Journal of Mathematical Chemistry 18 (1995) 115–119

A fast computer algorithm for finding the permanent of adjacency matrices

Gordon G. Cash

Environmental Effects Branch, Health & Environmental Review Division, Office of Pollution Prevention & Toxics, U.S. Environmental Protection Agency, 401 M Street, S.W., Washington, DC 20460, USA

Received 13 February 1995; revised 27 June 1995

A fast computer algorithm is described which brings computation of the permanents of sparse matrices, specifically, chemical adjacency matrices, within the reach of a desktop computer. Examples and results are presented, along with a discussion of the relationship of the permanent to the Kekulé structure count. Also presented is a C-language implementation which was deliberately written for ease of translation into other high-level languages.

The permanent of a matrix is conceptually similar to the determinant. The determinant of an $N \times N$ matrix is the sum of all possible products of elements $\sum(-1)^z a_{i,1}, a_{j,2}, a_{k,3} \ldots a_{x,N}$ such that $i, j, k \ldots$ are all different, and z is odd for half of the terms and even for the other half. The expression for the permanent is nearly the same but lacks the $(-1)^z$ multiplier (fig. 1). This report describes a fast computer algorithm which finds exact values for permanents of sparse matrices, with examples of chemical adjacency matrices up to 44×44, in a reasonable time on a desktop computer.

For a general $N \times N$ matrix **A**, per(**A**) is the sum of $N!$ terms, each of which is the product of N matrix elements. This is also true for the determinant $|\mathbf{A}|$, but elegant methods exist for shortening the calculation of $|\mathbf{A}|$. Some shortcuts also exist for per(**A**), but these give only bounds or estimates [1]. Since $20! \approx 2.4 \times 10^{18}$, com-

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

$$|\mathbf{A}| = +(2\times5)-(4\times3) = 10-12 = -2$$

$$\mathbf{per(A)} = +(2\times5)+(4\times3) = 10+12 = 22$$

Fig. 1. The determinant and permanent of a 2×2 matrix.

puting per(**A**) by brute force for any matrix of reasonable size clearly exceeds the capability of any machine existing or planned. It may be possible, however, to exploit the properties of unusual matrices to reduce this computation to a manageable size. The present study presents a technique for sparse matrices and uses as an example the adjacency matrix, a matrix widely utilized in chemical graph theory.

The simplest form of the adjacency matrix **A** is just an atom connectivity table in matrix form. Thus, $a_{i,j} = a_{j,i} = 1$ if atoms i and j are directly bonded; all other elements are zero. Hydrogen atoms are typically excluded. Therefore, for molecules of reasonable size almost all elements of **A** are zero, and all of its non-zero elements are one. Thus, only a tiny fraction of the $N!$ terms that make up per(**A**) for these matrices are non-zero, and those that are not are all $1^N = 1$. To calculate per(**A**), therefore, it is not necessary even to add the terms but only to count the non-zero terms. These facts are the key to the computer algorithm presented below.

It has been stated [2] that per(**A**) is equal to the square of the Kekulé structure count K for conjugated systems, but there seems to have been no systematic attempt to verify that assertion. For benzenoid hydrocarbons, $\det(\mathbf{A}) \equiv |\mathbf{A}| = K^2$, but this relationship is not generally true for other types of molecules. The author has determined that $\text{per}(\mathbf{A}) = K^2$ for some hydrocarbons for which $|\mathbf{A}| \neq K^2$ (fig. 2), but that equality fails for many nonalternant molecules such as fullerenes. (The author is conducting a study of the significance of per(**A**) for fullerenes, which will be published separately.) In any case, Klein and Liu [3] have illustrated a method due to Kasteleyn [4] for calculating K from what they term a signed adjacency matrix **S**. They claim this method is applicable to any planar chemical graph. The utility of per(**A**) values for adjacency matrices is likely to be elsewhere than determination of K values, but these other areas remain to be explored. The author hopes that the algorithm presented here will encourage such exploration.

For an adjacency matrix, the permanent computation reduces to finding the number of different products of matrix elements $a_{i,1}, a_{j,2}, a_{k,3} \ldots$ such that the elements are all equal to one and the first subscripts are all different, i.e., no two elements are in the same row. For each such combination that exists, the permanent increases by 1.

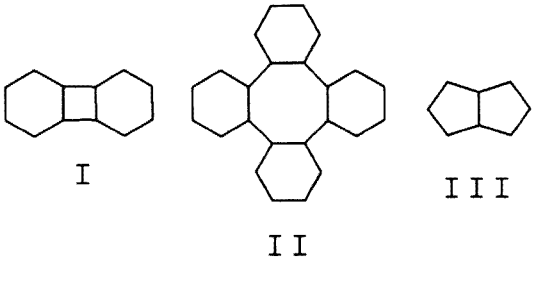

Fig. 2. I, Biphenylene; II, tetraphenylene; III, pentalene.

The flowchart for a computer algorithm that finds these combinations is shown in fig. 3. This algorithm first examines all $a_{i,1}$ until it finds an $a_{i,1} = 1$. Then, after storing the "active" i, it examines all $a_{j,2}$ until it finds an $a_{j,2} = 1$. Next, it checks to see that $i \neq j$. If $i = j$, it proceeds to the next $a_{j,2} = 1$; if not, it begins searching for an $a_{k,3} = 1$, keeping track of the "active" i and j so it can verify that $i \neq k$ and $j \neq k$. Whenever the tree thus grown reaches all the way to the last column in the matrix, the algorithm increments a counter. If the non-zero elements were other than one, it could compute a product and update a running total. The key to examining all possible combinations in a reasonable number of CPU cycles is timely pruning of the tree. Once the algorithm finds an $a_{i,1} = 0$, it never examines another tree beginning with that $a_{i,1}$. Similarly, if $a_{i,1} = 1$ but $a_{j,2} = 0$, it never examines another tree beginning with that $a_{i,1}$, $a_{j,2}$. Thus, the tree with $N!$ potential branches is quickly pruned to a manageable size.

There is no apparent general relationship between CPU time for this algorithm and the size of the problem. For fullerenes, it appears roughly true that each addition of two vertices increases CPU time by a factor of three. Nevertheless, the author has observed cases in which CPU times for two fullerenes of the same size differ by a factor of two, even when the values of per(A) were nearly equal. In general, however, it is not even clear that larger matrices necessarily take more time than smaller ones. The number of non-zero elements in the matrix is surely an important factor. Thus, on a machine equipped with an 80486 DX/33 CPU, a test

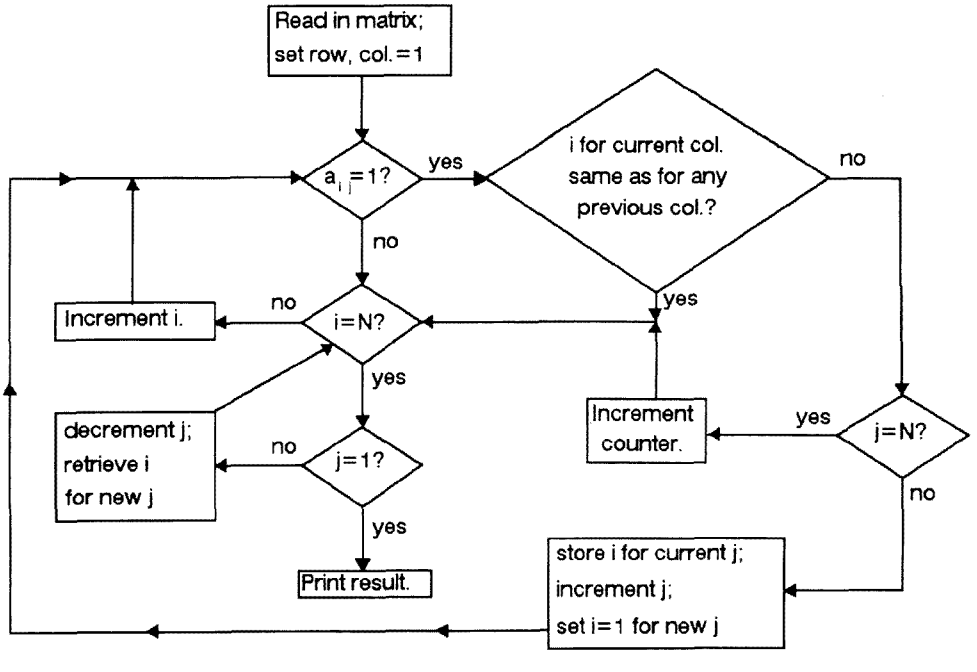

Fig. 3. Flowchart of algorithm for finding per(A).

```
/* Program to find the permanent of an adjacency matrix */
#include <stdio.h>
main()
{
     int a[150][150];          /* the adjacency matrix */
     int b[150];              /* stores active row number for each
                                 column */
     int d = 1, n, i, j, f, e;  /* various counters */
     long per = 0;             /* permanent counter
*/
     scanf("%d",&n);           /* read order of the matrix */
     for  (i=1;i<=n;i++){
          for  (j=1;j<=n;j++)
               scanf("%d",&a[i][j]);    /* get a matrix element */
     }
     s1: b[d]=1;               /* initialize row counter for
                                 column d */
     s2: if (a[b[d]][d])       /* a(i,j) = 0? */
            goto s7;          /* if not, test same row in previous
                                 columns */
     s3: if (b[d] == n)        /* i = N (last row)? */
            goto s5;          /* if yes, check whether you are in
                                 column 1 */
     (b[d])++;                 /* increment row number */
     goto s2;                  /* branch back to a(i,j)=0? for new
                                 row */
     s5: if (d == 1)           /* already in column 1? */
            goto s14;         /* if yes, then exit */
     d--;                      /* if no, go back to previous
                                 column */
     goto s3;                  /* go back to check for last
                                 (ultimate) row */
     s7: f=0;                 /* reset flag for same row, previous
                                 column */
     for (e=1; e<d; e++){      /* test same row, previous columns
                                 (up to d-1) */
         if (b[e] == b[d]){    /* if yes,        */
            f++;               /* set flag       */
            break;             /* then exit loop */
         }
     }
     if (f)                    /* if flag is set */
        goto s3;               /* go back to last row test  */
     if (d == n){              /* otherwise, if in last col */
        per++;                 /* increment permanent       */
        goto s3;               /* then resume testing rows  */
     }
     d++;                      /* if not in last column, increment
                                 column */
     goto s1;                  /* then test new column */
     s14: printf("%ld\n",per); /* print result */
}
```

Fig. 4. An implementation of the algorithm in the C programming language. This implementation deliberately avoids programming features, such as pointers, that are peculiar to C in order to facilitate translation into other languages.

matrix for a C_{44} non-alternant hydrocarbon with 22 divalent and 22 trivalent vertices gave per(**A**) = 1716 in 2.5 hours, while obtaining per(**A**) = 2436480 for a C_{44} fullerene took approximately 360 hours. (Fullerenes necessarily have all vertices trivalent.) Since $44! \approx 2.7 \times 10^{54}$, however, this latter result represents a reduction of tens of orders of magnitude in the resource demands of the problem. Increases in speed might be realized from further optimizing the computer code, and processors faster than the 80486 DX/33 are now generally available.

The algorithm presented above allows, in particular, rapid computation of per(**A**) for adjacency matrices of molecules large enough to be theoretically interesting. The specific implementation in the C language [5] was deliberately written for ease of translation into other high-level languages. This algorithm should be readily adaptable to other types of sparse matrices, including those with elements other than zero and one, thus permitting wider exploration of the significance and uses of permanents.

References

[1] H. Minc, in: *Encyclopedia of Mathematics and Its Applications*, Vol. 6, ed. G.-C. Rota (Addison-Wesley, Reading, MA, 1978) pp. 1–119.

[2] (a) N. Trinajstić, *Chemical Graph Theory*, 2nd Ed. (CRC, Boca Raton, FL, 1993) pp. 167-168;
(b) D.M. Cvetković, I. Gutman and N. Trinajstić, Chem. Phys. Lett. 16 (1972) 614;
(c) J.K. Percus, J. Math. Phys. 10 (1969) 1881.

[3] D.J. Klein and X. Liu, J. Comput. Chem. 12 (1991) 1260.

[4] (a) P.W. Kasteleyn, J. Math. Phys. 4 (1963) 287;
(b) P.W. Kasteleyn, in: *Graph Theory and Theoretical Physics*, ed. F. Harary (Academic Press, New York, 1967) chap. 2.

[5] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, 2nd Ed. (Prentice-Hall, Englewood Cliffs, NJ, 1988).