Title: Large-scale pattern matching - Read mapping

Area: Bioinformatics, String Algorithms, Algorithmic Engineering

Supervisor: Thomas Mailund <mailund@birc.au.dk>

When sequencing the genome of an organism an important step is to map (match) a very large number of reads (small strings of ~50 characters) to a reference genome (a much larger string of millions of characters). The aim is to find out where the reads match with few errors, e.g. match within a short edit- or Hamming-distance.

There are many algorithms and data structures to explore ranging from simple exact pattern matching algorithm like Knuth-Morris-Pratt and Boyer-Moore to more advanced methods based on hashing or data structures such suffix trees, suffix arrays, and the Burrow-Wheeler transform (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3375638/). There are (of course) also a lot of existing read mapping tools such, e.g. BWA (http://bio-bwa.sourceforge.net and http://sfg.stanford.edu/mapping.html).

The aim of this project is to implement a read mapper and evaluate its performance, in terms of quality, running time, and space consumption. This involves working with genomic data in standard formats such as FASTA, FASTQ and SAM, reading about and understanding algorithms and data structures for exact and inexact pattern matching from selected papers and books, and making efficient implementations of (a subset of) such algorithms and data structures in a programming language of your choice.

Finding tandem repeats in genomic data

Area: Bioinformatics, String Algorithms, Algorithmic Engineering

Supervisor: Thomas Mailund <mailund@birc.au.dk>

Various non-random patterns in genomic DNA are usually indicators of biological function. Searching for patterns in genomes, that are often billions of characters long, is a non-trivial task, however, and requires efficient data structures and algorithms. This project aims at locating all so-called *tandem repeats* in a string, that is, strings that are repeated next to each other in the genome (https://en.wikipedia.org/wiki/Tandem_repeat).

A suffix tree (https://en.wikipedia.org/wiki/Suffix_tree) is an efficient index structure for strings that enable many pattern matching algorithms. A straightforward construction takes $O(n^2)$, which obviously isn't a feasible solution if we need to search in strings that are hundreds of millions of characters long, but they can also be constructed in O(n) using e.g. McCreight's algorithm (https://dl.acm.org/doi/abs/10.1145/321941.321946). Once we have a suffix tree, we can efficiently search for tandem repeats using e.g. Stoye and Gusfield's algorithm (https://www.sciencedirect.com/science/article/pii/S0304397501001219), locating all tandem repeats in O(n log n + z), where n is the length of the string and z is the number of tandem repeats identified.

The project involves working with genomic data in standard formats, reading about and understanding relevant algorithms and data structures, and making efficient implementations of (a subset of) such algorithms and data structures in a programming language of your choice.

Title: Building phylogenetic trees using neighbor-joining

Area: Bioinformatics, Clustering, Algorithmic Engineering

Supervisor: Christian Storm Pedersen <cstorm@birc.au.dk>

Inferring the evolutionary relationships between a set of organisms is an important step in many biological or medical workflows. It is often referred to as building a phylogenetic tree and is often done by clustering the organisms according to estimates of their pairwise relationships. The neighbor-joining (NJ) method is a widely used method for constructing useful phylogenetic trees. Using the canonical NJ method by Saitou and Nei (https://academic.oup.com/mbe/article/4/4/406/1029664), it takes $O(n^3)$ time to build a tree of n organisms if their pairwise relationships (distances) are known. There are many variations of neighbor-joining and many heuristics for speeding it up. RapidNJ (http://birc.au.dk/software/rapidnj/) is one such heuristic.

The aim of this project is to describe, implement and experiment with methods for building phylogenetic trees. More specifically, to understand and describe approaches such as RapidNJ and examine how their running time in practice compares to canonical neighbor-joining. This involves reading about and understanding algorithms for construction of phylogenetic trees using NJ methods from selected papers and books, making efficient implementations of (a subset of) such algorithms in a programming language of your choice, and making experimental comparisons of the properties of the implemented algorithms using algorithms using suitable test data, e.g. simulated genomic data in standard formats such as FASTA format for sequences, Phylip format for distance matrices, and Newick format for trees.

Title: Comparison of multiple biological sequences

Area: Bioinformatics, String Algorithms, Algorithmic Engineering

Supervisor: Christian Storm Pedersen <cstorm@birc.au.dk>

Biological sequences (DNA, RNA, and proteins) can be modeled as strings over finite alphabets. The evolutionary relatedness of species can be inferred by comparing their biological sequences. Many algorithms exist for this purpose, and typically involves computing an alignment of the two or more strings. Computing a so-called multiple sequence alignment (MSA) is an important part of many bioinformatics analyses (https://en.wikipedia.org/wiki/Multiple_sequence_alignment), but in most scenarios it is a computational hard problem, and many heuristics and approximation algorithms therefore exist.

The aim of this project is to describe, implement and experiment with algorithms for computing MSAs. More specifically, to understand and describe approximation algorithms and heuristics for computing MSAs that are based on combining optimal pairwise alignments in a certain order such as Gusfield's 2-approximation algorithm (https://link.springer.com/article/10.1007/BF02460299) and heuristics based on (minimum) spanning trees (e.g. https://ieeexplore.ieee.org/abstract/document/1527521). This involves reading about and understanding algorithms for computing MSAs from selected papers and books, making efficient implementations of (a subset of) such algorithms in a programming language of your choice, and making experimental comparisons of the properties of the implemented algorithms using suitable test data, e.g. simulated genomic data in standard formats such as FASTA format for sequences and Phylip format for distance matrices.