Efficient Implementation of the TKF91-Model

Pierre Barbera and Sarah Lutteropp

September 10, 2015

Abstract

In this document we explain the optimizations we undertook in order to produce an efficient implementation of the TKF91 model by Thorne, Kishino and Felsenstein. We transformed the algorithm into log-space to avoid numerical underflow. This also allowed us to further simplify the formulas. Moreover, we improved data locality by storing the matrix entries in a special data structure. While we also tried different vectorization techniques, it turned out that our fastest code used no vectorization at all. We conclude that we simplified the sequential code to an extent that the benefit of using SIMD instructions cannot compensate for the additional overhead we need for vectorization.

1 Mathematical Optimization

Numerical Underflow Prevention As the TKF91 algorithm [5] performs successive multiplications on, in practice, limited precision floating point numbers, numerical underflow presented a major challenge. This already became an issue for short input sequences, well below 100 nucleotides. To resolve this issue we decided to transform all computations into log-space, adding up logarithms of probabilities instead of multiplying them. While it is still possible to experience under- or overflow after this transformation, it is highly unlikely for practical inputs. Consequently we did not observe any numerical underflow for the tested input parameters and sequence lengths up to and including 10000 nucleotides.

Simplifying the Formulas We were able to get rid of redundant computations be reusing the already computed values for previous matrix entries. For example, we observed that

$$M^0(i+1,0) = M^0(i,0)+\log(\gamma_{i+1})+\log(\zeta_{i+1})+\log(\beta(t))+\log(\pi_{a_{i+1}})+\log(\bar{p_0}(t)).$$

After replacing $\beta(t), \bar{p_0}(t), \gamma_i$ and ζ_i by their formulas in the computation of the matrix entries, we observed that some terms reappeared multiple times. Operating in log-space allowed us to further simplify the formulas. Especially the logarithmic rules $\log(a * b) = \log(a) + \log(b)$ and $\log(a/b) = \log(a) - \log(b)$

allowed us to replace multiplications and divisions with additions and subtractions. The resulting formulas can be found in Appendix A.

Precomputing the Logarithms While replacing a multiplication with two logarithm operations is more costly in general, we did not experience any computational overhead. This is because our formulas consist of sums of only 25 different logarithms which we precomputed in the code.

A major downside of using logarithmic transformation is that any error in its original computation accumulates over the runtime of the program. This occurs because we add up the logarithms multiple times. To investigate this further, we used the high precision mathematical library of the `boost`-framework [1]. It utilizes data types that can dynamically extend their floating point precision to avoid numerical errors and under/overflow. Our basic idea was to use this library only during precomputation, limiting its added overhead. However transforming these data types back into standard double values proved to be problematic, as it introduced additional loss of precision. As a consequence we decided to abandon this path.

2 Matrix Storage

Matrix as Array In the naive approach, we stored each of the matrices M^0, M^1 and M^2 in a separate array, using row-major order. The matrix index at position (i, j) corresponds to the index position $i * (m + 1) + j$ in its array (see Figure 3).

To visualize the shortcomings of this approach, we can use the following example: we assume that six rows of one matrix fit into one cache line. Performing one iteration of the main computation loop then has to load three cache lines: one per matrix. This is depicted in part a) of Figure 1. If we further assume that only two cache lines fit into the cache at one time, every iteration would then cause one of the lines to be swapped out, causing memory overhead.

Array-of-Structs Data Structure The dynamic programming step (see Algorithm 1) accesses the matrices M^0, M^1 and M^2 by the same index position in order to find the maximum entry. Thus we decided to improve data locality by storing the entries from matrices found at the same index position next to each other in memory. For this, we implemented a data structure called `MatrixEntry` that consists of three double values: m_0, m_1 and m_2. Then, we used the Matrix as Array approach as before, but with `MatrixEntry` structs as elements instead of `double` values (see Figure 2).

Continuing the example visualized in Figure 1, with six matrix lines filling one cache line, a cache line is now spread out over all the matrices. Consequently all operations of a single computational iteration will have to access, at most, two cache lines. In the same scenario as before where we can only fit two cache lines into the cache, cache misses will now occur only when their row boundaries are overstepped.

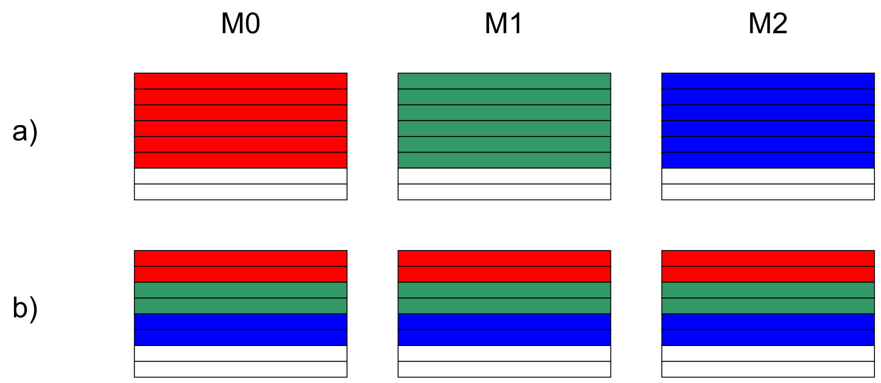

Figure 1: Relation of cache lines and matrices, using the method of allocating each matrix separately(**a)**) and the Array-of-Structs method (**b)**). Each color signifies a different cache line.

Through our analysis using the `perf stat` tool we found out that using the `MatrixEntry` data structure indeed reduced the number of page faults compared to storing each of the matrices separately in an array.

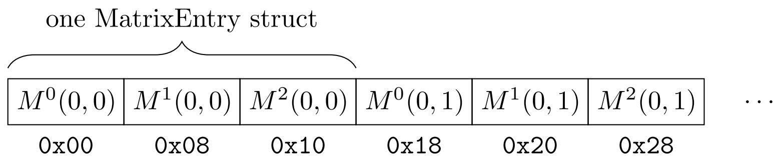

Figure 2: Array-of-Structs data structure used to store the three matrices in memory. Each `MatrixEntry` stores the entries of a single coordinate for all three matrices. The structs are stored contiguously, as the figure depicts, in row-major order.

Algorithm 1: The dynamic programming step, row-major version

```
...// some initialization, see Appendix A.1
for $i = 1, \ldots, n$ do
    for $j = 1, \ldots, m$ do
        ...// some initialization, see Appendix A.2
        $coord \leftarrow CO(i, j)$;
        $up \leftarrow CO(i, j-1)$;
        $diag \leftarrow CO(i-1, j-1)$;
        $left \leftarrow CO(i-1, j)$;
        $m[coord].m_0 \leftarrow m[coord].m_0 + \max\{m[left].m_0, m[left].m_1, m[left].m_2\}$;
        $m[coord].m_1 \leftarrow m[coord].m_1 + \max\{m[diag].m_0, m[diag].m_1, m[diag].m_2\}$;
        $m[coord].m_2 \leftarrow m[coord].m_2 + \max\{m[up].m_1, m[up].m_2\}$;
    end
end
```

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Figure 3: Row-major indexing

0	1	3	6	10
2	4	7	11	14
5	8	12	15	17
9	13	16	18	19

Figure 4: Wavefront indexing

Other Storage Attempts Since each iteration in the dynamic programming step accesses the top, left and upper-diagonal elements of the matrices, a wavefront-like storage of the matrix (see Figure 4) entries yields the lowest amount of expected cache misses. Finding an easy to compute closed formula that maps the index position (i, j) depicting row and column of a matrix to wavefront-coordinates turned out to be a challenge. The index of the diagonal is $i + j$, the sum of row index and column index. Finding the number of elements in the diagonal and especially the correct position within the diagonal proved more difficult though. Since we did not find an easy formula for the mapping, we tried to store the precomputed mappings of index positions into an extra matrix. This still slowed down the program as computing the mapped indices required more arithmetic operations than we needed for the TKF algorithm. Figure 5 shows the main problem that has to be solved in order to make wavefront indexing effective: Given an wavefront index k, how do we obtain the indices for the upper, upper-left-diagonal and left element of the matrix? While the needed offsets stay the same within a diagonal, they change between different diagonals in the matrix.

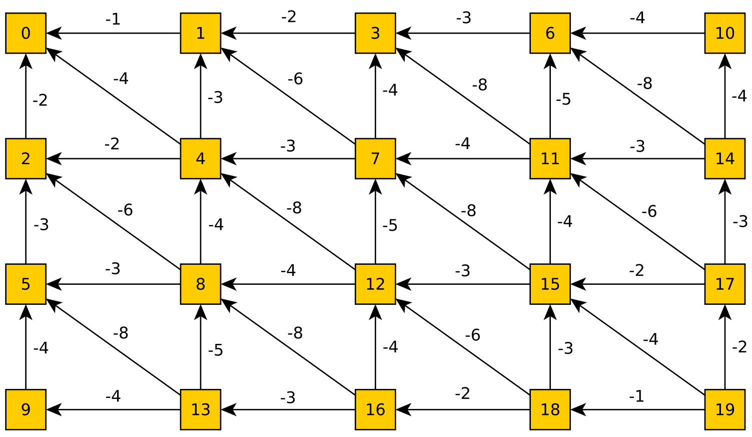

Figure 5: Offsets in wavefront indexing

3 Vectorization Attempts

Over the course of the practical we conducted partial vectorization of two of our implementations: the basic Log-space implementation (Section 1) and the improved Log-space implementation, using a more cache efficient data structure to store the matrices (Section 2).

In both cases we directed our focus primarily to vectorization of the part of the code that initializes the three matrices. Our analysis using `Valgrind callgrind` has revealed this portion to be the part in which $75 - 80\%$ of the runtime of the program is spent. Additionally, as the initialization calculations of each element in the three matrices are independent of each other, it presented itself as *embarrassingly parallel*.

There are several steps that we took to vectorize code in both instances. First, for vector intrinsics to function correctly, it had to be assured that the respective `load` and `store` operations only interact with correctly aligned memory addresses. Secondly, we neede to find an appropriate strategy to determine how to iterate through the matrices, and how to handle matrix dimensions that are not divisible by the width of the type of vector intrinsic used. Lastly we needed to store all calculation parameters in the appropriate vector types and we had to supplant all basic operations of the calculation by their respective vector intrinsics counterparts.

3.1 Log-space implementation

For the vectorization of the basic Log-space implementation we ensured correct alignment of memory using the `__attribute__(aligned(`*`size`*`))` keywords for stack allocation, and the `posix_memalign` function for heap allocation.

As a consequence, we had to change the iteration through the matrices to start at entry 0 of each row, as starting iteration at position 1 would result in memory access of the vector functions of not correctly aligned memory. Furthermore, as we stored the rows of the matrices contiguously in memory, we had to increase the size of the rows so as to ensure the correct alignment of every first entry of each row.

The strategy we chose to deal with row dimensions that are not divisible by the width of the vector operations was straight forward: compute as much of the row through vector intrinsics as possible, then compute the rest sequentially.

3.2 Version using Array of Structs data structure

While the strategy to deal with unfit row dimensions remained the same in the improved version of our program, we had to make several other major adjustments to be able to use vectorization in conjunction with the more cache-efficient data structure.

As Figure 2 shows, using the Array-of-Structs way allocating the matrices results in non-contiguous storing of the entries of the rows. Consequently the standard way of loading and storing values using the vector data types cannot be used here. As a workaround we implemented load and store operations specifically tailored to our data structure using vector functions that load and store single double values. An upside of this approach was that correct data alignment was no longer required. However we believe that this approach also carries significant performance overhead over the classically used operations.

3.3 Thoughts on wavefront vectorization

As the dynamic programming step of the TKF91 algorithm has data dependencies to the cells left of, above and diagonally left above of the cell to be computed, parallelization would require a *wavefront* approach.

In wavefront parallelism, matrix entries are not computed by row or by column, but rather by computing the entries of each *counter-diagonal*. Crucially, the counter-diagonals have to be computed sequentially and in-order when dependencies exist between a counter-diagonal and its previous neighbor. In this case, the current counter-diagonal is called the wavefront, giving this paradigm its name.

As the log-space transformed version of TKF91 reduces the operations that would require such a wavefront approach for parallelization to finding the greatest of two to three values, the relative amount of time spent doing so represents a negligible part of the overall algorithm. Nevertheless, we invested some time attempting a wavefront-vectorized version of the program.

Primarily, we attempted to find a suitable way of storing and accessing the cells of the matrices, such that data would be contiguous in the just described wavefront pattern rather than using row-major order. However we concluded that merely accessing a cell, using the usual row-, and column coordinates, would require significant computation by itself. Most importantly, it would

Logarithm Function	Average Edit Distance
`log` from `<math.h>`	2.15
`log_ru` from `crlibm`	33.15

Table 1: Average edit distances from the alignment returned by using `Boost.Multiprecision` of different logarithm implementations.

require significantly more operations than the actual computations done for each matrix entry.

Compounding to this realization was, that our preliminary benchmarks had shown absolutely no increase in performance using vectorization of either major implementation. Consequently we abandoned wavefront parallelization for this algorithm.

4 Benchmarking Results

We conducted all measurements on a desktop machine with 16 GB RAM and a Intel$^{\text{TM}}$ i7-2600 CPU with 4 physical cores and hyper-threading.

4.1 Different Logarithm Libraries

Additionally to using the `log` function from the standard C++ header `<math.h>`, we also used the `crlibm` library by Daramy et al. [3]. It includes versions of the natural logarithm computation, which allows the type of rounding to be explicitly specified.

In order to check the correctness of our resulting alignments, we computed the edit distance between the alignments we obtained by using different logarithm implementations. We assigned a cost of 1 to insertions, deletions and substitutions of single letters. We picked all pairs of sequences from the `BDNF_unaligned_sequences.fas` data set from `http://www.uni-oldenburg.de/fileadmin/user_upload/biologie/ag/systematik/download/Programs/benchMark_data.tar.gz`.

For the remaining parameters, we used $\lambda = 1, \mu = 2, \pi = (0.27, 0.24, 0.26, 0.23)$ and $\tau = 0.1$. The edit distances were computed in relation to using the logarithm function from `Boost.Multiprecision`.

In particular the `log_ru` function, which explicitly rounds up, produces a notable change in alignment compared to the logarithm function from `Boost.Multiprecision` as can be seen in Table 1. The computed likelihood scores were highly similar and the differences in the alignments were minimal as can be seen in Figure 6. We finally decided to use the `log_ru` function from the `crlibm` library because it had the most consent with the alignments returned by the reference implementation (`http://sco.h-its.org/exelixis/web/teaching/practical15/scaledCode/tkf91_scaling.tar.gz`).

Alignment using standard C++ header `<math.h>`
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATT---CTG-CTG-CAGTGG--T-A
```

Alignment using `Boost.Multiprecision`
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATT---CTG-CTG-CAGTGG--T-A
```

Alignment using `log_ru`
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATTC---TG-CTG-CAGTGG--T-A
```

Alignment from reference implementation
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATTC---TG-CTG-CAGTGG--T-A
```

Figure 6: Difference in alignments using the input parameters $\pi = (0.25, 0.25, 0.25, 0.25), \lambda = 1, \mu = 2, \tau = 0.1$

Length of Sequence	Number of Runs
10	10000
100	1000
1000	100
10000	10

Table 2: Number of runs per sequence length

4.2 Runtime Measurements

The input sequences used to measure execution time were four pairs of randomly generated sequences of length 10, 100, 1000 and 10000 nucleotides. Sequences did not contain ambiguous characters.

We measured the execution times around the kernel of the program, that is excluding any I/O required to load parameters and input sequences, or output of the algorithm. Kernel execution was performed multiple times per input and subsequently averaged. The number of runs executed was dependent on the size of the input, so as to balance accuracy and overall benchmark time (see Table 2).

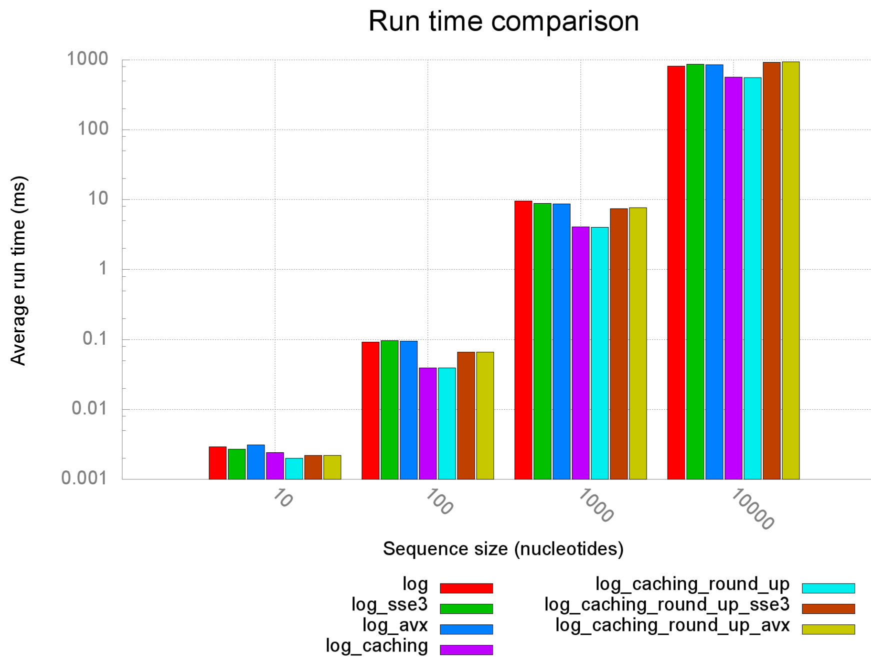

Figure 7: Comparison of average run times of the different implementations. `sse3` and `avx` post-fixes denote the vectorized versions of the programs, using SSE3 and AVX vector intrinsics respectively (Section 3). `log` denotes, that a version built on the log-space transformed version of TKF91 (Section 1). Versions using the Array-of-Structs method of storing matrices (Section 2) are denoted by the `caching`-keyword, and the `round_up`-keyword denotes the use of the `crlibm` library logarithm function `log_ru` (Section 4.1).

5 Future Directions

Compensated Summation Since the addition of the logarithms remains our main source of numerical error, one could try to use a compensated summation algorithm like the Kahan summation algorithm [4] instead of naive summation. The advantage of the Kahan summation algorithm is that it reduces the numerical error of a summation while not depending on special data types. Moreover, it does not add much computational overhead and is easy to implement.

Indexing by Diagonals As explained at the end of Section 2, we did not find an easy way to compute the wavefront indices of neighboring matrix entries. Another possible approach to do wavefront indexing would be to use a data structure that stores each wavefront diagonal in a separate array, as depicted in Figure 8. One could then reorganize our Array-of-Structs data structure using this approach instead of the row-major indexing.

In order to access the left and upper neighbor of a `MatrixEntry`, we need to do a case distinction: For the first $\max\{n+1, m+1\}$ diagonals, depicted in green in Figure 8, it holds that the left neighbor of a `MatrixEntry` at position i in the array for the kth wavefront diagonal is placed at position i in the array for the $(k-1)$th diagonal. The upper neighbor is at position $i-1$ in the array for the $(k-1)$th diagonal. For the remaining $\min\{n, m\}$ diagonals, the left neighbor of a `MatrixEntry` at position i in the array for the kth wavefront diagonal is placed at position $i+1$ in the array for the $(k-1)$th diagonal. The upper neighbor is at position i in the array for the $(k-1)$th diagonal.

While accessing the upper and left element becomes straight-forward this way, finding the correct way to access the upper-left diagonal element seems tricky on the first thought. But since we can go up and then left instead of going upper-left directly, the formula for the upper-left neighbor follows immediately.

Since we already store each wavefront diagonal separately in this storage approach, we expect it to be a good solution in order to make future wavefront vectorization attempts of the TKF algorithm easier and faster. We also hope to improve data locality by using this approach instead of the row-major indexing.

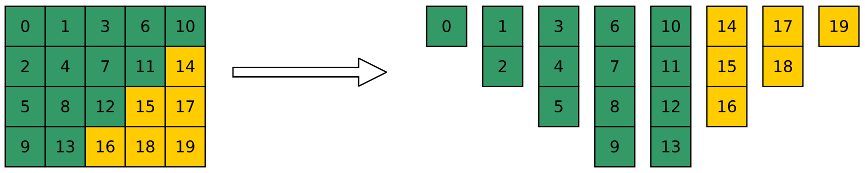

Figure 8: Storing each wavefront diagonal in a separate array. The first $\max\{n+1, m+1\}$ diagonals are drawn in green color.

References

[1] Boost Multiprecision. `http://www.boost.org/doc/libs/1_59_0/libs/multiprecision/`. Accessed 08.09.2015.

[2] How to deal with underflow in scientific computing. `http://stackoverflow.com/questions/9336701/how-to-deal-with-underflow-in-scientific-computing/9342513#9342513`. Accessed 08.09.2015.

[3] Catherine Daramy, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, and Jean-Michel Muller. Cr-libm – a library of correctly rounded elementary functions in double-precision, 2004.

[4] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965.

[5] JeffreyL. Thorne, Hirohisa Kishino, and Joseph Felsenstein. An evolutionary model for maximum likelihood alignment of dna sequences. *Journal of Molecular Evolution*, 33(2):114–124, 1991.

A Formulas

In the following formulas, $1 \leq i \leq n$ and $1 \leq j \leq m$, if not specifically denoted otherwise.

A.1 Initialization of the Matrices

$$\begin{aligned}
M^0(0,0) &= -\infty \\
M^1(0,0) &= \log(\gamma_0) + \log(\zeta_1) = \log(1 - \frac{\lambda}{\mu}) + \log(1 - \lambda * \beta) \\
M^2(0,0) &= -\infty \\
M^0(1,0) &= \log(\gamma_1) + \log(\zeta_1) + \log(\bar{p_0}) + \log(\pi_{a_1}) \\
&= \log(1 - \frac{\lambda}{\mu}) + \log(\lambda) + \log(1 - \lambda * \beta) + \log(\beta) + \log(\pi_{a_1}) \\
M^0(i,0) &= M^0(i-1,0) + 2 * \log(\lambda) + 2 * \log(\beta) + \log(\pi_{a_i}), i \geq 2 \\
M^1(i,0) &= -\infty \\
M^2(i,0) &= -\infty \\
M^0(0,j) &= -\infty \\
M^1(0,j) &= -\infty \\
M^2(0,1) &= \log(\gamma_0) + \log(\zeta_2) + \log(\pi_{b_1}) \\
&= \log(1 - \frac{\lambda}{\mu}) + \log(1 - \lambda * \beta) + \log(\lambda) + \log(\beta) + \log(\pi_{b_1}) \\
M^2(0,j) &= M^2(0,i-1) + \log(\lambda) + \log(\beta) + \log(\pi_{b_i}), j \geq 2
\end{aligned}$$

A.2 Further Initialization

$$\begin{aligned}
M^0(i,j) &= \log(\lambda) + \log(\beta) + \log(\pi_{a_i}) \\
M^1(i,j) &= \log(\lambda) - \log(\mu) + \log(\pi_{a_i}) + \log(\max\{P_{a_i \to b_j} * \bar{p_1}, \pi_{b_j} * \bar{p_1}\}) \\
&= \log(\lambda) - \log(\mu) + \log(\pi_{a_i}) + \log(1 - \lambda * \beta) + \max \begin{cases} \log(P_{a_i \to b_j}) - \mu * t, \\ \log(\pi_{b_j}) + \log(1 - e^{-\mu * t} - \mu * \beta) \end{cases} \\
M^2(i,j) &= \log(\lambda) + \log(\beta) + \log(\pi_{b_j})
\end{aligned}$$

A.3 Dynamic Programming Step

$$
\begin{aligned}
&M^0(i,j) = M^0(i,j) + \max\{M^0(i-1,j), M^1(i-1,j), M^2(i-1,j)\} \\
&M^1(i,j) = M^1(i,j) + \max\{M^0(i-1,j-1), M^1(i-1,j-1), M^2(i-1,j-1)\} \\
&M^2(i,j) = M^2(i,j) + \max\{M^1(i,j-1), M^2(i,j-1)\}
\end{aligned}
$$