A program is disclosed that reorganises the layout of a language dictionary which is implemented as a finite-state transducer in such a way that optimal use is made of cache memory when executing the reorganised finite state machine.

Finite-state processing typically has simple access code, so it is the speed of the memory access which is crucial for the performance. Modern computers contain several storage devices which are arranged in a hierarchy starting with disk storage which has a very large capacity but slow access speed and ending with processor cache memory (and registers) which has very small capacity but extremely fast access speed. As you move up each level in the storage hierarchy the memory gets faster, but the capacity decreases.

Finite-state transducers (FSTs) are implemented by mapping the nodes in the transducer's network into linear memory locations. It is apparent, that the choice of the mapping will have impact on the performance of finite-state processing, because it directly affects the performance of caching. This mapping is typically done by the internal logic of the builder, which may be good or bad in terms of cache friendliness, but could hardly expected to be the optimal unless this factor is explicitly considered during the builder design.

In modern computers, the hardware and operating system attempt to provide optimal cache usage by loading an entire block of memory into the cache at the same time. This strategy only works well because most programs show a pattern of accessing memory in a sequential way. However, if we are executing a finite-state machine whose nodes have been randomly mapped to memory locations, this will require frequent reloading of the cache. On the other hand, if the next visited node in the finite-state machine is stored close to the node currently being visited, there is an increased chance that the memory location containing this next node will already be in the cache.

Description of the Optimisation Algorithm

We tested four optimisation algorithms, shortly described below. The following is a detailed description of the algorithm which gave the best results in our experiments. It can be described as a depth first reordering using traversal frequency as a basis for ordering.

1. Construct an original FST with arbitrary placement of nodes. However each transition in the FST should have an additional field called visit_count, which is initially set to 0.
2. Use this original to perform dictionary look-ups on a large corpus which is statistically representative of the text which will be analysed with this dictionary in the future. Each time an arc in the FST is traversed, you increment the value of the visit_count field
3. Now you start the optimization phase, to do this you need to create two integer arrays forward_map and backward_map. Each of these arrays should contain as many elements as there are nodes in the FST and each element should be initially set to 0. You also need to generate a static integer variable max_assign containing the highest node number assigned so far (initially 0).
4. Start at node 0 of the original FST. Choose the outgoing transition from this node which has the highest visit_count. Assign this the next free space in the new FST (i.e. max_assign + 1) this is done by making an entry into the forward_map array and the backward_map array. The reason for doing this is

because this is the node which will most often be visited next and so for cache friendliness it should be as near as possible to this node.

5. Increment the max_assign variable and then repeat the previous step using the target node of the transition you just picked. This function can be recursively called with the one proviso that you need to check that the node you are examining has not already been assigned a new position (this can happen if the FST has loops).

6. When you reach a node which has no outgoing transition, you back track to the previous node in the path which got you there and repeat the algorithm with the node pointed to by the arc with the next highest visit_count.

7. When all transitions have been reassigned you step back further in the path until you reach the first node. When you have traced all paths from the first node, you have remapped the entire FST.

8. Now you start constructing the optimized FST. You do this by looking through the backward_map to find which node in the source FST corresponds to the node we are writing in the optimized FST. When writing out each node we will have to replace the state numbers on each of the outgoing transitions by consulting the forward_map array.

Measurement of Efficiency of Optimization algorithms

In order to test the effect of reordering upon actual performance, we conducted real world trails with the following optimisation algorithms:

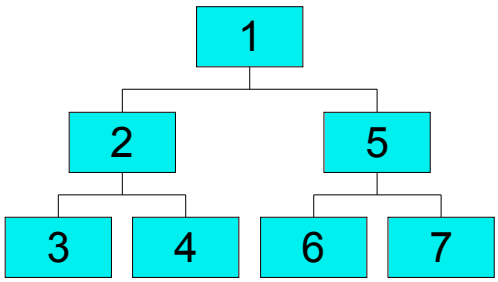
Algorithm A1: Assign node numbers in the same order as nodes are visited during look-up for N most frequently used English words. This algorithm does not calculate visit_count - it simply tries N most frequent words in descending order and assigns new state numbers to visited states starting from 1 (if a state is visited a second time then its number remains unmodified). Visited nodes will be renumbered as you described and moved to the beginning of FST. Unvisited nodes will be moved to the end of FST preserving their original relative order.

Algorithm A2: As in A1 except we also calculate a visit_count and then reorder states by visit_count with various restrictions. We do not reorder nodes whose visit_count > 5, because the statistics for such infrequent nodes are not reliable. During trials we limited the number of affected states to only top 1%, 10%, or 50% of states - thus we only affect order of nodes with reasonable statistical coverage.

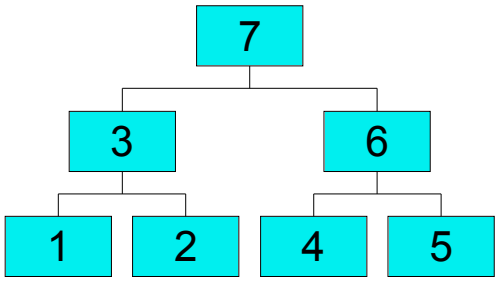
Algorithm A3: Reordering using standard depth-first graph traversal method. Node numbers are assigned in the order that nodes are visited during traversal of the graph in depth-first mode (i.e. starting at a parent node we visit the first child node and then visit all of the child's children before visiting the second child). There are two versions of this algorithm, preorder traversal and postorder traversal. In preorder traversal, the parent is assigned the lowest state number, while in the postorder traversal mode the children are all assigned lower state numbers then the parent node. These two ordering schemes are illustrated by the diagrams below.

Algorithm A4: A variation of A3 which is described in detail above. Graph traversal using depth-first, preorder (parent first) algorithm in which children of each node are traversed depending on their visit_count in descending order.

Preorder Traversal



Postorder Traversal



It is standard practice to alphabetically sort the source dictionary prior to building the FST. This is done because it is easier to maintain a dictionary in which the entries have a defined order. However, sorting the source dictionary can speed up the dictionary build program and also produces a FST which is organised in a fairly cache-friendly way. The reason for this is because the new states require by each new word that is added to the FST are normally added at the end of the existing FST and hence will tend to be close to the states required for processing the common prefix. In order to measure the effect of this ordering, we also conducted some experiments supposed to make dictionary less cache friendly.

Test T1: Shuffle the built dictionary by reversing the order of even numbered states. For example 1 2 3 4 5 6 7 8 9 => 1 8 3 6 5 4 7 2 9.

Test T2: Use randomly shuffled source dictionary file (Note: by default source files are sorted alphabetically).

For test purposes, we created an English FST dictionary and then reorganised the dictionary with each of the algorithms described above. We then tested the speed of our analysis tool using each of the reorganised dictionaries when used to analyse a sample corpus of English text. The test machine used has Intel Celeron FC-PGA 633MHz (66MHz system bus clock) with 128MB RAM (PC100) running Windows2000*. We repeated each test 100 times.

Reorganisation Algorithm	Speed (10**9 of UTF16 code units per hour)	Error (relative mean square error)	Acceleration
A4	14.512	1.525%	+3.986%
A3 (Preorder traversal)	14.266	0.492%	+2.224%
A3 (Postorder traversal)	14.155	0.627%	+1.428%
A2 - Top 1% of states	14.091	0.651%	+0.964%
A1 (100 words)	13.977	1.030%	+0.148%
Standard - Reference point	13.956	0.629%	0.000%
A1 (1000 words)	13.912	1.203%	-0.313%
A2 - Top 10% of states	13.551	1.044%	-2.902%
T2 - Randomly shuffled source file	13.484	1.479%	-3.385%
A2 - Top 50% of states	12.670	1.845%	-9.215%
T1 - Shuffle the built dictionary	12.503	0.564%	-10.410%

Conclusions:

Cache optimizing in the sense of reordering of states may affect performance up to 15%

Algorithm A4 gave the best results in our experiments

It is easy to deteriorate performance if cache optimizing (reach lower bound of speed)