Author:
Chalkley, Oli

Title:
Towards rational genome design

the genome design suite, whole-cell models and minimal genomes

Towards rational genome design:

the genome design suite, whole-cell models and minimal genomes

By

OLIVER CHALKLEY

Department of Engineering Mathematics
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

APRIL 2019

Word count: fifty six thousand

Abstract

Creating organisms with predetermined traits has long been a goal of synthetic biology and has the potential enable the creation of cell factories, medical tools and bio-materials. The genome is a key factor in controlling attributes of an organism and biologists have made great progress in writing and editing genomes, ushering in an era of synthetic organisms. However, limited understanding of how phenotypes emerge from genotypes has contributed towards a lack of genome design tools leaving genome engineers unsure what genome edits to perform or genomes to create. Computer models hold great potential in aiding the design of genomes but the tools are designed for specific models and do not adapt well. Additionally, the models focus on specific processes in a cell (e.g. metabolism) and miss control mechanisms that feed into those processes from other processes (e.g. gene regulation of metabolic enzymes). Whole-cell modelling may enable models that can incorporate systems-level effects and minimal genomes may simplify the genotype phenotype relationship. Unfortunately, there are currently no methods to accurately find minimal genomes and there is only one published whole-cell model that is hard to use and too computationally expensive to use for large-scale *in-silico* experiments.

This thesis aims to create tools to design *in-silico* genomes by enabling massive *in-silico* experiments on state-of-the-art computational models. These tools should be easily adaptable to different models, computers clusters, design goals, and design algorithms. As a proof-of-concept these tools will be used to try and reduce the genome of the only published whole-cell model. The results of this will be further analysed in the hope of learning about the whole solution space of genome reductions and how it may help genome reduction algorithms.

Towards these goals we present the genome design suite which enables massive *in-silico* experiments across multiple computer clusters and can avoid maximum simulation times imposed on the cluster. The code is designed to make it easily adaptable. The genome design suite was used to perform over 100,000 *in-silico* gene knockout experiments to develop genome reduction algorithms and reduce the whole-cell model of *M. genitalium* by 165 genes. This minimal genome is described biologically and further analysis of the simulations reveal multiple paths of convergence to the minimal genome, high and low-essential gene combinations and the role of dynamic gene essentiality in shaping the solution space.

We conclude that the genome design suite can aid *in-silico* genome design. The ability to avoid maximum simulation times on clusters and utilise multiple clusters enables larger-scale *in-silico* experiments giving new insights into solution spaces. Its adaptability allows it to evolve to new models, design goals, and design algorithms. It also enables the genome design tools built on it to utilise new models quickly and *vice versa*.

Dedication and acknowledgements

My PhD has taken me on a journey that I will never forgot. This amazing experience would not have been possible without the support of the following people.

First I would like to thank the genome design group. Without your ideas, support and encouragement I cannot imagine the path I would have taken. Each member made their own unique contributions to my progression that I greatly appreciate. Dr Lucia Marucci and Professor Claire Grierson I owe you a huge thank you - I truly cannot imagine a better combination of supervisors. As a team you gave the perfect balance of challenging my ideas whilst supporting my development - you were always patient, enthusiastic, or skeptical appropriately. More specifically, Lucia, thank you for your consistent belief in me and enthusiasm for my ideas. You always knew what I was talking about and was able to contribute to the progression of my work whether it be technical or general. Equally I would like to thank Claire whose never ending patience and enthusiasm was always a great support for me. You have a great talent in making me see things in another way and explaining the complexity of biology to me. No matter the what hurdle crossed my path, you knew how to advise me. Dr Oliver Purcell, I would like to thank you for collaborating with us - our long conversations about complex/crazy ideas were both enjoyable for me and critical to my progression. You also always brought a fresh perspective to the table. Joshua Rees, I'd like to thank you for the work we have done together. It was an absolute pleasure to work with someone that would instantly reply to my messages and be just as excited about a result even if it was a bank holiday Monday. Our complimentary strengths and weaknesses made us a good team and you provided some healthy competition to keep the ball rolling. Sophie Landon, I like to thank you for the great work and support you provided from the moment you started. Your ability to see through the complexity of a problem was always welcome and I always looked forward to hearing what insight you would bring to a conversation.

I would like to thank all members of the Bristol Centre for Complex Science for providing such a welcoming, inspiring, and supportive atmosphere. In particular I'd like to thank Alonso Espinosa Mireles De Villafranca for always having some relevant knowledge no matter what subject I was talking about. You were great at bouncing ideas ideas around and always provided a welcome distraction when necessary.

I would like to thank the Advanced Computing Research Centre and BrisSynBio for providing world-class high-performance computing facilities and support, as *standard* - this research would not have been possible without you. In particular I would like to thank Steve Roome, Simon Burbidge, Dr Matt Williams, and Dr Christopher Woods.

I would like to thank Dr Jonathan Karr and Professor Markus Covert for their advice on

whole-cell modelling, and Professor John Glass for sharing his advice and data on *Mycoplasmas*.

I owe a big thank you Jenna Barratt for her never ending patience, enthusiasm and inspiration. Thank you Joanne Pettitt for your advice, encouragement and proof-reading. Finally I would like to thank my family, *the Chalkleys*, Vince, Jan, Charlotte, Rob, and Ben for all of your support, encouragement, and understanding through this long journey.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: [redacted] .... DATE: 30/04/2019

Table of Contents

List of Tables

List of Figures

Figure **Page**

CHAPTER 1

BACKGROUND

1.1 Introduction

Biology has evolved a vast array of functions/attributes to persist on earth. Enzymes enable complex networks of reactions that enable chemical transformations that laboratories would be unable to reproduce without biological tools. For example, bacterial and fungal species have been shown to break down plastics, pollutants and other waste products[1] and others have been shown to produce biofuels, amino acids and other important pharmaceutical precursors like complex snake venoms[2]. These tiny chemical factories can work on minute scales but in massive armies providing intricate chemistry that is scalable. Furthermore, these cell factories exist in *bodies* that can respond to environmental changes/perturbations in both resilient and robust ways. Multicellular life performs cooperation and self-organisation like nothing else in the known universe and has resulted in the incredible data processing ability of brains, the orchestration of highly adaptable immune systems, and the creation of biomaterials with novel properties. It is no surprise that scientists have taken inspiration from biology for a long time, however, until relatively recently the complexity of biology required that engineers remove the biological parts and work with the concept within a simpler framework to be able to engineer it effectively (e.g. flight).

The twentieth century saw crucial breakthroughs in biological understanding that have ushered in the age of synthetic biology which attempts to combine engineering and biology. Synthetic biology has a broad range of goals, but a major one is the engineering of cells. Examples of engineered cells could be cell factories[3] that break down waste products and produce useful by-products like biofuels[4] or biological machines that could be used for medical applications[5].

Since the genome of an organism contains all the information needed for it to function, it is a prime goal to design an organism through its genome. Research on genome engineering is done on a wide range of biological life, but prokaryotes are often used due to their simplicity and fast life cycles. The following sections will investigate some key topics in organism design related to this thesis.

1.1.1 Genome engineering

Humans have been consciously and unconsciously[6] modifying the genomes of organisms for thousands of years through the process of *selective breeding* where individuals with desired traits are bred, and any offspring with superior phenotypes are selected for.

In the 1920s scientists started modifying genomes by *mutation breeding* which involved inducing mutations on common crops like barley. Mutations are randomly induced with X-rays, gamma rays and occasionally by chemical means. Offspring with desirable traits are selected for, and whilst there is no requirement to document plants that underwent mutation breeding, as of August 2007, there were 2,543 recorded plant varieties released that underwent this method. The true number is expected to be much higher[7].

While both methods provided great improvements on our ability to manipulate biology they both had major restrictions. For selective breeding, one has to find two organisms that have desired phenotypes and those organisms need to be able to mate. Furthermore, it is a matter of luck if offspring end up with the combination of traits desired. For mutation breeding, the mutations are completely random in where and how many mutations occur. The chance that the right combination of mutations actually happens is low, resulting in mostly undesirable and fatal mutations. However, without a good understanding of information processing and transfer in biology, it would be hard to improve on these methods.

Discovering the structure of DNA was a pivotal moment[8]. It helped to reveal the *code of life* and this combined with linking DNA-replication, transcription, and translation to discover the so-called *central dogma of biology*. In an effort to understand an organism's information storage and processing abilities scientists started to discover mechanisms that cells and virus' use to modify DNA, RNA, and proteins. Boyer and Cohen are credited with the invention of recombinant-DNA (rDNA) technology which was a key moment in genome editing methods. They were able to take two plasmids containing different antibiotic resistance markers, cleave them and combine them into one plasmid. This new plasmid was then transformed into *E. coli* which then showed resistance to both antibiotics[9]. They soon repeated the feat except this time the plasmid contained genes from the African clawed frog, *Xenophs laevis*, which the *E. coli* cells went on to express[10]. This research showed that genomic modification could be performed

with a precision that may enable traditional engineers to consider cells as a new tool to work with.

Many ways have been developed to perform genome editing, but most involve a similar process, i.e. the creation of some sort of vector (e.g. plasmids, bacteriophages, cosmids or phasmids) and integrating the vector into a cell. The techniques can roughly be categorised by whether DNA, RNA or protein is used to recognise the target site and if it makes double-strand breaks. These techniques not only have the ability to introduce DNA into the cell but also to cut, copy and paste parts of the existing genome. These methods faced many problems like low efficiency, *scarring* the DNA, and a limited number manipulations in one integration[11]. However, the invention CRISPR genome editing technology promises to greatly reduce these problems ushering a new era of precise, high efficiency, multiplexed, genome manipulation[12].

At the JCVI Gibson et al., in a *tour-de-force* of synthetic biology, published a paper in 2010 detailing the synthetic construction of a *M. mycoides* genome which was then transplanted into an empty *M. capricolum* cell[13]. They showed that the synthetic DNA was the only DNA in the new cell, including watermarks to prove it was synthetic, and the cell also started to behave like a *M. mycoides* cell.

With such great control over an organism's genome the natural next step is to start engineering organisms which scientists have started to do with mixed results. High throughput experiments and bioinformatics tools have enabled a better understanding of biological processes on certain scales but it is often still unclear how phenotypes emerge from genotype and so predicting the result of genome edits is hard. As a result, biologists often have to use random or loosely guided trial and error approaches[14]. So whilst great progress has been made in creating tools to create and edit genomes there has been very little progress in tools to help researchers decide what genome edits to make or genomes to build[15]. Using laboratory evolution to guide a cell to evolve certain attributes is probably the most deterministic way to engineer a cell and this still lacks the precision and rationale desired[16].

One way of making the design stage easier is to reduce the genome of a cell as much as possible in favourable and stable conditions without killing the cell. This concept is referred to as the *minimal genome* or *minimal gene set* and would provide the simplest possible example of a cell as the removal of any genetic material would result in cell death. The simplicity of such a cell may be an easier place to start when trying to understand the relationship between genotype and phenotype. It also has an additional benefit for metabolic engineers because the cell is not wasting resources on unnecessary processes which may make it easier to optimise the production of some desired molecule. Taking these ideas further, the minimal genome could be the base genome of all synthetic organisms. It is also believed that this may help in understanding what

makes something *alive* and the origins of life. Another way would be to use computer models to aid rational genome design[17, 18]. Gibson et al. at the JCVI are attempting to combine both approaches by creating a whole-cell model of a minimal synthetic organism[15].

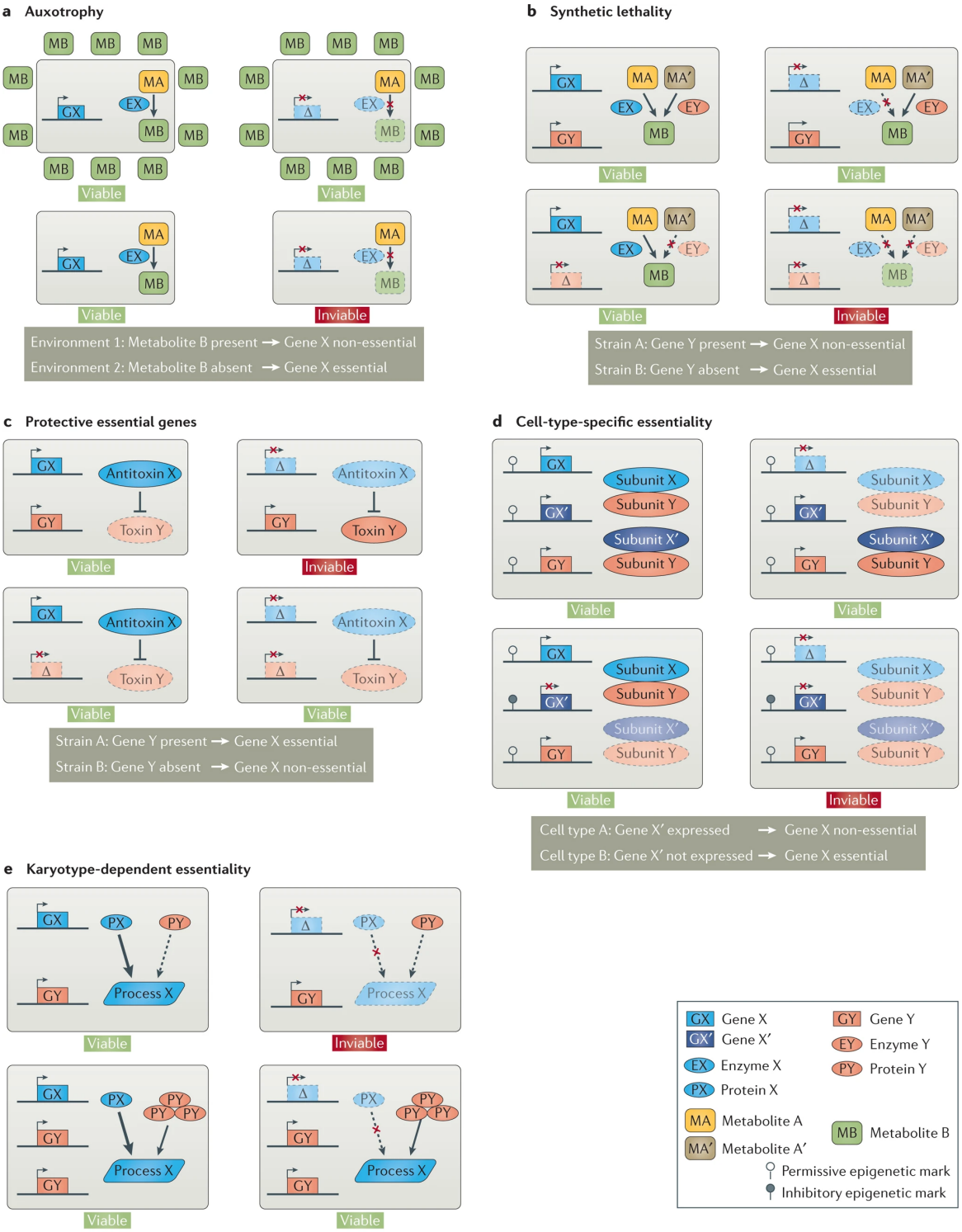
1.1.2 Minimal genomes

With the value of minimal genomes realised, scientists started to focus their attention on this goal.

In the 1960s and 1970s it became accepted that the *Mycoplasmas* were the smallest and simplest self-replicating organisms[20] which led Morowitz and Wallace to suggest that they may be near the top of the phylogenetic tree, i.e. a close relative of the last universal common ancestor[21]. Neimark contested this and suggested that their simplicity arose through degenerative evolution from more complex prokaryotes with cell walls[22]. Later this was confirmed in a phylogenetic study by Woese et al.[23]. *M. genitalium* is now believed to be the smallest, naturally occurring, self-replicating organism, making it a great organism to start with when looking at minimal genomes[24].

Based on the idea that all life on earth evolved from the last universal common ancestor, it was hypothesised that there might be one common set of genes that are essential to all life. It would follow from this that there is one minimal genome that all organisms can be reduced to. A goal of comparative genomics is to exploit this theory by finding all the genes in common between a set of organisms — if one were to sample enough different organisms, then they would eventually be left with only the genes essential to life. This research yielded useful information about minimal gene sets, but in terms of definitively finding the minimal gene set, it was flawed due to a trade-off between trying to avoid non-orthologous gene displacements and reducing genetic redundancy. Genetic redundancy can help protect a cell against randomness inside and outside of the cell and so cells will have similar redundancies with close relatives. In an attempt to minimise redundancies comparisons are, ideally, made on a large population of different cells with diverse phylogenetic histories. However, cells can evolve non-orthologous genes to do the same essential function and so will not be recognised as the same gene. The larger the number of genomes and the more phylogenetic diversity, the more likely it is to find non-orthologous genes[25, 26]. In practice researchers quickly realised that comparisons resulted in small amounts of conserved genes. A study by Lagesen et al. in 2010 compared the genomes of 1,000 bacterial species and found only 4 conserved genes (2 RNA and 2 protein-coding genes)[27] and in 2012 Lui et al. compared the genomes of 20 *Mycoplasma* species and found only 196 genes conserved[28]. It seems that comparative genomics may be good for approximating subsets of essential genes but seems to underestimate the minimal gene set significantly.

In 2006, Glass et al., performed global transposon mutagenesis on *M. genitalium* and found that



Nature Reviews | Genetics

382 of the 482 protein-coding genes are essential and note that this is significantly larger than predictions of comparative genomics studies at the time[29]. It is worth noting that this only looks at if single gene knockouts are essential or not and so assumes that multiple non-essential genes knocked-out in combination cannot be essential and that combinations of genes involving essential genes cannot be non-essential. In this thesis, if a gene is knocked-out on its own and it kills the cell then it is called singularly essential, if it does not kill the cell, then it is called singularly non-essential.

Over time it has become apparent that whether a gene is essential or not is dependent on the other genes present in the genome (genomic context), the environmental conditions (environmental context), and the ability of the cell to mutate to adapt to the loss of the gene (evolvability). These aspects will be discussed from the perspective of Rancati[19].

Environmental context is when a change in environmental conditions change the essentiality of a gene. For example, *Auxotrophy* (Figure 1.1(a)) is when gene A is non-essential when the cell is in an environment that is rich in metabolite M but if metabolite M is not present in the environment, then gene A becomes essential.

Genomic context is when other genes being present in the genome or not affect the essentiality of a gene. Rancati et al. give examples of synthetic lethal genes (Figure 1.1(b)) where

Figure 1.1 *(preceding page)*: Schematic representations illustrating different examples of context-dependent gene essentiality. a | A hypothetical gene X encodes enzyme X, which is required for the production of the essential metabolite B. In an environment where metabolite B is present, gene X is dispensable. When metabolite B is absent, gene X becomes essential. This phenomenon is also known as auxotrophy. b | Hypothetical genes X and Y encode enzymes performing redundant biochemical reactions. Whereas inactivation of either gene alone leads to viable cells, the simultaneous deletion of both genes causes cell death. This is an example of synthetic lethality. c | Hypothetical gene X encodes an inhibitor of toxin Y. In the absence of toxin Y, gene X is dispensable, but its activity is required for viability in the presence of the toxin. Gene X is an example of a protective essential gene. d | Hypothetical genes X and X' encode mutually exclusive and redundant subunits of an essential protein complex with subunit Y. In cells in which the expression of gene X' is epigenetically silenced, gene X becomes essential. This could form the basis of cell type-specific essentiality in multicellular eukaryotes. e | Hypothetical gene X encodes a protein that promotes essential process X. At a normal level of expression, the product of gene Y does not contribute to process X. Upon upregulation of protein Y (for example, due to aneuploidy of the chromosome encoding gene Y), a hidden function of protein Y is unmasked, leading to its promotion of process X. Therefore, the essentiality of gene X could be bypassed by the acquisition of mutations that upregulate gene Y. This is the basis of high copy number suppression screens and occurs frequently during adaptive evolution of yeast species. Figure adapted from Rancati et al.[19]

Definition based on	Extent of essentiality			
	No essentiality	**Low essentiality**	**High essentiality**	**Complete essentiality**
Context dependency	Dispensable in all environmental and genetic contexts	Dispensable in most environmental and genetic contexts	Indispensable in most environmental and genetic contexts	Indispensable in all environmental and genetic contexts
Evolvability following gene inactivation	No compensatory mutations required for survival	Compensatory mutations are required for survival. For these compensatory mutations, multiple independent compensatory mechanisms exist and/or the mutations occur at high frequency and/or they are easily selected and fixed in the population	Compensatory mutations are required for survival. For these compensatory mutations, only a few compensatory mechanisms exist and/or the mutations occur at low frequency and/or they are not easily selected and fixed in the population	No compensatory mechanism exists

Table 1.1: Definitions of gene essentiality - adapted from Rancati et al.[19]

there is one essential function that is supported by two different pathways and so disrupting one pathway is fine but disrupting both kill the cell. Protective essential genes (Figure 1.1(c)) are ones where one gene results in a fatal effect (e.g. creation of a toxin) and the other gene protects against the fatal effect (e.g. removing or breaking down a toxin). Cell-type-specific essentiality (Figure 1.1(d)) describes when cells of the same species (e.g. cell type or strain) have different gene essentiality — for example the genomic content can be very different between strains of the same species of bacteria potentially leading to different essentiality results for a particular gene. Finally, karyotype-dependent essentiality (Figure 1.1(e)) is where the regulation of a gene can affect the essentiality of another gene — for example gene A may be essential under "normal" expression of gene B, but when gene B is over-expressed gene A becomes redundant. Environmental and genomic context can also be linked — for example, synthetic lethality can be dependent on the environment.

Evolvability refers to the case when a gene is technically essential, but the cell evolves to compensate for the deletion. The example given for evolvability is when a gene essential for cytokinesis was deleted in budding yeast, some cells were able to evolve a new method of cytokinesis.

These effects combine to give a gradient of essentiality. Rancati et al. define the extent of essentiality in four categories for both context and evolvability. No essentiality is when a gene is dispensable in all contexts, and there are no compensatory mutations possible to enable survival. Low essentiality is when a gene is dispensable in most contexts and/or there are compensatory mutations that are likely to occur before the death of all mutant cells. High essentiality is when a gene is indispensable in most contexts and/or there are compensatory mutations that are possible but not likely to happen before all the mutant cells die. Complete essentiality is when a gene is indispensable in all contexts, and no compensatory mutations exist. This gradient of essentiality is summarised in Table 1.1.

With regards to the gradient of essentiality proposed by Rancati et al. this thesis focuses solely on genomic context and so Table 1.1 has been adapted to reflect this - see Figure 1.2.

Non-binary gene essentiality has significant ramifications for minimal genome research because it results in a combinatorial explosion of experiments needed to determine the minimal genome of an organism. In the binary case, one only needs to find the single gene essentiality of each gene in the genome, and then one knows that the minimal genome is simply the set of all the singularly essential genes. Just incorporating genomic context means that unless one can predict all the genes with non-static essentiality, then every possible combination of gene knockouts must be tried. There are $\sim 10^{157}$ unordered combinations of gene knockouts for an organism as small as *M. genitalium* which is likely to be underestimating the real number since there is evidence that the order of gene knockouts can alter the mutant phenotype[30]. Adding environmental context and evolvability adds even more complexity to the problem. It is clear that this would require a prohibitive amount of resources to perform the experiments for the foreseeable future. Therefore, methods need to be developed that can find minimal genomes in a smaller amount of experiments.

In 2016 Hutchinson et al. repeated the feat of transplanting a synthetic *M. mycoides* genome into an empty *M. capricolum* cell only this time reducing the synthetic genome to an approximation of the minimal genome[31]. Their method for reducing the genome utilised a divide and conquer strategy on only the singularly non-essential genes (without repetition with different initial conditions). This strategy means that their method cannot utilise high-essential genes to reduce the genome, they have no idea if there are multiple local minima, and they do not know if their reduction is close to the optimal local minima. Whilst the feat is a milestone for synthetic biology and the minimisation process is one of the best examples of rational genome design, again the results are dominated by chance, and much is still left to learn about genome reduction in *M. mycoides* and organisms in general. It has been suggested that computer models may help to rationally design genomes[15, 17].

1.1.3 Genome scale computational models

With an impossibly large number of gene combinations to test, experimental biologists have often had to rely on random guessing, luck, and simple trial and error methods for design[14]. In other disciplines, like physics for example, researchers often rely on mathematical modelling of systems to reduce the burden on experimentalists. Here we look at some traditional modelling approaches in biology - this is will not be a review of the field but rather a few choice examples to highlight some key points. For a more thorough discussion on mathematical formalisms in biology see the review by Machado et al.[32]. Areas of biology that modellers have often focused on are gene regulatory networks, signalling networks and metabolic networks as these areas are likely to provide the intricate network of feedbacks that enable cells to act as a complex adaptive system[33].

An organism's genome codes for all the functions utilised by the organism. However, in or-

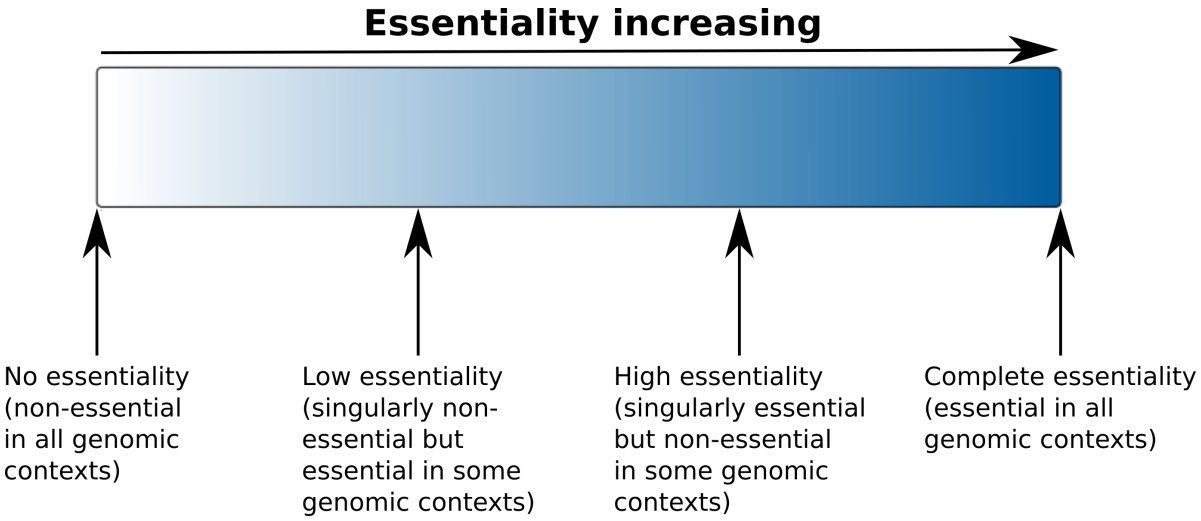


Figure 1.2: The gradient of essentiality adapted for gene knockout experiments in this thesis shows how genomic context turns gene essentiality from a binary concept to a transient one. The left and right extremes are genes that obey traditional ideas where they are either always non-essential or always essential, respectively. The gradient inbetween represents genes that can be both essential and non-essential depending on genomic context. Figure adapted from [19].

der to achieve the complex behaviour displayed by life, genes must be expressed in dynamic ways that change depending stimulus from both inside and outside of the cell. A gene is expressed through both transcription and translation. The cell can control the transcription of genes through transcription factors that are themselves coded by genes that may have transcription factors. The control of gene expression is often studied as *gene regulatory networks*. A mathematical formalism for this process is *boolean networks* that model the expression of genes as boolean variables in discrete time[32].

A cell can receive and respond to external stimuli and the study of this is called signal transduction. A cell combines these processes to form complex signalling networks. The process normally starts with an external molecule that binds to a receptor on the surface of the cell. The binding process starts a cascade of signals that trigger the cell's *response*. This often utilises gene regulatory networks to enact the desired response. Stochasticity is important in signalling networks due to the small number of signalling molecules, and so a common mathematical formalism is stochastic simulation which involves making assumptions about the probability distribution of reaction activation[32].

The metabolism is made up of large networks of chemical reactions. In chemistry, reactions are normally modelled by systems of ordinary differential equations. Attempts have been made in biology to follow this formalism but getting accurate data to parameterise these models is hard[34]. The metabolism imports and exports molecules to and from the cell and creates all the

molecules necessary to sustain biological function and so it is also related to gene regulatory and signally networks. Conversely, gene regulation and signal transduction can affect metabolic reactions.

It can be seen that all these processes are modelled in different ways and furthermore, have more representations that have not been discussed here[32]. Having different mathematical representations can cause problems when integrating systems together. For example, being able to model an entire system with a system of ordinary differential equations opens up many potentially useful tools like calculus and stability analysis[35, 36] - in some cases, the system may even be analytically tractable enabling exact mathematical representation. The advantages here is that one can not only simulate particular instances of a system but can also analyse the entire solution space (e.g. analyse the effects of parameters, initial conditions and stochasticity[①]). Modelling a system in one mathematical formalism is an ideal situation, however, sometimes it is not known how to represent systems in one formalism. In this case, different formalisms can, sometimes, be integrated by feeding the outputs of the model into each other through computer simulation. In order to do this, both systems need to represent quantities that can be transformed into the quantities used by the other formalism and *vice versa*. Whilst this enables the modeller to run specific simulations, the ability to analyse the whole solution space is severely depleted and requires the user to sample simulations with random parameters and initial conditions and hope that the important features become apparent. In some cases, even when the different systems work in transferable quantities, the modelled processes work on vastly different scales. For example, imagine the time scales that molecules interact on during chemical reactions and compare that to the time scales of a chaperone being created to fold a desired protein or even further still, imagine the time scales that a cell moves and responds to chemical gradients. Some mathematical formalisms require the same time scales, for example, systems ordinary differential equations, in this case, all modelled processes must act on the smallest time scale which results in an explosion of CPU work and data storage that can make it computationally intractable for meaningful durations of time. Integrating inter-related biological systems has been the focus of much biological research[37, 38] and will be elaborated one throughout this section.

A different approach to modelling the metabolism of a cell which is particularly relevant to this thesis is to use constraint-based methods on genome-scale metabolic models. Due to difficulties parametrising genome-scale metabolic models of ordinary differential equations, biological modellers turned to stoichiometric representations of the metabolism. The stoichiometry of a system of reactions can be represented in a matrix, $\overline{S} = [s_{i,j}]$, where each column, represents a reaction j, and each row represents a metabolite i. The value of the entry in the matrix, $s_{i,j}$, is the

[①] General relativity is an example of a system that fits into one mathematical formalism. Analysis of the solution space of this model led to the birth of the field of cosmology and made the first prediction of the existence of black holes.

stoichiometric coefficient of metabolite i, in reaction j where the sign is positive if the metabolite is being imported or produced and negative if it is being exported or consumed or *vice versa* depending on the preference of the modeller. Reactions transforming metabolites within the cell are referred to as internal reactions and reactions that import and export metabolites inside and outside of the cell (or across compartments within the cell) are called exchange reactions. Analysis methods were developed based on the principle of mass conservation of internal metabolites[39]. The mass conservation equation for a system of known volume is

$$\frac{d\underline{C}}{dt} = \overline{S} \cdot \underline{v} - d \cdot \underline{C} \tag{1.1}$$

where $\frac{d\underline{C}}{dt}$ is the rate of change of the vector of the concentration of m metabolites (mol/L) with respect to time, $\underline{v}$ is the vector of n reaction rates (mol/L/hr), also known as the flux vector, $\overline{S}$ is the stoichiometric matrix of size $m \times n$, and d is the specific dilution rate associated with the change in volume of the system (1/hr). Since the reaction rates happen on a much faster scale than the dilution rate, the concentration changes due to dilution is negligible and so the system at steady state can be written as

$$\frac{d\underline{C}}{dt} = \overline{S} \cdot \underline{v} = 0. \tag{1.2}$$

The only unknown in the equation is $\underline{v}$ which is a well studied problem in linear algebra. There are further restrictions on the problem because the reaction rates must obey thermodynamic constraints and so place minimum and maximum bounds on the flux vector $\underline{v}^- \leq \underline{v} \leq \underline{v}^+$. Cellular metabolism normally results in an under-determined system which means that there are considerably more reactions than metabolites. Trinh et al. cite three main ways to solve this problem for $\underline{v}$, metabolic flux analysis, flux balance analysis, and metabolic pathway analysis, however, we will focus solely on flux balance analysis - for more information please see [40, 41].

Since the system is under-determined (i.e. $m \ll n$) it is not possible to produce a unique solution by inverting the stoichiometric matrix. Flux balance analysis works by assuming that evolution has optimised certain metabolic attributes (e.g. growth rate) over time. If one is able to construct a realistic objective function that mimics this natural optimisation objective then there are optimisation algorithms that can find the flux distributions that maximise (or minimise) the objective function subject to constraints like substrate uptake rates, secretion rates, thermodynamic constraints, metabolic regulation and any other data sources that might be obtained [40]. Figure 1.3 depicts the FBA optimisation process and one can see that the model constraints produce a *flux cone* of possible solutions which then FBA finds the flux distribution on the edge of the cone that optimises some objective function. Models that contain all genes directly related to the metabolism are called genome-scale metabolic models.

Constraint-based genome-scale metabolic models have received much attention since their ability

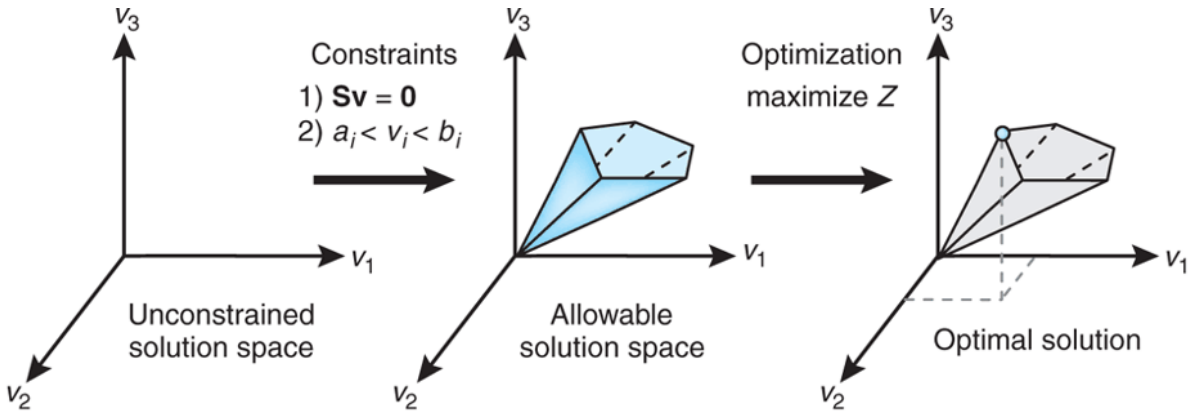


Figure 1.3: A visual representation of FBA optimisation of a hypothetical 3-reaction system. The solution space starts unconstrained (left). Constraining this solution space with the stoichiometric matrix and flux constraints reduces viable solutions to those inside the *flux cone* (middle). FBA identifies a single flux distribution on the edge of the flux cone that optimises some objective function (right). Figure adapted from [34]

to accurately predict growth rates was demonstrated. These models have gone on to demonstrate many other successes related to gene/reaction/metabolite essentiality studies, metabolic engineering and aiding biological understanding/discovery but the area has had to evolve to keep improving. For example, it was found that trying to construct an objective function that accurately describes a cell's metabolic *desire* combined with trying to understand global metabolic behaviour from a single flux distribution had limited use. Research has now moved away from this practice and started to take a more holistic approach of combining -omics data with a better understanding the whole solution space using Markov chain Monte Carlo sampling and metabolic pathway analysis. Early successes combining -omics data with constraint-based models highlighted how giving the model *context* enhanced its predictive capability. This effect of needing context can be explained by the model's inability to simulate non-metabolic processes that regulate cellular behaviour (e.g. expression regulation and signalling). Taking this idea further, researchers have attempted to integrate other biological processes like transcription and translation into these constraint-based models. Whilst these integrated models have shown some success they require more data to accurately parameterise the model and the models to be integrated need to be taken from their preferred mathematical formalism and *forced* into a linear optimisation formalism through the use of *coupling constraints*[42].

Many tools have been created and to some extent used to aid in organism design, but they rely on the stoichiometric formalism and require a specific implementation of that formalism (e.g. COBRA[43–45]). Little has been done to create general tools that can take into account different models and can utilise HPC facilities.

Whilst there has been much success in gene essentiality prediction using constraint-based models [46–50], there is still likely room for improvement by integrating feedbacks from other processes and not to mention there are many genes outside of the genome which can affect a cells ability to survive[37, 38].

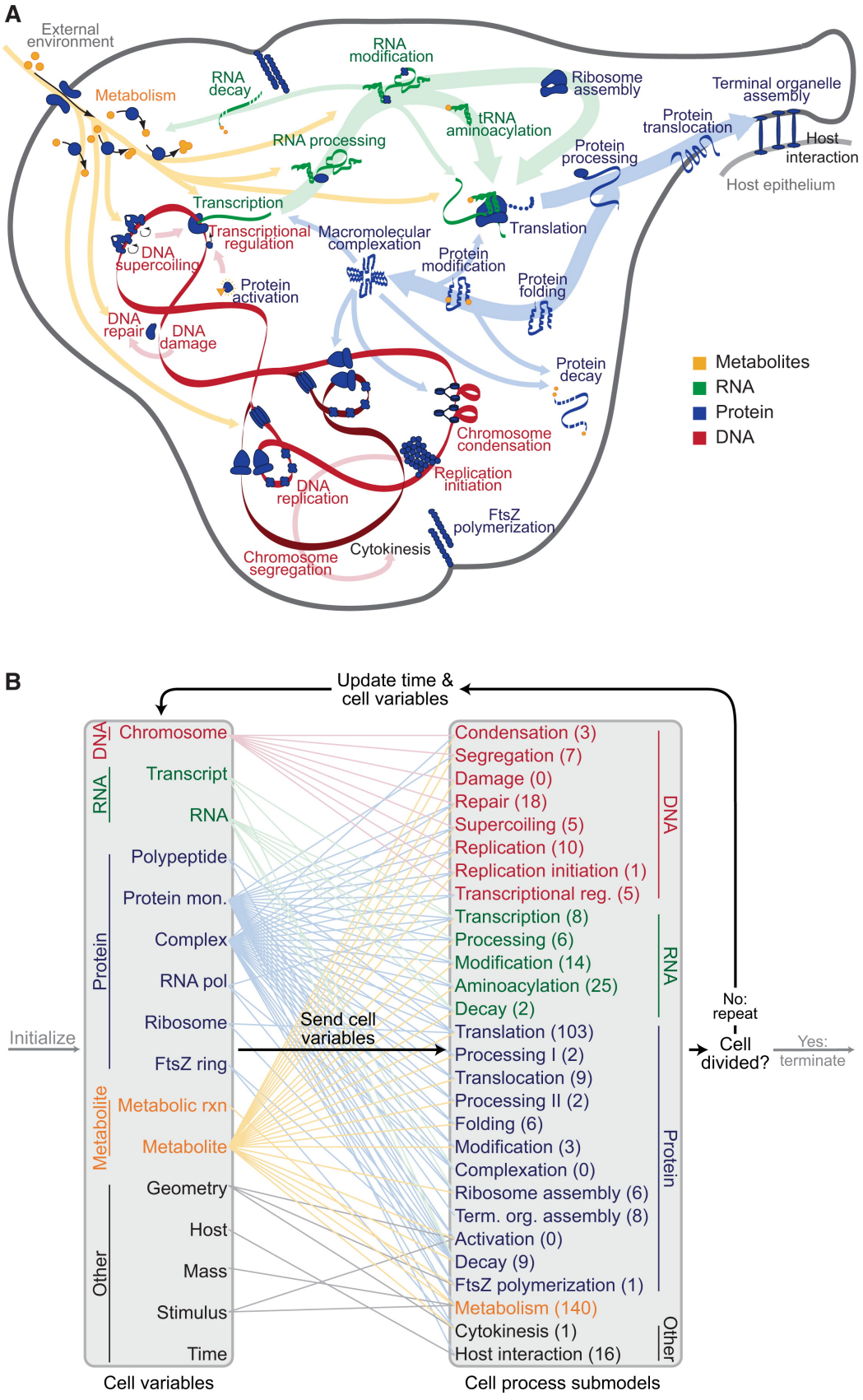
When thinking of a cell as a complex system combined with the idea that biology has *no privileged level of causation*[51], it is no surprise that researchers have been working towards a more systems view of the cell[37, 38]. Unfortunately, cellular processes are multi-scale in time and space, there is a lack of data to parameterise models, and there is no single mathematical formalism that integrates all biological processes making this a huge challenge in cellular/computational biology. To date, there is only one published whole-cell model which models the parasitic bacterium *M. genitalium* [52]. Rather than try to represent multiple biological processes into one mathematical formalism Karr et al. used existing, different, mathematical formalisms and integrated them computationally by assuming that each biological process is independent on one second time scales.

M. genitalium is smallest of the *Mycoplasma* family and is the smallest known naturally occurring self-replicating organism with only 525 genes. It lives in the urethral passage of humans and owes its simplicity to degenerative evolution[24]. Interestingly, this suggests that the urethral passage of humans is one of the most stable environments on Earth (Professor John Glass, personal communication February 2017).

Karr et al. defined 28 submodels of *M. genitalium* and independently built, parameterised and tested them (Figure 1.4A). Each submodel is represented by the most appropriate mathematical formalism, e.g. the metabolism was modelled using FBA and the degradation of gene products as poison processes. There are 16 cell variables that represent the state of the cell at any one time. By assuming that each submodel is independent on 1 second time scales, each submodel can be run individually and every second the cell variables are updated with the results of each submodel which is then fed back into the submodels for the next second (Figure 1.4B). This process is repeated until either the cell divides or 50,000 seconds pass without division.

The model is parameterised with over 1,900 experimentally observed parameters from over 900 publications. If data was not available for *M. genitalium* then data was taken from the nearest relative that had that data available. Most parameters were implemented as reported, but there were a few that had discrepancies between experimental observations and had to be carefully reconciled.

Once the model was built and parameterised, 128 wild-type simulations were run, and the cells showed key features from the training data like observed doubling time, cellular chemical


A
External environment
Metabolism
RNA decay
RNA modification
Ribosome assembly
Terminal organelle assembly
tRNA aminoacylation
Protein translocation
RNA processing
Protein processing
Host interaction
Host epithelium
Transcription
Translation
Transcriptional regulation
Macromolecular complexation
DNA supercoiling
Protein modification
Protein activation
Protein folding
DNA repair
DNA damage
Protein decay
Metabolites
RNA
Protein
DNA
Chromosome condensation
Replication initiation
DNA replication
FtsZ polymerization
Cytokinesis
Chromosome segregation
B
Update time & cell variables
Initialize
Send cell variables
No: repeat
Cell divided?
Yes: terminate
DNA
RNA
Protein
Metabolite
Other
Chromosome
Transcript
RNA
Polypeptide
Protein mon.
Complex
RNA pol
Ribosome
FtsZ ring
Metabolic rxn
Metabolite
Geometry
Host
Mass
Stimulus
Time
Condensation (3)
Segregation (7)
Damage (0)
Repair (18)
Supercoiling (5)
Replication (10)
Replication initiation (1)
Transcriptional reg. (5)
Transcription (8)
Processing (6)
Modification (14)
Aminoacylation (25)
Decay (2)
Translation (103)
Processing I (2)
Translocation (9)
Processing II (2)
Folding (6)
Modification (3)
Complexation (0)
Ribosome assembly (6)
Term. org. assembly (8)
Activation (0)
Decay (9)
FtsZ polymerization (1)
Metabolism (140)
Cytokinesis (1)
Host interaction (16)
Cell variables
Cell process submodels

composition, replication of major cell mass fractions, and gene expression.

Karr et al. went on to further test the model against independent experimental data. They found that metabolic flux ratios between certain pathways were in accordance with experimental data, metabolite concentrations were at similar levels to those of *E. coli*, it showed similar stochastic effects caused by the interaction between translation/degradation of proteins and mRNA expression, distribution of mRNA and protein levels were in line with reports, and they concluded that the model results are consistent with experimental data over multiple biological functions and scales.

Exploring the model showed that it is in line with several studies on DNA and RNA polymerase activity and sheds light on what the potential mechanisms might be. The model has three major cell phases, replication initiation, replication, and cytokinesis. It was noticed that whilst the overall cell cycle duration remained relatively consistent, the duration of replication and replication initiation varied a lot. On further investigation, it was noticed that regulation of cell cycle duration emerged from replication and replication initiation. This was because if replication initiation were faster or slower than average (e.g. because of initial conditions or stochasticity within the model) then there would be less or more time for the cell to build up the resources required for replication and thus replication would take longer or shorter, respectively. The model also shows that most of the cell's energy is spent on transcription and translation and that the cell produces significantly more energy than it consumes. Finally, the model accurately predicted the single-gene essentiality of 79% of the genes modelled. It is worth noting that there are still many genes with unknown function and in *M. genitalium* there are 124 - these are referred to as *uncharacterised* genes in the model. For this reason, the single gene knockout experiments were

Figure 1.4 *(preceding page)*: *M. genitalium* Whole-Cell Model Integrates 28 Submodels of Diverse Cellular Processes. (A) Diagram schematically depicts the 28 submodels as coloured words—grouped by category as metabolic (orange), RNA (green), protein (blue), and DNA (red)—in the context of a single *M. genitalium* cell with its characteristic flask-like shape. Submodels are connected through common metabolites, RNA, protein, and the chromosome, which are depicted as orange, green, blue, and red arrows, respectively. (B) The model integrates cellular function submodels through 16 cell variables. First, simulations are randomly initialised to the beginning of the cell cycle (left grey arrow). Next, for each 1 s time step (dark black arrows), the submodels retrieve the current values of the cellular variables, calculate their contributions to the temporal evolution of the cell variables, and update the values of the cellular variables. This is repeated thousands of times during the course of each simulation. For clarity, cell functions and variables are grouped into five physiologic categories: DNA (red), RNA (green), protein (blue), metabolite (orange), and other (black). Coloured lines between the variables and submodels indicate the cell variables predicted by each submodel. The number of genes associated with each submodel is indicated in parentheses. Finally, simulations are terminated upon cell division when the septum diameter equals zero (right grey arrow). Figure adapted from [52]

only tested on the 401 genes characterised in the model.

Whole-cell modelling shows great promise, but currently, the only published whole-cell model has significant barriers to use (Dr Oliver Purcell, personal communication December 2015 and Dr Jonathan Karr, personal communication February 2017)[53]. It is also worth looking at the publication history of this model in the 6 years since its release. In total there have been 8 papers published based on the whole-cell model. Three of these are tools from the same laboratory (Covert Lab, Stanford, USA), led by Karr, designed for use with the whole-cell model. One is the knowledge-base used to parameterise the whole-cell model[54], a second database was also published except this time the database was to store simulation data produced by the whole-cell model[55], and the third is a suite of tools to aid visualisation of simulation data[56]. Another one of the papers is a review on the difficulties of building and using whole-cell models[53], and the final three papers are the only papers to use the whole-cell model. The first is the original paper that published the model (with equal contributions from Karr and Sanghvi) and has already been discussed. The second, authored by Sanghvi, compared the growth rates of single-gene knockout mutants between the model and experimental data. Discrepancies were then used to identify problems in kinetic parameters of particular enzymes that they were then able to validate[57]. The third paper, authored by Karr, used the whole-cell to create data so that teams could develop and compare parameter estimation methods in a testable environment[58]. The fourth paper is the only publication using the whole-cell model to come from outside of Covert Lab[14]. Purcell et al. present an approach to modelling synthetic gene circuits using the whole-cell model, however, Karr is still an author on this paper. So it appears that building a whole-cell model is only one step towards enabling the biological community to fully utilise the technology. It appears that at least some of these problems stem from the fact that significant *in-silico* experiments require the use of a computer cluster as well as significant disk storage for the resulting simulation data. The large amount of data also makes analysis harder.

Overcoming the obstacles stopping the adoption of whole-cell models will take significant work and further exacerbating this problem is the fact that the field is young and so is likely to undergo rapid change. This means that developing these solutions is high risk for researchers because their solutions may not end up being applicable to cutting-edge models/technology by the time they are published. There are currently two unpublished whole-cell models under construction using the Covert Lab style submodel integration process (Dr Jonathan Karr, personal communication September 2017, and Professor Markus Covert, April 2017). The Covert Lab style whole-cell models will be able to utilise the many CPU cores in a HPC cluster to perform many simulations and to process and analyse the resulting data. A different modelling approach is also being developed that utilises high-performance GPUs to simulate biological systems on an atomistic level - they are currently not whole-cell models but recent research has worked up to

simulating the entire cytoplasm of an organism[59, 60]. Researchers at the JCVI are currently working with collaborators to create an atomistic whole-cell model of their synthetic minimal cell, JCVI-Syn3.0[15]. Finally, the references earlier in this section show how influential the Palsson Lab has been in producing cutting-edge biological models and took some of the earliest steps in integrating different biological processes - it would be surprising if whole-cell models were not produced from this laboratory in the future. Whilst it would be impossible to know what the future of whole-cell modelling holds, it is very likely to have periods of rapid development in a very competitive environment.

1.2 Aims and objectives

The overall aim of this thesis is to contribute towards genome design. Whilst the tools to create and modify genomes have been created, the tools needed to decide what genomes to create or what to part of a genome to modify are underdeveloped. The era of whole-cell models has just begun and shows much promise but are hard to use with few tools to aid in genome design. Minimal genomes may further help the rational design of genomes, but very few methods to reduce genomes have been created.

The contribution to genome design will start with tools to enable large-scale *in-silio* experiments using the whole-cell model of *M. genitalium*[52]. It is desired that these tools should aid *general* genome design by having the ability to optimise different biological functions with different optimisation methods so that they can keep up with contemporary genome design trends. It should also be general with regards to models and technology so that the tools are easily transferable to new models or HPC facilities. In addition to using whole-cell cell models to design genomes, one needs to be able to store, analyse and visualise the data so general tools for this should also be part of the suite of tools. These tools, with their generality, will also make it easier for more of the community to utilise these state of the art models.

This suite of tools will then be tested by developing methods to find *in-silico* minimal genomes. The analysis tools will then be used to understand as much about the data created as possible.

In summary, my aims are as follows

- Make whole-cell models easier to use for the community.
- Create a suite of tools to enable *in-silico* genome design.
  - Tools should be able to be easily adapted to new organisms, models, and design goals.
- Use tools to investigate genome reduction algorithms in the whole-cell model of *M. genitalium*.

- Analyse and understand the minimal gene set.
- Analyse and understand the solution space of reduced genomes.

1.3 Structure of thesis

The structure of this thesis is sequential. The methods chapter, chapter 2.3, details existing methods and tools used throughout the thesis. The remaining chapters are the results chapters followed by the conclusion chapter. The first results chapter, chapter 3, looked at preliminary work done to understand the scale of the problem and further justifies the route of the rest of the results. The second results chapter, chapter 4, illustrates the technical implementation of the tools created to aid our goals in rational genome design - this chapter is focused around code structure and how that enables our tools to be adaptable to different algorithms, different design goals, using different models on different clusters. The third results chapter, chapter 5, uses the tools created in chapter 4 to run massive *in-silico* experiments to create algorithms that can reduce genomes and to make non-viable genomes viable. The final results chapter tries to biologically interpret the minimal genome as well as taking a deeper dive into the data generated in order to better understand the solution space of reduced genomes. Finally, the conclusion chapter reflects on the research done and looks to the future.

1.4 The genome design group

Over time this project has grown from the solo PhD I originally proposed to a small group called the *genome design group* (GDG). It is a small group that works closely together, and so all members will be mentioned throughout the thesis and so will be introduced here.

- Dr Lucia Marucci - supervisor/PI.
- Professor Claire Grierson - supervisor/PI.
- Dr Oliver Purcell - external collaborator.
- Joshua Rees - PhD student.
- Sophie Landon - PhD student.

CHAPTER 2

METHODS

This chapter discusses the methods used throughout the thesis and provides references for more information. It is split into the sections for methods related to the whole-cell model of *Mycoplasma genitalium*, computing, and bioinformatics.

2.1 The whole-cell model of *Mycoplasma genitalium*

M. genitalium was simulated using the so-called Whole-Cell model [52]. It is the first and, at the time of writing, only published whole-cell model. The model is coded in Matlab and we used version 2012a [61] to run the simulations. For the rest of this thesis, the model will be referred to as the *whole-cell model of M. genitalium* and will not be referenced. For more information about how the model was constructed and what it has been used for please see section 1.1.3. This section has discussed some of the obstacles a user is likely to come up against when attempting to use the model due to technological challenges. However, there is a more basic challenge to overcome first - learning how to use the model. The supplementary information of the paper that published the whole-cell model provides data used in model development, data created in the making of the paper, and also a large document describing the biology captured in the model. The user manual, however, is much smaller and describes only a few usage examples. Unfortunately, these descriptions are either out-of-date or require the 'user-friendly configurator'. The out-of-date instructions have been replaced by using *simulation runners* and so published instructions should be ignored (including the initial setup commands) - Dr Jonathan Karr, personal communication November 2015. The online 'user friendly configurator' is not open to the general public (e.g. only members of Stanford University, USA) although it is possible to set up your own web-server to act as a personal 'user friendly configurator' - not only is this process not well documented, it requires quite advanced technical knowledge (e.g. web-servers and RDMSs)

and again, Karr now discourages this method in favour of using *simulator runners*. Since the new instructions have not been released or published anywhere the next part explains the process.

A simulation runner is a subclass that tells Matlab what kind of simulation the user would like to run. The following Matlab code snippet shows the runner created for gene knockouts and is used throughout this thesis.

```
classdef koRunner < edu.stanford.covert.cell.sim.runners.SimulationRunner
        methods
                function this = koRunner(varargin)
                        this = this@edu.stanford.covert.cell.sim.runners.
                            SimulationRunner(varargin{:});
                end
        end

    % jobNumber can be used to make a seed
    % Change the jobnumber as shown in this example:
    % runSimulation(..,'runner','AdjustParameters','jobNumber',1,..)
     properties
        jobNumber = 0;              % Preallocate jobNumber
        koList=0;
    end

        methods (Access = protected)
        function modifyNetworkParameters(this, sim)
            % set the seeed and display it
            timeNow = clock;
            seed = this.jobNumber * 10000 + timeNow(2)*24*31 + timeNow(3)
                *24 + timeNow(4);  %Override seed
            this.seed = seed

            % apply kos
            koList=this.koList
            sim.applyOptions('geneticKnockouts', koList)
        end
        end
end
```

A simulation can then be run using the following code.

```
cd PATH_TO_WHOLECELL_MASTER_DIR
addpath(PATH_TO_WHOLECELL_MASTER_DIR)
setWarnings()
setPath()
runSimulation('runner','koRunner','logToDisk',true,'outDir',
    SIMULATION_OUTPUT_DIR,'jobNumber',JOB_NUMBER,'koList',
    CELL_ARRAY_OF_GENE_CODES_TO_KO)
```

This code results in knocking genes out in the same way as the original publication[52] which is described in the supplementary information as *Gene disruption was implemented in two steps: (1) we modelled insertion of a transposon of length zero which reduces the stability of the terminal products of the deleted gene, and set the half-life of the RNA and protein products of the deleted gene to zero; and (2) to more quickly highlight altered phenotypes, we deleted all RNA and protein products of the deleted gene*.

In addition to the model there were several tools published in parallel, WholeCellKB [54], WholeCellSimDB [55], and WholeCellViz [56].

WholeCellKB is a knowledge base for biological organisms (with a focus on parametrising models). One was created for *M. genitalium* and used to parameterise the whole-cell model. The online version of this is often used when trying to understand what is happening in a simulation as one can look up, for example, genes, their products, and what reactions those products are involved in.

Two different PhD students (Joshua Rees and I) and a masters' student (Rick Vink, MIT, USA) attempted to use WholeCellViz but were unable to get it to work on any data created by our group and so was deemed inappropriate for our uses.

WholeCellSimDB was not used because it is not well documented, is intended to be situated on a single disk drive, and requires administrator privileges to setup. It was likely that our solution was going to have distributed disk and computational resources that were likely to change in the near term (this will be discussed in more detail in chapter 3) making a simple, adaptable bespoke solution more desirable than a complex *black box* centralised database.

2.1.1 Gene knockouts in the whole-cell model of *M. genitalium*

This thesis mostly focuses on the viability of gene knockout simulations, and so this is discussed in more detail here. The reader is advised to read section 1.1 for an introduction into the whole-cell model of *M. genitalium* and concepts of gene essentiality before proceeding with this section.

This thesis uses a very simplistic definition of *living* in the whole-cell model of *M. genitalium* - if the simulation divides then it is classed as alive. However, classifying cell viability in the whole-cell model of *M. genitalium* is non-trivial. Karr et al. [52] showed that their model correctly predicts the essentiality of 79% of the genes in *M. genitalium*. In order to understand the complications, this number will be investigated further.

There are 525 genes in this model, however, at the time of model construction, there were 124 genes with unknown function — these genes are transcribed and translated in the model, but their products are inert in the cell and eventually degrade or get broken down. The uncharacterised genes were removed from the test and leaves 401 functional genes. There were 29 genes that Karr classified as non-essential from a single life cycle but was experimentally shown to be essential. It is not possible for the model to simulate more than one life cycle of the cell (Professor Markus Covert and Dr Jonathan Karr, person communication February 2016), however, with further investigation Karr was able to create bespoke simulations that only used a sub-model of the whole-cell model to show that if the model was able to perform multiple generations that it would likely result in cell death. This was split into two different categories, *non-perpetuating* (23 genes) and *toxin accumulation* (6 genes). The former is caused by genes that disrupt molecules that cannot be set to zero at the beginning of a simulation (like chaperones) and so the downstream effects will not kick in until the molecule degrades which can take longer than a generation. The latter is caused by an accumulation of toxins which often do not reach lethal levels within one life-cycle. Having empirical single gene knockout data notified Karr et al. what simulations were not producing correct results which then enabled them to perform intensive investigations into why the results are wrong. From this standpoint, they were able to create bespoke functions that simulated only submodels of the whole-cell model for multiple generations and thus demonstrate that it was unlikely that the mutant would survive multiple generations. This method creates problems for our investigation. There are many more multiple gene knockouts than single gene knockouts making the kind of analysis they performed impossible. In addition to this, there is no experimental data of multiple gene knockouts for *M. genitalium* in order to tell us where to look. As more genes are knocked-out the chain of cause and effect of the knockouts becomes combinatorially larger making significantly harder to predict multi-generational effects and furthermore devising a function that will run submodels in such a way to convincingly show that the cell is unlikely to survive multiple generations. For these reasons it was decided to judge a cells viability on the single generation of data produced and the downfalls of this is dealt with in the same way that all model inaccuracies are dealt with in science, acknowledgement and transparency. As our work is a proof-of-concept we chose to use the model in its entirety and the generalisability of our tools will enable either future modifications to account for these inaccuracies or adaptability to new models that can account for

multiple generations and other improvements. It should be remembered that only performing a single life-cycle of a cell reduces a whole-cell model's ability to predict genome viability. It is worth noting that some single gene knockout mutants exhibit different phenotypes when the experiment is repeated further complicating the classification problem, but this is discussed in more detail in section 3 since this effect is not mentioned by the original authors and is discovered by results.

2.2 Computing

This section discusses the computational methods used in the thesis.

2.2.1 Operating systems, tools and programming languages

Writing code that is stable across dependency versions and operating systems (OSs) is a non-trivial job and normally requires a team of developers/testers. Additionally, this stage is normally one of the last stages of development. For these reasons, it was decided to stick to just one OS and try to avoid using multiple different versions of software.

Linux is known for its speed, stability, customisability, and is the OS on most HPC clusters and so was chosen as our desired OS. All scripting is done in Bash which is a common Linux scripting language (for more information see the official webpage [62]). A computer called the Hub was used and ran on the CentOS 6.6 distribution of Linux. BC3 is the UoB HPC cluster, it runs Scientific Linux 6.4 (Carbon) and uses the PBS 4.2.4.1 job scheduler — for more information about BC3 see the official webpage at [63]. The following is a template for a typical submission script for a job array on BC3.

```
#!/bin/bash

# COMMENTS

## Job name
#PBS -N JOB_NAME

## Resource request
#PBS -l nodes=NO_OF_NODES:ppn=NO_OF_CORES,walltime=HH:MM:SS
#PBS -q QUEUE_NAME

## Job array request
#PBS -t MIN_ARRAY_NUMBER-MAX_ARRAY_NUMBER
```

```
## designate output and error files
#PBS -e /PATH/TO/DIR/TO/SAVE/STDERR/FILE
#PBS -o /PATH/TO/DIR/TO/SAVE/STDOUT/FILE

# print some details about the job
echo "The Array ID is: ${PBS_ARRAYID}"
echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo PBS job ID is ${PBS_JOBID}
echo This job runs on the following nodes:
echo `cat $PBS_NODEFILE | uniq`

# load required modules
# e.g. module load apps/matlab-r2013a
echo "Modules loaded:"
module list

# code to be executed for each array job should go here
```

BG is the BrisSynBio HPC cluster at UoB and C3DDB is a commercial HPC cluster that is rented by Lu Lab, MIT. BG runs on Scientific Linux release 6.6 (Carbon) and uses the SLURM 14.03.0 and C3DDB runs on Red Hat Enterprise Linux Server release 6.6 (Santiago) using SLURM 14.11.4. The following is a template for a typical submission script for a job array on BG or C3DDB.

```
#!/bin/bash -login

# COMMENTS

## Job name
#SBATCH --job-name=NAME_OF_JOB

## What account the simulations are registered to
#SBATCH -A ACCOUNT_NAME

## Resource request
#SBATCH --ntasks=NUMBER_OF_TASKS
#SBATCH --time=D-HH:MM:SS
#SBATCH -p QUEUE_NAME
```

```
## Job array request
#SBATCH --array=MIN_ARRAY_NUMBER-MAX_ARRAY_NUMBER

## designate output and error files
#SBATCH --output=/PATH/AND/NAME/TO/SAVE/STDOUT.OUT
#SBATCH --error=/PATH/AND/NAME/TO/SAVE/STDERR.ERR

# print some details about the job
echo "The Array TASK ID is: ${SLURM_ARRAY_TASK_ID}"
echo "The Array JOB ID is: ${SLURM_ARRAY_JOB_ID}"
echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`

# load required modules
# e.g. module load apps/matlab-r2013a
echo "Modules loaded:"
module list

# code to be executed for each array job should go here
```

Having multiple computers to work means that remotely connecting to them is useful. All remote connections are made with SSH which is a cryptographic protocol for securely connecting computers over an unsecured connection.

It is common to transfer data between computers using `scp` which is a modification of the `cp` function to copy over SSH. However, when transferring large amounts of data `scp` is prone to stop due to connection errors. Since we are working with large amounts of data we will use `rsync` by default which is an improved version of `scp` (for more information on `rsync` see the official website [64]).

Most of the development code in this thesis was programmed in Python3.5 which is an open-source, general-purpose, interpreted programming language [65]. One of the goals of this thesis was to develop tools that could be used on new whole-cell models as and when they became available. To this end, a lot of the code follows the object-oriented paradigm [66, 67].

The class structure of the object-oriented code will be visualised using UML diagrams, which are automatically created by PyReverse which comes as part of PyLint (for more information see the

official website [68]).

Parallel computing is used in multiple different ways. On the level of a cluster queue manager, job arrays are used to submit multiple jobs simultaneously. On the level of Bash, the `&` command is used to run multiple processes in the background together. In Matlab, the `par` command is used to parallelise loops, and in Python the `multiprocessing` (if child processes do not need to spawn child processes) and `concurrent.futures` (if child processes do need to spawn child processes) libraries are used to parallelise tasks.

2.2.2 Data storage

SQLite3 is a lightweight RDMS that was used to prototype data storage solutions (for more information see the official website [69]). Python libraries were created to interface with SQLite3 databases using the `sqlite3` Python module.

Database schema is visualised using SchemaCrawler (for more information see the official website [70]).

The whole-cell model of *M. genitalium* automatically stores simulation data in compressed Matlab files [61].

When using Python data can be stored as CSV files but is normally stored as Pickles. Pickleing and unPickleing is a way of serialising and de-serialising Python objects and is often used to store/transfer data and objects. Pickle is part of the Python standard library and so is referenced to Python [65].

2.2.3 Analysis

Python has many standard data structures like lists, sets and dictionaries [65]. Numpy is a library built on top of this to improve linear algebra and other array processing tasks. Pandas is a library built on top of Numpy to give a higher level data manipulation experience plus many additions like useful manipulation, analytical, and plotting methods [71]. Python analysis in this thesis was normally done in Pandas DataFrames or standard data structures but also occasionally in Numpy [72].

The adjusted rand index (ARI) is often used in machine learning to compare clustering [73, 74]. This thesis calculates the ARI using the scikit-learn library in Python.

A distance matrix is a matrix that contains all the distances between pairs of objects. A distance is a function on a set, S, $\delta : S \times S \rightarrow [0, \infty) \in \mathbb{R}$ where the following conditions hold for all $x, y, z \in S$

1. $\delta(x,y) \geq 0$
2. $\delta(x,y) = 0 \iff x = y$
3. $\delta(x,y) = \delta(y,x)$
4. $\delta(x,y) \leq \delta(x,z) + \delta(z,y)$.

The distance matrix of a set of objects $S = \{s_1,\ldots,s_N | N \in \mathbb{N}\}$ is defined as $D = [d_{i,j}]$ where $d_{i,j} = \delta(s_i,s_j)$ for all $s_i,s_j \in S$. So the distance matrix of two 2D points, $p_1 = (1,2)$ and $p_2 = (7,3)$ using Euclidean distance as the distance measure results in the following distance matrix

$$D = \begin{bmatrix} \delta(p_1,p_1) & \delta(p_1,p_2) \\ \delta(p_2,p_1) & \delta(p_2,p_2) \end{bmatrix} = \begin{bmatrix} \sqrt{(1-1)^2+(2-2)^2} & \sqrt{(1-7)^2+(2-3)^2} \\ \sqrt{(7-1)^2+(3-2)^2} & \sqrt{(7-7)^2+(3-3)^2} \end{bmatrix} = \begin{bmatrix} 0 & \sqrt{37} \\ \sqrt{37} & 0 \end{bmatrix}.$$

Note that only the positive square roots were taken due to the property of distance being positive (see the list of properties of distance functions 2.2.3). More information can be found on the concepts of distance and matrices from most undergraduate textbooks on linear algebra [75].

Cytoscape is an application designed to visualise networks [76]. It has the ability to be automated, and third parties can create and publish plugins that others can easily use. Cytoscape is used heavily in the bioinformatics community and so has many plugins specialised to biological visualisation.

Scikit-learn is a Python library that creates a framework to train and test machine learning models and other machine learning related tasks [77].

PCA is an unsupervised machine learning algorithm used to reduce the dimensions of a dataset [78]. It is often used to visualise distance matrices that have 3 or more dimensions. PCA looks at how much effect each dimension has on the linearised variance of the dataset and attempts to compress as much of this variance into a user-specified number of dimensions. It is known as a lossy compression algorithm since information can be lost in the process. All PCA performed in this thesis was done with Pythons scikit-learn.

Analysis of data that is on different orders of magnitude can result in skewed results due to some parts of the data being significantly larger than the other and thus can lose quantitative and qualitative effects of the dataset. Multiple methods in the scikit-learn library explicitly require the data to be standardised. In order to normalise the scales whilst still retaining the shape of the data, it is common in statistics and machine learning to standardise the data. Standardising data transforms the data, $\underline{x} = [x_1,x_2,\ldots,x_N] \rightarrow \underline{z} = [z_1,z_2,\ldots,z_N]$, such that its

mean and standard deviation is $(\mu, \sigma) = (0, 1)$ by using the equation

$$z_i = \frac{x_i - \mu}{\sigma}.$$

All standardisation in this thesis is done in Python using the `sklearn.preprocessing.StandardScalar` method from scikit-learn.

Clustering is an area of unsupervised machine learning that involves organising data into clusters [79]. DBSCAN is a method in Python's scikit-learn library that automatically calculates the number of clusters in data and clusters all data points using density-based clustering.

All visualisations in this thesis from Python are made using matplotlib which is a library that enables the creation of visualisations [80]. Often this is used in combination with seaborn which is a library that attempts to make it easier for the user to create professional-looking visualisations from matplotlib [81].

A dendrogram is a plot that tries to illustrate clusters found by hierarchical clustering [82]. Dendrograms in this thesis are created with scikit-learn and matplotlib.

2.3 Bioinformatics

The GO resource [83, 84] aims to collate machine friendly data on genes, their products and functions. GO is a framework to describe biology and annotations are specific instances of the use of this structure.

KEGG [85–87] is a collection of databases related to biological knowledge on all scales from molecular to organism level and include things like biochemical reactions, genes/genomes, pathways, molecules, and organisms etc — for further reading please see [88].

The NCBI create, host and/or publish databases and tools related to biology [89].

BLAST is a series of tools by NCBI that compare DNA, RNA, or protein sequences to database sequences and calculates the statistical significance of their similarity [90, 91]. BLAST is often used to connect sequences with related resources like GO annotations or KEGG maps.

ClueGO is a Cytoscape plugin [92] that creates functional networks from lists of gene codes using resources like GO annotations and KEGG maps.

CHAPTER 3

INITIAL INVESTIGATION

An initial investigation into using the whole-cell model for genome design was performed in order to assess the situation and make a plan of action to achieve our stated goals.

Four tests were planned: (1) run 200 wild-type simulations, (2) run all single gene knockouts, (3) test some minimal genome predictions, and (4) run a basic genetic algorithm to try to reduce the genome and compare it to one that randomly guesses.

3.1 Initial tests

3.1.1 Wild-type simulations

It is possible to run a single life cycle of a cell using the *M. genitalium* whole-cell model on a standard PC. However, it drains all the resources of the computer and takes around a day to run

Wild type comparisons				
Statistic	Karr et al.		Chalkley	
	μ	σ	μ	σ
Life Cycle (h)	9.2820	0.8757	8.6927	0.9203
Growth (fgh^{-1})	1.0892	0.2198	1.0582	0.2427
Mass Doubling Time (h)	8.9357	0.7874	8.4575	0.8236
ATP Synthesis (s^{-1})	1426	592	1362	607

Table 3.1: Comparison of statistics between Karr and Chalkley wild type simulations where traditional statistical notation is used to describe the mean as μ, and the standard deviation as σ. Standard unit abbreviations are used where h is hours, fg is femtograms, and s is seconds.

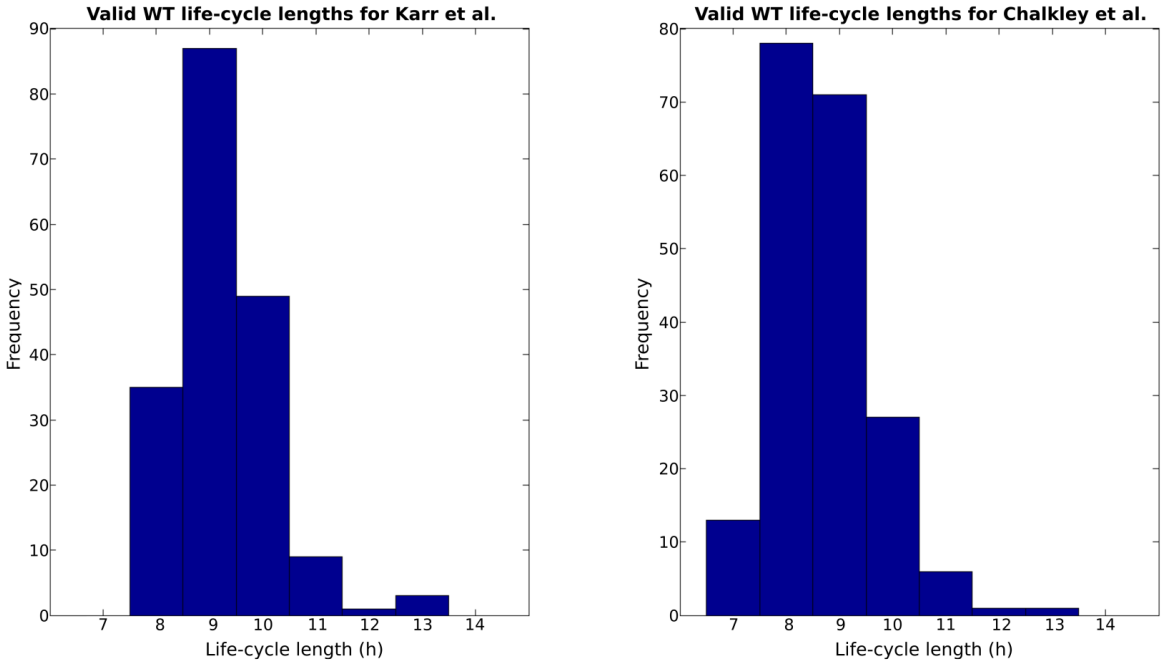


Figure 3.1: Histograms of cell life cycle lengths of thesis results (right) compared to Karr et al.[52] (left)

a single life cycle. Due to the stochastic nature of the model, each simulation needs to be repeated multiple times and so even the simplest experiments start to require days or even weeks - see section 1.1.3 for more information about the model and section 2.1 for more information on how the model was run. In order to reduce the simulation time of experiments it was decided to use the ACRCs HPC facilities at the UoB. Two hundred wild-type simulations were run on BC3 in order to learn how to do it and to check to see if the results are consistent with published results. BC3 is a HPC cluster with 3,568 2.6 GHz cores with 4GB per core - a general overview of the ACRCs facilities will be discussed in section 3.2.2

The 200 wild-type simulations were run so that comparisons could be made against the wild-type data from Karr et al.[52] which consisted of 192 simulations. The results can be seen in table 3.1. It should be noted that the simulations automatically stop if the cell has not divided within 50,000 seconds. This means that there is not a valid life cycle length for organisms that did not divide and so these simulations are removed from the data as done by Karr et al. Table 3.1 compares the mean (μ) and the standard deviation (σ) of the duration of the life cycle, growth rate, mass doubling time, and ATP synthesis between our 200 wild-type simulations and the 192 wild-type simulations published by Karr et al.[52]. One can see that all results except for life cycle length made an excellent match. Whilst the life cycle length was still a good match, the distributions were compared to check there was nothing unexpected happening. Figure 3.1 shows histograms of both sets of data and the distributions look sufficiently similar to satisfy concerns

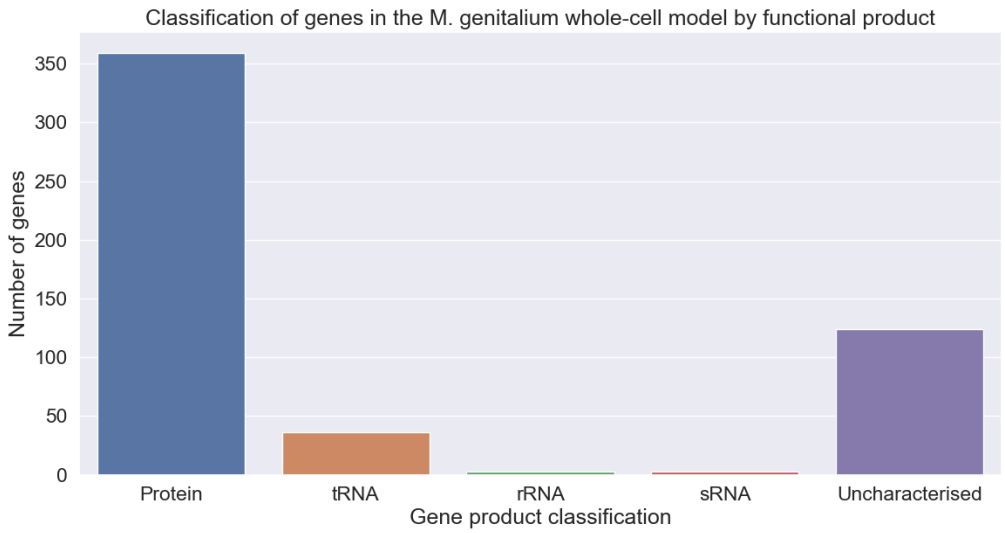


Figure 3.2: A bar chart of all the genes in the *M. genitalium* whole-cell model by functional product. Of the 525 genes in the model 359 code for proteins with known function, 36 code for tRNAs, 3 code for rRNAs, 3 code for sRNAs, and 124 code for proteins with unknown function.

of divergence between the two datasets.

3.1.2 Single-gene knockout simulations

In order to learn how to perform single gene knockouts in the model all possible single gene knockouts were performed on the cluster. This was part of the learning process but later Joshua Rees compared results of single gene knockouts to published results (this will be discussed in more detail later in Section 3.1.4). This experience highlighted the fact that there were three main classes of genes in the model, protein-coding genes, RNA coding genes, and uncharacterised genes. Figure 3.2 shows the number of genes dedicated these three categories. Of the 525 genes in *M. genitalium* the whole-cell model has 359 characterised protein-coding genes, 42 RNA-coding genes, and 124 uncharacterised protein-coding genes. The RNA-coding genes are made up of 36 tRNAs, 3 rRNAs, and 3 sRNAs. tRNA, and rRNA are used in translation and sRNA are used in regulating expression. The uncharacterised genes are protein-coding genes and are still coded into the genome. Their products are transcribed and translated but their products have no functional effects in the cell. For this reason, uncharacterised genes were removed as potential genes for deletion.

It is worth noting that the characterised proteins are not all characterised to the highest biological standards. For example, UniProt characterises less than 401 genes in *M. genitalium* (this will be discussed in more detail later in the section about Joshua Rees' work 3.1.4). For example, constructing a constraint-based model of the metabolism of *M. genitalium* from genomic

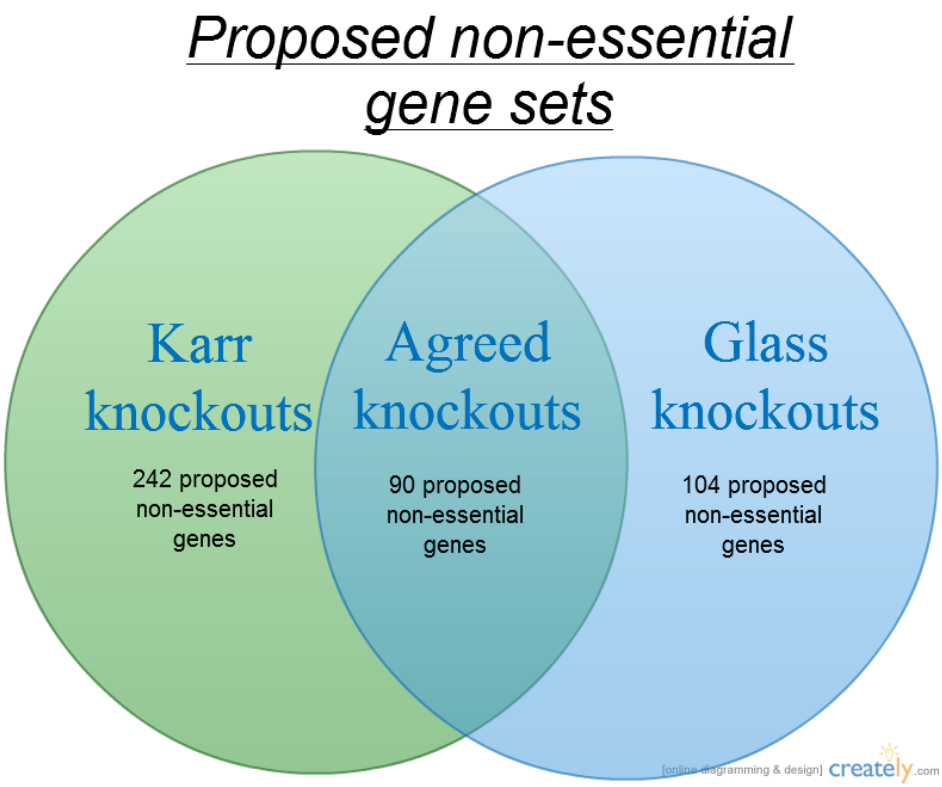


Figure 3.3: A venn diagram of the three *singularly non-essential gene sets*. The Karr and Glass sets, knockout 242 and 104 genes, respectively and both sets agree that 90 of those genes are non-essential.

data results in a network that cannot sustain a cell. In order to get around this problem Karr et al. looked at uncharacterised genes that partially match enzymes that are needed for the *M. genitalium* metabolism to function[52]. Reducing the requirement to classify gene function will mean that there is more chance of misclassified genes in the model and may explain some incorrectly classified single gene knockouts.

The next test was to try simulating some minimal genome predictions. Knocking out all singularly non-essential genes (i.e. removing the gene on its own does not kill the cell see chapter 1 for more details) has been suggested as a potential minimal genome[29] and so this was simulated for the set of singularly non-essential genes proposed by Glass et al.[29] and Karr et al.[52] — this is referred to as the *Glass set* and the *Karr set*. A third set was constructed by taking all the genes that both Glass and Karr classify as singularly non-essential — this is referred to as the *Agreed set*. Figure 3.3 shows a Venn diagram of the three sets of non-essential genes and it can be seen the Karr set is the smallest genome with 242 genes knocked-out (118 after removing the 124 uncharacterised genes), the Glass set has 104 genes knocked-out and the Agreed set knocks-out 90 genes. Collectively we refer to them as the *singularly non-essential*

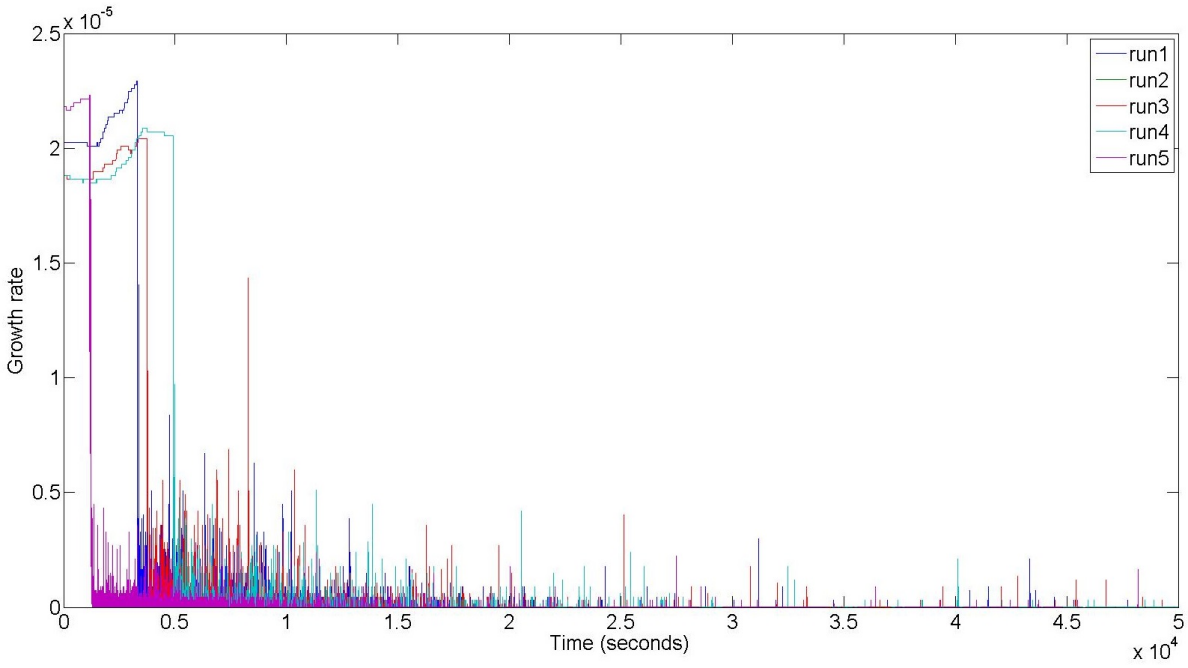


Figure 3.4: Time series of growth rates of the five runs of "Agreed knockouts"

minimal genome predictions.

All the genes from each of the three sets of singularly non-essential genes were knocked-out simultaneously and simulated using the whole-cell model of *M. genitalium* — this was repeated 5 times for each set to take into account the stochastic nature of the model. It was found that none of the 15 simulations produced a cell that divided. The Karr and the Glass knockout sets both produced a zero growth rate from start to finish but interestingly the Agreed knockout set showed a short spell of normal growth at the beginning of the simulation which then rapidly tended to zero (see Figure 3.4). One can see that even once the decline in growth rate starts there are still fluctuations of growth which start quite large but slowly decline with time. The initial conditions are randomised at the beginning of the simulation and so a cell always starts with a reasonable amount of resources other than any RNAs, proteins or macromolecular complexes that have had their corresponding genes deleted. The fact that the Karr and Glass knockouts showed no signs of life from beginning to end suggests that those gene sets miss so many functions that there are very few, if any, working biological processes going on inside the cell. Conversely, the normal initial growth, followed by "splutters" of growth from the Agreed knockout set suggest that there are enough genes to facilitate growth but there are some molecules that are not being produced by the cell that is available in the cell at the beginning of the simulation due to initial conditions, however, the cell consumes these molecules until they run out and then the cell dies. An example of this would be knocking-out a gene that codes for an enzyme that catalyses a reaction that produces an essential metabolite. When the simulation starts the counts of the enzyme will be

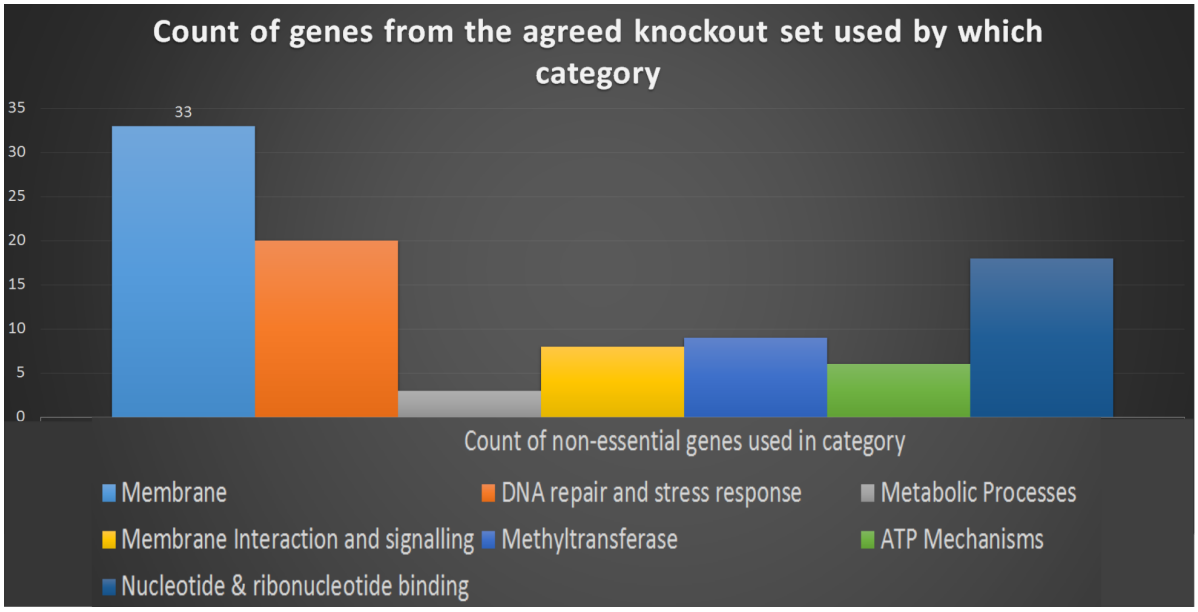


Figure 3.5: A bar chart of the number of genes from the Agreed knockout set that are used by what functional category

set to zero and so none of the essential metabolite will be produced but the counts of the essential metabolite will be randomly assigned at the beginning and so the cell will not die until all the metabolite is consumed. Since all the genes are singularly non-essential and do not produce a dividing cell the Agreed set is our first example of a low-essential combination, proving that the model is capable of simulating low-essentiality. It should be noted that essentiality must be considered relative to the experiment (this case is relative to the whole-cell model) which means that both the Karr knockout set and the Agreed knockout set can be considered as low-essential sets. The Glass set, however, contains genes that are singularly non-essential *in-vivo* but essential in this model and so cannot be classed as a low-essential set relative to the model.

An attempt was made to understand what was being removed from the Agreed knockout set biologically. Figure 3.5 shows how many genes from the Agreed knockout set were used for specific functions using the DAVID Bioinformatics Resource [93]. It can be seen that most of the disrupted genes are used in the membrane, DNA repair and stress response, and nucleotide and ribonucleotide binding. The smaller groups are methyltransferase, membrane interaction and signalling, ATP mechanisms, and metabolic processes.

Of the 90 Agreed knockouts, 4 are unknown in the model and by DAVID and several were known in the model but uncategorised by DAVID. There could be two reasons for the discrepancy. Firstly, online public databases are updated periodically and so can be relatively out of date and secondly, Karr et al. inferred the function of some unknown genes when building the model (as

discussed earlier in this section).

3.1.3 Preliminary algorithms for genome reduction

In order to get an idea of the challenges involved in developing algorithms a basic attempt to implement proof-of-concept genome reduction algorithms was performed. It was decided to assess the performance of basic use of biological knowledge and machine learning to reduce the whole-cell model of *M. genitalium*.

The genetic algorithm is a common general-purpose machine learning algorithm that attempts to mimic evolution by natural selection to optimise some objective function. It normally starts at generation 0 with random guessing. Then all the children a ranked by some objective function and the best are allowed to *mate* to produce a new generation of children that are made up of a random combination of the genes of both parents plus some random mutation. The children are then ranked alongside the fittest of the previous generation and then a new set of the fittest individuals is created to be the parents of the new generation. This is repeated until some terminating event happens (e.g. a maximum number of simulations or no improvement made over a certain number of generations[94]). This is referred to as *GA*. Genetic algorithms have been used in the past to design *in-silico* genomes of genome-scale metabolic models[95] (see Section 1.1.3 for more information on biological models) and so it was decided to attempt genome reduction of the whole-cell model of *M. genitalium* with a genetic algorithm.

Since the Agreed knockout set looked close to a viable combination so some inspiration was taken for the analysis discussed previously in the section (see figure 3.5). It was noticed that the genes seemed spread over categories such that categories with more genes have more genes knocked-out than ones with fewer genes. Additionally, the genes knocked-out were more likely to be involved in only one category. In order to test these observations, two genome reduction algorithms were proposed. One randomly picked genes such that genes in categories involving large numbers of genes are more likely to be picked than categories involving small amounts of genes. The other algorithm randomly picks genes to knockout such that genes that are involved in more different categories are less likely to be picked than genes involved in fewer categories. These are called the *AOL* and the *conn* algorithm, respectively.

A final algorithm of randomly guessing was implemented in order to have a bench-mark to compare the algorithms against. In order to get an idea of the probability of guessing a viable gene combination, an initial investigation of the solution space was performed. There is no such thing as order of gene deletion in the whole-cell model of *M. genitalium* and so the amount of unordered k-combinations of the 401 characterised genes is $\binom{401}{k}$. The most recent minimal genome predictions from JCVI are from Glass et al. [24], that suggest a reduction of *M. genitalium*

Figure 3.6: A comparison of four different genome reduction algorithms. The x-axis shows the number gene knockouts, the y-axis shows the percentage of viable gene knockouts found by the algorithm and the colour represents which algorithms were used. Random: random guess. AOL: Avoid overloading any functions with knockouts. Conn: Avoid highly connected genes. GA: the first generation of a genetic algorithm which was seeded by the viable sets found by Random. Random, AOL, and Conn only look for gene knockout sets of between 2 and 5 genes but this is not controllable for GA. GA attempted 7KO, 8KO, and 9KO sets but did not find any viable sets.

of around 100 genes. If one assumes that there is only one minimal genome and that there is a linear decay[①] to that point from the 118 singularly non-essential genes in the model then one can calculate the size of the solution space, estimate the number of viable combinations and thus the probability of randomly guessing a viable combination for each size knockout. Since the number of viable combinations linearly decreases whilst the potential solution space increases combinatorially it is expected that the probability of randomly picking a viable combination to rapidly diminish to zero. Using the linear model, $y = -1.18x + 119.18$ (constants to 2 d.p.) to estimate the number of viable combinations, y, given the size of the knockout set, x and $\binom{401}{x}$ to calculate the size of the solution space at that knockout size it can be seen that the probability of randomly picking a viable 1-gene, 2-gene, 5-gene, and 10-gene knockout set is $\sim 29\%$, $\sim 0.15\%$, $\sim 1 \times 10^{-7}\%$, $\sim 4 \times 10^{-16}\%$, respectively. Therefore, if the random algorithm were to randomly pick a knockout size from a uniform distribution and then pick a random combination uniformly then it would be highly unlikely to find a viable combination in a reasonable amount of simulations. For this reason, the random algorithm was set to only look for gene knockout sets between 2 and 5 genes. The same thing was done for the AOL and Conn but could not be done for the genetic algorithm since the knockout set size is determined by *evolution*.

[①] We actually expect the number of viable combinations to increase and then decrease rapidly but do not have data to estimate the position of the maximum and so use linear regression to keep things simple.

More details can be seen on the implementation of the algorithms in the appendix section A.1. All four algorithms were tested. The three algorithms that only have one generation were run so that roughly 200-gene knockouts were simulated for each knockout size resulting in 800 simulations. The ga algorithm ran 800 random simulations for generation 0 and then 200 simulations per generation thereafter. Due to the length of simulation time generation 0 used the results from the random algorithm which produced 36 viable genomes. Figure 3.6 compares the first generation of GA to the results of AOL, Conn and the random algorithm. The Conn algorithm performs slightly worse than randomly guessing which might suggest that highly connected genes should be targeted rather than avoided. The AOL algorithm performs significantly better than randomly guessing but by 5-gene knockouts any advantage is overshadowed by the explosion in the size of the solution space. So spreading knockouts over functions appears to be a good strategy from genome reduction although it is probably not powerful enough on its own to find the minimal genome. The genetic algorithm massively outperformed the other algorithms in every knockout size with 2-4-gene knockouts over 70% and 5-gene knockouts managing over 40% success rate. The algorithm also went further and found 6-gene knockouts and whilst it attempted some 7-9-gene knockouts it was unable to find any viable ones. Whilst the genetic algorithm clearly shows superior results, it should be noted that generation 1 would not have been possible without generation 0 which was taken from the random algorithms results and so is not a completely fair comparison. It does, however, highlight the main advantage of the genetic algorithm which is its ability to learn. Whilst the genetic algorithm is able to use memory to iteratively improve results the other algorithms are stuck with a static concept that cannot learn or adapt with new information.

Figure 3.7 takes a look at all the generations performed by the genetic algorithm and it can be seen that the algorithm was able to discover viable 10-gene knockout combinations. Earlier in this section we estimated the probability of randomly guessing a viable 10-gene knockout combination to be $\sim 4 \times 10^{-16}\%$ which as a probability is $\sim 4 \times 10^{-18}$ and so there is a $\sim 0\%$ chance of randomly guessing at least one viable 10-gene knockout combination in 2000 trials[②]. Whilst the general trend is to find larger and larger knockout sets one can see that a new generation does not guarantee a larger viable knockout set and in fact, the first viable 10-gene knockout set was found in generation 3. The algorithm reduced the genome by 10 genes in 7 generations (including the generation 0) giving a reduction rate of $\frac{10}{6} = \frac{5}{3}$ genes per generation. If one only counts generations until the first instance of the maximum generation (i.e. the first viable 10-gene knockout set was found in generation 3), then it reduced the genome by $\frac{10}{4} = \frac{5}{2}$ genes per

[②]Whilst this probability is strictly greater than zero, it is so small that Python returned exactly 0 due to rounding errors caused by the limitations of standard floating-point arithmetic in computers. Assuming a binomial probability distribution then the probability of finding no 10-gene knockouts in 2,000 trials is $\binom{2000}{0} \times (4 \times 10^{-18})^0 \times (1 - 4 \times 10^{-18})^{2000} \approx 0$.

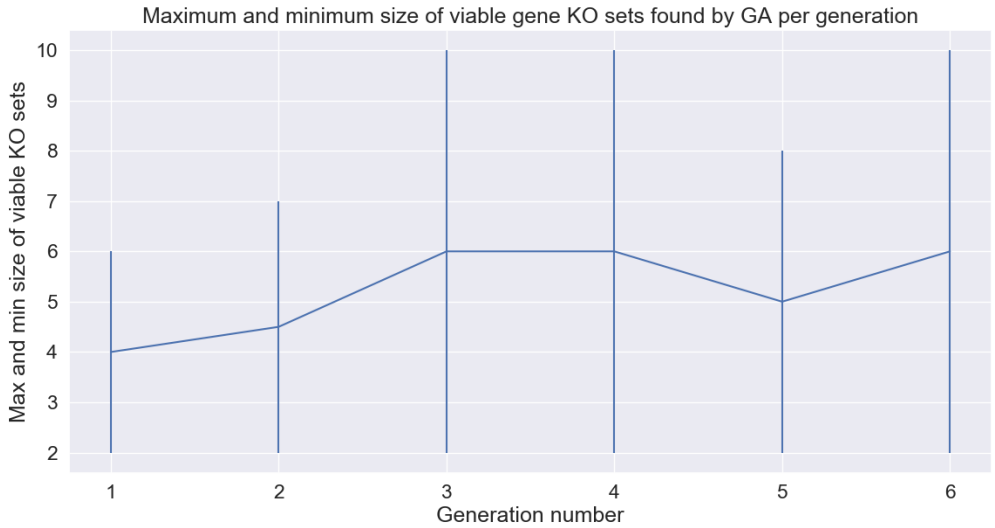


Figure 3.7: The per generation progression of the genetic algorithm where the top of each bar shows the maximum size of viable gene knockouts found and the bottom of the bar shows the minimum size of viable gene knockouts found for that generation. The point that connects each bar is simply the midpoint between the maximum and minimum.

generation. Furthermore, if generation 0 is not included then these statistics become, $\frac{5}{6}$ and $\frac{5}{3}$, respectively.

3.1.4 Joshua Rees

Joshua Rees is a PhD student who joined the genome design group after the initial work described in this section. Joshua and I worked closely in some areas and tried to keep our results comparable and so our contributions are intertwined. As a result, this section is dedicated to some of the work done by Joshua that will need to be referenced at some point in the thesis. After sharing my code and teaching Rees how to run gene knockout experiments on BC3 and BG he set out to extend my tests of theoretical predictions. It was found that minimal genome predictions were normally made on protein-coding genes and so it was decided to reduce our search space from 401 characterised genes to 359 characterised protein-coding genes (i.e. 42 RNA-coding genes were excluded from the set of genes to potentially knockout). It was found that one protein-coding gene MG_469 often crashed the simulation when knocked-out and so was removed from my search set, however, was left in Rees' set. Rees then performed single gene knockouts that were mostly in agreement with Karr et al.[52]. The disagreements looked like they were caused by single gene knockouts that produced inconsistent phenotypes; for a spreadsheet of the data please see section A.4. Rees then collected 13 minimal genome predictions from the literature and each set will be named after the lead author of the paper that the prediction was published in (small caps are used to signify the difference between my initial tests and Rees's more extensive

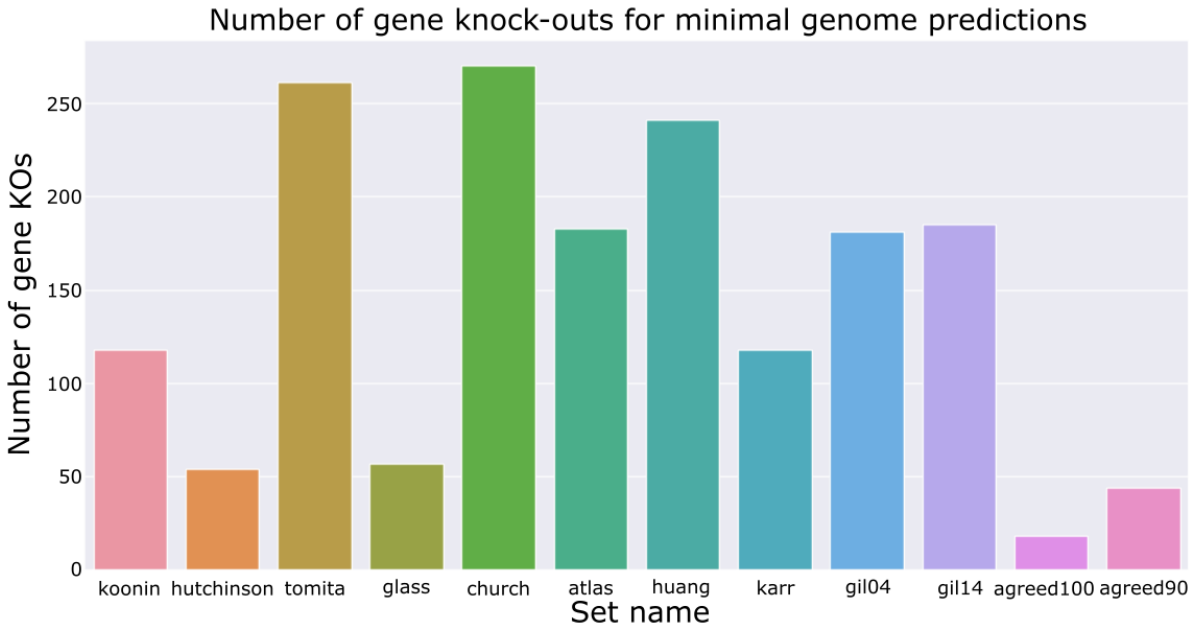


Figure 3.8: A bar chart of the number of gene knockouts required to test each of the minimal gene set predictions.

tests). The names of the sets with references to publications are the koonin[96], hutchinson[97], tomita[96], glass[29], church[98], atlas[99], huang[99], karr[52], gil04[100], and the gil 2014[99] sets. In addition to this, he created the agreed100 set and the agreed90 set which used the same concept as my agreed set except it took into account all the new predictions and only used the characterised protein-coding genes - the 100 and 90 are for 100% or 90% agreement, respectively. In the agreed100 set every prediction has to agree that a gene is non-essential to be knocked-out, in the latter set only 90% of the predictions have to agree that a gene is non-essential to knock it out.

Figure 3.8 shows that the number of gene knockouts predicted by each set varies greatly. The tomita, church, and huang sets had the most with 230 - 270-gene knockouts. There were between 170 - 190 in the atlas, gil04, and gil14 sets. Additionally, the karr and koonin sets have between 120 and 130-gene knockouts. Three of the smallest sets were the hutchison, glass, and agreed90 sets which have between 40 - 60-gene knockouts. However, the smallest number of knockouts required is the agreed100 set with only 18.

Rees, simulated all these sets and found that none produced a viable cell and is currently writing a paper with all the work we have done on the historical predictions - I will be second author on the paper.

Although I had already classified the genes of *M. genitalium* using DAVID ontology GO terms it

was done under standard conditions using multiple databases. Rees reclassified them using only the UniProt database as it has the most prestige. This thesis refers to my classifications unless otherwise stated. In order to get more information of the classifications please see section 6.1.

3.2 Knowledge consolidation

The initial tests performed in section 3.1 illustrated that traditional computational tools would not be able to cope with the large amount of data and the long-running time of the *M. genitalium* whole-cell model. The next section is dedicated to trying to define the problems and matching them to our resources with a plan of how to make them work together.

3.2.1 Estimating resource usage

It is impossible to know our exact resource usage *a priori* to doing the research and so first an upper and lower bound was proposed as the best case and worst-case scenario.

The only known way to guarantee that the minimal genome is found in a finite amount of time is to try every combination of gene knockouts and record the smallest viable genome (or equivalently the largest gene knockout set). This is known as a *brute force* method and will be our worst-case scenario. The whole-cell model of *M. genitalium* has no *order* of gene deletion which means that the amount of simulations is the number of unordered combinations of 401 genes. So the upper bound on the number of simulations needed is

$$N_{upper} = \sum_{x=1}^{401} \left[\binom{401}{x} \right] = \sim 5.164 \times 10^{120}.$$

At 200 simulations a generation it would take $\sim 2.582 \times 10^{118}$ generations to calculate every single gene combination.

The initial tests in section 3.1 showed that the genetic algorithm reduced the genome by $\frac{5}{6}$ genes per generation. It is worth noting that multiple statistics were provided but the slowest statistic was picked because the convergence to the minimal genome is likely to be logarithmic since it becomes harder and harder to find smaller genomes as there are less and less to find and so this early sample is likely to overestimate the overall speed of convergence (i.e. we have assumed a linear rather than logarithmic convergence). For this reason, the slowest rate was picked to reduce the overestimation. Additionally, the majority of the algorithm is generation 1 to the optimal solution and so generation 0 is not very representative of the overall convergence rate. The most recent minimal genome predictions from Glass et al. at JCVI estimate that one would need to knockout around 100 genes from *M. genitalium* to produce a minimal cell. Using this prediction it was estimated that the genetic algorithm would need ~ 120[③] generations to

[③] $100 \times \frac{6}{5} = 120$

find the minimal genome. 200 simulations per generation result in a total of $N_{lower} = \sim 24,000$ simulations required to find the minimal gene set as our lower bound.

The average size of a wild-type simulation was estimated at 239MBs (see section A.6) and so the upper and lower bounds of our storage requirements are $S_{lower} = 5.736$TBs and $S_{upper} = \sim 1.239 \times 10^{119}$TBs.

Initial development was performed on BC3 and so all data relates to BC3, however it should be noted that later it was found out that BG significantly outperforms BC3 in computation time.

The quickest simulation ran in about 5 hours and the walltime is set to 35 hours. On occasion, a simulation can take longer than 35 hours but since the standard walltime for a script is 35 hours it is not possible to know how much longer those simulations would go on for since the simulation is automatically cancelled at 35 hours. Since the length of a generation is determined by the longest simulation, it was decided that it is rare enough that the loss of information is less important than the increase in generation time. It is also worth noting that the only other thing that effects generation time is how busy the queue is but it is not easy to account for this without doing a very complex analysis of queuing times[④] and is excluded from our estimate. Since 200 simulations normally resulted in at least one simulation taking close to 35 hours[⑤], the estimated time of a generation is 35 hours.

Having calculated the generation time and the number of generations it is easy to see that the lower bound on simulation time to find the minimal genome is $T_{lower} = 35 \times 120 = 4,200$ hours which is just under 6 months. The upper bound is $T_{upper} = 35 \times \sim 2.582 \times 10^{118} = \sim 9.039 \times 10^{119}$ hours which is $\sim 1.031 \times 10^{116}$ years.

3.2.2 Available resources

Section 3.2.1 showed that the lower bound on resources would need TBs data and half a years worth of simulations not including research and development and an upper bound that in practical terms may as well need infinite resources and so efforts were made to maximise the amount of resources at our disposal.

Fortunately access to three high performance computing clusters were obtained as well as 9 TBs of temporary disk space plus 10 TBs of long term back-up. The following describes the resources acquired and credits any people or groups that granted it to us.

④BC3s queuing system takes into account everything that is in the queue at the time, how much resources you and everyone requires. It takes into account how much every individual has been using the cluster and how much each group and department have been using the cluster to ensure fair usage.

⑤This is taken from anecdotal observations.

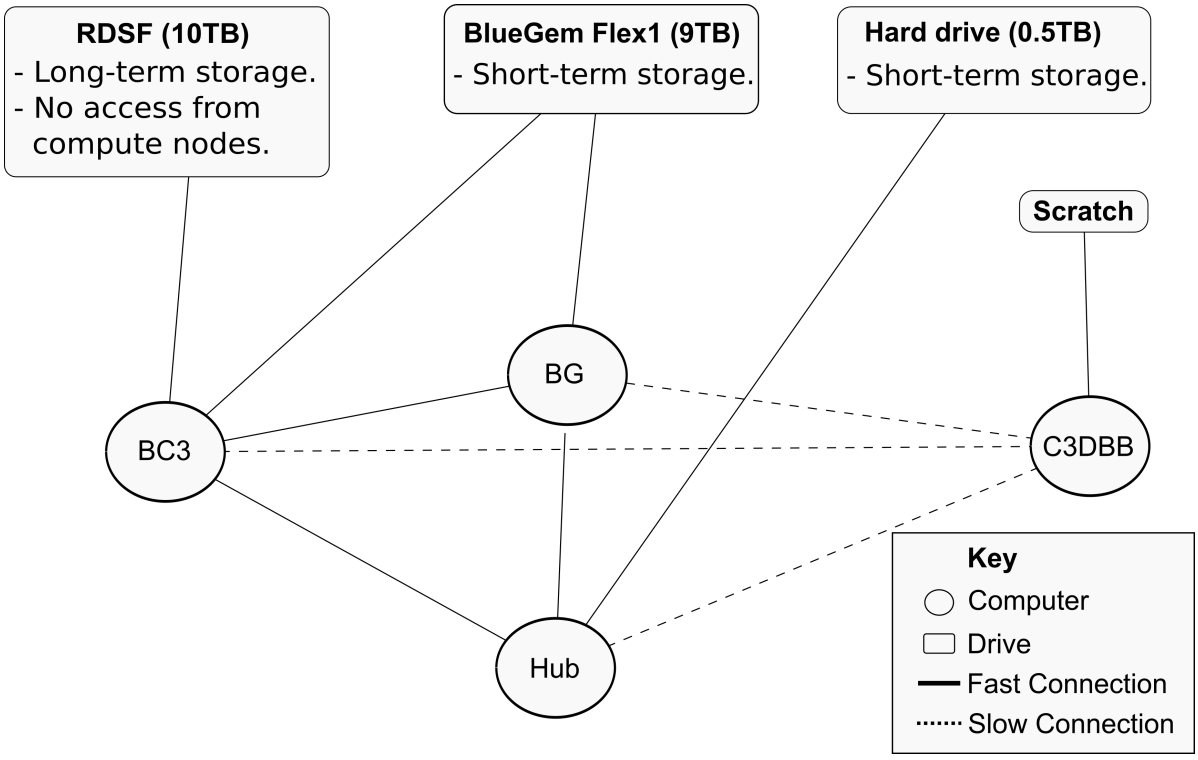


Figure 3.9: Used resources and how they are connected. BC3: BlueCrystal III — HPC Cluster. BG: BlueGem — HPC cluster. C3DDB: Commonwealth Computational Cloud for Data Driven Biology. Hub is an old PC that has been re-purposed to act as a server that controls everything. RDSF is a long term storage facility that is secure and has a distributed backup system — it is for storage only and is only accessible from the BC3 login-nodes. Flex1 is a disk with rapid read/write capabilities for use with HPC cluster compute nodes. It is more resilient to failure than normal disk drives but is not infallible and is also not backed up. In addition, this is officially classed as temporary storage and the 9TBs can be reduced at any time. Hard drive is the standard hard drive that comes with a PC and it belongs to the hub. The fast connections are the Universities fast intranet connections. The slow connections are connections that have to go through the internet.

HPC clusters:

- BlueCrystal Phase III (BC3),ACRC, UoB: 223 nodes with 16 x 2.6 GHz SandyBridge cores, 4GB/core and a 1TB SATA disk. Plus 18 high-memory nodes (256GB RAM), 100 nodes that host dual GPGPUs, and 76 NVIDIA K20 on 38 nodes.

- BlueGem (BG), BrisSynBio, ACRC, UoB: 900 Intel Haswell CPU cores, and 8 nVidia K80 GPUs spread over 53 compute nodes and 4 GPU nodes.

- Commonwealth Computational Cloud for Data Driven Biology (C3DDB): Lu Lab, MIT through our collaborator Oliver Purcell (see section 1.4 for more information on the GDG

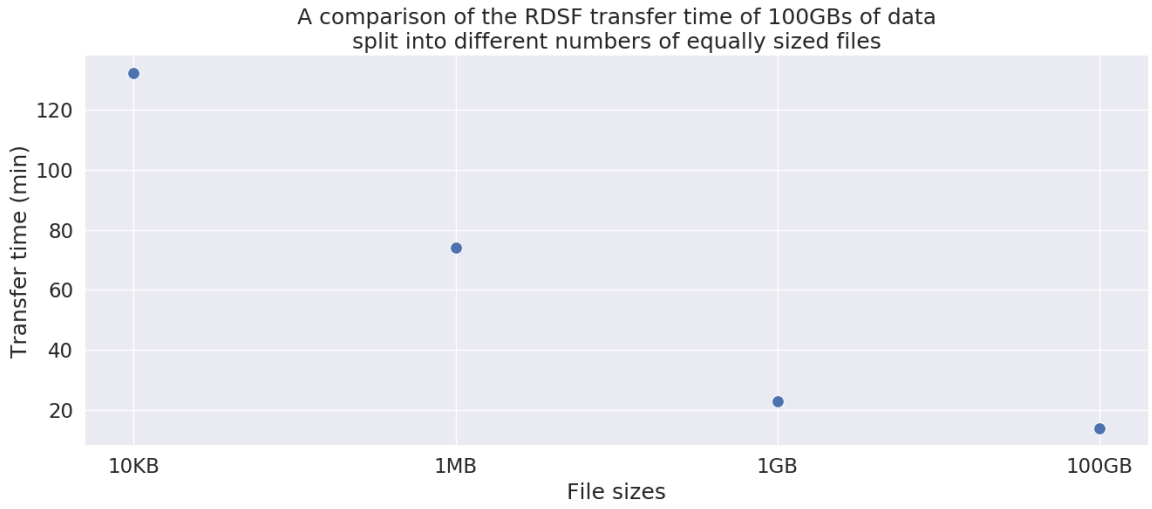


Figure 3.10: RDSF is designed to move bulk data on and off the disk but not for computation. In order to test this 100GBs of data was created in one text file. This was then repeated three more times except with 100GBs of data in multiple 1GB, 1MB, and 10KB files. The data was then transferred from the hub to RDSF and the y-axis gives the amount of time the data took to transfer.

team): 133 compute nodes with a total of 7200 cores with 61 terabytes of main memory, several different high-memory nodes, and 32 nodes dedicated to GPU accelerators.

Personal Computers:

- Hub: BCCS, UoB. This is an old desktop PC which I repurposed as a server that could monitor or control any processes that take longer than what is allowed on a cluster.

Disk drives:

- RDSF (10 TB): ACRC, UoB. This is a disk designed for long term storage only. It is resilient to faults and is backed up in 2 different locations. It does not have a fast read/write speed (relative to disks like flex1) and is only accessible through a BC3 login node (i.e. no compute nodes are allowed to access this drive).
- Flex1 (9 TB): BrisSynBio, ACRC, UoB. This is disk designed for direct contact with HPC cluster compute nodes and has fast read/write speeds. Whilst it is resilient to faults it is not backed up and the administrators only allow it for short term storage. In addition to this, the 9 TB of space can be reduced at any time.
- Scratch (1 TB): MIT. This is disk designed for direct contact with HPC cluster compute nodes and has fast read/write speeds. Whilst it is resilient to faults it is not backed up and the administrators only allow it for short term storage.

Figure 3.9 shows how all these resources are connected. This set-up will require bespoke tools if we are to utilise them to their maximum. It can be seen that the only way to access data from RDSF is to work from BC3. Also, note that all the resources are connected with high speed connections except the C3DDB which is limited by the speed of the internet. There is a very large difference in transfer speeds and so when talking about transferring data in the order of TBs this is a huge disadvantage to using the C3DDB cluster. In addition to this, it does not have direct access to the Flex1 disk drive which is likely to be the location of any databases which should be accessible to all members of the genome design group. All the HPC clusters have maximum simulation times of two months which is likely to cause problems due to our estimated lower bound being almost 6 months.

It should be noted that RDSF is only suitable for long term back-up and so cannot be used to store active data. It is designed to be fast at transferring bulk data and slow at small file read and write. Due to our large storage requirements, it is likely that data will need to be moved around fairly regularly which may have an effect on the way we chose to store data. In order to quantify this effect a test was created where 100GBs of data was created in the form of 1 100GBs files, or multiple 1GB, 1MB, or 10KB files and then transferred from the Hub to RDSFand timed. Figure 3.10 shows the time it took for the 100GBs of data to transfer and it can be seen that transferring large files is much better than transferring lots of small files. However, most of the performance benefit is gained at around GB file sizes.

3.3 Mathematical representations of a genome

Sometimes during this thesis it will be useful to think of the *M. genitalium* genome mathematically. In this section, we define a mathematical formalism with which to think about genomes that will be used throughout the thesis. This section is built from first principles that mostly use set theory that can be found in any introductory textbook for undergraduates. Curly braces, {}, will be used to denote sets. *Such that* and *and* will be denoted with a '|', and a '$\wedge$', respectively, within set generators.

There are 525 genes in *M. genitalium* and so the set of all these genes is $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{525}\}$. γ_i can be any representation of a gene but the simplest and most intuitive representation is a gene code.

An individual can be defined by it's genome, Θ, which is represented as a set of binary variables $\Theta = \{\theta_1, \theta_2, \ldots, \theta_{525}\}$ where each binary variable θ_i is equal to 0 if γ_i is knocked-out and or equal to 1 if γ_i is present in the organism. The wild-type genome is defined as $A = \{\theta_1, \theta_2, \ldots, \theta_{525}\}$ where $\theta_i = 1 \ \forall i$.

Since only one organism is being considered the whole genome, A, remains constant and thus it is possible to define a genome by the genes that are knocked-out, $K = \{\kappa_1, \kappa_2, \ldots, \kappa_N\}$, and the genes that are present, $\Upsilon = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_M\}$, where $\kappa, \upsilon \in \Gamma$. It will always be able to change between the genome, knockout, and present representations. For a given individual Θ_i with corresponding knockouts, K_i, and present genes, Υ_i, then the set of the knockouts combined with the present genes is always the set of all genes, $K_i \cup \Upsilon_i = \Gamma \ \forall i$ and there can never be any genes shared between them, $K_i \cap \Upsilon_i = \emptyset \ \forall i$.

This gives three equivalent representations of an individual, 1. the *genome* representation, Θ, 2. the *knockout* representation, K, and 3. the *present* representation, Υ. Since Γ is constant it is also possible to retrieve the knockout set from the present set of an individual and *vice versa*. In order to switch between the two representations the difference operator is defined as

$$\Delta(\chi) := \Gamma - \chi \tag{3.1}$$

where the knockout and present sets are collectively referred to as *knock sets* or the *knock representations* or symbolically as χ. It's easy to see that $\Delta(K_i) = \Upsilon_i$ and $\Delta(\Upsilon_i) = K_i \ \forall i$.

Furthermore one can convert from genome representation to knock representations. The conversion operator is defined as

$$\omega(\theta_i) := \gamma_i. \tag{3.2}$$

The inverse of this operator depends on whether the gene was present or knocked-out so given the i^{th} gene from individual, Θ_j, then

$$\omega^{-1}(\gamma_i) = \begin{cases} 0, & \forall \gamma_i \in K_j. \\ 1, & \forall \gamma_i \in \Upsilon_j. \end{cases} \tag{3.3}$$

Using the conversion operator the genome representation can be converted into knock representations by defining the transform operator

$$\Omega(\Theta_j) := \begin{cases} K_j = \{\omega(\theta_i) \mid (\theta_i \in \Theta_j) \wedge (\theta_i = 0)\} \\ \Upsilon_j = \{\omega(\theta_i) \mid (\theta_i \in \Theta_j) \wedge (\theta_i = 1)\} \end{cases} \tag{3.4}$$

and *vice versa* with it's inverse

$$\Omega^{-1}(\chi_j) := \{\theta_i = \omega^{-1}(\gamma_i) \mid \gamma_i \in \Gamma\}. \tag{3.5}$$

It can be useful to know how many genes are in a genome or how many genes have been knocked out of a genome. The number of genes in a genome or knocked out of a genome is simply the cardinality of the present and knockout sets, respectively. The number of genes in genome representation is the same as the present representation. Thus we define

$$\begin{aligned} \|K_i\| &= \sum_{\kappa_j \in K_i} [1] \\ \|\Upsilon_i\| &= \sum_{\upsilon_j \in \Upsilon_i} [1] \\ \|\Theta_i\| &= \sum_{k=1}^{525} [\theta_k] \end{aligned} \tag{3.6}$$

where $\|\Theta_i\| = \|\Upsilon_i\|$.

This section should have illustrated the three representations of an organism's genome, then showed that they are equivalent, how to convert between the representations, and how genome size and set cardinality is related and defined.

3.4 Discussion

This chapter has shown preliminary tests done on the whole-cell model of *M. genitalium*. After getting the model working for wild-type and gene knockout experiments Rees and I showed that minimal genome predictions in the literature produced non-viable cells according to the whole-cell model of *M. genitalium*. Some prototype genome reduction algorithms were tested and it showed that whilst biological knowledge and machine learning can help pick viable gene knockout combinations the benefits seem to be overshadowed adverse properties in the solution space. The size of the solution space explodes combinatorially whilst the number of viable genomes reduces as the number of knockouts increase, making it extremely hard to find viable genomes with large numbers of genes knocked-out. Algorithms with memory and learning showed the most promise, as demonstrated by the genetic algorithm and so this was used to estimate the amount of resources that were likely to be needed. The best-case scenario suggested that 6 months of simulation time and TBs of data storage would be needed - not including development time. The worst-case scenario would require an infeasible amount of resources. Realising the scale of the task, time was spent acquiring access to as much computing resources as possible.

It was found that the longest simulations allowed on any of the HPC clusters was only 2 months which would not likely be enough time to find a minimal genome, given our estimations and there might be just enough disk storage for one genome reduction experiment. To overcome these problems it was decided to try and find a way to run the simulations on a cluster but managing them from off the cluster in an automated fashion. The idea of a multi-generation algorithm was utilised such that a standard PC would manage an algorithm and submit simulations to a cluster

one generation at a time so that the results from each generation can be fed back to the PC so that it can learn to improve the next generation. This structure might also enable the integration of bespoke data processing/storage capabilities and the use of multiple clusters. This structure would require a lot of development and so in order to maximise the value created per unit of work the code would need to be easily adaptable to different clusters, models, design objectives and design algorithms.

The final section of the chapter presents a basic mathematical representation of genomes. As the thesis progresses a quantitative representation is useful to explain genome design strategies and processes in the code.

CHAPTER 4

GENOME DESIGN SUITE

In Chapter 3 we showed that the genetic algorithm would need at least half a year of simulation time, the capability of TBs of data storage/processing, and lots of research and development into how to best find a minimal gene set given the combinatorial explosion of gene knockout sets. It was decided to create a framework with which to run simulations that are too big for a cluster and to potentially split it across multiple clusters. This way it is possible to avoid the maximum simulation time on clusters and also increase the number of simulations run per unit time. This framework will enable us to run *in-silico* genome optimisation experiments but will produce large amounts of data and so a serious data storage solution needs to be designed and integrated into our suite of tools. Additionally, analysis and visualisation of such large datasets will be non-trivial, and so tools will need to be developed and integrated into the suite of tools. Nascent fields tend to evolve fast, and due to the scale of the challenge, we do not want to leave ourselves vulnerable to rapid change. In order to progress as the field evolves, our suite of tools should be as general as possible so that they can be easily adapted to different optimisation goals, optimisation algorithms, computer models, and computer clusters. Since the purpose of this suite of tools is to enable the rational design of *in-silico* genomes, it will be called the *genome design suite*. Some concepts in this chapter will be explained mathematically using the mathematical formalism of genomes described in section 3.3.

The purpose of this Chapter is to illustrate the code and the structure of the GDS. Much of the value of the GDS comes from the fact that it is designed to enable massive *in-silico* experiments where it is easy to change the algorithms, models or clusters used. To see how this generalisation is enabled requires an analysis of the code structure and additionally technical details will be discussed. Section 5.1 will discuss the theory behind the algorithms implemented

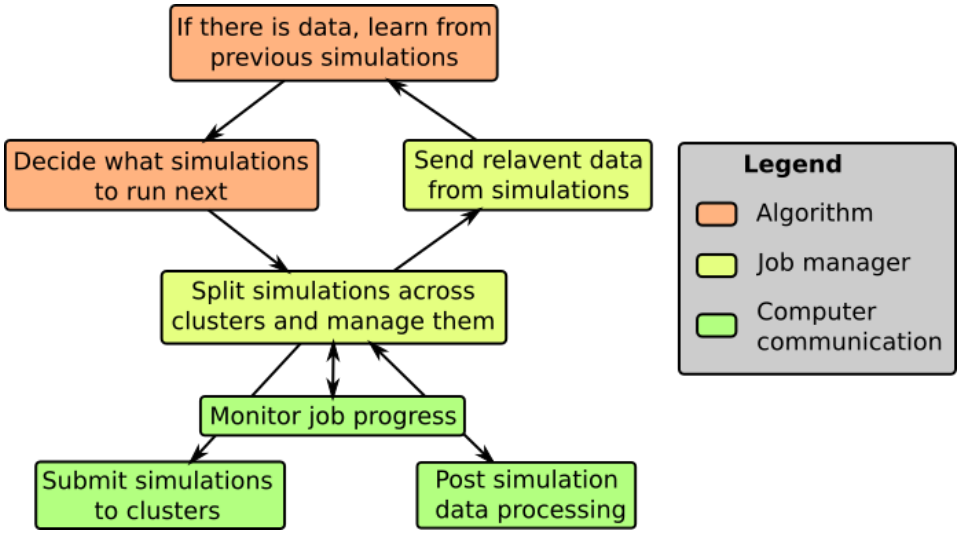


Figure 4.1: This shows how the three fundamental processes interact to design an *in-silico* genome. Process-1 is labelled as 'algorithm', process-2 is labelled as 'job manager', and process-3 is labelled as 'computer communication'.

and their results and so these two chapters are closely related and show complimentary perspectives.

In order to create a suite of libraries to enable *in-silico* genome design three fundamental processes were defined. 1. Decide what simulations to run next in-order to optimise a specified function and learn from previous simulations if there is any. 2. Organise simulations into batches and submit them to the computer cluster(s). Monitor all running jobs and when the jobs are finished perform any necessary tasks like data processing and updating databases. 3. Perform fundamental tasks on a remote computer like creating files, running code, and checking disk usage. Figure 4.1 shows how these processes interact to create an iterative process that learns over time. Furthermore, if coded correctly these fundamental processes can act as an abstract template enabling versatility of algorithms, models and HPC clusters.

These three processes were set as the fundamental sections of the code. It was decided to start with abstract classes as much as possible and use them to create a solid framework so that they can be generalised and adapted for different algorithms, models and computer clusters. However, since most HPC clusters run on Linux which is my personal preference of operating system, an assumption that all computers would run on a Linux operating system was made. The fundamental processes are coded into the following Python modules:

- Process-1, or *algorithms*, is in the `multigeneration_algorithm.py` module.

- Process-2, or *job manager*, is in the `batch_jobs.py` module.
- Process-3, or *computer communication*, is in the `base_connections.py` and `connections.py` modules.

It was decided that abstract classes would be used for the computer communication and the algorithms code. These classes would define the structure of any computer connection and any algorithm. The job manager code would assume that all algorithms and connections would follow the said structure thus making an abstract class unnecessary.

4.1 Hardware/software requirements

The Hub and all three computer clusters run on some form of Linux, and that is unlikely to change for the foreseeable future. Additionally, developing software on multiple operating systems is very time-consuming and not advisable in the proof-of-concept stage of development. For these reasons the GDS was developed assuming that all computers involved run on Linux. Additionally, it is assumed that all computers have SSH installed and set up and that the local computer (i.e. the users equivalent of Hub) has its .ssh/config file configured so that it is possible to access all computer clusters without needing human interaction (e.g. having to type a password).

Throughout this thesis any related examples will assume the following .shh/config file.

```
Host ssh_alias
        User user_name
        HostName address_to_remote_computer
        IdentityFile /home/user_name/.ssh/key_name
```

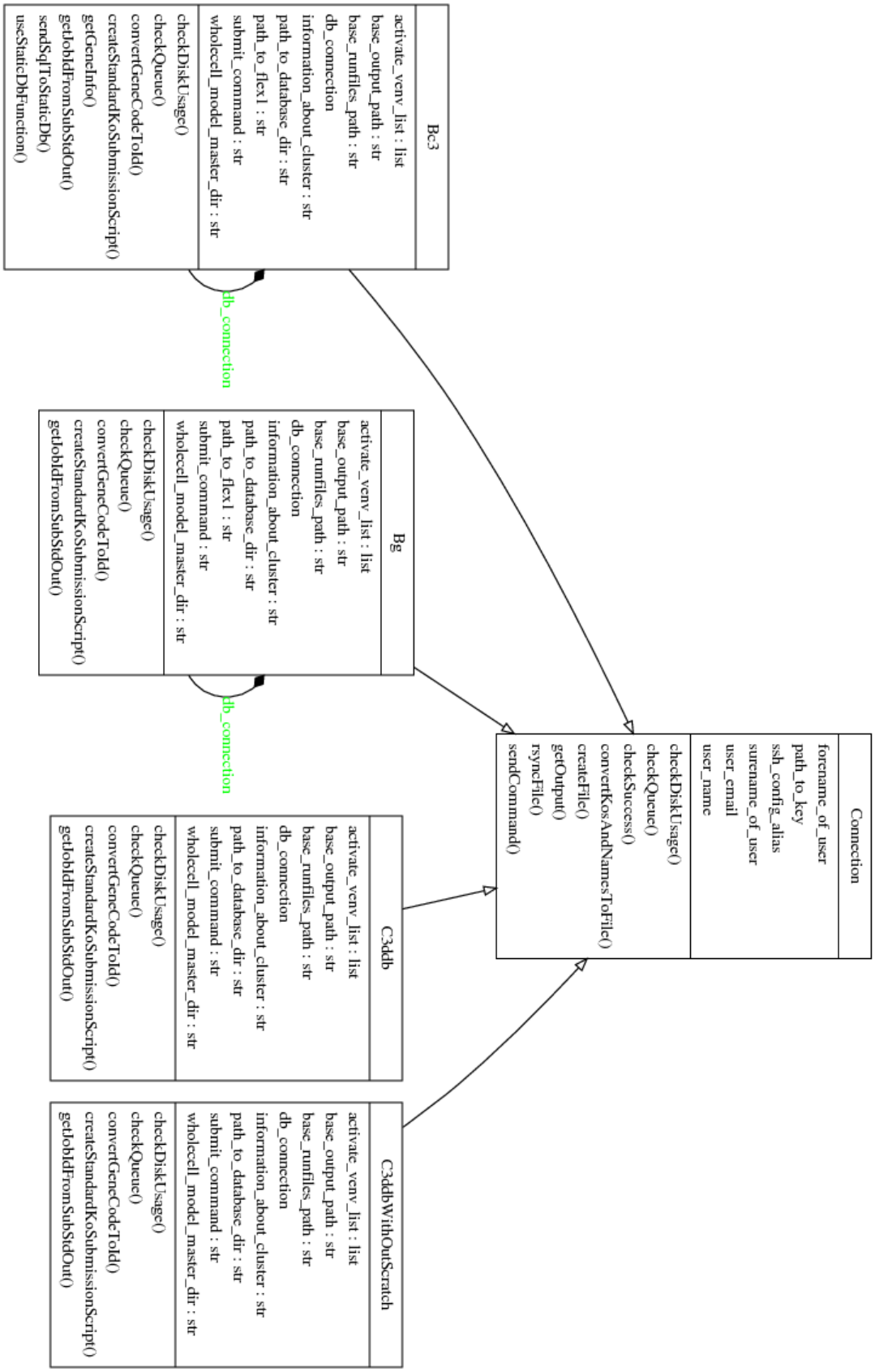
This setup means that in a Linux terminal on the local machine it is possible to connect to the remote machine, without manually entering a password, using the following command `ssh ssh_alias .`

4.2 Computer communication

Computer communications are kept in two modules, `base_connection` and `connections` . The former holds any abstract classes, and the latter holds the classes for specific computer connections. Figure 4.2 shows each class in these two modules, their class variables and methods, and how they inherit from one another.

4.2.1 The `Connection` class

The `Connection` class is the only class in the base_connection module and is the abstract class that defines the core structure of all connection classes. All the child classes that inherit from


Bc3
activate_venv_list : list
base_output_path : str
base_runfiles_path : str
db_connection
information_about_cluster : str
path_to_database_dir : str
path_to_flex1 : str
submit_command : str
wholecell_model_master_dir : str
checkDiskUsage()
checkQueue()
convertGeneCodeToId()
createStandardKoSubmissionScript()
getGeneInfo()
getJobIdFromSubStdOut()
sendSqlToStaticDb()
useStaticDbFunction()
db_connection
Bg
activate_venv_list : list
base_output_path : str
base_runfiles_path : str
db_connection
information_about_cluster : str
path_to_database_dir : str
path_to_flex1 : str
submit_command : str
wholecell_model_master_dir : str
checkDiskUsage()
checkQueue()
convertGeneCodeToId()
createStandardKoSubmissionScript()
getJobIdFromSubStdOut()
db_connection
Connection
forename_of_user
path_to_key
ssh_config_alias
surename_of_user
user_email
user_name
checkDiskUsage()
checkQueue()
checkSuccess()
convertKosAndNamesToFile()
createFile()
getOutput()
rsyncFile()
sendCommand()
C3ddb
activate_venv_list : list
base_output_path : str
base_runfiles_path : str
db_connection
information_about_cluster : str
path_to_database_dir : str
submit_command : str
wholecell_model_master_dir : str
checkDiskUsage()
checkQueue()
convertGeneCodeToId()
createStandardKoSubmissionScript()
getJobIdFromSubStdOut()
C3ddbWithOutScratch
activate_venv_list : list
base_output_path : str
base_runfiles_path : str
db_connection
information_about_cluster : str
path_to_database_dir : str
submit_command : str
wholecell_model_master_dir : str
checkDiskUsage()
checkQueue()
convertGeneCodeToId()
createStandardKoSubmissionScript()
getJobIdFromSubStdOut()

this class can be thought of as a portal to another computer.

Initialisation:
The `Connection` class is initialised with attributes: • The username of the account on the remote computer. • The SSH alias defined in section 4.1. • The path to the encryption key required to login as said user — see section 4.1. • The forename of the user. • The surname of the user. • The user's email.

Instance methods:
The `createFile` method turns a list of text into a file on the local computer and sets the file permissions if specified.

The `rsyncFile` method transfers files either locally or to the remote computer. This method uses the `rsync` function commonly found on Linux computers and can be installed with `apt-get install rsync` on Debian-based systems or `yum install rsync` on RPM-based systems.

The `convertKosAndNamesToFile` method is specific to gene knockout experiments with the whole-cell model of *M. genitalium*. It creates two files, one that contains all the sets of gene knockouts and one that contains a name for each set. Both files are ordered in the same way, so line 1 of the names file gives the name of the knockout set on line 1 of the gene knockout sets file.

The `sendCommand` method takes a list of shell commands (in this thesis always `BASH`) commands and runs them on the remote computer. It returns a dictionary with the `stdout` , `stderr` and `return_code` .

Static methods:
The `checkSuccess` method takes a function that needs to make a remote connection with a set of arguments and executes that function in a `while` loop until the return code signifies success. Having it loop normally can potentially overload the login server of the remote computer (like a DoS attack attack), and so this waits a certain duration of time before trying again. It starts relatively frequent and slows down, starting with every three seconds and ending by checking once every 24 hours. Once the function returns a successful return code, `checkSuccess` returns

Figure 4.2 *(preceding page)*: This UML diagram has a box for each class that has the class name followed by class variables followed by class methods. The arrows go from a child class to the parent class that it is inheriting from. Here one can see that the abstract class `Connection` is the parent class for all connection objects. The arrows pointing to themselves is because the BC3 and BG classes have instances of themselves in-order to standardise the way in which a connection connects to the database — see section 4.2.2.

the output with a successful return code. If the function does not return a successful return code within 7 days, then it exits with a return code of 13.

Abstract methods:
`checkQueue` and `checkDiskUsage` are methods that will often be used but vary from computer to computer. When creating a child class for a new computer to connect to, it is advised that these methods are properly overloaded because they are made abstract so that additional software that uses the framework can assume that this functionality is available. If a method(s) is not available, then compilation or potentially dangerous runtime errors may occur. However, anyone that does not wish to add this or the computer does not have the functionality can create the function name and use `pass` . The `checkQueue` method checks the queue on a remote cluster and the `checkDiskUsage` returns disk usage statistics of the remote computer.

4.2.2 Child classes of the `Connection` class

There are four child classes that inherit from the `Connection` class, the `Bc3` , `Bg` , `C3ddb` , and `C3ddbWithOutScratch` classes and act as a portal into the BC3, BG, and C3DDB clusters, respectively. This means that they have access to all the methods and variables from `connection` and will also need to overload any abstract methods. It is worth noting that almost all child class attributes should be the same so that a program using them always knows how to perform a specific task no matter what cluster it is connecting to. The only difference is the way that a specific task is performed on a specific computer. For example, the `createFile` method from the `Connection` class can be implemented in the `Connection` class because Python can create a file on any operating system in one single way. However, the `checkQueue` method must be implemented in the child classes because the queuing system depends on what cluster one is trying to connect to. Only the child attributes will be discussed to avoid repetition because the `Connection` class (i.e. the parent class) was described in section 4.2.1. Since the child attributes all perform the same function, just implemented differently, all the child class attributes will be discussed in one since they all have the same explanation except with slight differences in implementation. In order to see details of implementation, the reader is advised to view the source code supplied in the supplementary information (see base_connection.py and connections.py).

Initialisation:
The child `Connection` classes have the additional initialisation variables: • A path to a directory that all simulation output should be saved in. • A path to a directory that all files/data needed to submit jobs to the cluster should be saved in. • A path to the *WholeCell-master* directory that is necessary to run simulations using the whole-cell model of *M. genitalium*.

Abstract methods:

These classes do not dictate any abstract methods however the `Connection` class dictates that all child classes must have the `checkQueue` and `checkDiskUsage` methods.

The `checkQueue` method takes a job number as a parameter and returns all entries in the queuing system that have that specified job number. The TORQUE and SLURM clusters use different queuing systems and so the `checkQueue` methods are the same for the SLURM clusters (i.e. BG and C3DDB) but different for the TORQUE clusters (i.e. BC3).

The `checkDiskUsage` method returns the user's disk usage. BC3 has a custom command, `pan_quota` , for this and so uses this command to return the disk available and disk used and percentage used. To implement this on the SLURM clusters is not straight forward because there is no equivalent to `pan_quota` and using multiple shared file systems makes it more complicated than it would be on a normal PC. With that said, it would be possible to implement this, but at present, there is no pressing need for it. In the case of needing to implement a method because it is defined in as an abstract method in the parent class one can simply create the child instance method with only the `pass` command[①].

Instance methods:

The `createStandardKoSubmissionScript` is a method that creates an executable TORQUE or SLURM submission script (BC3 or BG and C3DDB, respectively) that will submit a batch of gene knockout simulations using the whole-cell model of *M. genitalium*. All the method needs to know is • The local path and file name of the submission script that it will create. • The name of the job that will be sent to the cluster queuing system. • The number of gene knockout sets in this batch of jobs. • The remote path and file name of a file that contains all the names of each of the gene knockout sets. [②] • The remote path and file name of a file that contains all the gene codes of each of the gene knockout sets. [③] • The number of times that the user wishes each gene knockout set to be repeated. • The path to the *WholeCell-master* directory. • The path that the simulation data output will be stored. • The path and file name that the simulation's standard out should be saved to. • The path and file name that the simulation's standard error should be saved to. The method checks that an unrealistic amount of simulations has not been given and then splits the simulations across array jobs and cores within an array job such that it gets through the cluster queue as quickly as possible.

[①]This is currently an acceptable solution because nothing that uses the GDS calls this method. However, in the future, this may change, and so any instances from this class used in this hypothetical future code will break or create a dangerous bug in the code.

[②]These names must be unique and must be in the same order as the gene knockout sets file — there is one name per line.

[③]These sets of codes must be in the same order as the gene knockout sets names file — there is one comma-separated set of gene codes per line.

When a job is submitted to a cluster queuing system a job number is normally returned to standard out. The `getJobIdFromSubStdOut` method takes the number from the standard out and remembers it so that the job's progress can be monitored.

The file `static.db` is an SQLite3 database that acts as the central authority on data related to *M. genitalium* and its whole-cell model (see section 4.5.1). Children of the `Connection` class will want to query this database for various reasons, and so there are four instance methods that relate to this.

There is a Python library in the same directory as `static.db` that makes querying the database easier. If one wants to send a raw SQLite3 query to `static.db` then the `sendSqlToStaticDb` method will do this and return the result. If one wants to use any of the other functions in the library, then the `useStaticDbFunction` can be used.

The `convertGeneCodeToId` method converts a tuple of gene codes into gene IDs.

The `getGeneInfo` method takes a tuple of gene codes and returns a dictionary containing the following attributes taken from the supplimentary information of [52] • gene code • gene type (e.g. mRNA or rRNA etc) • gene name • gene symbol • functional unit of gene product • Karr2012 deletion phenotype • essential in model according to • essential in experiment according to.

4.3 Job manager

The job manager libraries are kept in one module `batch_jobs` which contains two classes, `JobSubmission` and `ManageSubmission` . The job manager libraries do not have a rigid structure defined by abstract classes since this structure is defined in the computer communication and the algorithm libraries. The UML diagram of the `batch_jobs` module can be seen in figure 4.3. Whilst there is no inheritance structure between the two classes, it is important to note that they are intricately linked by the fact that the `ManageSubmission` class requires an instance of the `JobSubmission` class in order to be created.

4.3.1 The `JobSubmission` class

The `JobSubmission` class holds everything needed to submit a batch of jobs to a computer cluster.

Initialisation:
The class needs to be initialised with a • name for the submission. • A child class that inherits from the `Connection` class of the `base_connection` module. • A Python dictionary whose keys are unique names and the values are tuples of genes codes where each code represents a gene to

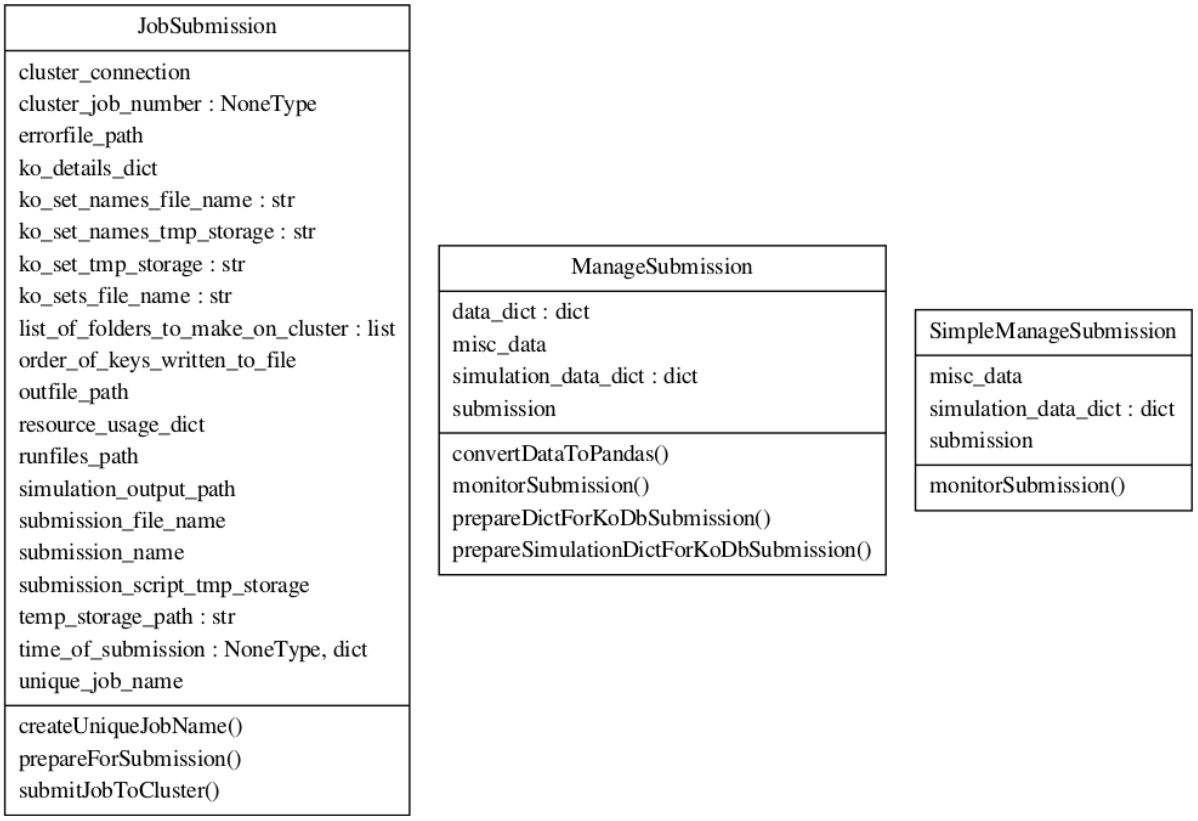


Figure 4.3: Class diagram for the batch_jobs.py module shows their arguments and methods. There are three classes that are not related by inheritance.

knockout. • A base path on the remote computer that the simulation data should be saved in. • A base path on the remote computer that the simulation standard error files should be saved. • A base path on the remote computer that the simulation standard out files should be saved. • A base path on the remote computer that the files needed to run the simulations should be saved. • The number of times the user wants each gene knockout set to be repeated. • A path to the *WholeCell-master* directory where the whole-cell model of *M. genitalium* is stored.

Instance methods:

The `createUniqueJobName` method creates a unique name so that files can be created and stored locally. It needs to be unique because an algorithm instance may want to submit more jobs than can be handled in one `JobSubmission` instance and so there will be multiple very similar instances running at the same. In order to make sure that similar instances do not interfere with each other's files, a directory name that is guaranteed to be unique is needed.

Files often need to be created and transferred to the remote computer before a job can be submitted to the cluster. `prepareForSubmission` is the method that does this.

The `submitJobToCluster` method, submits the job to the cluster, records the time and job number, and then deletes any temporary files created locally for the submission.

4.3.2 The `ManageSubmission` class

The `ManageSubmission` class submits a `JobSubmission` instance to a cluster and then monitors its progress in the queue. When the job is finished it converts the raw simulation output into Pandas DataFrames, updates `ko.db` , and remembers the average growth rate and division time of all the simulations.

Initialisation:
The class needs to be initialised with • an instance of the `JobSubmission` class. • Sometimes an algorithm needs to pass information specific to only that algorithm, and so there is a class variable that this can be passed to if necessary. • The class automatically submits the job contained in the `JobSubmission` instance which can be a problem for unit testing and so a variable is passed to tell the class whether to initialise normally or in test mode.

Instance methods:
The `prepareDictForKoDbSubmission` method creates a dictionary that is designed to be recognised by the `ko_db` module and so can be used to update the `ko.db` database. This returns the dictionary that will be submitted to the database, but all data related to the simulations in the job submission will not be filled in yet — it will only contain the common data like the details of the person who submitted the job, the details about the cluster, and the time that the job was submitted.

The `prepareSimulationDictForKoDbSubmission` method goes to the directory of a specific simulation on the remote computer to open the Pandas DataFrame, extract the average growth rate and the time step when the `pinchedDiamter` variable was first zero (i.e. the time of division — if the cell did not divide then it returns the number zero). It then returns a dictionary where the key is the gene knockout set that defines the genome of the organism, and the value is the average growth rate and division time of that organism.

The `monitorSubmission` method watches every simulation related to the job submission as it progresses through the queuing system by checking the queue after the first hour followed by 15-minute intervals after that. Occasionally some jobs might get lost in the queuing system or the simulation crashes, and this method will account for these events. When this method finds that simulations have finished, it converts the data from `.mat` files to Pandas DataFrames stored in `.pickle` files — this is done in parallel using `ProcessPoolExecutor` from the `concurrent.futures`

module. As each simulation finishes and the data is converted, the average growth rate and division time is also retrieved into a dictionary using the `prepareSimulationDictForKoDbSubmission` method — these are added to a dictionary that is stored as a class variable and so once all the simulations are completed the relevant data can be found all in one place. When the whole job submission is finished, the data is converted and the growth and division time data is collected then the method updates `ko.db` using the `ko_db` library which can be found on `flex1` in the same directory as the database (a copy of these things can be found in the supplementary information).

The `convertDataToPandas` method goes to the directory of the simulation data output and converts all the raw data from `.mat` files into Pandas DataFrames stored in `.pickle` files using the Python package, Pickle. It is worth noting that the `.mat` files are read by the `File` method of the `h5py` library. However, it was found that occasionally it threw an error whilst trying to read the file even though Matlab had no problem. The standard version of `.mat` file that Matlab uses is 7.0 which is a compressed version, and it appeared that the uncompressed version, 7.3, did not cause this error. In order to avoid these errors, code in the whole-cell model was modified so that it saved the files in version 7.3 rather 7.0 - for more information see section A.8 of the appendix.

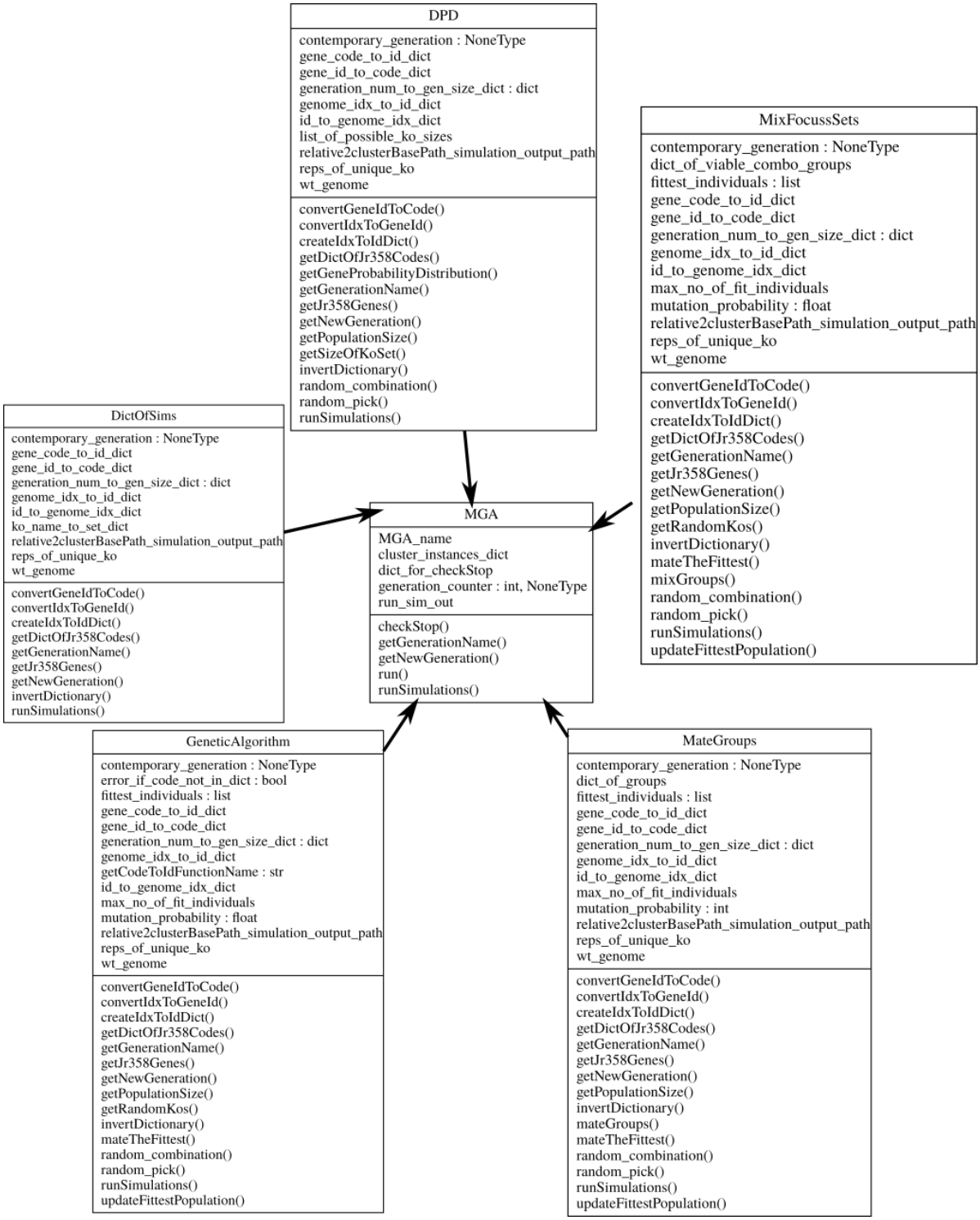
4.4 Algorithms

The algorithm libraries are stored in the `multigeneration_algorithm` module and is made up of 11 classes. The UML diagram of the module has been split into 2 figures due to the large amount of classes, arguements, methods, and relationships. These two figures can be seen in figures 4.4 and 4.5 and one can see that there is one parent class, `MGA` , that all 10 child classes inherit from.

4.4.1 The `MGA` class

The `MGA` class is an abstract class that acts as a template that all other algorithm classes should inherit from. The class must be abstract enough that it can act as a template for as many algorithms as possible. Figure 4.6 shows how the `MGA` class, and thus all algorithms, execute. One can see that all algorithms will be started by running the `run` method which then initiates a loop that does not stop until a specified maximum generation number is reached. Each iteration of the loop represents a single generation and each generation is created and simulated using the `runSimulations` method.

Initialisation:
The class needs to be initialised with: • a Python dictionary where the values are cluster connec-


DPD
contemporary_generation : NoneType
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
list_of_possible_ko_sizes
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGeneProbabilityDistribution()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getSizeOfKoSet()
invertDictionary()
random_combination()
random_pick()
runSimulations()
MixFocussSets
contemporary_generation : NoneType
dict_of_viable_combo_groups
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
max_no_of_fit_individuals
mutation_probability : float
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getRandomKos()
invertDictionary()
mateTheFittest()
mixGroups()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()
DictOfSims
contemporary_generation : NoneType
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
ko_name_to_set_dict
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
invertDictionary()
runSimulations()
MGA
MGA_name
cluster_instances_dict
dict_for_checkStop
generation_counter : int, NoneType
run_sim_out
checkStop()
getGenerationName()
getNewGeneration()
run()
runSimulations()
GeneticAlgorithm
contemporary_generation : NoneType
error_if_code_not_in_dict : bool
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
getCodeToIdFunctionName : str
id_to_genome_idx_dict
max_no_of_fit_individuals
mutation_probability : float
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getRandomKos()
invertDictionary()
mateTheFittest()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()
MateGroups
contemporary_generation : NoneType
dict_of_groups
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
max_no_of_fit_individuals
mutation_probability : int
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
invertDictionary()
mateGroups()
mateTheFittest()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()

tion instances that are available to run simulations on (i.e. child classes of the `Connection` class from the `base_connections` module). The keys of the clusters are unique names that label each cluster connection. • A Python dictionary that defines when the algorithm should stop running. At present this only has the functionality to stop at a predetermined generation in the future but additional options could be added. • The name given to this instance of the algorithm. • Once the class is initialised a class variable that remembers what generation the algorithm is on is created and set to `None` .

Instance methods:
The `checkStop` method checks to see if the generation counter (i.e. class variable 4.4.1) is less than the 'stop generation' number in the 'checkStop' dictionary (i.e. class variable 4.4.1). If the generation counter (4.4.1) is equal or more than the 'checkStop' dictionary (4.4.1) it returns `True` otherwise it returns `False` .

The `run` method is a `while` loop that runs the next batch of simulations and increments the generation counter (i.e. class variable 4.4.1) by one until the generation counter is greater or equal to the maximum generation (i.e. `while checkStop()!= True`).

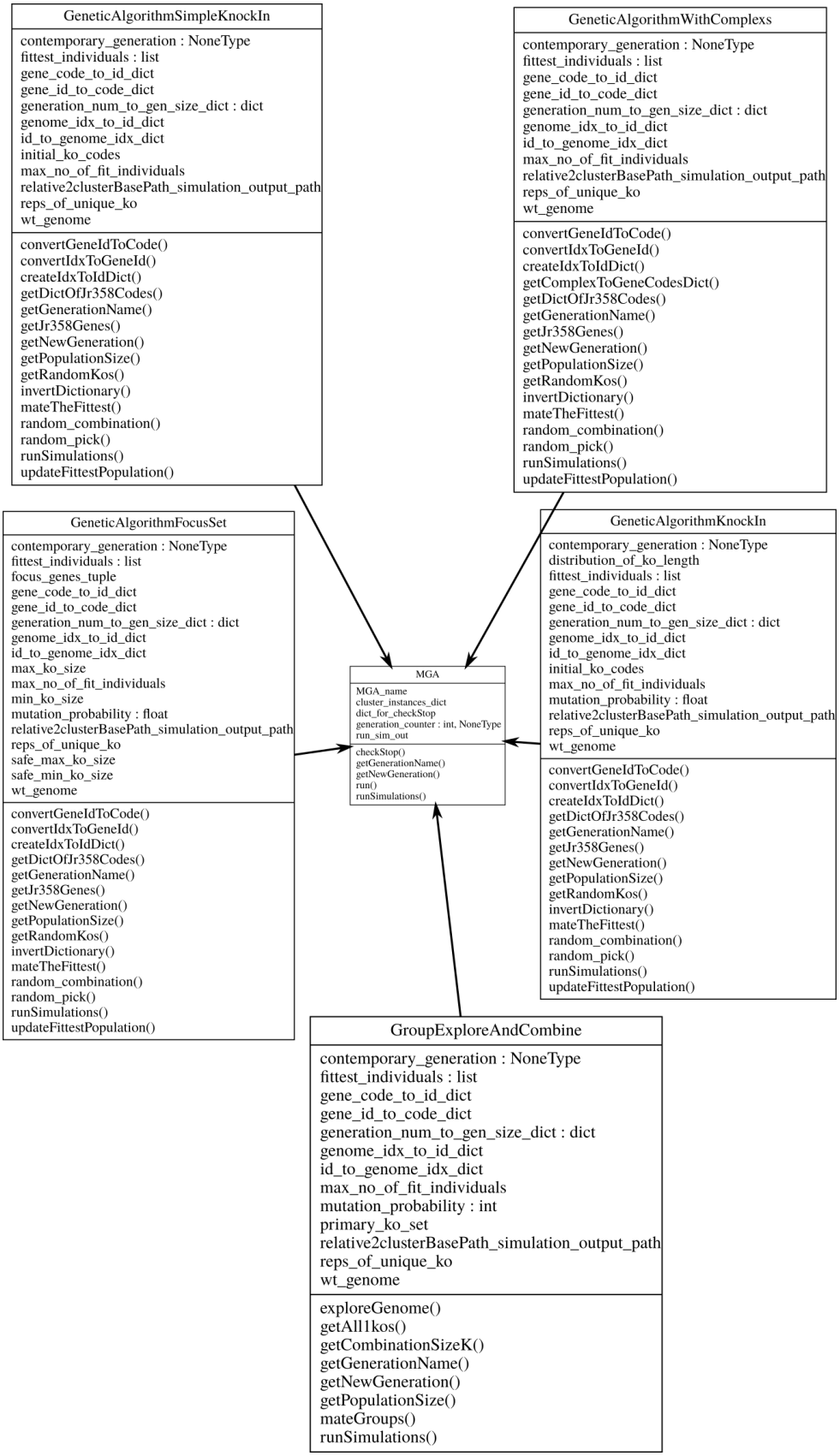
Abstract methods:
The `runSimulations` method is called by the `run` method and implements one generation of an algorithm. Since this is the abstract class it does not specify any algorithm and leaves it for the child classes to define. Two other abstract methods are set as they will ned to be called by the `runSimulations` method. These are the `getGenerationName` and `getNewGeneration` methods.

The `getNewGeneration` method will need to create the genomes of the children that need to be simulated in the next generation but is an abstract method and so left for the child class to define.

The `getGenerationName` method will return a name for each generation that can be used to identify what belongs to each generation. This is an abstract method and so is left for the child classes to define.

The following sections will look at the algorithms implemented using the GDS. All algorithms

Figure 4.4 *(preceding page)*: Class diagram for the parent class plus 5 (out of 10) child classes of the multigeneration_algorithm.py module. It shows their arguments, methods and their inheritance relationships. The remaining 5 child classes and their inheritance relationships can be seen in figure 4.5.


GeneticAlgorithmSimpleKnockIn
contemporary_generation : NoneType
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
initial_ko_codes
max_no_of_fit_individuals
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getRandomKos()
invertDictionary()
mateTheFittest()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()
GeneticAlgorithmWithComplexs
contemporary_generation : NoneType
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
max_no_of_fit_individuals
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getComplexToGeneCodesDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getRandomKos()
invertDictionary()
mateTheFittest()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()
GeneticAlgorithmFocusSet
contemporary_generation : NoneType
fittest_individuals : list
focus_genes_tuple
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
max_ko_size
max_no_of_fit_individuals
min_ko_size
mutation_probability : float
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
safe_max_ko_size
safe_min_ko_size
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getRandomKos()
invertDictionary()
mateTheFittest()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()
MGA
MGA_name
cluster_instances_dict
dict_for_checkStop
generation_counter : int, NoneType
run_sim_out
checkStop()
getGenerationName()
getNewGeneration()
run()
runSimulations()
GeneticAlgorithmKnockIn
contemporary_generation : NoneType
distribution_of_ko_length
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
initial_ko_codes
max_no_of_fit_individuals
mutation_probability : float
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
convertGeneIdToCode()
convertIdxToGeneId()
createIdxToIdDict()
getDictOfJr358Codes()
getGenerationName()
getJr358Genes()
getNewGeneration()
getPopulationSize()
getRandomKos()
invertDictionary()
mateTheFittest()
random_combination()
random_pick()
runSimulations()
updateFittestPopulation()
GroupExploreAndCombine
contemporary_generation : NoneType
fittest_individuals : list
gene_code_to_id_dict
gene_id_to_code_dict
generation_num_to_gen_size_dict : dict
genome_idx_to_id_dict
id_to_genome_idx_dict
max_no_of_fit_individuals
mutation_probability : int
primary_ko_set
relative2clusterBasePath_simulation_output_path
reps_of_unique_ko
wt_genome
exploreGenome()
getAll1kos()
getCombinationSizeK()
getGenerationName()
getNewGeneration()
getPopulationSize()
mateGroups()
runSimulations()

implemented using the GDS inherit all class attributes from the `MGA` class and so unless they are abstract methods then they will not be discussed again. Abstract methods are not defined in the parent class and so that definition will need to be given in the child class.

4.4.2 The `GeneticAlgorithm` class

Genetic algorithms are a great general purpose, easy to implement machine learning algorithm used to optimise objectives and have been used to in a wide variety of tasks. A genetic algorithm attempts to, roughly, mimic evolution by natural selection in order to learn. The population of individuals is made up of parents and children where the parents are the fittest individuals that survived whatever *natural selection* is placed upon them. Here the natural selection is normally implemented by some kind of objective function that one wishes to optimise. A set of genes defines each parent and the fittest individuals *mate* to create children that are made up of a random combination of the genes of both parents, plus some random mutation.

Figure 4.7 shows how each generation of the `GeneticAlgorithm` is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the parameters needed to initialise the `MGA` class the `GeneticAlgorithm` class requires • The maximum number of fit individuals that are allowed to survive each generation. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulation data will be stored in. This is relative to the cluster base path which is given by the connection class. • The probably that a mutation occurs whilst creating a child. • The name of a function (that exists in the child class) that can be called to get a dictionary that contains all the gene codes and IDs that make up the genome.

Abstract methods:
The `MGA` class defined three abstract methods that need to be defined in any child classes, `getGenerationName` , `getNewGeneration` , and `runSimulations` .

The `getGenerationName` method returns the string ‘genN’ where ‘N’ is the current generation number.

Figure 4.5 *(preceding page)*: Class diagram for the parent class plus 5 (out of 10) child classes of the multigeneration_algorithm.py module. It shows their arguments, methods and their inheritance relationships. The remaining 5 child classes and their inheritance relationships can be seen in figure 4.4.

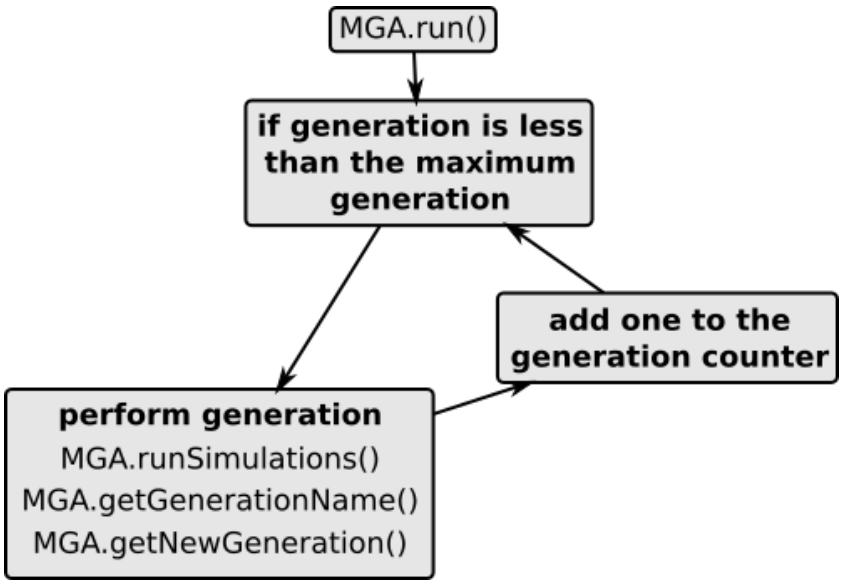


Figure 4.6: A schematic of the abstract class MGA. One can see that all algorithms will be started by using the `run()` method which initiates a loop that repeats until a maximum generation is reached. Each loop represents one generation of the algorithm and the simulations are chosen and run using the `runSimulations()` method which is undefined since it is an abstract class. This class needs to be defined by child classes that inherit from this class. `getGenerationName()` and `getGeneration()` are also abstract methods that will be utilised by child implementations of the `runSimulations()` method.

The `getNewGeneration` method decides what method to call to generate the next generation of children. For the genetic algorithm, this calls the `mateTheFittest` method if the number of fit individuals is greater than one. Otherwise, it calls the `getRandomKos` method.

The `runSimulations` method defines the algorithm over one generation. For `geneticAlgorithm` this means using the `getGeneration` method to get all the child genomes for the next generation of simulations as well as the number of clusters available. It then splits the children evenly over all the clusters (plus remainders) and creates `JobSubmission` instances for each set.

Parallel computing is required to create the `ManageSubmission` classes. Python executes all lines of code sequentially and waits for each line of code to finish before executing the next. The sequential nature of code execution means that normal code will submit the first batch of jobs to cluster-1 and then wait for the whole `ManageSubmission` process to finish before submitting the second batch of jobs to cluster-2. This sequential use of the clusters defeats the point of having multiple computing facilities, and so a parallel solution was created by using the `multiprocessing` library to `map` each job to their respect clusters. Due to the amount of time it takes to convert the simulation data output to Pandas DataFrames and the fact that it is not uncommon for lots of simulations to finish at a similar time the `ManageSubmission` class

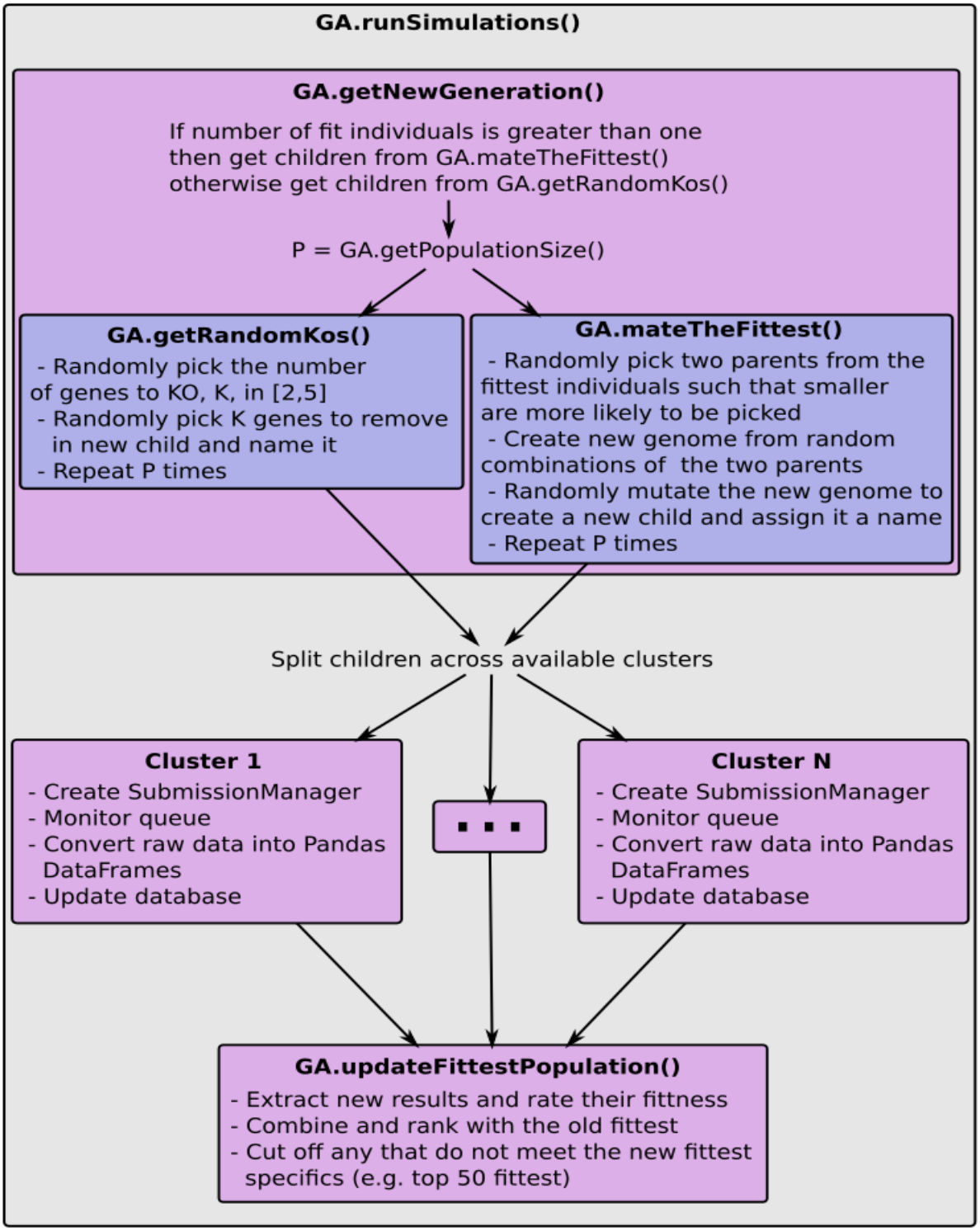


Figure 4.7: This diagram shows how the `runSimulations` method is implemented in the `GeneticAlgorithm` class. Grey boxes contain everything that happens within the `runSimulations` method. Lilac boxes contain any significant methods or classes called within the `runSimulations` method. Blue boxes contain significant methods or classes called within the lilac boxes. Here GA represents the `GeneticAlgorithm` class.

executes the `convertDataToPandas` method in parallel as well. So one can see that the process running the `ManageSubmission` class is already a child process from the parallelised mapping in the `runSimulations` method. Unfortunately, the `multiprocessing` library does not allow child processes to spawn new child processes and so the more popular library was dropped and replaced by the `futures` module of the `concurrent` library.

Once all the simulations for this generation have completed then the `runSimulations` method passes all the finished `ManageSubmission` instances to the `updateFittestPopulation` method in order to *learn* what happened in the current generation.

Instance methods:
The `getPopulationSize` method returns the desired population of children for this generation.

The `getRandomKos` method finds out what genes can be knocked-out out from the genome from class variables and uses the `getPopultionSize` method to find out how many children need to be created. It then uses these to create the desired number of children each with a random number of genes knocked out in the range $[2,5]$. The number of genes knocked out and which genes are knocked out are both picked from a uniform distribution. Each child name is made up of two parts, the first part is 'ko' and the second part is a number that starts at 1 and increments by one every time a new child is created so that each child has a unique name in the generation. When a new generation starts, the name counter goes back to 1. This method returns a dictionary where the keys are the names of the children, and the values are the gene codes of the genes that need to be knocked out — this dictionary will be referred to as the *child name to gene knockout set dictionary*.

The `mateTheFittest` method creates all the children for the next generation. The children are created by mimicking natural selection, and sexual reproduction by randomly selecting two parents from the fittest individuals found so far (natural selection) and creates a child by mixing the genomes of the two parents (sexual reproduction). The theory of this process is explained in section 5.1 however the modified exponential distribution will be discussed in a little more detail here. The `mateTheFittest` method aproximates the modified expontial distribution (see equation 5.4 where $\lambda = 2$) using code snippet 4.8. Here it can be seen that instead of using `int(np.around(np.random.exponential(2)+ 1))` it does not add the 1 to the random variable but instead generates the random variable in a while loop until it results in a number greater than zero. The idea behind this is that all the zero values get split (acording to the exponential distribution) across the higher values preserving the shape of the distrbution and shifting it across the x-axis by one. This raises the question of how good is our approximation of the modified exponential distrbution and does it preserve the requirement that the area under the distribution curve is equal to 1. I have not been able to prove this one way or another but a comparison to the

```
number_of_gene_mutations = 0
        while number_of_gene_mutations == 0:
                number_of_gene_mutations = int(np.around(np.random.
                    exponential(2)))
```

Figure 4.8: Code segment showing how the modified exponential distribution is calculated. The while-loop means that a 0 value will never be created, the `np.around` method performs standard rounding on the result, and the `int` method converts the data type from float to integer (`int` truncates all decimal places rather than rounding them and so rounding them first results in fewer loops).

actual distribution is provided. Ten thousand samples were taken from the modified exponential distribution (see equation 5.4 with $\lambda = 2$) and compared to 10,000 samples of the approximation of the modified exponential distribution in the `mateTheFittest` method. The result can be seen in figure 4.9 and shows that only small differences can be seen - one would expect small differences by random chance although no quantitative test is performed in this thesis. The conclusion here is that it is a reasonable approximation. However, there is no known reason to chose the approximation over the actual function as the latter is clearer and more efficient (due to the lack of a while loop) and thus is noted as a desired change to the code in future versions.

The `updateFittestPopulation` method takes the simulation results from a completed `SubmissionManager` instance and extracts all individuals that produced a dividing cell. The dividing cells are then combined with the current fittest individuals and ranked so that the smallest genomes are at the top and the largest at the bottom. The algorithm class is initialised with a maximum number of fit individuals, M, and so the top M individuals are taken from the new list and set as the new fittest individuals.

Instance methods for all child classes:
These are methods deemed generally useful and automatically get put into all child classes of the `MGA` class and will always be identical in implementation. These will not be defined again in the other child classes.

The `random_combination` method takes a Python iterable (e.g. a list) and the desired size and then picks a random subset of that size from a uniform distribution which is then returned to the user.

The `random_pick` method is the same as the `random_combination` method except it picks the iterable elements from a distribution defined by a iterable of probabilities passed by the user.

The `getJr358Genes` method returns a tuple of gene codes. These gene codes are defined as

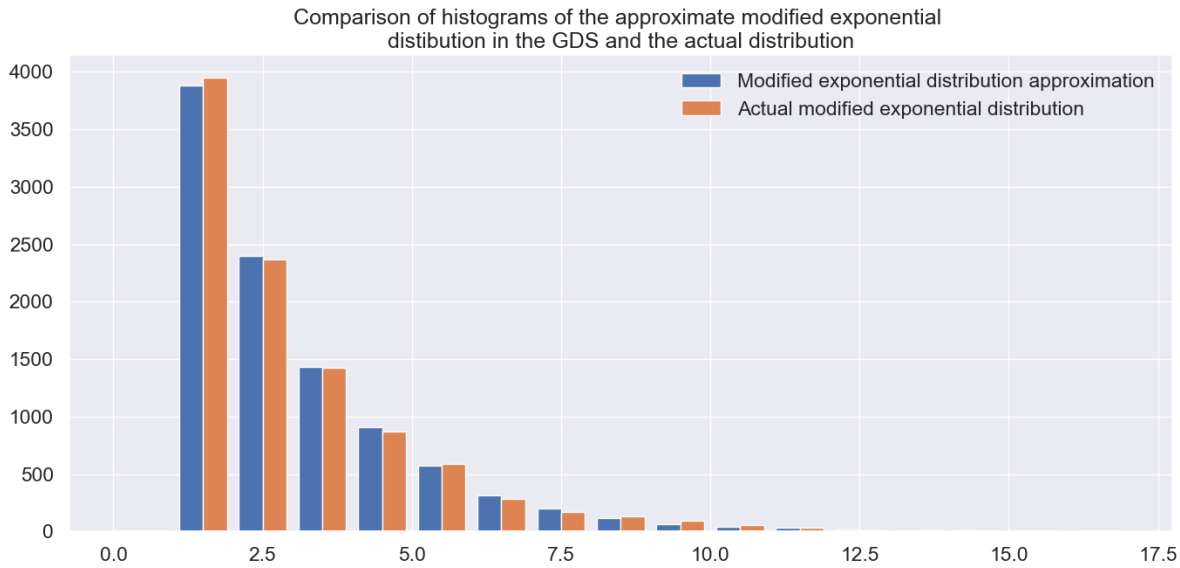


Figure 4.9: A comparison of histograms from the aproximate (see code segment 4.8) and actual (see equation 5.4) modified distributions the modified exponential distribution. Ten thousand data points were sampled from each distribution, binned in the same way and each bin plotted next to each other for comparison.

all of the protein-coding genes that are characterised in the whole-cell model of *M. genitalium* minus one that tends to crash the simulation (see Chapter 3 for more information on gene selection).

The `getDictOfJr358Codes` method returns a dictionary where the keys are gene codes returned by the `getJr358Genes` method and the values correspond to the ID used in our databases. This method takes an instance of the `Connection` class and uses that connection to get IDs directly from `static.db` so that all users are working off the same data source.

The `invertDictionary` method takes a dictionary and, assuming that the keys and values share a bijective relationship, returns a dictionary where the keys and values are swapped.

The `createIdxToIdDict` method takes a dictionary that converts gene codes into gene IDs and converts that into a dictionary that converts genome indexes into gene IDs.

The `convertIdxToGeneId` method takes a list of genome indexes and returns a corresponding list of gene IDs

The `convertGeneIdToCode` method takes a list of gene IDs and returns a corresponding list of gene codes.

4.4.3 The `MateGroups` class

The `MateGroups` class was created to give more control on which parents mate. The algorithm is passed an arbitrary amount of groups. Each group is a set of parents defined in terms of gene knockouts. The parents will not mate within their groups and will only mate with parents from other groups, giving the user some additional control over mating.

Figure 4.10 shows how each generation of the `Mategroups` algorithm is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `MateGroups` class needs: • A dictionary that contains the gene codes for the base genes and each group. The names are the keys, and the values are lists of gene codes that relate to each name. Each group is made up of sets of gene codes. The base genes are gene codes that do not make it into any of the groups and can be empty — it is included so that some genes can always be included no matter which groups are picked. • A maximum number of fit individuals allowed to survive a generation. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class. • The probability that a mutation occurs whilst creating a child.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method decides what method to call to generate the next generation of children. In this case, it calls the `mateTheFittest` method if the number of fit individuals is greater than one. Otherwise, it calls the `mateGroups` method.

The `runSimulations` method defines the algorithm over one generation. In this case it is the same as the `geneticAlgorithm` as it creates a new generation using the `getNewGeneration` (which is the big difference between the two algorithms) and then splits it across the available clusters and the manages the submissions — for a more detailed discussion of the method see section 4.4.2.

Instance methods:
All the methods in the `MateGroups` class are the same as the `GeneticAlgorithm` class, except `getNewGeneration` and the addition of a new method called `mateGroups` . The `getNewGeneration` method was discussed in the *abstract methods* section above so `mategroups` will be the only

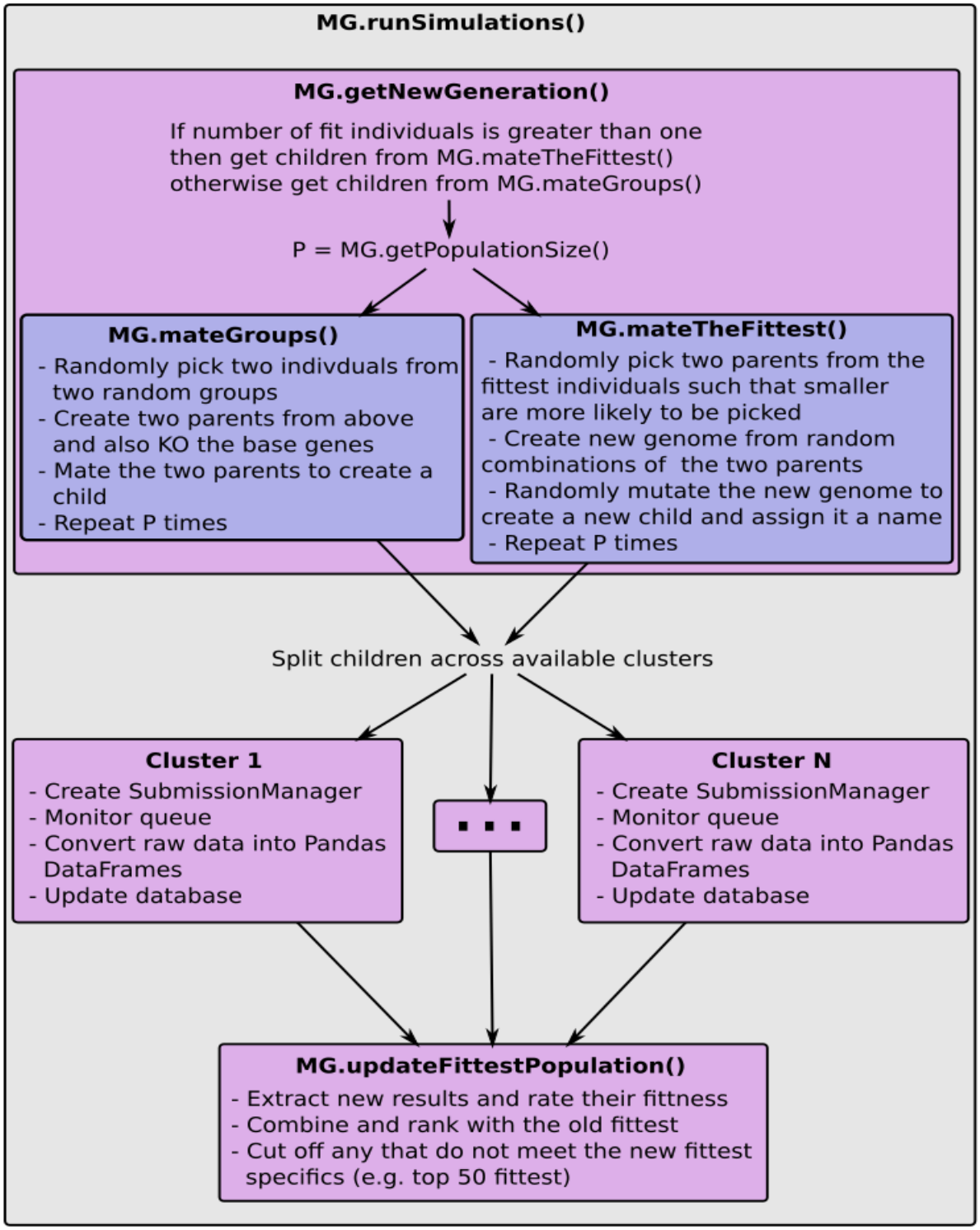


Figure 4.10: This diagram shows how the `runSimulations` method is implemented in the `MateGroups` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here MG represents the `MateGroups` class.

method described here.

The `mateGroups` method uses the `getPopulationSize` method to find out how many children need to be created and also retrieves each group of genes and the base genes. It then creates all the children and puts them into a child name to gene knockout set dictionary. The children are created by randomly selecting two of the groups of genes (such that larger groups are more likely to be picked) and selecting one of the gene sets from each of those two groups (uniformly). The base genes are added to both sets, and these are the genes knocked-out in a genome that define two parents. These two parents then mate with the same principle that two parents mate in the `mateTheFittest` method except there is no random mutation.

4.4.4 The `DictOfSims` class

The `DictOfSims` class was not created to be a *learning* algorithm, as it is meant to run for one generation and only simulate user defined genomes.

Figure 4.11 shows how each generation of the `DictOfSims` is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `DictOfSims` class needs: • A child name to gene knockout set dictionary. These are all the simulations that will be performed by the algorithm. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. This is left-over from the GeneticAlgorithm template and is not strictly necessary but left in as some slight modifications could make it useful again in the future should multi-generation functionality need to be incorporated in some cases. At present, the default setting of this is zero for every generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method simply returns the child name to gene knockout set dictionary passed by the user irrespective of the generation number.

The `runSimulations` method defines the algorithm over one generation. In this case it is the same as the `geneticAlgorithm` as it creates a new generation using the `getNewGeneration` method (which is the big difference between the two algorithms) and then splits it across the available

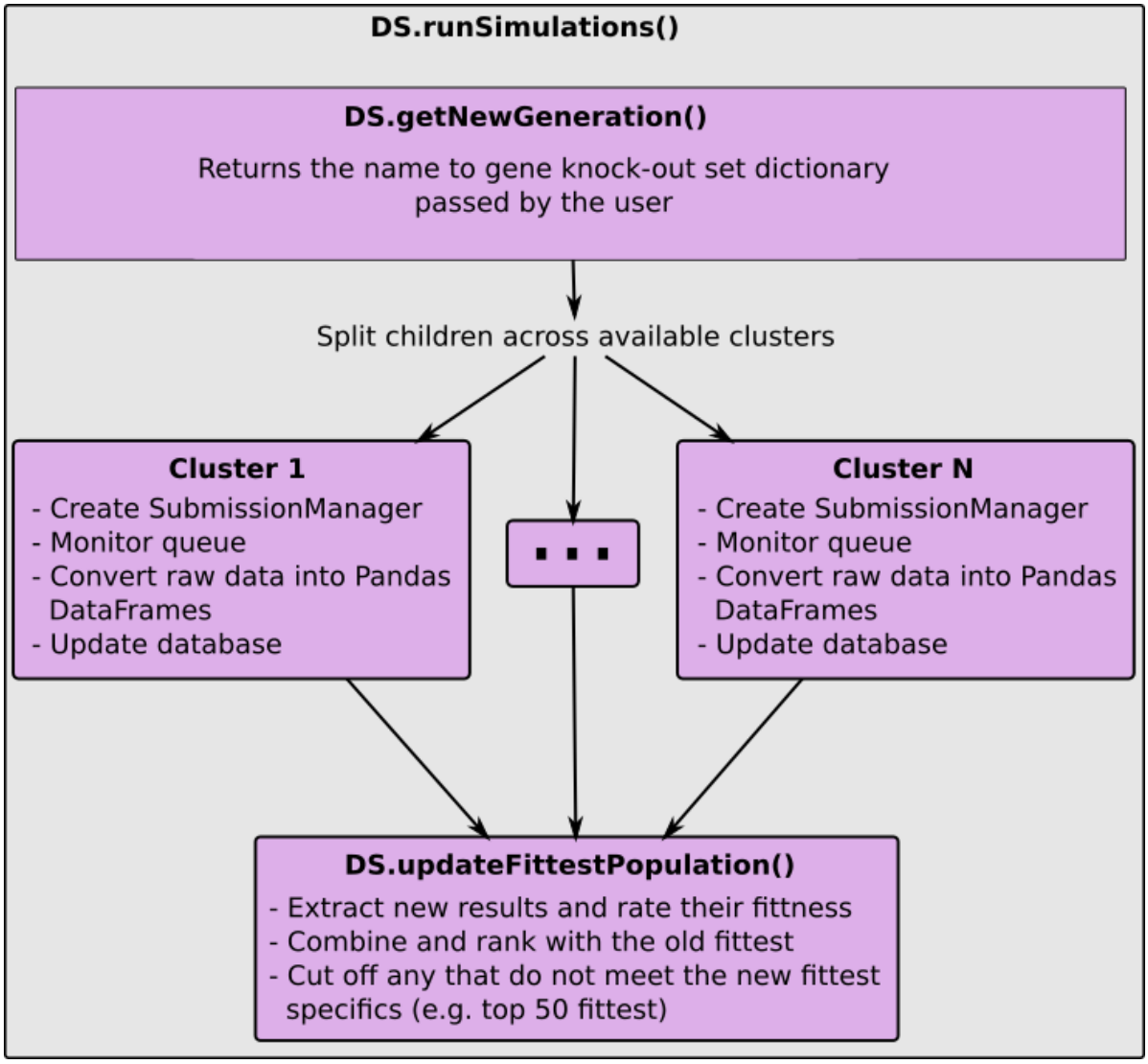


Figure 4.11: This diagram shows how the `runSimulations()` method is implemented in the `DictOfSims` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here DS represents the `DictOfSims` class.

clusters and manages the submissions — for more a more detailed discussion of the method see section 4.4.2.

Instance methods:

There are no other methods in the class that are not covered by previous chapters.

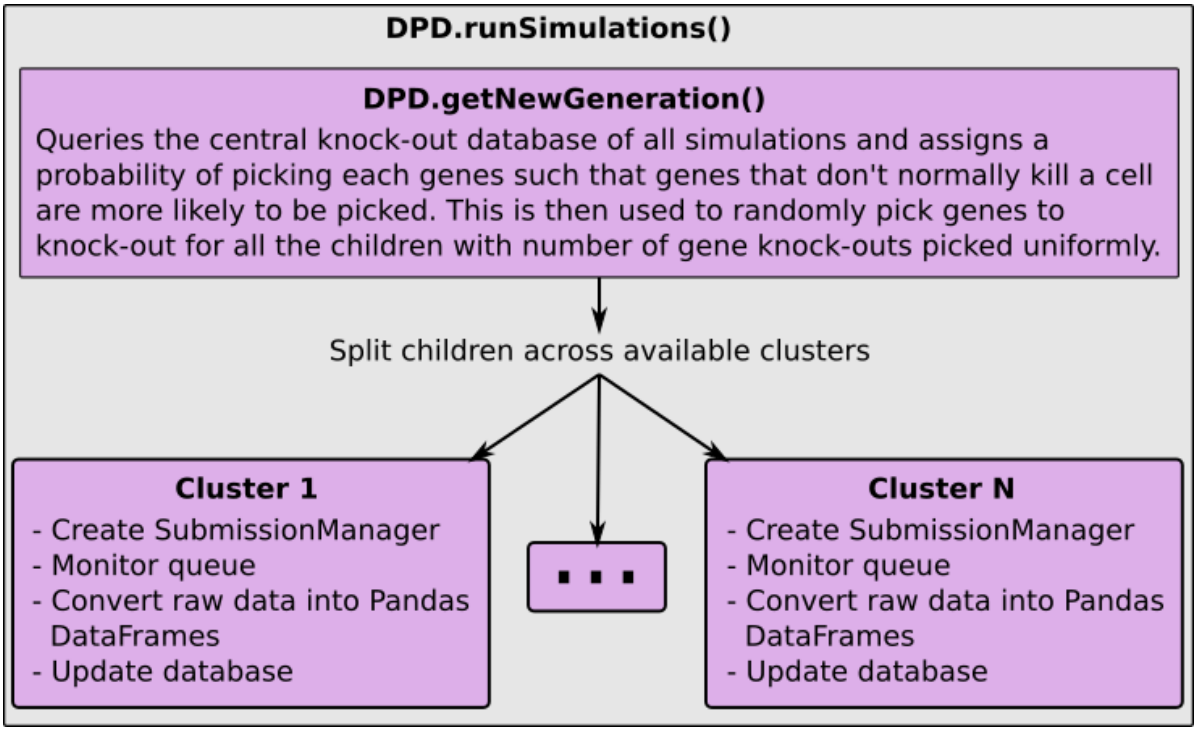


Figure 4.12: This diagram shows how the `runSimulations` method is implemented in the `DPD` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes.

4.4.5 The `DPD` class

The `DPD` class derives its name from the acronym of *Dynamic Probability Distribution*. I designed a learning algorithm that had a probability distribution of picking genes. The algorithm would randomly create children based on this distribution, and once the simulations had completed, it would change the probability distribution based on what was learnt from the most recent simulations. Over time this distribution should get better and better at predicting reduced genomes (or optimising any other objective function). Unfortunately, there was not time to fully implement this, but this class was the initial prototype that only takes into account single gene probabilities and not conditional probabilities based on what else is either present or knocked-out of the genome.

Figure 4.12 shows how each generation of the `DPD` is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `DPD` class needs: • A

list of all the possible gene knockout set lengths. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method gets the percentage that every gene has been in a viable combination using the `getGeneProbabilityDistribution` method and then uses that to create a distribution for picking genes such that genes that are normally knocked-out in viable combinations get picked more often. This distribution is then used to, randomly, pick sets of genes to knockout in children. The number of knockouts in a child is determined by randomly picking an element, uniformly, from the list of all possible gene knockout set lengths passed to the class when it was initialised. The number of children to create is determined by the `getPopulationSize` method.

The `runSimulations` method defines the algorithm over one generation. In this case it is the same as the `geneticAlgorithm` as it creates a new generation using the `getNewGeneration` method (which is the big difference between the two algorithms) and then splits it across the available clusters and manages the submissions — for more a more detailed discussion of the method see section 4.4.2.

Instance methods:
The `getSizeOfKoSet` method takes the list of all the possible gene knockout lengths passed by the user at initialisation and a corresponding list of probabilities and then returns one of the gene knockout lengths randomly according to the list of probabilities. It is worth noting that currently the `getNewGeneration` is hard-coded to pass this method equal probabilities (i.e. a uniform distribution) but the method was given more versatility so that the algorithm could be extended in the future with the less effort.

The `getGeneProbabilityDistribution` method uses a child that inherits from the `Connection` class to connect to `ko.db` through the `ko_db` library to return the *gene viability history* by calling the `getGeneViabilityHistory` method. The gene viability history is a dictionary that contains the proportion of times that a specific gene was knocked-out and produced a living cell for every gene.

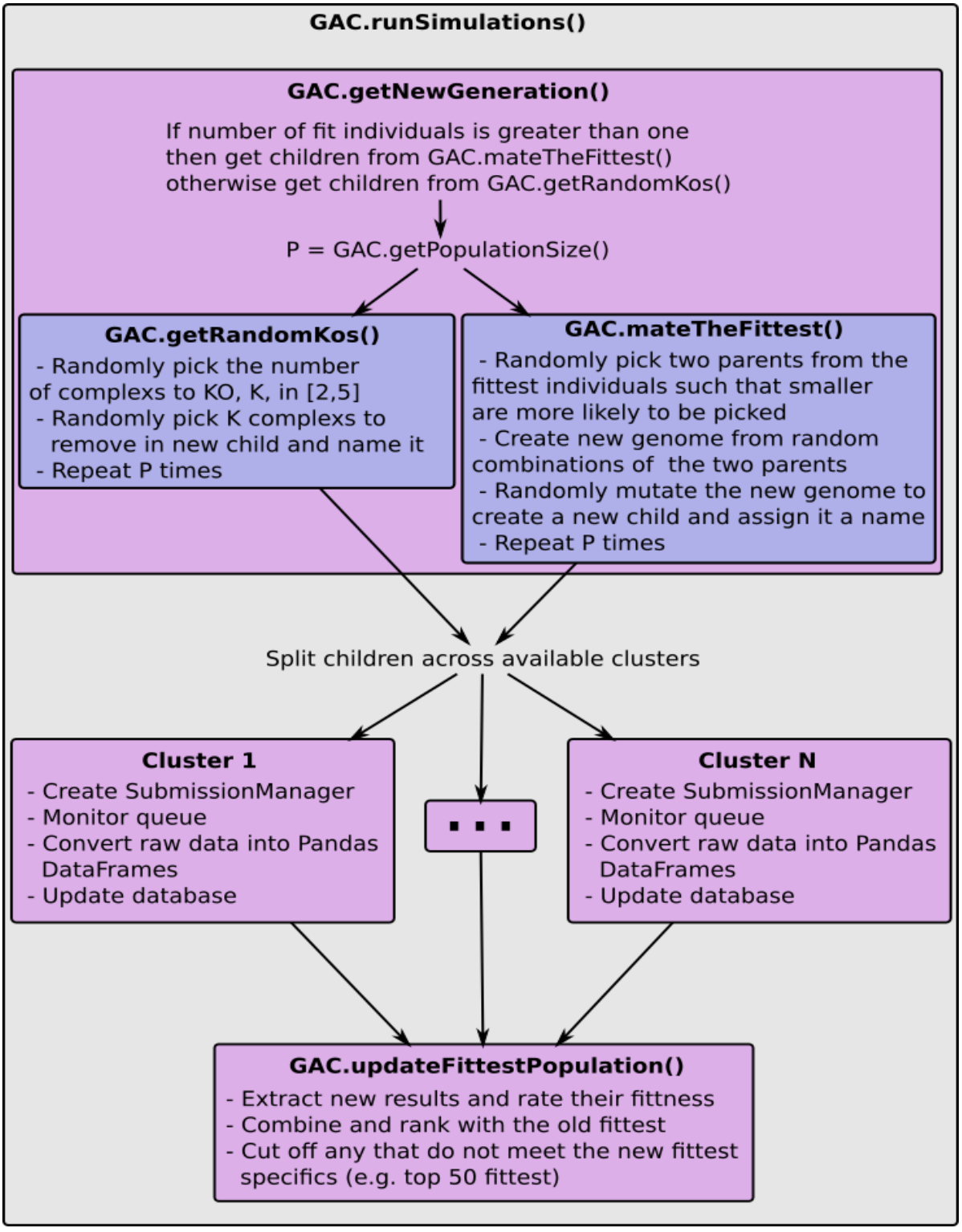


Figure 4.13: This diagram shows how the `runSimulations` method is implemented in the `GeneticAlgorithmWithComplexs` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here GAC represents the `GeneticAlgorithmWithComplexs` class.

4.4.6 The `GeneticAlgorithmWithComplexs` class

The `GeneticAlgorithmWithComplexs` class is the same as a standard genetic algorithm except this was created in an attempt to reduce the amount of gene combinations in the solution space. It assumes that a protein (and/or RNA) complex will only function in the model if it has all its components and so can treat the set of genes that make up a single complex as a single gene.

Figure 4.13 shows how each generation of the `GeneticAlgorithmWithComplexs` is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `GeneticAlgorithmWithComplexs` class needs: • The maximum number of individuals that can live on to the next generation. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method decides what method to call to generate the next generation of children. In this case, it calls the `mateTheFittest` method if the number of fit individuals is equal to the maximum number of fit individuals. If the algorithm has less than the maximum number of fit individuals and it is past generation 0, then it changes the child population size to be equal to that of generation zero (this is because generation zero normally has a bigger population because it is randomly guessing). Otherwise, it calls the `getRandomKos` method.

The `runSimulations` method defines the algorithm over one generation. In this case it is the same as the `geneticAlgorithm` as it creates a new generation using the `getNewGeneration` method (most of the difference can be found in the `getRandomKos` method) and then splits it across the available clusters and manages the submissions — for more a more detailed discussion of the method see section 4.4.2.

Instance methods:
The `getRandomKos` method works the same as the equivalent method in the `GeneticAlgorithm` class except it randomly picks complexes to knockout rather than individual gene codes. If a complex is chosen, then all genes that relate to a component of that complex are knocked-out. Genes whose products do not form a complex are treated as individual genes.

All other methods in this class are the same as the `GeneticAlgorithm` class and so will not be repeated here.

4.4.7 The `GeneticAlgorithmKnockIn` class

The `GeneticAlgorithmKnockIn` class works similarly to the `GeneticAlgorithm` class except it starts with a reduced genome and knocks genes in.

Figure 4.14 shows how each generation of the `GeneticAlgorithmKnockIn` class is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `GeneticAlgorithmKnockIn` class needs: • The maximum number of fit individuals that are allowed to survive each generation. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class. • The probability that a mutation occurs whilst creating a child. • The genes knocked-out in the initial organism. • A string telling the algorithm which probability distribution should be used to decide how many gene knockouts a child should have.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method decides what method to call to generate the next generation of children. In this case, it calls the `mateTheFittest` method if the number of fit individuals is greater than eleven. Otherwise, it calls the `getRandomKos` method.

The `runSimulations` method defines the algorithm over one generation. In this case, it is the same as the `GeneticAlgorithm` as it creates a new generation using the `getNewGeneration` method and then splits it across the available clusters and manages the submissions — for more a more detailed discussion of the method see section 4.4.2.

Instance methods:
The `getRandomKos` method works similarly to the equivalent method in the `GeneticAlgorithm` class, however, it randomly picks what genes to put back into the genome. Whilst the `GeneticAlgorithm` only has the option to choose the number of genes to knockout from the modified exponential

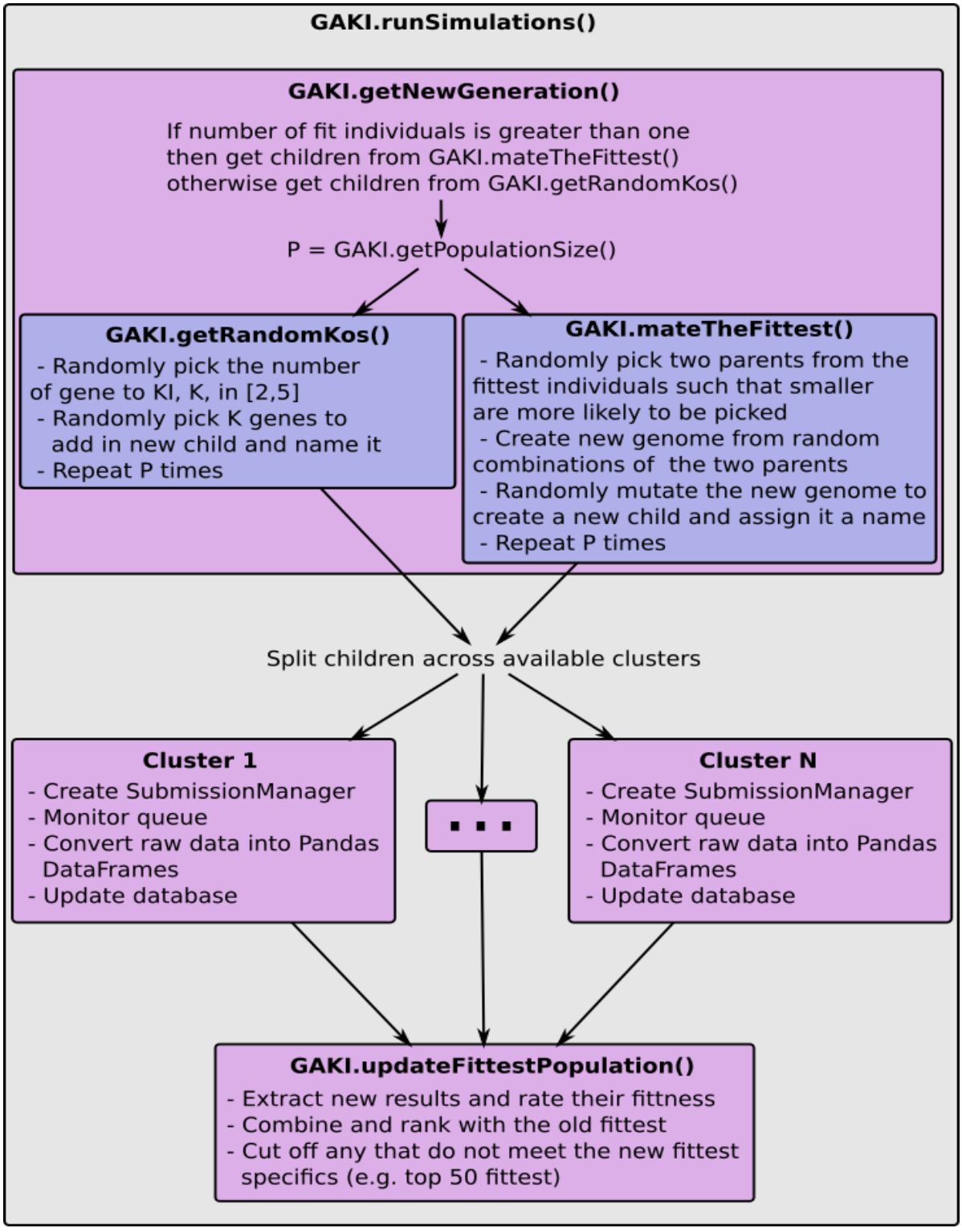


Figure 4.14: This diagram shows how the `runSimulations` method is implemented in the `GeneticAlgorithmKnockIn` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here GAKI represents the `GeneticAlgorithmKnockIn` class.

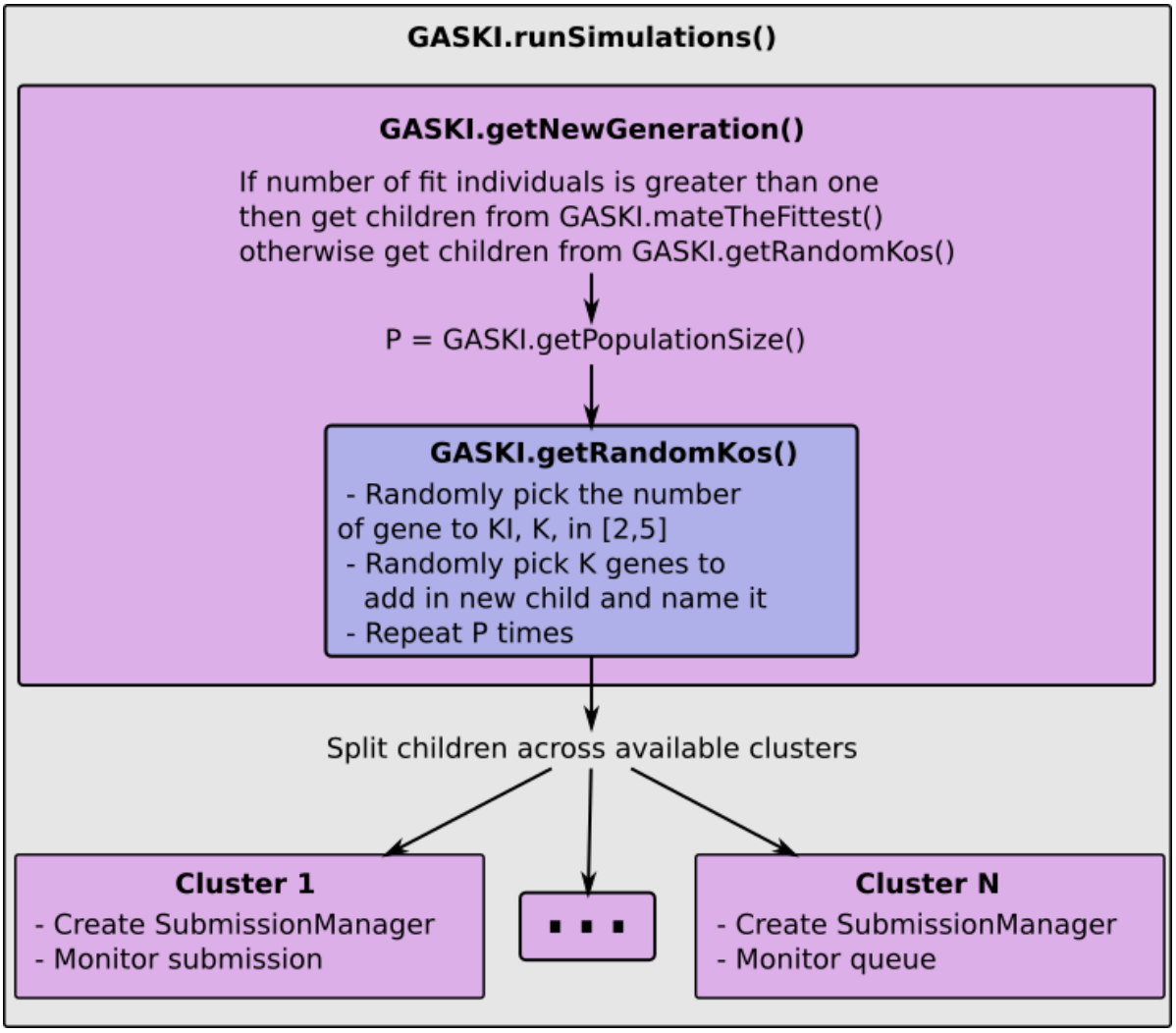


Figure 4.15: This diagram shows how the `runSimulations` method is implemented in the `GeneticAlgorithmSimpleKnockIn` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here GASKI represents the `GeneticAlgorithmSimpleKnockIn` class.

distribution, this algorithm chooses how many genes to add to the genome from either a uniform distribution or the modified exponential distribution (as decided by the user) with parameter of 10% of the size the original knockout set.

All other methods are the same as the `GeneticAlgorithm` class and so will not be repeated here.

4.4.8 The `GeneticAlgorithmSimpleKnockIn` class

The `GeneticAlgorithmSimpleKnockIn` class is exactly the same as the `GeneticAlgorithmKnockIn` class except the `updateFittestPopulation` method is left blank. Leaving this method blank has the effect that the algorithm never learns and so simply repeats the random guesses over and over again. It also only has the option to pick the number of genes to add back into the genome from a uniform distribution.

Figure 4.15 shows how each generation of the `GeneticAlgorithmSimpleKnockIn` class is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

4.4.9 The `GeneticAlgorithmFocusSet` class

The `GeneticAlgorithmFocusSet` class was created in order to explore certain specified subsets of genes.

Figure 4.16 shows how each generation of the `GeneticAlgorithmFocusSet` class is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `GeneticAlgorithmFocusSet` class needs: • The maximum number of fit individuals that are allowed to survive each generation. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class. • The probability that a mutation occurs whilst creating a child. • The set of genes that need to be investigated. • A minimum and maximum number of gene knockouts in an individual. There is a standard set and a safe set. The standard set gets used by default, but the safe set gets used if the algorithm is not finding enough viable solutions to progress to the next stage of the algorithm.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method decides what method to call to generate the next generation of children. In this case, it calls the `mateTheFittest` method if the number of fit individuals is greater than thirty. Otherwise, it calls the `getRandomKos` method. Additionally, if it calls the `getRandomKos` method later than generation 0 and there are no survivors, or there are survivors, but it is after generation 1 then it uses the safe gene knockout lengths. Otherwise, it uses the normal ones.

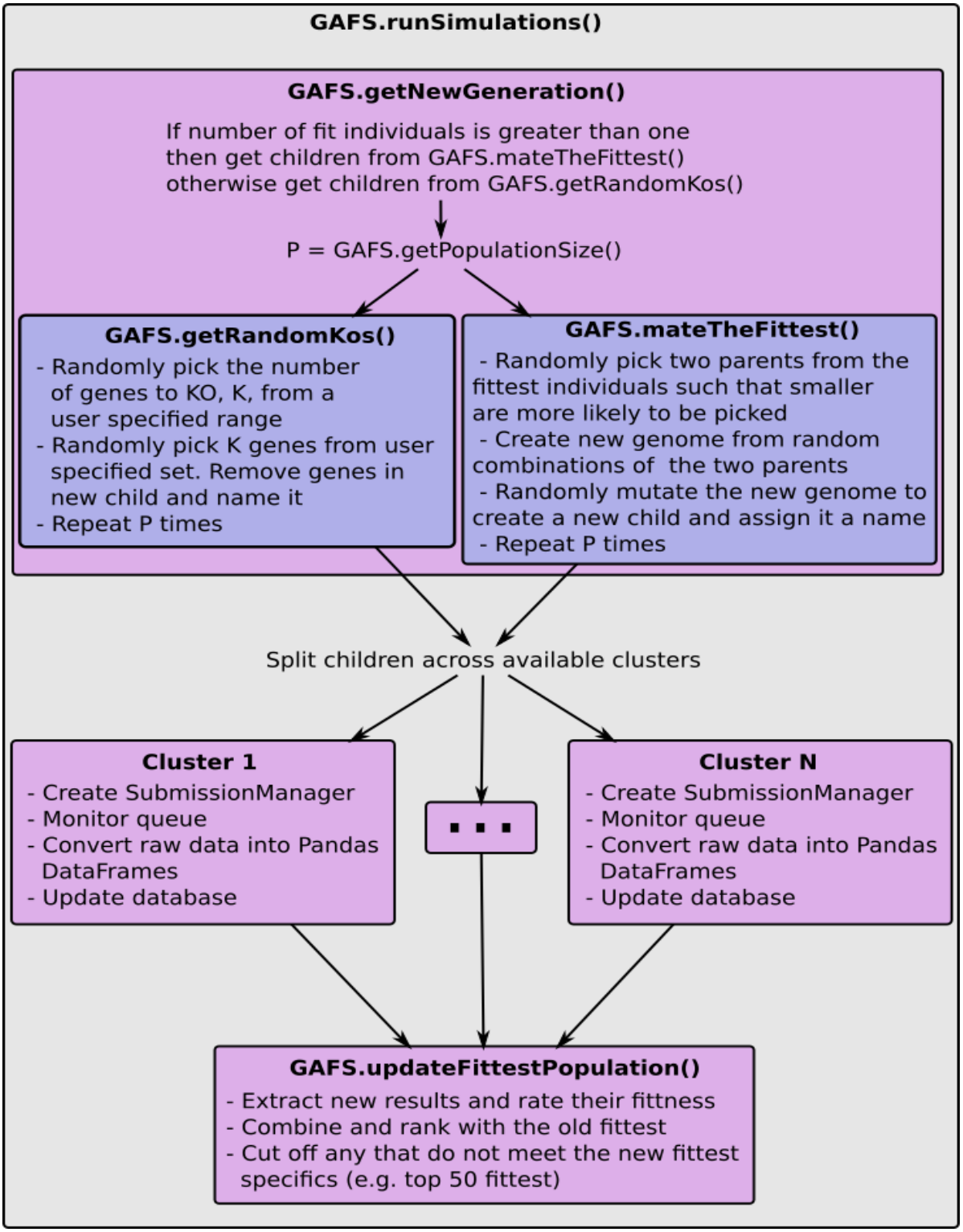


Figure 4.16: This diagram shows how the `runSimulations` method is implemented in the `GeneticAlgorithmFocusSet` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here GAFS represents the `GeneticAlgorithmFocusSet` class.

The `runSimulations` method defines the algorithm over one generation. In this case, it is similar to the `GeneticAlgorithm` as it creates a new generation using the `getNewGeneration` method and then splits it across the available clusters and manages the submissions — for a more detailed discussion of the method see section 4.4.2. The only difference in the method is that instead of splitting the children equally (plus a remainder) across each cluster this splits the children into groups of 200 and then assigns those groups of 200 equally over the available clusters.

Instance methods:
The `getRandomKos` method gets the number of children it needs to create and the set of genes that need to be investigated. This method then works the same way as the equivalent method in `GeneticAlgorithm` except rather than randomly picking any gene in the wild-type genome it only picks from the set of genes to be investigated given by the user.

Every other method in this class is the same as the `GeneticAlgorithm` class and so will not be repeated.

4.4.10 The `MixFocussSets` class

The `MixFocussSets` class was created to take groups of viable gene knockout sets where every individual from all groups is from the same organism but each group has genes knocked out from a disjoint set of genes, i.e. if one takes all the genes knocked-out in both group-1 and group-2 then there will be no gene in common between the two groups even though they are simulating the same organism. The knockout sets of each group can be added together to create new individuals. There is a difference here between mating two genomes which will be discussed in more detail in the below.

Figure 4.17 shows how each generation of the `MixFocussSets` class is executed and to see how this fits in with the entire process it should be compared to figures 4.1 and 4.6.

Initialisation:
In addition to the class variables needed to initialise the `MGA` class the `GeneticAlgorithmFocusSet` class needs • The maximum number of fit individuals that are allowed to survive each generation. • The number of times each simulation needs to be repeated. • The number of children needed in each generation. • A path that all simulations will be stored in. This is relative to the cluster base path which is given by the connection class. • The probability that a mutation occurs whilst creating a child. • A dictionary where the keys are the names of the groups and the values are sets of viable knockout sets.

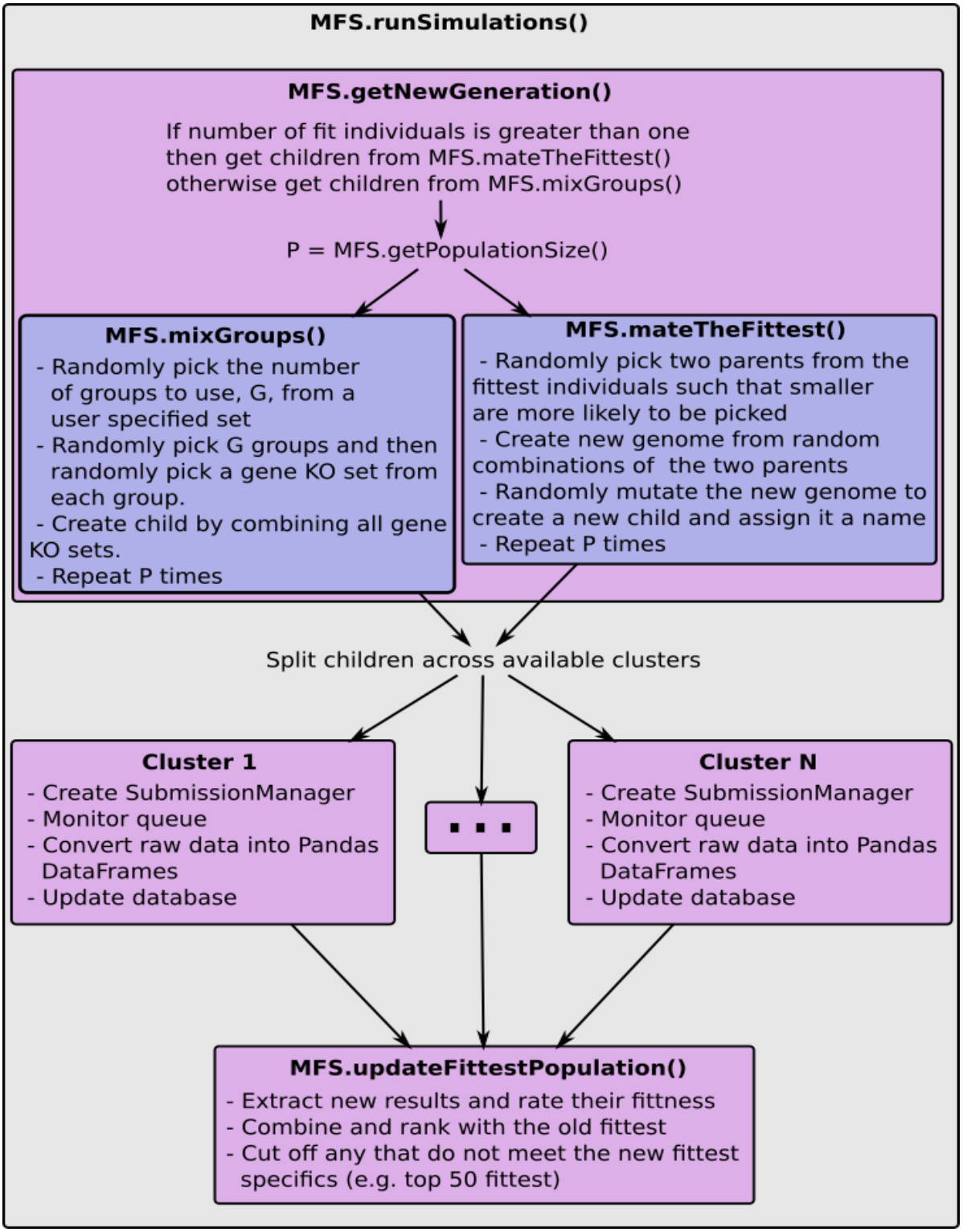


Figure 4.17: This diagram shows how the `runSimulations` method is implemented in the `MixFocussSets` class. The grey box contains everything that happens within the `runSimulations` method, whilst the lilac boxes contain any significant methods or classes called within the `runSimulations` method, and the blue boxes contain significant methods or classes called within the lilac boxes. Here MFS represents the `MixFocussSets` class.

Abstract methods:
The `getGenerationName` method returns the string 'genN' where 'N' is the current generation number.

The `getNewGeneration` method decides what method to call to generate the next generation of children. In this case, it calls the `mateTheFittest` method if the number of fit individuals is greater than eight. Otherwise, it calls the `mixGroups` method.

Instance methods:
The `mixGroups` method is given the minimum and maximum number of groups to combine and then selects a number of groups within that range from a uniform distribution. It then selects one gene knockout set uniformly from each of the selected groups and combines all the gene knockouts sets into one larger knockout set that defines a child for the next generation. The child-creation process is then repeated until there are enough children to simulate the next generation. It is worth noting the difference between *add* and *mate*. *Adding* refers to the fact that sets of gene knockouts are being *added* together — here as long as there are different genes knocked-out in each parent (which is the case because the set of genes are partitioned in the guess stage) the child genome is guaranteed to get smaller because the sets of genes being combined are disjoint. *Mating* refers to the *mating* of two parents in the same fashion of a genetic algorithm and so the two genomes are randomly mixed so to create one new child genome. This means that the child genome will inherit remaining genes as well as knocked-out genes and so the child genome could theoretically get larger, smaller, or stay the same.

All other methods in this class are the same as the `GeneticAlgorithm` class and so will not be repeated.

4.5 Data

The GDS enables complex data storage and processing tasks to be automated during the execution of an *in-silico* experiment. The *M. genitalium* whole-cell model has two data related problems and so this aspect of the GDS was utilised. The two data problems are the amount of data produced and the speed of data retrieval.

The raw simulation output of one life cycle is, sequentially in time, split into hundreds of compressed Matlab files. This means that if one wants to use a time series from a simulation, then every file needs to be opened, the relevant time series extracted and then all the sections of data concatenated to create the final time series. The data extraction process was timed (see

appendix A.7) by loading the growth rate of one simulation with 338 state files, and it took just over 31 seconds, not including the time it takes to load Matlab.

Due to the amount of data likely to be produced (see section 3.2.1) compared to the amount of space available (see section 3.2.2) it was decided that a single large central database would not be possible. However, because there are various different disk locations and RDSF is only appropriate for back-up some time was spent looking into having many smaller databases that could be moved about when necessary. Ideally, the smaller the database, the less unnecessary data is transferred, and so the quicker the analysis becomes. However, RDSF is designed for bulk transfer at the expense of smaller read/write operations (see section 3.2.2) and there is a non-zero storage overhead to an empty database. An initial investigation was made into creating a normalised SQLite3 database per simulation as this works out a convenient way to balance the database size trade-off. Unfortunately, SQL-based databases are not very efficient at storing data with regards to disk space. Additionally, members of the ACRC advised that data storage options were going to dramatically change in the near future (i.e. different storage places with larger capacity and different storage options). This combined with a lack of personal experience designing RDMSs or using the data meant that attempting a final database solution would not be an efficient use of time. Instead, it was decided to have a simple temporary data storage solution until more was known about the new data storage options and more experience using the data as well as designing/using RDMSs had been acquired.

It was decided to have three different data storage types. The first type was biological data about the model/organism and is stored in a normalised SQLite3 database on `Flex1`. The second type was an overview of simulation results that are stored in a normalised SQLite3 database on `Flex1`. Finally, key data from the raw simulation output would be extracted then converted into Pandas DataFrames and saved as Pickle files. All three processes are automatically done by the GDS and the implementation can be seen in section 4.2 and 4.3.

4.5.1 Biological data

Biological data was taken from the supplementary information of the *M. genitalium* whole-cell model publication and converted into a SQLite3 database. Due to the data being published in Excel spreadsheets, many characters were corrupted making the conversion process very time-consuming and so unfortunately not all data was transferred. However, there is still a significant amount of data relating to genes, RNAs, protein monomers, protein complexes, metabolites and metabolic reactions.

There are two spreadsheets in the supplementary information of the *M. genitalium* whole-cell model paper, mmc3.xls and mmc4.xls, that contain biological information about the model.

These will be referred to as MMC3 and MMC4, respectively.

In addition to the *M. genitalium* whole-cell model data, Joshua Rees's single gene knockout data, classification and comparison against Karr's single gene knockout classification was added to the database (see section 3.1.4).

There are 49 tables in this database, and so there is no easy way to present the data or the schema in one go, and so the schema will be split into sections of related tables. The sections will be tables related to genes, transcription units, protein monomers, macromolecular complexes, reactions and metabolites. Later, section 4.6.3 will provide some examples of actual data, and for a more in-depth understanding of the structure of the database and the data contained in it the reader is invited to explore the actual database which is called `static.db` either directly or through the GDS.

4.5.1.1 Genes

Tab *S2G-Gene disruption strains* of MMC3 contains data about single gene knockout tests that Karr et al. performed on the *M. genitalium* whole-cell model and this is contained in the `GeneKOData` table of static.db which can be seen with other data related to genes in figure 4.18.

All the other tables in Figure 4.18 come from tab *S3J-Genes* of MMC4 which contains data about the genes in the *M. genitalium* whole-cell model. These tables contain data about every gene's codes and names. It also contains the position and other genomic details like length, direction and sequence. There are details about homologs in other *Mycoplasmas* and some model organisms. There is empirical expression data for each gene under normal, cold and heat shock conditions.

Figure 4.19 is the table schemas for data taken from Rees's single gene knockout experiments on the *M. genitalium* whole-cell model. The `rees_10x_1ko_test` table contains the results of 10 repetitions of each single gene knockout supplied by Rees and the `rees_chalkley_19_1ko_comparison` table the summary of this and comparison against Karr's single gene knockout experiments taken from 3.1.4. It's worth noting that normally there would be a foreign key linking the `id` columns of the `rees_chalkley_19_1ko_comparison` and `rees_chalkley_19_1ko_comparison` tables to the `id` column of the `genes` table however these tables were created *on-the-fly* through the Pandas library in Python and so the foreign keys could not be put on and there has not been time nor reason to add it manually yet.

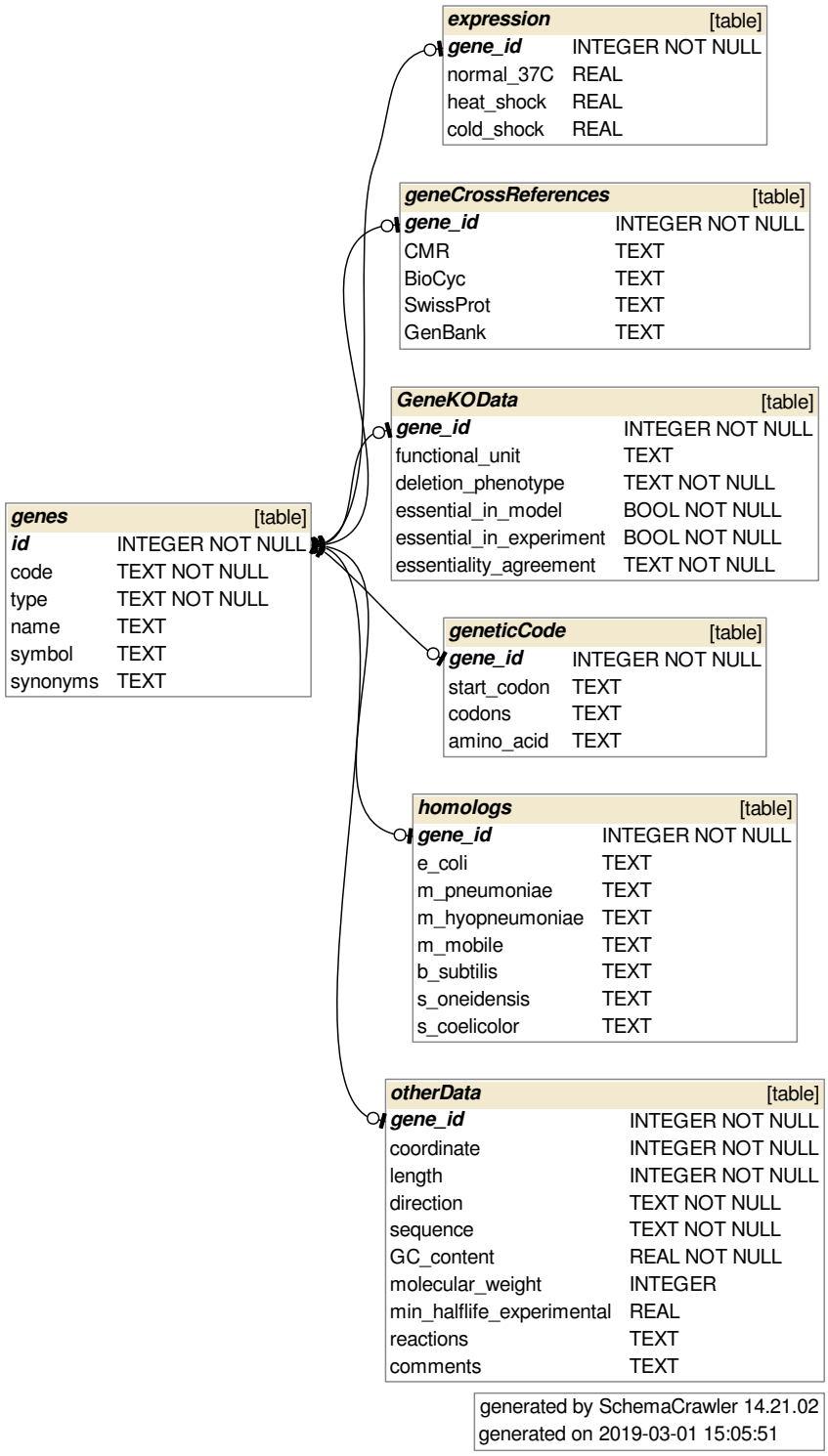


Figure 4.18: Database schema for data related to genes. All data is taken from MMC4 except for the `GeneKOData` table which is taken from MMC3.

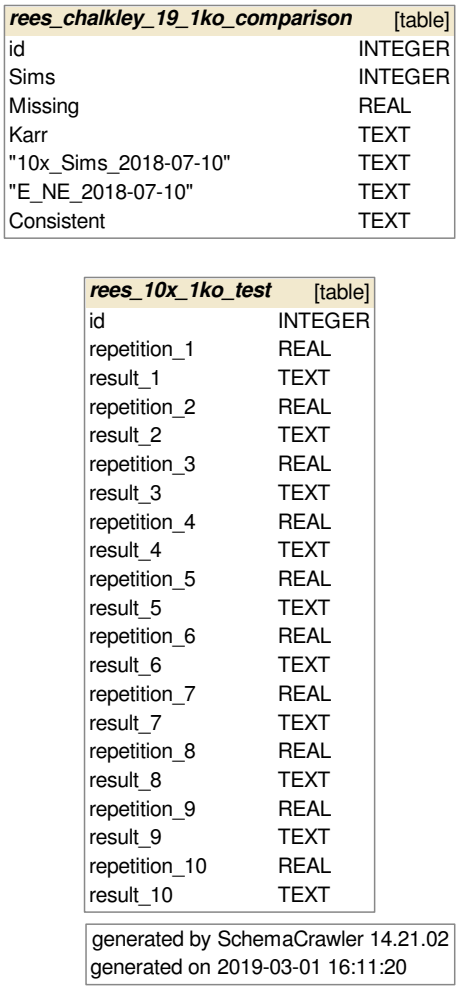


Figure 4.19: Database schema for data related to the comparison of single-gene knockouts between our simulations and Karr et al.[101].

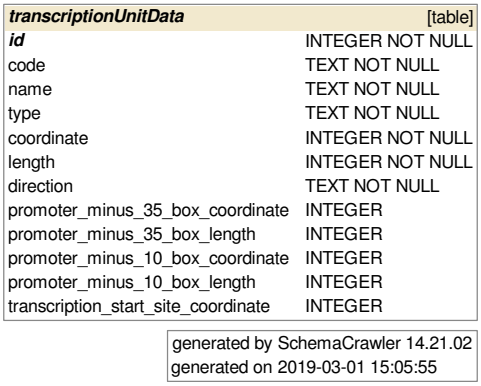


Figure 4.20: Database schema for data related to transcription units in the MG whole-cell model. All data is taken from MMC4.

4.5.1.2 Transcription Units

Tab *S3K-Transcription units* of MMC4 contains data about the transcription units in the *M. genitalium* whole-cell model and the tables related to this in static.db can be seen in figure 4.20. The table contains names, codes and other descriptors as well as structural information like length, direction, and promoters.

4.5.1.3 Protein monomers

Tab *S3M-Protein monomers* of MMC4 contains information about the protein monomers in the *M. genitalium* whole-cell model and the tables related to this data in static.db can be seen in figure 4.21. The data is split into 8 tables with each one dedicated to one of the following, identifiers, classifications, DNA footprint, folding, physical properties, signal sequence, structure and other.

4.5.1.4 Macromolecular complexes

Tab *S3N-Macromolecular complexes* of MMC4 contains information about macromolecular complexes in the *M. genitalium* whole-cell model and the tables related to this data in static.db can be seen in figure 4.22. The data is split into 5 tables with each dedicated to one of the following identifiers, classification, DNA footprint, folding, and other.

4.5.1.5 Reactions

Tab *S3O-Reactions* of MMC4 contains information about reactions in the *M. genitalium* whole-cell model and the tables related to this data in static.db can be seen in figure 4.23. The data is split into 9 tables with each one dedicated to one of the following identifiers, cross-references, enzyme

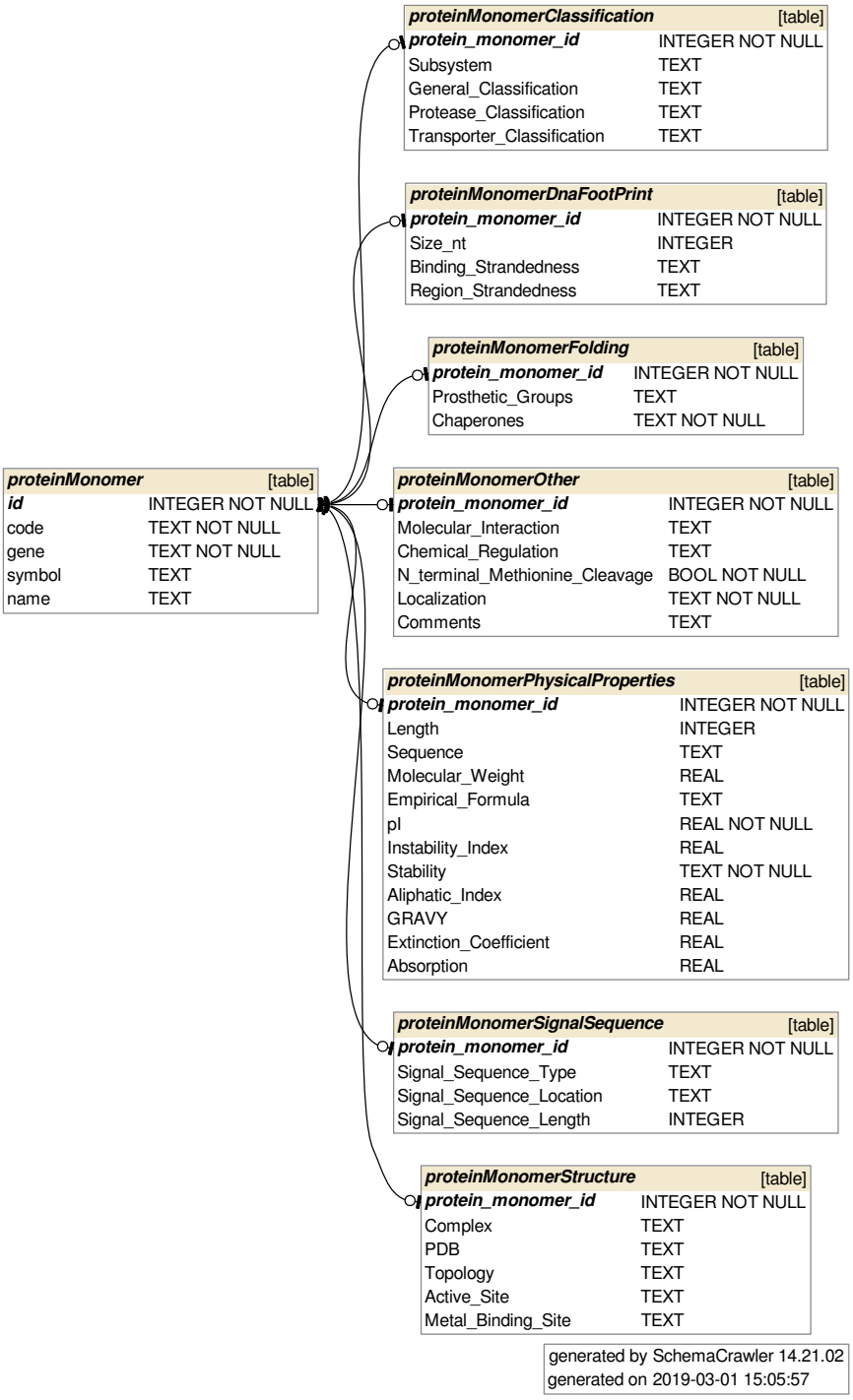


Figure 4.21: Database schema for data related to protein monomers in the whole-cell model. All data is taken from MMC4.

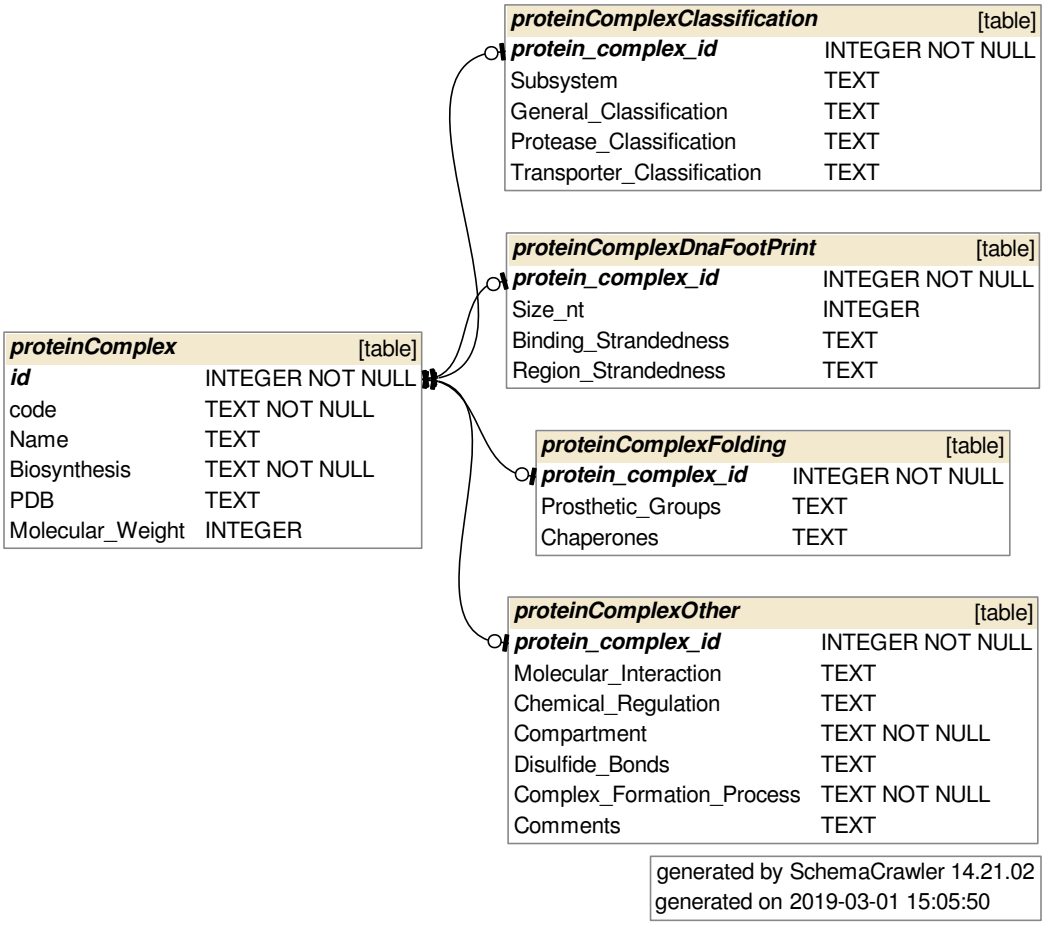


Figure 4.22: Database schema for data related to macromolecular complexes in the MG whole-cell model. All data is taken from MMC4.

catalysis, forward kinetics, backward kinetics, bounds on kinetics, macromolecule modification, thermodynamics and other.

4.5.1.6 Metabolites

Tab *S3G-Metabolites* of MMC4 contains information about reactions in the *M. genitalium* whole-cell model and the tables related to this data in static.db can be seen in figure 4.24. The data is split into 6 tables with each dedicated to one of the following identifiers, cross-references, categories, exchange bounds, physical properties, and other.

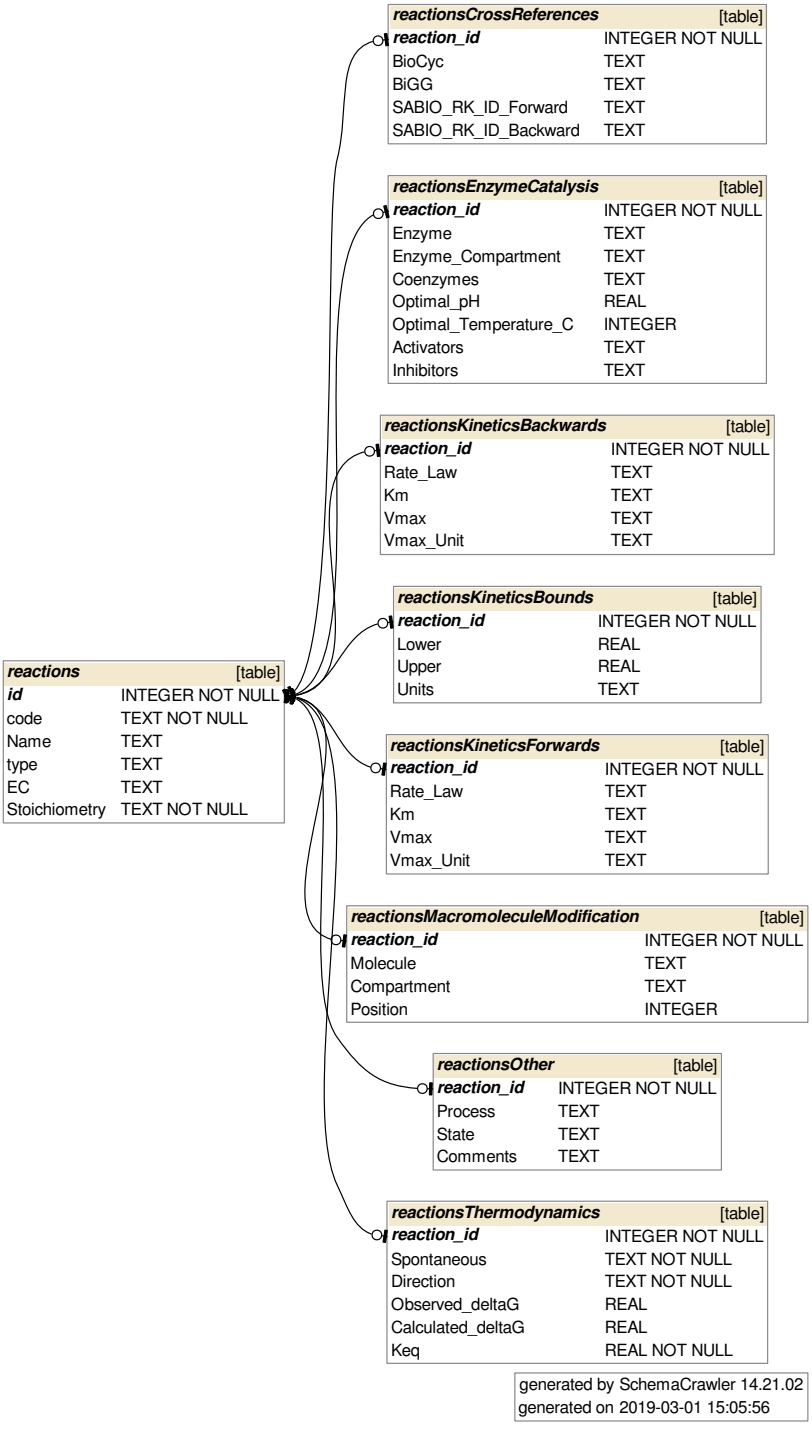


Figure 4.23: Database schema for data related to reactions in the MG whole-cell model. All data is taken from MMC4.

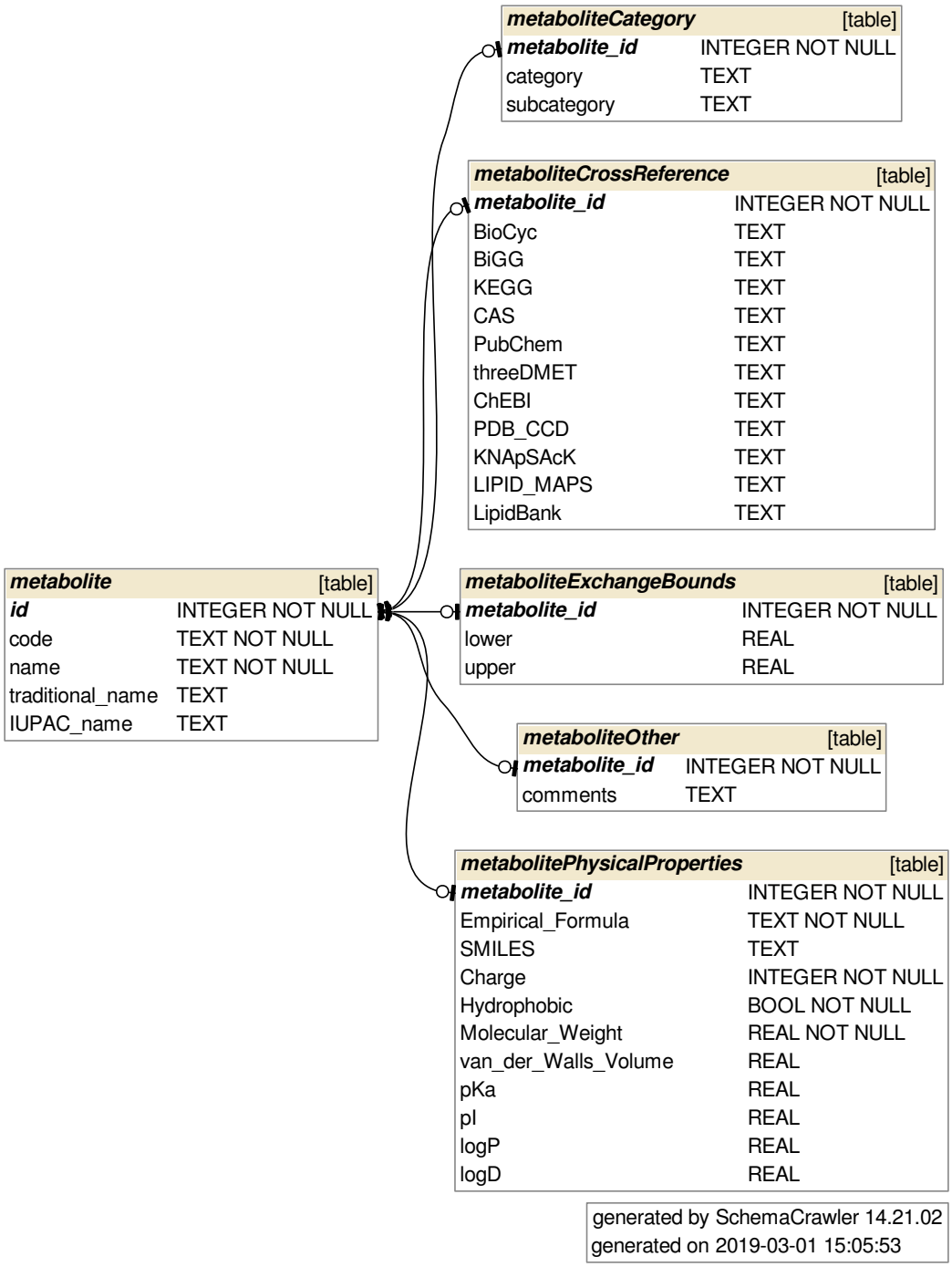


Figure 4.24: Database schema for data related to metabolites in the MG whole-cell model. All data is taken from MMC4.

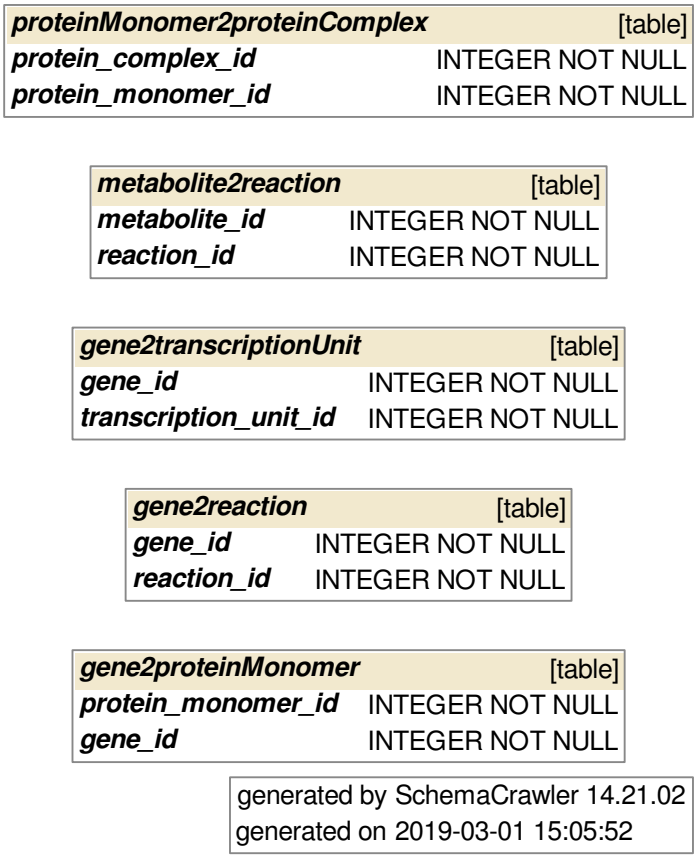


Figure 4.25: Database schema for tables that connect different aspects of biology. Moving from top to bottom it connects protein monomers to protein complexs, metabolites to metabolic reactions, genes to transcription units, genes to metabolic reactions, and genes to protein monomers.

4.5.1.7 Connector tables

The connector tables were manually created in order to link the main groups of genes, transcription units, protein monomers, macromolecular complexes, reactions and metabolites together in a traditional RDMS manner (i.e. with lookup tables). Each table takes two of the groups and connects the ID of one with the ID of another. Figure 4.25 shows the tables without the foreign keys to the corresponding group's identities table since it has had to be separated due to the number of tables in the database. Due to time constraints these tables have not been populated but section 4.6.3 shows that it is possible to do this for all the tables except `gene2transcriptionUnit` and `proteinMonomer2proteinComplex` . I believe that it is possible to do these two tables but have not had time to implement it.

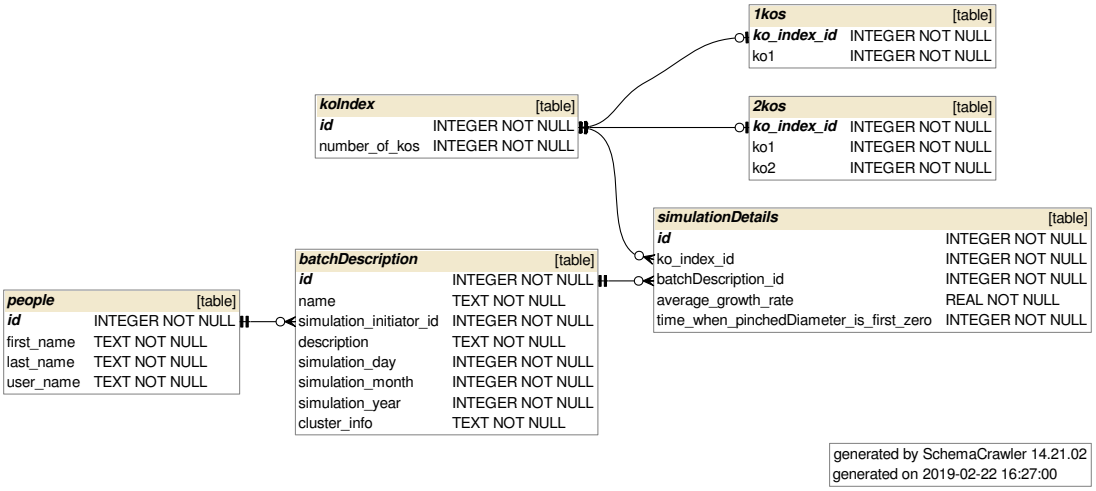


Figure 4.26: A diagram illustrating the schema of the `ko.db` database. There are 525 tables that record what genes were knocked-out of the genome for a given *in-silico* experiment. There is a table for each possible size of knockout, i.e. 1-525. This results in too many tables to visualise and so 523 tables have been removed, leaving just the single and double knockout tables as examples. All tables not related to gene knockouts remain in the diagram. Diagram produced by SchemaCrawler (see section 2.3).

4.5.2 Simulation overview

An SQLite3 database was created to store an overview of simulations. There are 529 tables in this database, one for the person that executed the batch of simulations; one that contains the name of the batch, the date of execution and the cluster that it was executed on; one that records the average growth rate and second of division of each simulation; and one that holds a unique ID of every genome simulated and the number of gene knockouts for that genome. The other 525 tables hold the genes knocked-out of each genome simulated where the table “1kos” hold all the single gene knockouts, “2kos” holds all the double gene knockouts, all the way up to 525-gene knockouts.

529 tables would be too much to visualise but all the tables related to gene knockouts have the same structure except the number of columns is the number of gene knockouts plus 1 (for the ID). Using this to exclude redundant information, a diagram of the database schema and how the tables relate to each other was created. Single and double knockout tables were included as an example, and the 3 to 525-gene knockout tables were excluded, and the resulting diagram can be seen in figure 4.26.

4.5.3 Simulation data

For reasons discussed at the beginning of this section the raw simulation output underwent a significant amount of data processing, the implementation of this can be seen in the `batch_jobs` module described in section 4.3 and requires an additional module on the computer cluster. Key data was removed, converted into Pandas DataFrames and saved as Pickle files - the rest of the data was deleted. Several saving options were programmed into the GDS for convenience. These had four categories, each stored in a separate Pickle file, a general overview, mature RNA counts, mature protein monomer counts, mature protein complex counts, and metabolic reactions fluxes and the user can choose any combination of these to store.

There is a Python library called H5py that has methods to read Matlab files. Unfortunately, this occasionally results in an error for the compressed files (version 7.0) which is the standard version for the *M. genitalium* whole-cell model. A change was made to the *M. genitalium* whole-cell model so that it saves the data in to uncompress Matlab files (version 7.3). H5py is then able to read the files to extract the necessary data and then saves it. The GDS requires the `extract_matFile_data_v73` module to be present of the computer cluster in order to do this data processing.

4.6 Analysis

Despite all the support provided by the GDS and the supporting databases, analysis of data is still not straight forward. Analysis of how all a cell's functions interact as well as problems with working with large datasets and combining simulation data with biological data it not easy without the GDS. The analysis module of the GDS was created to provide a framework and tools for analysing data from the *M. genitalium* whole-cell model with an aim to easily to generalise to other models/organisms as they become available.

This section will be split into the following sections, *an analysis framework*, *comparing and visualising genomes*, and *interpretting genomes biologically*.

4.6.1 An analysis framework

Whilst there are many aspects to the analysis of whole-cell models, for genome design it was decided to start with analysis of the genome. Since our *in-silico* experiments focus on the scale of the gene it was decided to start with the resolution of the gene. The code is structured so that within the main GDS folder there is an analysis folder that contains the `genome` module that contains the `Genes` class.

The `Genes` class creates a framework to analyse genomes at the resolution of the gene. Three

representations of a genome were proposed in section 3.3 — *Genome*, Θ, *knockout*, K, and *present*, Υ. Each representation can be useful in certain types of analysis, storage and simulation and so the class needs to be able to move between all three representations. However, it was decided that the main representation would be the genome representation that will also be quickly available within an instance of the class.

An instance of the `Genes` class is initialised with a GDS `connection` to the GDS databases, a list of all the possible gene codes, genome data, and data input type. The `connection` instance is used so that there can be a single remote source of data meaning that teams can work together knowing that all their analysis is equivalent and that when someone finds an update to the biology it can be uploaded to everyone all in one go. With datasets this large/complex it is essential that everyone is working from the same source since it only takes relatively small divergences to strongly inhibit individuals comparing/combining work later on. A list of all possible genes is passed because an analysis may want to study all the genome or only the protein-coding genes or some other subset of genes. Data on the genome(s) can be in any of the three forms or `None` where `None` creates an empty instance of the class. The data input type is a string that tells the class what representation the genomes are in. All genome data is stored in the instance of the class and then makes a second copy in genome form (if the passed data is not already in genome form).

The previous paragraph showed that a subset of a genome could be visualised. This is a collection of genes and not a genome anymore, however, for the rest of this section the term genome will still be used and it is implied that this could actually be a subset of a genome.

Genome data must be passed in the standard Pandas DataFrame form when in genome form where the column headers are the names of each genome and the indexes are gene codes. If in knockout or present form then it must be in the form of a dictionary that has the name of the genomes as keys and lists of gene codes (or IDs) as the genes that either remain or are knocked-out.

There are methods of the class that enable manipulation of the instances of the class and the genome data and so each will be discussed.

Should one want to add genomes to an existing `Genes` instance then the `appendGenomeDf`, `appendNameToKoIdSetDict`, or `appendNameToKoSetDict` methods can be used depending on the genome form of the input. If a batch of genomes from Pandas DataFrames were stored as a Pickle file then those genomes can be added to an existing instance of the `Genes` class by using the `appendGenomeDfFromPickleFile` method.

Often a user has a series of genomes stored in a text file in genome form. In this case, the `appendGenomeFromTxtFile` method will take the file name and automatically append it to the instance of the `Genes` class. Similarly, one may have genomes like this but as a Python list of binary strings. These can be converted to a Pandas DataFrame and passed to the `convertRawGenomeToDf` method to be added to an instance of the `Genes` class. If a SQLite3 database outputs genomes as binary strings then genomes can be appended to an existing instance of the `Genes` class by passing the path to the database and a corresponding SQL query to the `appendGenomeFromDb` method.

There are various methods to convert between forms that are named with the convention convert<form1>To<form2>. For example the method `convertRawGenomeToDf` converts a Pandas DataFrame of binary strings into the standard Pandas DataFrame form and the method `convertKoDictToGenomes` converts from knockout form to genome form. The method `convertKoIdDictToCodes` converts present or knockout representations from IDs to codes.

4.6.2 Comparing and visualising genomes

It is useful to be able to compare the difference between two or more genomes from the same organism, but there are many different perspectives one can take, so methods were created to investigate this.

The obvious starting place is to compare what genes are present or knocked-out between genomes. The `getGeneCodesBySimilarityClassification` method take two genomes with corresponding genome names and returns a dictionary that contains all the gene codes that were present in both genomes, the codes that were knocked-out in both genomes, and the codes that genomes did not agree should be present or knocked-out. The `summeriseEssentialityByGene` method returns a dictionary similar to the previous method except that it acts on all the genomes in genome form currently stored in the instance of the class. This dictionary only has three keys though. The *universal_essential_codes* are all the gene codes that were present in every genome, the *universal_non_essential_codes* are all the gene codes that were knocked-out in every genome, and the *transient_codes* are all the gene codes that were present in some genomes but knocked-out in others. The `plotDistributionOfGeneEssentiality` method plots the proportion of genomes that a gene was present.

It was decided that it would be useful to be able to visualise how similar/different sets of genomes were and so distance metrics were created. A distance metric, in this case, is simply defined as a metric that is 0 if the two genomes are identical and gets larger the more different they are. All distance metric methods take two genomes in genome form and are static methods

since they do not need access to class attributes, and it makes it easy to pass to other methods and even other classes.

The most obvious way is to look at the proportion of genes that are the same but this actually has three interpretations. The `proportionSimilarityDistance` compares each gene in both genomes. Any genes that are either both present or both knocked-out are scored with a 1 and otherwise, scored as 0 and then the proportion of ones is calculated to indicate what proportion of genes were the same. In order to turn this into a distance metric, we take one minus the proportion of genes that are the same. This distance metric results in 0 if the genomes are identical and 1 if not a single gene is the same. A similar metric is made but only comparing the genes present, `proportionKIsDistance` , or the genes knocked-out, `proportionKOsDistance` .

In statistics and machine learning it is common to compare two sets using the ARI (see Chapter 2.3). Since a genome is a set of genes, a distance metric was created using the ARI which is 1 minus the ARI. The distance metric results in 0 if the genomes are identical, 2 if they are opposite and around 1 if it is about as different as an average random guess. This is refered to as the *ARI distance metric* and is called by the `ariDistance` method.

Once one has a distance metric for two objects it is common to compare a set of objects by creating a distance matrix which calculates the distance between every object (see Chapter 2.3). Distance matrices are normally symmetric (i.e. $\delta(a,b) = \delta(b,a)$) and will be in all cases for this chapter. The `createDistanceMatrix` method takes 3 arguments, the distance function, the child process chunksize, and the number of cores. The distance function is a function that takes two genomes and returns a distance metric. A distance matrix is then created by calculating all the distances between all the genome form genomes in the instance of the class. Since creating a matrix is $\mathcal{O}\left(n^2\right)$ this quickly takes prohibitively long to calculate and so the task was coded to be able to calculate it using multiple cores in parallel. The number of cores argument specifies how many cores a user wishes to use - the default is set to `None` which will automatically use all available cores on the computer. Often transferring data to cores can slow down a parallel calculation and so the child process chunksize option allows the user to specify how many genomes to be transferred in one go. The default is set to 15 which is what generally performed quite well on a particular laptop, but users should investigate what is best for their machine. Due to potentially long calculation times a progress bar is shown for convenience. This only works on the local computer, attempts were made to add options that send the calculation to the computer cluster, but there are currently unresolved technical difficulties. Other improvements would be to cut the computation time in half by exploiting the symmetry of the matrix or potentially creating even larger improvements by writing a C++ function to create the matrix and creating bindings so it can be called from Python.

Visualising a distance matrix can be problematic since the number of dimensions is given by the number of objects. One common way is to use a dendrogram. The `plotDendogramOfMGSs` method creates a dendrogram from a distance matrix.

The dendrogram is useful for seeing clusters of similar objects, but as the numbers of objects or clusters increase it quickly becomes hard to interpret. The `plotDistanceMatrixWithPca` method takes a distance matrix and uses PCA to reduce the number of dimensions to two and creates a scatter chart (see Chapter 2.3 for more information about PCA). The method provides a much more intuitive way to visualise large sets of genomes but PCA is a *lossy* compression algorithm and so should be used with care and the dimensions can be hard to interpret since they are linear combinations of dimensions. Examples of these plots can be found in chapter 6.

It became clear that there is no easy way to literally visualise a genome and so heat map was modified so that each column represents a genome and each row represents a gene and the colour represents if the gene is present or knocked-out - this visualisation can be created using the `plotGenomes` method. However, there are too many genes to see anything other than a qualitative view. To improve this one can order the genomes by similarity by using the `orderGeneomesWithDbscan` and/or look at smaller subsets of genes.

4.6.2.1 Further comments on distance metrics

Information theory has been used to create distance metrics, and so it is suggested that an investigation into finding an information theoretical representation of the distance between genomes.

A common property of the distance metric used is that there is a strong relationship between the length of the genome and its similarity. This relationship makes sense because it can only be the same genome if it is of the same length, but there may be more to genome similarity. I believe that this effect may be strengthened by using symmetric distance metrics. It is common to use symmetric distance metrics, but perhaps this is not a good assumption in this case. For example, imagine that one genome is reduced by 10 genes and another genome is reduced by 200 genes, and instantly they are going to have quite large distance but what if all the genes that remain in the smaller genome also remain in the larger genome. In this case, it might be argued that the smaller genome is more similar to the larger genome, but the larger genome is more different from, the smaller genome. However, visualising non-symmetric distance matrices may be hard or even impossible, and so I suggest an investigation into the feasibility of this as if possible then could be a way of improving distance metrics of genomes.

All distance metrics discussed so far are very abstract and do not attempt to find a biologically meaningful representation of the distance between genomes. For example, the three distance metrics coded into the GDS discussed above assume that every gene is equal which may not be the case. For example, imagine a pathway or a function that involves x number of genes but as long as any y number of genes ($y < x$) remain in the genome then the cell lives which means that you have $\binom{x}{y}$ combinations of viable genomes, but there is very little difference between them. A viable genome that has genes knocked-out from a different pathway or function would be a very different genome. The problem with this is that lots of genes interact in different ways and so it may be hard to find some distance metric that incorporates it.

4.6.3 Interpreting genomes biologically

One major problem with the *M. genitalium* whole-cell model is that it is very hard to interpret the data biologically. In section 4.5.1 it was shown that a lot of important biological data was converted into machine friendly format and loaded into an SQLite3 database. The central database enables us to get the team's current biological knowledge automatically and in a machine-friendly format. In order to utilise this data to help us better interpret our results, methods were created in the anlysis part of the GDS that create bar charts and word clouds of sets of gene's functional groups. Also sets of genes could be automatically highlighted on KEGG maps through Cytoscape. However, no good example use-cases of these methods arose in the creation of this thesis and so have been omitted but the interested reader will be able to find the methods in the code.

4.7 Discussion

In this chapter we presented a python framework to enable multi-generation algorithms across multiple computing clusters, a bespoke data storage solution for gene knockout experiments using the whole-cell model of *M. genitalium*, and tools for genome analysis and biological interpretation of the whole-cell model of *M. genitalium*.

The framework for multi-generation algorithms was structured to enable it to be easily adaptable for different optimisations, models and clusters. We then added 3 HPC clusters, 1 model and 10 different optimisation algorithms.

Due to the large amount of data a temporary bespoke data storage solution was created to fit our situation and it was designed to be simple, distributable, compact, machine friendly, and quick/easy to access. The data was split across 2 SQLite3 databases and series of Pandas DataFrames stored as Pickles. The framework for multi-generation algorithms was adapted to transform all data into this format as and when simulations were performed.

The analysis part of the GDS was split into 2 sections, *genome analysis* and *biological interpretation*. Toy examples were used to demonstrate the function of the tools. The genome analysis was made up of presenting a Python framework to work with and manipulate genomes, then analysis tools were build upon this framework. This framework and tools were designed to be general analysis tools and so would work on any organism. The tools for biological interpretation, however, need to be model/organism specific and automatically communicate to with Cytoscape and static.db in order to interpret data.

Whilst the goals for GDS were largely met, there is still room for improvement. Whilst the code was structured so that it would be easy to adapt to different clusters, models and algorithms the structure is a little too simple and does not specify areas of code where some instance specific functions should go. This has resulted in the code, as new algorithms are added and extended the code becomes either messy with repeated code or complicated function calls. Whilst the GDS will be released open-source it currently lacks a user manual, tutorials and a community of users making it hard for users to get started. However, those able to use the GDS are rewarded with the ability to perform *in-silico* genome design experiments on scales previously not possible.

CHAPTER 5

MASSIVE *in-silico* EXPERIMENTS

This chapter has two themes. The first theme is covered in the first section (section 5.1) and looks at the theory behind the algorithms implemented in the GDS. The second theme is covered in the remaining sections and looks at the results gained from using the GDS. Chapter 4 looked at the design and implementation of the GDS and how it can easily be adapted for different algorithms, models or computers. The GDS enables *in-silico* experiments that are too big for a computer cluster - e.g. because the run-time is longer than the maximum walltime of the cluster's queueing system (around 2 months in our case for all three clusters), or because the number of cores required is more than what's available on any one cluster. This chapter will present the results of its use in enabling massive *in-silico* experiments of *M. genitalium* using the Karr et al.[52] whole-cell model.

5.1 Algorithm theory

The GDS is built around the concept of a *multi-generation algorithm*. The multi-generation algorithm is able to implement any kind of computational process that requires more computing

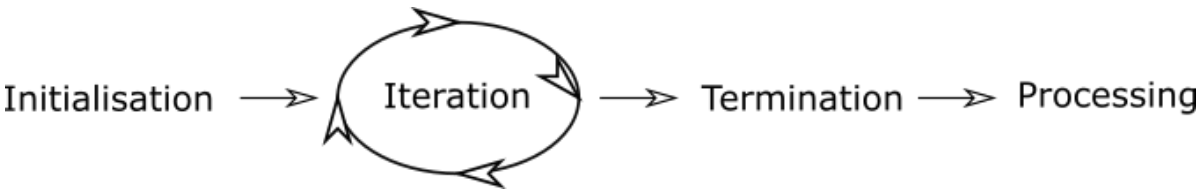


Figure 5.1: A flow diagram of the structure of a multi-generation algorithm. The iteration and the processing stages start and end the process and the iteration and termination stage define how many iterations are performed.

resources than what is available by splitting the process into subsets that are small enough to be run on the available computing resources and then combined to produce the end result. Each subset of computation is referred to as a *generation*. These subsets could be computed separately and collated at the end like calculating the average result of multiple independent simulations. Alternatively, the results of some subsets may become input parameters of other subsets like the generation of a gentic algorithm. The GDS assumes that a user's problem can be broken into subsets small enough to run on a single computational resource and so users should confirm that this is the case beforehand to avoid failure. It is also worth noting that it can still be used for lesser tasks that simply complete within the first generation. Figure 5.1 shows the four stages of a multi-generation algorithm. The *initialisation* stage sets everything up so that the algorithm can be performed on the resources available. The *iteration* stage is where the GDS splits the computation into subsets small enough to be run on the available resources and then runs them - it is worth noting that any subsets that don't require the previous generation's results can be run in parallel across multiple computational resources. The *termination* stage is how the GDS knows how to stop the iteration stage e.g. an optimisation algorithm might have converged to a solution. The *processing* stage then performs any computation that requires all of the *iteration* stage to finish first, like calculating the average result of all the iterative stages.

5.1.1 GA-type algorithms

This thesis uses the GDS in both ways described above (e.g. combining independent computational tasks and sequentially performing steps of an algorithm that is too time consuming for maximum simulation times of the available clusters). Whilst the GDS is flexible enough to work with any type of algorithm that fits the conditions described above (i.e. splitting the algorithm into subsets of computations small enough to work on the available resources) most of the algorithms explored here are either a genetic algorithm or are derivative in some way and so this will be described under the term *GA-like algorithms* and the common properties will be explored.

GA-type algorithms have 5 main components.

1. Chromosome encoding.
2. Fitness function.
3. Optimisation objective.
4. Selection.
5. Sexual reproduction.

Chromosome encoding in the GDS swaps between the three different but equivalent representations of a genome specified in section 3.3 depending on which is most appropriate. These

are the genome (Θ), knockout (K), and present (Υ) representations.

The **fitness function** is the genome length of dividing individuals and has three equivalent forms

$$\begin{aligned} f(\Theta_i) &= \delta_i \times \|\Theta_i\| \\ f(\Upsilon_i) &= \delta_i \times \|\Upsilon_i\| \\ f(K_i) &= \delta_i \times \|\Delta(K_i)\| \end{aligned} \tag{5.1}$$

where δ_i is a binary variable that represents if individual i divides or not,

$$\delta_i = \begin{cases} 1, & \text{if } \Theta_i \text{ produces a dividing cell} \\ 0, & \text{if } \Theta_i \text{ does not produce a dividing cell.} \end{cases}$$

$\|\Theta_i\|$ & $\|\Upsilon_i\|$ is the genome length of individual i and Δ is the difference operator that converts the knockout representation into the present representation. One can see that f is only equal to zero if the cell does not divide or if there are no genes present in the genome.

The **optimisation objective**, $\mathcal{O}$, has three equivalent forms

$$\begin{aligned} \mathcal{O}(\Theta_i) &= \min_{f(\Theta_i)>0} [f(\Theta_i)] \\ \mathcal{O}(\Upsilon_i) &= \min_{f(\Upsilon_i)>0} [f(\Upsilon_i)] \\ \mathcal{O}(K_i) &= \max_{f(K_i)>0} [f(K_i)]. \end{aligned} \tag{5.2}$$

The fitness function and optimisation objective help guide **selection**. Once a generation of genomes have been tested, the fittest individuals are selected to mate and create the next generation. The non-dividing genomes are removed. Two genomes are then selected from the remaining genomes such that smaller genomes are more likely to be picked. To describe this mathematically we will use the knockout representation of genomes as it is most intuitive. Let the set of all the fittest individuals of generation, G_i, be $F_i = \{K_j \mid f(K_j) > 0 \wedge K_j \in G_i\}$. Thus we define the probability of picking individual j from F_i as

$$\mathbb{P}(X = K_j) := \frac{f(K_j)}{\sum_{K_k \in F_i} [f(K_k)]}. \tag{5.3}$$

Sexual reproduction is split into two sections, *recombination* and *mutation*.

The algorithm for sexual reproduction is most intuitive when using the genome representation, Θ_i. Let two individuals, K_i & K_j, be randomly picked from the fittest set of generation i, F_i, to mate using equation 5.3. These are converted from knockout to genome representation using the inverse-transform operator and will become the parents of a new child,

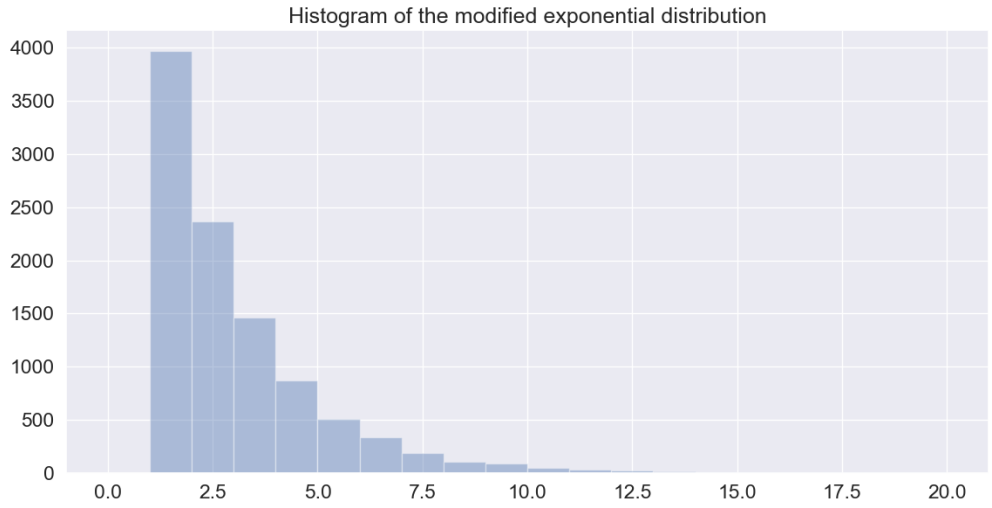


Figure 5.2: Histogram of the modified exponential distribution. 10,000 data points were sampled from the modified exponential distribution to create this histogram. The data had a minimum value of 1, a maximum value of 20, a mean value of 2.54050, and a standard deviation of 2.00049.

$P^1 = \Omega(K_i) = \{\theta_1^1, \theta_2^1, \ldots, \theta_{\|\Gamma\|}^1\}$ and $P^2 = \Omega(K_j) = \{\theta_1^2, \theta_2^2, ..., \theta_{\|\Gamma\|}^2\}$[①]. In order to create the child a random number in the range $x \in [1, \|\Gamma\|]$ is uniformly picked, this indicates how much of the child genome will be made up of parent-1. A subset of size x is uniformly picked from P^1, $C^{P^1} \subset P^1$, and the remaining genes will come from parent-2, $C^{P^2} \subset P^2$ - naturally $C^{P^1} \cap C^{P^2} = \emptyset$. C^{P^1} and C^{P^2} are combined to make a new genome, $C = C^{P^1} \cup C^{P^2} = \{\theta_1^{k_1}, \theta_2^{k_2}, \ldots, \theta_{\|\Gamma\|}^{k_{\|\Gamma\|}}\}$ where $k_m = 1$ if the m^{th} gene is taken from parent-1 or $k_m = 2$ if taken from parent-2. In algorithm form, two parents, P_1 and P_2, perform sexual reproduction by the following

1. Pick a uniformly distributed random integer, x, in the range $[1, \|\Gamma\|]$.
2. Sample x θ_i (without replacement) from P_1.
3. Take the remaining θ_j from P_2.
4. Combine both sets of genes to form the genome of the child.

Once sexual reproduction has occurred then random mutation must happen. Here a random number of genes are *flipped* by adding genes that were knocked-out or knocking-out genes that were present (i.e. θ_i is a binary variable and so can be *flipped* by adding 1 modulo 2, $\theta_i \rightarrow (\theta_i + 1) mod 2$). The number of genes to be flipped needs to be picked randomly in the range

[①]The user defines the total gene set, Γ, and thus its size on initialisation of the algorithm. The gene set may be the entire wild-type genome or could be some subset that excludes genes that the user does not want to knockout. In the case of this thesis, $\|\Gamma\| = 358$, which is the number of the characterised protein-coding genes minus one that tends to cause the simulation to crash. For more information on how the genes were chosen, see Chapter 3.

$[1, \|\Gamma\|]$. Whilst it is possible to pick this number uniformly, we expect that as the number of mutations increase the number of genes knocked-out become more random (i.e. lessons learnt from our algorithm are being forgotten due to random mutation) and experience from chapter 3 tells us that a few random knockouts even in a wild-type cell very quickly become lethal. Thus one might assume that uniformly picking the number genes to mutate in the range $[1, \|\Gamma\|]$ is often going to result in killing the cell. One could stick with a uniformly distributed random variable and narrow the range, say to $[1,5]$ but this significantly reduces the *randomness* of mutation. In order to balance this trade off it was decided to use a modified exponential distribution. The exponential distribution is a monotonically decreasing function which is qualitatively what we are looking for because large amounts of mutations are possible but less likely. The *rate parameter*, λ, sets the rate of decay of this continuous function. The exponential distribution produces, real-valued, random variables in the range $[0,\infty]$ but we would like the range to start at 1 as the user specifies at initialisation the probability of a mutation occurring and we want integer values of genes to knockout. Additionally, we would like the probably of mutating all (or more) genes to be almost impossible. To fulfil these requirements the modified exponential distribution was created. Let X_λ be an exponentially distributed random variable with rate λ then the number of mutations, M, can be calculated by

$$M = \lfloor X_\lambda + 1 \rceil \tag{5.4}$$

where $\lfloor\rceil$ denotes that the value calculated within is rounded and $\lambda = 2$.

Equation 5.4 is approximated by the `mateTheFittest` method of the GDS and it's implementation is discussed in more detail in section 4.4.2. Ten thousand samples were taken from the modified exponential distribution, and a histogram of the data can be seen in figure 5.2. The sample minimum, maximum, mean, and standard deviation are 1, 20, 2.54050, and 2.00049, respectively.

Figure 5.3 shows the iterative process of a GA-type algorithm. The observant reader may notice that iterating this process assumes that there is always a previous generation that can mate to produce offspring for the new generation. The first generation will, by definition, have no previous generation to gain it's population from. In order to start with some kind of population for mating, the algorithm is *seeded* with an initial population - this may be referred to as the *seed generation* or *generation 0*. This is a significant factor that distinguises many of the algorithms implemented and will be discussed in the next section along with other differences.

5.1.1.1 Standard genetic algorithm

Figure 5.4 shows how the GDS genetic algorithm reduces genomes. The algorithm can be split into two stages, the seed stage and the mate stage. The mate stage mimics evolution by natural selection such that smaller viable genomes are favoured. The mate stage, however, needs parents

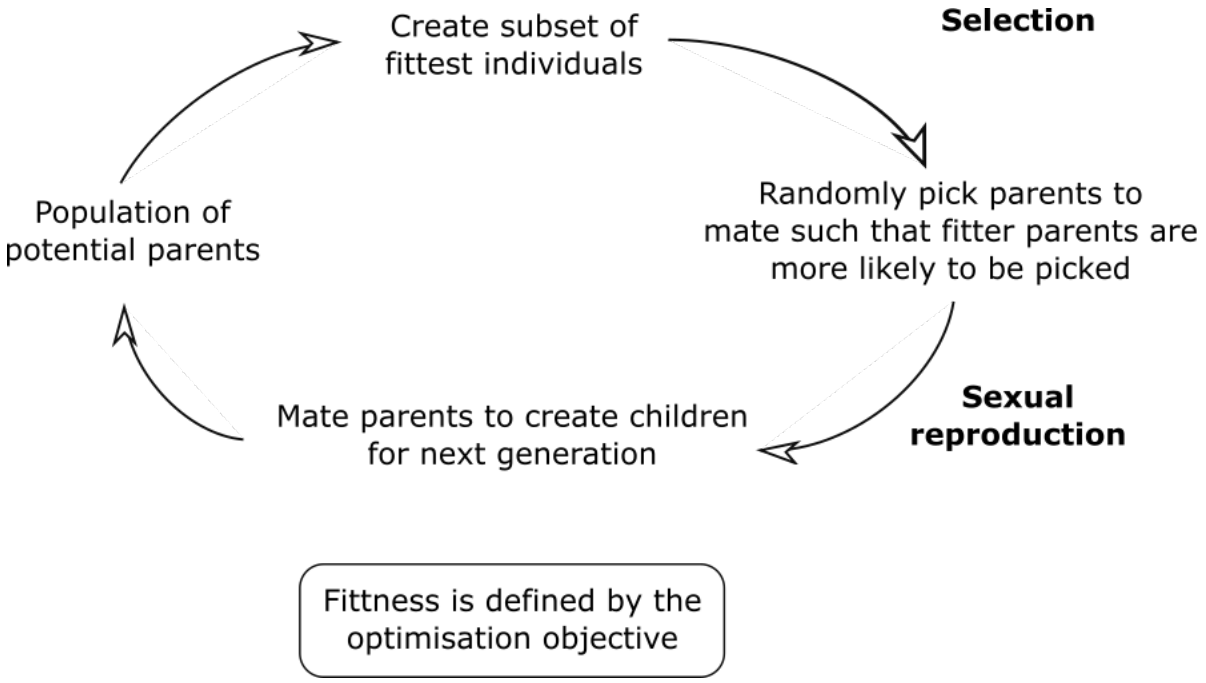


Figure 5.3: A flow diagram of the main part of a GA-type algorithm shows how fittness/optimisation, selection, and reproduction are combined.

to mate and create children and so these are randomly created in the seed stage and passed to the mate stage. The random gene knockouts that create the seed children are restricted to between 2 - 5-gene knockouts because it becomes hard to randomly guess viable gene knockout combinations greater than 5 within 200 simulations which we expected to be the standard generation size (see chapter 3).

5.1.1.2 Grouping genes by gene-product complexes

Using biological knowledge as shown in section 3.1.3 had not shown enough benefit to be implemented on the GDS. However, it became clear that the biggest problem for the genetic algorithm was the huge size of the solution space and so an attempt was made to try and reduce the solution space. Since gene products can combine to produce complexes (e.g. a ribosome) it was decided to group all the genes by their functional molecules. For example, let's consider a heterodimer, H, made up of the protein from gene A, pA, and the protein from gene B, pB. In previous genetic algorithms gene A and gene B could be knocked out individually (2 combinations), together (1 combination), or not at all (1 combination), but in this algorithm, both genes are treated as one so either both get knocked-out together (1 combination) or neither get knocked-out (1 combination). This reduces the numbers of combinations of genes (from 4 combinations to 2 combinations) and thus reduces the overall solution space. This is a specific example but the result can be generalised as follows. Let an arbirary complex be made up of x gene products then there are

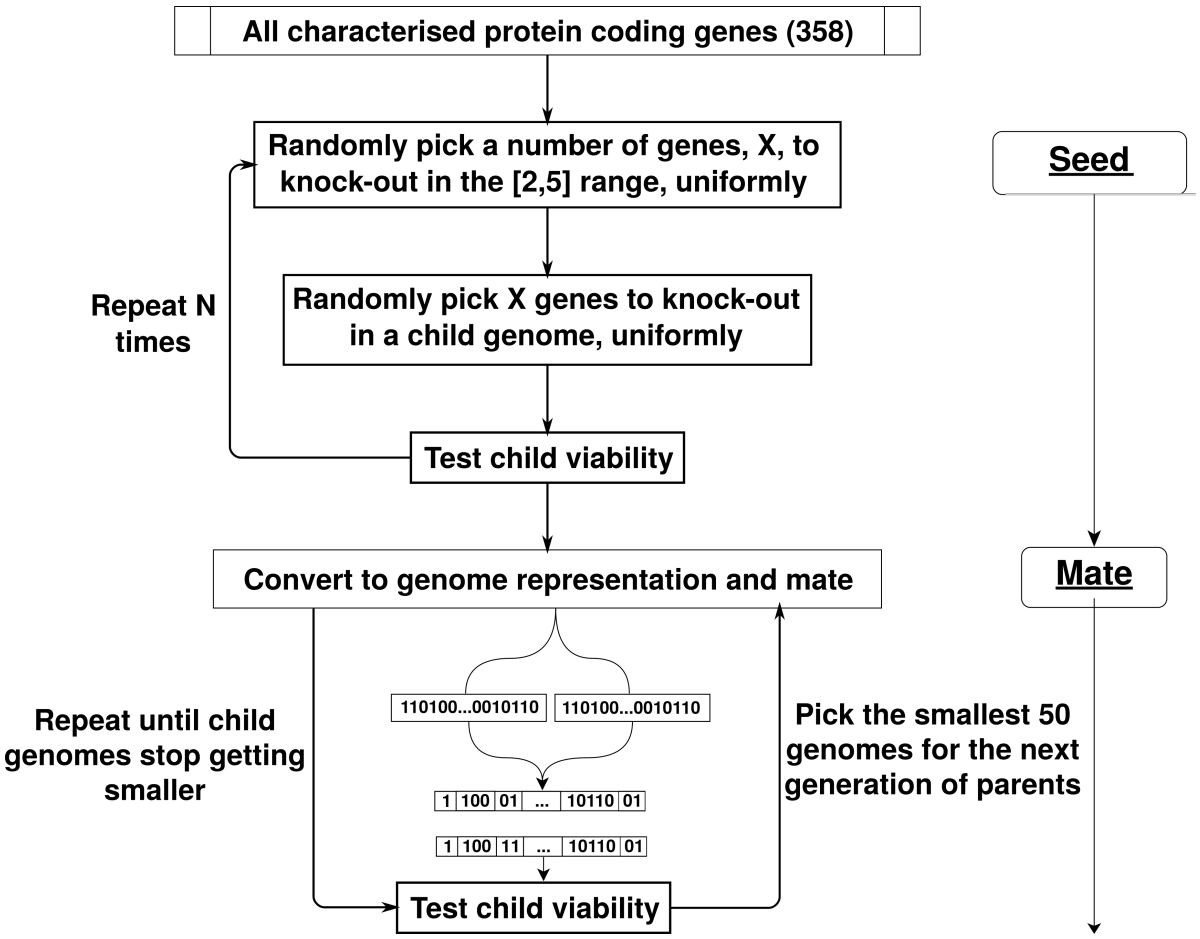


Figure 5.4: A flow diagram of how a genetic algorithm reduces genomes in the GDS. The seed stage randomly generates, N, children with between 2 - 5-gene knockouts, where, N, is given by the user. When enough viable children have been found then the algorithm moves to the mate stage where evolution by natural select is performed on sequential generations of children so that the children converge to the smallest viable genome.

$\sum_{i=0}^{x}\left[\binom{x}{i}\right]$ combinations of gene knockouts that affect that complex but in this algorithm you can only knockout all of the genes or none of the genes and so reduce the number of combinations by $\sum_{i=0}^{x}\left[\binom{x}{i}\right]-2$ combinations.

Data about the gene products were taken from the supplementary information of [52] and can be found in machine-friendly form in `static.db` (see section 4.5 for more details). For more information about the technical implementation see section 4.4.6 and the complex name to gene codes can be seen in method `getComplexToGeneCodesDict` .

5.1.1.3 Seed with theoretical predictions

Instead of seeding the genetic algorithm with random genomes with 2 - 5-genes knocked-out, all the non-viable minimal genome predictions discussed in chapter 3.1.4 were used. These genomes did not produce a viable cell in the wholecell model of *M. genitalium* but it was assumed that predictions from peer-reviewed literature is likely to be a better approximation of a minimal genome than randomly guessing. In order to help produce viable reduced genomes these were mated with larger, but viable, genomes in the hope that the genetic algorithm would (at some point, by random chance) replace the non-viable parts of the non-viable genomes with viable parts from the viable genomes. The viable genomes were taken from all viable genomes found (at the time) from the two original genetic algorithms and are described in sections 5.2.1.1 and 5.2.1.2. The non-viable genomes are much smaller genomes with between 45 - 269-gene knockouts, and the viable genomes are much larger with 2 - 27-gene knockouts.

5.1.1.4 Guess, add, and mate algorithm

GAMA is also an implementation of a genetic algorithm with non-standard seeding in the GDS. GAMA stands for the Guess, Add, and Mate Algorithm and the component names that make up the acronym are stages in the algorithm where the guess and add stages are seed stages, and the mate stage refers to the mating performed in a standard genetic algorithm. Figure 5.5 depicts the stages of the GAMA algorithm, and one can see the similarities to the standard algorithm except with two more complex seed stages rather than one standard random one.

In order to encourage the seed generation to seed the mate generation with the smallest genomes possible an assumption was made that genes changing essentiality were, on average, significantly less likely than it staying the same when reducing a genome. This assumption leads to our two *principles of static essentiality*: 1. knocking-out combinations of only non-essential genes are significantly more likely to produce viable cells than ones that knockout at least one essential gene, and 2. given multiple sets of viable gene knockout combinations, then combining the knockout combinations is likely to also produce a viable combination.

Guess

The guess stage uses item 1 of the principles of static essentiality by only looking at non-essential genes to knockout. However, the pool of 151 non-essential genes[②] still has a massive solution space ($\sim 2.9 \times 10^{45}$)[③] and at the time gene addition algorithms were showing signs, anecdotally, that they were struggling to find viable gene knockout combinations from subsets of non-essential genes (the gene addition algorithms are discussed in section 5.3) and so there were concerns that

[②] It is now agreed that there are 147 singularly non-essential genes but at the time of algorithm design there was uncertainty about the essentiality of some of the genes, and so an extra 4 genes were added.

[③] $\sum_{k=1}^{151} \binom{151}{k}$ where $\binom{n}{k}$ was estimated using the `scipy.special.comb` function in Python.

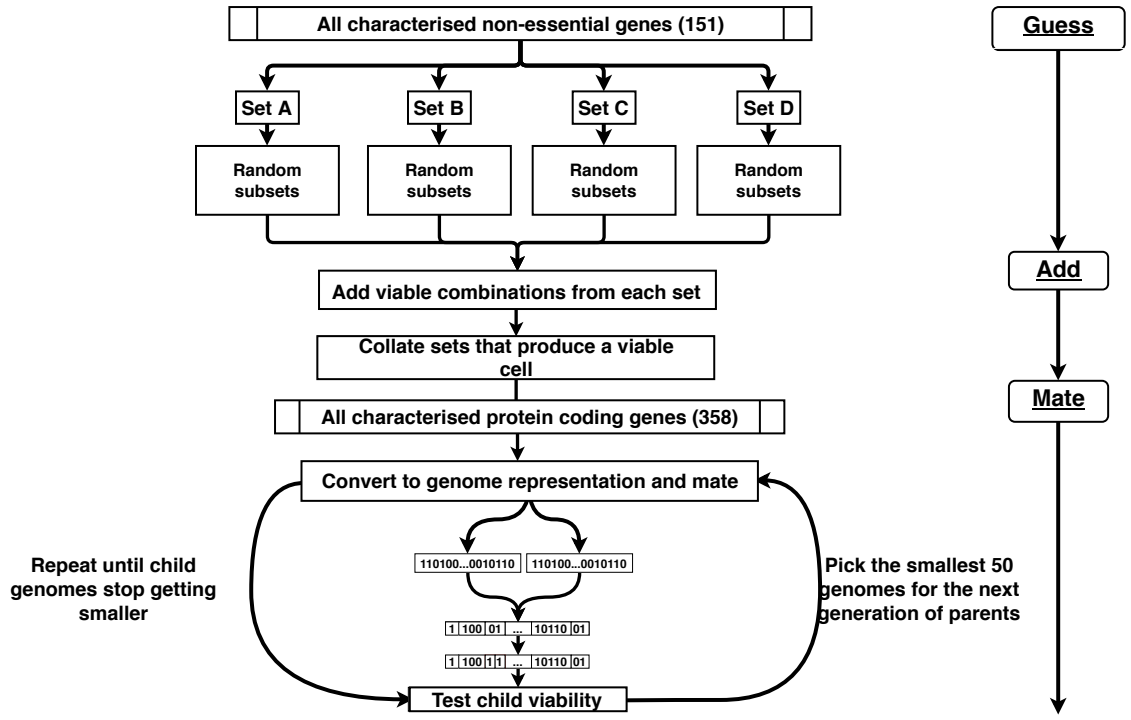


Figure 5.5: The Guess, Add, and Mate Algorithm (GAMA) attempts to seed a genetic algorithm (i.e. the mate stage) with genomes as small as possible in order to converge to a minimal genome as fast as possible. The guess stage partitions all the non-essential genes into 4 groups and then picks 400 random subsets from each group. Each subset represents a gene knockout combination which is simulated. The add stage randomly picks between 2 and 4 of the partitions and then randomly picks one viable knockout combination from each one and combines them to create one larger knockout combination which is then simulated. This process is repeated 2,000 times. The mate stage, instead of being seeded by standard random guesses, takes its seeds as the 50 smallest viable genomes from the add stage. This stage then acts as a normal genetic algorithm (see chapter 2.3 and section 4.4.2) but it now works on all the essential and non-essential 358 protein-coding genes. This figure was taken from [101].

the guess stage might struggle to find large viable gene knockout combinations. Since combinations are super-linear, if a set is partitioned into non-overlapping subsets, then the sum of all the combinations of the subsets will be significantly less than all the combinations of the super-set. The shrinking of the solution space also means that we are missing out on potential large viable combinations that span multiple partitions. This can be somewhat compensated for by utilising item 2 of the principles of static essentiality in the add stage (see section 5.1.1.4 for more details). It was decided to partition the 151 non-essential genes into four, roughly, equally sized subsets. These subsets had 36, 37, 38, and 40 gene codes in them and so the sum of the combinations of them are significantly smaller than the combinations of 151 ($1.58 \times 10^{12} \ll 2.9 \times 10^{45}$).[④]

[④] $\sum_{n \in N} \sum_{k=1}^{n} \binom{n}{k}$ where $N = \{36, 37, 38, 40\}$ and $\binom{n}{k}$ was estimated using the `scipy.special.comb` function in Python.

Four hundred random sets of genes (with replacement) were picked from each partition creating 1,200 gene knockout sets to simulate. The number of gene knockouts of each set were picked randomly from a uniform distribution in the range between 25 and the total size of the partition.

For technical details of the implementation of the guess stage see the `GeneticAlgorithmFocusSet` class in chapter 4. The simulations for this generation cannot be found in `ko.db` (see section 4.5 for more details on the databases) because the results failed to upload due to a technical error, for more information see section A.2.

Add

To take advantage of item 2 of the principles of static essentiality, the add stage randomly picks between two and four of the partitions from the guess stage and then randomly picks one viable gene knockout combination from each of the selected partitions, both random variables are from a uniform distribution. Each of the selected sets of gene knockouts are then combined to make one larger set of genes to knockout, which is then simulated. This child creation process is repeated 2,000 times.

For technical details of the implementation of the add stage see the `MixFousSets` class in chapter 4. The simulations for this generation can found under the experiment name 'mix_ne_focus_splits' in `ko.db` (see section 4.5 for more details on the databases).

Mate

A genetic algorithm is normally seeded with simple random guesses which result in children with very large genomes. Instead, GAMA is seeded by the 50 smallest viable genomes found by the add stage and then proceeds to mate them as a standard genetic algorithm would. The mate stage was implemented through the `MixFousSets` class but was a separate experiment to the add stage and the experiment name is 'big_mix_of_split_mixes' in `ko.db` (see section 4.5 for more details on the databases).

5.1.2 Dynamic probability distribution

The 'dynamic probability distribution' is based on the idea of picking gene knockouts for children based on a probability distribution. This probability distribution is updated after each generation of children have been simulated and so that the algorithm improves it's guesses each time. Figure 5.6 is a flow diagram that illustrates the principles of the algorithm. There were two initial hurdles at the beginning of the problem. (1) In order to learn from all simulations it would be necessary to read `ko.db` (see section 4.5 for more details on the database) from the compute nodes of clusters and calculate the distribution in a reasonable amount of time. (2) Since genomic

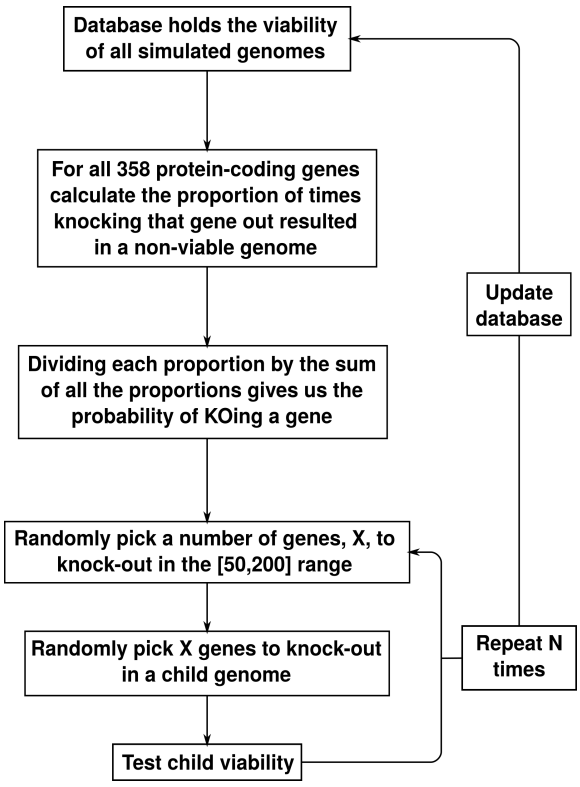


Figure 5.6: The dynamic probability distribution uses the entire database of simulations to assign probabilities of picking a gene to knockout based on the proportion of time it killed the cell in the past.

context effects the essentially of the genes, one needs to be able to calculate the conditional probability of a gene being non-essential given the existence of other genes in the genome. Calculating all possible conditional probabilities results in a combinatorial explosion that would make it unlikely that it is possible to compute/store/process/analyse efficiently enough to be a viable option. However, there might be ways to approximate the probabilities *on-the-fly* making it feasible. In order to test (2) easily, (1) needs to be implemented. (1) is also the quicker/easier problem to solve and so this was done first assuming that only single gene probabilities were needed (i.e. not calculating the conditional probabilities resulting in an averaging of the effects of genomic context). Due to time constraints only (1) was completed and (2) remains to be done. It was implemented to take into account every simulation that had been recorded in the database at the time. The probability of picking a gene to knockout was calculated by counting all of the genomes that successfully knocked the given gene out and dividing it by the total amount of genomes that had the given gene knocked-out - for more details on the technical implementation see section 4.4.5.

5.2 Genome reduction

The rest of this chapter now looks at the results of all the algorithms tested in the GDS. In an attempt to find the minimal genome of the *M. genitalium* whole-cell model [52] the GDS was used to perform massive *in-silico* experiments to reduce its genome.

Comparing the performance of different algorithms is not straight forward for a variety of reasons. Algorithms and instances of the same algorithm can have many different attributes like the length of time it was run for, the number of simulations it performed in that time, what computer cluster(s) were used and how busy the clusters were at the time. The smallest viable genome found could be misleading because some algorithms were only run for a few weeks and ran 1,000 simulations whilst others ran for six months and ran tens of thousands of simulations. The amount of genes knocked-out divided by the duration of time that the algorithm ran for would also be misleading for three reasons: (1) The amount of time simulations wait in the cluster queue depends on how busy it is. (2) Some queuing systems take into account how much a user and the user's group/school/department have used the cluster in the past so that people get fair usage. (3) The amount of time a simulation takes to run varies depending on the cluster. For example, BG takes between 5-15 hours to run the life cycle of a single *M. genitalium* cell (excluding queueing time) whereas BC3 takes between 5-35+ hours to do the same. A measure of effort might be more appropriate than time, like the number of gene reductions per simulation for example. Unfortunately, optimisation algorithms often converge logarithmically and so converged algorithms may show a much worse result per simulation than an algorithm that did not get a chance to converge. Standardising aspects of the test (like the number of simulations) would help, but the early stage of the research meant that flexibility was required and the amount of resources required to perform an experiment makes repeating experiments very hard.

5.2.1 Standard genetic algorithms

In Chapter 3 it was shown that a genetic algorithm showed promise at reducing genomes and so a genetic algorithm was implemented onto the GDS first. Chapter 2.3 discusses genetic algorithms more generally and chapter 4 discusses its implementation into the GDS.

Computer cluster queueing systems normally have an array submission option of some kind. These arrays enable users to submit multiple, similar jobs to the cluster as one array. These arrays are normally limited in size but the limit can vary widely across clusters, and it was decided to initially limit array sizes in the GDS to 200 jobs. This limit was mainly to enable easy compatibility across clusters and also there, anecdotally, appeared to be a noticeable increase in queuing time when 200 job arrays were regularly submitted, and so it was feared that regularly submitting larger arrays may significantly slow our queue progression due to overuse. Whilst

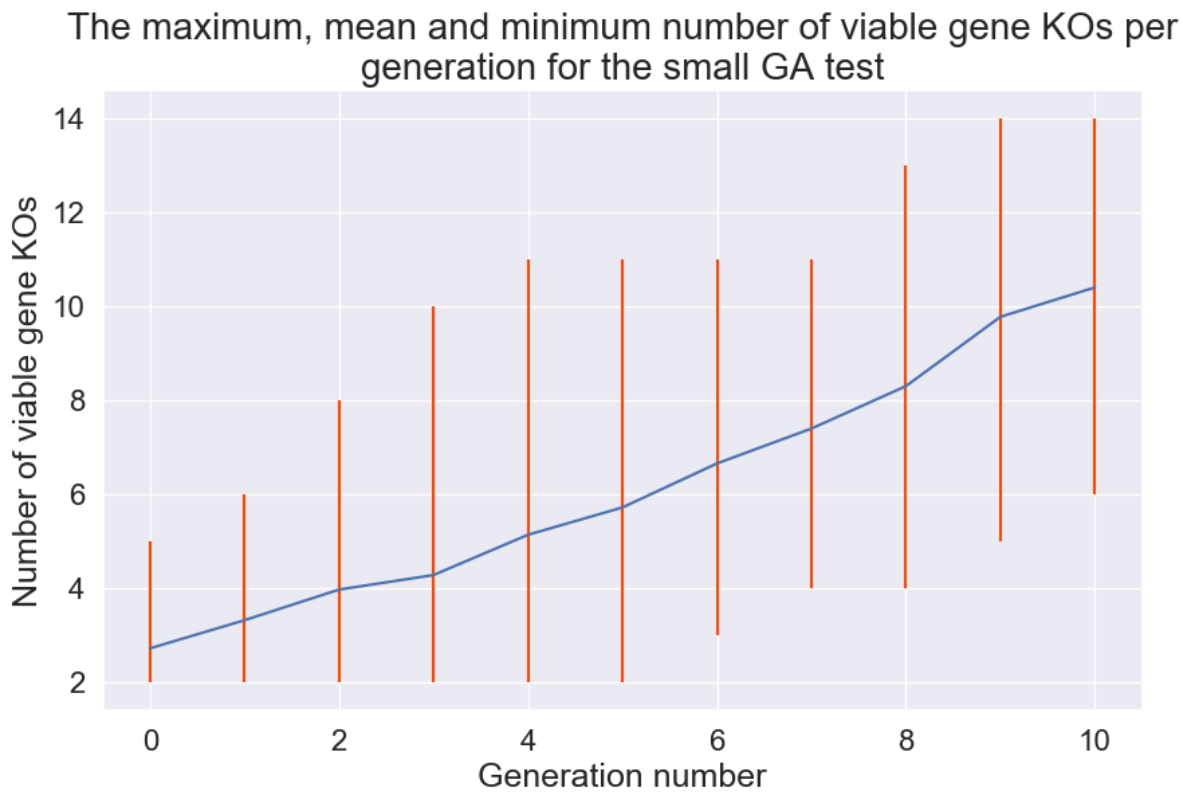


Figure 5.7: A plot of the number of genes reduced against the generation number. The blue line represents the mean gene reduction for that generation and the high/low error-bar points represent the maximum/minimum gene reduction in that generation.

the GDS could facilitate different array sizes depending on the clusters involved this option was not implemented since it was not deemed a priority improvement. Initially, if the GDS was given more than 200 jobs per array, then it would group them so that the amount of jobs is not more than 200 and that each job runs multiple simulations in parallel with one core per simulation. For example, if a batch of jobs needed 400 simulations, then 200 jobs would be created, but each job would request 2 cores and run 2 simulations in parallel. The downside to this is that requesting many cores per array job can significantly impact the speed at which one progresses through the cluster job queue. For this reason, it was expected that 200 simulations per generation would be the soft maximum/standard for experiments. How this is coded can be seen in the `createStandardKoSubmissionScript` method of the respective cluster class.

The first two genetic algorithm tests were run with 100 and 200 children per generation which were named the 'small GA test' or 'large GA test', respectively.

5.2.1.1 Small GA test

The 'small GA test' is a standard genetic algorithm implemented in the GDS. It was run on BC3, the maximum number of fit individuals that could survive to the next generation was 100, every generation had 100 children, and each child was simulated only once. The experiment name is GA_run_2017_11_30 and can be found in the databases described in section 4.5. The experiment was set to end after generation 100 but was stopped prematurely at generation 12 in order to free space on the cluster for other experiments.

This algorithm reduced the genome by 14 protein-coding genes in 11 generations (the total number of generations was 12) and 1,000 simulations (total simulations was 1,100). On average it reduced the genome by 0.014 genes per simulation. Generation 0 started on November 30$^{\text{th}}$ 2017, generation 10 started on December 20$^{\text{th}}$ 2017 and so ran for roughly 22 days, reducing the genome by roughly 0.64 genes per day. Linearly extrapolating[⑤] this result, shows that at least 79 generations would be needed to knockout 100 genes. Figure 5.7 shows the per generation genome reduction progress and one can see a fairly steady progression to knocking-out more genes over time. However, progress is slow. Linearly extrapolating for time[⑥] shows that the 79 generations would take 158 days or ~ 5.3 months.

5.2.1.2 Large GA test

The 'large GA test' is a standard genetic algorithm implemented in the GDS. It was run on BC3, the maximum number of fit individuals that could survive to the next generation was 100, generation 0 had 600 children whilst the other generations had 200 children, and each child was simulated only once. The experiment name is GA_full_run_2017_12_01 and can be found in the databases described in chapter 4.5. The experiment was set to end after generation 100 but was stopped prematurely at generation 25 in order to free space on the cluster for other experiments. In addition to this, the algorithm was paused after generation 17, and two changes were made to the mating process.

The first change was made to how the parent genomes were mixed to create a child. Originally, two parent genomes were randomly picked and then a random point on the genome was picked, and both the parent's genomes were split at that point. The child was then created by combining the top part of genome 1 with the bottom part of genome 2. This was changed to enable more diverse mixing of genomes and the number of genes, x, from parent 1 were picked randomly from a uniform distribution and the number of genes picked from parent 2 were $T - x$, where T is the total number of genes in a wild-type genome (i.e. $T = 358$). Then x and $T - x$ genes are randomly sampled from a uniform distribution from the parent 1 and parent 2 genomes, respectively.

⑤ $\frac{100}{14} \times 11 = 78.5$ (1 d.p)
⑥ $\frac{79}{11} \times 22 = 158$

Figure 5.8: A plot of the number of genes reduced against the generation number. The blue line represents the mean gene reduction for that generation and the high/low error-bar points represent the maximum/minimum gene reduction in that generation.

Both implementations can be seen in the `mateTheFittest` method of the `GeneticAlgorithm` class of the `multigeneration_algorithm` module, where the old version is in the comments above the new version.

The second change was made to how mutations were performed on a child. Originally, 10% of the time a child would have one gene mutated, which would be chosen randomly from a uniform distribution. The new implementation again only mutated 10% of children but this time the number of genes to mutate, x, was chosen from a modified exponential distribution so that mutating 2 genes was extremely likely and the probability of larger numbers of genes reduced exponentially. Preliminary tests showed that the largest number of genes picked over 1,000 trails often got as high as 12. Once the number of genes to mutate were known the genes were mutated by randomly sampling x genes from a uniform distribution.

This algorithm reduced the genome by 34 protein-coding genes in 25 generations (the total number of generations was 26) and 5,397 simulations (total simulations was 5,597). On average it reduced the genome by ~ 0.006 genes per simulation. Generation 0 started on December 1st

2017 and generation 25 started on January 18th and so ran for roughly 50 days, reducing the genome by roughly 0.68 genes per day. Linearly extrapolating[⑦] this result, shows that at least 74 generations would be needed to knockout 100 genes. Figure 5.8 shows the per generation, genome reduction progress and one can see a fairly steady progression to knocking-out more genes over time. However, again, progress is slow. Linearly extrapolating for time[⑧] shows that the 74 generations would take 148 days or ~ 4.9 months. It can also be seen that the distance between the minimum and maximum number of viable gene knockouts gets large from generation 18 onwards suggesting that the changes made may have slightly increased the number of viable gene knockouts found per generation. All other algorithms that have a mate stage will use the new implementation.

5.2.2 Dynamic probability distribution

The dynamic probability distribution queries `ko.db` to find all existing simulations that knock-out each gene and then use it to calculate the proportion of times is was involved in viable combinations and uses that as a probability of picking it again - for more information see section 5.1.2. It was run on BC3, the number of gene knockouts for a child was picked from a uniform distribution in the range 50-200, all generations had 200 children, and each child was simulated only once. The experiment name is dpd_18_01_11 and can be found in the databases described in section 4.5. The experiment was set to end after generation 100 but was stopped prematurely at generation 10 in order to free space on the cluster for other experiments.

This algorithm reduced the genome by 0 protein-coding genes in 10 generations (the total number of generations was 10) and 2,031 simulations (total simulations was 2,031). On average it reduced the genome by 0 genes per simulation and 0 genes per day. Linear extrapolation projects that no amount of generations would result in reducing the genome by 100 genes[⑨].

It is expected that the conditional probabilities would help increase the success of this algorithm. Additionally, looking to knockout sets in between 50 and 200 is a huge solution space to be looking at and that perhaps a variable knockout set size would help if it started small and got bigger as the algorithm found viable gene knockout sets towards the upper range. The worry here would be that the algorithm would not converge significantly faster than the standard genetic algorithm. However, it may be worth pursuing this algorithm since it has much more versatility than other algorithms. For example, the genetic algorithm's success in reducing the genome suggests that there is some gradient in the fitness function that the algorithm can climb. This gradient is not guaranteed to be the case, for instance, should a solution space have a mostly

⑦ $\frac{100}{34} \times 25 = 73.5$ (1 d.p)
⑧ $\frac{74}{25} \times 50 = 148$
⑨ $0 \times g = 0 \; \forall g$ where $g$is the number of generations and zero is number of genes reduced per generation.

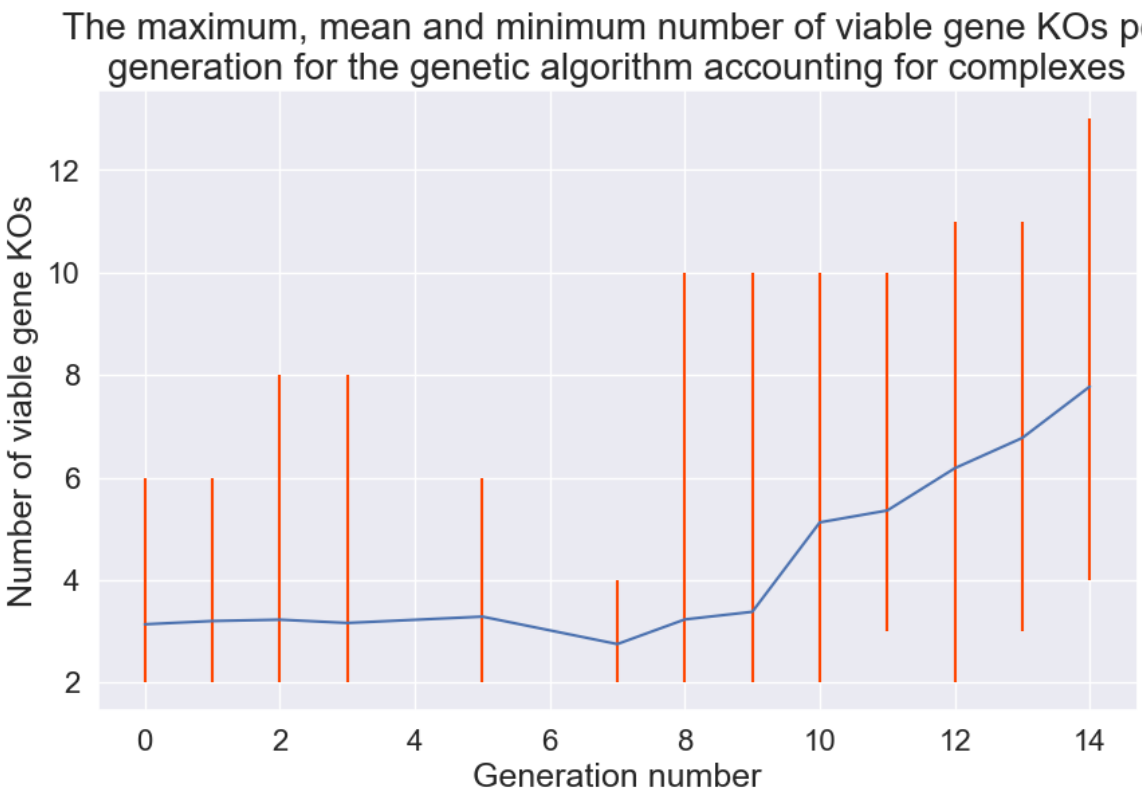


Figure 5.9: A plot of the number of genes reduced against the generation number. The blue line represents the mean gene reduction for that generation and the high/low error-bar points represent the maximum/minimum gene reduction in that generation.

flat gradient with sparse islands of locally optimal solutions, a genetic algorithm would perform very badly, but a dynamic probability distribution with conditional probabilities might take into account the islands and avoid flat gradients by building multiple distributions of each island. The conditional probabilities may also enable it to focus on multiple local minima simultaneously and furthermore analysis of the probability space may reveal more detail about the solution space.

5.2.3 Genetic algorithms with biological knowledge

An attempt to use biological knowledge to increase the convergence time of a standard genetic algorithm was used. Two main strategise were investigated, grouping genes by gene-product combinations as well using non-standard seeding strategies to start the *mate* stage with the smallest possible genomes.

5.2.3.1 Grouping genes by complexes

Efforts were made to reduce the size of the solution space by grouping genes by complexes that they form (see section 5.1.1.2 for the theoretical explanation). The ‘genetic algorithm with com-

plexes' is an implementation of this in the GDS. It was run on BG, the maximum number of fit individuals that could survive to the next generation was 100, all generations had 200 children, and each child was simulated only once. The experiment name is GA_with_complexes_bg_2017_02_05 and can be found in the databases described in chapter 4.5. The experiment was set to end after generation 100 but was stopped prematurely at generation 14 in order to free space on the cluster for other experiments.

This algorithm reduced the genome by 13 protein-coding genes in 15 generations (the total number of generations was 15) and 1,466 simulations (total simulations was 1,466). On average it reduced the genome by ~ 0.009 genes per simulation. Generation 0 started on February 5th 2018 and generation 14 started on March 3rd and so ran for roughly 26 days, reducing the genome by roughly 0.5 genes per day. Linearly extrapolating[⑩] this result, shows that at least 115 generations would be needed to knockout 100 genes. Figure 5.9 shows the per generation, genome reduction progress and one can see a fairly steady progression to knocking-out more genes over time. However, again, progress is slow. Linearly extrapolating for time[❶] shows that the 155 generations would take 169 days or ~ 5.6 months.

The missing minimum/maximum bars in figure 5.9 (e.g. generations 4 and 6) denotes missing data. I have not been able to deduce the cause of this but the most likely reason is cluster problems (e.g. the filesystem going down). This is likely to have caused a loss of not just new information but information existing at the time and so has detrimental effects to atleast generation 5 and 7. For this reason it is reasonable to believe that the results are underestimating the performance of the algorithm and should be re-done but unfortunately, there was not time to do this in this thesis.

5.2.3.2 Genetic algorithms with non-standard seeding

A genetic algorithm needs an initial population of individuals with which to perform natural selection on. This population is called the seed population and is generation 0 in the GDS implementation of a standard genetic algorithm. A genetic algorithm traditionally starts by randomly guessing the children for the seed generation and this is repeated until a minimum amount of individuals with a certain level of fitness has been found. All previous genetic algorithms discussed here had a seed population made by randomly guessing gene knockouts from a uniform distribution. The number of gene knockouts of an individual is randomly picked from a uniform distribution in the range 2-5. This limit was imposed because it is very unlikely to randomly guess a viable genome with more than 5 genes knocked-out in 200 guesses. Since the genetic

⑩ $\frac{100}{13} \times 15 = 115.4$ (1 d.p)
❶ $\frac{155}{15} \times 26 = 169$

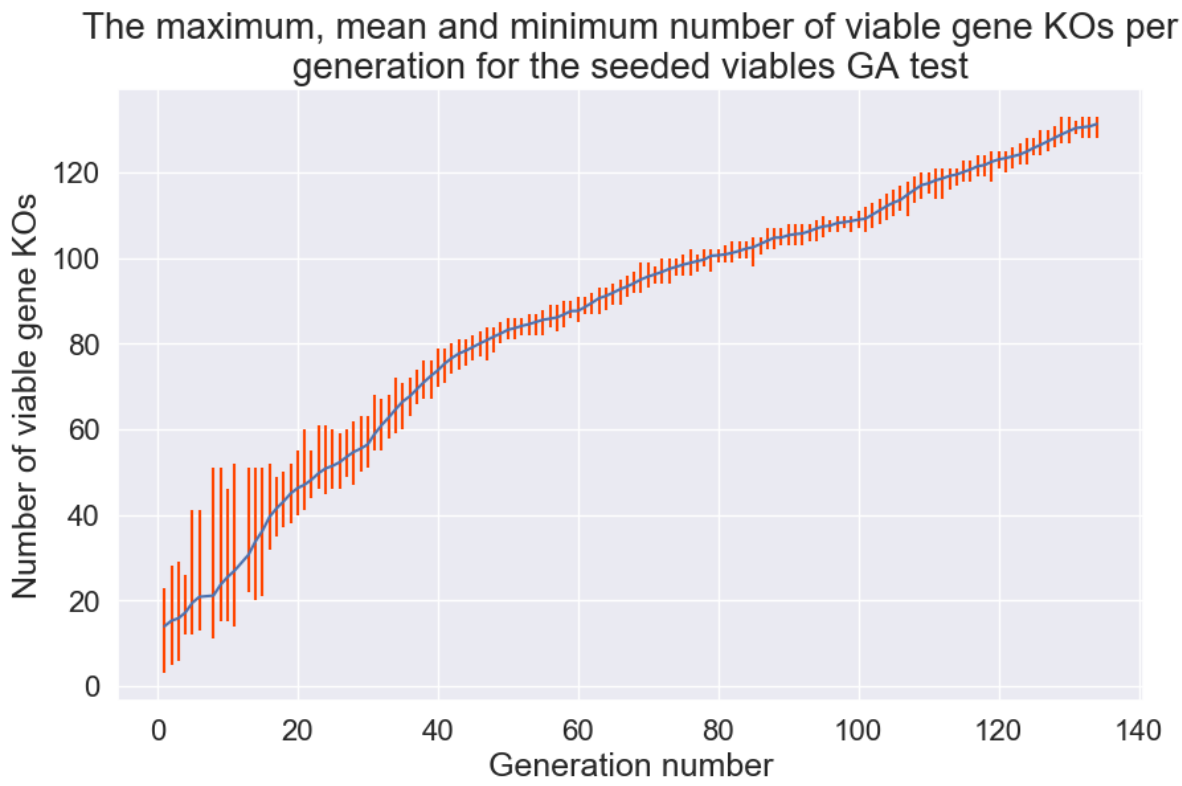


Figure 5.10: A plot of the number of genes reduced against the generation number. The blue line represents the mean gene reduction for that generation and the high/low error-bar points represent the maximum/minimum gene reduction in that generation.

algorithm reduces genomes slowly it was decided to investigate alternative methods to seed the genetic algorithm such that it could get to larger knockout combinations quicker.

5.2.3.3 Seed viable

The 'seed viable genetic algorithm' is an implementation of a genetic algorithm with non-standard seeding in the GDS. It aims to combine small non-viable minimal genome predictions with larger viable genomes to seed a standard genetic algorithm (see section 5.1.1.3 for a full discussion on this). It was run on both BG and BC3, and the maximum number of fit individuals that could survive to the next generation was 100. Generations 1-30 had 200 children each, generations 31-100 had 100 children each, generation 101-134 had 200 children each and each child was simulated only once. The experiment name is ga_seed_viables_bg_2017_01_02 and ga_seed_viables_cont_on_bc3_2017_01_02 and can be found in the databases described in section 4.5. The experiment was set to end after generation 100 and then changed to 200 but was stopped prematurely at generation 134 because the Hub had to be turned off to move office.

This algorithm reduced the genome by 133 protein-coding genes in 128 generations (the to-

tal number of generations was 135) and 15,087 simulations (total simulations was 15,675). On average it reduced the genome by ~ 0.009 genes per simulation. Generation 1 (generation 0 is skipped and the fittest individuals passed to generation 1 are the seeds discussed above) started on January 2nd 2018 and generation 134 started on May 10th and so ran for roughly 130 days or ~ 4.3 months, reducing the genome by roughly 1.02 genes per day. The algorithm first found a viable genome with 100 or more genes knocked-out in generation 72. Generation 72 started on March 6th 2018 and so took about 64 days or ~ 2.1 months. Figure 5.10 shows the per generation, genome reduction progress and one can see that the first 40 generations converge much faster than the previous algorithms discussed. There is a large range of the number of genes knocked-out in the early generations whilst the genetic algorithm explores all the combinations of the seeded genomes. The success of this also suggests that our assumption that the non-viable predictions did resemble a minimal genome was correct. As the generations progress it reverts back to a similar pace as a standard genetic algorithm and it also looks as if the range of number of knockouts increases a little at generation 100, which is consistent with the same observation for the large GA test, 5.2.1.2.

5.2.3.4 GAMA

The guess, add, and mate algorithm (GAMA) attempts to seed a gentic algorithm (the mate stage) with much smaller viable genomes by using the guess and add stages instead of a traditional random seed stage, for more information see section 5.1.1.4. In addition to the new seeding, it was suspected that given the size of the solution space, limiting the number of simulations in a generation to 200 children might be severely hampering progress. Due to this, larger generation sizes were planned, and instead of letting the cluster class spread multiple simulations over single jobs, a routine was implemented in the GAMA classes that simply split all the children into multiple sets of 200 single simulation job arrays and submit each job array in parallel to the cluster(s). This is implemented in the `runSimulations` method of the respective algorithms class in the `multigeneration_algorithm` module.

This algorithm reduced the genome by 165 protein-coding genes in 28 generations (the total number of generations was 47) and 32,928 simulations (total simulations was 51,119). On average it reduced the genome by ~ 0.005 genes per simulation. All 47 generations took less than 62 days, reducing the genome by 2.66 genes per day. The algorithm was meant to continue until 200 generations but was stopped manually because it went 20 generations without finding a smaller genome. All three stages took under 72 days or ~ 2.4 months.

Figure 5.11 shows that the guess and add stages rapidly find genomes with as many as 137 genes knocked-out. The guess stage finds viable genomes with as many as 40 genes removed which provides some validation for its use. In addition to this, the huge combinations of viable

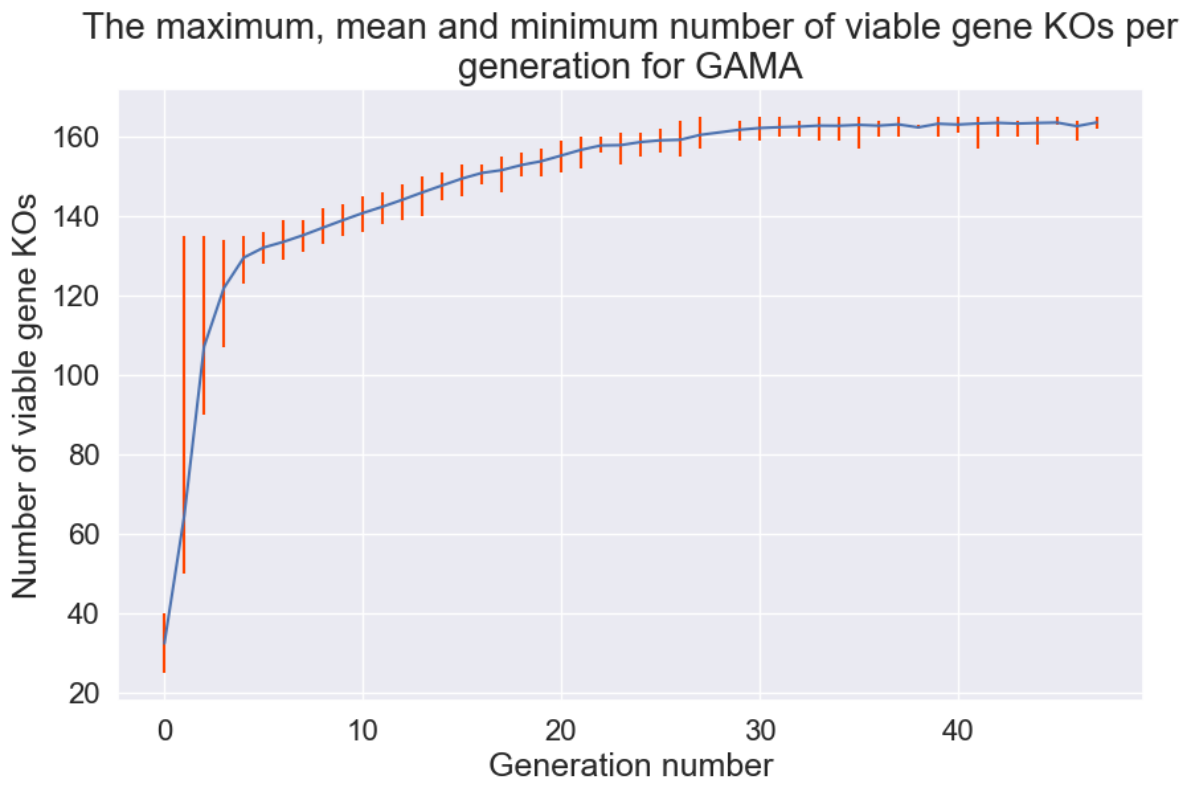


Figure 5.11: A plot of the number of genes reduced against the generation number. The blue line represents the mean gene reduction for that generation and the high/low error-bar points represent the maximum/minimum gene reduction in that generation.

gene knockouts found by the add stage validates this stage and supports our use of item 2 of the principles of static essentiality. It is worth noting that whilst the maximum amount of gene knockouts is high in the add stage, the mean is very low which may suggest that the large viable combinations were less common. The skew towards smaller viable knockout sets may illustrate that item 2 of the principles of static essentiality may not be very consistent. This inconsistency is not surprising since dynamic essentiality was expected but the algorithm just exploits the relative scarcity compared to static essentiality. The degree of success of the algorithm, however, may hint that dynamic essentiality is, perhaps surprisingly, uncommon in the *M. genitalium* whole-cell model though.

It is clear to see that the guess and add stages of the algorithm worked well. In order to quantify this we add the two following statistics. The first generation (guess) found viable genomes with up to 40 genes knocked-out in 1,600 simulations, removing 0.025 genes per simulation. The first two generations (guess and add) found viable genomes with up to 135 genes knocked-out in 3,600 simulations, removing 0.0375 genes per simulation. The guess and add stages took about ten days

to complete. Linearly interpolating these results[2] tell us that GAMA found a viable 100-gene knockout combination in two generations or about a week.

It may appear that GAMA is performing more simulations per unit time than the other algorithms but the first 18 generations of the seed viable algorithm performed 3,600 simulations which took about 9 days which is a day quicker than GAMA (both sets of simulations were performed on BG). This suggests that having the larger generation sizes does not speed up or slow the rate of simulations and that the algorithm itself is to credit for the fast convergence and not the larger size of the generations.

5.2.3.5 Comparison

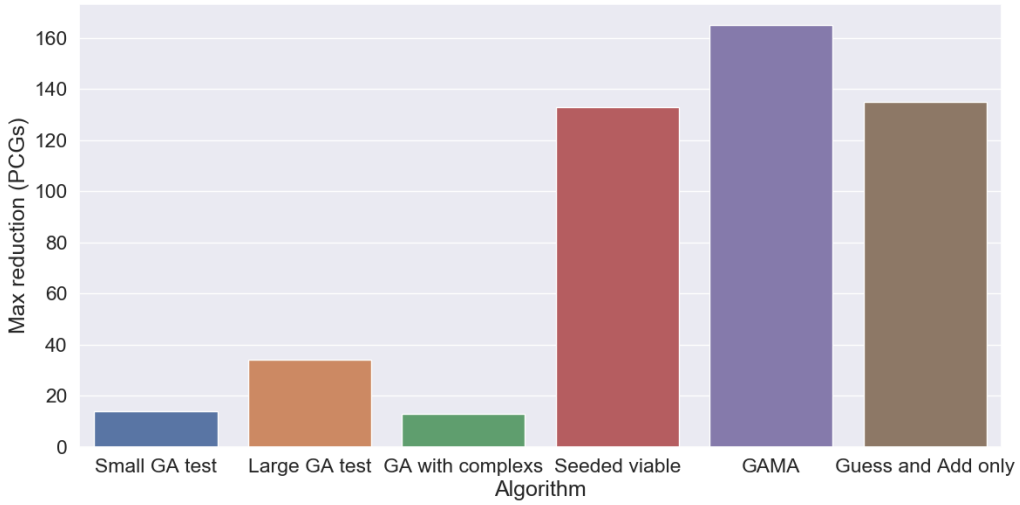
As discussed at the beginning of the chapter it was decided to prioritise flexibility to experiment with algorithms over maintaining a rigid structure to test on and so a direct quantitative comparison of the algorithms is not possible. However, comparing a few different statistics can give a rough idea of how the algorithms performed against each other. Figure 5.12 looks at overall measures of performance whereas Figure 5.13 looks at measures that try and account for some of the inconsistencies between experiments by comparing the performance finding exactly 100-gene knockouts. All algorithms discussed in this section are included plus one that looks at only the guess and add stages of the GAMA algorithm. The reason for this is two-fold. First, GAMA is the only algorithm to converge on a local minimal genome which means that it has a very long shallow gradient where it almost converged that the other algorithms haven't got to and so the guess and add stage is fairer comparison. Second, the mate stage is a standard genetic algorithm whereas the guess and add stages are the novel part of the algorithm and deserves closer analysis.

Figure 5.12(a) looks at the number of genes knocked-out in the smallest viable genome for each algorithm. GAMA is clearly the most successful here with the guess and add stages and the seeded viable algorithms showing impressive reductions. The standard genetic algorithms and the GA with complexes show much smaller totals but it is worth remembering that these ran a much smaller amount of simulations.

Figure 5.12(b) shows the average gene reduction per simulation for each algorithm. It can be seen that the GAMA algorithm performs the worst with this metric but the guess and add stages performs the best and knocks-out over 7 times more genes on average, illustrating just how much of a difference the logarithmic convergence makes. It seemed that increasing the number of children per generation helped the standard genetic algorithms find smaller genomes and so it was slightly surprising that the second largest was the small GA which significantly outperforms the rest. The GA with complexes performed very well with this metric, especially since it is

[2] $10 \times \frac{100}{135} = 7.41$ (2 d.p.).

(a)



(b)

Figure 5.12: A comparison of the genome reduction algorithms implemented on GDS. (a) Shows the largest viable combination of gene knockouts found by each algorithm. (b) Shows the average number of genes reduced per simulation for each algorithm.

suspected that technical problems hindered the algorithm's performance. The seeded viable algorithm performed the same as the GA with complexes. Whilst the seeded viable algorithm is not nearly as close to converging as GAMA, perhaps it also deserves to have a shorter subset of generations analysed. However, it is not as clear cut for the seed viable algorithm because the only part that is different in the algorithm is the seed generation but most of the increased performance happens in the generations after and so requires some kind of system to decide which generations should be included to make a fair comparison. How to create such a system is not currently clear and so remains an unanswered question in this thesis.

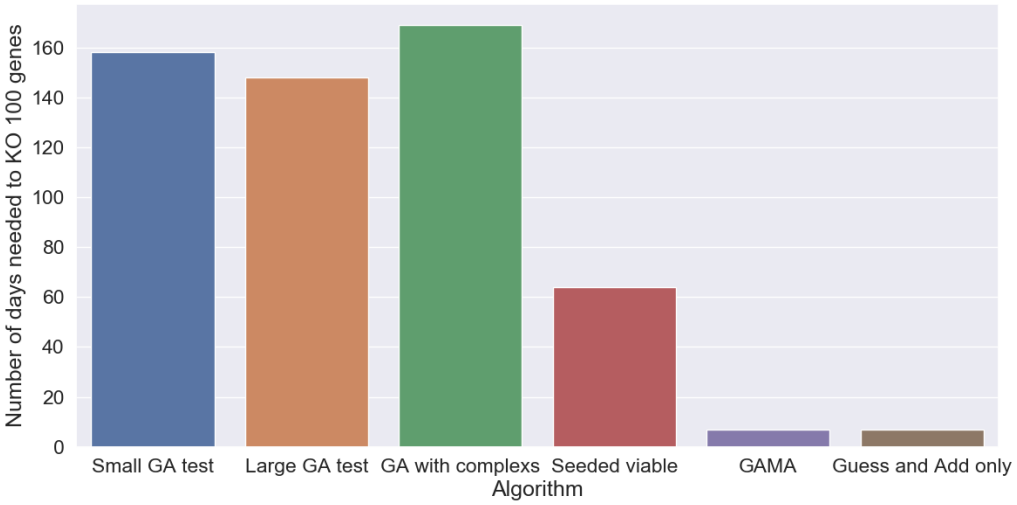
Figure 5.13(a) shows the number of days each algorithm took to find a viable genome with 100 genes knocked-out. GAMA and the guess and add stages are the same for this metric and the next (Figure 5.13(b)) since the algorithm had found 100 knockouts by the add stage and they are again, clearly outperforming the rest with only 7 days. The seed viable manages to do it in about 2 months, and the rest come in at around 5 months. It is worth remembering that the standard genetic algorithms and the GA with complexes did not actually find 100 knockouts and so their figures had to be linearly extrapolated. The linear extrapolation could mean that their figures are underestimated but given the expected logarithmic convergence there is a reason to doubt this. A more significant concession is that the all the GA tests were run on BC3, GAMA was run on BG, and the seed viable algorithm was run on a mix of both clusters. This is significant because BG runs a simulation quicker than BC3. The GA with complexes is by far the worst performer, and the small GA test surprisingly outperforms the seeded viable algorithm. These mixed results suggest that a closer analysis than performed in this thesis is neccessary to understand what is going on but one may speculate that the dominating factor is how many other people are using the cluster during an *in-silico* experiment.

Figure 5.13(b) shows the number of simulations needed to find a viable genome reduced by 100 genes. Again GAMA and the guess and add stages are the same and significantly outperform the rest.

5.3 Making non-viable genomes viable

Chapter 3 shows that minimal genome predictions in the literature do not produce a viable cell in the *M. genitalium* whole-cell model. Joshua Rees showed that adding the model's singularly essential genes and other genes with inconsistent phenotypes (see chapter 2.3) to all the predicted genomes fixed the cell in every case. This approach will be referred to as the Rees method, and it gives a nice clean result but it is not guaranteed to fix a non-viable genome nor that the viable genome is the smallest possible. In order to overcome these problems, it was decided to implement an algorithm on the GDS that would try and converge to the smallest viable genome from the

(a)



(b)

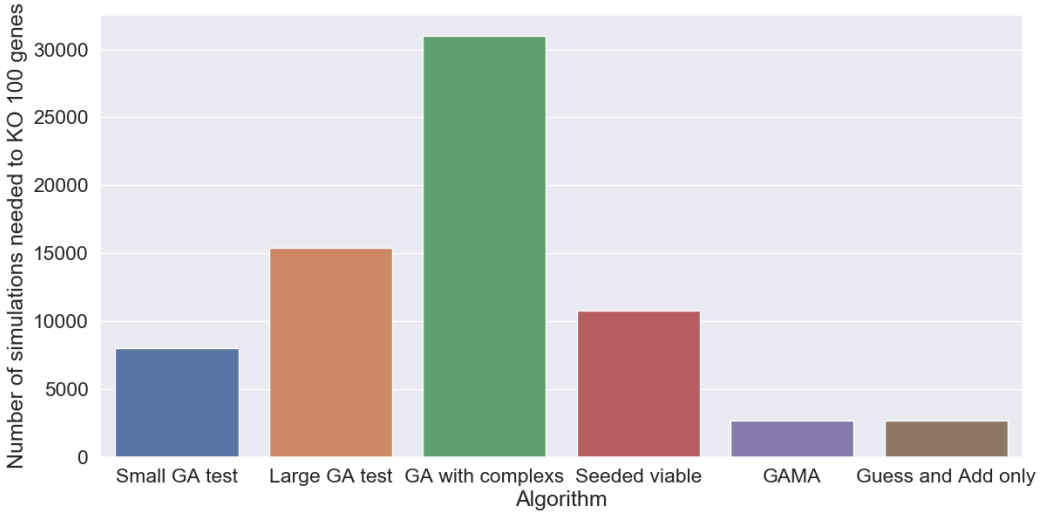


Figure 5.13: A comparison of the genome reduction algorithms implemented on GDS. (a) Shows the number of days each algorithm needed to reduce the genome by 100 genes. (b) Shows the number of simulations needed to reduce the genome by 100 genes.

initial non-viable genome. This approach will be referred to as the Chalkley method.

5.3.1 Genetic algorithm additions

It was decided to try and reverse the genetic algorithm so that it starts with a non-viable genome and attempts to knock genes back into the genome in order to try and find the smallest viable genome.

Preliminary tests showed that knocking all the essential genes back into the genomes did not fix any of the predictions. It was assumed that high essential genes would be rare and the essential genes combinatorially increase the solution space making it much harder to find viable genomes and so the gene addition experiments only searched through combinations of non-essential genes to add whilst all the essential genes are permanently present in the genome, until the mate stage.

5.3.1.1 Atlas

The genetic algorithm added genes to the atlas genome and found eight different viable genomes. The smallest genome had 59 genes knocked-out and the largest had 14 genes knocked-out. 3,158 genomes were simulated of which 2,908 were unique. The algorithm had 65 generations. In the database this has the experiment name ‘atlas_no_ess_ki_bg_2018_01_19’.

5.3.1.2 Church

The genetic algorithm added genes to the church genome and found one viable genome. The genome had 46 genes knocked-out. 496 genomes were simulated of which 489 were unique. The algorithm had 9 generations. In the database this has the experiment name ‘church_ki_no_ess_bg_2018_01_19’.

5.3.1.3 Gil04

The genetic algorithm added genes to the gil04 genome and found four different viable genomes. The smallest genome had 77 genes knocked-out and the largest had 19 genes knocked-out. 3,153 genomes were simulated of which 3,013 were unique. The algorithm had 65 generations. In the database this has the experiment name ‘gil04_no_ess_ki_bg_2018_01_19’.

5.3.1.4 Gil14

The genetic algorithm added genes to the gil14 genome and found two different viable genomes. The smallest genome had 49 genes knocked-out and the largest had 33 genes knocked-out. 458 genomes were simulated of which 445 were unique. The algorithm had 8 generations. In the database this has the experiment name ‘gil14_no_ess_ki_bg_2018_01_19’.

5.3.1.5 Glass

The genetic algorithm added genes to the glass genome and found 641 different viable genomes. The smallest genome had 41 genes knocked-out and the largest had 19 genes knocked-out. 3,204 genomes were simulated of which 1,662 were unique. The algorithm had 70 generations. In the database this has the experiment name 'glass_ki_no_ess_bg_2017_01_20'. This is a continuation of 'glass_ki_no_ess_bg_2017_01_17', which had 245 simulations.

5.3.1.6 Karr

The genetic algorithm added genes to the karr genome and found 83 different viable genomes. The smallest genome had 61 genes knocked-out and the largest had 3 genes knocked-out. 250 genomes were simulated of which 247 were unique. The algorithm had 2 generations. In the database this has the experiment name 'karr_ki_wo_ess_bg_2018_01_18'. This is a continuation of 'karr_ki_no_ess_bg_2018_01_21', which had 399 simulations.

5.3.1.7 Huang

The genetic algorithm added genes to the Huang genome and found 1 viable genome. The genome had 78 genes knocked-out. 497 genomes were simulated of which 496 were unique. The algorithm had 8 generations. In the database this has the experiment name 'huang_no_ess_ki_bg_2018_01_19'.

5.3.1.8 Koonin

The genetic algorithm added genes to the koonin genome and found 641 different viable genomes. The smallest genome had 63 genes knocked-out and the largest had 41 genes knocked-out. 2,633 genomes were simulated of which 759 were unique. The algorithm had 51 generations. In the database this has the experiment name 'koonin_no_ess_ki_bg_2018_01_19'.

5.3.1.9 Tomita

The genetic algorithm added genes to the tomita genome and found 5 different viable genomes. The smallest genome had 4 genes knocked-out and the largest had 2 genes knocked-out. 711 genomes were simulated of which 677 were unique. The algorithm had 11 generations. In the database, this has the experiment name 'tomita_ki_bg_2017_01_03'.

5.3.1.10 Hutchinson

The genetic algorithm added genes to the hutchinson genome and found 499 viable genomes. 889 genomes were simulated of which 696 were unique. The algorithm had 9 generations. In the database this has the experiment name 'hutchinson_ki_no_ess_bg_2017_01_19'. The smallest

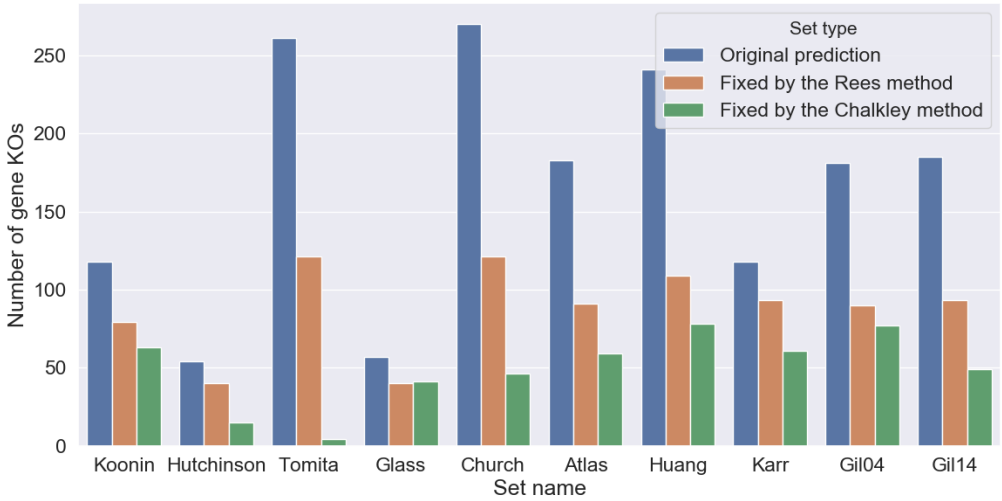


Figure 5.14: A comparison of number of gene knockouts for the non-viable minimal genome predictions and a fixed version created by the Rees method and by the Chalkley method.

genome had 15 genes knocked-out and the largest genome had 2 genes knocked-out. However, due to human error this algorithm knocked-out uncharacterised genes. Removing the uncharacterised genes reveals that the algorithm's smallest genome removed 15 genes and the largest genome only removed 1 gene.

5.3.1.11 Comparison of the Rees and Chalkley methods

Figure 5.14 compares the number of gene knockouts in the original non-viable genomes, the fixed genome using the Rees method, and the Chalkley method. One can see that most of the original predictions had significantly more genes knocked-out than either of the fixed sets. The Rees method is consistently better at fixing the non-viable genomes than the Chalkley method, and in fact, the Chalkley method only matches/beats the Rees method once (the Glass set but the Chalkley method only uses one less gene than the Rees method). Although most of the experiments were not run for very long (with some only simulating hundreds of children), the Chalkley method seemed to struggle the most for sets that had a big difference between the predicted set and the fixed set. The big difference results in a combinatorial explosion of genomes to try before success which suggests that, like the genetic algorithm for gene knockouts, the vast size of the solution space is again the problem. Whilst the Rees method outperformed the Chalkley method with this test, the Rees method is not guaranteed to fix the genome, and if it does not work, then there is no specification on what to do next, whereas the Chalkley method will always find a viable genome given enough time. Also, the Rees method is not guaranteed to find the smallest viable genome and could never find genomes with high essential genes knocked-out. When trying to fix a non-viable genome, one could start with the Rees method and then apply

the Chalkley method to the results of the Rees method. In order to improve how the Chalkley method deals with large solution spaces, one could use the principles from the GAMA algorithm (i.e. implement a guess and add stage instead of initial random guesses).

5.4 Discussion

This chapter presented the results of performing massive *in-silico* experiments enabled by the GDS in order to reduce the genome, and fix non-viable minimal genome predictions in the whole-cell model of *M. genitalium*. *In-silico* experiments were successfully run on BC3 and BG. Everything appeared to run fine on the C3DDB cluster except there was a *file-lock* restriction that prevented the GDS from writing Pickle files to the scratch drive and the system administrators were not able to correct the problem in a reasonable amount of time. Whilst *in-silico* experiments ran without a problem on a single cluster (even if there were multiple single-cluster experiments running simultaneously) attempts to run a single experiment that uses multiple clusters resulted in delays accessing ko.db due to simultaneous access attempts locking the SQLite3 database. It is currently unknown if the problem would resolve itself, given enough time, but SQLite3 is a light-weight RDMS that is not meant to get high volumes of access requests and so using a fully featured database would solve this problem - currently a PostgreSQL database is being constructed to overcome this problem.

All genome reduction algorithms worked to some extent but the biggest problem was to efficiently traverse the huge solution space in a reasonable amount of time. The genetic algorithm, dynamic probability distribution, genetic algorithm with complexes, and seed viable algorithms were all converging too slowly to wait for them to converge to a minimal genome. GAMA, which exploits the principles of static essentiality was able to converge to a minimal genome in 1.5-2.5 months.

Only one algorithm to fix non-viable genomes was implemented on the GDS and whilst it did perform its function the performance was deemed inadequate and was outperformed by a method created by Rees - although there there were complimentary strengths and weaknesses suggesting a combination of the two would be the best. Like the genome reduction algorithms the size of the solution space was the biggest problem and is likely to be the case for most design objectives that use combinations of genes to control the model.

GAMA was the only algorithm that was able to successfully navigate the huge solution space but even that algorithm took $\sim 1.5 - 2.5$ months on a HPC cluster. Additionally it relies on exploiting specific properties of the solution space (i.e. the principles of static essentiality) but we do not know if the model is accurate in this domain. Even if the model is accurate in this domain we do

not know if other organisms will have similar properties to their solution spaces. Furthermore, the principles of static essentiality may not help speed up the convergence of other objectives (e.g. metabolic engineering). As a result, methods to rapidly traverse the huge solution space to converge on general genome design objectives may become a significant goal of synthetic biology in the future.

CHAPTER 6

THE REDUCTOME

In chapter 5 we described massive *in-silico* experiments to reduce the genome of the *M. genitalium* whole-cell model and to fix non-viable genomes - this chapter will look deeper into the results. This thesis refers to a specific reduced genome as a reduction, but as of yet, there is no term to refer to the solution space of all reductions. Following the -omics naming convention, the set of all genes of an organism are its genome, all transcripts are its transcriptome etc, and now I define the set of all reduced genomes of an organism as its reductome. In other words, the reductome is the set of all possible combinations of genes in its genome, and I further define the viable and non-viable reductome as the combinations that produce a viable and non-viable cell, respectively.

A significant proportion of this chapter is based on a manuscript. Joshua Rees and I are co-first authors and it is titled 'Designing Minimal Genomes Using Whole-Cell models' which is currently being submitted to Nature Communications and can be viewed on bioRxiv[101]. A significant proportion of this chapter (particularly section 6.1) is included in the manuscript. A copy, supporting materials, and contributions can be found in the supplementary information of this thesis see appendix A.4. There is frequent references to the essentiality of genes and so the reader should be familiar with the terminology set out in section 1.1.2.

6.1 GAMA vs Minesweeper

All references to gene ontology in this thesis are done by my classifications except for this section which is done by Rees unless otherwise stated - for more information on the two different classifications see section 3.1.4 for justifications and appendix A.5 for data.

In parallel to the *in-silico* experiments described in chapter 5, Joshua Rees also developed a genome reduction algorithm. Joshua used his algorithm, *minesweeper*, to find a large reduction and it ran 2,000 simulations over a couple of days on BG. This algorithm looked for reductions by knocking-out genes only from the set of non-essential genes and the smallest genome found was called *minesweeper_256*. The genome of minesweeper_256 contains 256 protein-coding genes which makes it a 145 protein-coding gene reduction.

GAMA ran 51,000 simulations over roughly 2.5 months on BG, although the smallest genome was found in just over 1.5 months. The smallest genome found by GAMA was GAMA_236 and so is made up of 236 protein-coding genes which makes it a 165 protein-coding gene reduction. Interestingly, GAMA_236 knocked-out 18 singularly essential genes identifying them as potential high-essential genes.

More information about GAMA can be found in Chapters 4 and 5 as well as the manuscript which also contains information about minesweeper[101].

Having both found minimal genomes using different algorithms an investigation into what they were biologically and how they compare to each other was performed. The following sections use Rees' UniProt gene classifications which can be found in spreadsheets in the supplementary information - see section A.4.

Minesweeper_256 GO Term Analysis

We investigated what processes were removed in the creation of Minesweeper_256, using gene ontology (GO) biological process terms (see appendix A.5). The baseline M.genitalium whole-cell model has 259 genes of 401 genes (72% coverage) with GO terms on UniProt[102]. Minesweeper_256 has 186 (73%) genes with GO terms and 70 (27%) genes without. The 140 gene deletions did not impact 91 (59%) GO categories, impacted 22 (14%) GO categories, and removed 41 (27%) GO categories entirely, of which 29 (70%) were associated with a single gene (see appendix A.5).

The GO categories reduced include: DNA (replication, topological change, transcription regulation and initiation); protein (folding and transport); RNA processing; creation of lipids; cell cycle; and cell division. As the in-silico cells continue to function, we can assume that these categories could withstand low-level disruption.

Removed GO categories that group together multiple genes include: proton transport; host interaction; DNA recombination and repair; protein secretion and targeting to membrane; and response to oxidative stress. Removed GO categories that contain single genes include: transport

(proton, carbohydrate, phosphate and protein import, protein insertion into membrane); protein modification (refolding, repair, targeting); chromosome (segregation, separation); biosynthesis (coenzyme A, dTMP, dTTP, lipoprotein); breakdown (deoxyribonucleotide, deoxyribose, mRNA, protein); regulation (phosphate, carbohydrate, and carboxylic acid metabolic processes, cellular phosphate ion homeostasis); cell-cell adhesion; foreign DNA cleavage; SOS response; sister chromatid cohesion; and uracil salvage.

These deletions reduce the ability of *M. genitalium* to interact with the environment and defend against external forces. Internally this results in a reduction in control, from transport to regulation to genome management, and pruned metabolic processes and metabolites. The deletions leave Minesweeper_256's in-silico cell alive, but more vulnerable to external and internal pressures, less capable of responding to change, and more reliant on internal processes occurring by chance.

GAMA_236 GO Term Analysis

We investigated what processes were removed in the creation of GAMA_236, and compared to Minesweeper_256. GAMA_236 has 163 genes (69% coverage) with GO terms on UniProt[102], with 73 genes with no GO terms. The 165 genes deleted did not affect 83 (54%) GO categories, reduced counts in 17 (11%) GO categories, and removed 55 (35%) GO categories, 38 (69%) of which were associated with a single-gene (see appendix A.5). There were 8 unaffected and five reduced GO categories in Minesweeper_256 removed in GAMA_236, with one unaffected GO category unique to GAMA_236 (phosphate ion transmembrane transport). Four GO categories were reduced further in GAMA_236: DNA (transcription, transcription regulation, transport), and glycerol metabolic process.

The 14 additional GO categories removed include: DNA (transcription (termination, regulation of elongation, antitermination, initiation)); RNA (processing (mRNA, tRNA, rRNA), rRNA catabolic process, tRNA modification, pseudouridine synthesis); thiamine (biosynthetic process, diphosphate biosynthetic process); and protein lipoylation.

GO analysis of GAMA_236, when compared to Minesweeper_256, suggests a further reduction of both internal control and reactivity to the external environment.

Behaviour and Consistency of GAMA_236 and Minesweeper_256 Genomes

We investigated the characteristics of our two minimal genomes in terms of how consistently they produced a dividing in-silico cell and the range of possible behaviour they displayed. We simulated 100 replicates of an unmodified *M. genitalium* in-silico genome, Minesweeper_256, GAMA_236, and a single-gene knockout of a known essential gene (MG_006) for comparison. The

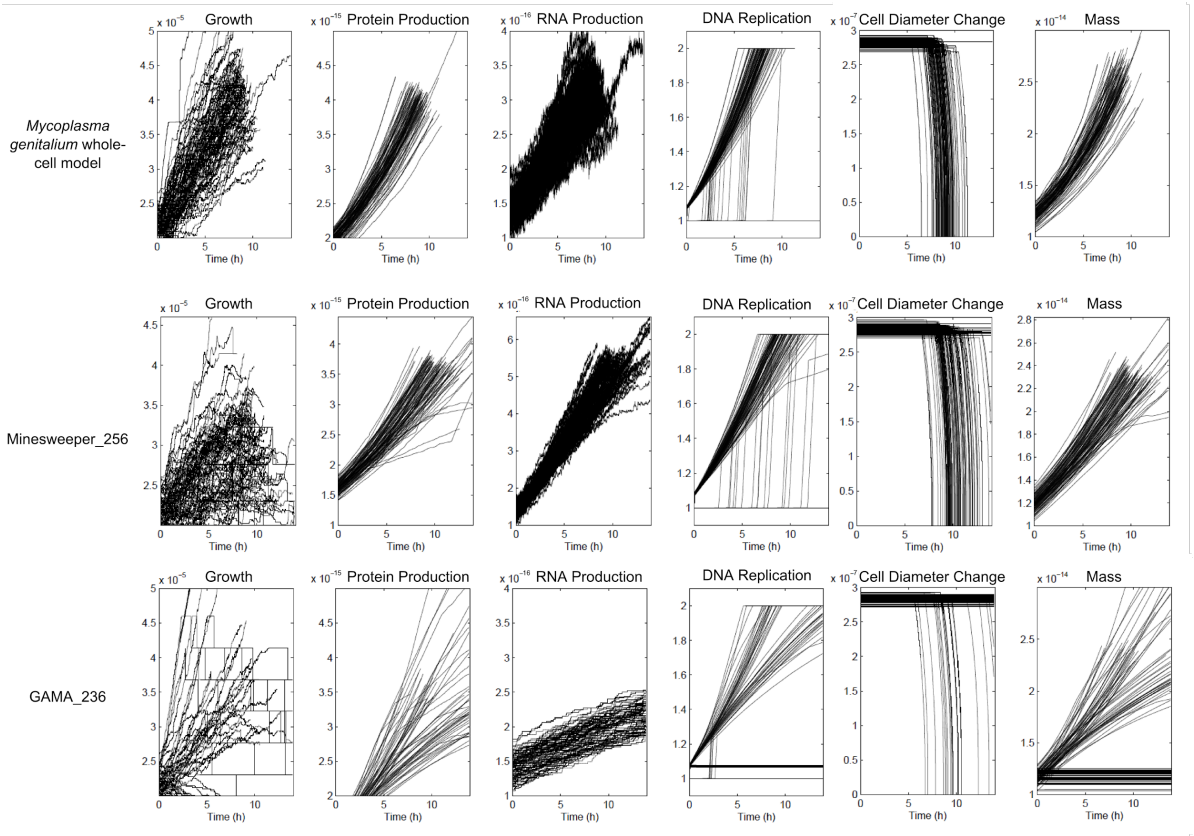


Figure 6.1: Comparison of unmodified *M. genitalium* whole-cell model, Minesweeper_256, and GAMA_236 outputs. 100 *in-silico* replicates, with time courses plotted for 6 cellular variables over 13.89 hours (the default endtime of the simulations). Top row is unmodified genome, showing the expected cellular behaviour (previously shown by Karr et al [52]) and is used for comparison. Minesweeper_256 and GAMA_236 show deviations in phenotype caused by gene deletions. Non aggregated data for each *in-silico* simulation is available (see appendix A.4).

rate of division (or not in MG_006 knockout simulations) was analysed to assign a phenotype penetrance percentage, quantifying how often an expected phenotype occurred. The unmodified *M. genitalium* and MG_006 knockout in-silico genomes demonstrated consistent phenotypes (99% and 0% divided, respectively). Minesweeper_256 was slightly less consistent (89% divided), while GAMA_236 was substantially less consistent, producing a dividing in-silico cell 18% of the time. This inconsistency is not entirely surprising given the greater number of gene deletions affecting essential gene functions (according to the GO term analysis).

In order to visualise the phenotypic penetrance Rees plotted the 100 replicates for the unmodified *M. genitalium* genome, Minesweeper_256, and GAMA_236 to assess the range of behaviour (Figure 6.1). The unmodified *M. genitalium* whole-cell model (Figure 6.1, top row) shows the range of expected behaviour for a dividing cell (in line with previous results [52]). Growth, protein production, and cellular mass increase over time, with most cells dividing at around 10 hours,

Gene	Annotation	GO Term (Biological Processes)	Non Essential In	Essential In
MG_039	N/A	N/A	GAMA_236	Minesweeper_256
MG_289	p37	transport	GAMA_236	Minesweeper_256
MG_290	p29	N/A	GAMA_236	Minesweeper_256
MG_291	p69	transport	GAMA_236	Minesweeper_256
MG_427	N/A	OsmC-like protein	GAMA_236	Minesweeper_256
MG_033	glpF	glycerol metabolic process	Minesweeper_256	GAMA_236
MG_410	pstB	N/A	Minesweeper_256	GAMA_236
MG_411	pstA	phosphate ion transmembrane transport process	Minesweeper_256	GAMA_236
MG_412	N/A	N/A	Minesweeper_256	GAMA_236
MG_305	dnaK	protein folding	M.g* whole-cell model	GAMA_236 and Minesweeper_256

Table 6.1: Low essential genes from Minesweeper_256 and GAMA_236 genomic contexts. Protein annotation and GO term obtained from KEGG [104] and UniProt [102], based on Fraser et al's Mycoplasma genitalium G37 genome [103].

though division can occur between 6 and 11 hours. RNA production fluctuates but increases over time. DNA replication follows a characteristic shape, with some simulations delaying the initiation of DNA replication past 9 hours.

By comparison, Minesweeper_256 (Figure 6.1, middle row) displays slower, and in some cases decreasing, growth over time which is capped to a lower maximum. Protein production and cellular mass are generated more slowly and present some erratic behaviour. The range of RNA production is narrower, compared to the unmodified *M. genitalium* whole-cell model. DNA replication takes longer and initiation can occur later (at 11 hours). Cell division occurs later, between 8 and 13.889 hours. A number of simulations can be seen failing to replicate DNA and divide.

Compared to the other genomes, GAMA_236 (Figure 6.1, bottom row) shows a much greater range of growth rates. Some grow as fast as the unmodified genome, some are comparable to Minesweeper_256, and some show very low or decreasing growth. Observable protein levels appear between 2 and 5 hours, followed by a slower rate of protein production in some simulations. Cellular mass is either similar to Minesweeper_256 or slower. The range of RNA production is reduced and the rate of RNA production is slower. Some simulations replicate DNA at a rate comparable to the unmodified genome, others replicate more slowly, some not completing DNA replication. Cell division occurs across a greater (6 - 13.889 hours). A number of simulations showing metabolic defects can be seen. These do not produce any growth, and can also be seen failing to replicate DNA and divide.

Genes with Low and High Essentiality

We analysed Minesweeper_256 and GAMA_236 to determine whether these were different minimal genomes, or GAMA_236 was an extension of Minesweeper_256. We conducted a gene content comparison of an unmodified *M. genitalium*, Minesweeper_256, and GAMA_236 genomes (see figure 6.2, created by Sophie Landon - see section 1.4 for information about the GDG), highlighting gene deletions unique to each minimal genome. We took this a step further and

Figure 6.2: Comparing the genomes of the *M. genitalium* whole-cell model, Minesweeper_256, and GAMA_236. The outer ring displays the *M. genitalium* genome (525 genes in total), with modelled genes (401) in navy and unmodelled genes (124, with unknown function) in grey. The middle ring displays the reduced Minesweeper_256 (256 genes) genome in light blue, with genes present in Minesweeper_265 but not in GAMA_236 in dark blue. The inner ring displays the reduced GAMA_236 (236 genes) genome in light yellow, with genes present in GAMA_236 but not in Minesweeper_265 in dark yellow. Figure produced from published *M. genitalium* genetic data [52] [103], with genetic data for Minesweeper_256 and GAMA_236 available in section A.4.

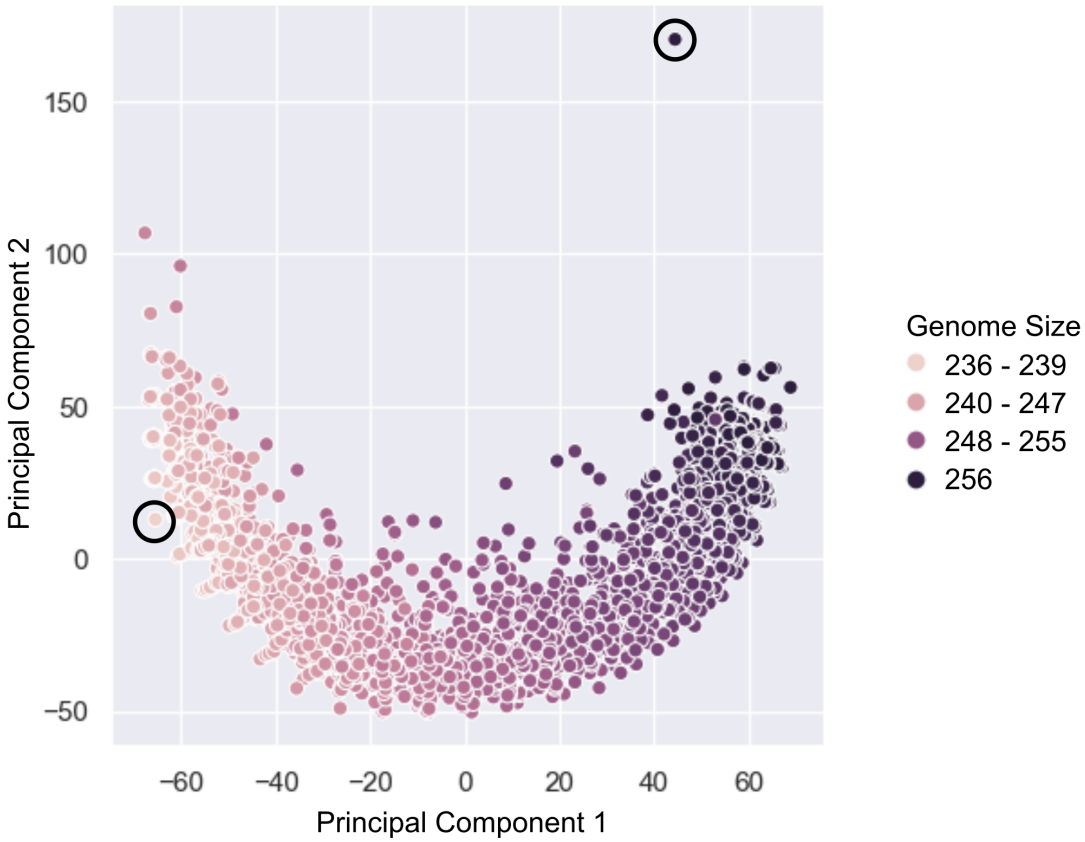


Figure 6.3: Comparing the genomes of Minesweeper_256 and 2954 GAMA genomes. The genome of Minesweeper_256 and all the genomes found by GAMA (that were the same size or smaller) were collated. Each point represents a single genome and is plotted based on a ARI distance (see section 4.6.2). The circled genome in the top right is Minesweeper_256 and the circled genome in the bottom left is GAMA_236.

compared Minesweeper_256 to all of the GAMA genomes 256 to 236 genes in size. Figure 6.3 shows the GAMA algorithm's avenue of gene reductions converging to a minimal genome, but Minesweeper_256 is clearly not on the same path of convergence.

Our comparison of the genomes found 18 genes knocked out in GAMA_236 that have high essentiality. They were defined as essential by single knockout in an unmodified *M. genitalium* whole-cell model, but could be removed in the genomic context of GAMA_236 without preventing division. Rees found that four of these 18 genes could be removed as a group in the genomic context of Minesweeper_256, but doing so greatly increased the number of non-dividing cells produced (see appendix A.4 to find data for the paper).

Rees also found that Minesweeper_256 and GAMA_236 each removed four unique genes (Table 6.1) which could not be removed (either individually or as a group) from the opposing mini-

mal genome (Minesweeper_256 or GAMA_236), without causing cellular death or mutations that prevented cellular division. We confirmed that these eight genes were individually non-essential. One additional gene, MG_305, could not be additionally removed in both GAMA_236 and Minesweeper_256. Our results demonstrate that these nine genes have low essentiality (see section section 1.1.2). To identify the cause of this synthetic lethality, we attempted to match the functions of these low essentiality genes (Table 6.1), as we anticipated finding redundant essential gene pairs or groups. We found two genes in GAMA_236 (MG_289, MG_291) had matching GO terms with the gene MG_411 in Minesweeper_256. These, and three other adjacent genes on the genome were tested by combinatorial gene knockouts in an unmodified *M. genitalium* whole-cell model genome (see appendix A.4 to find data for the paper). MG_289, MG_290, MG_291 were found to form a functional group, as were MG_410, MG_411, MG_412. Rees ran further simulations to show that these genes could be deleted individually and in functional groups from an otherwise unmodified *M. genitalium* whole-cell genome, and produce a dividing *in-silico* cell. However, any double gene deletion combination that involved one gene from each functional group resulted in a cell that could not produce RNA, produce protein, replicate DNA, grow or divide.

M. genitalium only has two external sources of phosphate, inorganic phosphate and phosphonate. MG_410, MG_411, and MG_412 transport inorganic phosphate into the cell, with MG_289, MG_290, and MG_291 transporting phosphonate into the cell [103], [104]. These phosphate sources proved to be a key difference between our minimal genomes. Minesweeper_256 removed the phosphate transport genes, relying on phosphonate as the sole phosphate source. GAMA_236 removed the phosphonate transport genes, relying on inorganic phosphate as the sole phosphate source. This can be seen in the GO term analysis, the phosphate ion transmembrane transport is still present in GAMA_236 but not in Minesweeper_256.

It has previously been theorised that individual bacterial species will have multiple minimal genomes [105], [106], with different gene content depending on the environment and which evolutionary redundant cellular pathways were selected during reduction. We would argue that one of these selected pathways is phosphate source, with minimal genomes differing by choice of phosphate transport genes and associated processing stages, equivalent to the phn gene cluster in *Escherichia coli* [107]. We could not however find any annotated phosphonate processing genes that had been subsequently removed in GAMA_236. We suspect that further 'pivot points', the selection of one redundant cellular pathway over another during reduction, will be identified in future in-vivo and *in-silico* bacterial reductions, increasing the base number of minimal genomes per bacterial species.

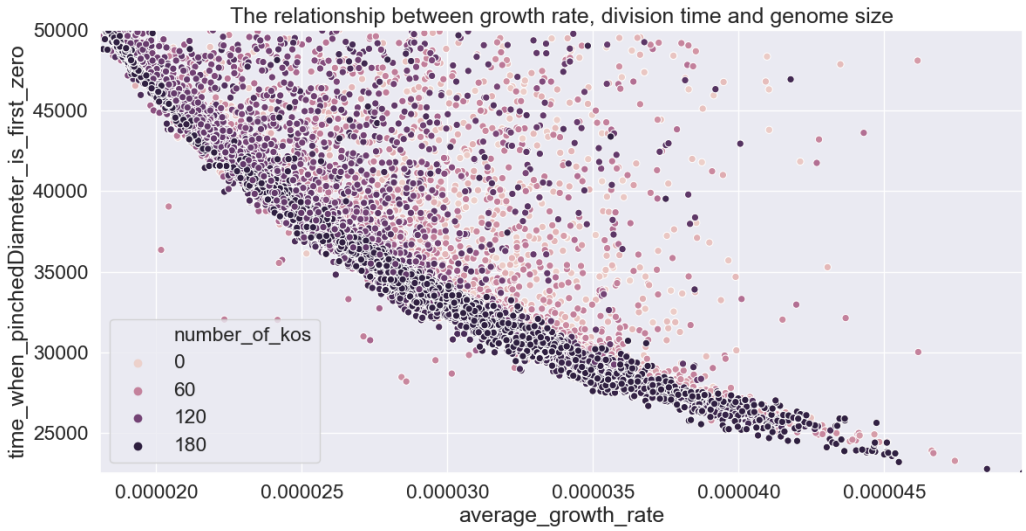


Figure 6.4: Scatter diagram of all viable genomes simulated by the GDS. Each point represents a simulation with the x-axis showing the average growth rate of the simulation, the y-axis showing the second that the simulation divided and the colour showing the number of genes knocked-out in the genome.

6.2 Analysis of all *in-silico* experiments in the genome design suite database

Section 6.1 showed that GAMA and Minesweeper had complementary strengths and weaknesses. Minesweeper provides a quick approximation of a minimal genome (~ 2 days). It is very inefficient with disk usage as it leaves the raw data untouched which is manageable because so few simulations are run. Whilst the small number of simulations result in very fast running time it also means that very little is learnt about the reductome as a whole. GAMA, on the other hand, takes much longer to run (~ 2 months) but is much closer to the actual minimal genome, more efficient with disk storage (although data is deleted) and provides a much clearer view of the reductome. These differences make comparison difficult and so much of the information gained from the GDS is not included in the paper and is in-fact restricted to only looking at the results from GAMA that are directly compariable to Minesweeper. This section is about doing a more thorough analysis of all the data produced by the GDS.

6.2.1 Genome comparison

Key statistics of all simulations run using the GDS are recorded in databases described in section 4.5. At the time of writing the GDS performed 107,946 simulations (including repetitions of the same genome) of which 40,173 produced a viable genome. Biomass production of a strain is an important factor that effects experimental cost/design. Figure 6.4 was created to get an overview

of the data and shows the relationship between average growth rate, the number of seconds it took to divide and the number of genes knocked-out of the genome. One can see that there is an edge made by the dark points (i.e. smallest genomes) that starts in the top left hand corner and ends in the bottom right. This edge is curved displaying a negative exponential relationship[①]. However, there are also genomes that appear to violate this relationship. This violation of the relationship suggests that there are multiple processes involved in the growth rate and the division time - some are involved in one and others are involved in both. The processes involved in both create the negative exponential relationship seen in the scatter plot whilst the processes involved in only one or neither can violate this relationship. Further analysis of these results could reveal the genes most important in the optimisation of growth rate, division time and biomass production overall. For example taking a few (say the top 1%) of each of the four extremes, growth-division: low-low, growth-division: low-high, growth-division: high-low, and growth-division: high-high and comparing the genomes between them should identify the genes responsible for each process individually and in combination. An example of this type of genome analysis can be seen ahead in figure 6.6 when hierarchical clustering and some other tools in the GDS were used to identify which genes defined two arms of convergence.

Multiple figures were created to try and get an idea of what the viable solution space of genome reductions looks like. All viable genomes with 100-gene knockouts or more were collected, a distance matrix was created using the ARI distance metric (see section 2.2.3 for general information on the ARI and distance matrices, and section 4.6.2 for information on the creation/implementation of the ARI distance metric and distance matrices in the GDS) and plotted using PCA (figure 6.5). There were 7,000 viable genomes found, and each one is represented by a point whose colour describes the size of the genome. The first thing to notice is that there are two very distinct arms of genomes. The largest genomes start in the top right-hand corner and get smaller as it goes down to the left until the genome gets to around 260 protein-coding genes when it curves back on itself until the smallest genomes, in pink, can be found on top of the red coloured genomes. The trend of genomes of a similar size being near one another is expected because the ARI distance metric looks at how many genes present or knocked-out they have in common and similar genomes have a similar number of knockouts. However, the smallest genomes being on top of the red and blue ones is unexpected. I believe that this is most likely the effect of loss of information due to the reduction dimensions through PCA. The two distinct arms of data show much more clearly that there are multiple paths of convergence and possibly multiple local minima than previous results (see section 6.1). One may argue that the strange placement of the smallest genomes may be caused by the need of another dimension to express another fork in the data and may also point out that the upper-arm of data could be made up of a large main arm

[①]The negative exponential relationship can be expressed as $y = AZ^{ax+b} + B$, where y is the time until division, x is the average growth rate and A, a, B, b, and Z are unknown constants that fit the equation to the edge described. Since it is a negative exponential we know that $a < 0$.

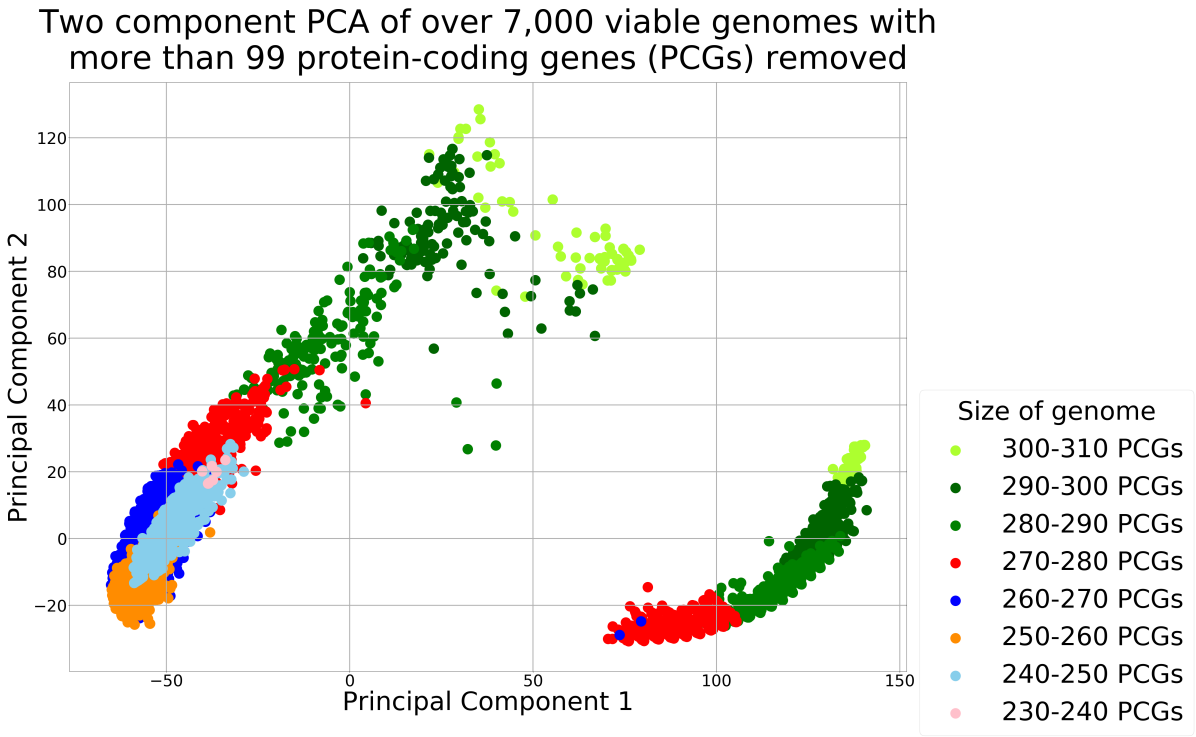


Figure 6.5: The ARI distance metric (see section 4.6.2) was used to create a distance matrix between all viable genomes with 100 or more genes knocked-out from the GDS and minesweeper_256. PCA was then used to reduce the number of dimensions to 2 and then each genome is plotted as a point with the colour indicating the size of the genome.

and a smaller second arm underneath. For now, it will be assumed that it is just the two obvious arms, but it is notable that that number could be as high as four.

In order to understand what the difference is between the two paths of convergence, the genomes were clustered using hierarchical clustering. Figure 6.6 shows that hierarchical clustering was able to perfectly group the two paths. Methods from the `analysis` module of the GDS were then used to look at which genes were always, sometimes, or never knocked out in each group. There were 40 genes that were never knocked-out in cluster 1 but were in cluster 2 and 26 genes that were always knocked out in cluster 1 but not in cluster 2. These differences in genes define the two arms. It could be possible that some of those differences are caused by random chance of the algorithm not trying those combinations yet. However, since the algorithm was converging down each arm and the arms look densely packed, we expect differences caused by chance to be rare. This would mean that the 40 genes never knocked-out in cluster 1 mostly can't be knocked-out without killing the cell and similarly, the 26 genes that were always knocked-out in cluster 1 needed to be knocked-out to avoid killing the cell. Both sets of genes contain singularly essential/non-essential genes suggesting both high and low essentiality is involved in defining the two paths of convergence.

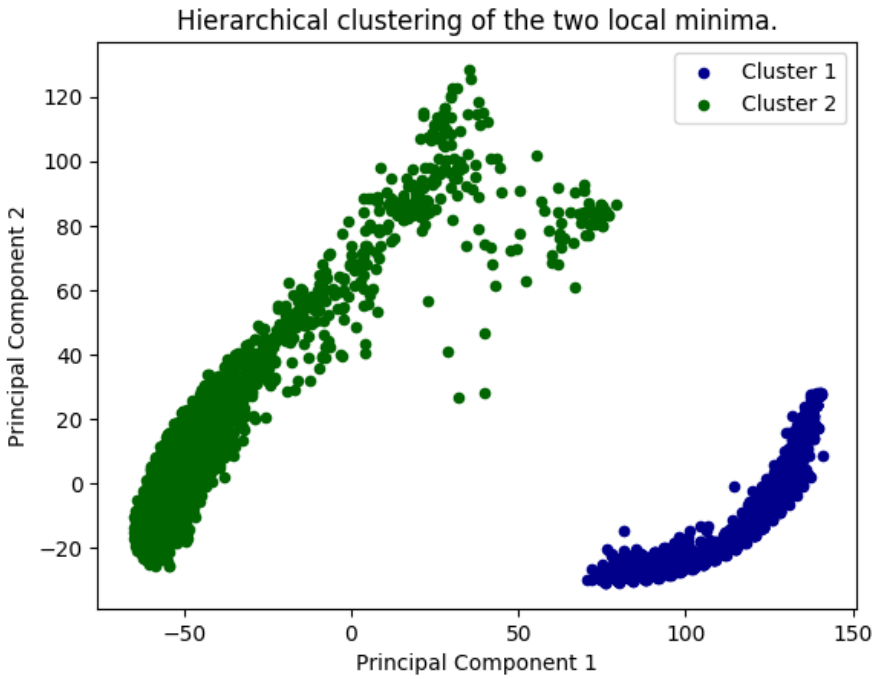


Figure 6.6: Visualisation of all viable genomes with over 99 genes knocked-out from figure 6.5 except now coloured by the groups found by hierarchical clustering.

Every single genome recorded in the GDS databases that produced a dividing cell was extracted, which produced over 40,000 genomes. The ARI distance was calculated between all genomes and plotted by reducing the dimension number of dimensions to two. Figure 6.7 shows each genome as a point, and the colour shows the size of the genome. The largest genomes can be found on the right of the diagram and get smaller as they move left. Again there appear to be two clear paths of convergence although they may be converging to the same minimal genome as the upper arm comes down as if it may have the potential to connect to the smallest genomes on the lower arm. At the top of the upper arm, there is a fork suggesting that there was an additional short or unsearched third path. Strangely there is a sparsely populated area that separates the small and large genomes of the lower arm. This could be the manifestation of complex patterns formed by genes changing essentiality but I believe that the add stage of the GAMA algorithm would have this effect since the guess stage would find between 20 and 40-gene knockouts sets that suddenly become sets with over 100 knockouts that acts much more like a jump than a solid path of convergence. This would be a sign of the effect utilised by item 2 of the principles of static essentiality (see section 5.2.3.4).

6.2.2 High and low essential genes

GAMA_256 is the only high-essential gene combination found so far and comparing the two minimal genomes enabled us to identify a low essential gene combination pair related to phos-

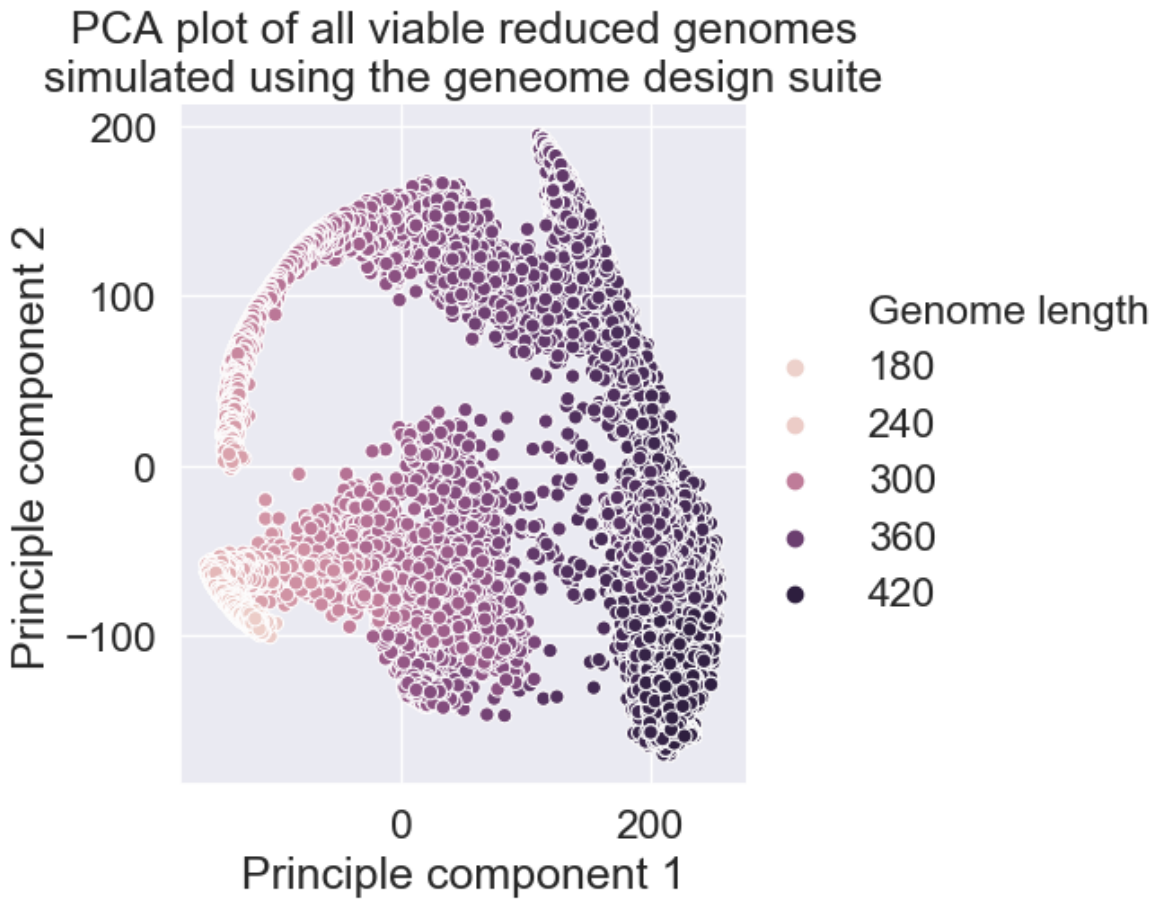


Figure 6.7: The ARI distance metric (see section 4.6.2) was used to create a distance matrix between all viable genomes found by the GDS. PCA was used to reduce the number of dimensions to 2 and then each genome is plotted as a point with the colour representing the size of the genome.

phate import/production (see section 6.1). In order to find more high and low essential genes the knockout database (see section 4.5) was queried. There are mtuliple different ways to detect potential shifts in essentiality, however, we will focus on one. This approach looks for only one shift in essentiality (i.e. sets on singularly non-essential genes becoming essential in combination or sets containing at least 1 singularly essential gene becoming non-essential in combination).

Low essential:

The strict approach of finding low-essential genes starts by extracting all non-viable gene combinations and groups by the number of gene knockouts. Any combination made up of only singularly non-essential genes are extracted. This results in non-viable combinations of singularly non-essential genes but consider the following problem using the mathematical representation specified in section 3.3. Let γ_1, γ_2 be singularly non-essential but essential in combination. Given an arbitrary singularly non-essential gene γ_i and $V(\gamma_1, \gamma_2, \gamma_i) = 0$ then it is likely that γ_i is not

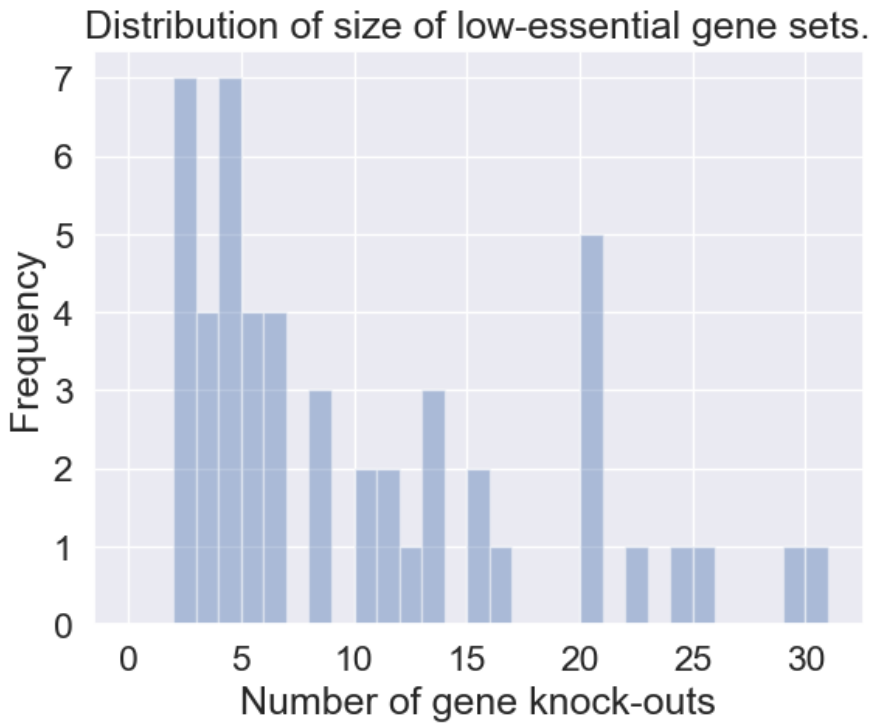


Figure 6.8: A histogram showing the distribution of gene knockout set sizes of low-essential genes.

contributing to a shift in essentiality and so should not be classed as low-essential. The exception to this is either $V(\gamma_1, \gamma_i) = 0$ or $V(\gamma_2, \gamma_i) = 0$ in which case $\{\gamma_1, \gamma_2\}$ and $\{\gamma_{1 \text{ or } 2}, \gamma_i\}$ are low essential sets but $\{\gamma_1, \gamma_2, \gamma_i\}$ is not. In order to remove the unwanted sets any low essential set that was a proper subset of another low essential set had the super-set removed.

This resulted in 50 low-essential genes being found, and the distribution of knockout size can be seen in figure 6.8. It can be seen that the minimum size set is 2 genes and the maximum is 31 genes, and although the amount of genes knocked-out is negatively correlated with the number of occurrences this is not a strict rule as demonstrated by the large number occurrences of 21-gene knockout sets. 92 different singularly non-essential genes were involved in at least one low essential combination. A low essential combination represents an essential biological function being disrupted, and so there should be some sort of relationship between the distribution of low essential gene combinations and the distribution of essential biological functions. However, it is feasible that one essential function could have multiple low essential gene combinations depending on the function and so is unlikely to be something as simple as being proportional to one another. Further research in this is suggested since it may enable the definition of the minimal cell in terms of essential functions. The scale of function may be a more appropriate scale to define a minimal cell on than genes since one function can be implemented by different

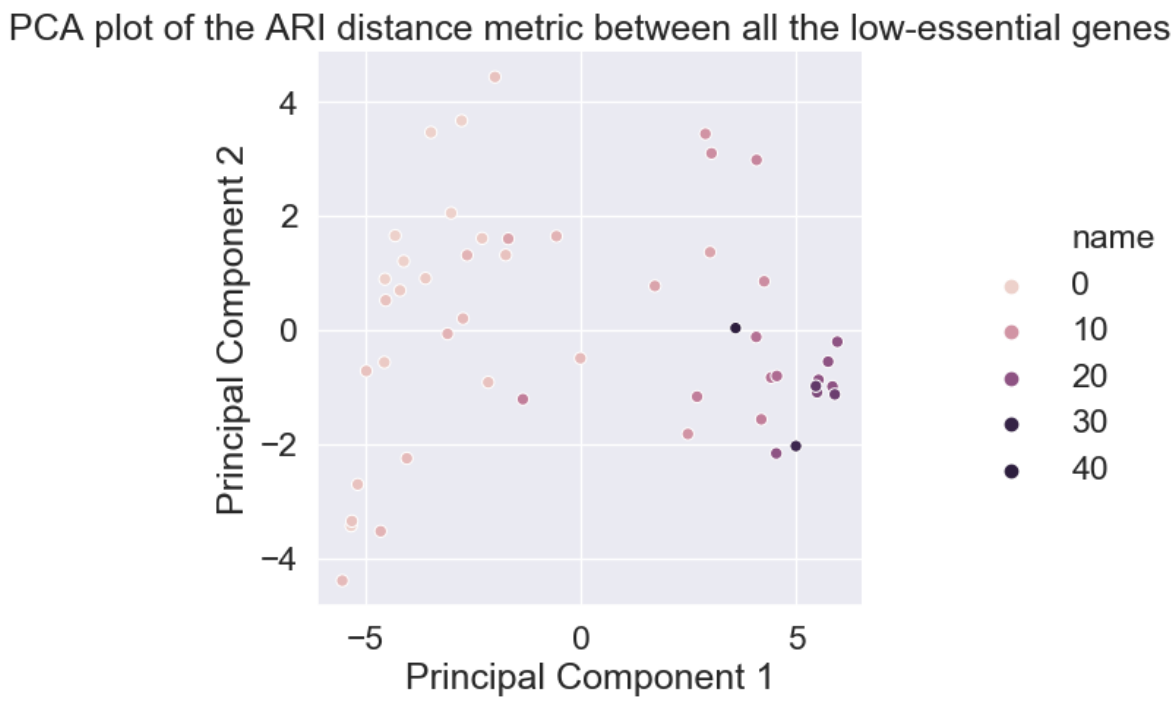


Figure 6.9: Scatter plot showing the ARI-distance of low-essential gene combinations using PCA. The number associated with the colour of each genome represents the number of genes knocked-out of the genome.

sets of genes - as demonstrated through non-homologous gene displacement which was discussed in chapter 1.

Figure 6.9 shows the ARI distance of all the low essential gene combinations coloured by the amount of genes involved in the low-essential set. The main pattern is that knockout sets of a similar size are more similar which suggests that the dominating pattern is based on the mathematics of the distance metric. There are possible clusters, but none of them are very distinct making it a very subjective decision.

High essential:

The strict approach for finding high-essential genes starts by extracting all viable gene combinations and grouping them by the number of gene knockouts. Any combination that contains at least one singularly essential gene is extracted as a high-essential combination. Then larger high-essential combinations are searched to find combinations that contain a smaller high-essential combination. If the larger combination does not contain singularly essential genes that do not belong to the smaller set, then the larger set is removed since, in terms of dynamic essentiality, the larger set is no different to the smaller set.

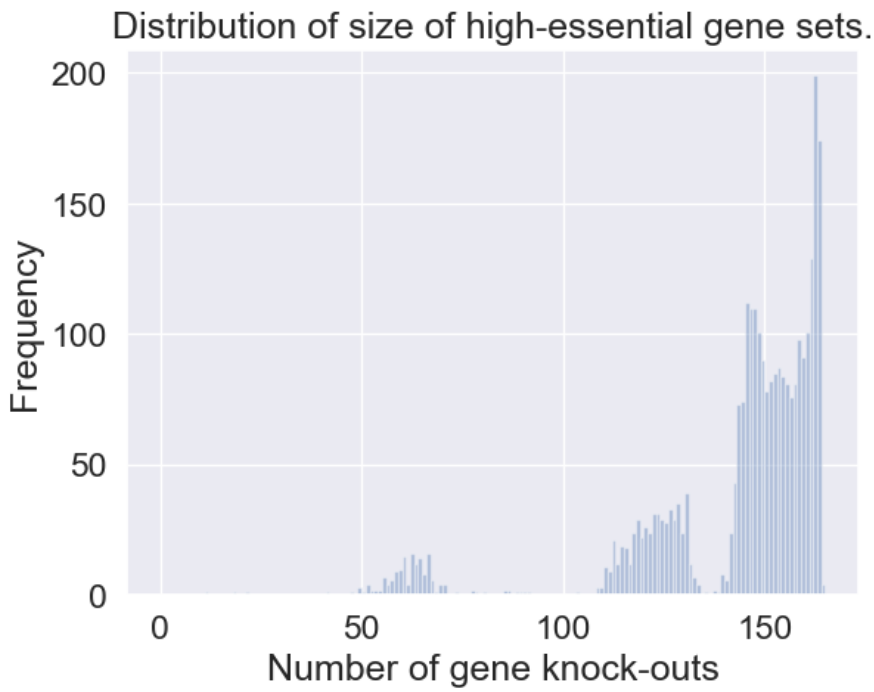


Figure 6.10: A histogram showing the distribution of gene knockout set sizes of high-essential genes.

This resulted in 2,909 high-essential gene combinations being found, and the distribution of knockout size can be seen in figure 6.10. The distribution of high essential genes is very different from the low essential genes. The high essential genes have many more instances of combinations than the low-essential combinations. The range is also much larger with a minimum gene knock-out set of 11 genes and a maximum of 165. Also in contrast to the low-essential genes, the number of high-essential genes are *positively* correlated to the size of the gene set. 266 genes are involved in high essential combinations. At around half of all the consistent genes, this is a surprisingly large number although it is worth remembering that high-essential gene sets can contain both essential and non-essential genes. A lot of this might be explained if it is common to get lots of similar combinations of high-essential gene sets - for example knockout any combination of a set of 50 singularly essential genes greater than 7. Also this number may be incorrectly high - later in section 6.1 we discuss the suitability of the whole-cell model of *M. genitalium* for essentiality classification and suggest that we are likely to be falsely identifying viable combinations due to only simulating one generation and also only repeating the experiment once.

Figure 6.11 shows the ARI distance of all the high-essential gene combinations coloured by the number of genes involved in the high-essential set. In contrast to the low-essential comparison, the high-essential sets show very distinctive clusters, and it is less dominated by genome size - although it is still relevant which is to be expected. If the hypothesis, in the previous

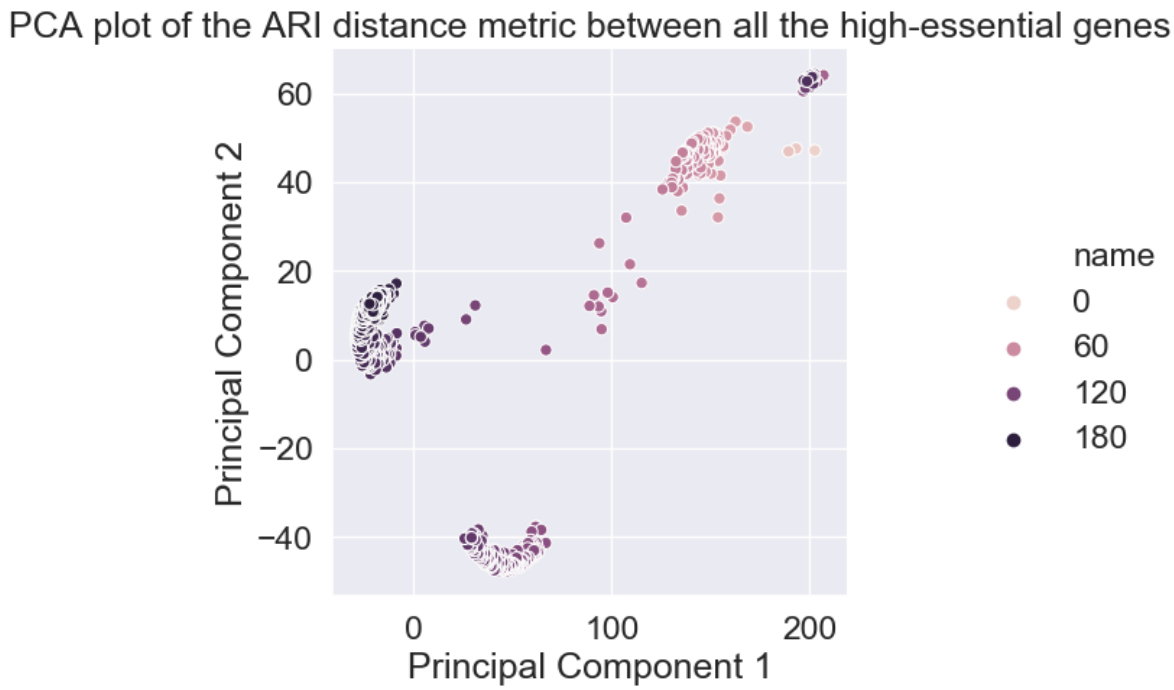


Figure 6.11: Scatter plot showing the ARI-distance of high-essential gene combinations using PCA. The colour of each point represents the number of genes knocked-out of the genome.

paragraph, that there are lots of combinations of similar high-essential gene sets is true, then one would expect that they would be shown as very tight clusters of high essential genes as seen in figure 6.11. The tight clustering may suggest that whilst there are thousands of high-essential combinations most of them are variations on the same pathway(s) which, in this case, would show that there are only 4-7 distinct high-essential sets. Since the high-essential genes enable the viable reduction of genomes, this may also provide information on the minimal gene set. For example, this figure may show that there are at least 4-7 local minima when reducing the genome of which 3 have over 100 genes knocked-out.

6.3 Design of experiment

All experiments done thus far have been *in-silico*, and so this section looks at how our findings may help guide experiments.

Whilst the *M. genitalium* whole-cell model has been tested against empirical data for single gene essentiality, its accuracy in the domain of multiple gene knockouts and changing genomic context is currently unknown. GAMA has been designed so that it can work on reducing the genome of any organism and furthermore the GDS is designed to easily be adapted to new algorithms, new models, and new computers. Whilst this enables one to quickly shift to the best available

models, it would still be desirable to manage the accuracies/inaccuracies of whatever model is being used. No model is a perfect representation of reality and so dealing with model inaccuracies when designing experiments is a useful skill.

The design of an experiment can drastically influence the impact of the results and so should be carefully considered. Here I suggest 3 main factors that influence experiment design.

1. Intended goal(s)
2. Resources
    - Time
    - Money
    - Manpower
3. Risk tolerance

Intended goals should be explicitly stated with priorities. Since there are multiple possible resource allocations, an analysis of the trade-off should be made in order to get a grasp of the options. A simple example could be that money will only allow work to be undertaken by two people for one year or one person for two years. Some kind of risk metric should be assigned to each goal and possibly an overall one too. This is very important to the design of experiment because someone might only desire a very big finding and not worry about a no-result as opposed to someone that wants as big a result as possible but needs to have some kind of result (e.g. perhaps a funding requirement).

Section 6.3.1 will look at the information gained so far and then we will look at how that can be used to design experiments.

6.3.1 Model accuracy profile

As previously stated, the *M. genitalium* whole-cell model has not been tested in the domain of predicting the essentiality of multiple gene knockouts. However, all knowledge should be utilised gauging the likely-hood of our reduced genome predictions.

The predictions of single gene knockouts have been tested against experiment. This means that one can group the genes by the accuracy of the single knockout predictions, i.e. true essential, true non-essential, false essential, false non-essential. It is reasonable to doubt the accuracy of multiple gene knockout predictions that contain genes that gave false results in the single gene knockout experiment. This results in one having strong doubts about the validity of viable genomes that have false non-essential genes knocked-out. Conversely, one might wonder if false

essential genes could be successfully removed from a viable genome. The latter would only be worth testing empirically since it is known that the model gives incorrect predictions with regards to that gene.

Another cause of uncertainty is inconsistent phenotypes. This inconsistency suggests that the model prediction is near the edge of a basin of attraction resulting in bifurcations from initial conditions or natural stochasticity. This means that knocking this gene out can result in qualitatively different results leaving the experimenter unsure if any, all or some of the results were correct. If a gene knockout is sensitive to initial conditions or stochasticity then one might expect it to pass this uncertainty to the effects of further gene disruptions, especially since one may expect multiple gene knockout experiments to reduce the stable solution space generally. For this reason, inconsistent genes are labelled high risk (even if the variable phenotypes are consistently essential or non-essential).

Since our purpose is to reduce genomes, testing low-essential combinations is not desired, however, these combinations contain a lot of information. A correct or incorrect prediction of a low essential gene set provides a lot of information about the capabilities of the model and if correct means the extremely high-resolution data created by the model may enable insight into biological mechanisms never seen/understood before.

High-essential genes represent the greatest gain since they are not only high-information experiments (like low-essential combinations), but they also reduce the size of the genome. Unfortunately, I believe that high-essential genes are high risk because of only simulating one generation and only repeating the experiment once combined with the fact that larger numbers of knockouts seem more likely to give inconsistent phenototypes.

6.3.2 Testing minimal genome predictions

The obvious way to test the predictions is to take the smallest prediction and test it experimentally. If the smallest genome does not work then the next smallest genome is tested. This can be repeated until a viable genome is found and this will be the smallest genome of the predictions. However, there are over 40,000 viable genome predictions and so could quickly become prohibitively expensive. In addition to this, as the algorithm converges on a minimal genome many variations of the same genome are produced which means that if the smallest genome does not work then many almost identical ones will follow and are also likely not to work as well. Given that the *M. genitalium* whole-cell model is the first, and currently only, whole-cell model and whole-cell models have not been tested on multiple gene knockouts one would expect there to be inaccuracies. On top of this there are several features of the model that make it sub-optimal for testing for essentiality (see chapter 2.3) plus there are genes that inaccurately

predicted the essentiality of single gene knockouts. What follows will be an attempt to manage model uncertainty when designing experiments.

JCVI successfully created JCVI-Syn3.0 by transplanting a significantly reduced *M. mycoides* genome into an empty *M. capricolum* cell [31]. The minimisation process was done by utilising a variation on a divide and conquer algorithm. We will refer to this as the JCVI divide and conquer algorithm and will be used to reduce risk should the model's accuracy be bad.

It seems that blindly trying all the largest viable gene knockouts is likely to be an inefficient approach, so more appropriate sets are considered. Take all the GAMA_236 gene knockouts and remove all high-risk genes. This leaves only the consistent singularly non-essential genes (89 genes) that the model did not predict low-essentiality. Knock all of these genes out of *M. genitalium* genome. If this does not produce a viable cell, then use the JCVI divide and conquer algorithm to find the largest viable subset of gene knockouts. This will be referred to as the viable, consistent non-essential genome. There is only one essential gene that correctly predicts single gene essentiality with a consistent phenotype involved in GAMA_236, so this should be knocked-out of the viable, consistent non-essential genome. In addition to this, there are 197 false-essential genes that may be worth exploring with a divide and conquer algorithm if there are enough resources.

There are 50 low-essential combinations and 2,909 high-essential combinations they may contain incorrectly modelled genes or inconsistent genes. Whilst this may not mean that the predictions are wrong, including them, is likely to increase the number of incorrect predictions and so depending on available resources one may only choose to explore the sets that do not contain the problem genes. There are 5 remaining low-essential gene sets and 0 remaining high-essential sets.

The genome reduction experiments described would be appropriate if the genomes were being created *de-novo* and transplanted into an empty cell. The only laboratory to successfully do this is in the JCVI, and so for most laboratories, knocking-out genes from the wild-type organism is more feasible but adds complexity because one needs to take into account the location of genes on the genome.

I created a way to overlay information about genes onto a genome in order to guide experiment design. This was the proof of concept that inspired Sophie Landon's development of the method used to create figure 6.2. Each equally sized segment in the graph represents a gene in the *M. genitalium* genome where 12 o'clock is the origin of replication and the colour of the segment represents information about the gene.

Figure 6.12 attempts to aid in the design of *in-vivo* gene knockout experiments. The green

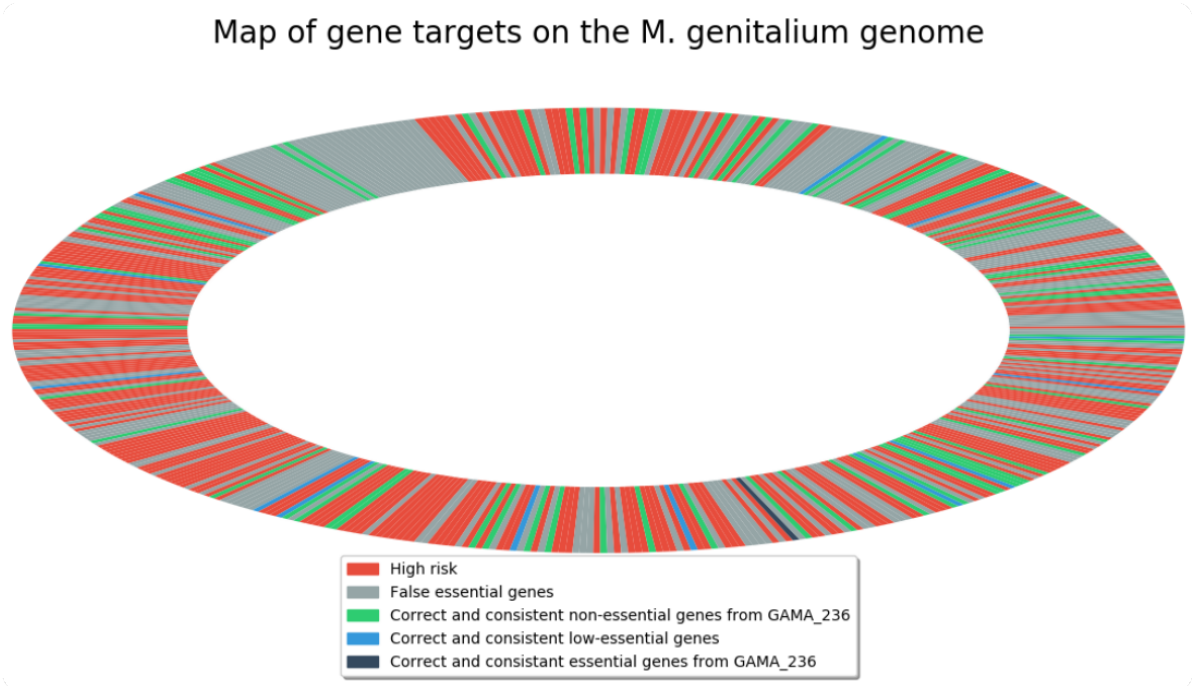


Figure 6.12: A depiction of the *M. genitalium* genome. Each equally sized segment represents a gene and 12 o'clock is the origin of replication. The colour of each segment represents information deemed useful for the design of experiments. Red genes are low information essential genes. Green and charcoal are sets of genes that have a high probability of being removed from the minimal genome. The blue genes are high probability low-essential genes. The grey genes are false-essential genes in the model and so may enable further reductions not predicted by the model.

segments represent genes that are most likely to be removed from the minimal genome. The genes are spread out across the genome meaning that an experiment would have to remove the genes with multiple iterations. There are a lot of small clusters however, meaning that this number will be smaller than the number of genes. All 77 green genes can be disrupted by removing 62 segments without disrupting any other gene colour. If false-essential genes were allowed to be removed, then this number could potentially be reduced even further to 57 segments whilst significantly increasing the overall gene reduction (there are multiple different implementations of this and so the actual figure would be down to the decisions of the experimenter). The low-essential genes are completely spread and so each gene would have to be disrupted individually. There is only 2 double, 1 triple, and 1 quadruple set of low essential genes and so is less of a problem. The false-essential genes not only account for a lot of the genes but there are large segments of them enabling one to test large knockout sets in a small amount of integrations which further adds to their allure as genome reduction targets.

One may notice that figures 6.12 and 6.2 are similar in concept but different in style and

content. The former figure was created using an analysis tool from the GDS and acts as a visual tool to guide the design of experiments for *in-vivo* genome reduction in *M. genitalium*. The latter, with the consent of Rees and I, was created by Sophie Landon based on my GDS tool but this figure compares the wild-type, GAMA_236, and minesweeper_256 genomes. Landon's figure was used for our manuscript [101] (see section 6.1) but the GDS tool was used here.

6.4 Discussion

In this chapter we tried to combine all the work done to get a better understanding of our data and the reductome more generally.

Initially, just the minimal genomes found by GAMA and Rees' minesweeper were compared and analysed. A biological description of the genes present and absent in the minimal genomes was presented and we were able to show that minesweeper_256 was on a different path of convergence to GAMA_236. Further investigation showed that in addition to the high-essential genes found by GAMA_236 there were also two sets of genes that were vital to phosphate production in the cell which when combined acted as a low-essential set. Minesweeper_256 and GAMA_236 both picked different sources of phosphate.

By utilising more of the tools in the GDS, further analysis of all the rest of the data produced by the GDS showed multiple paths of convergence to different local minima. It also appeared that high and low-essential genes were creating the arms of convergence suggesting that they are critical to the shape of the solution space.

Finally, it was suspected that the model was over-estimating the amount of genes it could knockout by utilising false non-essential genes and so steps were presented to try and reduce the risk of failure of *in-vivo* experiments and present our results in such a way to make design of experiment easier depending on the resources and goals.

CHAPTER 7

CONCLUSION

7.1 Introduction

The aim of this thesis, as set out in chapter 1, was to contribute to the genome design community. Synthetic biology has huge potential to enable the creation of bio-machines and bio-materials that could revolutionise many industries and society in general. Having created the tools to build and modify genomes, synthetic biology now lacks methods to rationally decide what genomes to build or what to edit on existing genomes. We identify two underdeveloped areas that may help in furthering the goal to rationally design genomes, whole-cell models and minimal genomes.

The tools that exist to help design genomes are mostly *in-silico* and are very specific to particular implementations of particular types of models (e.g. COBRA for genome-scale metabolic models). Furthermore, the models used have inaccuracies like not taking into account systems-level effects within the cell. Whole-cell models, potentially, offer a solution to these problems but the only existing whole-cell model is very hard to use and brings technical challenges of requiring run-times longer than the maximum amount of time allowed on most HPC clusters and producing vast amounts of data - for the type of large-scale *in-silico* experiments likely to be necessary for genome design goals. In addition to the *big data* challenges, it is also hard to relate the data back to biological processes which is needed to interpret and understand the simulations. As a result, none of the tools that have been developed on other models have been extended to the whole-cell model of *M. genitalium*.

Minimal genomes may also help rationally design genomes by reducing the genome into its simplest form enabling easier analysis of the genotype-phenotype relationship. It has also been

suggested that the minimal genome may act as a base genome with which to design all synthetic organisms. Despite the high potential rewards, efforts made have mostly fallen short. The area of comparative genomics under-estimates the size of the minimal gene set, at least partly due to non-orthologous gene displacement. Whilst this does suggest that the answer to the question, *is there only one minimal genome?*, is *no*, it does not help us to understand how common minimal genomes are, if a particular organism can have more than one minimal genome, nor what an organism's viable genome reductions might look like generally. The incredible achievement by JCVI to produce a synthetic organism with a severely reduced genome falls short in similar ways in that it has failed to help understand minimal genomes. Their algorithm does not tell us if a minimal genome was found and does not even give any idea of if it is near a local minimum or not. An understanding of the whole solution space of genome reductions would be necessary to rapidly find minimal genomes generally, discover specific and general attributes of the reductome, and if there are multiple minimal genomes then which one(s) should be used to develop synthetic organisms? Knowing the solution space of all genome reductions of an organism could also help genome design in general since genome design would be performed on the solution space of all possible viable genomes and further research may be able to combine patterns of the viable reductome with patterns related to the design goal (e.g. increased growth rate).

Focussing on these two *weak points* in existing research towards the rational creation of synthetic organisms, we wanted to create a suite of *in-silico* tools to make it easier to perform large-scale *in-silico* experiments on cutting-edge biological models. The tools should enable *in-silico* experiments that are longer than the maximum allowed length on any given Linux cluster as well as bespoke data processing/storage/analysis solutions and biological interpretation. Furthermore, these tools must be adaptable to work on different clusters, models, design goals and design algorithms. These tools would then be used to perform and understand some designated genome design goal. The proof-of-concept goal was to develop genome reduction algorithms and use them to find the minimal genome of the whole-cell model of *M. genitalium*. This minimal genome would be described biologically, and the process of finding it would be used to try and better understand the solution space of genome reductions and minimal genomes in general.

The rest of this chapter will be split into sections looking at the progress made with regards to the creation of the *in-silico* tools, the use of the tools to better understand minimal genomes, concluding remarks and future directions.

7.2 *In-silico* tools to aid genome design

Chapter 4 described the construction of the GDS. The suite enables the user to convert an old PC into a kind of server that can manage massive *in-silico* experiments across multiple computer

clusters. The GDS can perform *in-silico* experiments that require more time than is allowed by the cluster and spread an experiment over multiple clusters providing a larger amount of CPUs and GPUs. This technology can enable researchers to perform *in-silico* experiments on a scale not possible before which could be very useful for genome design given the long running time of the only whole-cell model. The adaptability of the tools means that researchers from completely different fields could also utilise it.

The code was designed to be adaptable for different clusters, models, design goals, and design algorithms. This versatility was demonstrated to work on three different HPC clusters (BC3, BG, and C3DDB), two different design goals (genome reduction and fixing non-viable genomes), and 5 different algorithms (genetic algorithm, GAMA, dynamic probability distribution, genetic algorithm for complexes, and seeded viables). Unfortunately, there was not time to demonstrate this on a different model, but there is no reason to believe that it could not be done given the success of all the other aspects. These attributes not only allows the GDS to evolve with the field as new technology, models, design goals, and design algorithms become available, but it enables other researchers to utilise it. For example, if a researcher wanted to find the minimal genome of another model they would simply need to add subclasses to enable it to utilise their computing facilities and to run gene knockout experiments on the new model. Once that is done, they can apply GAMA to find the minimal genome. They may choose to add subclasses that optimise the growth rate or the increased production of some desired molecule.

The data processing and storage solutions used were bespoke and designed specifically for our situation and so may not be appropriate for others, however, the GDS has the adaptability to be modified to work with whatever solutions are used by the user.

The analysis and visualisation created both general and model-specific tools. The general tools are for analysing, comparing, and visualising genomes on the scale of genes. The model-specific tools enable the user to interpret data from the whole-cell model of *M. genitalium* biologically. Whilst the latter is model specific, it does show a framework that can be followed to adapt the GDS to do the same for a different model. The GDSs ability to automatically use Cytoscape provides a great platform to explore biological networks, and although this thesis only used KEGG maps, utilising other Cytoscape plug-ins can reveal lots of data and state-of-the-art visualisations (e.g. -omics data integration). The analysis part of the GDS is designed to use data from a remote database enabling groups of researchers to work independently whilst knowing that their biological analyses are equivalent. It is hoped that this will make collaborations and comparisons much easier within research groups. Practically, the analysis part of the genome design was crucial in our biological understanding of results as well as our investigation into the shape of the solution space.

Whilst the GDS functions as intended, it is not as easy to use as desired. This is due mostly to a lack of documentation and an online community of users. Over time it is hoped that it will be possible to build these up, but until then users will have to rely on my advice and help when using it for new tasks. Additionally, the first implementation has highlighted design flaws that have resulted in the code not being as modularised as is possible leading to a confusing structure in places and repeated code which is generally advised against by software developers. These can be corrected, like any software development process, by releasing a new version.

7.3 Massive *in-silico* experiments and discovering the reductome

The GDS was used to perform massive *in-silico* genome design experiments. The first focus was to develop algorithms to reduce genomes, and it was shown that although biological information could help, a learning aspect was needed to combat the almost zero chance of guessing large viable genome reductions. It was hypothesised that this was due to the combinational explosion in the solution space combined with reducing instances of viable genomes as gene knockouts increased. Genetic algorithms have been used in the past for genome design goals in genome-scale metabolic models[108], but it was shown that the long running time of the whole-cell model of *M. genitalium* meant that the convergence rate was prohibitively slow. Various ideas were implemented using the GDS, but we will only discuss the two most interesting here.

First, generation 0 of a standard genetic algorithm was seeded with a combination of non-viable minimal genome predictions and much larger but viable genomes which resulted in a marked increase in the rate of convergence of early generations suggesting that, according to the whole-cell model, at least some of the predictions were approximations of a minimal genome. It is interesting that the whole-cell model found that minimal genome predictions in the literature over-estimated the number of knockouts despite us finding that the model is also likely to over-estimate the number of genes to knockout but the over-estimation is with different genes. Further analysis may reveal either problems with the model and/or problems with the methods used to make the predictions.

The second algorithm is GAMA which exploited mathematical properties of the *principles of static essentiality* in two stages, the guess stage and the add stage which allowed rapid convergence to very small genomes. The smallest genomes found were then used to seed the mate stages of a standard genetic algorithm. This algorithm had the best convergence rate whilst still being able to look for high-essential genes. As a result, GAMA found a minimal genome with 165 genes knocked-out, GAMA_236.

With the help of Joshua Rees we were able to describe GAMA_236 using UniProt gene classifications. Further analysis of all genomes simulated showed that there were multiple paths of convergence, suggesting multiple local minima. Furthermore, the arms of convergence appeared to be dominated by high and low-essential genes, suggesting that genes with dynamic essentiality cause the shape of the solution space. In the literature, until relatively recently, dynamic essentiality had only been talked about in very specific cases that were found by experiment. Rancati et al. [19] have started to think about these concepts more generally by introducing the gradient of essentiality. This thesis attempts to take this generality further by defining the reductome and highlights the importance of dynamic essentiality on the shape of the solution space. This combined with the mathematical representation of genomes in chapter 3.3 takes us a small step closer to the quantification of all genomes which may help minimal genome research and rational genome design more generally.

Whilst GAMA_236 showed the success of GAMA in finding minimal genomes rapidly, the inconsistency of the resultant phenotype highlighted some of the limitations of the model with regards to genome essentiality testing. We tried to take into account potential inaccuracies in the model and proposed a framework with which to design experiments based on the results produced and the experimenter's goals.

Although the GDS has enabled massive *in-silico* experiments, the huge size of the solution space has a been a problem in both the genome design goals, suggesting that this will be a common problem in general. We were able to make significant improvements to genome reduction algorithms, but it still required a computer cluster and $\sim 1.5 - 2.5$ months of simulation time. We were not able to show similar speed-ups in convergence for the genome fixing algorithm, although it is believed that GAMA could inspire a similar strategy for fixing non-viable genomes. Even so, further speed-ups and more general methods are desirable. The quantification of all genomes may help to find and exploit patterns in the solution space in order to converge much faster to an optimisation objective. Another problem with GAMA is that it relies on the principles of static essentiality and so should the model underestimate the amount of dynamic essentiality or there are organisms that have high amounts of dynamic essentiality then the algorithm will perform worse and maybe even become inappropriate for the task.

GAMA and all other algorithms tested are heuristic algorithms that do not guarantee the global minimum/maximum. Due to GAMA not increasing the number of gene knockouts in 20 generations we can be fairly confident that it is close to a local maximum but cannot say if it is the global maximum. Repeating GAMA with random initial conditions can help find other local maxima thus combining to build a picture of the global landscape. Additionally, further analysis of the over 100,000 simulations already performed by the GDS may help us identify other local

maxima to investigate. Hope of finding exact algorithms is most likely found by creating and utilising a full mathematical representation of all genomes.

7.4 Concluding remarks and future directions

This thesis has been successful in its goals to build tools to enable rational genome design, develop genome reduction and genome fixing algorithms, find the minimal genome and learn about the reductome of the whole-cell model of *M. genitalium*. However, much work is still needed in order to fulfil its potential.

The GDS needs further developing in order to be easier to use/adapt as well as being more robust to dependency versions and operating systems. Furthermore, proper documentation and tutorials need to be produced and a community of users built in order to make this a truly successful open-source project.

The algorithms can be developed more and new ones created (e.g. faster genome reduction, optimising growth rate, production of industrially useful chemicals, or other desired phenotypes). A proper framework should be created so that algorithms can be quantitatively tested against one and another. New models should also be added so that different models and organisms can be analysed and compared. Whilst much of this kind of research has already been done on genome-scale metabolic models, there may still be value in adding the models since splitting experiments across multiple computer clusters may enable analysis on a new scale - for example simulating cells of an organ or the human microbiome. More generally extension of the models, design goals and design algorithms available in the GDS is a very important step to releasing the potential impact of the GDS on rational genome design as well as helping to test and improve models.

Analysis tools can always be improved but integrating more biological knowledge is a valuable goal. Further investigation into different distance metrics and dimension reduction algorithms may improve our visualisation of genomes. Also, our tools looked at genomes at the scale of genes whereas base-pairs, super-families, or some other representation may aid our analysis.

Creating a basis with which to describe genomes mathematically combined with our analysis of the genome provided a glimpse at the possibility of a mathematical theory of genome viability. Creating a rigorous mathematical representation of genome viability could impact both genome reduction and genome design research by enabling the exploitation of patterns to traverse the space of genomes to optimise certain objectives rapidly.

One of the most important steps is to combine the GDS with *in-vivo* experiments to create a design, build, test test cycle. Currently the main restriction is likely to be the accuracy of computer models but the GDS will be able to help improve them and evolve with the space provided that there is a community there to support it.

GLOSSARY

Bash A unix shell and command language. 23, 26

C++ C++ is a fast compiled programming language. 99

ClueGO A Cytoscape plugin that creates functional networks from sets of genes based on resources like GO annotation and KEGG pathway. 28

Cytoscape A network visualisation application designed with bioinformatics applications. 27, 28, 101, 157

DBSCAN Density-Based Spatial Clustering of Applications with Noise is a method to automatically calculate the number of clusters in a dataset and then cluster it using density based clustering from Pythons scikit-learn library. 28

DoS attack A denial of service attack is one that overloads a computer that is providing a service by making more requests than it can process. This prevents the computer from dealing with any legitimate requests. 53

iterable In the Python programming language an iterable is an array that has an iterator which is a way of iterating through the array without holding the entire array in memory. 67

KEGG The Kyoto Encyclopedia of Genes and Genomes is a collection of databases related to biological knowledge on all scales from molecular to organism level. 28, 157

Matlab A numerical computing environment and proprietary programming language from MathWorks. 19, 20, 26, 59, 173, 174

matplotlib A Python library that enables the creation of visualisations. 28

NCBI The National Center for Biotechnology Information is create, host and/or publish databases and tools related to biology. 28

Numpy Is a scientific computing library in Python which specialises in array processing and linear algebra. 26

Pandas Pandas is a data analysis library for Python to make manipulating, analysing and visualising structured data quick and easy. 26, 86

Pandas DataFrame The Pandas library uses DataFrame objects to deal with two dimensional tabular data in Python. 26, 58, 59, 64, 85, 95, 97, 98, 101

Pickle The Pickle library is serializing and de-serializing a Python object structure. It is often used to store and transfer Python objects. 26, 59, 85, 95, 97, 101, 131, 170

PostgreSQL A fully featured relational database management system. 131

Python Python is a interpreted, general-purpose programming language. 25–28, 37, 56, 59, 64, 67, 86, 97, 99, 101, 170, 174

SchemaCrawler A free database schema discovery and comprehension tool **??**. 26

scikit-learn A Python library that creates a framework to train and test machine learning algorithms and other related tasks. 26–28

seaborn A Python library that attempts to make it easier to make professional looking visualisations with matplotlib. 28

SLURM The Slurm Workload Manager (formerly known as Simple Linux Utility for Resource Management or SLURM), is a resource manager used on computer clusters. 24, 55

SQLite3 A reduced feature relational database management system. 26, 56, 85, 94, 97, 101, 131

standard error The standard error is the errors output from a program and displayed to the terminal. 55, 56

standard out The standard out is the output of a program displayed to the terminal. 55–57

TORQUE The Terascale Open-source Resource and QUEue Manager, is a resource manager used on computer clusters. 55

UML Unified Modeling Language is a way of visualising computer code for design and description. It is often used to describe 'objects' in object oriented programming languages. 25, 53, 56, 59

walltime The maximum amount of time that one expects a job on a computer cluster to need. The walltime must be specified when the job is submitted. Once the job has been running for that amount of time the job will be automatically terminated - even if the job is not finished. 41, 103

Acronyms

E. coli *Escherichia coli* 2, 15

M. capricolum *Mycoplasma capricolum* 3, 8, 152

M. genitalium *Mycoplasma genitalium* 4, 8, 13, 15, 17, 19, 21, 22, 26, 29, 31–33, 35, 39, 40, 44, 46, 53–57, 67, 84–86, 89, 96, 101, 103, 110, 114, 123, 126, 131, 133, 135–140, 148–158, 160, 170, 172–174

M. mycoides *Mycoplasma mycoides* 3, 8, 152

ACRC advanced computering reseach centre 30, 42, 43, 85

ARI adjusted rand index 26, 99, 139, 142–145, 147, 148

BC3 BlueCrystal III 23, 30, 38, 41–44, 53–55, 114, 116, 118, 121, 126, 131, 157

BCCS Bristol centre for complexity science 43

BG BlueGem 24, 38, 41, 42, 53–55, 114, 120, 121, 124, 126, 131, 134, 157, 174

BLAST basic local alignment search tool 28

BrisSynBio Bristol synthetic biology research centre 42, 43

C3DDB Commonwealth Computational Cloud for Data Driven Biology 24, 42, 44, 54, 55, 131, 157

COBRA COnstraint-Based Reconstruction and Analysis Toolbox 12, 155

CSV comma separated file 26

FBA flux balance analysis 11–13

GAMA guess, add, and mate algorithm 122, 124, 131, 154, 157–159, 170

GAMA_236 GAMA_236 154, 158, 159

GB gigabyte 30, 42–44

GDG genome design group 42, 137

GDS genome design suite 49, 51, 55, 61, 84–86, 95, 96, 100–104, 107–110, 114–116, 120, 121, 125–127, 131, 141–145, 149, 153, 154, 156–161

GO gene ontology 28

HPC high performance computing 12, 16, 17, 23, 24, 30, 42–44, 46, 50, 101, 131, 155, 157

Hub the central computer hub 23, 44, 51, 121

JCVI J. Craig Venter institute 3, 4, 16, 35, 40, 152, 156

KB kilobyte 43, 44

MB megabyte 41, 43, 44

minesweeper minesweeper 154

minesweeper_256 minesweeper_256 154

MIT massachusetts institute of technology 21, 42, 43

OS operating system 23

PCA principle component analysis 27, 99, 142, 143, 145

RDMS relational database management system 19, 26, 85, 91, 131

RDSF research data storage facility 43, 44, 84, 174

rRNA ribosomal RNA 31

SQL structured query language 85, 97

sRNA small RNA 31

SSH secure shell 25

TB terabyte 41, 43, 44, 46, 49, 170

tRNA transfer RNA 31

UoB University of Bristol 23, 24, 30, 42, 43

APPENDIX A

The supplementary information that accompanies this thesis can be found attached as SI.zip. When unzipped this produces a SI directory - the contents of this directory is explained in various sections below.

A.1 Initial genome reduction test

This section gives details of the initial genome reduction tests in section 3.1.3.

There are 525 genes in the MG-WC model of which 401 are characterised according to Karr et al. [52]. However we could only find GO terms for 316 genes. In order to maintain a fair comparison across tests we restricted all algorithms to search only from these 316 genes.

One can look at biological function from many scales, from many individual pathways up to a few general terms like *metabolism* and *membrane*. We felt for some of our measures individual pathways would be too much detail and the most general terms would be too little detail and decided to pick something in the middle. As a result we have split the gene annotation of *M. genitalium* over 44 GO terms - see table A.1 in the appendix.

When looking at the relationship of genetic knockouts to functions for viable and non-viable mutants we hypothesised about ways to identify key genes for survival. We split this in two two main groups:

Avoid overloading functions:
Due to the fact that *M. genitalium* is already reduced significantly by evolution we started by

making the assumption that all the functions were essential - each function could be tested individually at a later date if necessary. With the assumption that every function is essential for the survival of the cell we hypothesised that one way to *protect* the cell was to spread our genetic knockouts across functions so as to try to avoid disrupting any of the functions. The number of genes related to a function are not necessary the same across functions and so it was decided to bias our search so that functions with more genes associated to it are more proportionally more likely to be picked. Our algorithm is such:

1. Pick a function such that functions with more genes are more likely to be picked.

$$\mathscr{P}(\text{gene KO comes from function } f_i) = \frac{G(f_i)}{\sum_{j=1}^{N} G(f_j)}$$

   where $\mathscr{P}(X = x)$ is the probability that $X = x$, f_i is function i, $G(f_i)$ is the amount of genes associated to function f_i and N is the total number of functions.

2. Pick a random gene from the selected function using a uniform distribution.

3. Repeat process until the desired amount of genes are picked.

Avoid highly connected genes:
This hypothesis came from the idea that the connectivity of a gene may effect the chances of it being essential or not. We decided that a gene that is involved in many different functions is more likely to disrupt something essential than a gene that is only involved in one function. In addition to this the different functions need to feedback on each other to create an entity that is responsive to it's situation and environment and so we further hypothesised that the highly connected genes were key to this phenomena and thus more likely to be essential. This algorithm takes the form:

1. Pick genes so that genes connected with fewer functions are favoured.

$$\mathscr{P}(KO = g_i) = \frac{1 - \frac{F(g_i)}{\sum_{j=1}^{M} F(g_j)}}{M - 1}$$

   where $\mathscr{P}(KO = g_i)$ is the probability that gene g_i is knocked out, $F(g_i)$ is the number of functions associated to gene g_i and M is the total number of genes.

2. Repeat until the desired number of genes have been knocked out.

Of course there are many other network related measures that maybe useful but we left these until later to test should we decide we wish to carry on along that path.

Machine learning:
We hoped to further improve our algorithm by incorporating machine learning of some kind.

There are many different types but the two that seemed most suitable and achievable within a relatively short time frame were an genetic algorithm and an algorithm that has a dynamic probability distribution of picking combinations of genes.

Genetic algorithm:
This would be a normal genetic algorithm that was modified to incorporate any measures explained above, e.g. avoiding overloading functions or highly connected genes. This could be incorporated either into the initial random guess or in the picking of mutations.

This method would be relatively easy to implement but potentially not very flexible when incorporating ways of avoiding/targeting certain genes.

Dynamic probability distribution:
One can seed a probability distribution based on the results on empirical single knockout data and other potential methods mentioned above, e.g. avoiding overloading functions or highly connected genes. Simulations can be run and then depending on the results the distribution of picking genes can change to better pick viable gene combinations.

This method gives much more freedom to to implement complex ways of picking genes but will be significantly harder to implement and test than the genetic algorithm.

Test algorithms:
It was decided that in order to rate our algorithms we needed something to compare against. We picked two test cases.

1. Random guessing.

2. A basic genetic algorithm.

The first algorithm simply picks a gene randomly from a uniform distribution. This was picked as the baseline to beat. Beating this might still not be very good so we included a basic genetic algorithm to give some extra perspective - in addition to being an initial *glance* at how it performs on this problem.

A.1 Basic genetic algorithm: Let the vector $\vec{P}$ be an individual in the population where $\vec{P} = [p_1, p_2, ..., p_n]$ where p_i is either zero or one which represents whether a gene is knocked out or not, respectively, and n is the total amount of genes.

1. Pick two individuals, $\vec{P}_1$ and $\vec{P}_2$, from the population randomly such that a higher objective function value has a higher chance of being picked.

2. Randomly choose to keep the top/bottom half of $\vec{P}_1$ and the bottom/top half of $\vec{P}_2$.

3. Randomly choose what proportion of each individual to take.

4. Stick the two parts together to make a new individual.

5. Randomly pick between 0-10% of the genome to mutate and then randomly mutate it.

No viable combination is ever dropped from the potential mating pool although the probably of being picked to mate increases with the amount of knocked-out genes. This is done in order to try an avoid loosing potential paths to smaller genomes. On the other hand we keep mutation rates relatively low due to the high time-cost of simulation.

A.1.1 GO Functions in the whole-cell model

A.2 The genome design suite

Source code for the genome design suite can be found in the supplementary information directory at 'SI/hub/gds/src'.

Python scripts that started massive *in-silico* experiments on the hub can be found in the supplementary information directory at 'SI/hub_code/gds/run_files'.

Simulation data stored in `ko.db` can be found in the supplementary information directory as 'SI/remote_db/ko.db'.

The Python module used to ensure safe writing to `ko.db` can be found in the supplementary information directory as 'SI/remote_db/ko_db.py'.

Biological data on the whole-cell model of *M. genitalium* can be found in the supplementary information directory as 'SI/remote_db/static.db'.

Due to there being TBs of simulation data and thousands of cluster submission scripts this data is available upon request. An example of of a basic_summary Pickle file is given: 'SI/remote_db/example_basic_summary.pkl'.

Due to a technical error ko.db was not updated with the results of the guess stage of GAMA. The results from the guess stage are summarised in Pickle files: 'SI/remote_db/sims_missing_from_ko_db_from_guess_

Table A.1: A list of all GO functions for the 316 genes in the Whole-Cell model

Function
DNA metabolic process
DNA replication
DNA-dependent DNA replication
RNA methylation
RNA phosphodiester bond hydrolysis
carbohydrate transport
cell cycle
cellular component disassembly
cellular homeostasis
cellular modified amino acid metabolic process
cellular protein metabolic process
cellular protein modification process
cytoadherence to microvasculature, mediated by symbiont protein
fatty acid metabolic process
ion transport
lipid metabolic process
monosaccharide metabolic process
nucleobase-containing compound metabolic process
organonitrogen compound metabolic process
oxidation-reduction process
protein folding
protein targeting
proteolysis
pseudouridine synthesis
purine ribonucleoside salvage
pyrimidine nucleoside metabolic process
rRNA metabolic process
rRNA processing
response to oxidative stress
small molecule metabolic process
transcription, DNA-templated
translational initiation
transport

A.3 Karr2012

The original whole-cell model of *M. genitalium* paper and supplementary information can be found in the supplementary information directory as 'SI/karr2012'.

A.4 Designing minimal genomes using whole-cell models

Joshua Rees and I are co-first authors on a paper called 'Designing Minimal Genomes Using Whole-Cell models' this can be view on bioRxiv at `https://www.biorxiv.org/content/10.1101/344564v3.supplementary-material` and is currently submitted to Nature Communications. A copy of the paper and all supporting documentation/data can be found in the supplementary information of this thesis in the directory named 'SI/rees_chalkley_2019'.

A.5 Gene ontology

The thesis uses two different gene ontology classifications one using standard DAVID ontology classification done by myself and one stricter one that only used the UniProt database done by Joshua Rees.

The data collected by myself is available through the analysis part of the genome design suite.

The data collected by Joshua Rees is available in the supplimentary information. Within the supplimentary information there is a directory called 'rees_chalkley_2019' that contains everything related to our paper. Tabs I-K of SI_media_-1.xlsx contain Rees' classifications.

A.6 Estimating data storage requirements

get_wt200_file_sizes.sh

```
#!/bin/bash

# define variables
base_path=/projects/Minimal_genome_desing/jr_sims
save_file=${base_path}/simulation_sizes_wt200.txt
base_sim_path=${base_path}/WT200

# create a file with the data column name at the top
echo "simulation size (bytes)" > ${save_file}

# loop through all the possible directories
```

```
for dir_name in {1..200}
do
    sim_path=${base_sim_path}/${dir_name}
    # if the directory exists then sum up all the sizes # of filesthat
       end in .mat and append that number to
    # save_file
    if [ -d "${sim_path}" ]
        then du -c $(find ${sim_path} -name '*.mat') | \
        tail -n 1 | awk '{print $1}' >> ${save_file}
    fi
done
```

get_mean_size_of_wt_sim.py

```
#!/bin/python3

# import pandas
import pandas as pd

# set variables
sim_size_file = '/projects/Minimal_genome_desing/jr_sims/\
simulation_sizes_wt200.txt'

# read data into pandas dataframe
sim_sizes = pd.read_csv(sim_size_file)

# calculate the mean in MBs and print the result
print('The␣average␣size␣of␣a␣wildtype␣simulation␣is:␣', \
sim_sizes['simulation␣size␣(bytes)'].mean()/1024.0, \
'MBs')
```

A.7 Timing data extraction from raw simulation output

The whole-cell model of *M. genitalium* outputs hundreds of compressed Matlab files (i.e. '.mat' files). Since every file has to be loaded even if only one time series is needed it became clear that data extraction was slow. For this reason the following script was run on one of the original wild-type simulations in order to quantify how long it takes.

```
% This was performed on my original wild-type data on RDSF:
% /projects/Minimal_genome_desing/mg-wc/original_data/wild_type/1
```

```
% and was done to time how long it takes to extract a time series from
    the compressed matlab files.

% RESULT: Elapsed time is 31.321722 seconds.
tic
min_state_no = 1;
max_state_no = 338;

growth_data = []
for idx = min_state_no:max_state_no
        file_name = ['1/state-' num2str(idx) '.mat'];
        state_mat = load(file_name);
        tmp_growth = squeeze(state_mat.MetabolicReaction.growth);
        growth_data = [growth_data;tmp_growth];
end
toc
```

The script was run on RDSF in the directory '/projects/Minimal_genome_desing/mg-wc/original_data/wild_ty
and loads the growth rate from each the 338 state-*.mat files from directory '1'. It took just over 31 seconds to achieve this.

A.8 Changing the default file saving behaviour in the whole-cell model of *M. genitalium*

The whole-cell model of *M. genitalium* saves simulation data into compressed Matlab files (i.e. either version 7.0 or version 6 if the amount data becomes larger than the maxmum allowed on version 7.0), however, this version sometimes causes errors uncompressing them with Python's `scipy` . In order to stop these errors, the source code of the model was changed to save into the uncompressed version (i.e. 7.3). The file version is specified in the `DiskLogger` class at '/projects/flex1/database/WholeCell-master/src/+edu/+stanford/+covert/+cell/+sim/+util/DiskLogger.m' on lines 916 and 932 - path is relative to an arbitrary BG login node but can also be found in the supplimentary information at the same path relative to 'WholeCell-master'.

BIBLIOGRAPHY

[1] S. Yoshida, K. Hiraga, T. Takehana, I. Taniguchi, H. Yamaji, Y. Maeda, K. Toyohara, K. Miyamoto, Y. Kimura, K. Oda, A bacterium that degrades and assimilates poly(ethylene terephthalate)., Science (New York, N.Y.) 351 (6278) (2016) 1196–9.
doi:10.1126/science.aad6359.
URL http://www.ncbi.nlm.nih.gov/pubmed/26965627

[2] T. Tasoulis, G. Isbister, T. Tasoulis, G. K. Isbister, A Review and Database of Snake Venom Proteomes, Toxins 9 (9) (2017) 290.
doi:10.3390/toxins9090290.
URL http://www.mdpi.com/2072-6651/9/9/290

[3] D. Choe, S. Cho, S. C. Kim, B.-K. Cho, Minimal genome: Worthwhile or worthless efforts toward being smaller?, Biotechnology journal (sep 2015).
doi:10.1002/biot.201400838.
URL http://www.ncbi.nlm.nih.gov/pubmed/26356135

[4] T. P. Howard, S. Middelhaufe, K. Moore, C. Edner, D. M. Kolak, G. N. Taylor, D. A. Parker, R. Lee, N. Smirnoff, S. J. Aves, J. Love, Synthesis of customized petroleum-replica fuel molecules by targeted modification of free fatty acid pools in Escherichia coli., Proceedings of the National Academy of Sciences of the United States of America 110 (19) (2013) 7636–41.
doi:10.1073/pnas.1215966110.
URL http://www.ncbi.nlm.nih.gov/pubmed/23610415http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3651483

[5] M. Kalos, B. L. Levine, D. L. Porter, S. Katz, S. A. Grupp, A. Bagg, C. H. June, T cells with chimeric antigen receptors have potent antitumor effects and can establish memory in patients with advanced leukemia., Science translational medicine 3 (95) (2011) 95ra73.
doi:10.1126/scitranslmed.3002842.
URL http://www.ncbi.nlm.nih.gov/pubmed/21832238http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3393096

[6] C. Darwin, The Variation of Animals and Plants under Domestication Volume 2, Cambridge University Press, 2010.

URL https://books.google.co.uk/books?hl=en{&}lr={&}id={_}uONOAO7qwYC{&}oi=fnd{&}pg=PA1{&}dq=Variation+of+Animals+and+Plants+under+Domestication{&}ots=hqeZySDFfI{&}sig=YY5rjn06wS5HccnZ8kvVEMC7bpU{#}v=onepage{&}q=VariationofAnimalsandPlantsunderDomestication{&}f=false

[7] H. J. Schouten, E. Jacobsen, Are mutations in genetically modified plants dangerous?, Journal of biomedicine & biotechnology 2007 (7) (2007) 82612.
doi:10.1155/2007/82612.
URL http://www.ncbi.nlm.nih.gov/pubmed/18273413http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2218926

[8] J. D. Watson, F. H. C. Crick, A Structure for Deoxyribose Nucleic Acid, Tech. Rep. 1 (1953).
URL https://www.3dmoleculardesigns.com/3DMD-Files/DNA-Discovery/PDFs/AnnotatedWatsonandCrickpaper.pdf

[9] S. N. Cohen, A. C. Y. Chang, H. W. Boyer, R. B. Hellingt, Construction of Biologically Functional Bacterial Plasmids In Vitro (R factor/restriction enzyme/transformation/endonuclease/antibiotic resistance), Tech. Rep. 11 (1973).
URL https://www.pnas.org/content/pnas/70/11/3240.full.pdf

[10] J. F. Morrow, S. N. Cohen, A. C. Chang, H. W. Boyer, H. M. Goodman, R. B. Helling, Replication and transcription of eukaryotic DNA in Escherichia coli., Proceedings of the National Academy of Sciences of the United States of America 71 (5) (1974) 1743–7.
doi:10.1073/PNAS.71.5.1743.
URL http://www.ncbi.nlm.nih.gov/pubmed/4600264http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC388315

[11] R. Chari, G. M. Church, Beyond editing to writing large genomes, Nature Reviews Genetics 18 (12) (2017) 749–760.
doi:10.1038/nrg.2017.59.
URL http://www.nature.com/doifinder/10.1038/nrg.2017.59

[12] C. A. Lino, J. C. Harper, J. P. Carney, J. A. Timlin, Delivering CRISPR: a review of the challenges and approaches, Drug Delivery 25 (1) (2018) 1234–1257.
doi:10.1080/10717544.2018.1474964.
URL https://www.tandfonline.com/doi/full/10.1080/10717544.2018.1474964

[13] D. G. Gibson, J. I. Glass, C. Lartigue, V. N. Noskov, R.-Y. Chuang, M. A. Algire, G. A. Benders, M. G. Montague, L. Ma, M. M. Moodie, C. Merryman, S. Vashee, R. Krishnakumar, N. Assad-Garcia, C. Andrews-Pfannkoch, E. A. Denisova, L. Young, Z.-Q. Qi, T. H. Segall-Shapiro, C. H. Calvey, P. P. Parmar, C. A. Hutchison, H. O. Smith, J. C. Venter,

Creation of a bacterial cell controlled by a chemically synthesized genome., Science (New York, N.Y.) 329 (5987) (2010) 52–6.
doi:10.1126/science.1190719.
URL http://www.sciencemag.org/content/329/5987/52.abstract

[14] O. Purcell, B. Jain, J. R. Karr, M. W. Covert, T. K. Lu, Towards a whole-cell modeling approach for synthetic biology., Chaos (Woodbury, N.Y.) 23 (2) (2013) 025112.
doi:10.1063/1.4811182.
URL http://www.ncbi.nlm.nih.gov/pubmed/23822510

[15] D. G. Gibson, Programming biological operating systems: genome design, assembly and activation, Nature Methods 11 (5) (2014) 521–526.
doi:10.1038/nmeth.2894.
URL http://www.nature.com/articles/nmeth.2894

[16] M. Dragosits, D. Mattanovich, Adaptive laboratory evolution – principles and applications for biotechnology, Microbial Cell Factories 12 (1) (2013) 64.
doi:10.1186/1475-2859-12-64.
URL http://microbialcellfactories.biomedcentral.com/articles/10.1186/1475-2859-12-64

[17] K. M. Esvelt, H. H. Wang, Genome-scale engineering for systems and synthetic biology., Molecular systems biology 9 (1) (2013) 641.
doi:10.1038/msb.2012.66.
URL http://www.ncbi.nlm.nih.gov/pubmed/23340847http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3564264

[18] S. Y. Lee, H. U. Kim, Systems strategies for developing industrial microbial strains, Nature Biotechnology 33 (10) (2015) 1061–1072.
doi:10.1038/nbt.3365.
URL http://www.nature.com/articles/nbt.3365

[19] G. Rancati, J. Moffat, A. Typas, N. Pavelka, Emerging and evolving concepts in gene essentiality, Nature Reviews Genetics 19 (1) (2017) 34–49.
doi:10.1038/nrg.2017.74.
URL http://www.nature.com/doifinder/10.1038/nrg.2017.74

[20] K. Dybvig, L. L. Voelker, D. Yogev, R. Rosengarten, R. Watson-McKown, K. S. Wise, K. Deitsch, E. Moxon, T. Wellems, J. Cairns, J. Overbaugh, S. Miller, D. C. Krause, M. F. Balish, M. W. van der Woude, A. J. Bäumler, S. Razin, L. Hayflick, Z. B. Zeng, C. C. Cockerham, M. Miyata, S. Seto, Mutation Models and Quantitative Genetic Variation, Genetics 38 (3) (2004) 183–90.

doi:10.1016/j.biologicals.2009.11.008.
URL http://www.sciencedirect.com/science/article/pii/S030090849900209 6http://www.genetics.org/content/133/3/729.shorthttp://www.sciencedirec t.com/science/article/pii/S1045105609001808http://cmr.asm.org/content/ 17/3/581.shorthttp://doi.wiley.com/10.1046/j.

[21] H. J. Morowitz, D. C. Wallace, Genome size and life cycle of the mycoplasma, Annals of the New York Academy of Sciences 225 (1 Mycoplasma an) (1973) 62–73.
doi:10.1111/j.1749-6632.1973.tb45637.x.
URL http://doi.wiley.com/10.1111/j.1749-6632.1973.tb45637.x

[22] H. Neimark, Origin and evolution of wall-less prokaryotes, The bacterial L-Forms. Marcel Dekkar Inc. New York (1986).
URL https://scholar.google.co.uk/scholar?hl=en{&}as{_}sdt=0{%}2C5{&}q =Origin+and+evolution+of+wall-less+prokaryotes.+Neimark{&}btnG=

[23] C. R. Woese, J. Maniloff, L. B. Zablen, Phylogenetic analysis of the mycoplasmas., Proceedings of the National Academy of Sciences of the United States of America 77 (1) (1980) 494–8.
URL http://www.ncbi.nlm.nih.gov/pubmed/6928642http://www.pubmedcentral. nih.gov/articlerender.fcgi?artid=PMC348298

[24] J. I. Glass, C. Merryman, K. S. Wise, C. A. Hutchison, H. O. Smith, Minimal Cells-Real and Imagined., Cold Spring Harbor perspectives in biology (2017) a023861doi:10.1101/cs hperspect.a023861.
URL http://www.ncbi.nlm.nih.gov/pubmed/28348033

[25] A. R. Mushegian, E. V. Koonin, A minimal gene set for cellular life derived by comparison of complete bacterial genomes., Proceedings of the National Academy of Sciences 93 (19) (1996) 10268–10273.
doi:10.1073/pnas.93.19.10268.
URL http://www.pnas.org/content/93/19/10268.abstract

[26] R. Gil, F. J. Silva, J. Peretó, A. Moya, Determination of the core of a minimal bacterial gene set., Microbiology and molecular biology reviews : MMBR 68 (3) (2004) 518–37, table of contents.
doi:10.1128/MMBR.68.3.518-537.2004.
URL http://www.ncbi.nlm.nih.gov/pubmed/15353568http://www.pubmedcentra l.nih.gov/articlerender.fcgi?artid=PMC515251

[27] K. Lagesen, D. W. Ussery, T. M. Wassenaar, Genome update: the 1000th genome–a cautionary tale., Microbiology (Reading, England) 156 (Pt 3) (2010) 603–8.

doi:10.1099/mic.0.038257-0.
URL http://www.ncbi.nlm.nih.gov/pubmed/20093288

[28] W. Liu, L. Fang, M. Li, S. Li, S. Guo, R. Luo, Z. Feng, B. Li, Z. Zhou, G. Shao, H. Chen, S. Xiao, Comparative Genomics of Mycoplasma: Analysis of Conserved Essential Genes and Diversity of the Pan-Genome, PLoS ONE 7 (4) (2012) e35698.
doi:10.1371/journal.pone.0035698.
URL https://dx.plos.org/10.1371/journal.pone.0035698

[29] J. I. Glass, N. Assad-Garcia, N. Alperovich, S. Yooseph, M. R. Lewis, M. Maruf, C. A. Hutchison, H. O. Smith, J. C. Venter, Essential genes of a minimal bacterium., Proceedings of the National Academy of Sciences of the United States of America 103 (2) (2006) 425–30.
doi:10.1073/pnas.0510013103.
URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1324956{&}tool=pmcentrez{&}rendertype=abstract

[30] P. Gawand, F. Said Abukar, N. Venayak, S. Partow, A. E. Motter, R. Mahadevan, Suboptimal phenotypes of double-knockout mutants of Escherichia coli depend on the order of gene deletions., Integrative biology : quantitative biosciences from nano to macro 7 (8) (2015) 930–9.
doi:10.1039/c5ib00096c.
URL http://pubs.rsc.org/en/Content/ArticleHTML/2015/IB/C5IB00096C

[31] C. A. Hutchison, R.-Y. Chuang, V. N. Noskov, N. Assad-Garcia, T. J. Deerinck, M. H. Ellisman, J. Gill, K. Kannan, B. J. Karas, L. Ma, J. F. Pelletier, Z.-Q. Qi, R. A. Richter, E. A. Strychalski, L. Sun, Y. Suzuki, B. Tsvetanova, K. S. Wise, H. O. Smith, J. I. Glass, C. Merryman, D. G. Gibson, J. C. Venter, Design and synthesis of a minimal bacterial genome, Science 351 (6280) (2016) aad6253–aad6253.
doi:10.1126/science.aad6253.
URL http://science.sciencemag.org/content/351/6280/aad6253.abstract

[32] D. Machado, R. S. Costa, M. Rocha, E. C. Ferreira, B. Tidor, I. Rocha, Modeling formalisms in Systems Biology., AMB Express 1 (1) (2011) 45.
doi:10.1186/2191-0855-1-45.
URL http://www.scopus.com/inward/record.url?eid=2-s2.0-84867843623{&}partnerID=tZOtx3y1

[33] N. D. Theise, M. D'Inverno, Understanding cell lineages as complex adaptive systems, Blood Cells, Molecules, and Diseases 32 (1) (2004) 17–20.

doi:10.1016/J.BCMD.2003.09.010.
URL https://www.sciencedirect.com/science/article/pii/S1079979603002523

[34] J. D. Orth, I. Thiele, B. Ø. Palsson, What is flux balance analysis?, Nature biotechnology 28 (3) (2010) 245–8.
doi:10.1038/nbt.1614.
URL http://www.ncbi.nlm.nih.gov/pubmed/20212490http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3108565

[35] S. Sastry, Nonlinear system : analysis, stability, and control, Springer, 1999.

[36] J. B. Carrell, Fundamentals of linear algebra, Tech. rep. (2005).
URL https://www.math.ubc.ca/{~}carrell/NB.pdf

[37] M. Tomita, Whole-cell simulation: a grand challenge of the 21st century, Trends in Biotechnology 19 (6) (2001) 205–210.
doi:10.1016/S0167-7799(01)01636-5.
URL https://www.sciencedirect.com/science/article/pii/S0167779901016365

[38] J. Carrera, M. W. Covert, Why Build Whole-Cell Models?, Trends in Cell Biology 25 (12) (2015) 719–722.
doi:10.1016/J.TCB.2015.09.004.
URL https://www.sciencedirect.com/science/article/pii/S0962892415001701

[39] C. T. Trinh, A. Wlaschin, F. Srienc, Elementary mode analysis: a useful metabolic pathway analysis tool for characterizing cellular metabolism., Applied microbiology and biotechnology 81 (5) (2009) 813–26.
doi:10.1007/s00253-008-1770-1.
URL http://www.scopus.com/inward/record.url?eid=2-s2.0-58149154663{&}partnerID=tZOtx3y1http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2909134{&}tool=pmcentrez{&}rendertype=abstract

[40] K. Plaimas, J.-P. Mallm, M. Oswald, F. Svara, V. Sourjik, R. Eils, R. König, A. P. Burgard, S. Vaidyaraman, C. D. Maranas, J. D. Orth, I. Thiele, B. Ø. O. Palsson, A. M. Feist, C. S. Henry, J. L. Reed, M. Krummenacker, A. R. Joyce, P. D. Karp, L. J. Broadbelt, V. Hatzimanikatis, B. Ø. O. Palsson, C. T. Trinh, A. Wlaschin, F. Srienc, N. D. Price, J. L. Reed, B. Ø. O. Palsson, C. B. Milne, P.-J. Kim, J. A. Eddy, N. D. Price, J. S. Edwards, B. Ø. O. Palsson, E. O'Brien, J. Monk, B. Ø. O. Palsson, M. W. Covert, C. H. Schilling,

B. Ø. O. Palsson, Elementary mode analysis: a useful metabolic pathway analysis tool for characterizing cellular metabolism., Biotechnology Journal 2 (5) (2009) 813–26.
doi:10.1007/s00253-008-1770-1.
URL http://www.sciencedirect.com/science/article/pii/S002251930192405 1http://www.ncbi.nlm.nih.gov/pubmed/26000478http://www.biomedcentral.com/1471-2105/1/1http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3160784{&}tool=pmcentrez{&}rendertype=abstract

[41] N. E. Lewis, H. Nagarajan, B. O. Palsson, Constraining the metabolic genotype-phenotype relationship using a phylogeny of in silico methods., Nature reviews. Microbiology 10 (4) (2012) 291–305.
doi:10.1038/nrmicro2737.
URL http://www.scopus.com/inward/record.url?eid=2-s2.0-84858439602{&}partnerID=tZOtx3y1

[42] A. Bordbar, J. M. Monk, Z. A. King, B. O. Palsson, Constraint-based models predict metabolic and associated cellular functions., Nature reviews. Genetics 15 (2) (2014) 107–20.
doi:10.1038/nrg3643.
URL http://www.nature.com/nrg/journal/v15/n2/full/nrg3643.html{#}t1http://dx.doi.org/10.1038/nrg3643

[43] S. A. Becker, A. M. Feist, M. L. Mo, G. Hannum, B. Ø. Palsson, M. J. Herrgard, Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox, Nature Protocols 2 (3) (2007) 727–738.
doi:10.1038/nprot.2007.99.
URL http://www.nature.com/articles/nprot.2007.99

[44] J. Schellenberger, R. Que, R. M. T. Fleming, I. Thiele, J. D. Orth, A. M. Feist, D. C. Zielinski, A. Bordbar, N. E. Lewis, S. Rahmanian, J. Kang, D. R. Hyduke, B. Ø. Palsson, Quantitative prediction of cellular metabolism with constraint-based models: the COBRA Toolbox v2.0, Nature Protocols 6 (9) (2011) 1290–1307.
doi:10.1038/nprot.2011.308.
URL http://www.nature.com/articles/nprot.2011.308

[45] C. J. Lloyd, A. Ebrahim, L. Yang, Z. A. King, E. Catoiu, E. J. O'Brien, J. K. Liu, B. O. Palsson, COBRAme: A computational framework for genome-scale models of metabolism and gene expression, PLoS Computational Biology 14 (7) (2018).
doi:10.1371/journal.pcbi.1006302.

[46] J. S. Edwards, B. O. Palsson, The Escherichia coli MG1655 in silico metabolic genotype: its definition, characteristics, and capabilities., Proceedings of the National Academy of

Sciences of the United States of America 97 (10) (2000) 5528–33.
URL http://www.ncbi.nlm.nih.gov/pubmed/10805808http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC25862

[47] C. H. Schilling, M. W. Covert, I. Famili, G. M. Church, J. S. Edwards, B. O. Palsson, Genome-Scale Metabolic Model of Helicobacter pylori 26695, Journal of Bacteriology 184 (16) (2002) 4582.
doi:10.1128/JB.184.16.4582-4593.2002.
URL http://www.ncbi.nlm.nih.gov/pubmed/12142428http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC135230

[48] J. Förster, I. Famili, B. Ø. Palsson, J. Nielsen, Large-Scale Evaluation of In Silico Gene Deletions in Saccharomyces cerevisiae, OMICS: A Journal of Integrative Biology 7 (2) (2003) 193–202.
doi:10.1089/153623103322246584.
URL http://www.liebertpub.com/doi/10.1089/153623103322246584

[49] Y.-K. Oh, B. O. Palsson, S. M. Park, C. H. Schilling, R. Mahadevan, Genome-scale reconstruction of metabolic network in Bacillus subtilis based on high-throughput phenotyping and gene essentiality data., The Journal of biological chemistry 282 (39) (2007) 28791–9.
doi:10.1074/jbc.M703759200.
URL http://www.ncbi.nlm.nih.gov/pubmed/17573341

[50] Y. Shinfuku, N. Sorpitiporn, M. Sono, C. Furusawa, T. Hirasawa, H. Shimizu, Development and experimental verification of a genome-scale metabolic model for Corynebacterium glutamicum., Microbial cell factories 8 (2009) 43.
doi:10.1186/1475-2859-8-43.
URL http://www.ncbi.nlm.nih.gov/pubmed/19646286http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2728707

[51] D. Noble, A theory of biological relativity: no privileged level of causation, Interface Focus 2 (1) (2012) 55–64.
doi:10.1098/rsfs.2011.0067.
URL http://rsfs.royalsocietypublishing.org/cgi/doi/10.1098/rsfs.2011.0067

[52] J. R. Karr, J. C. Sanghvi, D. N. Macklin, M. V. Gutschow, J. M. Jacobs, B. Bolival, N. Assad-Garcia, J. I. Glass, M. W. Covert, A whole-cell computational model predicts phenotype from genotype., Cell 150 (2) (2012) 389–401.
doi:10.1016/j.cell.2012.05.044.
URL http://www.sciencedirect.com/science/article/pii/S0092867412007763

[53] D. N. Macklin, N. A. Ruggero, M. W. Covert, The future of whole-cell modeling, Current Opinion in Biotechnology 28 (2014) 111–115.
doi:10.1016/J.COPBIO.2014.01.012.
URL https://www.sciencedirect.com/science/article/pii/S0958166914000251

[54] J. R. Karr, J. C. Sanghvi, D. N. Macklin, A. Arora, M. W. Covert, WholeCellKB: model organism databases for comprehensive whole-cell models., Nucleic acids research 41 (Database issue) (2013) D787–92.
doi:10.1093/nar/gks1108.
URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3531061{&}tool=pmcentrez{&}rendertype=abstract

[55] J. R. Karr, N. C. Phillips, M. W. Covert, WholeCellSimDB: a hybrid relational/HDF database for whole-cell model predictions., Database : the journal of biological databases and curation 2014 (jan 2014).
doi:10.1093/database/bau095.
URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4165886{&}tool=pmcentrez{&}rendertype=abstract

[56] R. Lee, J. R. Karr, M. W. Covert, WholeCellViz: data visualization for whole-cell models., BMC bioinformatics 14 (2013) 253.
doi:10.1186/1471-2105-14-253.
URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3765349{&}tool=pmcentrez{&}rendertype=abstract

[57] J. C. Sanghvi, S. Regot, S. Carrasco, J. R. Karr, M. V. Gutschow, B. Bolival, M. W. Covert, Accelerated discovery via a whole-cell model, Nature Methods 10 (12) (2013) 1192–1195.
doi:10.1038/nmeth.2724.
URL http://www.nature.com/articles/nmeth.2724

[58] J. R. Karr, A. H. Williams, J. D. Zucker, A. Raue, B. Steiert, J. Timmer, C. Kreutz, S. Wilkinson, B. A. Allgood, B. M. Bot, B. R. Hoff, M. R. Kellen, M. W. Covert, G. A. Stolovitzky, P. Meyer, P. Meyer, Summary of the DREAM8 Parameter Estimation Challenge: Toward Parameter Identification for Whole-Cell Models, PLOS Computational Biology 11 (5) (2015) e1004096.
doi:10.1371/journal.pcbi.1004096.
URL https://dx.plos.org/10.1371/journal.pcbi.1004096

[59] M. Feig, R. Harada, T. Mori, I. Yu, K. Takahashi, Y. Sugita, Complete atomistic model of a bacterial cytoplasm for integrating physics, biochemistry, and systems biology, Journal of Molecular Graphics and Modelling 58 (2015) 1–9.

doi:10.1016/J.JMGM.2015.02.004.
URL https://www.sciencedirect.com/science/article/pii/S1093326315000388

[60] I. Yu, T. Mori, T. Ando, R. Harada, J. Jung, Y. Sugita, M. Feig, Biomolecular interactions modulate macromolecular structure and dynamics in atomistic model of a bacterial cytoplasmdoi:10.7554/eLife.19274.001.
URL https://cdn.elifesciences.org/articles/19274/elife-19274-v1.pdf

[61] T. MathWorks, MATLAB 2012a, The MathWorks, Natick, 2012. (2012).

[62] F. s. f. GNU, GNU Bash, https://www.gnu.org/software/bash/.

[63] U. o. B. Advanced Computing Research Center (ACRC), BlueCrystal Phase 3, https://www.acrc.bris.ac.uk/acrc/phase3.htm.

[64] Rsync, https://rsync.samba.org/.

[65] G. v. . C. v. W. e. I. C. Rossum, Python tutorial, Python 206 (10) (1995) 1600–1600.
URL https://dl.acm.org/citation.cfm?id=869378{&}preflayout=flat

[66] J. C. Mitchell, Concepts in Programming Languages, Cambridge University Press, Cambridge, 2002.
doi:10.1017/CBO9780511804175.
URL http://ebooks.cambridge.org/ref/id/CBO9780511804175

[67] D. Phillips, Python 3 Object Oriented Programming, Packt Pub, 2010.
arXiv:arXiv:1011.1669v3, doi:10.1007/s13398-014-0173-7.2.

[68] PyLint, https://www.pylint.org/.

[69] SQLite, https://www.sqlite.org/index.html.

[70] SchemaCrawler, https://www.schemacrawler.com/.

[71] S. van der Walt, S. C. Colbert, G. Varoquaux, The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering 13 (2) (2011) 22–30.
doi:10.1109/MCSE.2011.37.
URL http://ieeexplore.ieee.org/document/5725236/

[72] W. Mckinney, pandas: a Foundational Python Library for Data Analysis and Statistics, Tech. rep.
URL http://pandas.sf.net

[73] W. M. Rand, Objective Criteria for the Evaluation of Clustering Methods, Journal of the American Statistical Association 66 (336) (1971) 846.
doi:10.2307/2284239.
URL https://www.jstor.org/stable/2284239?origin=crossref

[74] J. M. Santos, M. Embrechts, On the Use of the Adjusted Rand Index as a Metric for Evaluating Supervised Classification, Springer, Berlin, Heidelberg, 2009, pp. 175–184.
doi:10.1007/978-3-642-04277-5_18.
URL http://link.springer.com/10.1007/978-3-642-04277-5{_}18

[75] S. Boyd, L. Vandenberghe, Introduction to Applied Linear Algebra Vectors, Matrices, and Least Squares (2018).
doi:10.1017/9781108583664.
URL www.cambridge.org

[76] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, T. Ideker, Cytoscape: a software environment for integrated models of biomolecular interaction networks., Genome research 13 (11) (2003) 2498–504.
doi:10.1101/gr.1239303.
URL http://www.ncbi.nlm.nih.gov/pubmed/14597658http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC403769

[77] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, É. Duchesnay, Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research 12 (Oct) (2011) 2825–2830.
URL http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

[78] I. T. Jolliffe, J. Cadima, Principal component analysis: a review and recent developments, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 374 (2065) (2016) 20150202.
doi:10.1098/rsta.2015.0202.
URL http://rsta.royalsocietypublishing.org/lookup/doi/10.1098/rsta.2015.0202

[79] V. Estivill-Castro, Vladimir, Why so many clustering algorithms, ACM SIGKDD Explorations Newsletter 4 (1) (2002) 65–75.
doi:10.1145/568574.568575.
URL http://portal.acm.org/citation.cfm?doid=568574.568575

[80] J. D. Hunter, Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering 9 (3) (2007) 90–95.

doi:10.1109/MCSE.2007.55.
URL http://ieeexplore.ieee.org/document/4160265/

[81] M. Waskom, O. Botvinnik, P. Hobson, J. B. Cole, Y. Halchenko, S. Hoyer, A. Miles, T. Augspurger, T. Yarkoni, T. Megies, L. P. Coelho, D. Wehner, Cynddl, E. Ziegler, Diego0020, Y. V. Zaytsev, T. Hoppe, S. Seabold, P. Cloud, M. Koskinen, K. Meyer, A. Qalieh, D. Allan, seaborn: v0.5.0 (November 2014), Tech. rep. (nov 2014).
doi:10.5281/ZENODO.12710.
URL https://zenodo.org/record/12710{#}.XJKjU3BpHrc

[82] S. C. Johnson, Hierarchical clustering schemes, Psychometrika 32 (3) (1967) 241–254.
doi:10.1007/BF02289588.
URL http://link.springer.com/10.1007/BF02289588

[83] A. P. Davis, J. E. Richardson, S. Lewis, D. Botstein, J. C. Matese, H. Butler, C. A. Ball, L. Issel-Tarver, K. Dolinski, G. Sherlock, D. P. Hill, M. A. Harris, M. Ringwald, S. S. Dwight, A. Kasarskis, J. M. Cherry, J. A. Blake, G. M. Rubin, M. Ashburner, J. T. Eppig, Gene Ontology: tool for the unification of biology, Nature Genetics 25 (1) (2002) 25–29.
arXiv:10614036, doi:10.1038/75556.
URL http://www.ncbi.nlm.nih.gov/pubmed/10802651http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3037419http://www.nature.com/articles/ng0500{_}25

[84] The Gene Ontology Resource: 20 years and still GOing strong, Nucleic acids research 47 (D1) (2019) D330–D338.
doi:10.1093/nar/gky1055.
URL https://academic.oup.com/nar/article/47/D1/D330/5160994

[85] M. Kanehisa, S. Goto, KEGG: kyoto encyclopedia of genes and genomes., Nucleic acids research 28 (1) (2000) 27–30.
URL http://www.ncbi.nlm.nih.gov/pubmed/10592173http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC102409

[86] M. Kanehisa, M. Furumichi, M. Tanabe, Y. Sato, K. Morishima, KEGG: new perspectives on genomes, pathways, diseases and drugs, Nucleic Acids Research 45 (D1) (2017) D353–D361.
doi:10.1093/nar/gkw1092.
URL http://www.ncbi.nlm.nih.gov/pubmed/27899662http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5210567https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkw1092

[87] M. Kanehisa, Y. Sato, M. Furumichi, K. Morishima, M. Tanabe, New approach for understanding genome variations in KEGG, Nucleic Acids Research 47 (D1) (2019) D590–D595.
doi:10.1093/nar/gky962.
URL http://www.ncbi.nlm.nih.gov/pubmed/30321428http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC6324070https://academic.oup.com/nar/article/47/D1/D590/5128935

[88] M. Kanehisa, Post-genome informatics, Oxford University Press, 2000.

[89] D. L. Wheeler, T. Barrett, D. A. Benson, S. H. Bryant, K. Canese, V. Chetvernin, D. M. Church, M. DiCuccio, R. Edgar, S. Federhen, L. Y. Geer, Y. Kapustin, O. Khovayko, D. Landsman, D. J. Lipman, T. L. Madden, D. R. Maglott, J. Ostell, V. Miller, K. D. Pruitt, G. D. Schuler, E. Sequeira, S. T. Sherry, K. Sirotkin, A. Souvorov, G. Starchenko, R. L. Tatusov, T. A. Tatusova, L. Wagner, E. Yaschenko, Database resources of the National Center for Biotechnology Information, Nucleic Acids Research 35 (Database) (2007) D5–D12.
doi:10.1093/nar/gkl1031.
URL https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkl1031

[90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, Basic local alignment search tool, Journal of Molecular Biology 215 (3) (1990) 403–410.
doi:10.1016/S0022-2836(05)80360-2.
URL https://linkinghub.elsevier.com/retrieve/pii/S0022283605803602

[91] J. Ye, S. McGinnis, T. L. Madden, BLAST: improvements for better sequence analysis, Nucleic Acids Research 34 (Web Server) (2006) W6–W9.
doi:10.1093/nar/gkl164.
URL https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkl164

[92] G. Bindea, B. Mlecnik, H. Hackl, P. Charoentong, M. Tosolini, A. Kirilovsky, W.-H. Fridman, F. Pagès, Z. Trajanoski, J. Galon, ClueGO: a Cytoscape plug-in to decipher functionally grouped gene ontology and pathway annotation networks, Bioinformatics 25 (8) (2009) 1091–1093.
doi:10.1093/bioinformatics/btp101.
URL http://www.ncbi.nlm.nih.gov/pubmed/19237447http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC2666812https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btp101

[93] D. W. Huang, B. T. Sherman, R. A. Lempicki, Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources., Nature protocols 4 (1) (2009) 44–57.

doi:10.1038/nprot.2008.211.
URL http://www.ncbi.nlm.nih.gov/pubmed/19131956

[94] D. Whitley, A genetic algorithm tutorial, Statistics and Computing 4 (2) (1994) 65–85.
doi:10.1007/BF00175354.
URL http://link.springer.com/10.1007/BF00175354

[95] K. R. Patil, I. Rocha, J. Förster, J. Nielsen, Evolutionary programming as a platform for in silico metabolic engineering., BMC bioinformatics 6 (2005) 308.
doi:10.1186/1471-2105-6-308.
URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1327682{&}tool=pmcentrez{&}rendertype=abstract

[96] A. R. Mushegian, E. V. Koonin, A minimal gene set for cellular life derived by comparison of complete bacterial genomes., Proceedings of the National Academy of Sciences of the United States of America 93 (19) (1996) 10268–73.
doi:10.1073/PNAS.93.19.10268.
URL http://www.ncbi.nlm.nih.gov/pubmed/8816789http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC38373

[97] C. A. Hutchison III, Global Transposon Mutagenesis and a Minimal Mycoplasma Genome, Science 286 (5447) (1999) 2165–2169.
doi:10.1126/science.286.5447.2165.
URL http://www.sciencemag.org/content/286/5447/2165.short

[98] A. C. Forster, G. M. Church, Towards synthesis of a minimal cell, Molecular Systems Biology 2 (10) (2006) 1011–21.
doi:10.1038/msb4100090.
URL http://www.ncbi.nlm.nih.gov/pubmed/8849777http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC1369433http://msb.embopress.org/cgi/doi/10.1038/msb4100090

[99] R. Gil, The Minimal Gene-Set Machinery, in: Encyclopedia of Molecular Cell Biology and Molecular Medicine, Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany, 2014, pp. 1–36.
doi:10.1002/3527600906.mcb.20130079.
URL http://doi.wiley.com/10.1002/3527600906.mcb.20130079

[100] R. Gil, F. J. Silva, J. Peretó, A. Moya, Determination of the core of a minimal bacterial gene set., Microbiology and molecular biology reviews : MMBR 68 (3) (2004) 518–37, table of contents.

doi:10.1128/MMBR.68.3.518-537.2004.
URL http://www.ncbi.nlm.nih.gov/pubmed/15353568http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC515251

[101] J. Rees, O. Chalkley, S. Landon, O. Purcell, L. Marucci, C. Grierson, Designing Minimal Genomes Using Whole-Cell Models, bioRxiv (2019) 344564doi:10.1101/344564.
URL https://www.biorxiv.org/content/10.1101/344564v3

[102] R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, L.-S. L. Yeh, UniProt: the Universal Protein knowledgebase, Nucleic Acids Research 32 (90001) (2004) 115D–119.
doi:10.1093/nar/gkh131.
URL http://www.ncbi.nlm.nih.gov/pubmed/14681372http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC308865https://academic.oup.com/nar/article-lookup/doi/10.1093/nar/gkh131

[103] C. M. Fraser, J. D. Gocayne, O. White, M. D. Adams, R. A. Clayton, R. D. Fleischmann, C. J. Bult, A. R. Kerlavage, G. Sutton, J. M. Kelley, R. D. Fritchman, J. F. Weidman, K. V. Small, M. Sandusky, J. Fuhrmann, D. Nguyen, T. R. Utterback, D. M. Saudek, C. A. Phillips, J. M. Merrick, J. F. Tomb, B. A. Dougherty, K. F. Bott, P. C. Hu, T. S. Lucier, S. N. Peterson, H. O. Smith, C. A. Hutchison, J. C. Venter, The minimal gene complement of Mycoplasma genitalium., Science (New York, N.Y.) 270 (5235) (1995) 397–403.
URL http://www.ncbi.nlm.nih.gov/pubmed/7569993

[104] M. Kanehisa, S. Goto, KEGG: kyoto encyclopedia of genes and genomes., Nucleic acids research 28 (1) (2000) 27–30.
doi:10.1093/nar/28.1.27.
URL http://www.ncbi.nlm.nih.gov/pubmed/10592173http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC102409

[105] A. P. Burgard, S. Vaidyaraman, C. D. Maranas, Minimal reaction sets for Escherichia coli metabolism under different growth requirements and uptake environments., Biotechnology progress 17 (5) 791–797.
doi:10.1021/bp0100880.
URL http://www.ncbi.nlm.nih.gov/pubmed/11587566

[106] M. Huynen, Constructing a minimal genome, Trends in Genetics 16 (3) (2000) 116.
doi:10.1016/S0168-9525(99)01972-1.
URL https://www.sciencedirect.com/science/article/pii/S0168952599019721

[107] S. S. Kamat, H. J. Williams, F. M. Raushel, Intermediates in the transformation of phosphonates to phosphate by bacteria, Nature 480 (7378) (2011) 570–573.
doi:10.1038/nature10622.
URL http://www.ncbi.nlm.nih.gov/pubmed/22089136http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3245791http://www.nature.com/articles/nature10622

[108] K. Patil, I. Rocha, J. Förster, J. Nielsen, Evolutionary programming as a platform for in silico metabolic engineering, BMC Bioinformatics 6 (1) (2005) 308.
doi:10.1186/1471-2105-6-308.
URL http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-6-308