Technical Correspondence

COMPUTING IN THE HOME: SHIFTS IN THE TIME ALLOCATION PATTERNS OF HOUSEHOLDS

In Vitalari, Venkatesh, and Gronhaug's (*Commun. ACM 28*, 5(May 1985), 512–522) discussions of Rankings of Computer Use, they present data showing primary computer use as made up of 26 percent for business, 24 percent for word processing, 24 percent hobby, and 22 percent for entertainment. They speculate that if word processing is included, work at home is probably a primary use for 45 percent of the respondents.

Similarly, in their Concluding Remarks the authors state "The study also indicates that business and word processing are dominant uses for home computers. If this indicates a trend, the household of the future may be the site of more task-oriented behaviors. . ."

This seems a rather bold interpretation of the figures—even allowing the 5 percent for finance use to come under business and word processing use; the figures might just as well have been used to express amazement at the extent to which home computers are used for entertainment.

It would be interesting to discover the relative importance of the secondary use and the contributions made by business, word processing, and the other activities and to determine whether or not it is the task-oriented use which contributes most to the apparent changes in family behavior. This is a stimulating article and hopefully the authors will be able to pursue some of the issues raised.

Robert Isbister
Computer Unit
University of Stirling
Scotland

EXTENDED USE OF NULL PRODUCTIONS IN LR(1) PARSER APPLICATIONS

In Finn's article (*Commun. ACM 28*, 9(Sept. 1985), 961–972), it should be noted that the necessity of adding null productions is due to a limitation in the table generator. The need disappears if actions are given the status of tokens and can therefore appear directly in the body of productions, as is done in the Hughes Translation Table Generator (TTG). Additional productions are needed only for actions invoked from multiple points in the grammar, and those could be removed from the completed tables as an optimization.

Inclusion of actions as tokens raises the potential for obscuring conflicts or actually altering the grammar. Such problems occur less often than might be expected, however. They are checked by giving action tokens lower priority than string tokens and by informing TTG which actions consume or test the input string.

Action tokens can even be used to improve performance of the parser by eliminating most reductions. Reductions often impose a significant time and space penalty compared to shifts and combined shift-reduce operations, at least when using TTG tables. Thus, I frequently append the token and do nothing to specific productions of a known LR(1) grammar in order to convert reductions to shift-reduce operations.

In our environment, it is convenient to define rather fine-grained actions which require frequent regeneration of tables. However, one could define tokens corresponding directly to the null productions of the example grammar, so details may be altered without altering the tables.

George K. Tucker
Electro-Optical and Data Systems Group
Strategic Systems Division
Hughes Aircraft Company
El Segundo, CA 90245

A POLYNOMIAL TIME GENERATOR FOR MINIMAL PERFECT HASH FUNCTIONS

The perfect hashing algorithm in Sager's recent article (*Commun. ACM 28*, 5(May 1985), 523–532) suffers from a theoretical defect which can be shown to be inherent to all perfect hashing algorithms, and which I believe is also a practical impediment to perfect hashing. The complexity of the hash function generated by his algorithm grows rapidly with the number of keys to be hashed. That this is necessary in any perfect hash where the keys do not have an ordinal encoded into them can be shown as follows.

If N keys are to be hashed into M addresses, every possible set of N keys must have its own hashing function. Consider a computer with p operations

whose maximum word size for arithmetic is w. The number of executions of operations, t, is given by

$$(p2^w)^t = N!$$

which, solved for t, yields

$$t = O(N \log N).$$

Practical search algorithms which run in $O(\log N)$ time are well known, so that perfect hashing emerges as clearly suboptimal in large cases. If practical considerations are taken into account, along with the availability of good alternative schemes for small sets of keys, I suspect there are no cases where perfect hashing proves worthwhile.

We can weaken the insistence that the keys convey no information about an ordinality to a large degree without making perfect hashing more practical. Consider allowing the M addresses to have their contents in whatever order is convenient for the hash, and to be predictably distributed—fixed length records beginning at a known location, for example. If we further allow a predictable distribution in the keys, such as in the first N integers, clearly a perfect hash whose function runs in constant time is easily found. However, given a set of keys with a largely random distribution, such as words in a lexicon, an ordinality cannot be deduced. Cases between the two extremes of predictable and random distribution fall into the class of interpolation searches, for which practical results have been unexciting so far. And, of course, other theoretically optimal $O(\log N)$ search routines are well known and widely used.

It may seem paradoxical that no amount of time devoted to preprocessing a fixed set of keys in this way results in any saving. The problem lies in encoding their distribution into a hash function, which must take into account all possible permutations of keys, while the actual search need only look at those keys actually in the data. Recently some articles on perfect hashing have summaries in the form of "morals" drawn from the results presented. I would offer this: Finding something can be a lot faster than figuring out where it is.

Jeffrey Kegler
Lake Anne Software
1600 Chimney House Road
Reston, VA 22090

AUTHORS' RESPONSE

I am not quite sure that I understand what Mr. Kegler is trying to say but he appears to have missed the point of my article. His comment that "every possible set of N keys must have its own hashing function" appears wide of the mark. It is not *every set* but *some particular set* that interests us. I would like to refer Mr. Kegler to the technical report [2] in which a minimal perfect hash function, MPHF, for a particular set—the 256 most common words in the English language, is given.

The *generating* function has a worst-case time complexity exponential in N but an expected time complexity no worse than proportional to N^6. The *generated* function has constant time complexity provided we hold the parameters h_0, h_1, and h_2 constant and that we consider that the operations +, mod, and [ ] are performed in constant time. This should be clear from the form of the MPHF generated, $(h_0(w) + g[h_1(w)] + g[h_2(w)]) \bmod N$.

There appears to be a practical limit to the size of the sets for which the mincycle algorithm can successfully generate MPHF's. It is uncertain exactly where this limit lies. At the time of publication of my article, I felt that this limit was probably around 512, but a recent study by Hou[1] sheds some further light on this question. The nature and existence of this practical limit is, I believe, still an open question.

In any case for medium size static sets, say between 32 and 512, I know of no more practical search method than MPHF's as generated by the mincycle algorithm. Should Mr. Kegler or anyone else have information to the contrary, I would be very glad to hear about it.

Thomas J. Sager
Department of Computer Science
University of Missouri—Rolla
Rolla, MO 65401

REFERENCES

1. Hou, P.-P., and Sager, T.J. A Monte Carlo analysis of the mincycle algorithm for generating minimal perfect hash functions. Tech. Rep. CSc-85-3, Univ. of Missouri, Rolla, 1985.
2. Sager, T.J. A new method for generating minimal perfect hash functions. Tech. Rep. CSc-84-15, Univ. of Missouri, Rolla, 1984.

THE P² ALGORITHM FOR DYNAMIC CALCULATION OF QUANTILES AND HISTOGRAMS WITHOUT STORING OBSERVATIONS

Jain and Chlamtac's P² algorithms for the calculation of quantiles and complete histograms (*Commun. ACM 28*, 10 (Oct. 1985), 1076–1085) are valuable tools for simulation studies or performance measurement. However, applying the histogram algorithm to data about which little or nothing is known can yield extremely misleading results. The authors warn that the percentile algorithm should not be used to estimate quantiles which are close to discontinuities

TABLE I. Estimates of Quantiles From 650 Bernoulli Trials. The estimates of the 0.5, 0.6, and 0.7 quantiles are far from numbers which occur in the data.

p value	Estimated quantile
0.0	0.000
0.1	4.921E-13
0.2	2.994E-9
0.3	1.803E-6
0.4	9.363E-4
0.5	0.1450
0.6	0.1937
0.7	0.4815
0.8	1.000
0.9	1.000
1.0	1.000

since such estimates will naturally be extremely unstable; I wish to point out that a similar caveat applies to the histogram algorithm.

If one applies the histogram algorithm to data from a discrete distribution, several quantiles may be estimated incorrectly. Unlike the method of storing and sorting the observations, which at worst yields unstable estimates, the P^2 algorithms tend to interpolate values that never appear in the data. The problem is worst when the number of possible data values is small relative to the number of cells in the histogram. For example, a 10-cell histogram produced from 650 Bernoulli trials (0 or 1, each with probability 0.5) sometimes produced as many as three distinct extraneous quantiles (Table I).

To safely discover the most about data whose distribution is unknown, one could start by counting the frequencies of individual values. If the number of values exceeds the number of cells in the desired histogram, the data could be fed into the P^2 algorithm, which could then be used for the rest of the analysis.

David Gladstein
1119 Governor Winthrop Road
Somerville, MA 02145

AUTHORS' RESPONSE

We agree. Our warning about discrete distributions applies to both percentiles and histogram algorithms. Thanks for pointing it out explicitly.

Raj Jain
Digital Equipment Corporation
550 King Street (LKG1-2/A19)
Littleton, MA 01460-1289

STRUCTURED TOOLS AND CONDITIONAL LOGIC: AN EMPIRICAL INVESTIGATION

I read Vessey and Weber's article (*Commun. ACM 29*, 1 (Jan. 1986), 48–57) with interest. When I had finished reading the article, I found myself with a nagging problem: how much where the results influenced by the choice of COBOL as the language in which the final code was written?

This language choice was reflected in the form of the structured English as well. Suppose that they had allowed the use of some form of "case" statement, would this have affected their results? I suspect it would have, because my experience has been that multiway selection is particularly difficult in languages that do not have a "case" statement.

J. P. E. Hodgson
Department of Mathematics and Computer Science
Saint Joseph's University
Philadelphia, PA 19131

AUTHOR'S RESPONSE

We examined the performance of subjects in designing and coding logic for nested conditionals. Nested conditionals can be coded using the "case" construct only by using Boolean operators, that is, even a language that has a "case" construct requires the use of Boolean operators. As noted in the paper, Green, Sime, and Fitter [1] report difficulties in using Boolean operators. We used the nested conditional construct because of the difficulties reported by those authors.

It was essential to satisfy experimental conditions that subjects reproduce a standard format in their responses. Hence, our subjects were trained in using the nested conditional construct and all participants used that format in the experiment.

Iris Vessey
Department of Commerce
University of Queensland
St. Lucia, Queensland, Australia 4067

Visiting at
Graduate School of Business Administration
University of Minnesota
Minneapolis, MN 55455

REFERENCE

1. Green, T.R.G., Sime, M.E., and Fitter, M.J. The problems the programmer faces. *Ergonomics 23*, 9 (1980), 893–907.