Fra D. Gusfield, "Algorithms on Strings, Trees, and Sequences".

12.7. The Four-Russians speedup

In this section we will discuss an approach that leads both to a theoretical and to a practical speedup of many dynamic programming algorithms. The idea, comes from a paper [28] by four authors, Arlazarov, Dinic, Kronrod, and Faradzev, concerning boolean matrix multiplication. The general idea taken from this paper has come to be known in the West as the Four-Russians technique, even though only one of the authors is Russian.[10] The applications in the string domain are quite different from matrix multiplication, but the general idea suggested in [28] applies. We illustrate the idea with the specific problem of computing (unweighted) *edit distance*. This application was first worked out by Masek and Paterson [313] and was further discussed by those authors in [312]; many additional applications of the Four-Russians idea have been developed since then (for example [340]).

12.7.1. t-blocks

Definition A *t-block* is a t by t square in the dynamic programming table.

The rough idea of the Four-Russians method is to partition the dynamic programming table into *t-blocks* and compute the essential values in the table one t-block at a time, rather than one cell at a time. The goal is to spend only $O(t)$ time per block (rather than $\Theta(t^2)$ time), achieving a factor of t speedup over the standard dynamic programming solution. In the exposition given below, the partition will not be exactly achieved, since neighboring t-blocks will overlap somewhat. Still, the rough idea given here does capture the basic flavor and advantage of the method presented below. That method will compute the edit distance in $O(n^2/\log n)$ time, for two strings of length n (again assuming a fixed alphabet).

[10] This reflects our general level of ignorance about ethnicities in the then Soviet Union.

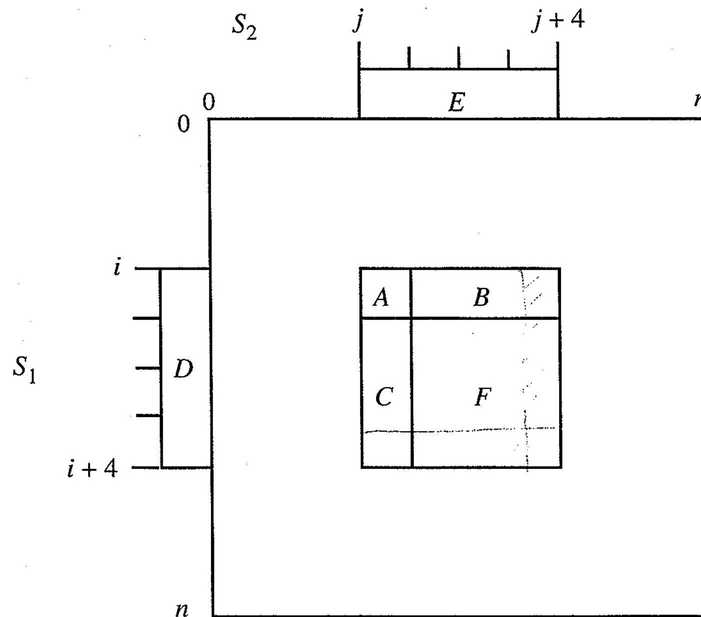

Figure 12.21: A single block with $t = 4$ drawn inside the full dynamic programming table. The distance values in the part of the block labeled F are determined by the values in the parts labeled A, B, and C together with the substrings of S_1 and S_2 in D and E. Note that A is the intersection of the first row and column of the block.

Consider the standard dynamic programming approach to computing the edit distance of two strings S_1 and S_2. The value $D(i, j)$ given to any cell (i, j), when i and j are both greater than 0, is determined by the values in its three neighboring cells, $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$, and by the characters in positions i and j of the two strings. By extension, the values given to the cells in an entire t-block, with upper left-hand corner at position (i, j) say, are determined by the values in the first row and column of the t-block together with the substrings $S_1[i..i+t-1]$ and $S_2[j..j+t-1]$ (see Figure 12.21). Another way to state this observation is the following:

Lemma 12.7.1. *The distance values in a t-block starting in position (i, j) are a function of the values in its first row and column and the substrings $S_1[i..i+t-1]$ and $S_2[j..j+t-1]$.*

Definition Given Lemma 12.7.1, and using the notation shown in Figure 12.21, we define the *block function* as the function from the five inputs (A, B, C, D, E) to the output F.

It follows that the values in the last row and column of a t-block are also a function of the inputs (A, B, C, D, E). We call the function from those inputs to the values in the last row and column of a t-block, the *restricted block function*.

Notice that the total size of the input and the size of the output of the restricted block function is $O(t)$.

Computing edit distance with the restricted block function

By Lemma 12.7.1, the edit distance between S_1 and S_2 can be computed using the restricted block function. For simplicity, suppose that S_1 and S_2 are both of length $n = k(t-1)$, for some k.

Figure 12.22: An edit distance table for $n = 9$. With $t = 4$, the table is covered by nine overlapping blocks. The center block is outlined with darker lines for clarity. In general, if $n = k(t-1)$ then the $(n+1)$ by $(n+1)$ table will be covered by k^2 overlapping t-blocks.

Block edit distance algorithm

Begin

1. Cover the $(n + 1)$ by $(n + 1)$ dynamic programming table with t-blocks, where the last column of every t-block is shared with the first column of the t-block to its right (if any), and the last row of every t-block is shared with the first row of the t-block below it (if any). (See Figure 12.22). In this way, and since $n = k(t - 1)$, the table will consist of k rows and k columns of partially overlapping t-blocks.
2. Initialize the values in the first row and column of the full table according to the base conditions of the recurrence.
3. In a rowwise manner, use the *restricted* block function to successively determine the values in the last row and last column of each block. By the overlapping nature of the blocks, the values in the last column (or row) of a block are the values in the first column (or row) of the block to its right (or below it).
4. The value in cell (n, n) is the edit distance of S_1 and S_2.

end.

Of course, the heart of the algorithm is step 3, where specific instances of the restricted block function must be computed. Any instance of the restricted block function can be computed $O(t^2)$ time, but that gains us nothing. So how is the restricted block function computed?

12.7.2. The Four-Russians idea for the restricted block function

The general Four-Russians observation is that a speedup can often be obtained by *precomputing* and storing information about all possible instances of a subproblem that might arise in solving a problem. Then, when solving an instance of the full problem and specific subproblems are encountered, the computation can be accelerated by looking up the answers to precomputed subproblems, instead of recomputing those answers. If the subproblems are chosen correctly, the total time taken by this method (including the time for the precomputations) will be less than the time taken by the standard computation.

In the case of edit distance, the precomputation suggested by the Four-Russians idea is to enumerate all possible inputs to the restricted block function (the proper size of the block will be determined later), compute the resulting output values (a t-length row and a t-length column) for each input, and store the outputs indexed by the inputs. Every time a specific restricted block function must be computed in step 3 of, the *block edit distance algorithm*, the value of the function is then retrieved from the precomputed values and need not be computed. This clearly works to compute the edit distance $D(n, n)$, but is it any faster than the original $O(n^2)$ method? Astute readers should be skeptical, so please suspend disbelief for now.

Accounting detail

Assume first that all the precomputation has been done. What time is needed to execute the *block edit distance algorithm*? Recall that the sizes of the input and the output of the restricted block function are both $O(t)$. It is not difficult to organize the input-output values of the (precomputed) restricted block function so that the correct output for any specific input can be retrieved in $O(t)$ time. Details are left to the reader. There are $\Theta(n^2/t^2)$ blocks, hence the total time used by the *block edit distance algorithm* is $O(n^2/t)$. Setting t to $\Theta(\log n)$, the time is $O(n^2/\log n)$. However, in the unit-cost RAM model of computation, each output value can be retrieved in constant time since $t = O(\log n)$. In that case, the time for the method is reduced to $O(n^2/(\log n)^2)$.

But what about the precomputation time? The key issue involves the number of input choices to the restricted block function. By definition, every cell has an integer from zero to n, so there are $(n+1)^t$ possible values for any t-length row or column. If the alphabet has size σ, then there are σ^t possible substrings of length t. Hence the number of distinct input combinations to the restricted block function is $(n+1)^{2t}\sigma^{2t}$. For each input, it takes $\Theta(t^2)$ time to evaluate the last row and column of the resulting t-block (by running the standard dynamic program). Thus the overall time used in this way to precompute the function outputs to all possible input choices is $\Theta((n+1)^{2t}\sigma^{2t}t^2)$. But t must be at least one, so $\Omega(n^2)$ time is used in this way. No progress yet! The idea is right, but we need another trick to make it work.

12.7.3. The trick: offset encoding

The dominant term in the precomputation time is $(n+1)^{2t}$, since σ is assumed to be fixed. That term comes from the number of distinct choices there are for two t-length subrows and subcolumns. But $(n+1)^t$ overcounts the number of different t-length subrows (or subcolumns) that could appear in a real table, since the value in a cell is not independent of the values of its neighbors. We next make this precise.

Lemma 12.7.2. *In any row, column, or diagonal of the dynamic programming table for edit distance, two adjacent cells can have a value that differs by at most one.*

PROOF Certainly, $D(i, j) \le D(i, j-1) + 1$. Conversely, if the optimal alignment of $S_1[1..i]$ and $S_2[1..j]$ matches $S_2(j)$ to some character of S_1, then by simply omitting $S_2(j)$ and aligning its mate against a space, the distance increases by at most one. If $S_2(j)$ is not matched then its omission reduces the distance by one. Hence $D(i, j-1) \le D(i, j) + 1$, and the lemma is proved for adjacent row cells. Similar reasoning holds along a column.

In the case of adjacent cells in a diagonal, it is easy to see that $D(i, j) \le D(i-1, j-1) + 1$. Conversely, if the optimal alignment of $S_1[1..i]$ and $S_2[1..j]$ aligns i against j,

then $D(i-1, j-1) \leq D(i, j)+1$. If the optimal alignment doesn't align i against j, then at least one of the characters, $S_1(i)$ or $S_2(j)$, must align against a space, and $D(i-1, j-1) \leq D(i, j)$. □

Given Lemma 12.7.2, we can *encode* the values in a row of a t-block by a t-length vector specifying the value of the first entry in the row, and then specifying the difference (offset) of each successive cell value to its left neighbor: A zero indicates equality, a one indicates an increase by one, and a minus one indicates a decrease by one. For example, the row of distances 5, 4, 4, 5 would be encoded by the row of offsets 5, -1, 0, $+1$. Similarly, we can encode the values in any column by such offset encoding. Since there are only $(n+1)3^{t-1}$ distinct vectors of this type, a change to offset encoding is surely a move in the right direction. We can, however, reduce the number of possible vectors even further.

Definition The *offset vector* is a t-length vector of values from $\{-1, 0, 1\}$, where the first entry must be zero.

The key to making the Four-Russians method efficient is to compute edit distance using only offset vectors rather than actual distance values. Because the number of possible offset vectors is much less than the number of possible vectors of distance values, much less precomputation will be needed. We next show that edit distance can be computed using offset vectors.

Theorem 12.7.1. *Consider a t-block with upper left corner in position (i, j). The two offset vectors for the last row and last column of the block can be determined from the two offset vectors for the first row and column of the block and from substrings $S_1[1..i]$ and $S_2[1..j]$. That is, no D value is needed in the input in order to determine the offset vectors in the last row and column of the block.*

PROOF The proof is essentially a close examination of the dynamic programming recurrences for edit distance. Denote the unknown value of $D(i, j)$ by C. Then for column q in the block, $D(i, q)$ equals C plus the total of the offset values in row i from column $j+1$ to column q. Hence even if the algorithm doesn't know the value of C, it can express $D(i, q)$ as C plus an integer that it can determine. Each $D(q, j)$ can be similarly expressed. Let $D(i, j+1)$ be $C+J$ and let $D(i+1, j)$ be $C+I$, where the algorithm can know I and J. Now consider cell $(i+1, j+1)$. $D(i+1, j+1)$ is equal to $D(i, j) = C$ if character $S_1(i)$ matches $S_2(j)$. Otherwise $D(i+1, j+1)$ equals the minimum of $D(i, j+1)+1$, $D(i+1, j)+1$, and $D(i, j)+1$, i.e., the minimum of $C+I+1$, $C+J+1$, and $C+1$. The algorithm can make this comparison by comparing I and J (which it knows) to the number one. So the algorithm can correctly express $D(i+1, j+1)$ as C, $C+I+1$, $C+J+1$, or $C+1$. Continuing in this way, the algorithm can correctly express each D value in the block as an unknown C plus some integer that it can determine. Since every term involves the same unknown constant C, the offset vectors can be correctly determined by the algorithm. □

Definition The function that determines the two offset vectors for the last row and last column from the two offset vectors for the first row and column of a block together with substrings $S_1[1..i]$ and $S_2[1..j]$ is called the *offset function*.

We now have all the pieces of the Four-Russians–type algorithm to compute edit distance. We again assume, for simplicity, that each string has length $n = k(t-1)$ for some k.

Four-Russians edit distance algorithm

1. Cover the n by n dynamic programming table with t-blocks, where the last column of every t-block is shared with the first column of the t-block to its right (if any), and the last row of every t-block is shared with the first row of the t-block below it (if any).
2. Initialize the values in the first row and column of the full table according to the base conditions of the recurrence. Compute the offset values in the first row and column.
3. In a rowwise manner, use the *offset* block function to successively determine the offset vectors of the last row and column of each block. By the overlapping nature of the blocks, the offset vector in the last column (or row) of a block provides the next offset vector in the first column (or row) of the block to its right (or below it). Simply change the first entry in the next vector to zero.
4. Let Q be the total of the offset values computed for cells in row n. $D(n, n) = D(n, 0)+Q = n + Q$.

Time analysis

As in the analysis of the *block edit distance algorithm*, the execution of the *four-Russians edit distance algorithm* takes $O(n^2/\log n)$ time (or $O[n^2/(\log n)^2]$ time in the unit-cost RAM model) by setting t to $\Theta(\log n)$. So again, the key issue is the time needed to precompute the block offset function. Recall that the first entry of an offset vector must be zero, so there are $3^{2(t-1)}$ possible offset vectors. There are σ^t ways to specify a substring over an alphabet with σ characters, and so there are $3^{2(t-1)}\sigma^{2t}$ ways to specify the input to the offset function. For any specific input choice, the output is computed in $O(t^2)$ time (via dynamic programming), hence the entire precomputation takes $O(3^{2t}\sigma^{2t}t^2)$ time. Setting t equal to $(\log_{3\sigma} n)/2$, the precomputation time is just $O(n(\log n)^2)$. In summary, we have

Theorem 12.7.2. *The edit distance of two strings of length n can be computed in* $O\left(\frac{n^2}{\log n}\right)$ *time or* $O\left(\frac{n^2}{(\log n)^2}\right)$ *time in the unit-cost RAM model.*

Extension to strings of unequal lengths is easy and is left as an exercise.

12.7.4. Practical approaches

The theoretical result that edit distance can be computed in $O\left(\frac{n^2}{\log n}\right)$ time has been extended and applied to a number of different alignment problems. For truly large strings, these theoretical results are worth using. But the Four-Russians method is primarily a theoretical contribution and is not used in its full detail. Instead, the basic idea of precomputing either the restricted block function or the offset function is used, but only for *fixed* size blocks. Generally, t is set to a fixed value independent of n and often a rectangular 2 by t block is used in place of a square block. The point is to pick t so that the restricted block or offset function can be determined in constant time on practical machines. For example, t could be picked so that the offset vector fits into a single computer word. Or, depending on the alphabet and the amount of space available, one might hash the input choices for rapid function retrieval. This should lead to a computing time of $O\left(\frac{n^2}{t}\right)$, although practical programming issues become important at this level of detail. A detailed experimental analysis of these ideas [339] has shown that this approach is one of the most effective ways to speed up the practical computation of edit distance, providing a factor of t speedup over the standard dynamic programming solution.