.

La fase de preprocés es en complexitat temporal O(m+) i espaial O()

La fase de cerca té el pitjor cas quadràtic però es pot demostrar que el nombre mig de comparacions en un text es entre 1/ i 2/(+1).

1.2.1 Característiques

- Simplificació de l'algoritme Boyer-Moore;
- Fàcil de implementar;
- Temps de preprocés O(m+) i espai O();
- Temps de fase de cerca en O(mn);
- El nombre de comparacions esperat per caràcter és de 1/ i 2/(+1).

1.2.2 Codi C

Codi C 2: `codi Horspool`

```
1  void HORSPOOL(char *x, int m,
       char *y, int n)
2  {
3  int j, bmBc[ASIZE];
4  char c;
5
6  /* Preprocessing */
7  preBmBc(x, m, bmBc);
8
9  /* Searching */
10 j = 0;
11 while (j <= n - m) {
12    c = y[j + m - 1];
13    if (x[m - 1] == c && memcmp(x,
          y + j, m - 1) == 0)
14       OUTPUT(j);
15    j += bmBc[c];
16    }
17 }
```

1.3 BNDM

L'algoritme BNDM (Backward Nondeterministic Dawg Matching) utilitza una taula B, la qual per cada caràcter guarda una mascara de bits. La màscara del caràcter es crea si i només si $xi = c$.

La fase de cerca es manté en una paraula $d = dm - 1 \ldots$d0, on la mida del patró m es inferior o igual a la mida de paraula de l'ordinador.

El bit di a la iteració k es posa a 1 si i només si $x[m-i..m-1-i+k] = y[j+m-k..j+m-1]$. A la primera iteració "d"s'inicialitza a 1m-1. La formula per actualitzar "d"segueix $d' = (d\&B[yj]) \ll 1)$.

Existeix una coincidència si i només si després de la iteració m el bit $dm - 1 = 1$.

Quan apareix una coincidència el prefix més llarg vist fins la coincidència determina el salt de finestra per la nova posició inicial.

1.3.1 Característiques

- Variant de l'algoritme Reverse Factor;
- Utilitza la simulació paral·lela de bit;
- Eficient si la mida del patró es inferior a la mida de paraula de l'ordinador.

1.3.2 Codi C

Codi C 3: `codi BNDM`

```
1  void BNDM(char *x, int m, char *
       y, int n) {
2  int B[ASIZE];
3  int i, j, s, d, last;
4  if (m > WORD_SIZE)
5  error("BNDM");
6
7  /* Pre processing */
```

```
8   memset(B,0,ASIZE*sizeof(int));
9   s=1;
10  for (i=m-1; i>=0; i--){
11  B[x[i]] |= s;
12  s <<= 1;
13  }
14
15  /* Searching phase */
16  j=0;
17  while (j <= n-m){
18  i=m-1; last=m;
19  d = ~0;
20  while (i>=0 && d!=0) {
21    d &= B[y[j+i]];
22    i--;
23    if (d != 0){
24  if (i >= 0)
25    last = i+1;
26  else
27    OUTPUT(j);
28     }
29     d <<= 1;
30   }
31   j += last;
32  }
33  }
```

1.4 BOM

Els algoritmes de tipus Boyer-Moore troben alguns sufixes del patró, però és possible trobar alguns prefixes del patró comprovant els caràcters de la finestra de dreta a esquerra i determinar la llargada del salt. Això és possible utilitzant el sufixe d'Oracle en el patró invertit. Aquesta estructura de dades es un Automat molt compacte que es capaç de reconèixer al menys tots els sufixes del patró i algunes altres paraules. La cerca de patrons amb l'algoritme utilitzant el patró invertit amb oracle es diu Backward Oracle Matching algorithm (les inicials del qual donen nom a aquest algoritme).

El sufix oracle d'una paraula "w"es un Automat Finit Determinista $O(w) = (Q, q0, T, E)$.

El llenguatge acceptat per $O(w)$ es tal que {u en * : existeix v en * tal que $w = vu$} en $L(O(w))$.

La fase de preprocés del Backward Oracle Matching algorithm consisteix en calcular el sufix d'oracle per l'invers del patró xR. A pesar del fet que aquest es capaç de reconèixer paraules que no son del patró, el sufix d'oracle pot ser utilitzat per fer una comparació de paraules tenint en compte que la única paraula de mida major o igual a m és reconeguda per el patró invers de oracle.

Durant la fase de cerca l'algoritme Backward Oracle Matching busca els caràcters de la finestra de dreta a esquerra amb l'autòmat O(xR) començant a l'estat q0. Aquest segueix fins que no hi ha una transició definida per el caràcter actual. En aquest moment la mida del prefix més llarg del patró, el qual és un sufix de la part comprovada es inferior que la mida del camí agafat de O(xR) des de l'estat q0 i l'ultim estat trobat. Sabent aquesta mida, es trivial calcular la mida del salt a executar.

L'algoritme Backward Oracle Matching té un cost quadràtic respecte el temps en el pitjor dels casos, però és optim en la mitjana. En aquesta el seu cost esdevé O(n.(logm) / m).

1.4.1 Característiques

- Versió de l'algoritme *Reverse Factor* utilitzant el sufix d'oracle de xR en comptes de l'automat de xR.
- Ràpid en entorns amb mides de patró molt llargs i petits alfabets;
- Fase de preprocés en complexitat temporal i espaial O(m);
- Fase de cerca en complexitat temporal O(mn);
- Optim en la mitja;

1.4.2 Codi C

Només la part externa de les transicions d'oracle s'emmagatzema en una llista de enllaços (un per estat). Els noms de les transicions no s'emmagatzemen però són computades per la paraula x.

Codi C 4: codi BOM

```
1  #define FALSE      0
2  #define TRUE       1
3
4  int getTransition(char *x, int p
       , List L[], char c) {
5      List cell;
6
7      if (p > 0 && x[p - 1] == c)
8         return(p - 1);
9      else {
```

```
10        cell = L[p];
11        while (cell != NULL)
12           if (x[cell->element] == c)
13              return(cell->element);
14           else
15              cell = cell->next;
16        return(UNDEFINED);
17     }
18  }
19
20
21  void setTransition(int p, int q, List L[]) {
22     List cell;
23
24     cell = (List)malloc(sizeof(struct _cell));
25     if (cell == NULL)
26        error("BOM/setTransition");
27     cell->element = q;
28     cell->next = L[p];
29     L[p] = cell;
30  }
31
32
33  void oracle(char *x, int m, char T[], List L[]) {
34     int i, p, q;
35     int S[XSIZE + 1];
36     char c;
37
38     S[m] = m + 1;
39     for (i = m; i > 0; --i) {
40        c = x[i - 1];
41        p = S[i];
42        while (p <= m &&
43               (q = getTransition(x, p, L, c)) ==
44               UNDEFINED) {
45           setTransition(p, i - 1, L);
46           p = S[p];
47        }
48        S[i - 1] = (p == m + 1 ? m : q);
49     }
50     p = 0;
51     while (p <= m) {
52        T[p] = TRUE;
53        p = S[p];
54     }
55  }
56
57
58  void BOM(char *x, int m, char *y, int n) {
59     char T[XSIZE + 1];
60     List L[XSIZE + 1];
61     int i, j, p, period, q, shift;
62
63     /* Preprocessing */
64     memset(L, NULL, (m + 1)*sizeof(List));
65     memset(T, FALSE, (m + 1)*sizeof(char));
66     oracle(x, m, T, L);
67
68     /* Searching */
69     j = 0;
70     while (j <= n - m) {
71        i = m - 1;
72        p = m;
73        shift = m;
74        while (i + j >= 0 &&
75               (q = getTransition(x, p, L, y[i + j])) !=
76               UNDEFINED) {
77           p = q;
78           if (T[p] == TRUE) {
79              period = shift;
80              shift = i;
81           }
82           --i;
83        }
84        if (i < 0) {
85           OUTPUT(j);
86           shift = period;
87        }
88        j += shift;
89     }
90  }
```

1.5 Karp-Rabin

Hashejar ens permet de manera simple evitar un nombre de comparacions de caràcter quadràtic en la majoria de situacions. En comptes de comprovar en cada posició del text si el patró coincideix, sembla més eficient comprovar només el contingut de les finestres que més *s'assemblen* al patró. Per comprovar la semblança de les dues paraules s'utilitza una funció de hash.

La fase de preprocés de l'algoritme Karp-Rabin consisteix en calcular el $hash(x)$. Aquest es pot resoldre amb un espai constant i un temps $O(m)$.

Durant la fase de cerca es suficient amb comparar el hash(x) amb el $hash(y[j..j+m-1]) for 0 j < n-m$. Si es troba alguna semblança es necessari comprovar la igualtat de $x = y[j..j+m-1]$ caràcter per caràcter.

La complexitat temporal de la fase de cerca es $O(mn)$. El nombre esperat de comparacions per caracter es de $O(n+m)$.

1.5.1 Característiques

- Utilitza una funció de hash.
- Fase de preprocés en temps O(m) i espai constant.
- Fase de cerca en temps O(mn).
- Temps d'execució esperat O(n+m).

1.5.2 Codi C

Codi C 5: `codi Karp-Rabin`

```
1  #define REHASH(a, b, h) ((((h) -
      (a)*d) << 1) + (b))
2
3  void KR(char *x, int m, char *y,
      int n) {
4     int d, hx, hy, i, j;
5
6     /* Preprocessing */
7     /* computes d = 2^(m-1) with
8        the left-shift operator */
9     for (d = i = 1; i < m; ++i)
10       d = (d<<1);
11
12    for (hy = hx = i = 0; i < m;
         ++i) {
13       hx = ((hx<<1) + x[i]);
14       hy = ((hy<<1) + y[i]);
15    }
16
17    /* Searching */
18    j = 0;
19    while (j <= n-m) {
20       if (hx == hy && memcmp(x,
            y + j, m) == 0)
21          OUTPUT(j);
22       hy = REHASH(y[j], y[j + m
            ], hy);
23       ++j;
24    }
25
26 }
```

2 Resultats

En aquesta secció veurem els resultats dels temps de cerca de cada un dels diferents algoritmes a analitzar. Es podrà veure de forma visual mitjançant plots generals. I en els casos en els que no es pugui apreciar la diferencia de millora entre un algoritme i l'altre, es podrà veure un cas d'execució particular per fer desempatar els dos algoritmes, i una gràfica en el que sí es podrà observar diferencia.

Finalment per cada una de les mides d'alfabet es presenta una petita gràfica lineal amb els rangs de mida de patró ideals per a cada algoritme.

2.1 Alfabet de mida 3

Per comprovar els resultats en l'execució dels diferents algoritmes en la cerca de patrons de l'alfabet de mida 3 s'ha realitzat primerament unes proves d'execució amb les següents característiques:

mida del text	1,4GB
mida patrons	$[2^0 \ldots 2^9]$
repeticions per patró	4

Els resultats d'aquestes execucions donen una visió molt clara en la distribució dels diferents algoritmes (figura 2), però existeix un punt de conflicte entre el Horspool i el BNDM en el qual no es pot apreciar amb les repeticions efectuades el punt a partir del qual un supera a l'altre.

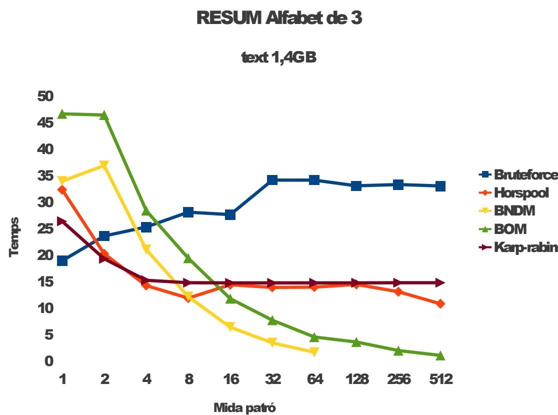

Figura 2: Grafica del temps per mida de patró per diferents algoritmes amb un alfabet de mida 3

Per aquesta raó s'ha hagut d'efectuar una prova més concreta i focalitzada en el punt de conflicte, amb més repeticions per mida de patró, així que les següents proves han seguit el següent patró:

mida del text	2GB
mida patrons	[5...9]
repeticions per patró	10

Per tant després de l'execució d'aquestes proves es pot veure clarament el punt en el que el BNDM supera en temps d'execució a l'algoritme de Horspool (figura 2).

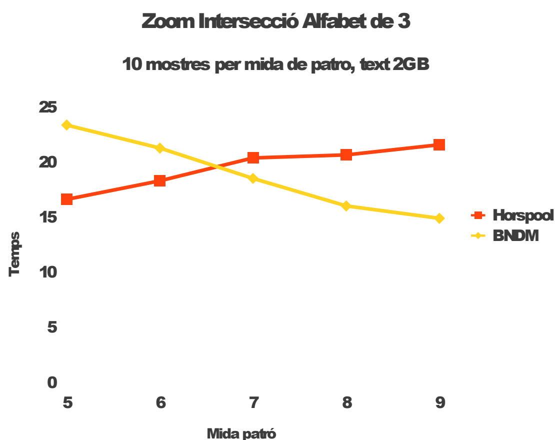

Figura 3: Zoom de conflicte entre Horspool i BNDM

Gracies a aquesta aproximació podem apreciar que amb patrons de mida sis l'algoritme Horspool segueix sent millor, en canvi en patrons de mida set el millor algoritme es millor el BNDM.

Tot seguit es pot veure un resum visual en el que es poden apreciar els intervals de mida de patró amb l'algoritme que hauríem d'utilitzar per trigar el mínim temps en analitzar el text (figura 4).

Alfabet de cardinalitat 3

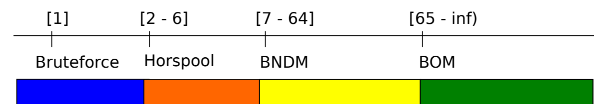

Figura 4: Valoració per mida del patró per un alfabet de cardinalitat 3

2.2 Alfabet de mida 4

Per comprovar els resultats en l'execució dels diferents algoritmes en la cerca de patrons de l'alfabet de mida 4 s'ha realitzat primerament unes proves d'execució amb les següents característiques:

mida del text	1,4GB
mida patrons	$[2^0 \ldots 2^9]$
repeticions per patró	4

Els resultats d'aquestes execucions donen una visió molt clara en la distribució dels diferents algoritmes (figura 5), però igual que en el cas de l'alfabet de mida tres, en l'alfabet de mida quatre també existeix un punt de conflicte entre el Horspool i el BNDM en el qual no es pot apreciar amb les repeticions efectuades el punt a partir del qual un supera a l'altre.

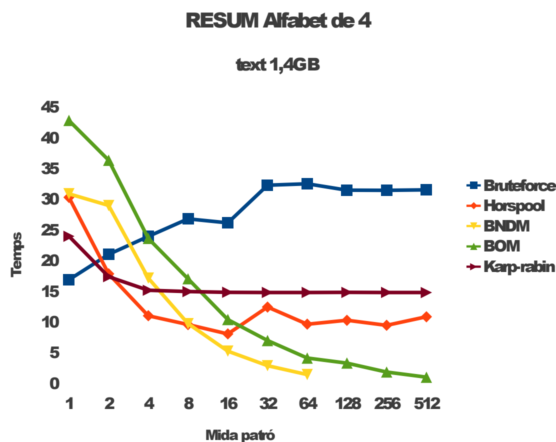

Figura 5: Grafica del temps per mida de patró per diferents algoritmes amb un alfabet de mida 4

Per aquesta raó s'ha hagut d'efectuar una prova més concreta i focalitzada en el punt de conflicte, amb més repeticions per mida de patró, així que les següents proves han seguit el següent patró:

mida del text	2GB
mida patrons	[5...9]
repeticions per patró	10

Per tant després de l'execució d'aquestes proves es pot veure clarament el punt en el que el BNDM supera en temps d'execució a l'algoritme de Horspool (figura 5).

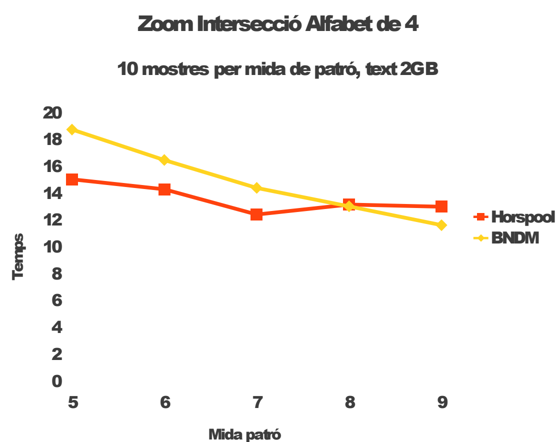

Figura 6: Zoom de conflicte entre Horspool i BNDM

Gracies a aquesta aproximació podem apreciar que amb patrons de mida sis l'algoritme Horspool segueix sent millor, en canvi en patrons de mida set el millor algoritme es millor el BNDM.

Tot seguit es pot veure un resum visual en el que es poden apreciar els intervals de mida de patró amb l'algoritme que hauríem d'utilitzar per trigar el mínim temps en analitzar el text (figura 7).

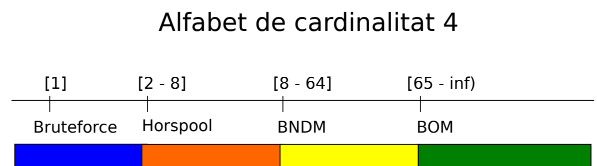

Figura 7: Valoració per mida del patró per un alfabet de cardinalitat 4

2.3 Alfabet de mida 5

Per comprovar els resultats en l'execució dels diferents algoritmes en la cerca de patrons de l'alfabet de mida 5 s'ha realitzat primerament unes proves d'execució amb les següents característiques:

mida del text	1,4GB
mida patrons	$[2^0 \ldots 2^9]$
repeticions per patró	4

Els resultats d'aquestes execucions donen una visió molt clara en la distribució dels diferents algoritmes (figura 8), però igual que en el cas de l'alfabet de mida tres i quatre, en l'alfabet de mida cinc també existeix un punt de conflicte entre el Horspool i el BNDM en el qual no es pot apreciar amb les repeticions efectuades el punt a partir del qual un supera a l'altre.

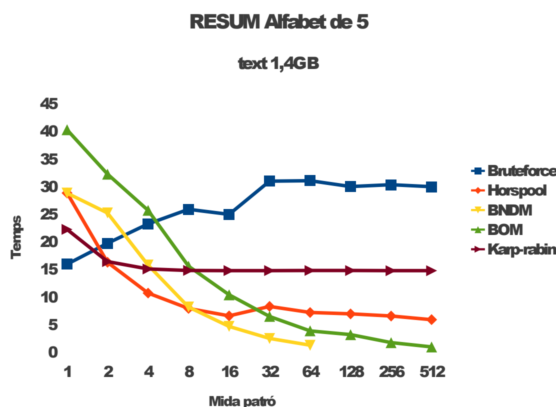

Figura 8: Grafica del temps per mida de patró per diferents algoritmes amb un alfabet de mida 5

Per aquesta raó s'ha hagut d'efectuar una prova més concreta i focalitzada en el punt de conflicte, amb més repeticions per mida de patró, així que les següents proves han seguit el següent patró:

mida del text	2GB
mida patrons	[5...13]
repeticions per patró	20

Per tant després de l'execució d'aquestes proves es pot veure clarament el punt en el que el BNDM supera en temps d'execució a l'algoritme de Horspool (figura 8).

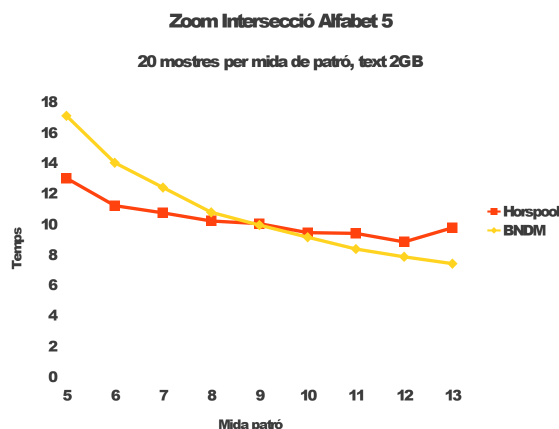

Figura 9: Zoom de conflicte entre Horspool i BNDM

Gracies a aquesta aproximació podem apreciar que amb patrons de mida sis l'algoritme Horspool segueix sent millor, en canvi en patrons de mida set el millor algoritme es millor el BNDM.

Tot seguit es pot veure un resum visual en el que es poden apreciar els intervals de mida de patró amb l'algoritme que hauríem d'utilitzar per trigar el mínim temps en analitzar el text (figura 10).

Alfabet de cardinalitat 5

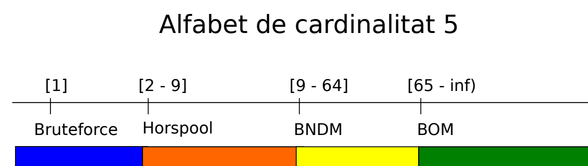

Figura 10: Valoració per mida del patró per un alfabet de cardinalitat 5

3 Conclusions

Un cop finalitzades totes les proves ja hem pogut contrastar els resultats, i gràcies a les gràfiques generades es pot observar de manera senzilla si els resultats han estat propers als esperats, així que detallem el que podem comprovar amb cada punt d'intersecció de la gràfica original. Ens centrarem exclusivament en la línia vermella que creua la gràfica 1 amb mida d'alfabet compresa entre tres i cinc.

En la gràfica es podia observar que amb alfabets d'aquest rang existeix una petita àrea en els patrons de longitud inferior a tres en la qual el Horspool es comporta pitjor que altres algoritmes, en el nostre cas es compleix amb l'algoritme de força bruta, justament amb patrons de un caràcter.

Després es pot observar un àrea que pertany al Horspool que arriba fins als patrons de mida nou. Aquí es torna a veure que la similitud amb els nostres resultats es molt alta.

Tot seguit el BNDM es converteix en el millor algoritme fins arribar a la mida de patró igual a la mida de la paraula de l'ordinador; en el nostre cas utilitzàvem un ordinador amb processador de 64 bits, i per aquest motiu a partir d'aquesta mida de patró hem descartat els següents resultats.

Per ultim amb patrons de mida superior l'algoritme BOM es el més eficient tant en la gràfica anterior, com en els resultats d'aquesta pràctica.

Per tant amb la realització d'aquesta pràctica finalment hem pogut observar que els resultats de l'execució dels diferents algoritmes de cerca segueixen la relació que existia en la gràfica que volíem comprovar en els casos d'alfabets de mida tres, quatre i cinc.

A Codis

Codi C 6: Generació de text aleatori

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <fcntl.h>
5 #include <sys/types.h>
6
7 #define EOS '\0'
8
```

```c
char alfabet[7] = {'T', 'G', 'A', 'C', 'B', 'D', 'E'};

int main(int argc, char *argv[])
{
  char filename[256];
  int size;
  int alf_size;
  int fd;
  unsigned long i;

  srand ( time(NULL) );

  size = atoi(argv[1]);
  alf_size = atoi(argv[2]);


  for (i = 0; i < size*1000000; i++)
  {
    printf("%c",alfabet[(rand() % alf_size)]);
    if ((time(NULL)%(rand()+1)) == 0)
    {
      srand ( time(NULL) );
    }
  }
  printf(EOS);
}
```

Codi C 7: Main de cerca en text

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "list.h"
#define EOS    '\0'
#define ASIZE 256
#define FALSE 0
#define TRUE  1

char alfabet[7] = {'T', 'G', 'A', 'C', 'B', 'D', 'E'};

unsigned long int readfile(char *filename, char *text[])
{
  FILE *fin;
  unsigned long int len;
  int i = 0;
  char c;

  if (!(fin = fopen(filename, "r"))) {
    fprintf (stderr, "I can't open the file: %s\n", filename);
    exit(1);
  }
  // llegim el nombre de caracters que hi ha al fitxer
  fseek(fin, 0, SEEK_END);
  len = ftell(fin);
  fseek(fin, 0, SEEK_SET);
  // demanem memoria per la variable text
  *text = malloc(len + 1);
        // bucle per llegir el fitxer
  while ((c = (fgetc(fin))) != EOF) {
    if ((c=='a')||(c=='c')||(c=='g')||(c=='t')||(c=='b')||(c=='d')||(c=='e')||(c=='A')||(c=='C')||(c=='G')||(c=='T')||(c=='B')||(c=='D')||(c=='E')){
      (*text)[i] = c;
      //printf("text[%d] = %c\n", i, (*text)[i]);
      i++;
    }
  }
  fclose(fin);
  return i;
}

int main(int argc, char *argv[])
{
    char pattern[257];
  char filename[256];
  char *text;
  unsigned long int  n, i, j, k;
  int pattern_size, alf_size;
  unsigned long int trobats;
  clock_t initemps;
  clock_t endtemps;

```

```
52
53     if (argc != 4) exit(0);
54
55     alf_size = atoi(argv[2]);
56     sprintf(filename,"%s",argv[1])
           ;
57
58     n = readfile(filename, &text);
59     printf("n=%d\n",n);
60     printf("size \t pattern \t
           trobats \t segons \n");
61     srand(0);
62     for (k = 1; k <= 512; k = k*2)
63     {
64       pattern_size = k;
65       for (i = 0; i < 4; i++)
66       {
67         for (j = 0; j <
               pattern_size; j++)
68         {
69           pattern[j] = alfabet[
                 rand()%alf_size];
70         }
71         pattern[pattern_size] =
               EOS;
72         //printf("%s\t", pattern);
73
74         initemps=clock();
75         //trobats = BF(pattern,
               strlen(pattern),text,n)
               ;
76         trobats = HORSPOOL(pattern
               ,strlen(pattern),text,n
               );
77         //trobats = BNDM(pattern,
               strlen(pattern),text,n)
               ;
78         //XSIZE = pattern_size;
79         //trobats = BOM(pattern,
               strlen(pattern),text,n)
               ;
80         //trobats = KR(pattern,
               strlen(pattern),text,n)
               ;
81         endtemps=clock();
82         printf( "%d\t%s\t%d\t%f\n"
               , pattern_size, pattern
               , trobats, (endtemps-
               initemps)/(double)
               CLOCKS_PER_SEC );
83       }
84     }
85     free(text);
86           return 0;
87 }
```

Referències

[1] "EXACT STRING MATCHING ALGORITHMS", Christian Charras - Thierry Lecroq, `http://www-igm.univ-mlv.fr/~lecroq/string/index.html`, Faculté des Sciences et des Techniques