BIOINFORMATICS

Vol. 18 no. 8 2002
Pages 1102–1108

Simplifying amino acid alphabets by means of a branch and bound algorithm and substitution matrices

Nicola Cannata, Stefano Toppo, Chiara Romualdi and Giorgio Valle

CRIBI Biotechnology Centre, Università di Padova, via Ugo Bassi 58/B, 35131 Padova, Italy

Received on September 6, 2001; revised on January 8, 2002; accepted on February 25, 2002

ABSTRACT

Motivation: Protein and DNA are generally represented by sequences of letters. In a number of circumstances simplified alphabets (where one or more letters would be represented by the same symbol) have proved their potential utility in several fields of bioinformatics including searching for patterns occurring at an unexpected rate, studying protein folding and finding consensus sequences in multiple alignments. The main issue addressed in this paper is the possibility of finding a general approach that would allow an exhaustive analysis of all the possible simplified alphabets, using substitution matrices like PAM and BLOSUM as a measure for scoring.

Results: The computational approach presented in this paper has led to a computer program called AlphaSimp (Alphabet Simplifier) that can perform an exhaustive analysis of the possible simplified amino acid alphabets, using a branch and bound algorithm together with standard or user-defined substitution matrices. The program returns a ranked list of the highest-scoring simplified alphabets. When the extent of the simplification is limited and the simplified alphabets are maintained above ten symbols the program is able to complete the analysis in minutes or even seconds on a personal computer. However, the performance becomes worse, taking up to several hours, for highly simplified alphabets.

Availability: AlphaSimp and other accessory programs are available at http://bioinformatics.cribi.unipd.it/alphasimp.

Contact: giorgio.valle@unipd.it

INTRODUCTION

DNA sequences are generally written using the standard alphabet of four symbols (A, C, G and T) representing the four bases, while proteins are represented by 20 symbols, one for each amino acid. However, for particular tasks, the use of simplified alphabets may be more convenient. A simplified alphabet has a reduced set of symbols, for instance we could define a two-symbol alphabet for DNA sequences where 'R' stands for purines and 'Y' for pyrimidines, or a two-symbol alphabet for proteins where 'H' stands for hydrophobic and 'P' for polar amino acid. In general terms, any complex alphabet can be simplified by grouping two or more letters of the original alphabet together and by representing them with a new symbol.

Simplified alphabets have already been applied in several fields of bioinformatics. For instance, the problem of finding consensus sequences in multiple alignments has been approached by means of different strategies including the implementation of simplified alphabets (Karlin and Ghandour, 1985; Sagot *et al.*, 1997). Similarly, simplified alphabets are used to study protein folding and in particular to investigate the problem of the minimal number of residue types required to properly fold a protein. Although a lot of work in this field has been done with the two-letter polar/hydrophobic alphabet (Dill, 1985; Kamtekar *et al.*, 1993), different reports suggest that wider alphabets would be more suitable (Riddle *et al.*, 1997; Wang and Wang, 1999; Murphy *et al.*, 2000).

Searching unexpected patterns using simplified alphabets

The search for patterns occurring at an unexpected rate is an established strategy for the identification of functional constraints in DNA and protein sequences. Several methods have been developed for the identification of such patterns (for a review see Brazma *et al.*, 1998). However, these strategies are less effective on patterns with degenerated positions, such as the DNA pattern $(RRY)_n$ that is known to be preferentially found in DNA coding sequences (Shepherd, 1981), which would be hardly detectable using the standard four-letter alphabet.

The same problem applies at protein level where some amino acids can often be replaced by others without any significant functional alteration. Also in these cases the identification of unexpected degenerated patterns could

be much easier with simplified alphabets; but since we do not know *a priori* the unexpected pattern to be searched we cannot design a specific simplified alphabet that could make the identification easier. A possible approach for searching unexpected degenerated patterns could be to re-write systematically a sequence using different simplified alphabets and searching all the resulting simplified sequences for unexpected patterns. However, to pursue such an approach we need a tool for producing and selecting simplified alphabets in a systematic way.

Thus, the question that we are addressing in this paper is how many simplified alphabets can be produced from a complex alphabet such as the 20-letter amino acid alphabet, and how can a general strategy be designed to score and select the resulting simplified alphabets.

SYSTEM AND METHODS

Simplified alphabets and set partitioning

In computational sciences the problem of simplifying alphabets can be related to the topic of set partitioning. In general terms a partition is an arrangement of a set into disjoint (non overlapping) subsets that completely cover the entire initial set. In this respect, a simplified alphabet of amino acids can be seen as one of the possible partitions of the original set of twenty letters. For instance, Serine–Threonine could belong to one subset and Leucine-Isoleucine to another subset, leaving the remaining 16 amino acids alone. The resulting partition would include 18 subsets and the corresponding simplified alphabet could be represented by 18 symbols, two of which representing multiple amino acids. More extreme partitions could have only two subsets, for instance polar / hydrophobic.

Stirling numbers and set partitioning

The first question to answer is how many different partitions can be obtained from a n-element set, here generically indicated as $\{1, 2, \ldots, n\}$. For example, given the set {1,2,3} we can have five different partitions that can be written in a simplified way with the following representation: {123}, {1,23}, {12,3}, {13,2}, {1,2,3}. To understand the problem, we should consider that the number of possible partitions results from the sum of those with one single subset, plus those with two subsets, and so on up to the partition with n subsets, that is $1 + 3 + 1 = 5$ in the above example (see also the third line of the diagram in Figure 1).

It can be easily demonstrated (see Figure 1) that the number of partitions with k subsets from a n-element set can be recursively calculated by the following relation, giving $S(n, 1) = S(n, n) = 1$ as starting conditions:

$$S(n, k) = kS(n-1, k) + S(n-1, k-1) \text{ with } 2 \leqslant k \leqslant n-1$$

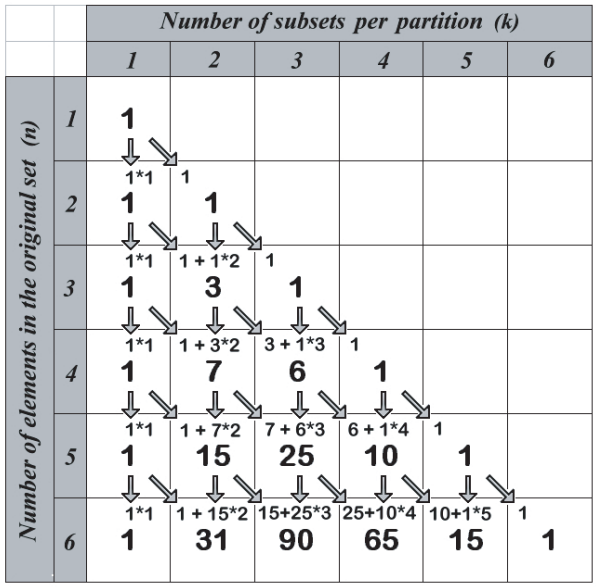

Fig. 1. The number of all the n-elements partitions with k subsets is known as Stirling number or $S(n, k)$, and is reported in bold at the centre of the cells of the figure. The value of each $S(n, k)$ can be calculated from the previous $(n - 1)$ line because all the $S(n, k)$ partitions can either derive by adding the new element independently within each of the k subsets of each of the $S(n-1, k)$ partitions or by adding it as a separate subset to the $S(n - 1, k - 1)$ partition (see also Figure 3). As a result we have $S(n, k) = kS(n - 1, k) + S(n - 1, k - 1)$. The sum of the $S(n, k)$ values of the n line is the total number of partitions of n elements.

This produces a series of numbers known as Stirling numbers, which are represented in bold at the centre of the cells of Figure 1. For any given n, the values of k range between 1 and n and the sum of the n resulting $S(n, k)$ values gives the total number of possible partitions of a set of n elements:

$$P(n) = \sum_{k=1}^{n} S(n, k)$$

Figure 2a shows the total number of possible partitions for the first 20 values of n, while Figure 2b shows the twenty individual values of $S(n, k)$ for $n = 20$. The huge number of possible partitions can be easily appreciated from Figure 2.

The aim of the work described in this paper was to develop an algorithm that, once a scoring system for the partitions was defined, would allow to rank and select the best partitions in an exhaustive and efficient way. The evaluation of set partitioning is not an easy computational task, being a well known NP-hard problem (Balas and

(a)

n values (no. of elements)	Number of possible partitions
1	1
2	2
3	5
4	15
5	52
6	203
7	877
8	4,140
9	21,147
10	115,975
11	678,570
12	4,213,597
13	27,644,437
14	190,899,322
15	1,382,958,545
16	10,480,142,147
17	82,864,869,804
18	682,076,806,159
19	5,832,742,205,057
20	51,724,158,235,372

(b)

k values *(no. of subsets)*	S(20, k) values (*partitions with k subsets*)
1	1
2	524,287
3	580,606,446
4	45,232,115,901
5	749,206,090,500
6	4,306,078,895,384
7	11,143,554,045,652
8	15,170,932,662,679
9	12,011,282,644,725
10	5,917,584,964,655
11	1,900,842,429,486
12	411,016,633,391
13	61,068,660,380
14	6,302,524,580
15	452,329,200
16	22,350,954
17	741,285
18	15,675
19	190
20	1

Fig. 2. The table on the left shows the total number of possible partitions of n-element sets, for values of n between 1 and 20, calculated as described in the text and in Figure 1. On the right are shown the partitions of 20 elements, divided according to the number of subsets. Therefore there are more than 51×10^{12} ways to make simplified amino acid alphabets and more than 15×10^{12} simplified alphabets where the twenty amino acids are represented by a reduced set of eight symbols.

Padberg, 1976). Given some scoring parameters, the task of searching for partitions above a given threshold is generally addressed by means of heuristic methods. In this paper we describe a simple algorithm based on a branch and bound approach, that under a variety of conditions allows a very rapid and exhaustive computation of partition weighting. We also describe an implementation of this algorithm to generate and select simplified alphabets of amino acids, on the basis of the substitution matrices such as PAM (Schwartz and Dayhoff, 1978) and BLOSUM (Henikoff and Henikoff, 1992) that are commonly used in computational biology.

ALGORITHM

A recursive algorithm to explore the set of partitions

As discussed in Figure 1, the number of possible partitions of a n-element set can be calculated recursively. Similarly, each individual partition can be generated using a recursive approach, as can be worked out from Figure 3 where partitions are shown as nodes within a graph structure.

Each node is linked to the nodes of the next order and

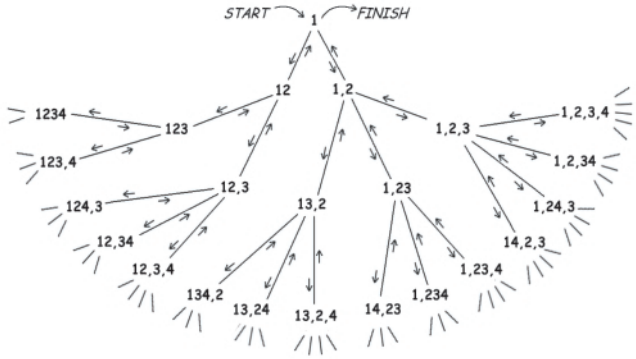

Fig. 3. Each individual partition is shown as a node within a graph structure. As a new element is added, each partition of k subsets will generate $k + 1$ new partitions, following a very simple rule (described in the text). Given any partition it is possible to generate all the partitions of order $n + 1$. Thus, we could develop a recursive procedure that is at the base of the branch and bound approach described in the text. The resulting algorithm will explore the graph following the arrows, but will enter only the branches that can lead to useful partitions.

every time we add an element each partition will generate a series of new partitions following a precise rule, that applies to all the nodes of the graph:

ADDING PROCEDURE
(add a new element to an individual partition)

a) Generate new partitions by adding the new element into each individual subset, thus generating as many partitions as the number of subsets

b) Generate a further partition by adding the new element to the partition as a single element subset

Every time a new partition is generated, the above adding procedure call itself in a recursive way. The only required control is for the maximal number of elements that should be verified every time we enter the adding procedure. Thus, every time the current number of elements reaches n, a full n-element partition has been completed and the program will return to a lower order continuing on the next branch of the graph (see Figure 3). For $n = 4$, the first steps of path will be as follows: {1}, {12}, {123}, {1234}, {123}, {123,4}, {12}, {12,3}, {124,3},... (where the partitions with the full set of four elements are underlined).

By means of this approach we can generate all the possible partitions of a n-element set. A simple implementation of this algorithm is shown in the program listed in Figure 4, written in the Perl language.

```
#!/usr/bin/perl
# program name:  partition

# split argument string into individual elements
@elem=split(//, $ARGV[0]);
$n=@elem; # stores number of elements in $n
die"Usage: partition abcdefg\n" unless $n>0;
&add ("", 0);
exit(0);

# This subroutine adds the m-th element
# to the current partition stored in $s
sub add() {
      my ($s, $m) = @_;
      my ($tmp, $i, $j, @c);
      # check for end of elements
      if ($m > $n) {
            # partition $s available for analysis
            print "$s\n";
      }
      else { # last element not yet reached
            # split partition on commas
            @c = split(/,/, $s);
            # add the new element into each subset
            for ($i=0; $i<@c; $i++) {
                  $tmp="";
                  for ($j=0; $j<@c; $j++) {
                        $tmp .= $c[$j];
                        $tmp .= $elem[$m] if($j==$i);
                        $tmp .= "," if($j<(@c-1));
                  }
                  # proceed recursion at m+1 level
                  &add ($tmp, $m+1);
            }
            # add new element to current partition
            $s .= "," if($s);
            # proceed recursion at m+1 level
            &add ("$s$elem[$m]", $m+1);
      }
}
```

Fig. 4. This simple program is written in *Perl* and implements the recursive algorithm described in the text and in Figure 3. It will produce a list of all the partitions of a given string. Within a partition the subsets are separated by commas.

It can be easily demonstrated that all the partitions of a n-element set can be generated by this way. Starting from an empty set, the first partition is generated by the step b of the adding procedure and is represented by the symbol '1' at the top of the graph in Figure 3. It is obvious that this is the only possible partition of a single element set. Assuming that we know all the partitions of a given level n (set of n elements), we can apply the above adding procedure to each of them thus generating all the partitions of level $n+1$. In fact, all the newly generated subsets will still be disjoint because the new element can be inserted only in one subset; furthermore, since there will always be one subset containing the new element we satisfy the conditions (disjoint and complete coverage) to have valid partitions. Finally, all the possible partitions are obtained because we add the new element in all possible ways. By induction we can extend the validity of this approach to any value of n.

Branch and bound using substitution matrices

The recursive algorithm described in the previous section easily allows the creation of all the possible partitions of a set; furthermore, it provides the basis for selecting partitions in an efficient way, using a branch and bound approach. In fact, given a scoring method for assessing simplified alphabets, we can define the cost of a simplified alphabet as a score difference from the fully implemented alphabet. Then, if we set a maximal cost, we can start to build partitions following the different branches of the graph shown in Figure 3. As we add a new element to a branch, we can verify the cost that has been so far accumulated in that branch and if the cost exceeds the fixed limit then the branch can be abandoned as it will not be able to yield suitable partitions. Thus we can limit our search only to the regions of the graph that can actually produce valid partitions.

Substitution matrices such as PAM (Schwartz and Dayhoff, 1978) and BLOSUM (Henikoff and Henikoff, 1992) are widely used for scoring alignments of protein sequences. In practical terms the score of an alignment can be calculated as a sum of all the substitution values defined in the matrix. If we use a simplified alphabet, then the score of the alignment of two sequences will be less accurate. However, some simplified alphabets will be better than others, depending on the degree and kind of simplification.

In our implementation the evaluation of a simplified alphabet reflects the approximation with which we can assign a score to the alignment of two identical sequences of twenty different amino acids.

Given a substitution matrix, we assume that the full alphabet of 20 amino acids has a score equal to the sum of the 20 exact matches, that is the score of the alignment of two identical sequences of 20 different amino acids. If we generate a new symbol, for instance {AR} representing either 'A' or 'R', then when we align {AR}:{AR} we should assume four possibilities: A:A, A:R, R:A and R:R. Without any further constraint, each possibility has a 25% chance to be correct. Therefore, assuming that S_{AA}, S_{AR} and S_{RR} are the scores for the corresponding amino acid pairs, then for a {AR}:{AR} match we should calculate an average score of $(S_{AA}+2S_{AR}+S_{RR})/4$, against an average score of $(S_{AA}+S_{RR})/2$ that we would obtain with two identical sequences using the original alphabet.

More generally, given a symbol representing a list of j amino acids, its score for a self-match will be the average value of the scores found at the corresponding intersections in the substitution matrix (see Figure 5). The cost for merging the j amino acids in one symbol will be calculated as the difference of the sum of the corresponding j scores taken from the fully implemented alphabet minus the sum of the analogous j scores of the simplified symbol. For instance, the cost for merging {I,

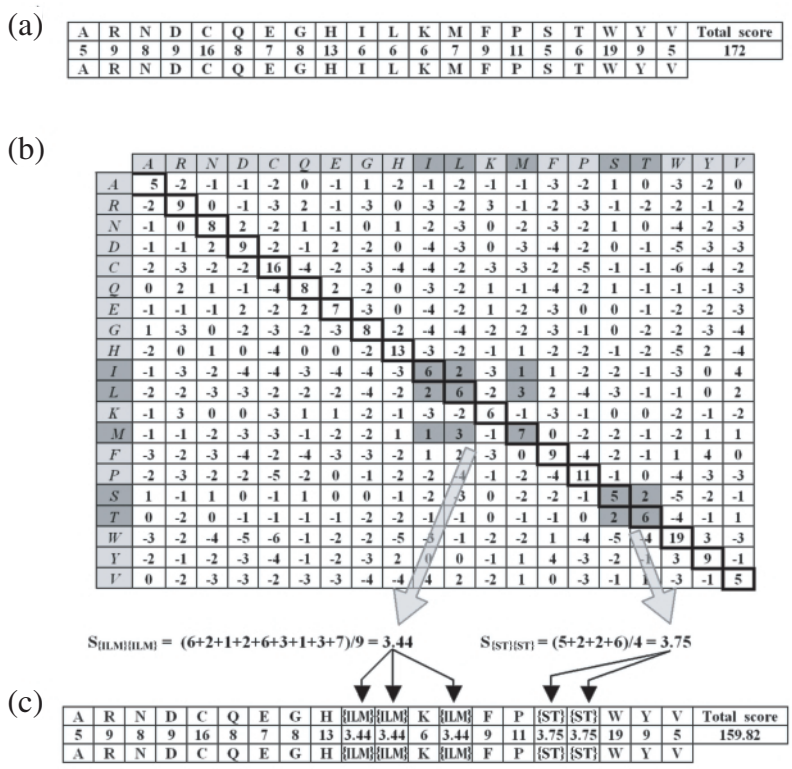

Fig. 5. (a) Shows the self-aligned sequence of 20 different amino acids, represented by the full alphabet, giving a score of 172 calculated as the sum of the 20 perfect matches found in the BLOSUM40 substitution matrix (panel b) (c) Shows the same alignment obtained with a simplified alphabet in which (I, L, M) and (S, T) are grouped in two subsets. The cost of each composite symbol and the cost of the entire simplified alphabet can be calculated as the differences from the original alphabet.

L, M} in the example of Figure 5 will be $(6 + 6 + 7) - (3.44+3.44+3.44) = 8.68$. The global cost of a simplified alphabet will result from the sum of the costs of the new symbols. Figure 5 shows an example of this calculation.

More sophisticated scoring procedures could be designed to take into consideration the frequency of each amino acid in normal protein sequences or the approximation deriving from scoring mismatches with simplified alphabets. However, the procedure proposed in this paper is very simple to understand and to implement, yielding simplified alphabets suitable for most applications.

IMPLEMENTATION

A fully functional program based on the above strategies has been developed using the C programming language and has been named AlphaSimp for 'Alphabet simplifier'. The program is freely available at the web site: {http://bioinformatics.cribi.unipd.it/alphasimp}. In this section of the paper we will refer to alphabets and symbols rather than partitions and subsets, but we would like to emphasize that an alphabet is equivalent to a partition and a symbol to a subset.

Five arguments must be passed to the AlphaSimp program. The first is the name of the file containing a valid substitution matrix in the standard format in which the first line contains the list of one-letter symbols and the following lines the substitution values. In the current implementation the original alphabet can be any size between 1 and 20 symbols that must be encoded by one single character and the values of the matrix must be integers in the range from -127 to $+128$.

The evaluation of alphabets is done on the basis of four different parameters: (1) maximal number of symbols (i.e. maximal number of subsets per partition); (2) minimal number of symbols; (3) maximal cost per subset; (4) maximal cost for the entire simplified alphabet. These four conditions are checked every time the *add* procedure is entered. Thus, the branch and bound strategy can be exploited not only to discard branches that exceed the threshold for the maximal cost of a simplified alphabet, but also for the conditions defined by the other parameters.

Figure 6 displays some results obtained with AlphaSimp using the BLOSUM40 substitution matrix (Henikoff and Henikoff, 1992). The BLOSUM40 matrix was used as an example; however other matrices could be used, including those based on chemical features of amino acids, leading to different results.

The example of Figure 6 shows two different ways to run AlphaSimp. The normal way will produce the full list of alphabets satisfying the parameters, producing an output similar to that of Figure 6b that shows a selection taken from the top 130 16-symbol alphabets, with the maximal cost of each subset set to 16.0 and the maximal cost of the alphabet also set to 16.0. The running time to produce such a list is about one second on a normal personal computer such as the 600 Mhz PentiumIII system running under Linux OS, that was used in the trials presented in this paper.

Alternatively, if the maximal cost for the entire simplified alphabet is set to zero, AlphaSimp will return only the best alphabet and the parameter defining the maximal cost for a subset will be interpreted as a percentage value of the total cost. In Figure 6a the 20 best alphabets with number of symbols between 1 and 20 are reported together with their global cost and time required for the calculation.

Since the costs of simplified alphabets increase with the extent of the simplification, the option of defining a maximal number of symbols may be very useful, although it is not compulsory. For instance, the command **`'alphasimp blosum40 20 1 15 15'`** will employ the BLOSUM40 substitution matrix to produce simplified alphabets of any number of symbols (from 20 to 1) with a maximal cost of 15. Thus, there will be no restriction on the number of symbols. As a result 4 063 simplified alphabets will be produced in a couple of seconds, as follows:

1	alphabet of 20 symbols, cost 0
175	alphabets of 19 symbols, costs ranging from 1.5 to 15
2461	alphabets of 18 symbols, costs ranging from 5.0 to 15
1385	alphabets of 17 symbols, costs ranging from 8.5 to 15
41	alphabets of 16 symbols, costs ranging from 13.0 to 15

No other simplified alphabets will be produced as the minimal cost for an alphabet of 15 or less symbols will be above the threshold of 15. For some applications it may be useful to produce a list of all simplified alphabets, without any restriction regarding the number of symbols (as above). However it is advisable to set a threshold for a maximal cost, otherwise the entire set of 51 724 158 235 372 simplified alphabets will be produced.

A further option that may be useful is the possibility to set a maximal cost for symbols. For instance, the command **`'alphasimp blosum40 20 1 5.5 60'`** will find in about one second all simplified alphabet of any size, with the only restriction that a symbol cannot have a cost higher than 5.5, this will produce 1990 simplified alphabets with a number of symbols between 13 and 20 and with global scores between 0 and 36.5.

(a)

Subsets	*Best partitions*	*Cost*	*Time*
1	ARNDCQEGHILKMFPSTWYV	190.500	00:00.00
2	ARNDCQEGHKPST,ILMFWYV	160.462	00:00.03
3	ARNDQEGHKPST,C,ILMFWYV	140.583	00:11:12
4	ARNDQEGHKPST,C,ILMFYV,W	120.917	02:42:00
5	AGPST,RNDQEHK,C,ILMFYV,W	105.705	08:46:07
6	AGPST,RNDQEK,C,H,ILMFYV,W	92.967	13:44:52
7	ANDGST,RQEK,C,H,ILMFYV,P,W	81.000	11:57:31
8	ANDGST,RQEK,C,H,ILMV,FY,P,W	69.167	05:42:47
9	AGST,RQEK,ND,C,H,ILMV,FY,P,W	58.500	02:03:10
10	AGST,RK,ND,C,QE,H,ILMV,FY,P,W	50.000	00:47:05
11	AST,RK,ND,C,QE,G,H,ILMV,FY,P,W	41.667	00:14:00
12	AST,RK,ND,C,QE,G,H,IV,LM,FY,P,W	35.167	00:04:13
13	AST,RK,N,D,C,QE,G,H,IV,LM,FY,P,W	28.667	00:01:07
14	AST,RK,N,D,C,Q,E,G,H,IV,LM,FY,P,W	23.167	00:00.17
15	A,RK,N,D,C,Q,E,G,H,IV,LM,FY,P,ST,W	18.000	00:00.03
16	A,RK,N,D,C,Q,E,G,H,IV,LM,F,P,ST,W,Y	13.000	00:00.00
17	A,R,N,D,C,Q,E,G,H,IV,LM,K,F,P,ST,W,Y	8.500	00:00.00
18	A,R,N,D,C,Q,E,G,H,IV,LM,K,F,P,S,T,W,Y	5.000	00:00.00
19	A,R,N,D,C,Q,E,G,H,IV,L,K,M,F,P,S,T,W,Y	1.500	00:00.00
20	A,R,N,D,C,Q,E,G,H,I,L,K,M,F,P,S,T,W,Y,V	0.000	00:00.00

(b)

Ranking	*Partitions*	*Total costs*	*Max subset*
1	A,RK,N,D,C,Q,E,G,H,IV,LM,F,P,ST,W,Y	13.000	4.500
2	AS,RK,N,D,C,Q,E,G,H,IV,LM,F,P,T,W,Y	13.500	4.500
3	A,R,N,D,C,Q,E,G,H,IV,LM,K,FY,P,ST,W	13.500	5.000
4	AST,R,N,D,C,Q,E,G,H,IV,LM,K,F,P,W,Y	13.667	8.667
5	A,R,N,D,C,QE,G,H,IV,LM,K,F,P,ST,W,Y	14.000	5.500
6	AG,R,N,D,C,Q,E,H,IV,LM,K,F,P,ST,W,Y	14.000	5.500
7	A,RK,N,D,C,Q,E,G,H,ILV,M,F,P,ST,W,Y	14.000	6.000
8	AS,R,N,D,C,Q,E,G,H,IV,LM,K,FY,P,T,W	14.000	5.000
...	...	...	...
122	AG,R,N,D,C,QE,H,IV,LM,K,F,P,S,T,W,Y	16.000	5.500
123	A,RK,ND,C,Q,E,G,H,IV,L,M,F,P,ST,W,Y	16.000	6.500
124	AG,RK,N,D,C,Q,E,H,ILV,M,F,P,S,T,W,Y	16.000	6.000
125	A,RK,N,D,C,Q,E,G,H,ILMV,F,P,S,T,W,Y	16.000	11.500
126	A,R,N,DE,C,Q,G,H,IV,L,K,M,FY,P,ST,W	16.000	6.000
127	A,R,N,DE,C,Q,G,H,IV,LM,K,FY,P,S,T,W	16.000	6.000
128	A,RK,N,D,C,QE,G,H,ILV,M,F,P,S,T,W,Y	16.000	6.000
129	A,RK,N,D,C,Q,E,G,H,I,LMV,F,P,ST,W,Y	16.000	8.000
130	A,RK,N,D,C,Q,E,G,H,IMV,L,F,P,ST,W,Y	16.000	8.000

Fig. 6. (a) Shows a list of the best twenty simplified amino acid alphabets, with a number of symbols between 1 and 20, calculated using the BLOSUM40 substitution matrix without any discrimination for unbalanced subsets. The global cost and time required for the calculation on a normal personal computer are also shown. (b) Shows a selection taken from the top 130 alphabets of 16 symbols, without any discrimination for unbalanced subsets. The total cost for the simplified alphabet is also shown together with the maximal cost for a single subset. The running time to produce such a list is about one second on a normal personal computer.

DISCUSSION

From Figure 2 it can be noticed that most of the 51×10^{12} possible simplified amino acid alphabets have a number of symbols between 7 and 9, with a clear peak at 8 that can produce more than 15×10^{12} simplified alphabets.

This figure does not seem to agree with Figure 6a that shows the times required for identifying the best simplified alphabets. It could seem plausible that given a target number of symbols (k), the execution time would reflect the number of possible alphabets with such a number of symbols. Instead it appears that it is much shorter than expected when the number of symbols is higher. For instance, there are 61×10^9 possible 13-symbol alphabets and the program can find the best in 67 seconds (see Figure 6a). In contrast, there are 'only' 45×10^9 possible four-symbol alphabets, but it takes 2 hours 42 minutes (about 120 times longer) to find the best. This discrepancy is due to the branch selection procedure that is much more effective when the maximal allowed cost for the alphabets is smaller and therefore more selective enabling the program to discard more branches (see Figure 3), taking a better advantage of the branch and bound strategy. And it is obvious that the cost of alphabets increases with the level of simplification (see Figure 6a).

To speed up the running time of AlphaSimp it is possible to set a limit for the cost of a single subset. If this limit is fixed to a moderate value, often there will be a negligible variation of the results with a gain in the performance. As the limit becomes more selective, all the unbalanced partitions will be progressively filtered out. An extreme case of unbalanced partition is shown in Figure 5a that shows the best three-symbol alphabet calculated with the BLOSUM40 matrix is {ARNDQEGHKPST, C, ILMFWYV}. This is a very unbalanced partitions since the three subsets have a respective cost of 92.583, 0.000 and 48.000, giving a total cost of 140.583. If we run the program after setting the maximal cost for a single subset not greater than 35% of the total cost, we obtain the following three-letter alphabet {ACGPST, RNDQEHK, ILMFWYV}, where the three subsets have a cost of 48.571, 47.167 and 48.000 giving a total cost of 143.738. As expected, the time required to run the program decreases quite dramatically, from 672 seconds to 79 seconds.

To make an approximate estimate of the gain obtained by applying the branch and bound algorithm, we produced a program that performs the same operations of AlphaSimp without skipping any branch, thus producing and analysing every possible simplified alphabet. We could calculate that on our computer the average scanning rate was about 6 million simplified alphabet per minute, with an estimate time of 17 years to complete the analysis of the 51×10^{12} possible simplified amino acid alphabets!

In conclusion, the approach presented in this paper allows an exhaustive analysis of all the possible simplified amino acid alphabets, using a branch and bound algorithm together with standard or user-defined substitution matrices that are used for scoring the alphabets. The performance of the program is extremely good when the extent of the simplification is contained and the simplified alphabets are maintained above ten symbols. Under these conditions the program is able to complete the analysis in minutes or even seconds with a normal personal computer. However, the performance becomes worse, taking up to several hours, for highly simplified alphabets.

ACKNOWLEDGEMENT

The financial support of Telethon (Italy) grant number B.57 is gratefully acknowledged. N.C. is supported by the grant Cofin99 'Bioinformatica e Genomica'.

REFERENCES

Balas,E. and Padberg,M.W. (1976) Set partitioning: a survey. *SIAM Review*, **18**, 710–760.

Brazma,A., Jonassen,I., Eidhammer,I. and Gilbert,D. (1998) Approaches to the automatic discovery of patterns in biosequences. *J. Comput. Biol.*, **5**, 279–305.

Dill,K.A. (1985) Theory for the folding and stability of globular proteins. *Biochemistry*, **24**, 1501–1509.

Henikoff,S. and Henikoff,J.G. (1992) Amino acid substitution matrices from protein blocks. *Proc. Natl Acad. Sci. USA*, **89**, 10915–10919.

Kamtekar,S., Schiffer,J.M., Xiong,H., Babik,J.M. and Hecht,M.H. (1993) Protein design by binary patterning of polar and nonpolar amino acids. *Science*, **262**, 1680–1685.

Karlin,S. and Ghandour,G. (1985) Multiple-alphabet amino acid sequence comparison of the Ig-kappa chain constant domain. *Proc. Natl Acad. Sci. USA*, **87**, 8597–8601.

Murphy,L.R., Wallqvist,A. and Levy,R.M. (2000) Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Eng.*, **13**, 149–152.

Riddle,D.S., Santiago,J.V., Bray-Hall,S.T., Doshi,N., Grantcharova,V.P., Yi,Q. and Baker,D. (1997) Functional rapidly folding proteins from simplified amino acid sequences. *Nat. Struct. Biol.*, **4**, 805–809.

Sagot,M.F., Viari,A. and Soldano,H. (1997) Multiple sequence comparison—a peptide matching approach. *Theor. Comp. Sci.*, **180**, 115–137.

Shepherd,J.C. (1981) Method to determine the reading frame of a protein from the purine/pyrimidine genome sequence and its possible evolutionary justification. *Proc. Natl Acad. Sci. USA*, **78**, 1596–1600.

Schwartz,R.M. and Dayhoff,M.O. (1978) Matrices for detecting distant relationships. *Atlas of protein structure*, **5**, 353–358.

Wang,J. and Wang,W. (1999) A computational approach to simplifying the protein folding alphabet. *Nat. Struct. Biol.*, **6**, 1033–1038.