# Worst-Case Analysis of Read's Chromatic Polynomial Algorithm

Timothy R. Walsh
Department of Computer Science, UQAM, Montreal, Quebec, Canada

**Abstract:** The worst-case time-complexity of Read's edge-addition/contraction algorithm for computing the chromatic polynomial of an n-vertex graph is shown to be $O(n^2B(n))$, where $B(n)$ is the nth Bell number, which grows faster than $c^n$ for any c but slower than n!. The factor $n^2$ can be reduced to n, and the space-complexity from $O(n^3)$ to $O(n^2)$, by storing in arrays the information needed to reverse each transformation made on the graph.

A graph G is assumed to be finite (with n vertices), unoriented and without loops or parallel edges. A *t-colouring* of G is a mapping from the vertex-set of G into the set {1,...,t} of colours so that no two adjacent vertices have the same colour. The *chromatic polynomial* P(G,t) of G is a function of t whose value for a given t is the number of t-colourings of G.

Once the polynomial P(G,t) has been calculated, one can decide in polynomial time if G has a t-colouring for a given t by substituting t into the polynomial: G has a t-colouring if and only if $P(G,t)>0$. Since the problem of deciding whether G has a t-colouring is NP-hard [1, p. 392], it is unlikely that a polynomial-time algorithm will be found for computing the chromatic polynomial of a graph. The most commonly-used algorithm is the one proposed by Read [3]. It has been referenced and improved upon in several places in the literature (see, for example [5]) but its worst-case time- and space-complexity have never been published. We show here that the space-complexity is $O(n^3)$ and the time-complexity is $O(n^2B(n))$, where B(n), the nth Bell number, grows faster than $c^n$ for any c but slower than n!, and we modify the algorithm so that it runs in $O(nB(n))$ time and uses $O(n^2)$ space.

The following facts are shown in [3] and in other places:

1. P(G,t) is a polynomial of degree n in t.

2. If G is a tree then $P(G,t)=t(t-1)^{n-1}$.

3. If G is not connected then the chromatic polynomial of G is the product of the chromatic polynomials of its connected components.

4. If G is a complete graph then

$$P(G,t) = t(t-1)(t-2)...(t-n+1). \qquad (1)$$

5. If G is not a complete graph, let x and y be any two non-adjacent vertices. Let G+{x,y} be the graph obtained from G by adding the edge {x,y} to G. Let G(x=y) be the graph obtained from G by identifying the vertices x and y,

where a vertex v is adjacent to the common vertex xy in G(x=y) if and only if v is adjacent to at least one of x and y in G. Then

$$P(G,t) = P(G+\{x,y\},t) + P(G(x=y),t). \qquad (2)$$

The algorithm in [3] for calculating P(G,t) is a recursive one based on (1) and (2). We give a non-recursive version (see Figure 1) for the purpose of analysing its space- and time-complexity.

```
Procedure PolyChrome(G: n-vertex graph;
                     C: ARRAY[1..n] OF INTEGER);
  FOR i:=1 TO n DO C[i]:=0 END FOR;
  m:=n; H:=G; (* H is the current graph, which has m vertices *)
  Stack:=EMPTY;
  LOOP
    IF H has two non-adjacent vertices x and y THEN
      PUSH H+{x,y} onto Stack;
      H:=H(x=y); m:=m-1;
    ELSE
      C[m]:=C[m]+1;
      IF m<n THEN
        POP from Stack into H; m:=m+1;
      ELSE
        EXIT
      END IF
    END IF
  END LOOP
END PolyChrome.
```

**Figure 1**
**A non-recursive version of Read's chromatic polynomial algorithm.**

We find P(G,t) from formula (3) below:

$$P(G,t) = C[1]t + C[2]t(t-1) + ... + C[n]t(t-1)...(t-n+1). \qquad (3)$$

We analyse the space- and time-complexity of this algorithm under the assumption that the graph is represented by an adjacency matrix.

The space-complexity is easy to analyse: at any time the stack will contain one graph with i vertices for each i from n down to m for a maximum of n graphs, each occupying $O(n^2)$ space, for a total of $O(n^3)$.

To analyse the time-complexity we note that everything done to a graph H - creating it, searching it for non-adjacent vertices, pushing it onto the stack and popping it off again - takes $O(n^2)$ elementary operations, so that the total number of operations performed is $O(n^2)$ multiplied by the number of graphs the algorithm has to treat. The branching tree for this algorithm is a binary tree whose nodes are the graphs H the algorithm has to treat, with G as its root and the complete graphs as its leaves, and each internal node H for which the non-adjacent vertices x and y have been found has H+{x,y} and H(x=y) as its children. Each time the algorithm treats a complete graph it increases C[m] by 1 for some m, $1 \le m \le n$, so that the number of complete graphs it treats is

$$C[1] + C[2] + \dots + C[n], \tag{4}$$

and since the total number of nodes of a binary tree is twice the number of its leaves minus 1, the total time complexity of the algorithm is O($n^2$ multiplied by the sum in (4)).

The worst case of this algorithm clearly occurs when G has no edges. In this case $P(G,t)=t^n$. Substituting this value of P(G,t) into (3) we find that for each value of k, C[k]=S(n,k), a Stirling number of the second kind [4, p. 33]. Substituting these values into (4) we obtain the Bell number B(n). The worst-case time-complexity of this algorithm is thus $O(n^2B(n))$.

There is an asymptotic estimate for B(n) in [2]:

$$B(n) \sim (R+1)^{-1/2}\exp[n(R+R^{-1}-1)-1], \tag{5}$$

where

$$Re^R = n. \tag{6}$$

From (6) it is clear that R approaches $\infty$ as n approaches $\infty$, so that, from (5), B(n) grows faster than exp(nc) for any constant c, and thus faster than $c^n$ for any c. On the other hand, from (5) it is also clear that B(n) grows more slowly than $\exp(nR)=(n/R)^n$ (from (6)); dividing by Stirling's approximation $(2\pi)^{1/2}e^{-n}n^{n+1/2}$ for n! we get $(e/R)^n/(2\pi n)^{1/2}$, which tends to 0 as n tends to $\infty$, so that B(n) grows more slowly than n!.

```
Procedure FastPolyChrome(G: n-vertex graph (n by n matrix);
                         C: ARRAY[1..n] OF INTEGER);
  FOR y:=1 TO n DO
    C[y]:=0;
    D[y]:=number of vertices x<y adjacent to y   (* O(n) time *)
  END FOR;
  E:=EMPTY;       (* E is a 2-column stack for storing pairs of identified vertices *)
  FOR m:=2 TO n DO P[m]:=EMPTY END FOR;
                  (* P[m] is a 2-column stack for edges added to m-vertex graph *)
  m:=n; y:=n;                       (* End of initialization in O(n^2) time. *)
  LOOP                              (* each iteration treats a new graph *)
    WHILE (y>1) AND (D[y]=y-1) DO y:=y-1 END WHILE;
(* At this moment the vertices of the current graph H are all the m vertices not in the second
column of E. These vertices include 1,2,...,y, and all the others are adjacent to each other and to
each of 1,2,...,y in H. The edges of H are all the edges in the subgraph of G induced by these
vertices plus the edges in P[m],...,P[n] whose endpoints are in H. *)
    IF y>1 THEN                     (* there is a vertex x<y not adjacent to y *)
      x:=largest vertex <y not adjacent to y;(* O(n) time *)
      PUSH (x,y) onto E;            (* y is going to be 'deleted' from H *)
      FOR i:=y-1 DOWNTO x+1 DO      (* i must be adjacent to y *)
        IF i is not adjacent to x THEN
          make i adjacent to x;
          PUSH (x,i) onto P[m-1]; D[i]:=D[i]+1;
        END IF;
      END FOR;
      FOR i:=x-1 DOWNTO 1 DO
        IF i is adjacent to y but not to x THEN
          make i adjacent to x;
          PUSH (i,x) onto P[m-1]; D[x]:=D[x]+1;
        END IF;
      END FOR;                      (* H has been transformed into H(x=y) *)
      m:=m-1;            (* y is deleted from H but not from its n by n matrix! *)
      y:=y-1;                (* to start search for new pair (x,y) from y-1 *)
    ELSE                                              * H is an m-clique *)
      C[m]:=C[m]+1;
      WHILE P[m] NOT EMPTY DO
        POP from P[m] into (x,y);
        make x and y non-adjacent; D[y]:=D[y]-1;
      END WHILE;(* all edge-additions to H (and one contraction if m<n) reversed *)
      IF m<n THEN
        POP from E into (x,y);(* y and its incident edges are still in the matrix
                    and H has been restored from H(x=y) for the vertex pair (x,y) *)
        m:=m+1;
        make x and y adjacent; D[y]:=D[y]+1;
        PUSH (x,y) onto P[m];   (* H has been transformed into H+{x,y} *)
      ELSE              (* the original graph G has been restored for future use *)
        EXIT
      END IF m<n
    END IF y>1
  END LOOP
END FastPolyChrome.
```

**Figure 2: A faster version of the algorithm in Figure 1.**

A modification suggested in [3] which appears to make the algorithm more efficient is to apply it to each connected component and then multiply the resulting chromatic polynomials. The worst case of this modified algorithm is a tree, with chromatic polynomial $t(t-1)^{n-1}$. A similar substitution into (3) shows that (4) evaluates to B(n-1). This means that the modified algorithm takes as much time to treat the worst-case graph on n vertices as the original algorithm takes to treat the worst-case graph on n-1 vertices - a gain in efficiency, but not a dramatic one.

Another alternative suggested in [3] is to modify (2) to

$$P(G,t) = P(G-\{x,y\},t) - P(G(x=y),t), \qquad (7)$$

so that in the algorithm we are deleting rather than adding edges, terminating with empty graphs rather than complete ones, adding $(-1)^{n-m}$ to C[m] and interpreting C[m] as the coefficient of $t^m$ in P(G,t). The worst-case time-complexity is now O($n^2$ times the sum of the absolute values of the coefficients C[m]). The worst case is now the complete graph with chromatic polynomial t(t-1)...(t-n+1). The coefficient of $t^k$ in this product is s(n,k), a Stirling number of the first kind [4, p. 33], and the sum over k of their absolute values is n!. Since B(n) grows more slowly than n!, deleting edges will, in the worst case, take longer than adding them. If we modify the edge-deletion algorithm so that it terminates with trees rather than empty graphs, then the worst-case time-complexity is (n-1)! instead of n!, so that again the modified algorithm takes as much time to treat the worst-case graph on n vertices as the original algorithm takes to treat the worst-case graph on n-1 vertices. In particular, this modification cannot make edge-deletion competitive with edge-addition in terms of worst-case time-complexity.

We propose here a modification to the edge-addition algorithm (see Figure 2 above) which reduces the space-complexity from O($n^3$) to O($n^2$) and the time-complexity from O($n^2$B(n)) to O(nB(n)), a more significant reduction than working with connected components since, as we have seen, B(n)/B(n-1) grows more slowly than n. This is possible because only O(n) operations are actually performed in passing from H to H(x=y). We store the pair (x,y) in a 2-column stack E. Each edge added, either in identifying x with y or joining x to y, is stored in a 2-column stack P[m], where m is the number of vertices in the new graph. The stack E has one pair for each i from n down to m, the size of the current graph; so that E has n-m+1 pairs and takes O(n) space. The stacks P[m] are disjoint from each other and from the edge-set of G, so that together they occupy O($n^2$) space, making the total space-complexity O($n^2$). Whenever a given P[m] is being treated, all the stacks P[i], i<m, are empty; so the P[m] can all be combined into a single (array-implemented) stack P with P[n] on the bottom, P[n-1] on top of it and so on, provided that the sizes of the P[m] are stored in a size-n array. This is the approach we used in our C-implementation of this algorithm (a listing is available upon request). The time-cost of transforming H into H(x=y) can easily be seen to be O(n) by examining the part of the code between IF y>1 and the matching ELSE. The WHILE loop after the ELSE may be longer than O(n), but if we charge the cost of popping an edge from P[m] to the graph to which we added that edge when we pushed it onto P[m], then the total cost is O(n) per graph treated, or O(nB(n)) altogether in the worst case as claimed.

As mentioned above, these estimates are made under the assumption that the graph is represented by an adjacency matrix. No efficiency gain is achieved by representing the neighbours of each vertex by a list, since the complete graph of size m must be treated for every m from n down to the chromatic number of G. However, if the complement of G is sufficiently sparse, some space can be saved by representing the non-neighbours of each vertex by a list. The total space-complexity is now O(n+e'), where e' is the number of edges in the complement of G (O(e') is a sharper bound than $O(n^2)$ on the size of P). The algorithm in Figure 2 can be applied to this data structure to work in time O(d') per graph treated, where d' is the maximum valency of the complement of G, but since the worst case occurs when the complement of G is dense rather than sparse, the worst-case time-complexity will not be improved by this modification and we have not attempted it.

Another modification is presented in [5]: delete edges which are part of a chordless cycle and terminate with graphs with no chordless cycles - *chordal graphs* - for which the chromatic polynomial can be easily calculated. Alternatively, one could add chords to chordless cycles and terminate with chordal graphs. Possibly one of these two algorithms, or an adaptive approach which deletes edges from sparse graphs and adds edges to dense graphs, will turn out to have an exponential rather than a super-exponential worst-case time-complexity, which would make it a significant improvement over any of the algorithms presented in [3]. We leave the analysis of these algorithms as an open problem.

## REFERENCES

1: A. Aho, J. Hopcroft and J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Wokingham, Berkshire, U.K., 1974.

2: L. Moser and M. Wyman, An asymptotic formula for the Bell numbers, Transactions of the Royal Society of Canada, Volume XLIX, Series III, Section 3 (1955), 49-54.

3: R. Read, An introduction to chromatic polynomials, Journal of Combinatorial Theory 4 (1968), 51-71.

4: J. Riordan, An Introduction to Combinatorial Analysis, J. Wiley and Sons, New York and London, 1958.

5: D.R. Shier and N. Chandrasekharan, Algorithms for computing the chromatic polynomial, Journal of Combinatorial Mathematics and Combinatorial Computing 4 (1988), 213-222.