Propagation Tables in Hoffman

Brent Baccala

May 23, 2008

Hoffman is a program to solve chess endgames using retrograde analysis. Retrograde analysis is only useful in the endgame, runs very slowly, and produces enormous amounts of data. Its great advantage lies in its ability to completely solve the endgame. In a very real sense, a retrograde engine has no "move horizon" like a conventional chess engine. It sees everything.

A Hoffman analysis can be quite space-intensive. Since its memory utilization pattern is basically random, Hoffman will begin to swap dramatically and suffer a disastrous drop in performance once its working set size exceeds the machine's available memory. To alleviate this, the program can be operated in a mode where it fills a series of *propagation tables*, writes each one out to disk when full, then reads them back in sequentially during the next pass. Although less efficient than when the working set can be contained in memory, propagation tables allow the program to build tablebases of essentially unlimited size with no swapping and reasonable CPU utilization.

Propagation tables are implemented using an in-memory bucket sort and then a merge tree (described by Knuth in [1]) to combine the disk files. Probably the most novel aspect of the implementation (to my knowledge) is the use of inversion in a finite field to smooth the distribution of index numbers.

The Algorithm

The basic operation of the program is to make multiple sweeps through an "entries" table, with the processing of each entry triggering changes to other, related entries. The basic problem is that the entries table can not fit into memory. Therefore, we store the entries table on disk and sweep through it sequentially. To avoid the random disk accesses that would be required by the update operation, we maintain a sorted table (the "proptable") in memory that contains the information needed to perform an update. A complete proptable can not fit in memory, either, so we write partial proptables out to disk as they fill, each to a different file. Once a pass is complete, the output proptables become input proptables for the next pass. Each (sorted) input proptable is read sequentially along with the entries file, the updates from all the proptables are passed through an in-memory merge tree, and the sorted updates are applied sequentially as we move through the entries table, generating a new set of output proptables as we go.

Bucket Sort

The in-memory sorted table is built using a varient of bucket sort; see also Knuth's discussion of the address calculation sort on p. 99 of [1].

The basic idea is to start with an empty array and insert the entries into roughly the positions they are predicted to occupy based on their keys. The sort works best when these keys (called indices, in our case) are evenly spread, so we invert the indices modulo a prime number in order to achieve this even spread. This is computationally expensive, but we expect to be disk-bound anyway.

The table is initialized to all-ones (since zero may be used an as index, and "inverts" to itself). Given an index to be inserted, we divide it by a scaling factor (the truncated integer ratio between the number of indices and the length of the proptable) and attempt to insert at that point in the proptable. If the slot there is empty, we insert and are done. If the slot is occupied, we compare its entry to the new one and either merge them (if they are equal) or begin scanning in the appropriate direction until we find either an empty slot (we use it and are done) or two adjacent entries whose indices bracket the one we're trying to insert.

In this case, we begin scanning simultaneously in both directions from the adjacent entries (one slot to the left, one slot to the right, then two to the left, two to the right, etc) until we either find an empty slot or have hit some limit on how far we may search (currently 25 slots in either direction). If we didn't hit the limit, we shift a block of entries one slot to move the empty slot between the two adjacent entries originally identified and insert the new element there. If we hit the limit, we declare the table "full", write the occupied slots out to disk by making a linear sweep through the entire table (clearing the slots as we go), and insert the new element into the now-empty table at its originally predicted location.

This algorithm could probably use a careful complexity analysis, specifically with regard to tuning how it chooses when to write the table out to disk. The 25 slot limit is just a guess that seems to trigger at about 50-60% occupancy.

The Merge Tree

A merge tree is used when reading the proptables back in from disk. Its design was taken almost directly from [1], §5.4.1 Multiway Merging and Replacement Selection. The key observation is that the input files are sorted, so we can read them sequentially, and need only a method of selecting which of them has the next item in a global sort.

The merge tree is illustrated in Figure 1; it is implemented as an array of proptable entries with the array indices shown (the exact size is scaled according to the number of input propfiles). The sorted entries from the input propfiles are fed in at the left. Moving right, at each node a comparison is made between the two nodes linked from the left and the lowest entry is selected and placed into the array at that point. In the example, the first proptable is fed (one entry at a time) into the array at index 8, and the second proptable is fed into the array at index 9. The smaller of the two entries at array indices 8 and 9 is copied into the array at index 4, and so on.

Clearly, array index 1 will contain the lowest entry from all the inputs. It is removed and processed.

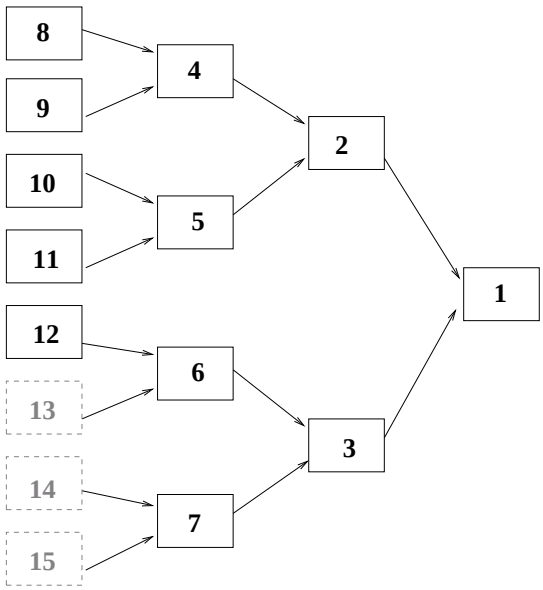

Figure 1: A Merge Tree

Let's say it was originally from the first proptable (index 8). The next entry from the first proptable now needs to be inserted at index 8 and the tree updated. Yet note first that the only updates required are along the path from index 8 to the root at index 1. Note further that although the node numbers have been assigned in a fairly obvious way, the index number of the node to the right of any given node can always be computed simply by right shifting one bit.

So our update algorithm is quite simple. Along with the entries moving through the array, we also track where they originally came from (using a parallel array). We remove the next entry to be processed from array index 1, take the next entry from that file and insert it into the array at the correct index on the left hand side. Then we loop on the array index, right shifting by one bit each time around and at each step comparing entries at indices $2i$ and $2i + 1$ and putting the smaller of the two in index i. We terminate when we've recomputed the value in index 1, which is the new next element to be processed.

If we don't have an even power of two number of input proptables, then we round up, and some of the slots in the array (the gray dashed boxes in Figure 1) are initialized with an "infinite" (i.e, all-ones) value. The algorithm then proceeds normally. Likewise, as input is exhausted from individual files, we insert all-ones values into the corresponding array entries. Once an all-ones value appears at node 1, we have processed all the available input.

References

[1] Knuth, Art of Computer Programming, Vol 3, 2nd Ed.

[2] `http://en.wikipedia.org/wiki/Bucket_sort`