Dr. Carlo Pescio

Minimal Perfect Hashing

Published in Dr. Dobb's Journal, No. 249, July 1996.

Hashing is a powerful and simple technique for accessing information: each key is mathematically converted into a number, which is then used as an index into a table. In typical applications, it's common for two different keys to map to the same index requiring some strategy for collision resolution. However, many applications use a *static* set of records; for instance, the table of keywords for a compiler, the contents of a dictionary, or the topics in a help file. In these cases, it's nice to have a *perfect* hashing function, one that maps different keys into different numbers. With a perfect function, you could retrieve the record (or detect that no record exists) in exactly one probe, without worrying about collisions. A special case of perfect hashing is *minimal* perfect hashing, were no empty locations in the table exist. Minimal perfect hashing satisfies the ideal of fast performance and low memory usage. However, finding perfect hashing functions is difficult. In this article, I'll present a simple algorithm to find perfect (and optionally minimal) hashing functions for a static set of data.

Cichelli's functions

A family of perfect hashing functions was introduced by R. Cichelli in [1]. Cichelli suggests an exhaustive search to find such functions, and discusses some heuristics to speed the search. I have extended the heuristics to obtain a general implementation and speed up the original algorithm by more than an order of magnitude.

Figure 1:
Minimal perfect hashing for days of the week

Word	Hash
sunday	0
monday	5
tuesday	3
wednesday	6
thursday	4
friday	1
saturday	2

Letter	Value
f	-8
m	-4
s	-9
t	-7
w	-6
y	3

Cichelli's hashing functions are very simple. Given a word *w*, the hash value is Value(First_Letter(*w*)) + Value(Last_Letter(*w*)) + Length(*w*). The mapping function Value() is normally implemented as a table that maps characters into numbers. Figure 1 is an example of table and hash values of a minimal perfect hashing function for the days of the week. Note that you only need to keep a small table for Value(), since only first and last letters need to be mapped. Calculating the hash value takes only two accesses in the table and two additions.

Cichelli's algorithm

The original algorithm uses an exhaustive approach with backtracking. Every possible mapping is tried for *Value*(), and whenever an assignment results in a collision, the search path is pruned and another solution is tried. The algorithm is easy to code, but can take exponential time (in the number of words) to calculate.

Cichelli suggested two ways to speed up this approach. One way is to sort the keys: compute the frequency of the first and last letter, then sort the words according to the sum of the frequency for the first and last letters. This guarantees that the most common letters are assigned values early, and increases the probability that collisions are found when the depth of the search tree is still small. This reduces the cost of backtracking.

This isn't always sufficient, however. Consider the set of keys:

{*aa, ab, ac, ad, ae, af, ag, bd, cd, ce, de, fg*},

which sorts to

{*aa, ad, ac, ae, ab, af, ag, cd, de, bd, ce, fg*}.

Now, the value of the key *cd* is determined as soon as *ac* is given a value, but a test for collision on *cd* is not done until five more branches in the search tree are taken. This makes a collision on *cd* much more costly than it ought to be.

Cichelli suggested a second heuristic to address the problem. If a key in this sorted list has letters that have already occurred, the key is placed immediately after the one that completes its mapping. In the aforementioned example, *cd* will be moved immediately after *ac*. While doing so, you must try to maintain the original sorting order whenever possible: listing 1 is a snippet of my C++ implementation.

There are five steps to Cichelli's algorithm:

(1) Calculate the frequency of first and last letters.
(2) Order the keys in decreasing order of the sum of the frequencies.
(3) Reorder the keys from the beginning of the list, so that if a key's first and last letters have already appeared, that key is placed next to the completing key in the list.
(4) Search cycle: for each key, if both values for first and last letters are already assigned, try to place the key in the table (step 5); otherwise, if only one value needs to be assigned, try all the possible assignments. If both values must be assigned, use two nested loops to try all the possible assignments for the two letters.
(5) Given the key, the length, and the *Value* mapping, compute the address in the table. Backtrack to step (4) if there is a collision; otherwise place the key in the table. If is the last letter, a perfect hashing has been found; otherwise, recursively go into step (4) for the next key.

The table size is not fixed. If the table size is equal to the number of keys, we are looking for a minimal perfect hashing; otherwise, some empty buckets remain at the end of the process.

Improvements

Cichelli did not suggest any way to restrict the range of values to try for each letter. This range plays an important role in the algorithm. A small range may result in no solutions, and an overly large range may increase the width of the search tree unmanageably.
I've developed a general rule that works fine for most sets, resulting in a moderately small range that does not exclude solutions. Since you add the length of the word and you want to obtain small index values (including zero), most of the letters will be assigned to negative values. Also, the letter values will never be very large because the size of the table is an upper bound for the index.
I chose a range of [-*MaxLen*, *TableSize* - *MinLen*), where *MaxLen* is the maximum length of a key belonging to the set; *MinLen* the minimum length of a key; and *TableSize* the size of the hash table. The range is normally small (for the keys in figure 1, assuming minimal hashing, it is equal to [-9,1)) but still effective.

This can be further improved by adapting the range dynamically. Consider what happens when one of the two letters has already been assigned a value. This occurs twice in the algorithm: when only one value must be assigned and in the inner of the two nested loops. In this case, you already know one of the values (say *v*) and the length of the word, so you can recalculate the range as [-*v* - *Len*, *TableSize* - *v* - *Len*], where *Len* is the length of the key to which you are trying to assign an index. This optimization alone accounted for more than an order-of-magnitude improvement in the performance of the algorithm: I've frequently seen a speedup of 40 or more. Note that the dynamic range may occasionally extend the original one: for instance, in figure 1 *y* is assigned a value of 3, that is outside the static range. Further optimization of the dynamic range can be obtained by considering which buckets are free. If you keep track of the first and last free buckets in the hash table, the dynamic range can be restricted to [*FirstFree* - *v* - *Len*, *LastFree* - *v* - *Len*]. Listing 2 is my C++ implementation of steps (4) and (5). I also added some logic to stop when the first solution is found.

The result is a fairly quick algorithm: a minimal perfect hash for the 32 keywords of ANSI C (see figure 2) can be found in less than one second on a Pentium/90, and the 46 non-colliding keywords of the C++ need about four seconds. The complete C++ implementation of my algorithm is available in the Dr. Dobb's forum of directly from me via email. It has been compiled and tested on several different architectures and operating systems, and it should work without changes on most systems.

Figure 2:
Minimal perfect hashing for ANSI C keywords

Letter	Value	Letter	Value
a	-5	l	29
b	-8	m	-4
c	-7	n	-1
d	-10	o	22
e	4	r	5
f	12	s	7
g	-7	t	10
h	4	u	26
i	2	v	19
k	31	w	2

Word	Hash	Word	Hash
auto	21	int	15
break	28	long	26
case	1	register	18
char	2	return	10
const	8	short	22
continue	5	signed	3
default	7	sizeof	25
do	14	static	6
double	0	struct	23
else	12	switch	17
enum	4	typedef	29
extern	9	union	30
float	27	unsigned	24

for	20	void	13
goto	19	volatile	31
if	16	while	11

Collisions resolution and extensions

Some words, as the C++ keywords *double* and *delete*, results into a collision no matter which assignment is selected. There are several ways to overcome this problem: the first is to enhance the algorithm to consider letter positions other than the first and last. Another is to consider more than two letters. For instance [2] Haggard and Karplus tried using all letters with encouraging results.

Another idea is to partition the set of words in two or more sets. This is effective also when there are many words, and the algorithm takes hours (or days) to calculate a perfect hashing. The best idea would be to partition the set of keys into disjoint sets of first and last letters. This allows you to use a single hash table simply by adding an offset to each partition. Unfortunately, the distribution of letters usually does not allow such a partition, and you must often resort to multiple tables, and consequently to multiple probes.

I'm currently investigating two very promising ideas, one based on the concept of articulation point in a graph, which should reduce backtracking, and another based on a look-ahead technique to prune dead paths more quickly. Both techniques should reduce the computational complexity of the algorithm in the average case, so that larger set of keys can be conveniently mapped. If you are interested in the results of my research, drop me an email message.

eng	mphash.html

Bibliography

[1] R. Cichelli, "*Minimal Perfect Hashing Made Simple*", Comm. ACM Vol. 23 No. 1, Jan. 1980.

[2] Haggard and Karplus, "*Finding Minimal Perfect Hash Functions*", ACM SICCSE Bull., Vol. 18, No. 1, Feb. 1986

Biography

Carlo Pescio holds a doctoral degree in Computer Science and is a consultant and mentor for various European companies and corporations, including the Directorate of the European Commission. He specializes in object oriented technologies and is a member of IEEE Computer Society, the ACM, and the New York Academy of Sciences. He lives in Savona, Italy and can be contacted at pescio@eptacom.net.

```
Listing 1

void WordSet :: Reorder()
  {
  // reorder items in the set, so that if an item
  // first and last letter occurred before, the
  // item is moved upward in the list. This heuristic
  // maximize the likelihood that collisions are found
  // early in the assignment of values.

  const unsigned NO_OCCURRENCE = MAXINT ;
  for( unsigned i = 1; i < card; i++ )
    {
    const Word& w = *(set[ i ]) ;
    char first = w.First() ;
    char last = w.Last() ;
    unsigned firstOccurredIndex = NO_OCCURRENCE ;
    unsigned lastOccurredIndex = NO_OCCURRENCE ;
    for( unsigned j = 0; j < i; j++ )
      {
      const Word& prev = *(set[ j ]) ;
      if( lastOccurredIndex == NO_OCCURRENCE &&
          ( prev.Last() == last || prev.First() == last ) )
        lastOccurredIndex = j ;
      if( firstOccurredIndex == NO_OCCURRENCE &&
          ( prev.First() == first || prev.Last() == first ) )
        firstOccurredIndex = j ;
      if( firstOccurredIndex != NO_OCCURRENCE &&
          lastOccurredIndex != NO_OCCURRENCE )
        {
        // both letters occurred before: move the item upward, by
        // pushing down the ones between the maximum occurence
        // index and the item index itself. This preserve the
        // original ordering as much as possible
        unsigned firstToMoveDown =
           max( firstOccurredIndex, lastOccurredIndex ) + 1 ;
        unsigned lastToMoveDown = i - 1 ;
        Word* itemToMoveUp = set[ i ] ;
        for( unsigned k = lastToMoveDown; k >= firstToMoveDown; k-- )
          set[ k + 1 ] = set[ k ] ;
        set[ firstToMoveDown ] = itemToMoveUp ;
        break ; // item 'i' handled
        }
      }
    }
  }



bool HashFinder :: Search( unsigned index )
  {
  const Word& w = words[ index ] ;
  if( value[ w.First() ] != NO_VALUE && value[ w.Last() ] != NO_VALUE )
    {
    // try to insert into the table
    int h = Hash( w ) ;
    if( h < 0 || h >= maxTableSize || ! buckets.IsFree( h ) )
      return( FALSE ) ;
    else
      {
      buckets.MarkUsed( h ) ;
      index++ ;
      int card = words.Card() ;
      if( index == card )
        {
        cout << "------------\n" ;
        cout << "perfect hashing found\n" ;
        for( int i = 0; i < CHAR_MAX; i++ )
          if( value[ i ] != NO_VALUE )
            cout << (char)i << " = " << value[ i ] << "  " ;
        cout << endl ;
        for( i = 0; i < card; i++ )
          cout << words[ i ] << " = " << Hash( words[ i ] ) << endl ;
        // calculate and print loading factor
        int used = 0 ;
        for( i = 0; i < maxTableSize; i++ )
          if( ! buckets.IsFree( i ) )
            used++ ;
        cout << "loading factor: " << (float)used/maxTableSize << endl;

        // decide if it is time to stop now
        stopNow = stopFirst || ( used == maxTableSize && stopMinimal );
```

```
          buckets.MarkFree( h ) ; // restore for backtracking
          return( TRUE ) ;
          }
        else
          {
          bool res = Search( index ) ;
          buckets.MarkFree( h ) ; // restore for backtracking
          return( res ) ;
          }
        }
      }
    else if( value[ w.First() ] == NO_VALUE &&
             value[ w.Last() ] == NO_VALUE )
      {
      // must find values for both first and last;

      int firstValToAssign = w.First() ;
      int lastValToAssign = w.Last() ;
      bool found = FALSE ;
      for( int i = minVal; i < maxVal && ! stopNow; i++ )
        {
        value[ firstValToAssign ] = i ;
       // here we can change the heuristics: we already assigned one of
       // the values, so we have modified the boundaries.
       // Now we must have FirstFree <= to-assign + 'i' + len <= LastFree
        int minVal2 = buckets.FirstFree() - i - w.Len() ;
        int maxVal2 = buckets.LastFree() - i - w.Len() + 1 ;
        for( int j = minVal2; j < maxVal2 && ! stopNow; j++ )
          {
          value[ lastValToAssign ] = j ;
          bool res = Search( index ) ;
          found = found ||  res ;
          }
        }
      value[ firstValToAssign ] = NO_VALUE ; // restore for backtracking
      value[ lastValToAssign ] = NO_VALUE ;
      return( found ) ;
      }
    else
      {
      // must find valid values for the one without values
      int valToAssign ;
      int valAssigned ;
      if( value[ w.First() ] == NO_VALUE )
        {
        valToAssign = w.First() ;
        valAssigned = w.Last() ;
        }
      else
        {
        valToAssign = w.Last() ;
        valAssigned = w.First() ;
        }

      // see above for heuristics used here
      int minVal2 = buckets.FirstFree() - value[ valAssigned ] - w.Len();
```

```
  int maxVal2 = buckets.LastFree() -value[valAssigned] - w.Len() + 1;

  assert( value[ valToAssign ] == NO_VALUE ) ;
  bool found = FALSE ;
  for( int i = minVal2; i < maxVal2 && ! stopNow; i++ )
    {
    value[ valToAssign ] = i ;
    bool res = Search( index ) ;  // will take first branch this time
    found = found ||  res ;
    }
  value[ valToAssign ] = NO_VALUE ; // restore for backtracking
  return( found ) ;
  }
}
```