A NEW HASHING METHOD WITH APPLICATION FOR GAME PLAYING

Technical Report #88

Albert L. Zobrist

April 1970

Abstract

A general method of hash coding is described together with an application for programs which play board games such as checkers, chess, and GO. An auxiliary method which detects retrieval errors is proposed. The error rate can be precisely controlled depending upon how much space in the hash table is devoted to the auxiliary method.

Introduction

When a computer program stores an item in a large table, subsequent reference or retrieval of that item may necessitate a search of the table. This is so unless a table address for the item can be calculated in a systematic fashion from the item itself. A function which converts items into addresses is called a hashing algorithm, and the resulting table a hash table. The application of these functions is the subject of a small and specialized literature, but will soon be covered by Knuth[1].

A good hashing function has two properties: randomness in the distribution of addresses and speed of operation. Two items may produce the same hash address. We shall call such an occurrence a clash. A clash cannot be detected unless the items or derived quantities are stored in the table. The stored quantities which may be used to resolve clashes will be called keys.

If a game playing program uses lookahead and evaluation followed by minimaxing to find its move, and if the alternate move sequences give frequent duplication of board configurations, then it might be useful to store the backed up values of the board configurations encountered. A hash table would enable fast retrieval of these values. The Greenblatt chess program[2], one of the most successful game players, uses a 32K hash table to store such information. A mild acquaintance with game playing programs is assumed for what follows.

Suppose that a board configuration is reached at depth 5 which has already been fully evaluated at depth 3. Then the value from the node at depth 3 may be given to the node at depth 5 with no further lookahead or evaluation necessary. Since most lookahead algorithms require search to a fixed or calculated depth under the assumption that backed up values are more reliable, the values retrieved in this manner should come from the same depth or closer to the top of the tree. If α–β minimax is used (see the Greenblatt reference), many partially analyzed nodes are left behind as the tree is traversed. For such node, it is known that their value is greater or less than some number. This information may yield α–β cutoffs at a node, or at least remove a node from further consideration.

A game player which uses this procedure along with hash table storage needs to store the level and value of completely evaluated nodes, or the level and value range of partially evaluated nodes. Although this could be done with no errors by storing the board configurations as a key for each node, we might tolerate some errors if, as a result, operations could be made more efficient. There are two types of errors worth considering. Type 1 errors result if keys are not stored. If two different board configurations yield the same hash address, the error is not detected and the wrong value may be given to a node of the lookahead. Type 1 errors also occur if keys are sorted but they are inadequate to resolve all clashes. If type 1 errors are controlled, the type 2 errors may occur. Entries in the hash table may be overwritten by subsequent entries. Some nodes may have to be evaluated again, but no wrong values are inserted into the lookahead. We shall give a general description of the hashin method before considering its application to game playing.

Description of the method

The exclusive or (XOR) operation will be used. Assume the following properities for the XOR of a random sequence (r_1) of n-bit integers:

P1) r_i XOR $(r_j$ XOR $r_k) = (r_i$ XOR $r_j)$ XOR r_k

P2) r_i XOR $r_j = r_j$ XOR r_i

P3) r_i XOR $r_i = 0$

P4) if $s_i = r_i$ XOR r_2 XOR $\ldots$ XOR r_i then $\{s_i\}$ is a random sequence.

P5) $\{s_i\}$ is uniformely distributed.

Suppose we desire hash codes for the subsets of a finite set S. An easy method would be to assign n-bit integers from a random sequence to the elements of S and let the hash code of a subset be the XOR of the integers assigned to the elements of that subset[1]. By P1 and P2 this code is unique, and by P4 and P5 it is randomly and uniformly distributed. If any element is added to or deleted from a subset, the code changes by the integer which corresponds to that element.

[1]Example: `https://gist.github.com/miku/d53e2facdefec281f4de1cb2c219bc6f`

Other applications result if the objects to be hashed can be ut into a 1-1 relation woth the subsets of a finite set in a natural way. Consider the problem of hashing English words of ten characters or less. The set S could be the 260 elements of the form:

$$\begin{aligned}
s_{1,1} &= \text{letter A in position 1}\\
s_{1,2} &= \text{letter A in position 2}\\
&\vdots\\
s_{1,10} &= \text{letter A in position 10}\\
s_{2,1} &= \text{letter B in position 1}\\
&\vdots
\end{aligned}$$

Thus every word will correspond to a unique subset of S, since two different words must differ by some letter at some position. If 260 integers h_{ij} from a random sequence are assigned to the elements of S, then a hash code is obtained. For example:

$$\text{hash(CAT)} = h_{3,1} \quad \text{XOR} \quad h_{1,2} \quad \text{XOR} \quad h_{20,3}$$

Note that not all subsets of S correspond to legal English words or even to letter strings, e.g. $\{s_{1,1}, s_{2,1}\}$.

The clustering of errors

Although the hash codes are uniformly distributed, the idiosyncrasies of a particular problem may produce an unusual number of clashes. Continuing our example for English words, if the following clash occured:

$$\text{hash(CAT)} = \text{hash(DOG)}$$

then

$$\begin{aligned}
&\text{hash(CAT)} = \text{hash(DOG)}\\
&\text{hash(CATNIP)} = \text{hash(DOGNIP)}\\
&\text{etc.}
\end{aligned}$$

Similar problems might occur for any application of XOR hash coding. The basic method depends upon the uniform distributuion of codes for the subsets of a finite set S. If two different subsets S_1 and S_2 have the same hash code, then for any subset S_3 disjoint from $S_1 \cup S_2$ we have

$$\text{hash}(S_1 \cup S_2) = \text{hash}(S_2 \cup S_3)$$

and for any subset $S_4 \subset S_1 \cap S_2$ we have

$$\text{hash}(S_1 - S_4) = \text{hash}(S_2 - S_4)$$

Hopefully, XOR hash coding will show advantages for some application which outweights its disadvantages.

Application to game playing

Consider the class of games in which men are places or removed from a board, but are not moved: Go, tic-tac-toe, GO-MOKU, quobic, hex, etc. Suppose that there are m distinguishable types of men and n locations on the board.

There are at most $m \times n$ possibilities for placing a type of man at a board loaction, thus every game configuration corresponds to a unique subset of there $m \times n$ possibilities. If an $m \times n$ array of integers is taken from a random sequence, then one integer can be assigned to each placement possibility, and the ash code for a board configuration would be the XOR of all the men on the board.

If the status of teh game is updated with each move, then only one XOR is needed for the placement of aman. Since XOR is usually a fast machine operation, this is probably the fastet hashing method available. By properties P1-P3 the removal or capture of a man can be accomplished with the same XOR operation used for its placement. The integer used for placement will cancel the integer used for removal. During lookahead, the hash codes can be stored recursively, this only ne XOR operation will be needed for each node visited.

For games of movement such as checkers and chess, a move can be treated as the removeal of a man at one position and his placement at another. Promotions can be treated as the removal of one type of man the replacement of another. Chess and checker moves will typically incove two XOR operations, but such moves as castling, captures, and multiple captures may require three or more.

The storage requirement for the array of integers is note excessive. Checkers uses 4 types of men and 32 board locations, chess uses 12 types of men and 64 board locations, and GO uses 2 types of men and 361 board locations. Since only distunguishable types of men and board locations are noted, this merhod is guaranteed to recognize all equivalent board configurations as they occur.

Our method has assumed that the moves keading to a board configuration have no effect upon its evaluation. Bug some states of a game may depend upon the last move or previous moves. In chess, a pawn is en passant only if it has just moved, and a king may castle with a root only if both pieces have not previously moved. In GO, certain captures are forbidden for one turn only. If such conditions are used in evaluation, then provisions can be made to incorporate them into the hash code. For example, four integers may be reserved for the four castling opportunities. When a castling opportunity is remove, its integer is XOR ed.

If type 1 errors are to be reduced or eliminated, the it is necessary to store a key which is capable of detecting clashes. However, no description of the board configuration is produced by our method, asige from the hash code itself. A simple solution is to produce a second XOR hash code usind an independent random sequence, and store this second code as a key. If two different board configurations produce the same primary hash code, the it is unlikely that the secondary hash codes will be equal as well. The secondary hash code need not have the same number of bits as the first, which is determinded by the length of the hash table. If the levels and values stored in the ash table occupy part of a word, the the remaineder of the word can be used for the secondary hash code. It is fairly obvious that the greater the number of bits in the secondary hash code, the less chance there is of type 1 errors.

Estimation and control of errors

If two different board configurations produce the same primary and secondary hash code, the a type 1 error may occur. A wrong value would be inserted into the move tree search, with some chance of the error backing up to the top of the tree. Because of the nature of XOR hashing, successors of the two board configurations will produce an unusual number of type 1 errors as well, making it difficult to estimate the true probability that na error will affect the move decision. We shall assume that every type 1 error is a fatal error and estimate their probability for different key sizes.

Suppose that the hash table has 2^n positions and that the keys have $n+m$ bits each. Suppose that during the move tree search, 2^n values were retrieved from the hash table with a probability

$$\frac{2^{n+m}-1}{2^{n+m}}$$

that each was correct. For an apporoximation we assume that these probabilities are independent (which is conservative in view of the clustering of errors) yielding the following probability that no errors occur:

$$\text{p(no errors)} = (1 - \frac{1}{2^{n+m}})^{2^n} = (\frac{1}{e})^{2^{-m}}$$

The approximation can be used to yield the following probabilities:

m = number of extra bits	probability of no error
0	0.36778
1	0.60653
2	0.77879
3	0.88251
4	0.93941
5	0.96924
6	0.98449
7	0.99222
8	0.99611
9	0.99805
10	0.99901

If a game player uses a 32K hash table then 20 bit keys would yield a 3% error rate. 25 bit keys would yield .1% error rate. Many computers have a word size capable of storing such key as well as the value and level, so a 32K hash table would fit in 32K words. This control of key size and of the frequency of errors is a major advantage of the method outlined in this report.

Comparison with another method

It is interesting to consider a current hashing method which could be applied to game playing programs. First obtain an integer which descibes the board configuration. Then, divide the integer by the hash table size and use the remainder as a hash address and the quotient as a key. However, the integer which descibes the board configuration may occupy several (or many) computer words, thus, the divide will be complecated and slow. The quotient may also occupy several words, thus most of the hash table would be occupied by these keys if type 1 errors were to be avoided.

Conclusion

A new hashing method has been described, but it is of dubious value for general applications. Its specific application to game playing (where errors may be tolerated) shows the following advantages:

1) It is easy to apply. For example, its implementation for checkers, chess, and GO are roughly the same.

2) It is fast.

3) It enables precise control of the serious errors which result from the retrieval of wrong information from the hash table.

4) It allows choice of the size of keys used to control the errors in 3).

This method is currently being implemented for two game playing programs (checkers and chess) at the University of Wisconsin. However, I would be interested in hearing of any other applications or potential applications it may have.

Acknowledgements

This work was partially supported by NSF grant GJ583 and NIH grant MH12266.

References

[1] Knuth, D. The Art of Computer Programming, vol. 3 (to appear)

[2] Greenblatt, R., D. Eastlake, and S. Crocker, The Greenblatt chess program, Proceedings Fall Joint Computer Conference, Thompson, Washington D.C., vol. 31, Nov. 1967.