COMPSCI369 - S1 2020

Assignment 3

Due date: 22:00 26 May 2020

0 Instructions

This assignment is worth 7.5% of the final grade. It is marked out of 75 points.
Provide a solution as a Python notebook that includes well documented code with the code/test calls to reproduce your results. Include markdown cells with explanation of the results for each question. Submit to Canvas

- the `.ipynb` file with outputs from the executed code
- a `.html` version of the notebook with all outputs of executed code showing. (To get this format, export from the notebook viewer or use `nbconvert`.)

Within the notebook, set the random seed to some integer of your choosing (using `random.seed`) so that the marker can recreate the same output that you get. You can reset the seed before each question if you like.

1 Genome Assembly

Due to limitations of the modern sequencing technology, we can only obtain short continuous sequence fragments (or reads) from the genome sequences at a time. Given millions of such short reads, can we reconstruct the full DNA sequences as close as possible? This is essentially the *DNA sequence assembly* problem, that can be approximated to the *Shortest Common Superstring (SCS)* problem: *Given a set of strings s_1, $s_2, \ldots, s_n$, find a shortest string s that contains all of them as substrings.*

SCS problem is a NP-complete problem, so the early DNA sequences algorithms used a simple *greedy* strategy to the assembly problem, that find the SCS by iteratively joining and merging overlapping reads until all reads have been merged.

In this assignment you will implement the greedy sequence assembler. You will also implement the read generator, i.e simulate the *shotgun sequencing* process, that randomly partitions a genome sequence S into fragments of certain length l. The process is repeated a large number of times (also known as cloning) to amplify the biological signal, needed for proper identification of the short reads. In the lecture, you saw example of greedy DNA reconstruction given all the l-length reads defined over the original DNA sequence for a fixed value of l. Here, we will not assume that all possible reads of length l will be given (generated), and we will allow for reads with different length. But, we will keep the assumption that the reads will be error free.

Question 1: Shotgun Sequencing (10 Points)

Implement a class *ShotgunSequencing* with the following attributes and methods:

(a) *sequence*
stores a DNA sequence.

(b) *readSequence(filename, i=1)*
reads only the i-th sequence from the filename, a file in fasta format, and stores

the sequence in *sequence*. The parameter i can take only positive values; if the file does not have i sequences, then return None.

(c) *shotgun(l_min,l_max)*
returns the sequence *sequence* as an `array` of fragments with random length $len(f)$ that is uniformly distributed between *l_min* and *l_max*. Note that the last generated fragment can be shorter than *l_min*. You can safely assume that the fragments cover the whole sequence, and no fragment gets lost.
Example: S='ATTCGGT' could be fragmented as 'AT', 'TCGG', 'T' for $2 \leq len(f) \leq 4$.

(d) *cloning(l_min,l_max,n)*
calls the function $shotgun(min, max)$ n times and returns **one** sorted `array` with all fragments.

Question 2: Sequence Assembler (35 Points)
Implement a class *SequenceAssembler* which assembles the fragments from the above exercise, with the following attributes and methods:

(a) *graph*
stores the overlap graph.
Hint: Think about the below given methods which you have to implement before you decide on how to store your graph. You need to have easy access to all in-coming and out-going edges of a node. You will need to remove nodes and their corresponding edges. Why are adjacency lists the best representation for graphs to use for this problem?

(b) *calculateOverlap(fragmentA,fragmentB)*
returns the length of the overlap if fragmentB follows fragmentA.
Example: *calculateOverlap('AAT','ATG')* should return 2, while *calculateOverlap('ATG', 'AAT')* should return 0.

(c) *createOverlapGraph(fragments)*
creates an overlap graph where each fragment is considered as node. Edges are directed and weighted, with the weights representing length of fragment overlaps. Keep edges that have non-zero weights. Nodes representing fragments that are substrings of other fragments (nodes) can be removed from the graph.

(d) *getMaxEdge()*
returns the edge with highest weight.

(e) *assembling()*
implements the greedy algorithm for Shortest Common Superstring where you merge the nodes until one is left and return the final sequence.

Test your solution for different combinations of *l_min*, *l_max* and n

Sequence	*l_min*	*l_max*	*n*
Test	5	10	3,5,10
	10	20	
	20	30	
Real mRNA	50	100	5,10,15
	100	200	
	200	500	

and interpret the result for the reconstruction of the two sequences (short-length test sequence, and real-length mRNA from the human genome) provided in the file `A3_DNAs.fasta`. When interpreting the results, take into consideration the following questions:

- How well is the sequence reconstructed as the fragments vary in size?
- Does the reconstructed sequence have a similar length to the original sequence?
- What is the effect of cloning on the two sequences?

2 Stochastic simulation

A standard model in epidemiology is the SIR model of infectious disease spread. It has a population of N hosts is divided into 3 *compartments*, so is known as a compartmental model:

- the S compartment of those who are susceptible to the disease
- the I compartment of those who are infectious with the disease
- the R compartment of those who are recovered from the disease and now immune (or, more generally, those who are removed from the epidemic either through recovery with immunity or due to death).

We assume that $S + I + R = N$.
The model can be thought of as deterministic or stochastic. We consider the stochastic version here. Times between all events are exponentially distributed with the following rates which depend on the current state of the outbreak, assumed to be (S, I, R):

- the rate of transmissions is βSI and the new state is $(S - 1, I + 1, R)$, and
- the rate of recoveries is γI and the new state is $(S, I - 1, R + 1)$.

Question 3: Simulating outbreaks (30 Points)

(a) At what point will the epidemic finish?

(b) Write method `rand_exp` that takes a rate parameter λ as input and produces as output an exponentially distributed random variable with rate parameter λ.

(c) Write method `sim_SIR` that takes as inputs N, S_0, β, γ and produces as output a list of the event times and the number susceptible, infected and recovered at each time point. All outbreaks start at time $t = 0$.

(d) Run a simulation with $N = 1000, S_0 = 10, \beta = 2.2, \gamma = 2$ and plot the number infected through time.

(e) Run an experiment and report the results to approximate the probability that a large outbreak occurs using the same parameters as above but with only one initial infected. What has usually happened if there is no outbreak?

(f) The reproduction number $R_0 = \beta/\gamma$ of the epidemic is the mean number of transmissions by a single infected in an otherwise susceptible population. Using the same parameters as above but allowing β to vary, select five values of R_0 near to or at 1 and explore whether or not you get an outbreak. Report and explain your results.

(g) Suppose now that the infectious period is fixed, so that hosts are infectious for exactly 1 time unit. Is the process still Markov? How would you go about writing code to simulate such an epidemic? (You do not have to actually write the code here.)