Comparison of Hidden Markov Models and Sparse Bayesian Learning for Detection of Copy Number Alterations

Anamaria Crisan
Bioinformatics Training Program
University of British Columbia
Vancouver, BC V6T 1Z4
anamaria.crisan@gmail.com

Jing Xiang
Department of Electrical
and Computer Engineering
University of British Columbia
Vancouver, BC V6T 1Z4
jxiang@ece.ubc.ca

Abstract
Abnormal genome copy number alterations (CNAs) are associated with many severe diseases. Advances in microarray technology have greatly improved the resolution of detection of DNA copy number changes. This poses a challenge to existing computational methods to process the data accurately and efficiently. We compare two approaches to CNA detection for speed and accuracy. The first is a modified version of the popular technique, hidden Markov models (HMMs) called CNA-HMMer which is written in MATLAB. We then implement the recently proposed Genome Alteration Detection Algorithm (GADA) in the same language and compare the results obtained from both methods. Because GADA uses a compact representation called piece wise constant (PWC) vectors to describe DNA copy numbers, the MATLAB computations are simple because we can exploit the structure of the matrices when detecting CNAs in sparse Bayesian Learning (SBL). Backward Elimination is then used to eliminate irrelevant breakpoints. We show that both the performance and accuracy of the MATLAB implementations of the two methods vary considerably.

1 Introduction

The investigation of copy number alterations (CNAs) has important medical applications as these changes in DNA indicate diseases including cancer, autism and various genetic diseases. Array comparative genomic hybridization (aCGH) is a high throughput genetic technique that is used to identify chromosomal aberrations [7]. These are sections of DNA that have increased or decreased DNA copy numbers. The general method of characterizing CNAs is to first identify the breakpoints, which are the locations where copy number alterations occur. The regions between the breakpoints are then called segments. Once these are determined, they are classified as a loss, neutral or gain. A loss refers to a segment that has a lower DNA copy number relative to a normal sample. It follows that a gain refers to a segment that has an increased DNA copy number. Neutral segments exhibit normal DNA copy numbers.

This is a difficult problem because the observed CNAs are often affected by various sources of noise which interfere with the detection of boundaries. In addition, the technology of producing array platforms has advanced such that the probes for detection are
smaller resulting in higher resolution for detection of CNAs [7]. These considerations make it computationally challenging to identify variant regions accurately and efficiently.

Recent comparative studies [4,11] have found that circular binary segmentation (CBS) [6] was one of the most accurate approaches to detecting CNAs but also one of the slowest. It requires $O(n^2)$ computations. Using hidden Markov Models (HMMs) is another popular technique being applied to CNA detection. The HMMs are used to divide the given genetic sequence into segments which are then classified as a neutral, loss or gain. This standard approach was extended to use a mixture of Gaussians which made the model more robust to outliers, producing software called CNA-HMMer which runs in $O(n)$ time [9]. Recently, Pique-Regi proposed a new approach termed GADA (Genome Alteration Detection Algorithm) [8] that claims to be comparable in accuracy to Shah's approach but is a considerable improvement on runtime even though the complexity of the algorithm is also linear time. The GADA method uses a novel piecewise constant (PWC) representation of the breakpoints and then employs a combination of the sparse Bayesian Learning (SBL) suggested by Tipping [10] and backward elimination. The authors claim that it is 100 times faster than CBS [8].

1.1 Our Contributions

Our objective is to compare the GADA and the CNA-HMMer applications for speed and accuracy. Since the GADA method was implemented in C whereas the CNA-HMMer method was written in MATLAB, it would be difficult to compare their runtimes since applications usually run considerably faster in C. Thus, to obtain a fair comparison, we have implemented Pique-Regi's GADA application in MATLAB. The motivation behind choosing MATLAB as the platform for comparison is that if the GADA method proves to be a more efficient, then it can be used with existing MATLAB software at the BC Cancer Agency for further analysis of CNAs.

We ensured that our implementation reproduced the results of the Pique-Regi code exactly. In addition, we compared our results to the CNA-HMMer results in terms of segment classification, and also investigated the performance of both methods.

2 Methods

We performed all analyses on an Intel Duo core 1.60 GHz processor but we only employed one processor. There was 1 GB of RAM and our operating system was Windows Vista. The GADA method was implemented and run in MATLAB 8.0 (Mathworks, Natick, MA). We compared this with Shah's CNA-HMMer which was also run on version 8.0 of MATLAB.

2.1 PWC Representation and the Breakpoint Detection Problem

Pique-Regi’s most important contribution that is critical to GADA’s efficient implementation is their description of the copy numbers along the chromosome using PWC (piecewise constant) vectors. Any PWC vector $x$ with $K$ breakpoints can be represented by a linear combination of $K$ step vectors $f_i$ and a constant vector $f_0$ (Fig. 1). This description can be written in matrix notation shown below.

$$x = Fw$$

(1)
The full description of the representation including a complete set of equations for the formulation of the breakpoint detection problem is provided by Pique-Regi [8]. In general, since the copy number changes are small compared to the number of probes, equation (1) has a sparse representation in the $F$ basis, which can be exploited. The goal is to find $\hat{x} = F\hat{w}$ such that $\hat{x}$ is the closest to the observed $y$ subject to the constraint that $y$ has only $K$ non-zero components of $\hat{w}$. The closeness is represented by the least squares error measure and the sparseness is the $l_0$ norm. The optimization then constructs an $l_0$ regularization problem. However, as the authors mention, the solution would involve solving $\binom{M}{K}$ least squares problems which is intractable for chromosome lengths of $M$ and $K$ breakpoints. While it is conceivable to replace the $l_0$ norm with the $l_1$ norm so that methods such as basis pursuit [1] or lasso [3] can be used, the performance of these methods is not optimal because the columns of $F$ are have high collinearity. Thus, we need another method for this specific application of CNA detection.

### 2.2 Sparse Bayesian Learning

The problem can be solved using Sparse Bayesian learning (SBL), an empirical Bayes approach which is described in Tipping’s work [10]. In addition, the problem is cast as a maximum a posteriori (MAP) estimate. The general idea is that we assume that the observation model is normal. Then, the prior distribution of the weights is a hierarchical prior where $\alpha$ is a vector of hyperparameters from a gamma distribution. The prior selected is useful because given the hyperparameters, the posterior weight distribution is also normal and can be approximated by point estimates. The full description with the relevant equations is provided by Pique-Regi [8]. It is important to note that the MAP is given by the posterior mean and we replace $\sigma$ and $\alpha$ by their point estimates resulting in the following equations.

$$
\Sigma = (\sigma^2 F'F + \text{diag}(\alpha))^{-1} \quad \mu = \sigma^2 \Sigma F'y
$$

The maximum likelihood estimation of the hyperparameters is obtained by the EM algorithm [5].

### 2.3 SBL Implementation in MATLAB

The unique aspects of the GADA method arise because of their use of the PWC implementation. It would require $O(M^4)$ operations where $M$ is the number of probes in a chromosome to compute Equation (2) for an arbitrary $F$ matrix. However, because we can exploit the unique structure of $F$, $\Sigma$ and $\mu$ can be efficiently computed in $O(M)$ steps. The full pseudo-code that we followed for the SBL algorithm is written by Pique-Regi [8]. Additional time is saved as breakpoints with low statistical scores are removed. Parameter estimation for the Pique-Regi algorithm takes $O(|I|)$ time where $I$ is the set of breakpoints.
As the SBL algorithm removes breakpoints from the candidate breakpoint list, the GADA algorithm becomes faster, which is why it outperforms many other CNA detection algorithms [8]. In addition to the increased speed of the algorithm in C, the PWC representation also allows for a rather elegant implementation in MATLAB. The important computations are highlighted below.

The $H$ matrix, $H = G^{-1} = (F'F)^{-1}$ for our special $F$ matrix can be computed directly. It is a symmetric tridiagonal matrix. The main diagonal is given by

$$h_0(j) = \frac{(M - l_j)(l_{j+1} - l_{j-1})}{M (l_{j+1} - l_j)(l_j - l_{j-1})}$$

And the upper and lower diagonals are given by the expression below.

$$h_1(j) = -\sqrt{(M - l_j)(M - l_{j+1})}$$

It is important to note that our MATLAB implementation follows the Pique-Regi C code and not their paper in which the equation for $h_1$ is printed as the positive square root and not the negative square root. The equations are simplified and each term is represented as a vector in MATLAB. All computations are done with vectors. Once the vectors representing the diagonals are computed, we use the spdiags MATLAB function to construct the sparse $H$ matrix. Details are in Section 1.1 of the Appendix.

One of the most challenging aspects of implementing the SBL algorithm in MATLAB is determining how to exploit the tridiagonal structure of the matrices. For example, in order to compute $\Sigma$, we need to find the inverse of the $T$ matrix.

$$\Sigma = \sigma^2 T^{-1}G^{-1} \quad T = (\sigma^2G^{-1}A + I)$$

Because the $T$ matrix is a sparse tridiagonal matrix and is ill-conditioned, Pique-Regi finds the pseudo-inverse of the matrix. However, our attempts at doing so in MATLAB were unsuccessful and resulted in numerical errors and memory problems. Instead, we solve a linear system of equations using the Thomas Algorithm [2] which is designed for tridiagonal systems. These details regarding derivations and implementation information are written in section 1.4 of the Appendix. The source code for the Thomas Algorithm is provided in section 2.1.

2.3 Implementation of Breakpoint Ranking using Backward Elimination

After SBL finds a candidate set of breakpoints, the set is further pruned by Backward Elimination. Again, our MATLAB implementation follows Pique-Regi’s C code implementation which is not exactly what’s written in the paper. In their written paper, Pique-Regi includes a $\sigma^2$ term when calculating the statistical score [8] which is not in their implementation. The process of ranking breakpoints that we used is the same. We remove breakpoints with the lowest statistical score until all remaining breakpoints have a higher score than a specified threshold, the BE critical value. We also remove breakpoints whose segment size is smaller than a previously specified minimum segment size.

2.4 Comparing the CNA-HMMer and GADA Methods

We compare both the runtime and accuracy of the two algorithms. We provide each algorithm with two input files: one obtained from the Pique-Regi source code, and another from the CNA-HMMer source code. Both input files contain synthetic data. To compare the
running time, we use built in MATLAB functions to time how long it takes to execute each algorithm. Next, we record all the breakpoints and their corresponding classifications detected by GADA and compare them to the CNA-HMMer classification.

3 Results and Discussion

We re-implement the Pique-Regi GADA algorithm [8] in MATLAB to compare its performance and accuracy to the CNA-HMMer method [9]. Prior to comparing the two methods, we demonstrate that our implementation is able to output exactly the same results as the Pique-Regi C code. We validate our claim by observing the same output when both implementations were provided the same input. The input files used are those that originally accompanied the Pique-Regi source code.

Comparing the CNA-HMMer method to the GADA method is a difficult task as the two algorithms are considerably different. First we compare the runtime between our MATLAB implementation of GADA and the CNA-HMMer implementation. Next we compare the classification of the segments between the two methods (recall that they can be classified as a loss, gain or neutral segment).

3.1 Comparison of Performance

We find that the CNA-HMMer algorithm outperforms the MATLAB GADA algorithm. Using the input file from the Pique-Regi source code, the MATLAB GADA implementation takes 211 iterations of EM to converge while the CNA-HMMer took only 7. The total runtime of the CNA-HMMer algorithm is 3.0 seconds, while the GADA algorithm takes 30 seconds. However, CNA-HMMer is certainly slower than the C implementation of GADA which takes under a second to complete all calculations.

Upon extensive analysis of our implementation, we believe that the bottleneck occurs at one particular step. In the SBL pseudo-code written by Pique-Regi [8], the \( \Sigma \) matrix was computed according to Equation (5). As mentioned earlier, the GADA C code calculates the pseudo-inverse of the \( T \) matrix. However, this is not possible in MATLAB because the matrix is ill-conditioned. We avoid this problem by using the Thomas Algorithm to solve a system of linear equations. The backslash operator (\( \backslash \)) in MATLAB normally exploits the sparse matrix architecture to expedite the calculation. However, in this case it also produces poor results because the \( T \) matrix is ill-conditioned. We find that the Thomas Algorithm solves for \( \Sigma \) correctly (refer to Appendix section 1.4 and 2.1) but it requires \( O(n) \) time to execute [2]. Since we supply the algorithm with a sparse \( G^T \) matrix of size \( K \) by \( K \), where \( K \) is the total number of breakpoints, the algorithm will take longer than \( O(n) \) time (refer to Appendix section 2.2). In Figure 2 below, we demonstrate that in fact, this step occupies the majority of the computational time in the EM step. Although the current implementation of the Thomas Algorithm is the source of the bottleneck in speed, we believe that it can be modified to improve efficiency of the calculations by further exploiting the tridiagonal structure of the \( G^T \) matrix. Once the Thomas Algorithm is optimized, the MATLAB version of GADA should prove to be faster than CNA-HMMer.

3.2 Comparison of Accuracy

Comparing the accuracy of GADA and CNA-HMMer is challenging as the format of results produced by each method is significantly different. The GADA algorithm returns a list of breakpoints, only a subset of probes, which are classified as a loss, neutral or gain. However, the CNA-HMMer algorithm returns the probability of every probe being either loss, neutral or gain, as well as a final classification. To make the comparison, we take all the GADA breakpoints and manually compare their classification to the CNA-HMMer classification. We found that the classifications did not agree. Often, where GADA predicts a breakpoint which is loss or gain, the CNA-HMMer predicted it to be neutral. The tests are
performed on two data sets. The first data set is an input file provided by Pique-Regi and the second is an input file that accompanied the CNA-HMMer source code. The classifications generated by the two methods are not at all consistent. For example, when CNA-HMMer is run with Pique-Regi data, it classifies all the breakpoints as neutral. Although using the input from CNA-HMMer on the GADA algorithm produced different results, it is important to note that the original GADA C code implemented by the authors returns the same results that we produce with our MATLAB GADA implementation. Thus, the discrepancy is between the two algorithms themselves and is not the result of implementation in MATLAB.

![Graph showing comparison of total execution time of EM step vs. execution time of solving for Σ with the Thomas algorithm](image)

**Fig. 2.** Comparison of total execution time of EM step vs. execution time of solving for Σ with the Thomas algorithm. The x-axis indicates the amount of time to execute the operation and the y-axis shows the number of iterations.

### 3.3 Sources of Error

Possible sources for error include the inputting process of MATLAB. When we import data into MATLAB, it only records up to four decimal places of precision as opposed to the six digits we provide. While we believe this may affect our results slightly, it most likely is not a major source of discrepancy between the two methods.

The predictions of both algorithms are extremely sensitive to the input parameters. Both algorithms have very different default values for the alpha and beta parameters. CNA-HMMer prefers very high alpha and beta values, while GADA is set with a beta value of zero, and small alphas. Thus, it is possible that the parameters are not being set optimally and that there is a systematic method for selecting parameters that will result in more consistent results.
4 Conclusions

We have implemented a recently proposed algorithm called GADA in MATLAB. This method uses a novel representation called PWC vectors that result in useful properties including sparseness. The combination of sparse Bayesian Learning and Backward Elimination is then used to detect copy number changes. Because the PWC representation allows for computation with matrices that possess special structure, simple MATLAB code can be written for parameter estimation. We employ the Thomas Algorithm to solve a linear system of equations, which differs from the original implementation which takes the pseudo-inverse. Our implementation produces identical results as the original C implementation written by the authors. However, when we compare the GADA method with CNA-HMMer, our implementation is slower and also produces inconsistent results. Future work should focus on optimizing the Thomas Algorithm such that it further exploits the tridiagonal structure of the matrices to make the algorithm run faster. In addition, more investigation of setting appropriate parameters needs to occur. As Shah mentions in his paper [9], if CNA-HMMer eventually adds levels to the hierarchy of the model making it robust to parameter settings, then we would no longer need to select parameters by hand which may help us obtain consistent results. We anticipate that additional work with the MATLAB GADA implementation will make it a faster method than HMMs, while maintaining comparable accuracy.

Acknowledgments

We would like to thank Dr. K.P. Murphy and Dr. S.P. Shah for their assistance and guidance throughout the course of the project.

References


APPENDIX

1. SBL MATLAB Implementation Details

1.1 Computing the H Matrix

The equations below can be simplified for implementation as was done by Pique-Regi in their C code. The MATLAB code shows how we computed the diagonals in MATLAB and how we used the sparse matrix representation in MATLAB to construct the final H matrix.

\[
\begin{align*}
    h_0(j) &= \frac{(M - i_j)(i_{j+1} - i_{j-1})}{M (i_{j+1} - i_j)(i_j - i_{j-1})}, \\
    h_1(j) &= -\frac{\sqrt{(M - i_j)(M - i_{j+1})}}{M(i_{j+1} - i_{j-1})}
\end{align*}
\]

\[
\begin{align*}
    \text{ho} &= \text{diag}(G_{inv}) \text{ is size } M-1; \\
    \text{h1} &= \text{diag}(G_{inv},1) \text{ is of size } M-2 \text{ both the upper and lower diagonal are the same} \\
    \text{M-vector} &= \text{repmat}(M,M-1,1); \\
    i &= (1:M-1)'; \\
    \text{ho} &= ((\text{M-vector}-i) \cdot (i+\text{ones}(M-1,1))) / M) \cdot 2;
\end{align*}
\]

\[
\begin{align*}
\% h1 is represented by the following equation, but below is in vectorized form: \\
\% We divide by two here because, the h1 is actually based on the ho/2 \\
\% rather then calculate ho, then h1, then multiply ho by 2 (as they did \%
\% in the C code) we choose instead do do it this way \\
\% h1 = -sqrt(ho(i+1)*ho(i)) / 2 \\
\% H = G^{-1} \\
\end{align*}
\]

\[
\begin{align*}
\text{Tri_diag}_G_{\text{inv}} = \text{spdiags}([\text{h1}; 0], \text{ho}, [0; \text{h1}], [-1; 0; 1], M-1, M-1);
\end{align*}
\]

1.2 Computing the T Matrix

Below we compute the diagonals of the T matrix shown in the equation.

\[
[t_0, t_1] \leftarrow T = (\sigma^2 G^{-1} A + I)
\]

\[
\begin{align*}
\% \text{COMPUTE T method in their code.} \\
\text{to} &= \text{ho} \cdot \text{alpha}(1:K,1) \cdot \text{sigma}_2 + \text{ones} \cdot \text{length}(\text{ho}),1; \% \text{main diagonal of the T} \\
\text{tu} &= \text{h1} \cdot \text{alpha}(2:K) \cdot \text{sigma}_2; \% \text{the upper diagonal of the T matrix} \\
\text{tl} &= \text{h1} \cdot \text{alpha}(1:K-1) \cdot \text{sigma}_2; \% \text{calculate the lower diagonal matrix of T}
\end{align*}
\]
1.3 Computing the weights using the Thomas Algorithm

The MATLAB code for computing the weights is shown below. We have used the Thomas Algorithm to solve the tridiagonal linear system.

\[ T_w = w_0 \]
\[ W = \text{thomas}(to, tu, tl, wo); \]

1.4 Deriving and Computing the Sigma Matrix

Given that the T and H are defined as the following,

\[ T = (\sigma^2 G^{-1} A + I) \]
\[ H = G^{-1} = (F'F)^{-1} \]

below we show that by substituting the T and \( G^{-1} \) matrices that we compute in the implementation, we produce the correct expression for \( \Sigma \).

\[ \Sigma = \sigma^2 T^{-1} G^{-1} \]
\[ = \sigma^2 (\sigma^2 G^{-1} A + I)^{-1} G^{-1} \]
\[ = \sigma^2 (\sigma^2 G^{-1} A + F^{-1})^{-1} G^{-1} \]
\[ = (A^t G + \sigma^2 I)^{-1} G^{-1} \]
\[ = (A G + \sigma^2 I) G^{-1} \]
\[ = (AG^{-1} + \sigma^2 IG^{-1}) \]
\[ = (A + \sigma^2 G^{-1}) \]
\[ = (\sigma^2 G^{-1} + A) \]
\[ = (\sigma^2 G + A)^{-1} \]
\[ = (\sigma^2 F'F + \text{diag}(\alpha))^{-1} \]

Again, we use the Thomas algorithm to solve the tridiagonal linear system.

\[ \Sigma = \sigma^2 T^{-1} G^{-1} \]
\[ TX = G^{-1} \quad \text{(thus, } X = T^{-1} G^{-1}) \]
\[ \text{SIGMA} = \text{sigma}_2*(\text{thomas}(to,tu,tl,\text{Tri diag G inv})); \]

2. Thomas Algorithm

The Thomas algorithm, also called the tridiagonal matrix algorithm is a simplified method of Gaussian Elimination for tridiagonal systems. In these systems, the solution can be obtained in linear time \( O(n) \) instead of cubic time \( O(n^3) \).

2.1 MATLAB Code for the Thomas Algorithm

The following code was obtained from MATLAB Central.

http://www.mathworks.com/matlabcentral/fileexchange/1359

```matlab
function x = thomas(varargin)
% THOMAS Solves a tridiagonal linear system
% x = THOMAS(A,d) solves a tridiagonal linear system using the very efficient Thomas Algorithm. The vector x is the returned answer.
```
A*x = d;  
| a1  b1  0  0  0 ... 0 \ | x1 \ | d1 \ 
| c1  a2  b2  0  0 ... 0 | x2 | | d2 | 
| 0  c2  a3  b3  0 ... 0 | x | x3 | = | d3 | 
\ 0  0  0  cn-2 an-1 bn-1 / \ xn / \ dn / 

- The matrix A must be strictly diagonally dominant for a stable solution.
- This algorithm solves this system on (5n-4) multiplications/divisions and (3n-3) subtractions.

x = THOMAS(a,b,c,d) where a is the diagonal, b is the upper diagonal, and c is the lower diagonal of A also solves A*x = d for x. Note that a is size n while b and c is size n-1.
If size(a)=size(d)=[L C] and size(b)=size(c)=[L-1 C], THOMAS solves the C independent systems simultaneously.

ATTENTION: No verification is done in order to assure that A is a tridiagonal matrix.
If this function is used with a non tridiagonal matrix it will produce wrong results.

[a,b,c,d] = parse_inputs(varargin{:});

% Initialization
m = zeros(size(a));
l = zeros(size(c));
y = zeros(size(d));
n = size(a,1);

%1. LU decomposition

L = / 1 \
| 11 1 | 
| 12 1 | 
|    : : :    | 
| \ 1n-1 1 \ / 

U = / m1 r1 \
| m2 r2 | 
| m3 r3 | 
|    : : :    | 
| \ mn / 

ri = bi -> not necessary
m(1,:) = a(1,:);
y(1,:) = d(1,:);

for i = 2 : n
    i_l = i-1;
    l(i_l,:) = c(i_l,:)./m(i_l,:);
    m(i,:) = a(i,:) - l(i_l,:).*b(i_l,:);
    y(i,:) = d(i,:) - l(i_l,:).*y(i_l,:);
end
%2. Forward substitution (L*y=d, for y)

```matlab
%y(1) = d(1);
%for i = 2 : n
%   y(i,:) = d(i,:) - l(i-1,:).*y(i-1,:);
%end
```

%3. Backward substitutions (U*x=y, for x)

```matlab
x(n,:) = y(n,:)./m(n,:);
for i = n-1 : -1 : 1
    x(i,:) = (y(i,:) - b(i,:).*x(i+1,:))./m(i,:);
end
```

```matlab
function [a,b,c,d] = parse_inputs(varargin)

if nargin == 4
    a = varargin{1};
    b = varargin{2};
    c = varargin{3};
    d = varargin{4};
elseif nargin == 2
    A = sparse(varargin{1});
    a = diag(A);
    b = diag(A,1);
    c = diag(A,-1);
    d = varargin{2};
else
    error('Incorrect number of inputs.')
end
```
2.2 Analysis of the Execution time for the Thomas Algorithm

1) thomas(to,tu,tl,wo)
2) thomas(to,tu,tl,Tri_diag_G_inv)

wo is a K by 1 column vector, where K is the total number of breakpoints, and Tri_diag_G_inv is a K by K sparse matrix. Below is a comparison of the amount of time it takes to execute (1) compared to (2).

![Graph showing comparison of execution time for the Thomas Algorithm](image-url)