Appl. Math. Lett. Vol. 1, No. 1, pp. 25-28, 1988
Printed in the U.S.A.
0893-9659/88 $3.00 + 0.00

Finding and Applying Perfect Hash Functions

Nick Cercone
School of Computing Science
Simon Fraser University
Burnaby, British Columbia, Canada V5A 1S6

1. Introduction and Background

Perfect hash functions, a deterministic refinement of the key-to-address transformation techniques, provide single probe retrieval of keys from a static table. Given a set of N keys and a hash table of size r = N, a perfect hash function maps the keys into the hash table with no collisions since the function locates each key at a unique table address. The loading factor [LF] of a hash table is the ratio of the number of keys to the table size N/r. A *minimal* perfect hash function maps N keys into N contiguous locations for a LF of one.

Perfect hash functions are difficult to find, even when almost minimal solutions are accepted. Knuth [1] estimates that only one in 10 million functions is a perfect hash function for mapping the 31 most frequently used English words into 41 addresses. Cichelli [2] devised an algorithm for computing machine independent, minimal perfect hash functions of the form:

hash value = hash key length + associated value of the key's first letter + associated value of the key's last letter

Cichelli's hash function is machine independent because the character code used by a particular machine never enters into the hash calculation. The algorithm incorporates a two-stage ordering procedure for keys which effectively reduces the the size of the search for associated values but excessive computation is still required to find hash functions for sets of more than 40 keys. Cichelli's method is also limited since two keys with the same first and last letters and the same length are not permitted.

Our objective of was to develop faster and more general algorithms for finding perfect hash functions of the general form of Cichelli. Three procedures for specifying the hash identifier were implemented, including: (1) a previously defined hash identifier (a la Cichelli); (2) a hash identifier determined by an automatic procedure; and (3) a hash identifier specified by the user interactively. The algorithms were programmed in APL and Pascal; the performance and results of each algorithm were evaluated, see Cercone et al. [3] for a complete description of these algorithms. For this note we only consider the third of these algorithms, *algorithm CBK*.

Cichelli's algorithm uses key length and the first and last letters (without regard to letter position) as the hash identifier. The number of keys which can be distinguished is restricted to P*CH(A',2) where P is the maximum key length, CH is the familar *choose* function, and A' is the cardinality of the alphabet. Integer assignment values are found using a simple backtracking process. Cichelli proposes no method of choosing a value of m, the size of the domain of associated letter values. This is an important parameter of the problem since m is the branching factor of the backtrack search tree.

Cichelli first arranges the keys in decreasing order of the sum of frequencies of occurrence of the first and last letters. This sorting implicitly arranges the letters so that letters which occur most frequently are assigned integer values first. During the second step of the ordering any key whose hash value has already been determined, because its first and last letters have both occurred in keys previous to the current one is placed next in the list. This double ordering strategy arranges the static set of keys in such a way that hash value collisions will occur and be resolved as early as possible during the backtracking process. When collisions occur at the root of the search tree, pruning can eliminate large subtrees and greatly reduce the cost of finding an acceptable assignment of integers.

In the interactive system [the CBK algorithm] the user specifies a set of letter positions and whether or not to include the key length in the hash identifier. The program then tests the user's selection for key discrimination, inviting the user to try again if any two keys cannot be distinguished. The system takes into account the position of occurrence of letters and therefore has the greatest possible discriminatory power of the three algorithms we developed. There is no set of distinct lexical keys which cannot be distinguished by this system. No upper bound is placed on the size of associated letter values.

2. Performance Comparison: The CBK Algorithm vs. Cichelli's Algorithm

Cichelli's algorithm was implemented as a Pascal program; the interactive system was written in APL. All programs were run on an IBM 4341 computer under the Michigan Terminal System [MTS] time-sharing operating environment. Both algorithms were tested with some representative keysets, and the execution time, maximum number of keys which can be processed, and the loading factor of the resulting hash tables were compared.

Analytic comparison of the relative performance of backtracking algorithms is difficult, Knuth [4]. The number of basic operations of the algorithm and the memory requirements should be considered in algorithm expense. Krause [5] estimates the number of times basic operations are performed by these algorithms.

Execution time for Cichelli's algorithm rose rapidly with increasing keyset size; no results were returned within 2 hours for keysets larger than 64 (Table 1). The CBK algorithm found minimal perfect hash functions for keysets of N ≤ 64 and returned almost minimal solutions for N ≤ 500 (Table 1). [An alternative algorithm we developed earlier returned perfect minimal hash functions for keysets of up to 200 keys but performed poorly beyond that point].

Hash Key Set	Cichelli		CBK	
31 Most Frequent English Words	T=290	LF=0.97	T=1763	LF=1.0
33 Basic Keywords	N/A		T=0.669	LF=1.0
34 ASCII Control Codes	T=1833	LF=1.0	T=1993	LF=1.0
36 Pascal Reserved Words	T=579	LF=1.0	T=2609	LF=1.0
40 Pascal Predefined IDs	T=360641	LF=1.0	T=3060	LF=1.0
42 Algol-W Reserved Words	N/A		T=0.616	LF=1.0
64 Most Frequent English Words	T>>1 hour		T=2933	LF=1.0
76 Pascal Reserved + Predefined Ids	no results		T=3414	LF=0.98
100 Most FrequentEnglish Words	no results		T=5190	LF=0.96
200 Most Frequent English Words	no results		T=8986	LF=0.70
500 Most Frequent English Words	no results		T=33505	LF=0.61

Table 1. Comparison of time [T] (in milliseconds) and loading factor [LF] on some representative key sets.

To summarise, the two major problems with Cichelli's algorithm are: (i) the loading factors of the solutions produced degenerate quickly for keysets of more than 40 keys; and (ii) the mechanism used for distinguishing keys is not adequate for many problem sets. Our refinements led to the development of the substantially different CBK algorithm and addressed these problems directly with moderate success with respect to problem (i) and total success with problem (ii). The CBK Algorithm outperforms Cichelli's (and all others reported to date, see [3] for a synopsis of other approaches) and shows promise for further development. This algorithm does require additional storage to maintain separate associated value tables for each letter position selected.

3. An Example Interactive Terminal Session

The record of a terminal session illustrating the CBK algorithm finding a minimal perfect hash function for the 76 Pascal identifiers is illustrated below. The listing is annotated with comments enclosed in { } brackets.

```
# RUN *APL PAR=D
# EXECUTION BEGINS 00:38:45
> )LOAD PERFECT 300000
> saved 12:59:00 06/26/85
                    MINIMAL, BACKTRACKING, 3 LETTER POSITIONS - 76 PASCAL KEYWORDS & IDENTIFIERS
> HASH                                                 { preorders data for associated value calculation}
>  WORDS TO BE HASHED: ab                              { ab is a variable containing the Pascal IDs }
>  LETTERS TO BE USED: 1 2 4                           { for assignment of associated values }
>  IS BLANK TO BE A CHARACTER (Y/N): N                 { or use last letter of word if appropriate }
>  IS LENGTH TO BE PART OF FUNCTION (Y/N): Y
>  ORDER BY PRODUCT OR MINIMUM (P/M): P                { product of letter frequencies or like Cichelli's }
>  CPU SECONDS USED IN HASH IS 1.707
>  THE DATA IN CORRECTED PREORDER FORM:
> round eoln downto do record reset repeat read readln rewrite real for to text set trunc true pred page case ord or write writeln arctan
> put false cos const not procedure function succ file sin nil then chr char while goto get end and in integer mod eof label in boolean sqr
> sqrt pack packed maxint begin array until unpack output dispose of it abs div exp new odd var else type with input program otherwise
> BASH                                                 { invoking the second-ordering part }
>  LOADING FACTOR 0.5 TO 1 RANGE: 1
>  NUMBER OF ALLOWABLE BACKTRACKS: 10
>  BASHING STARTED AT 1985 7 12 14 12 52 470           TIME DURATION WAS 0 0 0 0 0 11 914
>  CPU SECONDS USED IN BASH IS 1.551                   NUMBER OF TIMES THROUGH BASH MAIN LOOP IS 78
>  TERMINATION AFTER BACKTRACK 3          LETTERS USED 1 2 4          OFFSET USED 0
>  LETTER VALUES

>  'A'  3 13 35   'E'  0  1  2   'I'  8 21 16   'N' 12 31  0   'R'  0 11  3   'V' 50  0 37
>  'B' 35 29  0   'F'  8 12 44   'K'  0  0 36   'O'  6  0  0   'S'  3  0 25   'W'  1  0 51
>  'C'  2  0 13   'G' 32  0 57   'L' 29 64  8   'P'  0  0 35   'T'  1 60  9   'X'  0 27  0
>  'D'  0 54  5   'H'  0 29 46   'M' 38  0  0   'Q'  0 43  0   'U'  4 11 30   'Y'  0 64  0
>  HASH TABLE
>   2 DO       3 TO        4 EOLN     5 ROUND    6 DOWNTO   7 RECORD   8 RESET    9 REPEAT
>  10 READ    11 REWRITE  12 READLN  13 REAL    14 FOR     15 TEXT    16 SET     17 TRUNC
>  18 TRUE    19 PAGE     20 PRED    21 CASE    22 OR      23 PUT     24 NOT     25 ORD
```

```
>  26 WRITE     27 SIN        28 WRITELN   29 ARCTAN      30 COS      31 SUCC       32 CONST      33 PROCEDURE
>  34 THEN      35 FILE       36 GOTO      37 CHR         38 CHAR     39 END        40 FUNCTION   41 IN
>  42 AND       43 WHILE      44 NIL       45 GET         46 MOD      47 EOF        48 INTEGER    49 LABEL
>  50 BOOLEAN   51 FALSE      52 SQR       53 PACK        54 ARRAY    55 PACKED     56 UNTIL      57 BEGIN
>  58 OUTPUT    59 SQRT       60 ABS       61 DIV         62 LN       63 DISPOSE    64 OF         65 EXP
>  66 IF        67 NEW        68 ODD       69 VAR         70 ELSE     71 TYPE       72 WITH       73 MAXINT
>  74 INPUT     75 PROGRAM    76 UNPACK    77 OTHERWISE
```

4. Applications of Perfect Hash Functions - Natural Language Lexicon Design

Retrieval methods usually assume equal likelihood of retrieval for each data item (Knuth [1]). Cichelli [2] pointed out the utility of perfect hash functions for use in compilers. It is well documented in the literature of lexicography (Carroll et al. [6]) that this is not the case for the English language (or, presumably, for any natural language). We propose to make use of information about the frequency of occurrence of English words and a judicious mix of common search and hash encoding techniques to provide an efficient organisational strategy for a natural language lexicon.

One approach which utilises the CBK Algorithm is illustrated in below. Satisfactory experimental results have shown that 500 words can be placed in a non- colliding hash table in under 20 seconds. Nevertheless the LF is only about .67 which we feel is unsatisfactory; increasing the LF results in a substantial increase in computation. When more than one hash function is used, an offset can be manipulated to start the next group of 500 words in the sparse part of the table occuppied by the previous group of 500 words, typically resulting in a loss of about only 10% of storage space. In this example the lexicon is divided into group of 500 lexical items (more or less) and the CBK Algorithm is applied successively, manipulating the OFFSET to interleave the 500-word pieces to effectively increase the LF to an acceptable level. Our experimental results fitted the first 500 word chunk into a table of size 750; the first offset was set to 550, the index where the application of algorithm CBK to the second 500 work lexical chunk began to place items. The first 1000 words thus fit into a space of 1340 spaces. We continue this process until we have the dictionary we desire or we exhaust our computer memory. This technique effectively makes use of unused spaces from previous applications of algorithm CBK. This technique is illustrated below using 100 word chunks (because of space limitations) which are non-minimally hashed.

Interleaved Lexicon - 500 Most Frequently Used English Words (MFEW).

```
> HASH                                              { preorders data for associated value calculation }
>  WORDS TO BE HASHED: c1                           { c1 is a variable containing the 1st 100 MFEW }
>  LETTERS TO BE USED: 1 2 L                        { for assignment of associated values }
>  IS BLANK TO BE A CHARACTER (Y/N): N              { or use last letter of word if appropriate }
>  IS LENGTH TO BE PART OF FUNCTION (Y/N): Y
>    THAN CONFLICTS WITH THEN                       THERE CONFLICTS WITH THESE
>  WOULD YOU LIKE TO TRY A DIFFERENT ROUTE: Y
>  LETTERS TO BE USED: 1 2 3 4                      { for assignment of associated values }
>  IS BLANK TO BE A CHARACTER (Y/N): N              { or use last letter of word if appropriate }
>  IS LENGTH TO BE PART OF FUNCTION (Y/N): Y
>  ORDER BY PRODUCT OR MINIMUM (P/M): P             { product of letter frequencies or like Cichelli's }
>  CPU SECONDS USED IN HASH IS 2.518
>  THE DATA IN CORRECTED PREORDER FORM:
> the then these when she we they there me he were her more be been them than that what war was has some men man this their his time
> him made say may for first shall would come can could must our one on i in an any or are well will only but out into from who to so not no its
> it at should before is as your you said had any my by how now of if us a over upon with little do up all two have like  such very about
> every great other which people
> BIND 0                                            { invoking the second ordering part - nonbacktracking }
>   BINDING STARTED AT  1985 7 30 14 34 36 450      TIME DURATION WAS 0 0 0 0 0 2 438
>   CPU SECONDS USED IN BASH IS  1.377              NUMBER OF TIMES THROUGH BASH MAIN LOOP IS  75
>{ at this stage the first of the five tables given below appeared, subsequent invocations of the "HASH" and "BIND n" functions resulted in
>  the other 4 tables given successively. Note that three different hash functions were utilised to construct this single table of 500 MFEW }
>LETTERS USED 1 2 3 4    LETTERS USED 1 2 3 4    LETTERS USED 1 2 3 4    LETTERS USED 1 3 L    LETTERS USED 1 2 4 L
>OFFSET IS 0             OFFSET IS 95            OFFSET IS 195           OFFSET IS 295         OFFSET IS 390
>LETTER VALUES           LETTER VALUES           LETTER VALUES           LETTER VALUES         LETTER VALUES
>'A'  4  6 14 54         'A' 20  0  4 26         'A' 26  1 20  0         'A' 16 48 56          'A' 22 20 13 47
>'B'  9 28  0  0         'B'  7  0  6  0         'B'  4  0 40  0         'B' 15 83  0          'B' 16  0  0  0
         :                       :                       :                      :                     :
>'R'  0 38  3  5         'P'  0  0  0  0         'R' 68 25 12 36         'Q' 55  0  0          'Q' 14  0  0  0
         :                       :                       :                      :                     :
>  HASH TABLE                {after c1, the 1st-100 MFEW have been analysed}
>    3 THE        4 THEN        5 THESE     6 WHEN       7 SHE        8 WE         9 THEY       10 THERE
>   11 ME        12 HE         13 WERE       . . .      99 OTHER    100 WHICH     132 FROM     133 PEOPLE
>LOADING FACTOR IS: 100/133 = .752
>
```

```
> HASH TABLE                   {after c2, the 2nd-100 MFEW have been analysed and added}
> ...        99 OTHER          100 WHICH       101 MONEY       102 POWER       ...            131 YEARS      132 FROM
> 133 PEOPLE 134 FOUND         135 MAKE        ...             199 THOUGHT     203 AWAY       209 SINCE      227 DID
>LOADING FACTOR IS: 200/227 = .881
>
> HASH TABLE                   {after c3, the 3rd-100 MFEW have been analysed and added}
> ...        199 THOUGHT       200 SET         201 LET         202 CENT        ...            227 DID        228 SEEN
> 229 MEANS  230 MORNING       ...             299 NECESSARY   300 THEMSELVES                 317 KNOWN      323 OFF
>LOADING FACTOR IS: 300/323 = .929
>
> HASH TABLE                   {after c4, the 4th-100 MFEW have been analysed and added}
> ...        299 NECESSARY     300 THEMSELVES  301 SEEMS       302 SEEMED      ...            323 OFF        324 CAUSE
> 325 FLOUR  ...               395 SUBJECT     396 BEGINNING   397 YESTERDAY   405 VIEW       407 ASK        408 KEEP
> 418 DIFFERENT 420 REAL
>LOADING FACTOR IS: 400/420 = .952
>
> HASH TABLE                   {after c5, the 5th-100 MFEW have been analysed and added}
> ...        395 SUBJECT       396 BEGINNING ...               421 COMES       ...            498 IMPORTANT  504 UNLESS
> 519 ELECTRIC 526 GUNS        552 KNOWLEDGE
>LOADING FACTOR IS: 500/552 = .906
> TOTAL TIME IS: 5 HASHS - 11.913 SECONDS      5 BINDS - 6.789 SECONDS      TOTAL - 18.702 SECONDS
```

Since large lexicons typically require secondary storage media a major concern is to minimise retrievals from secondary storage. The CBK algorithm can include the 732 most frequently used English words, which make up 75% of running text, in a single almost-minimal hash table, giving one-probe retrieval in 75% of the cases. A second hash function could map the remaining approximately 50,000 words into 50 subsets of about 1000 words each. This second hash function could be based on the ordinal positions of letters in the alphabet rather than on the machine character code in order to preserve machine independence. The 50 subsets of 1000 words each could be stored separately in secondary memory. For each subset an almost-minimal perfect hash function could be computed, storing the associated values in the same secondary memory location as the lexical information itself. If the key we are searching for is not in the table of most-frequent words, then a hash would be performed to select the proper second-level table from a secondary storage medium; this table would then be searched using its own perfect hash function. This organisation would allow us to retrieve any key with three hash calculations and one probe of secondary memory,

References

[1] Knuth, D. (1973) **The Art of Computer Programming 3: Sorting and Searching**, Addison Wesley, Reading, Mass.

[2] Cichelli, R. (1980) Minimal Perfect Hash Functions Made Simple, **CACM 23**, 17-19.

[3] Cercone, N., Krause, M., and Boates, J. (1982) Minimal and Almost Minimal Perfect Hash Functions Search, **Computers and Mathematics 9(1)**, 215-232.

[4] Knuth, D. (1975) Estimating the Efficiency of Backtrack Programs, **Mathematics of Computation 29**, 121-136.

[5] Krause, M. (1982) Perfect Hash Function Search, M.Sc. Thesis, Computing Science Department, Simon Fraser University, Burnaby, British Columbia.

[6] Carroll, J., Davies, P., and Richman, B. (1971) **The American Heritage Word Frequency Book**, American Heritage Publishing Company, Inc., New York.