SPACE OPTIMAL CHROMATIC POLYNOMIAL ALGORITHM

Larry Basenspiler¹, Thomas Hain²
Division of Computer and Information Sciences
University of South Alabama
Mobile, AL 36688

Ben King²
The MITRE Corporation
McLean, VA 22102

Abstract

We describe an algorithm for calculating the chromatic polynomial of a graph. Along with some algorithmic and programming improvements, the data structure used makes the space requirement virtually optimal—linear in the size of input and independent of the density of the graph. A program based on the best previously existing algorithm was able to handle sparse graphs on up to 15 vertices. Our vectorized algorithm enabled us to process such graphs on up to 27 vertices.

1. Introduction

The chromatic polynomial P(G;λ) of a graph G(V,E) can be calculated by means of the recursive "chromatic reduction" process introduced by R. Read [3], namely:

\[ P(G;λ) = P(G-e;λ) - P(G/e;λ), \]

where G-e is the subgraph of G obtained by deleting an edge e, and G/e is a contraction of G, that is, a graph, obtained from G by replacing two vertices connected by e with a single vertex, and retaining all adjacencies of G. By repeated application of the chromatic reduction, we can express the chromatic polynomial P(G;λ) in terms of the chromatic polynomials for trees, for which analytical expressions are known. The theory and study of the properties of chromatic polynomials can be found in [3].

The reduction process can be illustrated by a binary computation tree (which we will call S-tree) as follows.

For each node of the S-tree, its left child is an edge subgraph, and its right child is a contraction of its parent. The leaves of the S-tree are trees. The number of leaves of

the computation tree depends both on the number of vertices, \( n = |V| \), and the number of edges, \( m = |E| \), of the graph. Obviously the length of the leftmost branch of the S-tree is \( n-2 \), the length of the rightmost branch is \( m-(n-1) \). The computation tree yields an intuitive measure of the complexity of any algorithm for calculating chromatic polynomials. The execution time must grow exponentially with the size of the graph, and the space requirement depends on the specific design.

In section 3 of this paper we describe our algorithm (which we will call A-Chrom) for calculating the chromatic polynomial of a graph. A-Chrom uses minimal space, and its execution speed allows significantly larger graphs to be handled than previously existing algorithms. The efficiency of the algorithm is due to its design and the choice of the data structure introduced in section 2. We will compare the performance of A-Chrom with the algorithms presented in [4] (the fastest thus far, and denoted as R-Chrom), in [1] (denoted as B-Chrom), and in [2] (denoted as NW-Chrom).

2. Data Structure and Primitives

Conventional computer representations of a graph are the adjacency matrix (whose element \((ij)\) is a boolean value "true" if vertices \(v_i, v_j\) are adjacent, and "false" otherwise), adjacency lists, and the edge list (consisting of two arrays containing corresponding left and right ends of all edges of a graph). In adjacency matrices, a large amount of storage is wasted by allocating an addressable unit of memory to store each binary valued element. We allocate a single bit per adjacency matrix entry, a technique easily implemented in modern programming languages, by forming a hexadeci-

---

¹ Partially supported by National Science Foundation grant No. CCR-8804250 and No. ASC-8900690.
² Partially supported by National Science Foundation grant No. ASC-8900690.
mal number out of a row of the adjacency matrix and treating it as a bit pattern. The representation of each row as a word will be denoted as A-form. The word length forms an upper limit (64 in the case of our implementation on a Cray) on the number of vertices of input graphs. The whole adjacency matrix can be then viewed as a vector of words, which we denote as the G-vector.

We now analyze the complexity of the operations defined on the data type 'G-vector', that is, access to, and deletion of a vertex; access to, deletion, and contraction of an edge. Access to a vertex in A-form is direct; for access to an edge in A-form, we use the identity matrix in A-form, that is, a predefined template array containing successive powers of 2. For example, in order to delete edge (i,j), word i in the G-vector is XOR'ed with word j of the template array. Thus an edge deletion takes one operation, XOR, following access to the edge. NW-Chrom and R-Chrom, which use edge list graph representation, are designed so that an edge to be deleted is accessed directly, thus edge deletion also requires one operation.

In the matrix representation of a graph, in order to contract edge (v_i,v_j), rows i and j are bitwise OR'ed, respectively, columns i and j are bitwise OR'ed, and the j-th row and column are deleted. In our implementation using A-form, however, the algorithm is designed so that j is always the largest label, thereby avoiding deletion of the v_j word by simply reducing the size of the vector. OR'ing the columns in A-form would be much more complex and take \( n^2 \) operations. As will be described in Section 3, the design of A-Chrom eliminates the need to operate on columns. Thus edge contraction involves one operation of replacing word i by the result of the OR operation on words i and j. By design, contracting an edge (v_i,v_j) in both NW-Chrom and R-Chrom has been reduced to the merging of two (short for sparse graphs) lists, and takes time proportional to the sum of degrees, \( d_i + d_j \), of vertices v_i and v_j.

### 3. Algorithm A-CHROM

A-Chrom incorporates the underlying ideas in B-Chrom and R-Chrom, and uses A-form graph representation. A-Chrom starts by finding a spanning tree of the input graph G and appropriately relabeling G (cf. [4]). This becomes the root of the S-tree. Figure 1(a) depicts a graph where edges of the found spanning tree are shown solid, and all other edges, called chords, are shown dashed (cf. [4]).

A-Chrom proceeds by processing the graph in the following manner. For the root of the S-tree, push its left child, G-e, on a stack. Contract e and repeat the process with G/e as the new root, until the latter is a leaf, i.e. no chords remain in the underlying graph. Obviously the stack may contain up to \( n-3 \) graphs, and the \( n^{th} \) graph from the bottom of the stack is of dimension \( n-i+1 \). Next, pop a graph off the stack and repeat the above process until the stack is empty (cf. [1]). Note that, unlike in the case of R-Chrom, the chosen data structure makes it unnecessary to check whether a new edge (which may appear in a contraction) is a tree edge or a chord.

The adjacency matrix of the graph in Figure 1(a) is shown in Figure 1(b). In each row of the matrix, the edges of the spanning tree are represented by the leftmost "1" and shown separated from the rest of the matrix by the staircase line. All other "1"s up to the diagonal represent chords. The staircase line is reflected in the array of column indices, LEFT.

![Graph and Matrix](image)
The basic action of each step in the calculation of the chromatic polynomial is the replacement of graph $G$ by graphs $G-e$ and $G/e$. Aside from the complexity of the primitives discussed above, this requires $O(2^n)$ creations of computation tree nodes. The manner in which the primitives, as well as node copying, are implemented is crucial for the effectiveness of any chromatic polynomial algorithm.

Prior to operations on an edge, A-Chrom (and B-Chrom) have to search sequential bit positions (matrix elements) in a row for a "1". A-Chrom reduces overall searching time due to the presence of two additional arrays. For the $i$th $G$-vector on the stack, $G_i$, element $\text{RIGHT}[i]$ contains the column index of the rightmost "1" in the row pointed to by $\text{HOR}[i]$ to keep track of how far left we have searched (see Figure 1(b)). Arrays HOR and RIGHT of pointers make the average time to scan for a chord in each step proportional to the average vertex degree of $G$, $d_{\text{avg}}$. The initial position of each $\text{RIGHT}[i]$ for $1 \leq i \leq n-3$ is $i-1$, that is, conceptually we consider only the lower triangle (delineated by dashed line in Figure 1(b)) of the adjacency matrix. When a $G$-vector $G_i$ is popped off the stack, the edge $e=(\text{HOR}[i], \text{RIGHT}[i])$ is deleted. If $G_{i-}e$ is not a leaf, $G_{i/e}$ is obtained as follows. Words $\text{HOR}[i]$ and $\text{RIGHT}[i]$ are OR'ed into word $\text{RIGHT}[i]$, and bits 1 to $i-1$ of the latter reflect the appropriate changes. However, bits in the $\text{RIGHT}[i]$ positions of each word below $\text{RIGHT}[i]$ must be the result of OR'ing themselves with bits in $\text{HOR}[i]$ position in the same word (which is equivalent to bitwise OR'ing of columns). The latter bits are all "0"s since they are symmetric with the bits to the left of $\text{RIGHT}[i]$ in the $\text{HOR}[i]$ word. Bits in the $\text{RIGHT}[i]$ positions of each word above $\text{RIGHT}[i]$ are in the upper triangle of the adjacency matrix and are not taken into consideration. Therefore OR'ing the columns is unnecessary. Array HOR carries out two more tasks in A-Chrom. When $\text{HOR}[i] = 2$, the $S$-tree node is a leaf. When $\text{RIGHT}[i]$ meets $\text{LEFT}[i]$, $\text{HOR}[i]$ is decremented, and only $\text{HOR}[i]$ words are pushed on the stack, regardless of the actual dimension of the current graph.

Let us evaluate the time required to place a new node of the $S$-tree on the stack (COPYING in Table I below). On step $i$, B-Chrom copies an $(n-i) \times (n-i)$ matrix in $O(n^2)$ time. NW-Chrom copies an edge list of variable length in time $m-i$, where $1 \leq i \leq m-3$. R-Chrom is more economical in that it moves $(m-n+i)$ edges to the next node. A-Chrom designed to efficiently take advantage of vectorization, has achieved a speedup of a factor of three. R-Chrom uses levels of indirection in referencing the edge list which inhibits vectorization. Since only minor loops were vectorizable, the vectorization speedup was limited to slightly less than two.

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Deletion</th>
<th>Contraction</th>
<th>Copying</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-Chrom</td>
<td>$\approx d_{\text{avg}}$</td>
<td>1</td>
<td>1</td>
<td>$n-i$</td>
</tr>
<tr>
<td>B-Chrom</td>
<td>$\Theta(d_{\text{avg}})$</td>
<td>1</td>
<td>$\Theta(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>NW-Chrom</td>
<td>1</td>
<td>1</td>
<td>$\Theta(d_{\text{avg}})$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>R-Chrom</td>
<td>1</td>
<td>1</td>
<td>$\Theta(d_{\text{avg}})$</td>
<td>$O(m-n)$</td>
</tr>
</tbody>
</table>

Table I: Complexities of primitives

4. Algorithm Performance

Table I summarizes, for all four algorithms, the time requirements (per step) of the primitives. All expressions for A-Chrom are valid only for values $n < 64$. As one can see, the overall time per step (say for contraction and copying) for A-Chrom is significantly less than that of B-Chrom and NW-Chrom and close to (or, for fairly sparse graphs, slightly greater than) that of R-Chrom. However, as the number of edges of a graph increases, the copying time of R-Chrom approaches $\Theta(n^2)$.

Both A-Chrom and R-Chrom have been implemented on a vector processor (a Cray X-MP/24). A-Chrom, designed to efficiently take advantage of vectorization, has achieved a speedup of a factor of three. R-Chrom uses levels of indirection in referencing the edge list which inhibits vectorization. Since only minor loops were vectorizable, the vectorization speedup was limited to slightly less than two.

In our experiments R-Chrom processed wheel-like graphs (with $d_{\text{avg}} = 4$) on up to 15 vertices. A-Chrom processed such graphs on up to 27 vertices with slightly better speed on comparable graphs. For "mid-density" graphs (with $m = n(n-1)/4$) R-Chrom ran out of space on graphs with over 10 vertices. A-Chrom handled such graphs on 15 vertices, and within a three minute time limit. The execution speed of A-Chrom on comparable mid-density graphs
was 20% higher than that of R-Chrom.

We now summarize space requirements for all four algorithms. B-Chrom, designed for dense graphs, utilizes a stack of contractions each of which is an \( n \times n \) matrix. The maximum stack size is for \( n-2 \) such matrices. Therefore B-Chrom requires \( \Theta(N^3) \) space. NW-Chrom, designed for sparse graphs, utilizes a stack of deletions of size \( \Theta(nm) \) [2].

R-Chrom, also designed for sparse graphs, is very economical in its space utilization by storing, for each node of the S-tree, only \( m-n-i \) edges. However, since R-Chrom is designed so that it must store all non-leaves of the S-tree, the space requirement grows exponentially with the size of the input graph. In A-Chrom, for an input graph on \( n \) vertices the maximum stack size is \( n-2 \) G-vectors of length \( n \), plus the three one-dimensional arrays of length \( n \). Thus the space requirement for the execution of A-Chrom is proportional to \( n^2 \), which is virtually optimal, i.e. linear in the size of input (quadratic in the number of the graph vertices).

A-Chrom, as well as B-Chrom, has a fixed space requirement, independent of the number of edges, and, with minor changes, both are applicable for processing sparse and dense graphs. R-Chrom would require significant changes and allocation of increasing space for copying queue elements while processing dense graphs.

Acknowledgements

We are very grateful to Dr. Ronald Read who graciously provided us with useful materials including his code for R-Chrom, which we used as a baseline in the benchmarking of our program. Participation in this work of Stephanie Vause, supported by NSF grant ASC-8900690 and Michael Chadzik supported by NSF grant CCR-8804250 has been very helpful. The Alabama Supercomputer Network (ASN) provided us with the necessary time on their Cray.

References


