Advanced Algorithms in Bioinformatics

Exercise 1: Read mapping with semi global alignment

Group 5: N. Güttler, K. Liebers, F. Mattes

18. Mai 2012

1 Implementation

The computation of a semi global alignment with dynamic programming is the same as to perform the *Smith-Waterman* algorithm, with the difference, that the initial gap costs for one of the sequences are not equal to zero. Moreover it computes the minimum instead of the maximum for each cell.

1.1 Smith-Waterman

We used a Smith-Waterman implementation very similar to the pseudo code found in the lecture script. One Column of the DP-matrix is stored as an integer array of length m (the length of the read) at any time during the algorithm. Two additional single integer variables (Cn and Cp) are used to store temporary values for the score minimizing function.
The memory consumption for this implementation is in $O(m)$ and it proved to be considerably faster than any previous attempts using *vector* or *list* based data structures for the matrix.
One disadvantage of this implementation is the lack of a complete matrix for backtracking since only one column is saved. We solved this problem by simply using the same algorithm backwards from the known end position of every confirmed hit from the forward run. For every hit we had to compute an additional $m \cdot (m + k)$ big DP-matrix considering the worst case of a hit with k errors having k gaps introduced in the *read* sequence. We did not save the backtracking trace since we are only interested in the start position of a hit and not in the actual alignment.
Since we filter suboptimal hits (caused by the 'shifting' read sequence across the optimal site) before this step, the computational overhead for the additional matrices is negligible if k is not chosen too high.

1.2 Ukkonen Trick

The Ukkonen trick was implemented exactly like in the script pseudo code by only computing the current column up to a 'last active cell' which is updated for every new column in constant amortized time. Since the last active cell is defined by the last score in the column that is better than or equal to k, the algorithms complexity changes from $O(n \cdot m)$ to $O(n \cdot k)$.
Our results confirm a large decrease in run time with the Ukkonen trick - compared to the unmodified Smith-Waterman algorithm.

2 Results/Observations

We expect runtimes in order of $\mathcal{O}(mn)$ (classical approach) and $\mathcal{O}(kn)$ (Ukkonen's algorithm) with n = text length, m = pattern length and k = number of allowed errors. In fact, the runtime is reduced extremely, if the program uses the Ukkonen's trick.

With $k = 0$ and filtering the results, i.e., suboptimal alignments were ignored, following runtimes were noted:

Reads file's name	Ukkonen trick?	Running time [sec]		Nr. of occurrences
		exercise1.cpp	Razers	
50_100	no	290.39	11.79	31
	yes	13.60		
50_1k	no	2915.29	12.42	289
	yes	136.47		
100_100	no	584.70	11.95	16
	yes	13.79		
100_1k	no	5840.71	13.23	189
	yes	136.68		
400_100	no	2278.69	12.24	11
	yes	15.66		
400_1k	no	23179.58	16.19	54
	yes	137.23		

Program 'exercise1' was executed on the linux machine *andorra*[1]. The *Razers*[2] program ran on a windows machine, since the supplied binary file (*razers3*) could not get started under linux.

The values of the table above can be interpreted as a lower bound for the running time of the respective data sets. With increasing k the complexity will obviously increase, too - since more occurrences will be found than in the case of perfect matching ($k = 0$). Thus more DP-matrices as well as more cell values within the *Ukkonen* algorithm need to be calculated.
E.g. 28.89s and 52 occurrences, 42.35s and 79 or 51.55s and 108 occurrences were observed by running the first data set with *Ukkonnen* trick and $k \in \{1, 2, 3\}$ respectively (to mention some of these expected increases).

Razers was always faster than our cpp-program. Even if *Razers* calculates more start positions: With default values it computes the start positions for reads up to a so called 'percent identity threshold' of 92. That means the allowed number of errors varies depending on the read's length. Using the last version and/or evoking the program with additional options, so that only the calculation for a given k are effectuated, would allow the program to run even faster.

Regarding the accuracy of the program's calculations, we could establish that the number of and the occurrences themselves were the same in all cases. This was observed by filtering the reads that match perfect ($k = 0$), i.e., considering only the reads with entries equal to 100 in the last column.

[1]andorra.imp.fu-berlin.de
[2]Version: 'RazerS_20100618', called without any options