Appendix A. DNA Design Details

A DNA comparator accepts two strands of DNA as input and outputs the genetic distance between the strands. A dynamic programming solution of the DNA comparator can be easily achieved. Array position (0,0) in the upper left of the array contains value 0, and positions $(i,0)$ and $(0,j)$ contain i and j respectively, where the size of the two DNA strands are $m = \max(i)$ and $n = \max(j)$. Place one DNA strand above the array's columns 1 through m, and the other to the left of the array's rows 1 through n. If the left strand element j is the same as the top strand element i, then array element $A_{i,j}$ takes its value from element $A_{i-1,j-1}$. Otherwise, $A_{i,j} = \min(A_{i-1,j}, A_{i,j-1}) + 1$. The genetic distance will be the value in $A_{m,n}$.

This dynamic program needs to be solved along the diagonal, i.e. from the top-left corner towards the bottom-right corner. One way to do this is to solve each of the cells along the line perpendicular to the diagonal, or the anti-diagonal, of the array before moving on to the next anti-diagonal. This can be done in a linear array where on each clock tick the cells in the array represent those of the next anti-diagonal. The comparison of DNA nucleotides, which also needs to be done before the dynamic program can be solved, can occur in a second linear array where one strand of DNA nucleotides is input from one side, and the other from the other side (Figure 15). The results of the nucleotide comparison can be passed on to the anti-diagonal array. Since both strands move through the array on each cycle, they actually move past each other at a rate of two elements per cycle. To prevent this, the elements are only input into the array on every other cycle. That way each element from one strand will be compared with each

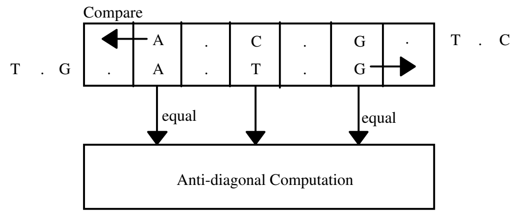

Figure 15. Two linear arrays used in the DNA comparator.

element from the other. The disadvantage of this is that only half of the cells in each array are doing work at any given time.

By examining the rules and the starting condition of the dynamic program it is clear that each element of the array must have a value that is either one larger or one smaller than both its left and top neighbor. Because of this, the diagonals of the dynamic array alternate even and odd. Another interesting feature of this array is that the last row's leftmost value is the length of the left DNA strand and its rightmost value is the genetic distance between the two strands. If we output the value of the last row, we could examine these outputs using software to determine if each value is one more or one less than the one before it. Then from the original strand length and the pattern of increases and decreases we could arrive at the genetic distance.

Of course, we could also just look at the last value output for the genetic distance, but the advantage of the cumulative method is that it can be completed just as easily using only the last two bits of each value. Now we can throw out the upper bits of every number, requiring only two storage bits per cell. Even if the DNA strands are hundreds of elements long, the entire array can be filled out with values less than four, and the genetic distance can be determined by performing the pattern of increments and decrements on the original strand length.

Furthermore, since we know ahead of time which diagonals are even and which are odd, we do not even need the last bit. The entire table can be filled out with only one bit of memory per cell, and the knowledge of which diagonals are even and which are odd. There are several nuisances with this implementation, however. First, there are two different cell types, even and odd, both of which have slightly different logic implementations. This means more work for the designer, and a less robust implementation since the timing of the initial inputs depends on the string length (the first elements of each string need to meet each other in an even cell). Secondly, the logic implementations for each cell are complicated enough that it may have been simpler to keep the lower bit.

The above implementation has been described in several papers[12][9][7][17]. However, all that absolutely needs to be known at the output in order to determine the genetic distance is whether each cell in the last row of the dynamic array is one larger or one smaller than its left neighbor. It turns out that this can be solved for in much the same way, and the logic to do so is simpler. In the new array we want to store two values in each cell. GTL is true if a cell is *greater than* its *left* neighbor, and GTA is true if a cell is *greater than* the neighbor *above* it. If the input nucleotides are equal, a cell in the original array takes the value of its upper left neighbor. So in the new array, GTL is true if the left neighbor's GTA is false (the cell's value in the original array is bigger than its left neighbor's only if the left neighbor's value is smaller than the cell above it). Similarly, GTA is only true if the above neighbor's GTL is false.

If the DNA elements are not equal, the cell's value in the original array would be one larger than the smaller of its neighbors. So in the new array GTL will be true if the left neighbor is smaller than the above neighbor, or if both neighbors are the same size. GTL will only be false if the left neighbor is larger than the neighbor above. In this case, the difference between the left and upper neighbors must be exactly two, because each of these values can only be one away from the upper-left neighbor. In other words, if the left neighbor is larger than the neighbor above, the left neighbor's GTA will be true, and the above neighbor's GTL will be false. Thus the current cell's GTL is false if the left GTA is true and the above GTL is false, and conversely GTL is true if the left GTA is false or the above GTL is true. A symmetrical argument shows the GTA is true if the above GTL is false or the left GTA is true.

Defining the left neighbor's GTA to be LGTA and the above neighbor's GTL to be AGTL, the array can be filled in using the equations:

$$\mathrm{GTL} = (\mathrm{equal} \cdot \overline{\mathrm{LGTA}}) + \left(\overline{\mathrm{equal}} \cdot \left(\overline{\mathrm{LGTA}} + \mathrm{AGTL}\right)\right) \tag{2a}$$

$$\mathrm{GTA} = \left(\mathrm{equal} \cdot \overline{\mathrm{AGTL}}\right) + \left(\overline{\mathrm{equal}} \cdot \left(\overline{\mathrm{AGTL}} + \mathrm{LGTA}\right)\right) \tag{2b}$$

This calculation can be done using the linear array as described earlier. To get the output values out of the linear array, two null inputs were added. If Null_Left is false, the GTL equation above is used. If it is true, GTL = AGTL. Similarly, if Null_Right is false, GTA is calculated as above. If it is true, GTA = LGTA. When the genetic strand is finished the null signal is turned on, and the cells start sending their data toward the outputs at the outermost cells in the linear array. This means that the output is not completely ready until the number of null signals input plus the length of the spaced out strand is twice the length of the linear array. If a strand has length N, it takes 2N cycles to input the strand into the circuit because only one strand element can be input every two cycles. In 2N more cycles, the entire output string has been read.

The throughput of one string comparison every 4N cycles can be doubled by interleaving two problems. Since at any given time only half of the linear array is being used, another problem can be solved simultaneously on the unused portion. To do this, the strands of the two problems should be interleaved, so that each cycle the strand that is being input alternates. This way, each individual strand is only input on every other cycle as allowed. The output will come interleaved as well. With this scheme, two comparisons of strands of length N are completed every $4N$ cycles.