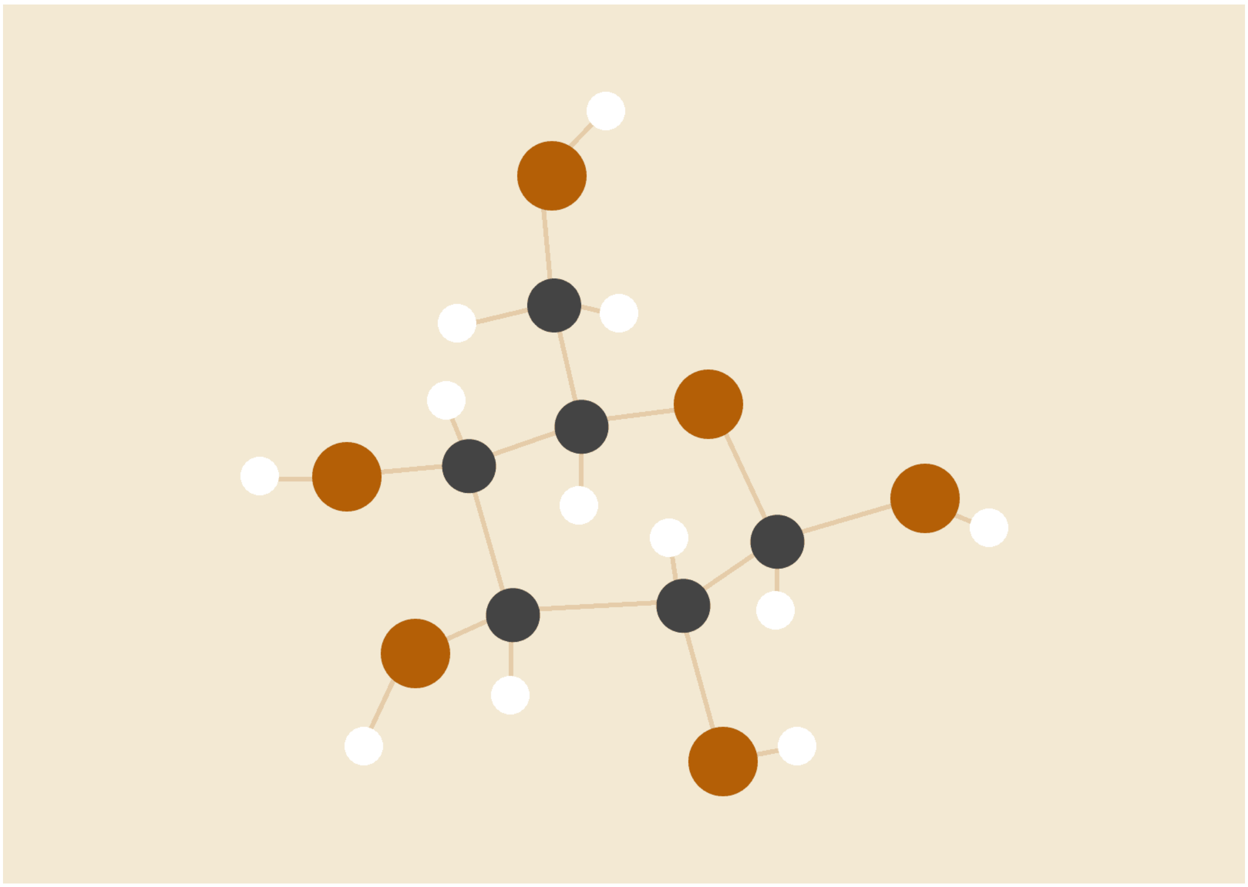
Bacteria population dynamics *with GraphX*

Software for Big Data



Claudio Ciano

Raffaele Lacatena

Emanuele Varriale

Introduction

The goal of this project was to build a simulation framework to study the population dynamics of different strains of bacteria. These strains live inside the vertices of a graph, where they can reproduce, die, and travel along the edges connecting the vertices. All of these actions are chosen according to probabilistic laws.

We build the graph using the GraphX library as a graph processing system and simulate its dynamics with its Map/Reduce implementation through message passing. We show a few interesting results about the population dynamics.

Why GraphX?

We choose to use GraphX because it is optimized for running graph algorithms in a highly parallel fashion. We can define the behaviour of a vertex and let the framework run the code for every vertex in the graph.

Let’s take a step back to frame GraphX within the Big Data world.

Spark and RDDs

GraphX is part of the Apache Spark framework, that, thanks to its in-memory data processing, can be much faster (10-100 times) than Hadoop MapReduce, since the intermediate reads and writes on disk are avoided. Nonetheless, Spark can run on top of Hadoop and can read and write on its distributed file system, HDFS.

Spark’s main data structure is the RDD: *Resilient Distributed Dataset*. It is a collection of objects distributed across the computing nodes, that is usually stored in-memory, but can also be written to disk. It is resilient because the framework itself handles node failures, and rebuilds RDDs as needed.

The abstraction of RDDs, that are *immutable*, partitioned, logical collections of records, and the APIs to control them, allows more general programming models than the “simple” Map-Reduce and thanks to Spark’s native language, Scala, that is very concise and powerful, many different applications can be developed.

GraphX: main features

The purpose of GraphX is to allow a more intuitive way to handle graph data. Relational databases, with a table for vertices and one for edges, do not play well with graphs: if, for example, we want to find friends-of-friends-of-friends in a social network, we would have to join the edge table with itself many times, even if we are only interested in a single user. Each join would be an entire MapReduce job, with slow I/O operations.

GraphX also saves a graph with two RDDs, one for the vertices and one for the edges, but it is highly optimized for graph traversing (as opposed to a RDBMS). More specifically, GraphX works with property graphs, that can associate arbitrary attributes to each vertex or edge. In this way one can easily traverse the graph following the edges, but also transform vertices or edges in bulk.

aggregateMessages: MapReduce on graphs

The core of GraphX is the method `aggregateMessages`, which implements Map/Reduce for graphs. This method is reminiscent of the Bulk Synchronous Parallel execution model from Google Pregel, the first graph processing framework. In BSP each Map/Reduce operation consists of two phases, *local computation* and *communication*, which together form a *superstep*. Synchronization is achieved through barriers between phases and supersteps, i.e. waiting until every local computation is ended before communicating and vice versa.

GraphX is vertex-centric, and therefore in Map/Reduce operations each vertex works as a logical unit that aggregates information from its neighbours. More specifically, the graph method `aggregateMessages` takes as parameters two user defined functions, `sendMsg` and `mergeMsg`, which are essentially the map and the reduce functions.

`sendMsg` takes as parameter an `EdgeContext`, a GraphX object that allows reading and sending data from/to the vertices at both ends of an edge. In this function one can then define the local computation and what information to propagate. Every message is then sent (mapped) to the appropriate vertex and the function `mergeMsg` is in charge of aggregating these messages in each vertex.

`aggregateMessages` finally returns an `RDD` containing tuples with `VertexID` and the result of `mergeMsg` for each vertex, concluding the Map/Reduce operation. How we use `aggregateMessages` will be the central part of our program, but first let's review our simple biological model of bacteria population dynamics.

The biological model

Heterogeneity in bacteria populations

Typically, epidemiologic classification of various microscopic organisms (bacteria included) is based on a number of factors including morphology/structure, mode of reproduction as well as ecology, among others.

Bacteria within a species display enormous phenotypic and genotypic heterogeneity; in the following we will refer to those isolates as strains. This term is therefore used as a synonym for variant or subtype.

Typically, a strain can be isolated in order to grow a clonal cell colony, where bacteria belong to the same variant.

New strains can be created due to mutation or “swapping” of genetic components, according to the specific behaviour of the organism in study.

Pictorial representation different strains of a selected species

Strains can be distinguished from each other by DNA sequence variations present in some housekeeping gene fragments located around the bacterial genome (for more information about sampling techniques in molecular biology check MLST assay [6]) and by other methods we will not deepen in this text.

For our purpose all we care about is to distinguish among strains as simply as possible: our population is a vector whose components represent the population sizes of different strains.

Why do we care about bacteria strains and their evolution in time?

For the sake of clarity let us define the following terms :

- **Holobiont**: the host plus all of its symbiotic microbes
- **Hologenome**: the genome of the holobiont
- **Supragenome**: core genome plus the distinctive genes of every strain

As a matter of fact we know from biology that biodiversity within a species plays a key role in many situations:

- Disease-state hologenomes will often display reduced complexity (for example, Clostridium difficile overgrowth in the intestine following antibiotic treatment [1] or a reduced gut microflora associated with patients with inflammatory bowel disease [2])

- Contaminated soil has been shown to have reduced microbial complexity [3]
- The variability within a species can be very significant, so that the supragenome is several times larger than the core genome, which is common to every strain. When characterizing (phenotyping) the diseases caused by independent isolates of non typeable Haemophilus influenzae [4] or Streptococcus pneumoniae [5], one finds a wide spectrum of possibilities ranging from localized chronic infections to sudden death.

In the following we'll use Gini Index as measure for hologenome complexity, defined as:

$$H = 1 - \sum_{i=1}^{n_s} f_i^2$$

where n_s is the number of different strains and f_i the relative population size of the *i*-th strain. When only one strain is present the heterogeneity is minimum, because $f_{i*} = 1$ and $f_{j\neq i*} = 0$, so that $H = 0$. The index is instead maximum when every strain has the same population size and $H = \frac{n_s - 1}{n_s}$.

Modelling of fitness, reproduction and interaction

A key concept in population genetics is the fitness of an individual/clonal group in a given environment. It describes individual reproductive success, i.e. the ability, based on heritable physical traits, to survive and reproduce on average.

In our model, from the total of N_i bacteria of the *i-th* strain, R_i individuals are chosen to reproduce (each producing one offspring with probability r_i) and D_i are chosen to die (with probability d_i) according to a binomial distribution which depends on fitness through its moments:

$$\Delta N_i = R_i - D_i$$
$$R_i \sim B(N_i, r_i)$$
$$D_i \sim B(N_i, d_i)$$

Fitness can be either fixed or dynamic according to whether or not an interaction term is present. Our model in fact accommodates also interactions among strains; this is accomplished by making d_i and r_i functions of other strains, e.g. $d_i = f_i(N_j)$.

The nature of interaction goes from mutualism (in which both species benefit) to commensalism (where one party benefits and does no appreciable harm to the other) to parasitism (where one of the species benefits at the expense of the other).

Various examples of symbiotic relationships within ecosystems exist between each of the microbial strains and the host, and also among the members of each microbiome [0].

This gives rise to completely different dynamics which can model many other biological systems (not only bacteria).

As a matter of example, we will explore in the following a well known kind of interaction in ecological systems frequently used to describe two species interacting, one as a predator and the other as prey: the Lotka-Volterra model.

Graph topology

Our bacteria live and reproduce on a connected 2D graph, whose topology we need to consider. Choosing a network pattern is one of the most important decisions for our simulation in order to correctly represent our system.

Different choices can be supported by scientific literature: nodes may interact directly with first neighbours or with distant nodes; this is common in many host contact network interaction models.

The latter case has been deeply explored in literature (see [7] for more details) and it may be helpful in modelling nosocomial (hospital) contaminations where the movement of patients, people or things may act as a transport vector.

Contact network in gut microbiota should have a direction of flow [8]; this behaviour may be modelled considering nodes as elements of an ordered set.

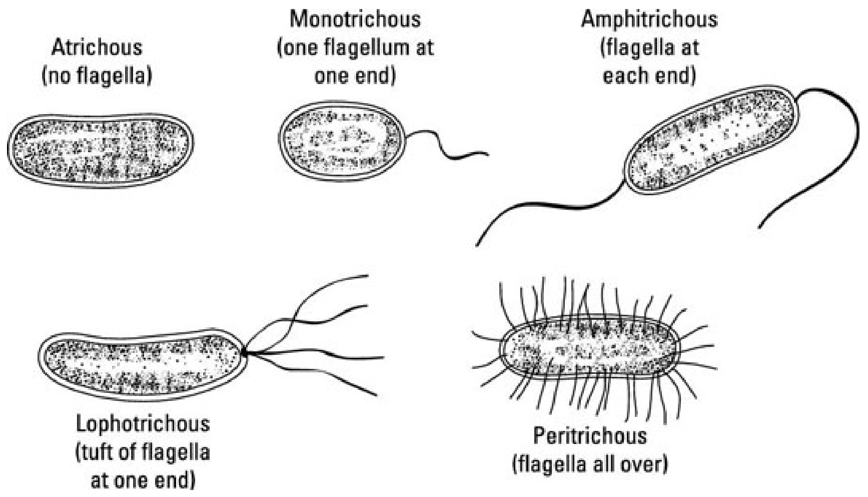
In the following we will work with a fully connected weighted graph, in which all nodes are linked together, and each edge is associated with a weight which depends on euclidean distance between nodes, so that longer edges are more difficult to cross.

Migration

Bacteria are motile organisms, they move around for several reasons and in many different ways. Some of them have *flagella* (lash-like appendage that protrudes from the cell body) which propels the cell forward and backwards

Bacteria without flagella can also move around by a type of motion called *gliding*. Gliding allows microorganisms to travel along the surface of low aqueous films, independently of propulsive structures such as flagella or pili. Although this kind of motion is very common among bacteria its dynamic is not yet fully understood.

Often they move toward or away from something based on the chemical gradient nearby in a process called *taxes*. Both chemical gradients (oxygen or sugar) and photogradients (light) may trigger the motion.



But how fast do they move?

Fascination with this topic is as old as the microscope itself. In 1683 Leeuwenhoek wrote to the Royal Society about his pioneering studies reporting some observations with his primitive microscope.

Today we know [8] that bacteria can reach speeds from $2\ \mu m/min$ (Halobacterium Halobium) up to$1000\ \mu m/s$ (Ovobacter Propellens).

Bacteria spread also due vector transportation; these latter might be inanimate (like droplets of water) or living hosts (most eukaryotic species are colonized by a microbial community).

All this information might provide us with some indications on how to choose topology and migration properties in order to reproduce realistic conditions.

Other assumptions and further development

Our attempt to build a flexible framework for biological simulation of bacteria only scratches the surface of what happens in a real environment.

When a bacterial cell divides, the two daughter cells are generally indistinguishable (and this is exactly our assumption through this work); thus, a single bacterial cell can produce a large population of identical cells or clones.

Occasionally, a spontaneous genetic change occurs in one of the cells. This change (mutation) is heritable and passed on to the progeny of the variant cell to produce a subclone with different characteristics from the original (wild type) parent.

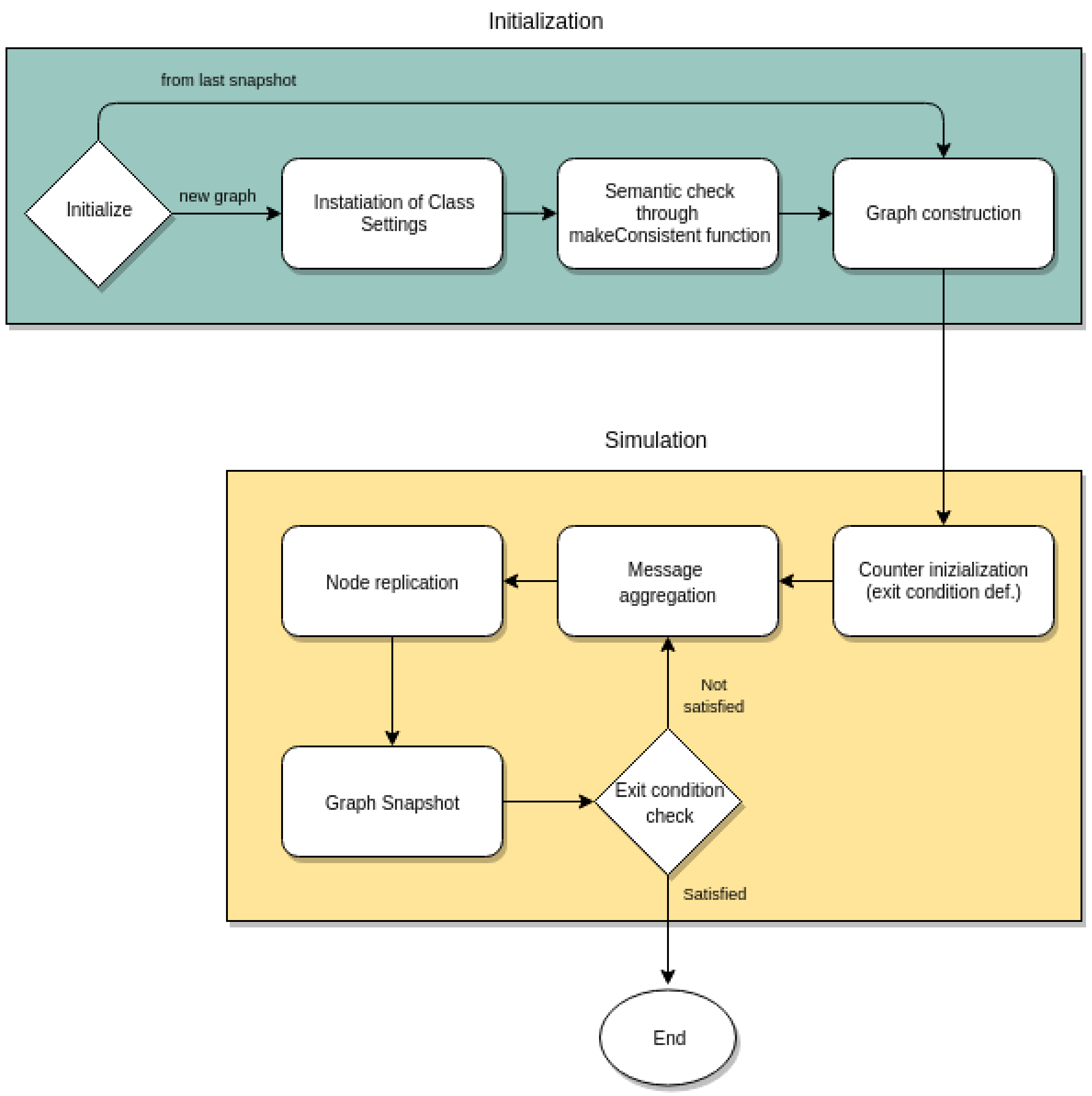
We will not include such mutation effects, which may be modelled as spontaneous conversion from one strain to another during replication. New strains could also arise and develop during the simulation with properties to be defined.

Other spontaneous events may occur (as gene exchange between bacteria, bacteriophages may influence strain dynamic and fitness [9])

The code

A high-level description

In the following, we give a high-level description of the code, highlighting the GraphX feature we have exploited. For a quick reference, see the following schematic representation:



Graph Initialization: settings

In order to start the simulation one needs to create an instance of the class **settings** where the fundamental parameters of the model are set up. Here we mention some of the most important classes and variables involved in this task:

Class **settings :**

transfP	migration probability among nodes
ceiling	threshold that prevents node population to diverge
halving	cut off distance for migration
pop	array of instances of class **Node**
strains	array of instances of class **Strain**
dimspace	specify the side length of a square which contains our graph
minVerticesdistance	minimum distance between Nodes selectable from `makeConsistent()` method
maxEdgeDistance	maximum distance between Nodes selectable from `makeConsistent()` method
range_popSize	range of values which getCasualNodes() method uses in order to set random population if needed
nIterations	number of iterations to simulate

Class **Node :**

distribution	array of integers whose components are strain population sizes
location	instance of class **Point** locates the Node in 2D space

Class **Strain :**

deathP	death probability of each bacteria per iteration
reprP	reproduction probability of each bacteria per iteration
deadlyInter	a dictionary containing the IDs of the strains interacting with the current one as keys, the intensity of the interaction as values. See `replication()` method for more details.
reproductiveInter	same as deadlyInter, but with positive interactions.

The **Strain** class is the only mandatory parameter we have to set up. Initialization procedures involve `makeConsistent()` and `getCasualNodes()` methods which tidy things up when node locations and strain distributions are missing from **settings** or are incorrectly set.

Node attributes are described by the class **properties**, whose most important variables are:

Class **properties :**

distribution	array filled with the population sizes of each defined **Strain**
buffer	each node fills this array during the message-passing phase. It contains incoming and outgoing bacteria.
edgeInfo	this dictionary stores information about traffic on each edge. Its purpose is to allow a proper initialization of the graph from through `getLastSnaphsot()` method.
nEdges	node out-degree.

Furthermore, edge attributes are described by the class EdgeProperties:

Class **EdgeProperties :**

distance	euclidean distance computed from location in setting.
traffic	stores traffic information cumulatively. This is needed in order to properly set the last state of the graph from a snapshot.

Graph Initialization: nodes and edges construction

Graph initialization is carried out through instantiation of **BGraph** class which creates a GraphX graph based on our settings.

Class **Bgraph :**

internalSettings	stores settings defined earlier.
graphx	build the GraphX graph from defined settings.

This class also contains some additional features we'll discuss in the following. These methods may be invoked at runtime in order to change graph dynamics.

We'll omit those we consider to be self-explanatory:

`addNode(node:Node): Bgraph`

`dropNode(id:Long): Bgraph`

`addEdge(id1:Long , id2:Long): Bgraph`

`dropEdge((id1:Long , id2:Long): Bgraph`

`addFullConnection(): Bgraph` returns a new fully connected graph

`dropAllEdges(): Bgraph` returns a new graph where all edges are dropped

Simulation steps

Each step of the simulation involves the whole graph (all the edge triplets) and is performed through message passing, aggregation and evolution of nodes.

At each step a new graph is created and its state is captured and saved as an object. This allows the user to initialize the simulation from a previous state. This particular task is carried out by invoking `getLastSnapshot()` method.

Map and reduce tasks underneath `aggregateMessages()` logic, involve several user defined methods that we summarize in the following:

`selectToMigrate()` and `selectFromDistribution()` fill source and destination buffers with values according to the following rules:

- Each population always migrates on all its outer edges.
- Each node selects a maximum number of migrants to send. This value is deterministic and depends upon the current population size divided by the node out-degree.
- The effective size of migrants for each destination is stochastically chosen according to a binomial distribution. The number of binomial extractions depends on the node's population, on the distance from the destination and on a user defined transfer probability which is chosen in settings (transfP).
- Each node is self connected (there's always a closed loop) and it sends itself a message in order to update its current state

`sendMsg()` sends serialized properties classes to both source and destination vertex, that are used in **`mergeMsg()`** to update the buffer that is finally summed to the distribution array in the node properties class inside the **`compute()`** function.

`replication()` function upgrades the current population according to the following rules:

- Death and reproduction probabilities are modified by the presence of an interaction term and by some balance factor which avoids population divergence, by reducing the reproduction probability and increasing death probability, as population size approaches the ceiling variable defined in settings.
- We draw the number of reproducing and dying bacteria from binomial distributions according to the probabilities computed before.

Data extraction and analysis

Graph dynamic produces data in runtime. These are collected in the following dictionaries, grouped together in a List:

- data.vertexPosition
    - Key : Node ID
    - Value: Node location (2Tuple)
- data.edgePosition
    - Key : (Source Node ID, Destination Node ID)
    - Value: (Source Node Location(2Tuple), Destination node location (2Tuple))
- data.vertexDynamic
    - Key : Node ID
    - Value : (Total population, Gini Index)
- data.vertexDistribution
    - Key: Node ID
    - Value: (Strain1 numerosity, Strain 2 numerosity, …) (n-Tuple)
- data.edgeDynamic
    - Key : (Source Node ID, Destination Node ID)
    - Value : History of traffic on this edge

We chose Python for data analysis.

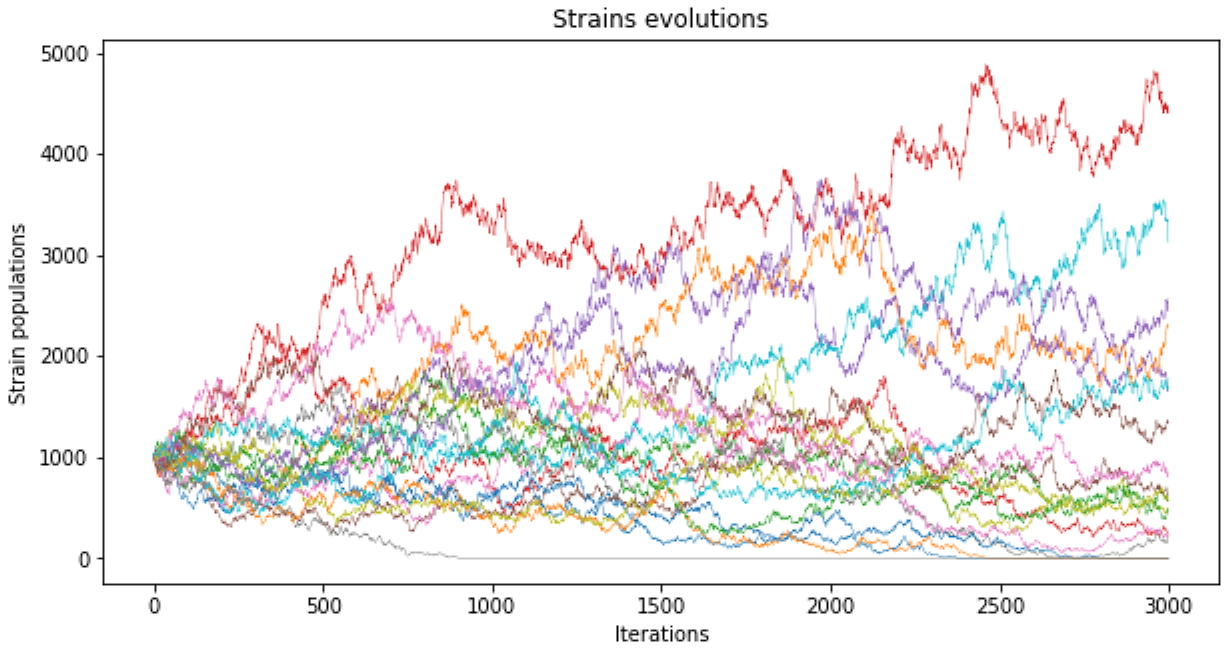
Results

Simulation 0: No interaction, no migrations

Our first simulation sets a background against which we'll observe special cases. In the following we'll assume no interactions among strains, nodes isolated from one another and neutral selection, i.e. equal reproduction and death rates. More precisely these parameters rules the evolution of the graph:

`maxTransf`	`0`
`transfP`	`0`
`deathP`	`0.5`
`reprP`	`0.5`
`deadlyInter`	`Map()`
`reproductiveInter`	`Map()`
`Node[distribution]`	`(1000,1000,1000,1000,1000)`

We observed the evolution of 10 independent nodes with 5 strains each:



As we can see each strain population behaves like a random walk with an absorbing barrier for $N_i = 0$.

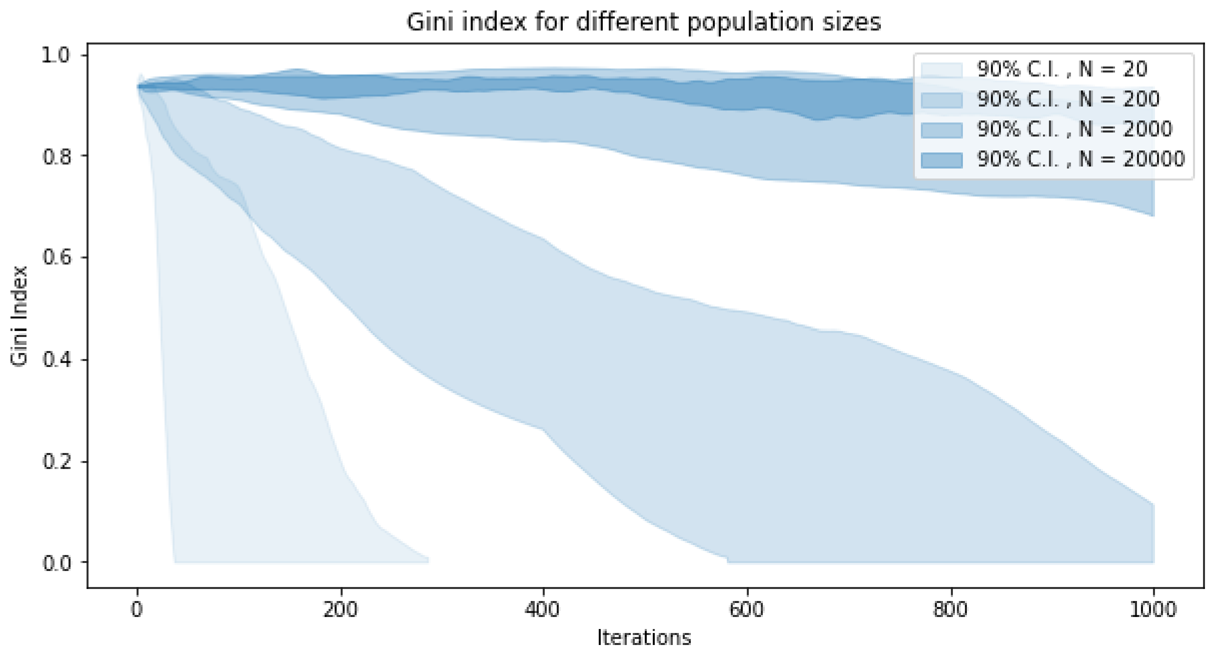
We can easily observe that the overall variance does increase over time (this kind of dispersion is one of the most important fingerprint for a random walk process).

When the i-th strain population size reaches zero-value by chance, it gets trapped there forever because there are no longer bacteria of that strain. This phenomenon may reproduce significant genetic drift (change in the frequency of an existing gene variant) in isolated environments, giving rise to clonal colonies where only one strain per node survives.

Clearly, the probability of a strain to become extinct depends on the total iteration time (as variance increases) and on the starting size of the population.

Let us define as clonality time the iteration for which only one strain remains in a node, or, equivalently, when the Gini index becomes 0 because there is no heterogeneity left.

In the following graph we plot the 10-90 % confidence intervals of the Gini index over time, as the initial population size N varies from 20 up to 20000.



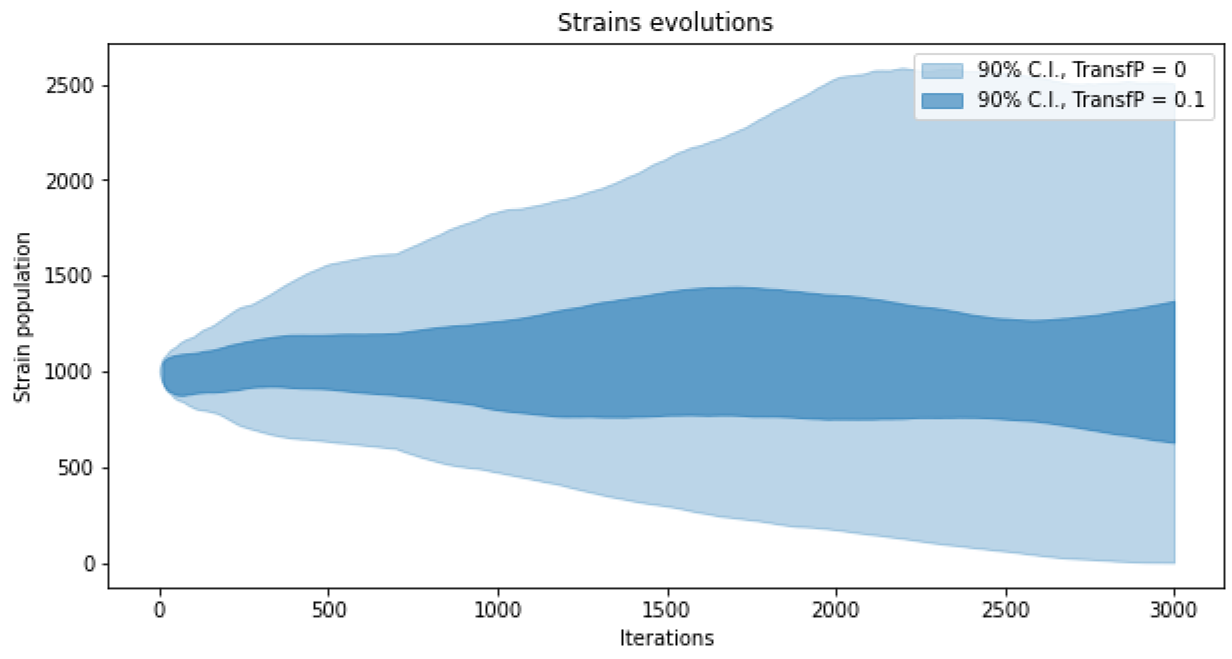
Here we can see that clonality time increases with the initial population size, as expected: for small populations there is a higher probability for one strain to hit the absorbing barrier of $N = 0$, due to random noise. When population size grows, the effect of the noise (which can be seen as the variance of the random walk) becomes more and more negligible ($\frac{N}{\sigma_N} \sim 1/\sqrt{N}$), so that, by the law of the large numbers, the actual variation in population at each time step becomes closer and closer to its expected value of 0.

Simulation 1: No interaction, migrations

Here we make a comparison between the results of Simulation 0 with another simulation, in which we allow bacteria migration over the edges of the graph. The parameters are the same as before except for the non-zero transfer probability, as can be seen in the following table:

maxTransf	0.5
transfP	0.1
deathP	0.5
reprP	0.5
deadlyInter	Map()
reproductiveInter	Map()
Node[distribution]	(1000,1000,1000,1000,1000)

To make the comparison we plot both the confidence intervals of population sizes from Simulation 0 (light blue) and Simulation 1 (blue) over time:



As we can see the possibility of migration stabilizes population sizes, as their variance increases less over time than in Simulation 0.

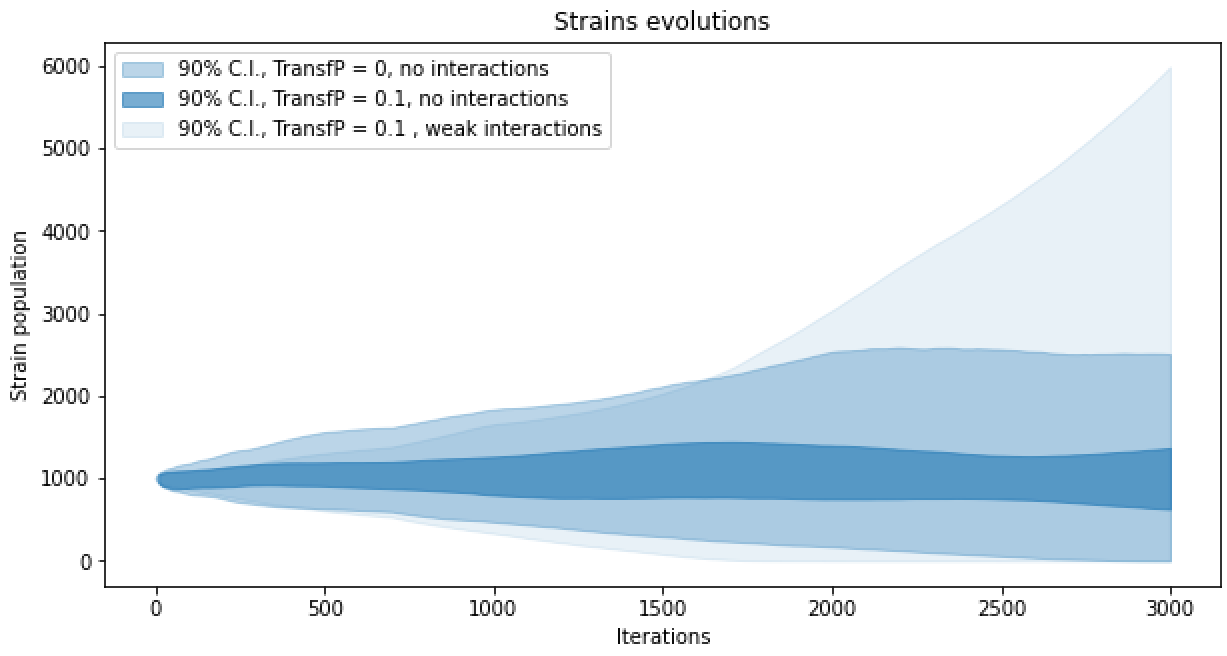
We can explain this behaviour with our implementation of bacteria migration: variance becomes more stable over time and this is due to the fact that the number of migrants per strain sent for each iteration depends on their relative abundance: when a strain is dominant in a node, more of its bacteria are going to migrate.

Simulation 2: Interaction, migrations

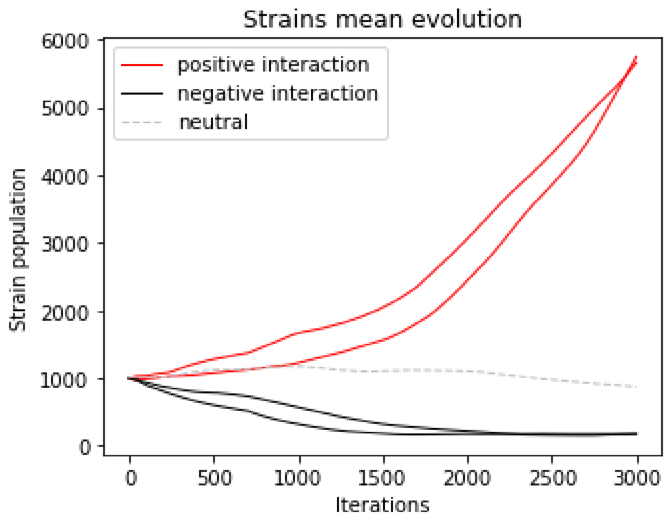
In the following simulation we'll merge results of Simulation 1 with another experiment in which we allow migration and weak interactions among strains.

Our settings provide as before five strains in which Strain 0 and Strain 1 experience mutualistic interaction while Strain 2 and Strain 3 compete. Strain 4 does not have any interaction. Our simulation settings is the following:

maxTransf	0.5
transfP	0.1
deathP	0.5
reprP	0.5
deadlyInter	(2->3,1e-2)(3->2,1e-2)
reproductiveInter	(0->1,2e-3)(1->0,2e-3)
Node[distribution]	(1000,1000,1000,1000,1000)



As expected the presence of a mutualistic interaction pushes some nodes towards more populated configurations.

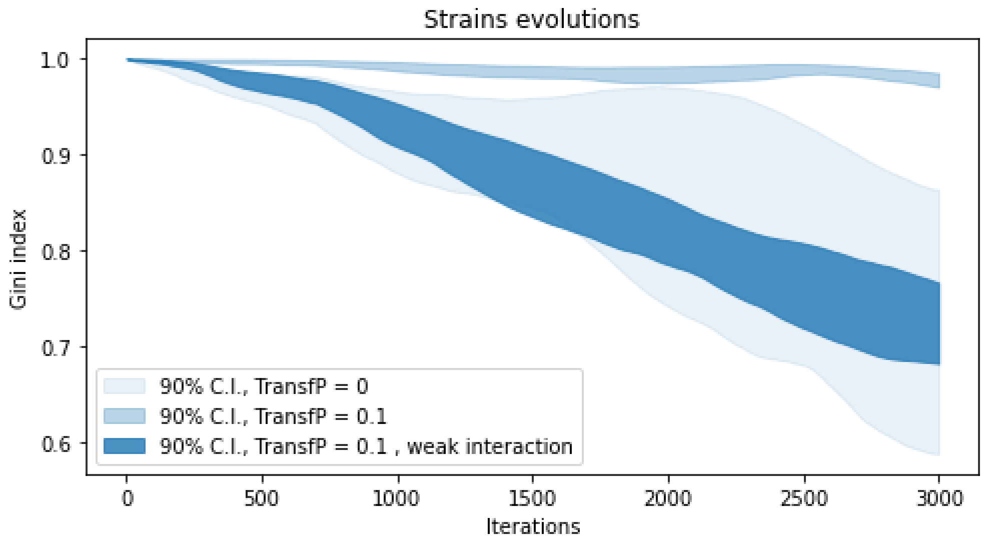


This behaviour may be better understood by plotting the mean evolution on the graph of each strain.

Mutualistic and competitive strains behave very differently in respect to the neutral one and this is evident from the first iterations.

Mutualistic strains become fitter with time as they help each other, whereas competitive strains' population gets smaller.

The presence of interactions among strains breaks the stability of the index as seen in Simulation 1. Mutualistic strains become eventually fitter than the competing ones but also fitter than the last strain, which does not interact.



Looking at the Gini index we recognize the narrow line on the top as being the effect of variance reduction due to migration explained earlier.

The interactions have pulled the system away from the neutral selection (where every fitness is the same), and therefore the first two strains eventually dominate and the heterogeneity index becomes smaller.

Simulation 3: Lotka-Volterra

We know from theory that in many biological systems in which species compete as prey and predators, the populations change through time according to a pair of first-order nonlinear differential equation:

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} =- \gamma y + \delta xy$$

Where x represents preys and y predators.

Let us suppose that two strains interact on our Node. As stated above strain variation per iteration is:

$$\Delta N_i = R_i - D_i$$

$$R_i \sim Bin(N_i, f(r_i, rInt_{i->j}))$$

$$D_i \sim Bin(N_i, f(d_i, dInt_{i->j}))$$

For simplicity: $rInt_{i->j} = rInt_{j->i}$ and $dInt_{i->j} = dInt_{j->i}$. We'll denote these probabilities simply as $rInt$ and $dInt$

If $f(r_i, rInt) \approx r_i + k \cdot rInt$ with k being proportional to N_j, we get:

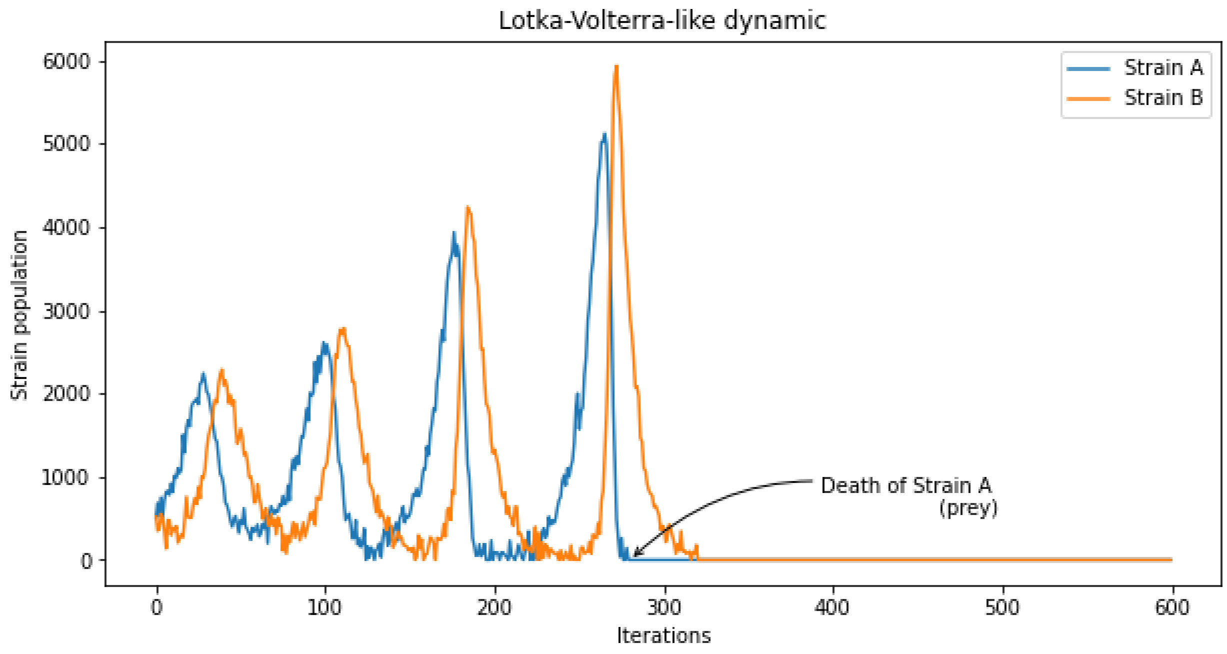
$$E[R_i] \approx N_i(r_i + N_j rInt)$$

$$E[D_i] \approx N_i(d_i + N_j dInt)$$

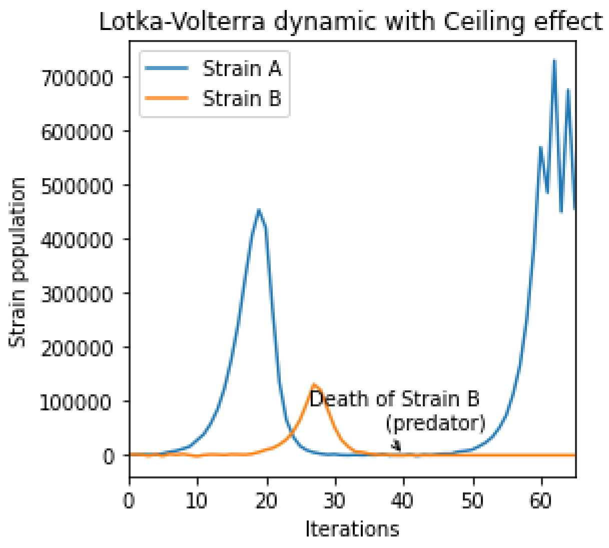
and same for j.

Approximating the first derivative in Lotka-Volterra with the forward finite difference and taking the expected value of our system evolution we can identify the Lotka-Volterra parameters with ours, and get a similar dynamic (albeit stochastic), with the following settings:

`maxTransf`	`0`
`transfP`	`0`
`deathP`	`0.5`
`reprP`	`0.5`
`deadlyInter`	`Map(1L->1e-4)`
`reproductiveInter`	`Map(0L->1e-4)`
`Node[distribution]`	`(500,500)`
`ceiling`	`1000000`



Our simulation shows the coexistence of Strain A (preys) and Strain B (predators) in a transient. Here Strain A becomes extinct at some point in time due to random fluctuations in the proximity of the absorbing barrier defined above. As a consequence, Strain B, whose existence depends on Strain A, vanishes suddenly after.



Here we explore another dynamic in which predators become extinct before the preys. This has a different consequence on preys which are now free to reproduce without the negative feedback given by the predators.

In order to avoid strain population size to diverge, we fixed the ceiling parameter to 1000000. As one can see, the effect of the ceiling becomes appreciable when population size reaches 600000 units.

The upper barrier lowers the reproduction rate as explained in details before. This behaviour mimics a bottleneck: the environmental condition modulates fitness which suddenly goes down.

Conclusions

We have used GraphX as a basis to build a simulation framework that allows us to model the time evolution of a dynamical graph of different bacteria strains that can interact with each other and travel along the edges.

The flexibility and power given by the Scala language allowed us to exploit Spark GraphX and its methods to efficiently explore and update the graph. Even if we were new to Scala, we managed to write our program in a relatively easy way: the abstraction of RDDs and `aggregateMessages()` let us worry only about the program logic, while the framework handles the complex operations of parallel computing and fault tolerance.

Writing an iterative simulation software as our own with Hadoop MapReduce would require a lot of MapReduce jobs for each iteration, with expensive I/O operations which are avoided by using Spark. By exploiting Spark's optimization engine, we can get a scalable parallel program with a reasonable computing time. Not having a cluster to work with, we have run the program locally, but of course with more computing nodes the runtime becomes smaller [10].

With our simulations we have obtained a few known results of population dynamics such as the random walk behaviour and genetic drift in neutral selection, which are then compared to slightly more complicated scenarios that include migration, which stabilizes the population, and finally the interaction of strains. In this last case the selection is no longer neutral, as mutualistic strains will eventually be fitter than neutral and competing ones, so that their population becomes much larger.

As a final simulation, we have considered the famous Lotka-Volterra prey-predator system, and also here we managed to see the typical periodic solutions (limit cycles) with a few differences given by stochastic nature of our framework.

Our first experience with Scala and GraphX has showed how easy it has become to write a parallel program without having to consider the complex technical details of cluster computing. Furthermore, the range of possibilities offered by these frameworks is limited only by the fantasy of the coder, as we (hope to) have shown.

References

[0] Ehrlich et al.: *“What makes pathogens pathogenic”* Genome Biology 2008
[1] Job, Jacobs: *“Drug-Induced Clostridium-Difficile associated disease”* Drug Safety 1997
[2] Manichanh et al.: *“Reduced diversity of faecal microbiota in Crohn’s disease revealed by a metagenomic approach.”* Gut 2006
[3] Müller et al.: *“The diversity and function of soil microbial communities exposed to different disturbances.”* Microbial Ecology 2002
[4] Buchinsky et al.: *“Phenotypic plurality among clinical strains of non-typeable Haemophilus influenzae determined by symptom severity in the Chinchilla laniger model of otitis media.”* BMC Microbiology 2007
[5] Forbes et al.: *“Strain-specific virulence phenotypes of Streptococcus pneumoniae assessed using the Chinchilla laniger model of otitis media.”* PLoS ONE 2008
[6] Jolley et al.: *“Using multilocus sequence typing to study bacterial variation: prospects in the genomic era”* Future Microbiology 2014
[7] Buckee et al.: *“The effects of host contact network structure on pathogen diversity and strain structure” doi: 10.1073/pnas.0402000101*
[8] "Bacterial motility." McGraw-Hill Encyclopedia of Science. and Technology, 1960: 63.
[9] Murray et al., *Medical Microbiology*, 5th ed., Chapter 5
[10] Teixeira et al., *“Using Spark and GraphX to Parallelize Large-Scale Simulations of Bacterial Populations over Host Contact Networks”* Algorithms and Architectures for Parallel Processing 2017