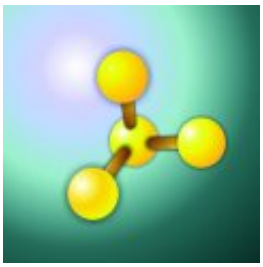
Crossroads
The ACM Student Magazine

Modeling Protein Dependency Networks using CoCoA

by ***Grey Ballard***



Abstract

In an interdisciplinary effort to model protein dependency networks, biologists measure signals from certain proteins within cells over a given interval of time. Using this time series data, the goal is to deduce protein dependency relationships. The mathematical challenge is to statistically measure correlations between given proteins over time in order to conjecture probable relationships. Biologists can then consider these relationships with more scrutiny, in order to confirm their conjectures. One algorithm for finding such relationships makes use of interpolation of the data to produce next-state functions for each protein and the Deegan-Packel Index of Power voting method to measure the strength of correlations between pairs of proteins. The algorithm was previously implemented, but limitations associated with the original language required the algorithm to be re-implemented in a more computationally efficient language. Because of the algebraic focus of the Computational Commutative Algebra language, or CoCoA, the algorithm was re-implemented in this language, and results have been produced much more efficiently. In this paper I discuss the algorithm, the CoCoA language, the implementation of the algorithm in CoCoA, and the quality of the results.

Introduction

Signal transduction networks are essential for cellular signaling both within individual cells and among cells. Biological study of these networks reveals how cells respond to different stimuli in their environment, ranging from pharmaceuticals to carcinogens. To understand signal transduction networks within cells, biologists must be able to identify the relationships among the associated proteins. Protein modification measurements produce time series data, but the challenge is to determine the dependency pathways between proteins. For their work in gene expression, Laubenbacher and Stigler [**6**] developed an algorithm using techniques based in computational algebra. Later, Allen et al. adapted their algorithm for use in protein dependency network modeling. They also applied a method of statistically measuring the strength of correlation between proteins. The heuristics of this method are discussed in detail in [**2**] and will be mentioned briefly in the following section. This paper addresses the implementation of these heuristics in the Computational Commutative Algebra, or CoCoA, language. Note: CoCoA should not be confused with Cocoa, a registered trademark of Apple Computers. In this paper, I will provide a brief introduction to the CoCoA language and show how well-suited it is for implementing algorithms dependent on computational algebra.

Laubenbacher and Stigler showed that in order to model gene expression, the Boolean network method could be generalized to one based on a polynomial ring over a finite field, $Z_p[x_1,\ldots,x_n]$. (Members of this

ring are polynomials with indeterminates $x_1,\ldots,x_n$ whose coefficients are integers modulo p, where p is a prime number.) In a Boolean network, p is 2, but Laubenbacher and Stigler's algorithm works for any prime p. Their algorithm produces polynomial functions that map a set of protein values at one time point t_i to the set of values at the next time point t_{i+1}. All the interpolated functions generated from this time series data are members of $Z_p[x_1,\ldots,x_n]$. Allen et al. [**2**] use these next-state functions to determine the influence of protein modifications over others in the network.

Laubenbacher and Stigler's method makes use of the Buchberger algorithm [**3**] to produce polynomials in a normal form. However, these polynomials are not unique unless viewed as representatives over an ideal of the associated polynomial ring, and they depend heavily on variable order (protein order in this case). A strong dependency should be manifested in most protein orderings, so the normalized next-state functions are recomputed for many different permutations, and the prevalence of the relationships is statistically measured. This dependency consensus is determined by the Deegan-Packel Index of Power [**7**], a game theoretic technique.

Algorithm

Given a set of time series data for a number of proteins (matrix M with rows representing time points and columns representing proteins), the algorithm developed by Allen et al. produces a matrix of strength-of-correlation values between pairs of proteins. This strength-of-correlation value for each ordered pair of proteins determines whether or not the second protein depends on the first. Current parameters of the algorithm include the number of permutations, N, and the value of p in Z_p.

The first step of the algorithm is to discretize the raw time series data based on the prime p (scale each entry from 0 to p). Next, remove duplicate rows and columns from matrix M. (Proteins with identical values will have identical results and can be grouped.) The following steps need to be repeated N times to show which dependencies endure for different orderings of the proteins.

- Reorder the columns of M using a random permutation. This ordering of the proteins is necessary since Buchberger's algorithm uses the term and variable orders of $Z_p[x_1,\ldots,x_n]$ very specifically to determine polynomial normal forms.
- Use Laubenbacher and Stigler's method (using Buchberger's algorithm) to compute next-state functions for each protein. These next-state functions will be polynomials in $Z_p[x_1,\ldots,x_n]$, a polynomial ring, and they represent the dependencies among the proteins.
- Using these functions, update the matrix that stores the Deegan-Packel Index of Power dependency value between every pair of proteins.

After looping the previous few steps N times, determine which Deegan-Packel Index of Power values represent strong correlations. The current method for this distinction is selecting values that are a certain number of standard deviations above the mean value.

The time complexity of the algorithm is polynomial in the number of proteins, n, and exponential in the number of time points, k. Specifically, the time complexity is $O(n^3k^2) + O(n(k^3+k)(\log p)^2+k^2n^3) + O(n^2(k-1)2^{ck+k-1})$ where c is a constant.

Maple Implementation

The initial implementation of the algorithm discussed in the previous section was written in the Maple language with the CASA [5] package. This allowed for a relatively fast prototype to be implemented that could be used for testing and empirical analysis. In fact, for a number of relatively small data sets, with no more than 30 proteins and 7 time points, this implementation worked well. However, as the number of proteins increased in later data sets, it became increasingly obvious that a better implementation was needed.

One problem with the first implementation was memory management. As the number of proteins increased, the amount of sampling using different permutations also increased. Despite efforts to force effective garbage collection, the amount of memory used continually increased. Even with a large number of proteins, the initial iterations ran quickly, but the later ones were significantly slowed, due to excessive memory paging.

Another issue was keeping the computations within the polynomial ring $Z_p(x_1,\ldots,x_n)$. This required numerous calls to the modulo function. Though simple to compute, the function calls are fairly expensive.

In addition to the time requirements of the algorithm itself, the Maple implementation had unacceptable overhead and inefficiencies associated with it. In an effort to overcome this limitation, I decided to implement the algorithm in a language better suited for computation using polynomials over finite fields.

CoCoA Implementation

CoCoA [4] is a powerful tool for computational algebra. It is an interpreted programming language designed for algebraic computations using numbers and polynomials within a specified ring. The algorithm for making protein dependency conjectures requires extensive and exact algebraic computation, and the built-in features of CoCoA make it an ideal choice for implementation. Because of its efficient memory management system, it is also able to handle a large number of proteins and permutations in a reasonable amount of time. In this section, I will discuss some of the relevant features of CoCoA and how they were used in this implementation. Figures 1-4 are examples taken from the actual implementation&#8212they are not pseudo-code. The resemblance of CoCoA syntax to pseudo-code illustrates its intelligibility.

The algorithm requires constant interaction with lists and matrices. CoCoA syntax for list manipulation is simple and intuitive. Figure 1 is an example of a user-defined CoCoA function called `Permute` that utilizes lists as well as a pseudo-random number generator that selects a random number inclusively between its parameters. I use the random permutation returned from `Permute` to redefine the protein order within each loop of the main algorithm.

```
Define Permute(N)

  // create a list of integers (1 to N) and an empty list

  TempList := 1..N;

  ReturnList := NewList(0);

  // randomly move elements of TempList to ReturnList

  For I := 1 To N Do

    J := Rand(1,N-I+1);

    Append(ReturnList,TempList[J]);

    Remove(TempList,J);

  EndFor;

  Return ReturnList;

End;
```

Figure 1: Function Permute generates a randomly permuted list of numbers from 1 to N.

Perhaps a more important and unique feature of CoCoA is that all computations are done with respect to the current ring. If the current ring is $Z_7[x,y,z]$ (Z_7 denotes integers modulo 7), then CoCoA evaluates $6x*5y$ as $2xy$ ($30 \equiv 2 \mod 7$). CoCoA allows different methods for defining rings. The Use command changes the current ring, and subsequent operations are evaluated under the new ring. In my implementation, I use the Using command to set aside a block of code for operations under the ring $Z_p[x_1,...,x_n]$, where p is prime and n is the number of proteins, as seen in Figure **2**.

```
// create polynomial ring over integers mod P

Ring ::= Z/(P)[x[1..N]];

Using Ring Do

  /* Perform operations under Z_P[x_1,...,x_N] */

EndUsing;
```

Figure 2: After defining a specific ring, CoCoA allows the user to perform operations under that ring with the Using command.

In order to perform the Buchberger algorithm, heavy computation within our ring is necessary. Interpolation of the time series data creates complicated polynomials that represent the next-state functions for each protein. In order to be representative of dependency relationships, these next-state functions have to be simplified into normal form based on the Gröbner Basis. Computing the normal form of a polynomial in a quotient ring is a tedious process, but in the CoCoA implementation it is coded (within the ring) relatively simply, as shown in Figure **3**.

```
// replace each polynomial in NextState list with its normal form

For K := 1 To Len(NextState) Do

  NextState[K] := NF(NextState[K],GrobnerBasis);

EndFor;
```

Figure 3: After interpolating next-state functions for each protein using the biological data, I use the NF command to compute normal forms of the polynomials.

CoCoA's memory management is fairly unusual&#8212it has three types of memory: working, global, and ring-bound. By default, defined variables are stored in working memory. In this case, the variables are accessible from all rings, but user-defined functions will not have access to them unless they are local to the function. Global variables are accessible from all rings and user-defined functions, but the variable names must include the prefix Memory in order to be stored in global memory. The third type of memory, ring-bound memory, is less commonly used. A ring-bound variable behaves similar to a global variable except that, because it is "bound" to a certain ring, it is destroyed when the ring ceases to exist. In my implementation, I use working and global memory. The main loop of the algorithm includes a random reordering (defined by the Permute function) of the columns of the matrix M. Because M is a parameter of the main function, it is stored in working memory. By storing a matrix with permuted columns in global memory, I can access the permuted matrix M from our user-defined functions within the main function. An example of a variable stored in global memory is the Dependency matrix used in Figure **4**.

Another interesting CoCoA feature that makes implementation easy is the string manipulation. In order to arrive at a value for the Deegan-Packel Index of Power between proteins, votes are tallied from the next-state functions in normal form. If the next-state function for protein A includes variables representing proteins B and C, then the dependency (B,A) receives a 1/2 vote and the dependency (C, A) receives a 1/2 vote. If protein A's next-state function also includes protein D, then each dependency relationship would receive a vote of 1/3. This method of voting only depends on one variable simply appearing in another variable's next-state function. To tally these votes, I convert the next-state functions to strings using CoCoA's Sprint command, create substrings for each variable representing a protein, and then use CoCoA's IsIn command to check if a substring (variable) appeared in a string (next-state function) for each dependency relationship. This process

is encapsulated in the function TallyVotes shown in Figure **4**. TallyVotes is called for each protein ordering, and it continually updates the Dependency matrix, which is the final result of the CoCoA implementation. The matrix is statistically analyzed to determine which elements (protein dependencies) represent strong correlations.

```
Define TallyVotes(P)

  For J := 1 To P Do

    // initialize Vote list of P elements of value 0

    Vote := NewList(P,0);

    Count := 0;

    // if variable x[K] appears in next state function of protein J, give it a vote

    For K := 1 To P Do

      Substr := "x[" + Sprint(K) + "]";

      If Substr IsIn Sprint(MEMORY.NextState[J]) Then

        Vote[K] := 1;

        Count := Count + 1;

      EndIf;

    EndFor;

    // normalize votes for protein J and update dependency matrix

    For I := 1 To P Do

      If Not Count = 0 Then Vote[I] := Vote[I] / Count EndIf;

      MEMORY.Dependency[J][I] := MEMORY.Dependency[J][I] + Vote[I];

    EndFor;

  EndFor;

EndDefine;
```

Figure 4: Function `TallyVotes` updates the matrix for P proteins storing the Deegan-Packel Index of Power strength-of-correlation values between pairs of proteins. If a protein variable appears in the normalized next-state function of another protein, it receives a fraction of a vote, and the strength-of-correlation value is increased. This function uses string manipulation, and the `IsIn` command to determine votes. Note that the `NextState` list and `Dependency` matrix are not passed in as parameters but do require the prefix `MEMORY` because they are stored in global memory.

Conclusion

CoCoA is designed to handle algebraic problems, and it is the most suitable language for implementing algorithms that require algebraic computation. In nearly all high-level languages, performing algebraic computation requires programming overhead. Modular arithmetic, performing calculations within a certain ring, finding the intersection of ideals, and other such algebraic operations are rarely built into a language. In order to use such languages, a programmer needs to code the steps for all of these nontrivial operations. The transparency of these operations in CoCoA greatly decreased the difficulty in implementation.

CoCoA overcomes the limitations of the Maple implementation of the algorithm. Memory management inefficiencies no longer prolong the execution time of the algorithm. This re-implementation proved to be a key step in the advancement of the greater research project. Using the original implementation, generating a matrix of strength-of-correlation values among 30 to 100 proteins required a dedicated machine almost a day. Some of the computations on later protein sets that involved hundreds of proteins never finished execution, even after weeks of running. The execution of the CoCoA implementation for 70 proteins and 7 time points ran in approximately 15 minutes. The efficiency of this new implementation allows us to consider much larger sets of raw data, which were previously ignored.

Although the implementation of this algorithm was my first experience using CoCoA, I found the language to be both syntactically straightforward and helpful in its documentation. Using both online manuals and the help commands (?), I was able to learn the basics of the language and understand simple examples in a relatively short time. The syntax for control structures and common data types are simple and intuitive, but it is also advantageous to become acquainted with the capabilities of relevant built-in algebraic commands before writing raw code. CoCoA is designed to handle many complex computations with commands in simple syntax (like the Normal Form (`NF`) command mentioned above), but the programmer must be aware of the available options in order to take advantage of them to save time and effort.

In addition to improving the efficiency in execution, coding in CoCoA has enabled the integration of this algorithm into the larger schema of protein dependency conjectures. For example, CoCoA is now used in the discretization of the raw data for input into the algorithm and in the creation of graphs that represent the protein networks from the output of the algorithm. Further research is underway in order to determine values of the prime p and the number of necessary permutations N that will best model the actual dependencies. The implementation allows for modification of these parameters, and other changes to the algorithm can be quickly and easily matched with changes in the CoCoA code. Thus, with the proper implementation, the execution of the algorithm will no longer limit the options of the greater research project.

Acknowledgments

The author was supported by a grant from the Wake Forest Research Fellows Program. This material is based on work supported by the NSF-NIGMS Program in Mathematical Biology through a grant, NIH R01-GM075304. The author would like to thank David John, professor of Computer Science at Wake Forest University, for his guidance and mentorship throughout this project.

References

1

E. E. Allen, J. S. Fetrow, L. W. Daniel, S. J. Thomas and D. J. John. Algebraic dependency models of protein signal transduction networks from time-series data, *Journal of Theoretical Biology*, 238:317-330, 2006, < **http://www.sciencedirect.com/science/article/B6WMD-4GJM41T-2/2/f6cd00bc6b46ef4d2726e3bf3c28e1b2**> (22 March 2006) .

2

E. E. Allen, J. S. Fetrow, D. J. John, and S. J. Thomas. Heuristics for dependency conjectures in proteomic signaling pathways. *Proceedings of the 43rd Annual Association for Computing Machinery Southeast Conference*, pages 75-79, 2005.

3

B. Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bull.*, 10(3):19-29, 1976.

4

CoCoA Team. CoCoA: a system for doing Computations in Commutative Algebra, <**http://cocoa.dima.unige.it**> (22 March 2006).

5

Gebauer, Kalkbrener, Wall, Winkler. *CASA: A Computer Algebra Package for Constructive Algebraic Geometry*, ISSAC'91, pp 403-410, 1991, software information available at <**http://www.risc.uni-linz.ac.at/software/casa/**> (22 March 2006).

6

R. Laubenbacher and B. Stigler. A computational algebra approach to the reverse engineering of gene regulatory networks. *Journal of Theoretical Biology*, 229:523-537, 2004.

7

A. D. Taylor. *Mathematics and Politics: Strategy, Voting, Power and Proof*. Springer-Verlag, 1995.

Biography

Grey Ballard (**ballgm2@wfu.edu**) is an undergraduate student majoring in Mathematics and Computer Science at Wake Forest University. After graduation, he will attend graduate school in Mathematics. His other interests include soccer, movies, and music.