Fast Matching of CBG Patterns with Applications to Protein Matching

Scott Hazelhurst and Gloria Aikhorin

School of Computer Science, University of the Witwatersrand, Johannesburg
Private Bag 3, 2050 Wits, South Africa

Abstract

The large data set sizes produced in many biological applications, makes pattern matching in computational biology a challenge. We present a technique for pattern matching an important class of protein patterns. We show how such a protein pattern can be represented as a logical expression, from which a circuit can be automatically synthesised, and implemented on field programmable gate arrays, which leads to highly parallelisable solutions. The method was tested on the Prosite database, and almost all the patterns could be dealt with very efficiently leading to throughput rates in most cases excess of 10^8 symbols per second.

1. Introduction

Advances in molecular biology have created huge databases of biological information and understanding of how to use the data. One important class of data is protein sequences. Large databases of protein information exist and new protein sequences are created all the time.

Proteins, macromolecules critical for living organisms, are made up of amino acids, and the primary structure of a protein is its linear sequence of amino acids. Most living organisms' proteins are made up from 20 amino acids. In this paper, these amino acids are represented by letters from the set


d and proteins as strings from \( A^* \).

Much of the functionality of a protein is determined by its three-dimensional shape, but it is not easy to determine and use the shape information effectively. One approach to understand the functionality of a protein sequence is to compare a new sequence against sequences already in a database. This can be done in several ways: the approach explored in this paper is check to see whether a protein sequence contains patterns with known functionality. Given the large size of data sets, it is important to be able to process very large amounts of data quickly.

1.1. Classes with bounded gaps

Patterns of interest can be described in many ways. Sometimes, the pattern can just be given verbatim. For example, the sequence that represents the glycosaminoglycan attachment site is \( S \rightarrow G \rightarrow x \rightarrow G \) (the serine amino acid, followed by a glycine amino acid, followed by anything followed by another glycine). We use \( x \) to represent any symbol.

However, it is typical for patterns to be more complex. A simple example is the tyrosine kinase phosphorylation site, represented by the pattern

\[ [RK] \rightarrow x(2,3) \rightarrow [DE] \rightarrow x(2,3) \rightarrow Y, \]

This site contains an arginine or lysine amino acid (\( R \) or \( K \)) followed by 2 or 3 other amino acids, followed by an aspartic or glutamic acid amino acid (\( D \) or \( E \)) followed by any 2 or 3 amino acids followed by the tyrosine amino acid.

There are many ways of representing patterns. An obvious way is to use regular expressions. However, regular expressions are more powerful than necessary, and so the complexity of regular expression matching may not be cost-effective. Classes of characters and bounded-sized gaps (CBGs) are a simple, yet expressible method of expression many patterns. A CBG is a sequence of elements, where an element is either:

- A class of amino acids. (In the above example, the class \([RK]\) is the set containing arginine and lysine.)
- A bounded gap. This element says that we can have a sequence, bounded in length, of any amino acids before the next match must take place. A gap may either be of precise length (e.g. \( x(3) \), a gap of exactly 3) or bounded between two ranges (e.g. \( x(4,9) \) a gap of between 4 and 9 inclusive).

Useful syntactic sugar allowed is to express the set of amino acids not in a class. For example, in the syntax of the prosite database, whereas \([RK]\) represents an element that matches either lysine or arginine, the element \(\{R,K\} \) matches any amino acid except lysine or arginine. This is just shorthand for \(A \setminus [RK]\). In addition, one can specify that an element is repeated by a certain factor: \([RK][3]\) is short-hand for \(\{RK\}^3\).

1.2. Contributions of paper

This paper proposes a new solution to CBG matching. For performance reasons, a hardware solution is proposed using field programmable gate arrays (FPGAs). FPGAs offer the possibility of fast implementation compared to software, with much greater flexibility than ASICS, and highly parallelised solutions for very high performance.

The basic idea used is that for each CBG pattern being looked for, we build (automatically) a specialised circuit for matching for that particular pattern (rather than building a general circuit for doing general matching). This enables us to build optimised circuits that do not require external memory (for example for a table). The general methodology is as follows:

- We take the CBG representing the pattern and represent it as a boolean expression.
- The boolean expression is converted into a circuit represented as a VHDL program.
- The VHDL program is automatically synthesised into a circuit using standard FPGA design tools, from which the FPGA bitstream is derived.
The structure of the paper is as follows. Section 3 shows how CBGs can be represented as boolean expressions. Section 4 gives background on FPGAs and explains how the boolean expressions are converted into CBGs. Section 5 presents experimental results, discusses their significance and compares to previous work. Finally, Section 6 concludes and presents future research.

2. System Architecture

Our general architecture is shown in Figure 1. Given a CBG we create a specialised circuit that matches for that CBG (how this circuit is created will be shown in subsequent circuits). The external interface of the circuit is as follows: there is a 5-bit data input, a clock input and a reset input; there is one output, which indicates whether there is a match. Amino acids are encoded in 5 bits (more detail on this later). In each clock cycle, one amino acid is fed in. The Match output goes high exactly if the buffers in the circuit contain a match that pattern.

Internally, there are two main parts of the circuit. A series of shift-registers, each 5-bits wide, stores a segment of the sequence. On each clock cycle a new piece of data enters from the left, and the data moves right one step through the shift-registers. If the longest minimal sequence that matches the CBG is of length 4, there will be 16 registers. (Since data flows from left to right, the rightmost register contains the earliest amino acid currently stored in the circuit.) The other main part of the circuit is a piece of combinational logic that does the actual matching. It uses the data stored in the shift registers to make a decision where there is a match.

The overall system performance is determined by the maximum speed that system clock can be set at — the limiting factor is the longest combinational delay in the Matcher part of the circuit.

We emphasise that for each CBG a new circuit is created. This relies on the fact that FPGAs can be reprogrammed quickly. The key part is the Matcher. We now show how this can be generated automatically in two steps: we first represent a CBG as a boolean expression, and then convert the expression to circuit.

3. Representing CBGs as boolean expressions

Boolean expressions are a very powerful and flexible language for describing properties. We have used the basic idea described here in different areas of pattern matching in different areas with great success [4, 6]. The virtue of the approach is the great semantic elegance and the tools for reasoning and manipulating boolean languages. And while boolean expressions are a much more powerful language than CBGs, we shall not be paying any extra overhead for the extra power of the language — the cost of matching will be determined by the complexity of the patterns, not by the complexity of the language.

Given a pattern \( p \), we wish to derive a boolean expression \( \phi_p : A \rightarrow \{0,1\} \), which given a sequence of amino acids returns true iff the pattern matches start of the sequence. Suppose we need to inspect \( n \) elements of the sequence in order to determine the truth of the expression, then we need \( n \) variables (over the amino acids) in the expression. Let these variables be \( z_0, \ldots, z_{n-1} \).

For example, the CBG \( S = G \rightarrow G \rightarrow G \) is represented by the expression \( z_0 \land z_1 = G \land \neg z_2 = G \). Given values for the \( z_i \) we can determine whether the pattern matches. To determine whether a pattern matches anywhere in a sequence, we repeatedly substitute in values for the \( z_i \). That is, we first check to see whether the pattern matches a sequence at position 0 by substituting the \( i \)-th element of the sequence for \( x_i \) (for all \( i \)) and evaluating the expression. If it’s true, we know there’s a match at position 0; if not, we move on. In general to determine whether the pattern matches a sequence at position \( j \), we substitute the \((i + j)\)-th element of the sequence for \( x_i \) (for all \( i \)) and evaluating the expression.

The pattern \( \text{[R]} \rightarrow x \rightarrow \text{[DE]} \rightarrow x \rightarrow Y \) is represented by the expression
\[
(\exists i \forall j \in [0..3]) (z_i = 0 \land z_j = 1) \land (z_j = 1 \lor \neg z_j)
\]

The pattern \( \text{[R]} \rightarrow x \rightarrow \text{[DE]} \rightarrow x \rightarrow Y \) is represented by the expression
\[
(\exists i \forall j \in [0..3]) (z_i = 0 \land z_j = 1) \land (z_j = 1 \lor \neg z_j)
\]

3.1. Formal translation

The formal translation from CBG to boolean expression is given as follows. \( \text{match}(p, j) \) is a boolean expression which represents whether the pattern \( p \) matches a sequence at position \( j \). Let \( p \) be a class of amino acids.

\[
\text{match}(p_0, \ldots, p_m, j) = \bigvee_{0 \leq i \leq m} \text{match}(p_i, \ldots, p_m, j + 1)
\]

If the first element in the pattern is a gap of between size \( b \) and \( j \), then there is a match if the rest of the pattern matches at position \( b + 1 \) or \( b \), or at position \( j \). Let \( p_0 \) represent the gap \( (b_0, j) \).

\[
\text{match}(p_0, \ldots, p_m, j) = \bigvee_{0 \leq i \leq m} \text{match}(p_i, \ldots, p_m, j + k)
\]

3.2. Encoding using boolean-valued variables

In the description above, the variables \( z_i \) take on amino acids as their values. For implementation reasons, it is not possible to directly use variables that have such values. Instead, we encode amino acids, first as a number and then as a bit-vector (of size 5 since there are 20 amino acids). For example, the amino acid alanine is encoded by \( 0 \) (encoded by \( (f, f, f, f, f) \)), and the amino acid lysine by \( 11 \) (encoded by \( (f, f, f, f, f) \)). Then for each \( z_i \), we introduce 5 boolean valued variables \( z_{i,0}, \ldots, z_{i,4} \), and so think of \( z_i \) as a vector variable.

Thus, the high-level boolean expression \( z_0 = A \) (A being the code for alanine) is implemented as
\[
\langle z_0, z_0, z_0, z_0, z_0 \rangle = (f, f, f, f, f)
\]

This simply boils down to the expression \( i \langle 0, 0, 0, 0, 1 \rangle = (f, f, f, f, f) \) using the convention of juxtaposition for conjunction and priming for negation. Similarly \( z_0 = K \) (K being the code for lysine) is implemented as
\[
\langle z_0, z_0, z_0, z_0, z_0 \rangle = (f, f, f, f, f)
\]
This reduces to the boolean expression $x_0 \lor x_2 \lor x_3 \lor x_4$. 

Given this encoding, the algorithm outlined in the previous section can be efficiently implemented. Any CBG can be reduced to a boolean expression over a set of boolean variables.

To represent and manipulate these boolean expressions we use reduced, ordered binary decision diagrams (BDDs) [11]. A BDD is a directed acyclic graph that represents a boolean function. It can be thought of as an efficient representation of the Shannon expansion of the function. Each non-terminal in the graph refers to a test of a particular variable’s value and the edges to the two children of the non-terminal represent the paths taken by either possible value (0 or 1). The low edge corresponds to the case where the variable is assigned 0 and the high edge corresponds to the case where the variable is assigned 1. The terminal nodes of the graph are the boolean constants 0 and 1. The value of the function for a given assignment of variables is determined by starting at the root of the BDD and following the edges at each non-terminal as dictated by the values of the variables until a terminal node is reached. The value at this node gives the value of the function.

BDDs must be reduced – contain no redundancy in the form of duplicate nodes and redundant tests – and ordered – have the variables appear in the same order on any path from root to leaf. These restrictions result in BDDs possessing some useful properties, including extremely compact representations of many boolean expressions. However, the size of the BDD is fairly sensitive to the variable ordering chosen – this can mean the difference between a BDD that is quadratic and one that is exponential in size for a given boolean expression [1].

To illustrate, Figure 2 shows a BDD for the function $(u \lor v) \land (w \lor x) \land (y \lor z)$. In the figure, the dotted lines show the low edge, and the solid lines show the high edge.

In summary, given a CBG, we can represent it as a BDD. We next discuss how this can be converted into a circuit.

4. Implementation on FPGAs

Given the boolean expression, we can create a circuit and implement that circuit on an FPGA. Section 4.1 introduces FPGAs, and then Section 4.2 shows how the BDDs are used to program the FPGAs.

4.1. FPGAs

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be programmed by the end user to implement various logic circuits. Over the last decade programmable logic devices have expanded in functionality and size, and the tools for programming them have improved in ease of use and sophistication [3, 9]. The Xilinx families of FPGA are good examples [10]. Such an FPGA consists of a set of logic blocks (called configurable logic blocks – CLBs – in Xilinx jargon) connected to each other. CLBs are directly connected to each other, but there are other routing resources to allow relatively distant CLBs to communicate as well as the fast and efficient distribution of common data and control signals (such as clocks and reset lines).

Input/Output Blocks (IOBs) control I/O to the FPGA. Modern FPGAs tend to have other resources such as significant on-chip RAM, clocks and multipliers. Signal processing is a well-known application area for FPGAs. A simplified picture of an FPGA is given below. The description above is simplified – for example some FPGAs divide the CLBs into slices, and each slice is implemented by number of look-up tables (LUTs).

The CLBs are the heart of the FPGA. Each CLB takes a number of inputs (e.g. 6) and produces a number of outputs (e.g. 3). Thus, each CLB computes a boolean function of its inputs by using look-up tables to determine the appropriate output. We
program the FPGA by deciding what values the CLB’s look-up tables should have. FPGAs range in size from 64CLBs (the equivalent of under 2000 logic gates in the XC4000 series) to over 10K CLBs (the equivalent of 8M gates) in the Virtex-II range.

4.2. FPGA implementation of CBG matching

We take the BDD representing the CBG and convert it into a VHDL program. This in turn is taken as input by the Xilinx Foundation 4 tool. The design tool then synthesises the circuit from the VHDL program, determines how the circuit will be partitioned across the FPGA/CPLD, and how the CLBs and programmable interconnects will be programmed. We used the Xilinx Foundation 4 tool.

The translation from BDD to VHDL program is straightforward. Each node in the BDD is implemented as a multiplexer. This is shown schematically in Figure 3.

In the BDD, the evaluation of the BDD goes top down – we move down the DAG choosing the appropriate child according to the value of the variable. In our translated circuit evaluation goes bottom-up. The two leaves of the tree are considered as the constant inputs (0 and 1) to the circuit and the value of the variable of each node is used to choose which of the leaf values should be propagated to the parent node.

This method of translation was chosen because it was the simplest to implement. A number of other approaches for the synthesis of good circuits from BDDs (and other decision diagram representations) have been proposed, with claimed better performance with regard to space, delay and power consumption (e.g. see [2, 5]).

5. Experimental Results

We tested these ideas out on the Prosite database [8]. Of the 1568 patterns in the data base, 1332 (85%) are represented by CBGs. We took each of these CBGs and converted these into a boolean expression represented by BDDs.

The smallest CBG required 20 BDD nodes, and the largest CBG required 34736, the average being 264. However, the high standard deviation (1876) shows that a more detailed analysis is needed.

Figure 4 shows for each CBG in the database the number of BDD nodes needed to represent that CBG. The y-axis is plotted on a log-scale. As can be seen, the vast majority of the CBGs require fewer than 100 nodes, and only a handful require more than 1000 nodes.

This is summarised in the histogram in Table 1. This shows for various ranges of BDD size (as expressed in number of BDD nodes):

- the number of CBGs which require that many BDD nodes; and
- the cumulative percentage of CBGs that require that many BDD nodes or fewer (rounded to one decimal place).

![Figure 4: BDD Size of all CBGs in the Prosite database](image)

<table>
<thead>
<tr>
<th>Number of BDD Nodes</th>
<th>Frequency</th>
<th>Cumulative Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–20</td>
<td>287</td>
<td>16.0</td>
</tr>
<tr>
<td>21–50</td>
<td>123</td>
<td>76.9</td>
</tr>
<tr>
<td>51–100</td>
<td>108</td>
<td>93.5</td>
</tr>
<tr>
<td>101–200</td>
<td>52</td>
<td>96.0</td>
</tr>
<tr>
<td>201–300</td>
<td>19</td>
<td>97.0</td>
</tr>
<tr>
<td>301–500</td>
<td>3</td>
<td>98.0</td>
</tr>
<tr>
<td>501–1000</td>
<td>2</td>
<td>99.8</td>
</tr>
<tr>
<td>1001–2000</td>
<td>2</td>
<td>100.0</td>
</tr>
<tr>
<td>2001–3000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3001–4000</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Histogram of BDD sizes for all CBGs in the Prosite Database

The size and performance of the FPGA circuit will be strongly correlated to the size of the BDDs. In principle, the cost doing the match will be determined by the depth of the BDD (the number of variables) rather than the size of the BDD. However, in practice the FPGA tools will find it more difficult to synthesise, optimise, place and route the larger resulting VHDL programs. Therefore we took a number example CBGs to determine size and performance of a range of different size circuits.

Table 2 shows the results of 10 experiments with CBGs of different size. The first column shows the accession number used to identify the CBG in the Prosite database. The second column shows the number of BDD nodes used to represent the expression. The third to fifth column show the performance characteristics of the FPGA circuit generated from the BDD. These figures were obtained from the Xilinx Foundation 4 tool.
for the XCV2vp50-6ff1517 FPGA. As of January 2002 it was the largest in Xilinx’s Vortex-II Pro range, with 22592 CLBs. Column 3 shows how much of the FPGA is utilised (measured as a percentage of the FPGA CLBs used). Column 4 shows the maximum combinational delay (in nanoseconds) for the matching circuit. From this, a predicted throughput million of amino acids is shown in column 5 as millions of amino acids per second that can be processed.

To summarise these results we would expect to be able to match 93% of the patterns at a throughput in excess of 100 million amino acids per second, and the worst case throughput should be more than 30 million amino acids per second. In all cases FPGA utilisation was very low.

We also repeated these experiments on the Vortex XC400E, a middle-of-the-range FPGA in the older Vortex-E range. Performance ranged from 50% slower on the PS0032 CBG to 132% slower on the PS01352. The

The one negative performance figure not reflected here is the cost of synthesising the FPGA circuits. For the smaller circuits, this was not a problem at all. However, the largest circuits took hours to synthesise. To some extent, we can pre-compute the synthesised circuits and store them for future use (the resulting bitstreams are not particularly large, and the actual programming of the FPGAs takes seconds at most). However, such large synthesis times are clearly a problem in other circumstances where it would be desirable to pattern match on the fly. This problem is addressed under future research.

The best comparative work is that of Navarro and Raffinot [7], who designed a sophisticated and ingenious bit-parallel algorithm for implementation in software to solve the problem. The basic idea of the algorithm is to exploit as much as possible the word-level parallelism of modern instructions sets. They tested their algorithm on the same data set as ours, but were not able to deal with 11% of the CBG patterns in the database. Experimenting on a 500MHz Pentium III, they were able to match at a rate of between 5Mb/s and 20Mb/s. For the same patterns we would match at over 100Mb/s. However, comparison is difficult because of the big difference in technology and the age of technology. What we do claim is that our basic approach is at least as competitive as theirs, not limited by the size of the pattern in the same way, and as discussed in the next section, easier to parallelise further.

6. Conclusion

We have proposed an FPGA based solution to the problem of matching CBGs representing protein patterns. We have achieved good throughput and utilisation results. Given these positive results, there are a number of areas that we would like to take forward.

Complementing the BDD representation with memory: As can be seen 97% of the CBGs have very small BDD representations. However, there are some that are much larger. Even though the largest CBGs can still be handled easily, capacity utilisation becomes significant, logic synthesis more expensive and we must questions how the method will scale (though our approach seems more scalable than existing techniques). The worst cases for us were those cases where there were several relatively large ranges of gaps in a CBG where there are many alternatives that can tested. We believe that complementing the BDD expressions with simple counters will enable us to have much more compact representations. We have done some preliminary investigation in this regard and are positive about this approach in not only improving the matching cost, but also making a significant improvement in synthesis costs.

VHDL synthesis: Improved synthesis of VHDL code from the BDD representation could lead to much more efficient implementations of the matching circuit.

Parallelism: There is considerable scope for improved parallelism.

- At the moment, the matching circuit is just a large combinational circuit. However, it would be possible with
the introduction of some registers to pipeline the different levels of the BDD. This would mean that we could run the clock at a much faster rate.

* Given the FPGA utilisation, it would be possible to have a number of copies of the matching circuit on the same FPGA. For most CBGs we could easily have up to 100 copies of the same matching circuit on one FPGA. Thus, in each clock cycle, we would be able to match up to 100 positions in the search string at a time.

Even taking into account increased routing costs, we should see a close to linear improvement in performance. Thus we conjecture that we could match at up to 10G amino acids per second for over 90% of the amino acids. This could be increased further by using multiple FPGAs.

However, though protein databases are large they are not so large that they would make matching at such high rates practically much more useful than matching at 100M per second. Thus rather than the above two suggestions, we believe that a more productive way of utilising the large amount of parallelism.

* We can match for several CBGs at the same time. The experimental results are such that we should be able to easily match for dozens of patterns at the same time. We need to do more experimentation to get a better idea of how far this can be pushed but there should be no reason why a user shouldn’t be able to achieve significant extra performance by doing multiple matching in parallel.

**Cleverer encoding of amino acids:** Would different encodings of the amino acids be better? As there are 20 amino acids, we can encode each amino acid using 5 bits. We simply ordered the amino acids in alphabetical order and then numbered them 0 through 19. It is possible a different encoding would be better. In real proteins one amino acid may be substituted for another, often without significant change in functionality (that is why patterns contain classes of amino acids). Amino acids have different properties (e.g. large/small, positive/negative, hydrophobic/hydrophilic) and it is more likely that a like amino acid will be substituted for a like amino acid. An encoding where we have bits to represent each major property and then a few bits to represent the amino acids with that property may require us to use 7 or 8 bits instead of 5. However, it could mean that expressions representing classes of amino-acids would be smaller (e.g. the expression that represents all large, hydrophobic amino acids would contain only 2 variables).

**Generalisations to richer patterns:** Finally, given the good performance obtained on CBGs, it is worth exploring what other types of patterns the method can deal with effectively.

### 7. References


