reason I think the small machine will always have a certain use in companies of a certain size. The problems of sharing a large machine between several autonomous users are not just technical problems concerned with parallel programming and time-sharing; they involve problems of conviction and competence, and the secrecy of data supplied by different clients for the use of one machine.

Competence in the technical activity of putting several jobs on to the machine and sharing time by micro-seconds does not overcome sociological problems and business administration problems. These must be solved, however, before the type of commercial time-sharing which Dr Wilkes foresees can come about. Therefore, for a long time to come, I believe there will be a future for the small machine.

References

1. A. W. Burks, H. H. Goldstine, and J. Von Neumann. '*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*. Institute for Advanced Study, Princeton, N.J. (1946).
2. J. Von Neumann. '*First Draft of a Report on the EDVAC*.' Moore School of Electrical Engineering, University of Pennsylvania (1945).
3. M. V. Wilkes. 'The second decade of computer development'. *The Computer Journal*, **1** (1), 98 (1958).

Short Notes

Implementation of Karp–Luby Monte Carlo Method: An Exercise in Approximate Counting

Richard Karp and Michael Luby introduced a powerful framework for the construction of Monte Carlo algorithms to solve hard counting problems [cf. *Journal of Complexity* 1 (1), 45–64 (1985)]. They then applied it, as a special case, to the problem of counting the number of satisfying truth-value assignments for a Boolean formula in disjunctive normal form. In this paper, we describe an implementation of that algorithm. Our experiments show that it indeed works very well in practice.

Received June 1987

1. Introduction

Counting is difficult. To quote Valiant:[1]

> 'Numerous problems in the mathematical and physical sciences can be reduced to questions of counting solutions in combinatorial structures. Much effort has been put into developing analytic techniques for doing this effectively for the various problems that arise most frequently. A glance at the literature, however, suggests that the search for positive results has had only very limited success, and that for the majority of questions we still cannot count exactly in any effective sense'

Similarly, while estimating the number of simple (non-self-intersecting) paths between two corners of a grid graph, Knuth observes:[2]

> 'Of course, I have only generated an extremely small fraction of these paths, so I cannot really be sure; perhaps nobody will ever know the true answer.'

In complexity theory, the notion of # P-completeness formalises the difficulty of counting problems. Introduced by Valiant, this class typically contains problems polynomially equivalent to the counting problems associated with many NP-complete problems such as counting the number of Hamilton circuits in a graph. The reader is referred to Gary and Johnson,[3] and Valiant[4] for the fundamentals of NP- and # P-completeness. Angluin[5] and Stockmeyer[6] offer several theoretical results about # P. Problems which are # P-complete are at least as hard as NP-complete problems, making it unlikely that a polynomial algorithm exists to solve them.

Fortunately, the prospects are not as dim as they look, mainly due to tools like Karp and Luby's innovative Monte Carlo framework that offers an attractive alternative for several problems of this sort.[7] (Hammersley and Handscomb[8] give an overview of Monte Carlo techniques. Fishman[9] studies such techniques in reliability area.) Using randomisation, Karp and Luby propose fast algorithms which report an 'almost correct' answer 'almost surely', provided that one is willing to spend an extra effort to make the quoted notions more and more refined. Specifically, such an algorithm returns after a polynomial effort in the problem size, ε, and δ, an answer A^* for a counting problem whose exact answer is A such that

$$\text{Prob}\{A^{-1}|A^* - A| > \varepsilon\} < \delta$$

Here $||$ denotes the absolute value.† In other words, the method computes the answer within *relative error* at most ε and attaches to it a confidence value of at least $1-\delta$. Statistically speaking, only 100δ percent of the time the returned result would not obey the relative error bound. Here. ε and δ are small positive constants specified by the user. Note however that such algorithms generally have to make $(p\varepsilon^{-1}\log\delta^{-1})^{O(1)}$ trials where p is a measure of the problem size. Thus, while one can theoretically choose ε and δ as close to zero as required, it is prohibitively expensive to use very small values.

In this paper, we demonstrate that this is not a serious issue since, using liberal values like $\varepsilon = 0.1$ and $\delta = 0.1$ will provide the almost right answer. Therefore, there is essentially no need to resort to conservative values like say, $\varepsilon = 0.001$ and $\delta = 0.001$. Specifically, we describe an implementation of a Monte Carlo algorithm by Karp and Luby for counting the number of satisfying truth-value assignments for a Boolean formula given in disjunctive normal form.[10] For brevity, we shall frequently cite results from Ref. 10 without elaboration. The reader is asked to consult that paper for a complete description.

2. Problem definition, notation and Karp–Luby Algorithm

We closely follow the notation of Ref. 10. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean *variables*. Thus each x_i can be either 0 or 1. The members of $X \cup \bar{X}$ where $\bar{X} = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$ are called *literals*. Here $\bar{x}$ denotes the complement of x. A *clause* is a logical 'and' of a set of literals. A *disjunctive normal form* (*DNF*) *formula* is the logical 'or' of a set of clauses. A *truth-value assignment* is a function f from X to $\{0, 1\}$. A truth-value assignment f is a *satisfying truth-value assignment* for a given DNF formula F if F evaluates to 1 upon substitution of $f(x_i)$ for each variable x_i.

Let F be given as $\bigcup_1^m C_i$ where C_i's are the clauses. Let N_F denote the number of truth-value assignments satisfying F. In this paper we shall deal with the problem of computing N_F exactly or approximately. (In the latter case it will be denoted by N_F^*.) This problem will be called CNTSAT in the sequel. It is known that CNTSAT is # P-complete. Exact but inefficient ways of computing N_F will be postponed until Section 3. Now we shall briefly summarise the Karp–Luby algorithm to compute N_F^*.

† In the sequel, it will also be used to denote the number of elements in a set but no confusion will arise.

The algorithm of Karp and Luby to approximately solve CNTSAT is based upon the following crucial observation:

> Let the universe S be the set of all tuples (i, x) such that x is a truth-value assignment yielding $C_i = 1$. Let R be the set of all those tuples (i, x) such that C_i is the lowest-numbered clause satisfied by x. Then $|R| = N_F$.

A *trial* of the algorithm consists of drawing a member of S randomly and testing whether it lies in R. Let us assume, without loss of generality, that the clauses contain no contradictory pair of literals or no repetitions of the same literal.

The algorithm starts initialising N, to 0 and computing $|C_i|$'s and $|S|$. (By a slight abuse of notation, $|C|$ denotes the number of satisfying truth-value assignments for clause C.) Computing the former values is trivial, i.e. $|C_i| = 2^{n-k}$ where k is the number of literals occurring in C_i. (Remember our assumption in the preceding paragraph.) It is also noted that $|S| = \sum_1^m |C_i|$ and $|R| = |\bigcup_1^m C_i|$. Thus $|S|/|R|$ is at most equal to m. This bound is essential since using Bernstein's inequality Karp and Luby prove that a total of

$$N = \text{ceil}\,(|S||R|^{-1}\ln(2\delta^{-1})4.5\varepsilon^{-2})$$

Trials would be required for the Monte Carlo experiment. At each trial, the algorithm computes a tuple (i, x) randomly, as noted above. It then determines the lowest-numbered clause C_l satisfied by this x. If $l = i$ it increments the number of successful trials N_s by one. The final step of the algorithm is to report $N_s N^{-1}|S|$ as the answer N_F^*. By the nature of the method, this answer is guaranteed to be 'good' by the formula

$$\text{Prob}\{N_F^{-1}|N_F - N_F^*| > \varepsilon\} < \delta$$

Thus, with probability at least $1-\delta$, the value N_F^* reported by the algorithm is a fine guess for N_F, i.e. it is off at most with relative error ε.

As for the complexity of the algorithm, Karp and Luby proved that $O(m^2 n)$ is the

bound. Briefly, the algorithm makes $N = O(m)$ trials each taking $O(mn)$ time. Clearly, the algorithm's set-up time is only $O(mn)$.

3. Implementation and computational experience

Our implementation consists of three programs: COMB, CRIBLE and KL. They are all written in Franz Lisp[11,12] running under UNIX‡ and are available upon request. The first two programs, verify the answer provided by the last one. We now give a brief description of each program.

COMB reports N_F exactly. It does so by generating all combinations of the variables and testing each combination to see if it satisfies F. *since there are* 2^n combinations to be tested, this method is feasible only when n is small.

CRIBLE also reports N_F exactly. It employs the well-known 'formule du crible' attributed to Silva and Sylvester.[13] Briefly, let $B_1, B_2, \ldots, B_m$ be finite sets. Then the formula states

$$\left|\bigcup_1^m B_i\right| = \sum_{I \in \pi - \varphi} (-1)^{|I|+1} \left|\bigcap_{i \in I} B_i\right|$$

where π denotes the power set of $\{1, 2, \ldots, m\}$. Note that in our case, $B_i = C_i$. Again, since there are $2^m - 1$ terms, contributing to the sum, this method is feasible only when m is small.

KL reports N_F approximately and with an attached probability using the algorithm explained in the previous section. Initially, the DNF formula F is in a file. In addition to F, the file contains the values of n, ε, and δ. For example, the following is an image of the file which contains the formula

$$F = x_1 x_4 \cup x_1 \overline{x_2} x_3 \cup x_1 x_2 x_5 \cup x_1 x_3 \overline{x_4 x_5},$$

a simple test for Karp–Luby algorithm.

This example is taken from Ref. 10;

```
(setq n 5)
(setq F'((1 4)(1 -2 3)(1 2 5)(1 3 -4 -5)))
(setq delta 0.1)
(setq eps 0.1)
```

(Note how the complements are denoted by negative numbers.)

As soon as this file is read by KL, m is determined and a check is made to see if the formula is 'valid'. After this, depending on the values of n and m, either COMB or CRIBLE is used to compute the exact value of N_F. (Obviously, for large values of n and m, neither will be satisfactory.) Then KL is called to compute N_F^*.

Whenever KL needs to make a binary choice, it uses the Lisp function *random*. Normally, (*random UpperBound*) returns a random number between 0 and *UpperBound* − 1. Thus, a good way to get a 'well-mixed' sequence of 1s and 0s is to use *random* 1001) and regard returned values less than 500 as 0s and values equal or greater than 500 as 1s. (N.B. Franz Lisp returns a deterministic sequence for (*random* 2)!) KL also needs a way to generate random reals between 0 and 1. It was found that calling (*random* 1001) and dividing the result by 1000 was good enough.

‡ UNIX is a registered trademark of A. T. & T. Bell Laboratories.

Finally, KL needed a way to select clause i with probability $|C_i|/|S|$. This was, following Karp and Luby's 'trick' mentioned in Ref. 10, implemented as follows. Create an array $TABLE$ of n entries with the following contents

$$TABLE[i] = |S|^{-1} \sum_1^i |C_i|.$$

To select a clause with a probability as mentioned above, we first get a random real r between 0 and 1. Then, using binary search we find the least j such that $TABLE\ [j] \geqslant r$. Clause j is the required clause. This method works extremely well in practice using the real number generator described above.

The test cases tried by KL are listed in Appendix 1. Table 1 in Appendix 2 Summarises the results of these tests. Table 2 in appendix 2 shows the timing information for the undertaken cases. All timing information in this paper is in CPU seconds excluding the garbage collection. Further experiments with individual formulas are tabulated in Tables 3(*a*)–(*g*) in Appendix 3. These are important to appreciate the robustness and efficiency of KL. In all the examples we have tested, the answers were always very close to the exact answer. Notice that using KL, we were able to answer some tough questions for COMB and CRIBLE such as F_9, F_{10}, and F_{11}. However, due to the proven efficiency of its underlying algorithm, in this paper our main goal was to establish the field performance of KL, in terms of closeness to the real answer using pseudo-random number generators such as the one provided by Lisp. (Thus, in a way we are not really too interested in the timing figures since we already know KL's asymptotic worst-case performance.) We believe that we have proved that KL is extremely robust in that sense.

4. Summary

We have given an implementation of a Monte Carlo algorithm due to Karp and Luby[10] to solve CNTSAT, i.e. the problem of counting the number of satisfying truth-value assignments for a Boolean formula in disjunctive normal form. Our experience* shows that the algorithm is indeed very effective.

Two conclusions can be drawn from this study. The first is that Monte Carlo methods which employ approximation and randomisation hand in hand are very useful and should be used under appropriate circumstances.† (Another encouraging clue as to the value of such methods is the emergence of so called 'simulated annealing' or 'cooling' techniques[15] to solve NP-complete problems approximately.) The general conclusion is that, despite the general opinion, the complexity theory is offering down-to-earth and practicable algorithmic advice.

Acknowledgement

This work has been completed while the author was a visiting researcher with the University of Utrecht, Department of computer Science, Utrecht, the Netherlands.

* Luby himself has also implemented several algorithms in his Ph.D. dissertation.[16]

† Cf. Karp's Turin lecture for his views on this subject.[14]

V. AKMAN*
Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, the Netherlands.

References

1. L. G. Valiant, Negative results on counting. In *Lecture Notes in Computer Science 67 (Proceedings of the 4th GI Conference, Aachen, W. Germany)*, edited K. Weihrauch, pp. 38–46. Springer-Verlag (March 1979).
2. D. E. Knuth, Mathematics and computer science: Coping with finiteness. *Science* **194**, 1235–1242 (1976).
3. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco (1979).
4. L. G. Valiant, The complexity of enumeration and reliability problems. *SIAM Journal on Computing* **8** (3), 410–421 (1979).
5. D. Angluin, On counting problems and the polynomial time hierarchy. *Theoretical Computer Science* **12**, 161–173 (1980).
6. L. Stockmeyer, On approximation algorithms for # P. *SIAM Journal on Computing* **14** (4), 849–861 (November 1985).
7. R. M. Karp and M. Luby, Monte Carlo algorithms for enumeration and reliability problems. *Proceedings of the 24th IEEE Foundations of Computer Science Symposium*, pp. 56–64 (1983).
8. J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*. Methuen, London (1964).
9. G. S. Fishman, *A Monte Carlo sampling plan for estimating reliability parameters and related functions*. Tech. Rep. UNC/ORSA/TR-85/7, Curriculum in Operations Research and Systems Analysis, University of North Carolina at Chapel Hill (June 1985).
10. R. M. Karp and M. Luby, Monte Carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity* **1** (1), 45–64 (1985).
11. J. K. Foderaro and K. L. Sklower, *The FRANZ LISP Manual*. University of California, Berkeley (September 1981).
12. R. Wilensky, *LISPcraft*. W. W. Norton & Company, New York (1984).
13. L. Comtet, *Analyse Combinatoire (Tome Second)*. Presses Universitaires de France, Boulevard Saint-Germain, Paris (1970).
14. R. M. Karp, Combinatorics, complexity, and randomness. *Communications of the ACM* **29** (2), 98–109 (February 1986).
15. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, Optimisation by simulated annealing. *Science* **220** (4598), 671–680 (13 May 1983).
16. M. Luby, Monte Carlo methods for estimating system reliability. Tech. Rep. UCB/CSD 84/168. Computer Science Division, University of California, Berkeley (June 1983).

* Now at: Department of Computer and Information Science, Bilkent University, PO Box 8 06572, Maltepe, Ankara, Turkey.

APPENDIX 1
List of tested Boolean formulas

Below, we list the formulas used to test KL. For other parameters such as n the reader is referred to Table 1 in Appendix 2. For ease of typesetting, we choose to stick here to the notation used by KL (cf. Section 3).

- F_1 = ((1 2 3) (1 −2 −3) (−1 −2 −3) (−1 2 3) (3 6) (3 −7) (6 7) (1 2) (1 3 −4) (4 7) (4 5 −6) (2) (3 6) (1 3 −7))
- F_2 = ((1 2 5) (1 4 −8 9) (2 3 4 5 6 −10) (1 −3 4 5) (1 2 3 4 5 6 7) (−2 −3 −4) (−2))
- F_3 = ((1 4) (1 −2 3) (1 2 5) (1 3 −4 −5))
- F_4 = ((1 2 3 −7) (1 2 −3 4) (1 −4 5 7) (1 −2 3 −4) (−1 2 6 7) (−1 −2 3 4) (−1 −2 −3 4) (−1 2 −3 4))
- F_5 = ((1) (−1) (2) (−2) (3) (3) (5) (−5) (8) (−8))
- F_6 = ((1 −2 −3) (1 5 −6) (3 4 5) (1 2 3 −4) (1 2 3 4 5 6) (2 −3))
- F_7 = ((−1 2 −3) (−2 −3) (−2 −3 4) (1 4 −5 −6) (1 2 3 4 5) (2 3 4 5 6))
- F_8 = ((1 2) (2 3 4) (4 5 6) (5 6 7) (−1 −2 −3) (−2 −7) (5 6))
- F_9 = ((1 −2 −11) (1 2 3 4 −5 −9 17) (1 2 3 4 5 6 7 8 9 11 −19 20) (1 3) (1) (1 7 9 13 15 −16 −17 −20) (1 4 5 6 −8 −10 −11 −12 −13) (1 2 3 4) (1 5 −19 −20) (1 4 6 8 9) (1 −4 −5 6 −7 −12 −13) (1 20) (1 −19 20) (1 2 20) (1 11 12 13 14 18 −19) (1 10 20) (1 13 14 16) (1 2 3 4 5) (1 −2 −3 −4 −5) (1 2 6 8 −10))
- F_{10} = ((1) (1 2) (1 2 −4) (1 10) (1 2 3 −9) (1 25) (1 24 25) (1 −18 19) (1 23 24) (1 3 5 19) (1 3 −4 −25))
- F_{11} = ((1 2 3) (1 2 3 4) (1 2 3 4 5 6) (1 2 3 4 6 7 8 9) (1 2 3 4 9 10 11 −12) (1 2 3 4 −15 −16 −17) (1 2 3 4 17 18) (1 2 3 −18 20 21) (1 2 3 5 −21 22 23 24 25) (1 2 3 9 20) (1 2 3 5 7 −20) (1 2 3 18 −19) (1 2 3 −4 −16))
- F_{12} = ((1 2 3 4 5 6 7 8 9) (−1 −2 −3 −4 −5 −6 −7 −8 −9) (10 11 12 13 14 15) (−10 −11 −12 −13 −14 −15) (1 2 3 4 5 6 7 −8 9 −10 −11 −12 13 −14 −15))
- F_{13} = ((−1 2 3 4 5 6 7 8 9 10 11) (1 2 3 4 5 6 7 8 9 10 −11) (1 2 3 4 5 6 7 8 9 10 −11) (−1 −2 2 3 4 5 6 7 8 9 10 11))
- F_{14} = ((1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18) (1 2 3 4 5 6 7 8 9 10 11 12 13 −14 15 −16 17 18 19 20) (−1 −2 −3 4 5 6 7 8 9 10 20) (−1 −2 −3 4 −5 −6 −7 8 9 12 15 17 −18 20))
- F_{15} = ((1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19) (2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20) (3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20) (4 −5 −6 −7 −8 −9 −10 −11 −12 −13 −14 −15 −16 −17 −18) (5 −6 7 −8 9 −10 −11 −12 −13 −14 15 19 20) (−1 −2 −3 −4 −5 −6 −7 −8 7 −9 −10 −19 −20))

APPENDIX 2
Experimental Results for the formulas in Appendix 1

The following table summarises the results of our experiments. Throughout this Appendix we utilised $\varepsilon = 0.1$ and $\delta = 0.1$. The last column that we denoted by *Err* gives the relative error in the approximate answer. The last two columns were computed with higher precision and then rounded to the values shown.

Table 1. Overall counts $\varepsilon = \delta = 0.1$)

F	n	m	N	N_s	$\|S\|$	N_F	N_F^*	*Err*
1	7	14	18874	6298	368	124	122.8	+0.01
2	10	7	9437	6889	920	664	671.6	+0.01
3	5	4	5393	3933	18	13	13.1	+0.01
4	7	8	10785	10135	64	60	60.1	+0.00
5	10	10	13481	2720	5120	1024	1033.0	+0.01
6	6	6	8089	6696	45	37	37.2	+0.00
7	6	5	6741	6195	24	22	22.1	+0.00
8	7	7	9437	5426	160	92	92.0	−0.00
9	20	20	26962	7458	1931520	524288	534280.7	+0.02
10	25	11	14829	3928	65011712	16777216	17220716.5	+0.03
11	25	13	17526	6138	12124160	4194304	4246153.9	+0.01
12	15	5	6741	6711	1153	1149	1147.9	−0.00
13	12	4	5393	4043	8	6	6.0	−0.00
14	20	5	6741	5510	709	581	579.5	−0.00
15	25	6	8089	8027	13568	13472	13464.0	−0.00

To compare the timings of the programs Lisp function *ptime* was used. Normally, *ptime* also returns the time spent in collecting garbage. This is discounted in the following figures. When left unspecified, the exact answer was computed by inspection. (For example, one can easily see that F_9 reduces to x_1. Thus the answer becomes $2^{19} = 524288$. It must be remarked that, in general, a counting program may gain considerably on the average by first attempting to simplify.) The column specified as 2^n is the maximum possible count; thus N_F can be at most this much. The column *Perc* gives the ratio of the exact answer to the maximum possible.

APPENDIX 3
Execution time vs. ε and δ for some formulas

The following tables give some statistics to demonstrate the effect of changing ε and δ. We tried here both simple formulas (Tables 3(*a*)–(*c*)) and more difficult ones (Tables 3(*f*)–(*g*)).

Table 3(*a*). Results for F_1

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	14507	4892	124.1	+0.00	458.4
0.1	0.3	11952	4024	123.9	−0.00	377.8
0.1	0.4	10140	3439	124.8	+0.01	316.1
0.1	0.5	8734	2980	125.6	+0.01	270.6
0.2	0.1	4719	1551	120.9	−0.02	148.0
0.3	0.1	2098	717	125.8	+0.01	64.8
0.4	0.1	1180	371	115.7	−0.07	37.0
0.5	0.1	755	258	125.7	+0.01	23.6

Table 3(*b*). Results for F_2

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	7254	5285	670.3	+0.01	271.8
0.1	0.3	5976	4367	672.3	+0.01	229.9
0.1	0.4	5070	3605	654.2	−0.01	193.6
0.1	0.5	4367	3085	649.9	−0.02	163.5
0.2	0.1	2360	1675	653.0	−0.02	91.3
0.3	0.1	1049	733	642.9	−0.03	39.8
0.4	0.1	590	428	667.4	+0.00	21.8
0.5	0.1	378	262	637.7	−0.04	14.2

Table 3(*c*). Results for F_3

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	4145	3048	13.2	+0.01	70.6
0.1	0.3	3415	2492	13.1	+0.01	58.1
0.1	0.4	2897	2126	13.2	+0.01	48.9
0.1	0.5	2496	1803	13.0	+0.00	42.3
0.2	0.1	1349	983	13.1	+0.01	23.0
0.3	0.1	600	434	13.0	+0.00	10.5
0.4	0.1	338	261	13.9	+0.07	5.7
0.5	0.1	216	164	13.7	+0.05	3.8

Table 3(*d*). Results for F_4

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	8290	7777	60.0	+0.00	246.8
0.1	0.3	6830	6396	59.9	−0.00	201.2
0.1	0.4	5794	5432	60.0	+0.00	169.4
0.1	0.5	4991	4676	60.0	−0.00	151.1
0.2	0.1	2697	2556	60.6	+0.01	78.7
0.3	0.1	1199	1131	60.4	+0.01	35.5
0.4	0.1	675	631	59.6	−0.00	19.8
0.5	0.1	432	404	59.8	−0.00	12.9

Table 3(*e*). Results for F_5

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	10362	2166	1070.2	+0.04	292.8
0.1	0.3	8538	1711	1026.0	+0.00	236.0
0.1	0.4	7243	1427	1008.7	−0.01	208.9
0.1	0.5	6349	1293	1061.1	+0.04	182.7
0.2	0.1	3371	667	1013.1	−0.01	93.9
0.3	0.1	2854	354	936.5	−0.08	41.7
0.4	0.1	843	178	1081.1	+0.05	22.7
0.5	0.1	540	118	1118.8	+0.09	15.2

Table 2. Overall timings (seconds)

F	2^n	N_F	*Perc*	COMB	*CRIBLE*	*KL*
1	128	124	0.97	4.7	—	585.3
2	1024	664	0.65	122.9	13.7	358.1
3	32	13	0.41	1.7	0.7	93.4
4	128	60	0.47	6.5	7.6	314.5
5	1024	1024	1.00	21.6	105.9	370.2
6	64	37	0.58	2.3	1.6	199.9
7	64	22	0.34	2.2	0.8	143.8
8	128	92	0.72	3.9	7.8	243.2
9	1048576	524288	0.50	—	—	1768.0
10	33554432	16777216	0.50	—	1125.6	992.7
11	33554432	4194304	0.12	—	—	1357.1
12	32768	1149	0.30	—	1.5	345.6
13	4096	6	0.00	359.6	0.5	203.1
14	1048576	581	0.00	—	1.7	539.9
15	33554432	13472	0.00	—	4.4	1008.0

Table 3(*f*). Results for F_9

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	20724	5739	534886.8	+0.02	1384.8
0.1	0.3	17075	4723	534264.6	+0.02	1132.1
0.1	0.4	14485	4047	539652.1	+0.03	948.2
0.1	0.5	12477	3457	535165.9	+0.02	808.6
0.2	0.1	6741	1894	542693.8	+0.03	434.9
0.3	0.1	2996	849	547350.0	+0.04	196.0
0.4	0.1	1686	457	523549.6	−0.00	108.9
0.5	0.1	1079	288	515549.4	−0.02	69.3

Table 3(*g*). Results for F_{10}

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	11398	2959	16877492.2	+0.01	772.3
0.1	0.3	9391	2331	16136971.6	−0.04	640.9
0.1	0.4	7967	2135	17421865.8	+0.04	512.4
0.1	0.5	6863	1868	17695159.3	+0.05	446.5
0.2	0.1	3708	1021	17901013.5	+0.07	248.7
0.3	0.1	1648	457	18028126.4	+0.07	111.4
0.4	0.1	927	245	17182167.7	+0.02	63.1
0.5	0.1	594	149	16307651.7	−0.03	39.8

Improved Recursion Handling through Integrity Constraints

A new form of database integrity constraint is introduced which describes the structure present in data when that structure takes the form of a graph, tree or list. This type of integrity constraint is of particular benefit in reducing the overhead when detecting termination in the case of recursive queries.

Received September 1987

1. Introduction

Some database integrity constraints control the values which an attribute can adopt (e.g. age < 21) or the existence of values in other relations, e.g.

> if there is an entry for supplier n in the supplier-parts relation, then there must be a corresponding entry for supplier n in the supplier relation.

Other constraints (data dependencies) are concerned with the relationship between the sets of values in one set of attributes with the sets of values in some other, e.g. functional dependencies, multivalued dependencies, etc.

However, there are other relationships which may be present in the data in a database which are not readily captured by existing constraints. For example, the data stored in a relation may represent some form of structure such as a graph, tree or list. Apart from some limited cases, it is not immediately obvious how one can represent information such as this using existing integrity constraints.

If the data in a relation (or set of relations) do satisfy some structural form, one may wish to impart this information to a database management system so that it can check that this form is maintained just as it checks the integrity of data as specified by other integrity constraints. For this purpose a new form of constraint, called a *structure constraint*, is proposed.

Besides the obvious use of such a constraint in maintaining the integrity of data in a database, this type of constraint carried with it another significant advantage. It can be used to produce a more efficient system for handling recursion in databases by reducing the overhead incurred in run-time detection of termination.

The following section describes the basic structural forms used in this constraint. Section 3 details the structure constraints and gives examples while Section 4 discusses their role in improving the efficiency of dealing with recursive relations. Section 5 looks at some theorems relating to these constraints.

2. Structural forms

This section considers some basic structural forms which may exist in a set of data stored in a relation. Before defining these structures, consider briefly some terminology which will be used. Let I be a set of constraints. SAT(I) denotes the set of relations that satisfy each of the constraints in I. Let I_1 and I_2 be constraints or sets of constraints.

$$I_1 \Rightarrow I_2 \text{ if } \mathrm{SAT}(I_1) \subseteq \mathrm{SAT}(I_2)$$

A tuple is a set of mappings from attributes to domain values. A relation is a set of tuples. Let R be a relation over a set, U, of attributes and let $t \in R$. For $W \subseteq U$, we let $t(W)$ denote the restriction of t to the set W. Let R be a relation over a set, U, of attributes, and let A, $B \subseteq U$. R obeys the *functional dependency* $A \rightarrow B$ if for any pair of tuples $t_1, t_2 \in R$, $t_1(A) = t_2(A)$ implies that $t_1(B) = t_2(B)$.

suppose that R is a relation with the relational scheme $R(A, B, C)$ where A and B are attribute sets and C is an abbreviation for the rest of the attributes. Then:

Definition

A *chain* over a pair of attribute sets A and B of $R(A, B, C)$ is a sequence of tuples (a_1, b_1, c_1), $(a_2, b_2, c_2) \ldots (a_n, b_n, c_n)$ where $b_i = a_{i+1}$ for every , $0 < i < n$.

Definition

The *length* of a chain is the number of tuples, n, in the sequence.

2.1 Directed acyclic graph

A typical example of this type of structure occurs in the part of relation, where an object is described in terms of its component parts. An illustration of this is given in Table 1.

Since this graph is intended to be acyclic, no part should be a sub-part of itself. Thus the addition of a tuple such as

part-of (5, 1, 1)

would be invalid as it would violate the condition of acyclicity.

A formal definition of this type of structure is as follows:

Definition

A *directed acyclic graph structure* (*dag structure*) is present in attribute sets A and B of a relation R with relation scheme $R(A, B, C)$, written as *graph* (A, B), if there is no chain of the form $(a_1, b_1, c_1), (a_2, b_2, c_2) \ldots (a_n, b_n, c_n)$ in which $a_1 = b_n$.

2.2 Directed tree structures

A special case of a dag structure is the tree structure. This is identical to the tree structure in QBE [1, 2]. An example of this is shown in Table 2.

Table 1. Example of acyclic directed graph structure

part	part 1-no	part 2-no	no-of-pts
	1	2	1
	1	3	2
	2	4	4
	2	5	2
	3	5	7
	51	55	2
	100	101	2
	101	102	4
	101	103	2
	103	104	2
	100	104	2