School of Physics and Astronomy

Evolution of Metabolic Networks
MPhys Project Report

Balint Borgulya
21th March 2016

Abstract

We consider a process resembling biological evolution of metabolic networks restricted to a small but realistic set of molecules to examine the effect of mutation rate, carbon source and objective function on the patterns of evolution. By evolving the trunk of the glycolytic pathway we find that the rate of mutation only influences the time it takes to improve the networks, while the presence of multiple nutrients and more complex objective functions produces more diverse networks and improvement patterns.

Signature: Date:

Supervisor: Dr. Bartek Waclaw

Contents

Personal statement

I spent the first two weeks reading literature suggested by my supervisor, and getting familiar with C++, as I have not worked with it before. I started developing the simulation software in early October. I originally used Boost Graph Library objects to store the reaction network, this way I could simulate simple trial networks at the end of October. It became apparent then that the functionality provided by Boost comes at a large performance cost, so in the first few weeks of November I restructured the program to use vectors and custom data structure to link reactions and substrates instead of the ones provided by Boost. I later implemented a pre-calculation of the reaction neighbour-list substantially speeding up the calculation. In the last week of November I implemented outputting the reaction networks into the `XGMML` file format that Cytoscape can read for visualisation of the networks.

Over the winter break I attempted creating an algorithm that would execute a depth first search through all reactions in order to find alternative pathways to start the MN from, unfortunately this was very ineffective, so I used a modified version of the trunk of normal glycolysis as the initial network. Over the break I also read articles in the field and I started drafting the report itself.

In January I started running the first simulations with the modified trunk of the glycolytic pathway as the initial network. I also implemented the scripts necessary to perform parallel simulations on multiple computers in the physics cplab, and collecting the results of these simulations. As the simulation results started to flow in, I started to create analysis tools for them, plotting relevant summary statistics. The format of the log file the simulation outputs while running changed and expanded during this time as additional summary statistics were added to it.

In February I changed the algorithm to use index-based cell types in the population population, where instead of 100 metabolic networks I only store 100 references to types of metabolic networks. This was a performance boost for the simulations with a low mutation rate, as the iterations when no mutations happened could be calculated nearly instantly, since the flux balance analysis problem wasn't necessary to solve in these cases.

In February and March I run into problems with the IT infrastructure at the simulating machines, slowing down the progress. I have also had simulations that were started with bad parameters. These errors were fixed when discovered.

I started writing the report in January, and worked on it continuously, however the majority of the work on it was done in March.

The project's GitHub page [5] and the commit history there serves as diary of the project.

Definitions

Metabolite: a chemical compound produced in the metabolic network.

Nutrient: chemical compound a cell consumes in order to provide itself with energy and materials.

Enzyme: biological catalyst molecules, can lower the activation energy of a reaction therefore increasing the flux through that reaction.

ATP - Adenosine triphosphate: the energy currency molecule of cells, has 3 phosphate groups. Energy is released when converting ATP to ADP (removing a phosphate group), and can be stored by converting ADP to ATP (adding a phosphate group).

ADP - Adenosine diphosphate: key molecule in the cell's energy management, has one less phosphate group than ATP.

DNA - Deoxyribonucleic acid: the molecule carrying the genetic information required for reproduction and functionality of most living organisms and some viruses. Most DNA molecules (such as the human DNA) form the well known double helix.

Genome: the complete genetic information required for the construction and functioning of the organism. Most cells store their genome in the form of DNA.

Gene: a proportion of the genome responsible for a certain function of the cell (in our case the ability to perform a certain reaction).

Population: a set of organisms competing and breeding in the same environment.

Generation: Natural unit of time for a population. A generation is the time it takes for N reproductions to take place in a population of N cells (on average every cell reproduces once).

Flux: (through a reaction): the rate — on an arbitrary scale — at which the given reaction converts its reactants to the products. It is usually denoted by v.

Allele: a variant of a certain gene. In our case an allele is the existence (or non-existence) of a certain reaction within the cell's network.

Fitness: a quantity measuring the cell's adaptation to its environment. The higher it is the more chance the cell has to reproduce.

Glycolysis: the metabolic pathway processing glucose into pyruvate and energy (in the form of the net 2 ADP $\rightarrow$ ATP conversions).

Trunk of glycolysis: also known as the pay-off phase of glycolysis: the part of glycolysis that starts with G3P (Glyceraldehyde 3-phosphate) and end in pyruvate. It is called the pay-off phase as this is the part that produces ATP (while in the previous part an investment of ATP must be made).

Stoichiometric coefficient: the number of molecule of the given type participating in a reaction. In the reaction $2\ H_2 + O_2 \rightarrow 2\ H_2O$ the stoichiometric coefficient of H_2 is -2, for O_2 it is -1, and for H_2O it is 2.

1 Introduction

All life on Earth is based on carbon, and it is hypothesized that if we ever find life elsewhere in the universe, it will also be carbon based. Carbon can bond with carbon and other different atoms (especially hydrogen, oxygen and nitrogen) to build complex organic molecules. Cells use such complex organic molecules for a variety of functions such as constructing their membranes, enzymes and their whole structure using these molecules as building blocks. Cells therefore require organic and inorganic molecules that provide energy, and atoms necessary to synthesise these cellular building blocks.

The network of chemical reactions responsible for these processes is called the metabolic network (MN) of the cell. MN-s consist of multiple metabolic pathways, that perform certain function(s) that the organism requires. An example of a metabolic pathway is glycolysis that is the pathway processing glucose, a highly energetic organic compound into molecular building blocks such as pyruvate and energy, stored in the cell's currency metabolite ATP.

Considering the possibilities of chemistry it is easy to see that cells have infinitely many possible ways of metabolising their nutrients to the products they need, if we only impose the constraint that the individual atoms have to be conserved on both sides of reactions. Although different organisms utilize different chemical reactions in their MN-s, their core pathways are very similar among many different organisms. Certain pathways are conserved partially — eg. for processing glucose the Embden-Meyerhof-Parnas pathway [32] is used by most modern cells, some prokaryotes however use the Entner-Doudoroff pathway [12]. An other highly conserved pathway, the citric acid cycle (also known as the Krebs cycle) is used by all aerobic organisms[1]. The reason for these similarities is not yet known, but most authors agree that it is a result of the evolution of the MN-s.

Biological evolution is a series of changes in the genetic material of organisms that is passed down to successive offsprings. The process of evolution uses simple facts: when organisms reproduce they don't always create perfect copies of themselves, resulting in a variation in the population they live in. This variation can cause different organisms to suit their environments better than others, and so those better suited will be able to reproduce more, passing on the genetic material responsible for the adaptations to their offsprings. The less suited organisms will reproduce less, and therefore eventually their genetic traits will disappear from the population. Biological evolution is responsible for the development of organisms from single-cells to multi-cell organisms as well as differentiation to plants and animals, and for the emergence of humans.

In this project we will examine whether the high degree of preservation of MN-s is an inherent feature of their evolution. To do so we will evolve populations of metabolic networks performing a similar role to the trunk of glycolysis using an algorithm resembling biological evolution. We restrict the networks to a simplified yet realistic subset of organic molecules and a toy model of chemistry. We will examine the reproducibility of evolution, i.e. whether the same MN-s and patterns of evolution emerge if the tape of life is played again under different conditions.

[1] Organisms that live in the presence of oxygen

In the rest of this chapter we will explain how MN-s can be modelled mathematically as graphs, how biological evolution can be simulated on a computer, and why this project is of interest to a physicist.

In Chapter 2 we will show the methods used in the simulation with emphasis on the subset of molecules used, the calculation of the free energy change, the implementation of evolution in the code, the fitness-comparison of our organisms, and the data analysis.

In Chapter 3 we summarize the results of the project by show the results of the population simulations under different conditions.

In Chapter 4 we draw conclusions from the results of our simulations and consider them in the big picture of the mathematical understanding of biological evolution.

The Appendices show the source code of the software we developed to perform the simulations.

1.1 Metabolic networks as graphs

Thorough mathematical investigation of MN-s of different organisms reveal remarkable similar topological properties, that resemble the social network of humans and the internet. These are small world character, scale-freeness, error-tolerance and modularity[22].

A small-world network [38] is one in which the number of steps to connect any given pair of nodes — in our case chemical compounds — is small, and this number stays constant even as the network grows. In case of the human social networks this is colloquially known as the six degrees of separation [23]. A small-world character is desirable for a MN as it allows the network to withstand the loss of some of its nutrients. If a certain one is absent, an alternative pathway not much longer than the original can become active, producing the absent nutrient.

In a scale-free network the degree distribution[2] is $P(k) \sim k^{-\gamma}$ for some constant γ, resulting in a few highly connected nodes (hubs) and many scarcely connected nodes. This provides error-tolerance against random errors to the network, as unless a hub is removed, the performance (in our case the fitness of the MN) doesn't decrease significantly. It also makes the network different from a random graph [4], where every node is connected with a constant probability p, resulting in a Poisson distribution for the connectivity ($P(k) \approx \frac{\lambda^k e^{-\lambda}}{k!}$), as well as from a regular lattice, where every node is connected to a constant number of other nodes.

A module is "by definition, a discrete entity whose function is separable from those of other modules" [19]. Within a module, generally there are multiple possible pathways to achieve the same goal using so called precursor molecules. The module first converts its input to precursors, and then these are converted to the final product of the module. Usually the module also has the capability to transform precursors to other precursors, providing redundancy to the pathways. Precursors are useful, as if a module lacks one of its inputs, it is possible that through the conversion of other precursors it can still be functional, even if it at a reduced efficiency. Precursors also help evolution, and adaptation, as if the module can convert a single compound to precursors, a similar compound

[2]The probability that a uniformly chosen node is connected to k other nodes.

can also be easily converted to the same precursor using perhaps a few additional reactions. For example if the cell is capable of processing glucose, using the same enzymatic pathways it is also capable of processing 40 other similar molecules [1].

1.2 Biological Evolution

Evolution works by exploiting the copying errors made in the genetic code of organisms when they reproduce. After such a reproduction the resulting offspring's performance might differ slightly from that of its predecessor at a certain task. In some cases this change is positive, in the sense that it better equips the organism to suit its environment and allows it to multiply faster. In this case it and its consequent offsprings can outcompete the original organism and the unmutated ones. As Charles Darwin wrote "...Natural selection acts only by taking advantage of slight successive variations; she can never take a great and sudden leap, but must advance by short and sure, though slow steps." [7] It is difficult to imagine how highly sophisticated functions such as the eye, or feathers for flight, or even a complex metabolic network could have evolved using slight variations. The authors of [1] argue that complex structures have evolved non-adaptively as exaptations, as byproducts of evolution of other functions. The authors of [25] consider simple digital organisms to examine how complex features may have evolved. They find that sophisticated features appear from mutations that destroy simpler functions, but they provide a high enough evolutionary advantage that they fix in the population quickly via the fist mutant reproducing and spreading the genes responsible for them.

Organisms need a way of passing down their genetic information to their offsprings. Most of them code their genetic material in the form of DNA. This uses a 4 letter alphabet of different base pairs to describe the genetic information necessary for the reproduction and functioning of the organism. When the organism multiplies this information must be duplicated and passed on to the offsprings. A typical bacterial genome consists of $\sim 10^6$ base pairs, and the human genome has 3.4×10^9 base pairs. Even though a multitude of error correcting mechanisms are in place [17], errors are made, for example the genome duplication Escherichia coli has an error rate[3] of $10^{-9} - 10^{-11}$[14]. These errors are most often point mutations that cause a small change in the genetic code of the organism.

The genome of our organisms consist of the list of reactions they are able to perform. This can be considered as a binary string of length 11790, the number of reactions in our 4 carbon network, with 0 for those reactions that do not appear, and 1 for those that do. In our model point mutations correspond to adding or removing a reaction from the MN of the cell.

Apart from random point mutations the genetic information of certain organisms can also change by the process of horizontal gene transfer. This is a method allowing Bacteria and Archaea to exchange genetic material between cells, and it is thought to be the main reason why bacteria can "learn" from each other eg. resistance to antibiotics [18, 31]. We will examine how the presence of horizontal gene transfer influences the evolutionary process.

In real life cells use a variety of mechanisms to change the genetic information of cells

[3]Errors per base pair

such as gene duplication or gene insertion. We will not consider these in our model.

One of the largest problems in the mathematical modelling of biological evolution is the complexity of the fitness landscape. The fitness landscape is the function mapping the set of reactions in the metabolic network to a real number — the fitness of the MN. The true form of this function is not known therefore first toy landscapes, later simplified empirical landscapes were used to test phenomena. Recently the mapping of real fitness landscapes was started which confirm previous assumptions about the simplified models [8]. We will consider a realistic fitness landscape depending on the steady state of flux through our MN-s. The methods for calculating it are discussed in Section 2.4.

1.3 Modelling glycolysis

When modelling metabolic networks one often has to make simplifications both in theoretical and computational research. These simplifications occur in defining the chemistry the organisms can use, the objective function of cells, the environment the they live in, as well as their anatomy.

Modelling the intricacies of chemistry is a hopeless task with today's computers; the fully quantum mechanical treatment of even a few thousand atoms is beyond the capabilities of supercomputers. To overcome this barrier artificial chemistries are often used, that simplify the rules, but grasp some important concept of chemistry[9]. The authors of [20] use linear molecules consisting of 3 possible artificial atoms (called "1", "2", and "3" with the numbers showing how many bonds an atom can make with other atoms) to construct a metabolic network of organisms, and examine how they evolve. It finds a high degree of modularity by calculating the position of synthetic lethal genes[4] of these organisms. The authors of [16] consider "chemical reactions as graph rewriting operations, and uses a toy-version of quantum chemistry to derive thermodynamic parameters".

The thermodynamic constraints on evolution play an important role in the emergence of the MN-s currently used by organisms. The authors of [6] considers an exhaustive search of alternative metabolic networks to the trunk of glycolysis using the same set of compounds and reactions used in this work. It finds that in environments similar to those found in a living cell (in terms of temperature and metabolite concentrations) the real glycolytic pathway provides a higher flux of ATP than every alternative, however if the physiological conditions are changed the real pathway can be outperformed. In this work we consider glycolysis from an evolutionary viewpoint.

Glycolysis is a highly conserved metabolic pathway that plays an important role in the metabolism of organisms in all three domains of life. It processes glucose into energy (in the form of ATP) and pyruvate, a simpler 3 carbon molecule. Glucose is an energy rich compound used by many organisms to store energy for a prolonged time, but it is also used as a source of carbon. Bacteria such as Escherichia coli can use glucose as a source of carbon for the manufacture of the metabolites it needs for its growth (eg. amino acids and nucleotides)[24].

[4]Two or more genes whose simultaneous removal is lethal to the organism, but if removed individually the organism stays viable.

The glycolytic pathway can be divided into two parts: the preparatory phase and the payoff phase. In the preparatory phase glucose is split into two 3 carbon molecules: dihydroxyacetone phosphate (DHAP) and glyceraldehyde 3-phosphate (G3P) with the investment of 2 ATP. DHAP is then isomerized to an other G3P molecule, and G3P is processed in the payoff phase. In this phase G3P is transformed into pyruvate yielding 2 ATP for both G3P molecules therefore giving a net yield of 2 ATP for every glucose processed. The pyruvate output of glycolysis is generally further processed, for example in aerobic organisms it is converted into CO_2 while releasing more energy in the citric acid cycle.

When organisms need to store energy they use the pathway of glyconeogenesis that creates glucose from simpler carbon molecules, using many of the reactions of glycolysis in the opposite direction.

1.4 Evolution in the eyes of a physicist

Statistical physics makes very good predictions about systems where the number of particles is on the order of Avogadro's number, if these systems are in equilibrium or are closed. Living organisms are neither closed systems, nor are they in equilibrium. In order to maintain their non-equilibrium state they must obtain non-equilibrium materials — food or other nutrients — from their environment. If they fail to do so they reach equilibrium, or in other words die [29]. Therefore to consider living organisms we need the tools of non-equilibrium statistical physics.

Out of equilibrium systems are an actively researched area in statistical physics, and numerous concepts of it can be applied to biology. The authors of [11] examine self replication — a process all living organisms must perform — and derive a lower bound on the heat produced when a single bacterium replicates while being coupled to a thermal bath.

One of the first people to quantitatively examine evolutionary processes, R. A. Fisher wrote that [the theorem of natural selection] "bears some remarkable resemblances to the second law of thermodynamics." [15]. Both systems of statistical physics and evolving populations consist of a large number of particles (organisms) following relatively simple rules, based on probabilities. Due to the probabilistic nature of these systems exact solutions are very difficult to reach if even possible, so one can only predict the large scale behaviour of the whole system. Stochastic methods originally developed in statistical physics play a crucial role in modelling the evolution of populations [3]. In this work we will use such a method, the Moran process [28] to simulate the evolution of populations, but an other example for such model is the Wright-Fisher model [13].

Our model resembling biological evolution corresponds to a random walk in the space of available MN-s, that tries to optimize the fitness of our networks. The fitness landscape of even simple organisms such as ours is a very complicated function with many local minima and it is not known if the global optimum is accessible from every point in the space of all MN-s while transitioning through viable MN-s. The authors of [2] considered MN-s over a large set of chemical reactions and found that in all but the simplest cases the global optimum could be reached. Stochastic methods such as our model of evolution

are successful in leaving local minima, however can also be prone to oscillations around the global minimum

The process of biological evolution is an optimization process with inherently random elements. As such it is well suited to traverse a rugged fitness landscape in search of a global optimum, however unlike most stochastic optimization algorithms it can not easily traverse large obstacles in the fitness landscape. This is simply because if an organism becomes considerably less fit than its peers it will be quickly eliminated by the survival of fittest phenomena.

Our work uses the formula well known from statistical physics to calculate the Gibbs free energy for our reactions in specific thermodynamic conditions. This formula uses the Gibbs free energy of reactions in standard conditions, which were obtained from experimental results where available, and estimated [6] where not. The fully quantum mechanical calculation of the free energy changes of reactions is feasible and could lead to more accurate answers using the same evolutionary methods described in this paper. Such a calculation of thousands of Gibbs free energies would be a job worthy of its own project, therefore we do not pursue it in this work.

2 Methods

2.1 Anatomy of the organisms simulated

A set of compounds and reactions (as used by [6]) was provided for the project by Bartlomiej Waclaw the project supervisor. This is the set of chemical compounds and reactions our organisms are allowed to use. The list of compounds consisted of 1966 CHOPN molecules of at most 4 carbon atoms, along with 13 so called internal metabolites. CHOPN molecules contain only carbon, hydrogen, oxygen, phosphor and nitrogen. Internal metabolites are compounds that are not CHOPN molecules fitting the previous criteria, but are essential cofactors to the reactions in the MN-s. The concentrations of the most important internal metabolites is listed in Table 3. In Table 1 we show an extract of the list of compounds.

There are 11790 reactions of the at most 4 carbon compounds in the database used along with the ID of the compounds and the products and the free energy change of the reaction at the conditions described in [6]. An extract of the list of reactions is shown in Table 2. The free energy changes are later calculated for arbitrary conditions as discussed in Section 2.2.

Our program simulates the evolution of MN-s restricted to the metabolites and reactions provided. This is a realistic constraint as the real trunk of the glycolytic pathway uses only such molecules.

In our model we assume that the direction of a reaction is determined only by its free energy change as defined under the physiological circumstances of the simulation. It is also assumed that given two reactions, both with free energy change larger (or smaller) than some pre-defined limit l the maximal flux through both reactions is the same. This is not necessarily the case in real cells. Apart from the free energy difference, the speed

Compound ID	Free energy of formation ΔG_0 (kJ/mol)	Chemical formula	Name	Charge
-7	-2292.39	ATP	—	-4
-6	-1424.91	ADP	—	-2
...				
-1	-155.758	H_2O	water	0
0	62.6568	CH_3-CH_2(OH)	ethanol	0
1	-247.929	CH_2-COOH	acetate	-1
2	23.8646	CH_2-CHO	acetaldehyde	0
3	-833.15	CH_2-CH_2p	—	-2
4	-1107.16	CH_2-COp	acetylphosphate	-2
5	10.21	CH_2-CO(NH_2)	—	-2
...				

Table 1: Extract of the data file containing the compounds used

Reaction ID	ΔG_0 (kJ/mol)	Compounds	Products
2	-28.3268	0,-7	3,-6
4	8.24999	1,-7	4,-6
5	-16.561	1,-7,-10	5,-6,-8
10	-7.83516	3,-1	0,-8
14	-41.6634	6	2,-1
18	-4.66344	6	38,-1
22	42.0077	7,-3	14,-4
25	257.103	7	50,-1
31	-25.83	8,-7	20,-6
...			

Table 2: Extract of the data file containing the reactions

and direction of the reaction can be influenced by the free energy landscape between the initial and final states, and also by the cell via the use of enzymes.

The anatomy of our cells is as follows: the cells have internal metabolites present inside their membranes with concentrations as shown in Table 3. The cells can import and export H_2O and CO_2 to and from their environments as these compounds can pass through the cell membrane. They can also import one predefined nutrient to process, and they can dispose one target molecule. Later this assumption will be relaxed, allowing the network to use multiple nutrients, and produce multiple products. The products of the network are only disposed of from the viewpoint of the metabolic pathway. They are often further processed by subsequent pathways, such as pyruvate in Section 1.3. The nutrient of the cell is available in excess in order to ensure that the fitness of the MN is limited by the reactions within the MN rather then the nutrient availability.

In certain simulations we allow the MN to dispose of phosphate (Pi); in real MN-s this could be used by other processes within the cell, such as the upper part of glycolysis. The network can also exchange the reduced form of nicotinamide adenine dinucleotide (NADH) to its oxidized form NAD^+. The oxidization of NADH occurs both in aerobic organisms (through oxidization via oxygen) and in anaerobic[5] ones. In metabolic networks NAD^+ and NADH play an important role in redox reactions by carrying electrons between reactions[24].

The success of a cell within a population is measured by its reproduction rate. If it reproduces often, it ensures its genetic material will be carried on by its offsprings. This means that if via a mutation a cell obtains a reproductive advantage, this allele can spread within the population due to the increased reproduction rate. Reproduction is a very energy demanding process, so we measure the fitness by the cell's energy output. ATP acts as the energy currency in cells, and the ADP $\rightarrow$ATP conversion rate is considered to be proportional to the growth rate of the cells. We therefore use the flux through an auxiliary reaction[6] performing this reaction. Later we consider more complex objective functions that reward biomass production, which is also a requirement for reproduction.

When organisms are considered as part of a population, the fitness of any individual depends on the fitness of others within the population. In a population where all but one cell has a fitness of 0.1 (on an arbitrary scale) and a single cell has fitness 0.5, this cell is obviously the fittest, and will reproduce more likely than the others. The same cell with fitness 0.5 in a population where the rest of cells have fitness 10, is much less likely to reproduce. Each of our cells has probability to reproduce, that is proportional to their current fitness value, normalized with the cumulative fitness of the population. We do not consider cells with negative fitness, considered unviable for reproduction. At each reproductive step in the algorithm the cell to reproduce is drawn from a probability density function that is proportional to the fitness of each cell (up to normalization), using the Moran process described in Section 2.3.

We consider the size of the population to be constant, fixed by the carrying capacity of the environment. This means that every time a cell reproduces, a cell must die to accom-

[5]Organisms that do not require oxygen. They can oxidize via alternative compounds, including iron and sulfate.

[6]Defined in Section 2.4

Temperature	293 K
[ATP]	$10^{-1}M$
[ADP]	$10^{-2}M$
[AMP]	$10^{-4}M$
[NAD^+]	$10^{-1}M$
[NADH]	$10^{-4}M$
[Pi]	$10^{-3}M$
[PPi]	$10^{-3}M$
[CO_2]	$10^{-5}M$
[NH_3]	$10^{-5}M$

Table 3: The environmental variables considered

modate the newborn one. The deceasing cell is chosen uniformly from the population, therefore giving the expected lifetime of a cell to be 1 generation.

The cells in a population are considered to be close to each other, so any two cells can meet and exchange genetic information in the process of horizontal gene transfer.

2.2 Calculating the free energy change

The Gibbs free energy is the amount of energy at a given reaction that can be used to do work, or if it is negative, then the amount of work that has to be done on the system for the reaction to happen.

The list of reactions provided by the project supervisor contained the following for each reaction: the list of compounds the reaction uses, the list of products the reaction produces and the free energy change of the reaction determined experimentally when known, otherwise estimated.

The free energy changes provided were at the conditions $T = 25^oC$ pH= 7.0 $I = 0.2M$ (ionic strength) and all metabolite concentrations set to 1 M calculated by [6].

Assuming a reaction of the form $n_1A + n_2B \leftrightarrows n_3C + n_4D$ where the concentration of substrate A is denoted by $[A]$, and similarly for the other substrates, n_1 denotes the number of molecules of type A that take part in the reaction and ΔG^0 the free energy change of this reaction at standard conditions, the change in free energy for arbitrary T and concentrations is given by:

$$\Delta G = \Delta G^0 + RT \ln \frac{[C]^{n_3}[D]^{n_4}}{[A]^{n_1}[B]^{n_2}} \tag{1}$$

At the beginning of the program the reactions are read in from a file (Appendix A line 113), and the free energy changes are re-calculated (line 143) for the specified conditions as in Table 3. The internal metabolites that are not shown in the table do not appear in the reactions used by our MN-s. Their concentrations have been set to 1.

The constraints imposed on the flux of each reaction, depending on the free energy change of them are as follows: for the m^{th} reaction the flux v_m has to satisfy $0 \leq v_m \leq 1$

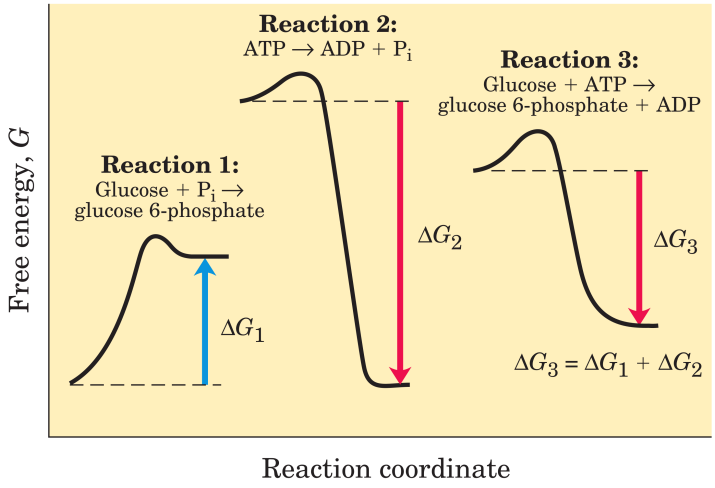

Figure 1: Free Energy landscapes of three reactions, showing the different flux-constraints imposed in our model. Figure reproduced from [24]

if $\Delta G \leq -L$ eV and $-1 \leq v_m \leq 0$ if $\Delta G \geq L$ eV. As in the real world the reactions happening in a cell are usually reversible. An example for this is is the reaction of carbonic acid and water $H_2CO_{3(l)} + H_2O_{(l)} \leftrightarrows HCO^-_{3\,(aq)} + H_3O^+_{(aq)}$. We allow reactions with free energy change between a constant $-L$ and L eV to happen in both directions. We use the value of $L = 10$ eV but it is a parameter easily changed in the program. In Figure 1 we show the three possible cases Assuming $\Delta G_1 > L$ Reaction 1 would have its flux restricted as $-1 \leq v_1 \leq 0$, also if $\Delta G_2 < -L$ v_2 is restricted as $0 \leq v_2 \leq 1$, and if $|\Delta G_3| \leq L$ we have $-0.5 \leq v_3 \leq 0.5$.

In real life the likelihood of these reactions happening, and so the flux through them depends on the height of the potential barrier between the original substrates and the end-products. Such potential barriers are shown on Figure 1. If the barrier is high the reaction might not happen unless the amount of energy required to overcome it is invested. Cells are able to lower the heights of such barriers using catalytic enzymes, therefore our model is not unrealistic by neglecting their role in determining the directions and fluxes of the reactions.

2.3 In silico Evolution

To simulate biological evolution on a computer we need three things: replication, mutation and competition. The sophisticated interplay of random mutations and the more deterministic survival of fittest phenomenon give rise to the success of evolution. The basics of these concepts can easily be implemented in our program, giving a process similar to biological evolution. Our simulations always start with a homogeneous population, with 100 identical cells. The population later changes as a result of random mutations, but stays largely homogeneous for small mutation rates, as shown in Section 3. The real evolution of the genomes of organisms uses a variety of methods some of which are discussed in Section 1.2. Of these, our cells use point mutations and later horizontal gene transfer.

Point mutations either add or delete a reaction to the MN. Simply adding a reaction to those available to the cell at random from the list of all the reactions is unlikely to result in a reaction that could be used by the cell. This is because with a high probability the added reaction would be disconnected from the current MN. As we show in Section 2.4 the flux through these reactions would be zero in the flux balance analysis problem. Therefore when adding reactions, we only consider those that are linked to one of the current reactions, either by using a compound that the network is capable of producing, or providing one that it can consume. This way the graph of the reactions and compounds available to the cell stays a connected graph (disregarding deletions). Choosing the list of reactions that can be added uses the neighbour-list defined in our algorithm in Appendix F line 21. When deleting a reaction every currently available reaction has equal probability of being deleted. This can and does result in disconnected graphs, but such disconnected parts are usually small, otherwise they bear too high a cost to maintain and are therefore an evolutionary disadvantage.

The most elementary event in a population of our organisms is the reproduction of a cell, implemented using the Moran process [28] At every step of the Moran process a single member of the population is chosen to reproduce (with or without mutations) and an other member of the population dies, to keep the size of the population constant. At every such step the cell has some probability of mutating, both for a point mutation (p_{point}), and later for horizontal gene transferring a reaction (p_{HGT}) from an other randomly chosen cell. This process is a random walk in the space of all possible genes or MN-s allowed by the chemistry of the cells. This walk is then mapped to the fitness landscape using the objective function of the cells.

There are two contributions to the fitness measure of a cell, its energy production, and the number of reactions in its MN. The first objective function we used awarded a fitness score of 1 for every ADP $\rightarrow$ ATP executed by the cell. For every reaction in the MN, the cell pays a small cost k_{reac} from the fitness score. This is because cells usually control their reactions through the use of enzymes that they have to manufacture, and code in their genetic code. Should we not include this cost the cell's MN-s could grow without bounds until every reaction in our list would appear within them. This complete network could then be considered a best network. Later we consider more complex objective functions that reward biomass production, at different weights compared to the energy production.

The probability density function (PDF) of which cell is selected for reproduction is proportional to the fitness of each cell, with the caveat that cells with negative fitness are deemed unviable and are not allowed to reproduce. To know this PDF it is necessary to know the fitness of each cell at any given time point (for the normalization), and therefore it would be impractical to use this to choose the cell that will reproduce. Instead we sample as follows:

1. We uniformly chose a cell c_i in our population.
2. We generate a random number from the distribution $U(r, M)$ where M is the maximal fitness value a cell can take.
3. If the fitness of c_i, $F_{c_i} \geq r$ we accept the change, if not we start the process again.

The code section performing this task is shown in Appendix C lines 213-219. This process can be shown to result in a sampling of cells equivalent to that of the PDF proportional to the fitness of each cell.

2.4 Flux Balance analysis

To calculate the throughput of the metabolic network of our organisms we use flux balance analysis [30].

The goal of flux balance analysis is to calculate the steady state fluxes ($\mathbf{v}$) through all the reactions, subject to some constraints, while maximizing a linear function $Z = \mathbf{c}^T\mathbf{v}$ of the fluxes called objective function, in the solution space allowed by the constraints of the network.

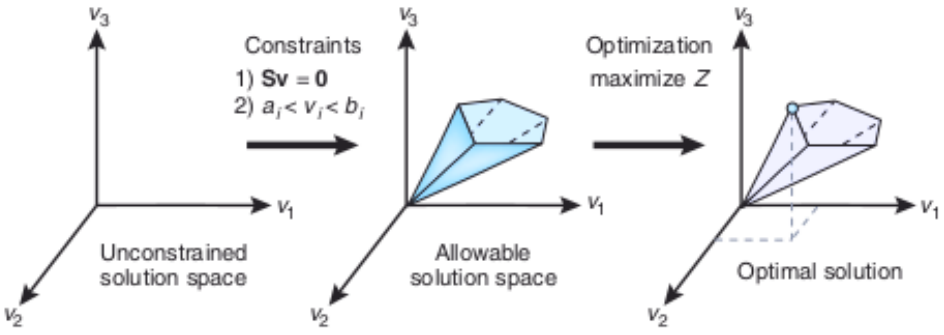

Figure 2: The process of flux balance analysis. First we apply the steady state constraints along with further constraints on the fluxes, then we maximize the objective function in the allowable solution space found. Figure reproduced from [30]

When using flux balance analysis we represent the MN as a stoichiometric matrix S ($m \times n$), with each row corresponding to a compound in the network (n compounds), and each column corresponding to a reaction (m reactions). In each column we have the stoichiometric coefficients of the given reaction that correspond to the compound of the given row. Negative stoichiometric coefficients are used for the metabolites consumed, positive for those produced, and zeros for the metabolites that do not participate in the given reaction.

We want to find a set of steady state fluxes $\mathbf{v} = (v_1, v_2, ..., v_m)$ that obey the principle of mass conservations. If we represent the concentrations of our compounds by $\mathbf{x} = (x_1, ..., x_n)$ this condition is expressed as $\frac{d\mathbf{x}}{dt} = 0$, no compound is depleted or accrued, the rate a metabolite is consumed must equal the rate it is produced. Using the stoichiometric matrix the steady state condition can be expressed as $\frac{d\mathbf{x}}{dt} = S\mathbf{v} = \mathbf{0}$ (the allowable solution space in Figure 2). These equations take the form

$$\begin{aligned} S_{1,1}v_1 + S_{1,2}v_2 + ... + S_{1,m}v_m &= 0 \\ S_{2,1}v_1 + S_{2,2}v_2 + ... + S_{2,m}v_m &= 0 \\ &... \\ S_{n,1}v_1 + S_{n,2}v_2 + ... + S_{n,m}v_m &= 0 \end{aligned} \tag{2}$$

The network must however exchange some special compounds with its environment such as the nutrients and the end-products. For example the nutrients of our network on Figure 3 is CO=CH−CH(OH)−COp and water, while its end-products are lactate and CO_2. Incorporating the exchange of certain compounds with the environment of the MN could be done through relaxing the constraint that the equation corresponding to the given compound among those in Equation 2 have to equal zero. This introduces complications, as often the amount of these compounds produced appears in the objective function $Z = \mathbf{c}^T\mathbf{v}$. To overcome this we introduce auxiliary reactions to the stoichiometric matrix S, for the compounds the cell consumes and produces. Such an auxiliary reaction is for example " $\rightarrow H_2O$", that "creates water out of nothing", or in other words corresponds to the potential water intake of the cell. These auxiliary reactions can be used to provide nutrients to the cell and remove its end-products.

We want to maximize the objective function $Z(\mathbf{v})$ in the allowable solution space. The form of the objective function is $Z = \mathbf{c}^T\mathbf{v}$ with the elements of $\mathbf{c}$ being the weights of the fluxes through each reaction in the objective function. In the simplest cases $c_i = 0$ for all but a single value of i, in which case the goal is to maximize the flux through a single reaction, which in the literature is often chosen to be the biomass removing auxiliary equation [30]. In our case the original objective function used is the one converting ATP to ADP. This reaction occurs in cells, (ATP $+\mathrm{H_2O} \rightarrow$ ADP + P_i), and it releases energy. As cells store energy in ATP molecules, by maximizing the flux through the ATP producing reaction, we are maximizing the energy output of the cell. In many cases this is synonymous with maximizing the growth rate of the cell. The auxiliary reaction corresponding to the ATP $\rightarrow ADP$ conversion does not contain the release of phosphate in this test network. This is because the nutrient of the network contains phosphate while the end-product does not. In such cases the phosphate produced by the ATP hydrolysis would have to be removed by an other auxiliary reaction. To make the program a bit more efficient we do not consider this additional reaction now, however in later simulations we will.

There are many software packages capable of solving linear programming problems, we use the Gnu Linear Programming Kit (GLPK) [26]. GLPK is implemented in C, and has all the functionality we need to calculate the fitness of our metabolic network. For the code calculating the fitnesses see the function calcThroughput in Appendix C lines $321 - 476$.

Flux balance analysis can be used to incorporate a wide range of other concepts, such as non-linear objective functions, regulatory and energy balance analysis constraints [33]. Our choice of implementation limits us to use only linear objective functions; this issue is addressed in Section 4.

Example of flux balance analysis

Here we give an example of how flux balance analysis is used to calculate the fitness of a MN. We consider a toy network, that has an easily verifiable solution. The network is shown on Figure 3, it is a simple network that was used to test the part of the program calculating the optimal fluxes.. For the purpose of this example let us consider the cost

of a reaction is $k_{reac} = 0.01$.

The metabolites of the network (in green) have been denoted by letters A, B and D for brevity. The stoichiometric matrix of this metabolic network is shown at Equation 3 (below), with the metabolites the rows correspond to are shown on the left of the matrix. The first three columns of the matrix correspond to reactions 5151, 5133 and 131 respectively. The other columns correspond to auxiliary reactions, providing the nutrient of the network (A), removing the end-product (lactate), providing water, providing CO_2, and converting ATP to ADP in this order. Auxiliary reactions are not shown on the figure of the MN-s.

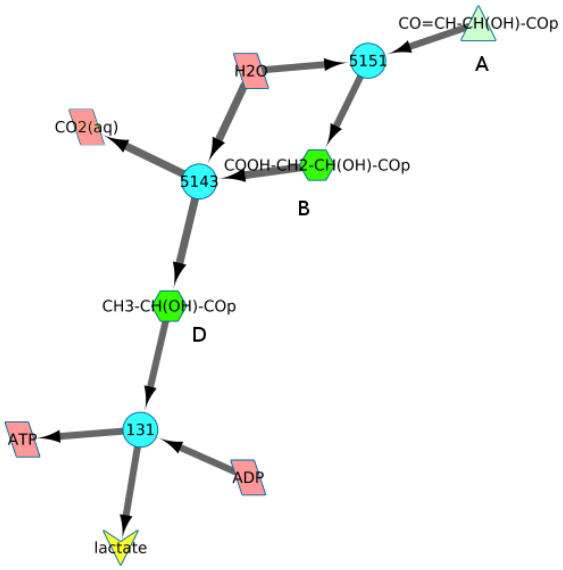

Figure 3: Example of a simple network. Reactions: light blue, metabolites: green, nutrient: light green, end-product: yellow, internal metabolites: red

$$\begin{matrix} \texttt{A} \\ \texttt{B} \\ \texttt{D} \\ \texttt{lactate} \\ \texttt{water} \\ \texttt{CO}_2 \\ \texttt{ATP} \\ \texttt{ADP} \end{matrix}, S = \begin{pmatrix} -1 & 0 & 0 & +1 & 0 & 0 & 0 & 0 \\ +1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & +1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 & -1 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & +1 & 0 & 0 \\ +1 & +1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & +1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & +1 \end{pmatrix} \tag{3}$$

The linear equations implied by $S\mathbf{v} = \mathbf{0}$ for the example network of Figure 3 are shown below, at Equation 4. Constraints on the system are imposed in two ways. Firstly by $S\mathbf{v} = 0$ we require that the fluxes represent the steady state solution, and secondly by restricting the values of v_i by inequalities ($v_{lower,i} \leq v_i \leq v_{higher,i}$) we can bound the flux of individual reactions, their directions, and the amount of compounds the cell can take up and dispose of.

$$\begin{aligned}
\texttt{A} &\quad -v_1 + v_4 = 0 \\
\texttt{B} &\quad v_1 - v_2 = 0 \\
\texttt{D} &\quad v_2 - v_3 = 0 \\
\texttt{lactate} &\quad v_3 - v_5 = 0 \\
\texttt{water} &\quad -v_1 - v_2 + v_6 = 0 \\
\texttt{CO}_2 &\quad v_1 + v_2 - v_7 = 0 \\
\texttt{ATP} &\quad v_2 - v_8 = 0 \\
\texttt{ADP} &\quad -v_2 + v_8 = 0
\end{aligned} \tag{4}$$

The bounds on the given reactions are as follows:

$$\begin{aligned}
0 \leq v_1 \leq 1 \\
0 \leq v_2 \leq 1 \\
0 \leq v_3 \leq 1 \\
0 \leq v_4 \leq 10 \\
0 \leq v_5 \leq \infty \\
0 \leq v_6 \leq 40 \\
0 \leq v_7 \leq 40 \\
0 \leq v_8 \leq \infty
\end{aligned} \tag{5}$$

The limiting fluxes should in any case be those of the real reactions (in our case v_1, v_2, and v_3. The cell's nutrient intake (v_4) is set to a high but finite value, this is important as a this puts a high bound on the fitness function. Such a theoretical upper bound for the fitness function must be known, as it is used in the algorithm when we select the next cell that reproduces. The cell's water intake and CO_2 disposal are limited by a high value for the same reason. Care is taken in order to keep these values high enough not to be limiting for the evolution of the network. The pyruvate removal and the ATP $\rightarrow$ ADP conversion reactions need not be bounded, as they are limited by the nutrient uptake of the cell.

The weights of the objective function in this case are $c_8 = 1$ and $c_i = 0$ for $i \neq 8$, giving the value of the objective function to be $Z = v_8$, the energy output of the cell. This is to be maximized within the constraints.

The optimal solution of the equations with the bounds given at Equation 5 is

$$v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1, v_5 = 1, v_6 = 2, v_7 = 2, v_8 = 1 \tag{6}$$

This gives the value of the objective function to be $Z = v_8 = 1$. As the MN has 3 reactions the final fitness value for it would be $Z - 3 \times k_{reac} = 1 - 3 \times 0.01 = 0.97$

2.5 Data analysis

To visualize the resulting MN-s we use the open source biological network visualization tool Cytoscape [34]. Figures showing metabolic networks in this work have been generated using this software (eg. Figure 3). Our program was equipped with the functionality to export MN-s into the `XGMML` format that Cytoscape can read (Appendix C lines $56-152$).

While MN-s can be visually compared using Cytoscape, doing so for more than a handful of networks is a cumbersome task, therefore we considered some summary statistics to compare MN-s resulting from our simulations.

While the simulation is running, approximately every 10000 mutations some key descriptors of the populations are written out to a log file. This allows us to monitor the progress of the populations while the simulation is running, and also to plot the timeline of their evolution. Examples of such graphs can be found in Section 3 (eg. Figure 5). The descriptors calculated are: the fitness of the fittest network in the population, the average fitness of the population, the entropy of the population, the number of reactions in the fittest network, the average number of reactions in the population, the number of used reactions[7] in the best MN, and the average number of such reactions per MN in the population.

Entropy is used as one of the measures of the diversity of the population. It is calculated as: $S = -\sum_{i=1}^{M} n_i \log n_i$ where n_i denotes the number of cells of type i, and the sum runs through all the different types of cells in the population. This entropy measure is linearly proportional that used in statistical physics: $S_{statphys} = \sum_i p_i \log p_i$, with p_i being the probability of a given state. As we are considering a population with $N = 100$ cells, the minimal entropy is $S_{min} = -100 \log 100 = -460.52$, this value is attained when every cell in the population is identical ($p_{i_1} = 100$, and $p_i = 0$ for other i values), as is the case initially in our populations. its maximal value is $S_{max} = 0$ attained when every cell is different in the population. This happens when the mutation rate is very high (eg. $p_{point} = 1$).

To compare networks between populations we define two similarity indices $M_{i,j}$ between two MN-s i and j, calculated as follows: we count the reactions appearing in both MN-s, then divide this count by the number of reactions in the larger of the two networks. This means $0 \leq M_{i,j} \leq 1$, with these extreme values attained if the two MN-s share no reactions, and if they are identical, respectively. We calculate two similarity indices for each pair of MN-s, one for every reaction in the two MN-s ($M_{i,j,all}$) and one only for those reactions that have non-zero flux through them in the optimal solution ($M_{i,j,used}$). This is an important distinction, as for a high mutation rate, and small k_{reac} value the two MN-s might have a lot of unused reactions that differ amongst them, but their used reactions could be the same. As the indices are symmetric ($M_{i,j,all} = M_{j,i,all}$ and $M_{i,j,used} = M_{j,i,used}$) instead of two separate plots, we plot

$$M_{i,j} = \begin{cases} M_{i,j,all} & i \leq j \\ M_{i,j,used} & \text{otherwise} \end{cases}$$

Thus we show the similarity indices calculated using all reactions above the diagonal of the plot, and those calculated using the used reactions below the diagonal. The diagonal entries are always 1-s as any network is equivalent to itself. We plot the two regions separately to make distinguishing between the two parts easy.

While the simulations are running, we save the states of the populations at 10 occasions, both to be used as checkpoints if the calculation doesn't finish, and also to be able to

[7]Reactions with non-zero flux in the optimal solution to the flux balance analysis problem

look at how the networks evolved. At these checkpoints the MN of every cell in the population are written out (in the `XGMML` format). This allows both the visualization of the networks and also their comparison using the similarity indices between them. As plotting the similarity indices as a matrix (as on Figure 10) for every population at every checkpoint would be impractical, we instead consider the similarity indices of MN-s within a population compared to the fittest network in the population, resulting in a vector of $N-1$ similarity indices (we don't compare the fittest network with itself). This would correspond to the first column (or row) of each population on the similarity matrix plot, for example Figure 10.

These similarity indices are then summarized using their Inverse Participation Ratio. The similarity indices of each population are divided into 10 bins, and the IPR of the proportion of elements in the bin are calculated, using $IPR = \sum_{i=1}^{10} \frac{1}{p_i^2}$ where p_i is the proportion of similarity indices in bin i (this ensures normalization, $\sum_{i=1}^{10} p_i = 1$). The minimum of this IPR value is 1, attained for a completely homogeneous population, while the maximum of 10 is attained when each of the 10 bins have an equal amount of similarity indices within them (the most disordered state). At each checkpoint the IPR of the two similarity indices $M_{i,j,all}$ and $M_{i,j,used}$ are calculated along with the IPR of the fitness values at the checkpoint. As we determine the size of the bins empirically, by dividing the distance between the smallest and the largest values into 10 equal parts, without specifying a minimal bin size, we sometimes get high IPR values for populations that could otherwise be considered rather homogeneous. For example the population of 10 cells with fitness values 0.5, 0.501, ... 0.51 would score an IPR value of 10 by this method, whereas these differences are most often caused by the networks having a different amount of reactions in them. In retrospect this problem could have been circumvented by using a minimal bin size of $\sim 10k_{reac}$.

2.6 Technical remarks

Whenever we add a reaction to a MN we have to find the reactions that connect to the current network, through at least one shared metabolite, as described in Section 2.3. It is very time consuming to list all metabolites currently in the network, then search through all the reactions to see if any of the used metabolites are used within them. To speed up the process of selection, at the beginning of the program a neighbour-list is generated that contains for every reaction a list of neighbouring reactions, that share at least 1 compound with it, excluding internal metabolites. For example in our example network on Figure 3 reactions 131 and 5143 are neighbouring, as they share the metabolite CH_3-CH(OH)-COP, but reactions 131 and 5151 are not neighbouring. Using this neighbour-list our program can now rapidly find the possible reactions to add to a network.

An important component of a Monte Carlo simulation is the source of pseudorandom numbers. In our program we use the Mersenne Twister generator [27], provided by the Boost C++ libraries [35]. This generator is fast, provides high-quality pseudorandom numbers and has a period of $2^{19937} - 1$, and will therefore provide more than enough pseudorandom numbers for our simulations.

To speed up calculations we performed multiple parallel simulations of populations on

different computers. In order for every population to be different, we used different initial values to seed the pseudorandom number generator. For the sake of reproducibility these seeds are noted with the output of the simulations.

In a population usually there are multiple "copies" of any given cell, as the rate of mutations is low, eg. for $p_{point} = 0.1$ only every tenth reproduction results in a new type of cell. To make our program more efficient we consider the population as a list of cell types $\{t_i\}$, $i = 1...N$, and a list of cells $\{c_j\}$, $j = 1...N$. The cells store integers referencing to the given type, $c_5 = 2, c_2 = 1$ means that the fifth cell is of type 2, and the second one is type 1. If cell 5 dies and cell 2 is chosen to reproduce without mutation, this can be expressed as $c_5 \leftarrow c_2$ making $c_5 = c_2 = 1$. The most time consuming operation in our program is solving the linear programming problem that needs to be calculated whenever a new cell is created. By using this index-based population we have no need to solve it when reproduction happens without mutation. For lower mutation rates such as $p_{point} = 0.01$ this means a significant performance increase.

3 Results

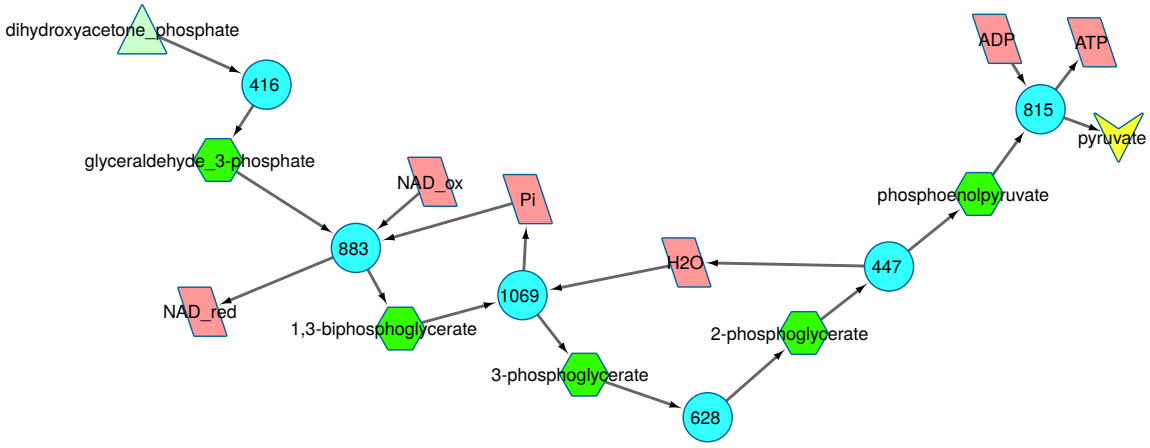

Figure 4: Trunk of the glycolytic pathway with the additional reaction at the beginning

The first simulation we run used a modified version of the trunk of glycolysis as a starting network. This modified network starts from DHAP, one of the products of the upper part of glycolysis, and ends in pyruvate. The starting network is shown on Figure 4. Reaction 416 was added to the usual lower glycolytic pathway at the beginning of the MN. This reaction transforms DHAP into G3P the compound normally considered the first metabolite of lower glycolysis. Apart from this the ATP producing reaction 633 was removed, in exchange for 1069, that fulfills a similar role to 663, but instead of creating ATP from ADP it discards the phosphate group it sheds from 1,3 biphosphorglycerate.

The fitness cost of having a reaction within the MN was set to $k_{reac} = 0.001$ with the rate of point mutations as $p_{point} = 1$ in order to have as much change and improvements as possible. The fitness function used here was the flux of ADP to ATP conversion.

The initial ATP flux of this network is 0.5, with the limiting reaction being 883, that has a free energy change of 1.71 eV at the conditions given by Table 3. This is a good initial network as it has plenty of potential to improve, eg. by "rediscovering" reaction 633, or finding completely new paths from the nutrient of the MN to the end-product. For the sake of brevity we shall refer to this simulation as Simulation 1.

When running the simulations, we started with 20 identical, uniform populations with 100 cells in each, and provided a different seed for the pseudorandom-number generators used, therefore the populations evolved independently from one another. Not all the 20 populations finished the simulation due to technical difficulties. Of those that did, we show diagnostic plots for 3 typical populations that showed some improvement during the simulation.

The fitness of the best network within 3 typical populations of Simulation 1 and the average fitness of those populations are shown as a function of the number of generations since the initial network on the first two graphs of Figure 5. We normally save the summary statistics approximately every 10000 mutations, in this case 100 generations, but keep the values of these summary statistics for the intermittent generations in-memory. If there was a significant improvement in the fitness of the population we write the intermittent steps out too, so the transition of the population can be observed.

An extract of such intermittent steps is shown on Figure 6 from Simulation 2, that has a lower mutation rate than Simulation 1. The number of generations it takes for a cell with relative fitness advantage s to take over a population is calculated as $T \approx \frac{s+2}{s} \ln N$ [37]. This is called the expected time of fixation. In the population shown on Figure 6 the original population has a fitness of 0.48 and the fitness of the mutant is 0.98, therefore the fitness advantage is $s \approx 2$, giving us the time for fixation as $T \approx \frac{2+2}{2} \ln 100 \approx 9.21$ generations. This coincides with the observed time of fixation, the time it takes for the average fitness of the population to "catch up" with the fitness of the best cell. We plot a population from Simulation 2 rather than Simulation 1, as the calculation of the aforementioned formula uses the assumption that mutations are rare. This assumption is better satisfied in Simulation 2 where the expected number of mutations per generation is 1, as opposed to 100 in Simulation 1.

As such fixations remain invisible on a plot whose x axis is on the order of $10^5 - 10^6$ such as Figure 5, the first and second pair of graphs are very similar. In consequent diagrams we shall show only the graphs for the average of the population (average of fitness, total number of reactions, and number of used reactions).

All populations in Simulation 1 have improved their fitness by increasing the ADP ATP conversion rate by at least 0.5 some of them by 1. The magnitudes of any single improvement was 0.5 — this means every pathway uses at least one reaction that is considered to be bidirectional in our model. The times of improvements are distributed approximately uniformly and after any jump-improvement in all curves we can observe a slight decline in fitness. This phenomena is due to the MN-s obtaining more and more reactions of which few are only used, as can be seen on graphs 3 and 4 of Figure 5. Observe that the number of reactions in the MN-s start to increase when the fitness does. The number of used reactions are shown on the last pair of graphs on Figure 5, for the best network and for the average network. The reaction network with ATP production of 0.5 uses ~ 6

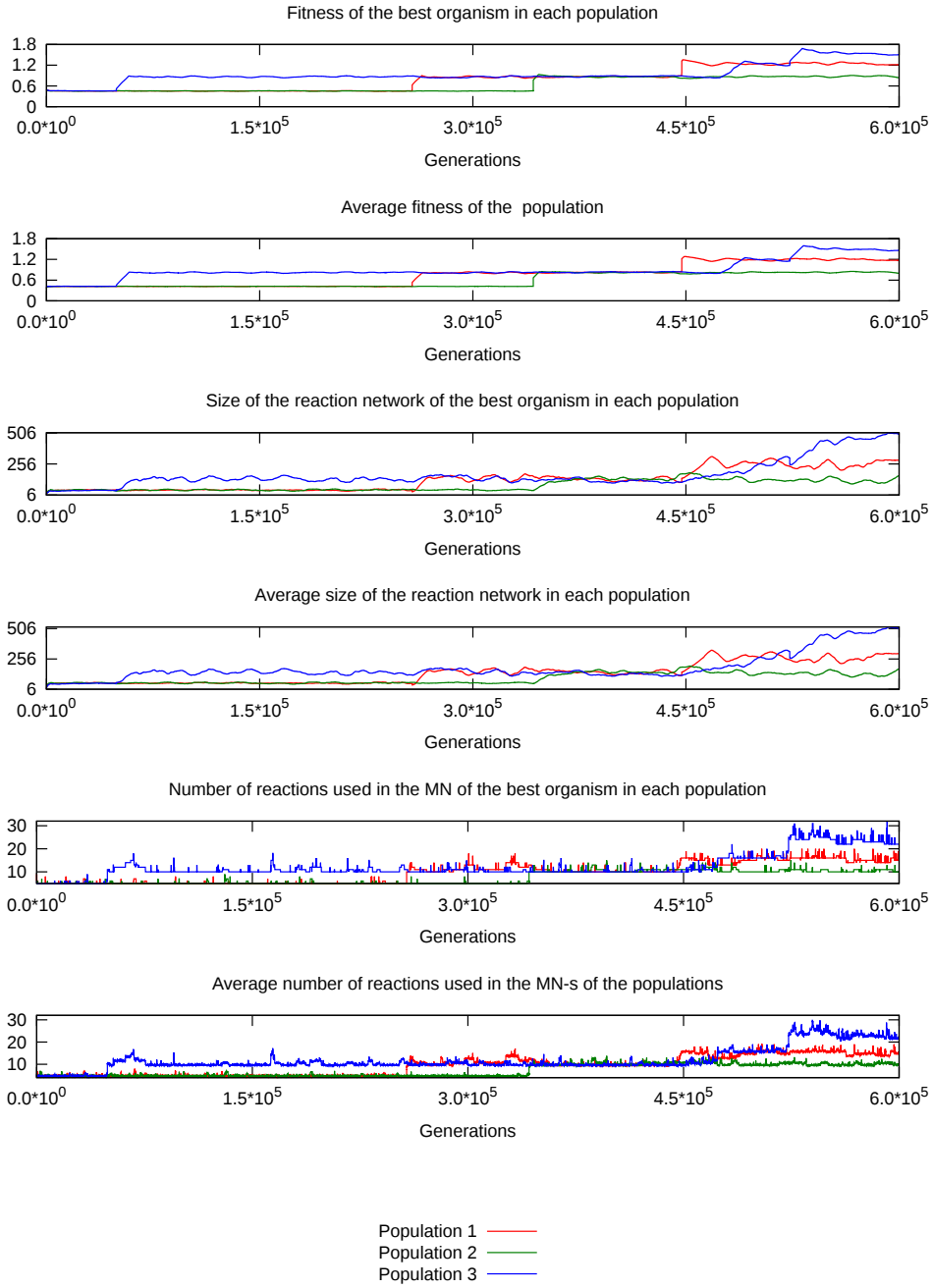

Figure 5: Diagnostic plots for 3 typical populations of Simulation 1

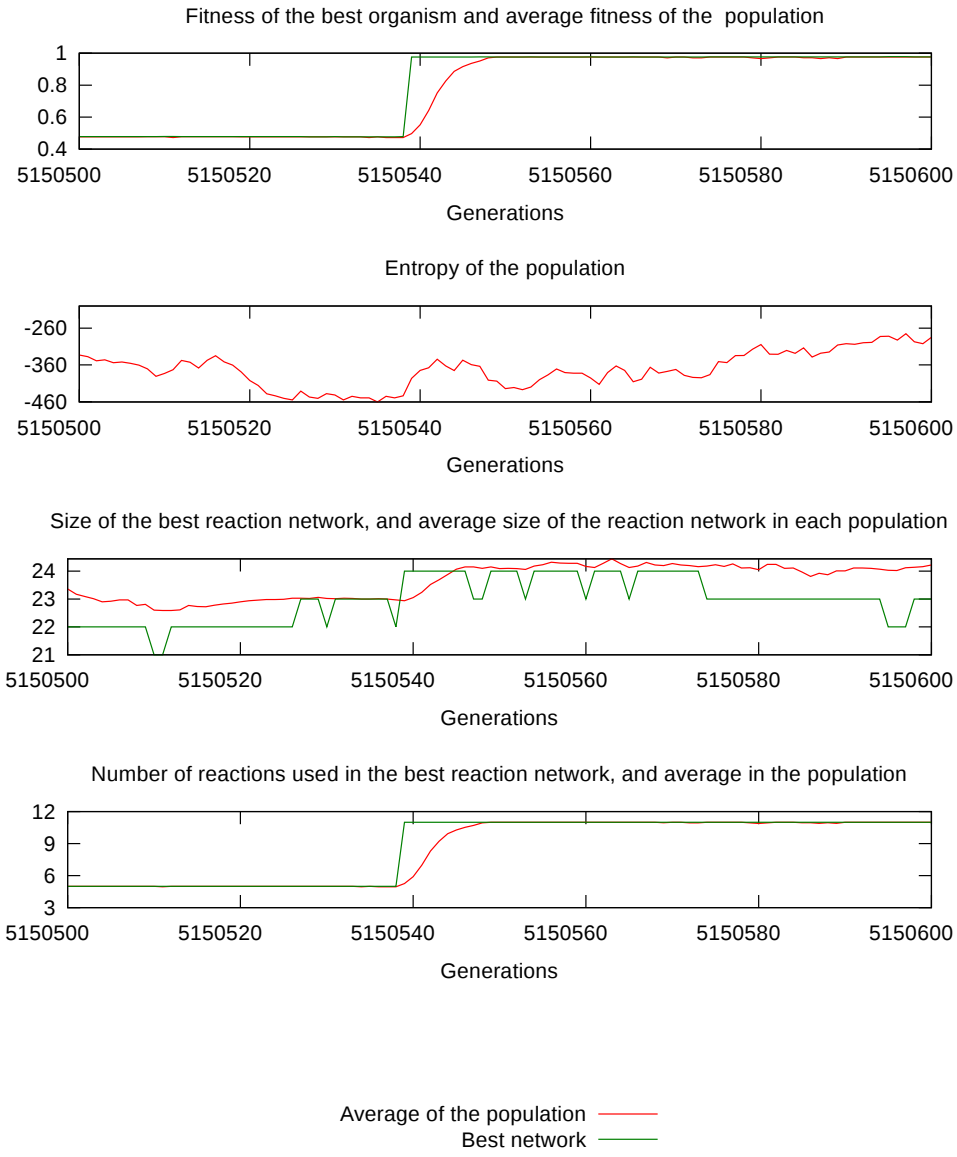

Figure 6: Extract of summary statistics of Population 1 of Simulation 2 showing the fixation of a beneficial mutation

reactions, to produce 1 ATP the network needs ~ 10, and for 1.5 ATP it needs 14.

The used reactions of the best performing network of Population 3 of Simulation 1 are shown on Figure 7. The network found alternative pathways to bypass the restrictions imposed by the bounds on the reaction fluxes. The two longer alternative pathways (to the right) meet at phosphoenolpyruvate, and there they diverge again, one converting an previously produced AMP (adenosin monophosphate) to ATP while the other converting ADP to ATP. We observe similar meeting and diverging paths in the simulations that follow. The reactions of the real glycolytic network are shown in yellow on this figure. In real glycolysis 3-phosphorglycerate is isomerized to 2-phosphorglycerate, this network therefore performs almost all reactions that real glycolysis does, but not in a single path.

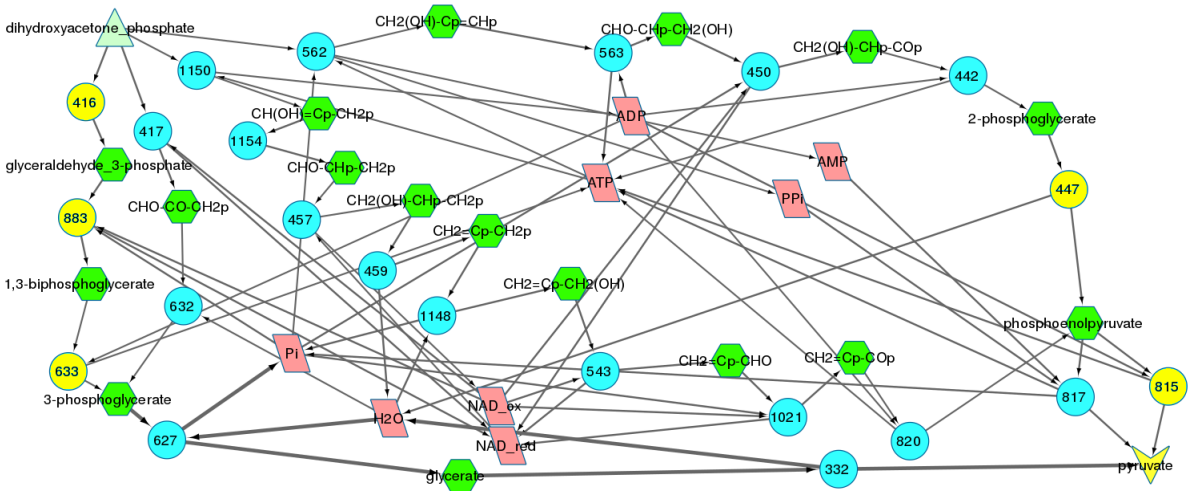

Figure 7: Reactions with nonzero flux in the fittest network at the end of the Simulation 1 in Population 3. The reactions of real glycolysis have been coloured yellow. Thin arrows mean a $v = 0.5$, thick ones mean $v = 1$. Arrows pointing to reactions mean the reaction takes the origin of the arrow as reactants, arrows originating from reactions mean the targets are products of the reaction.

An overview of the MN-s is shown on Figure 8 at three points in time, the beginning of the simulation (left), at the midpoint, after 1.5×10^5 generations, and the final stage. The disconnected nodes at the bottom of the left plot are yet unused internal metabolites. Note the small disconnected components of the middle and the left plots. These are a result of the mutation method implemented in our program, described in Section 2.3.

On the middle and left images of Figure 8 some central nodes can be observed around which the network appears to be centered. To examine this feature we plot the node degree distribution of the final MN on Figure 9. We find a small number of highly connected nodes (hubs) and many less connected ones. One might call this a scale free network, therefore we fitted a power law distribution of the form $P(n) = n^{-\gamma}$, shown on Figure 9. We found the parameter estimate to be $\gamma = 1.236$. Similar distributions can be found when examining final networks of other simulations.

The similarity indices for this simulation are shown on Figure 10, for all the reactions in the MN-s above the diagonal, and for the reactions with nonzero flux below it. Note that in every population we have 100 cells, and therefore in this figure we show $300 \times 300 = 9000$

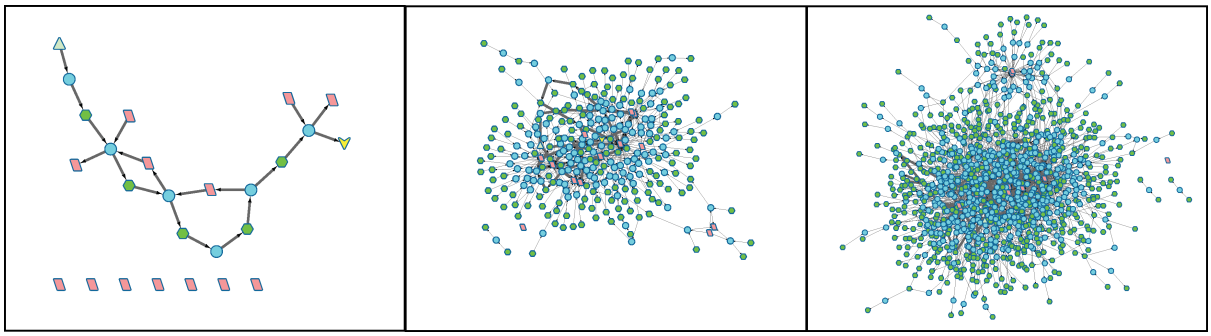

Figure 8: Overview of the best performing metabolic network of Population 3 of Simulation 1 at three stages of its evolution, at the start of the simulation, at the midpoint, and at the end

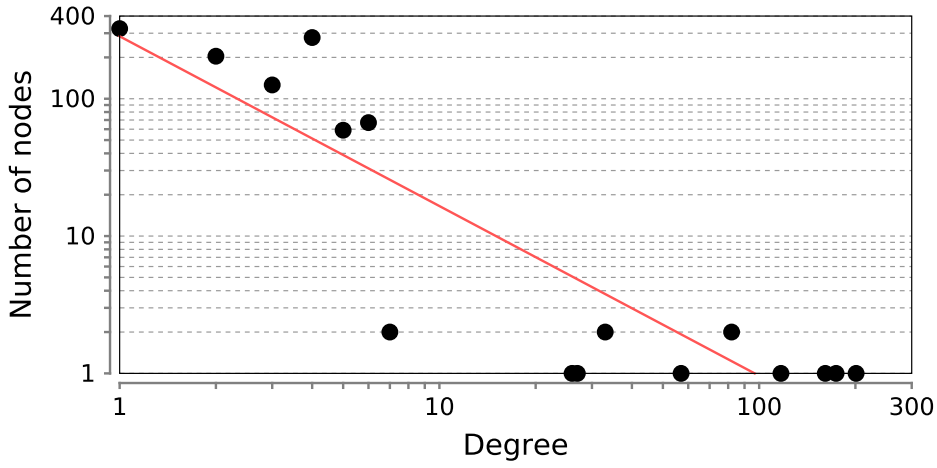

Figure 9: Node degree distribution of the final network of Population 3 of Simulation 1 with a power law distribution fitted

similarity indices. The used reactions are very similar within the populations, but show a small degree of inter-population similarity. The red and purple stripes in the upper triangular part within Populations 2 and 3 show MN-s that use different reactions from the rest of the population. This is a mutation, either neutral or deleterious, that has spread to part of the population. We know the mutation is not beneficial, as if it was it would appear in the fitness plots of Figure 5, and would quickly overtake the population with a high probability. When looking at the complete reaction network, the similarity becomes smaller within the populations, and disappears almost completely between populations.

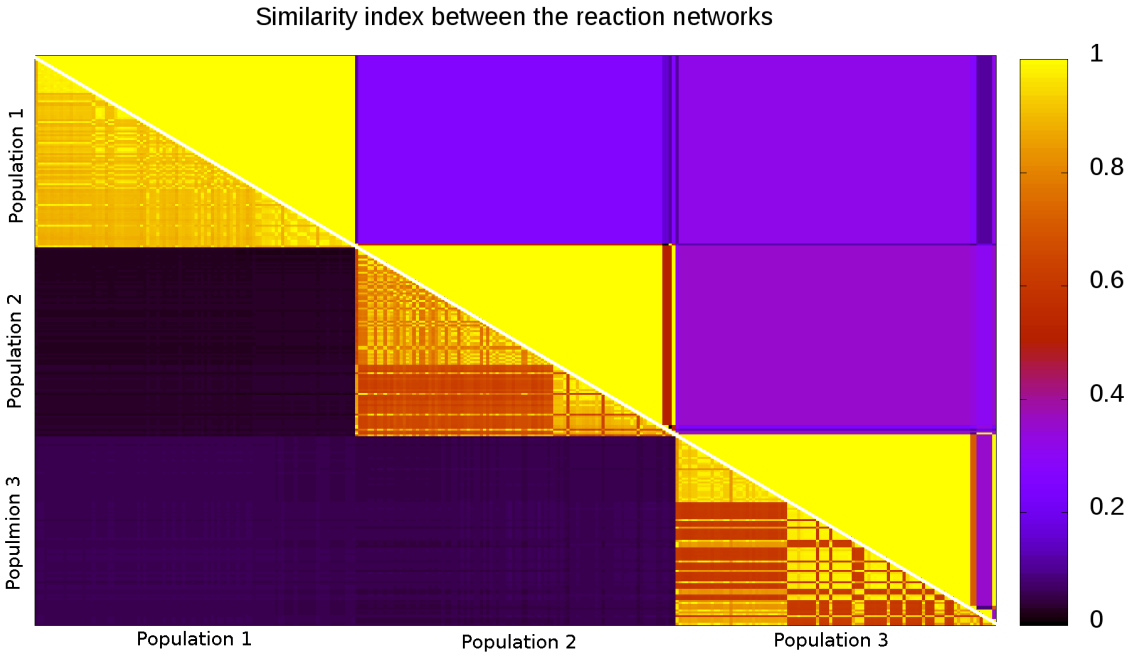

Figure 10: Similarity matrix of the first 3 populations of Simulation 1

The IPR of the similarity indices for every reaction within an MN, comparing each member of 3 populations to the best performing network in the given population is shown on the left of Figure 11 at each of the 10 checkpoints. The IPR of the similarity index for used reactions are shown in the middle of this figure. It shows that the set of reactions within populations is diverse, due to the very high mutation rate, however the used reactions are more similar in a population. The fitness values within a population are very similar, with the exception of Population 3 at checkpoint 6. This outlier value is most likely the effect of the population being overtaken by a mutation at the point of saving the checkpoint. The IPR plots of not shown populations are very similar to the ones shown.

3.1 The effect of mutation rate

The next pair of simulations were run to examine the effect of mutation rates on the patterns of evolution. The initial MN and the objective function of these simulations was the same as for Simulation 1. The probability of point mutation was set to $p_{point} =$

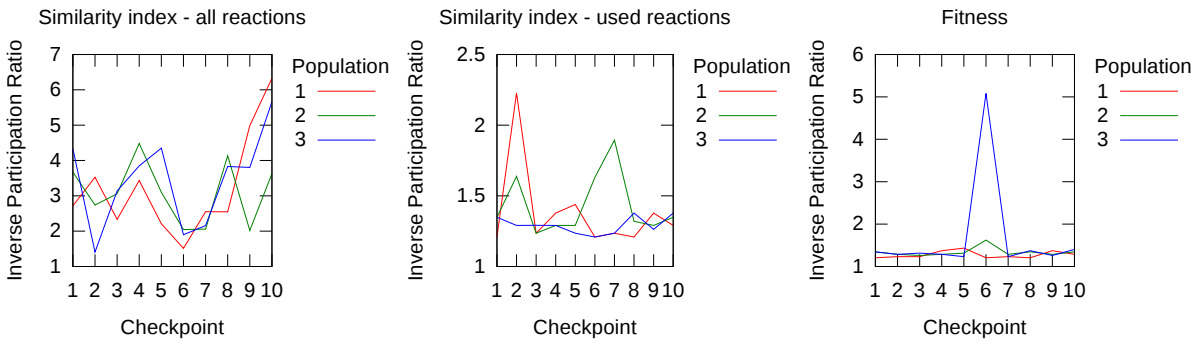

Figure 11: Inverse participation ratio of the two similarity indices and the fitness of 3 population of Simulation 1 at each checkpoint

0.01 with the cost of reaction $k_{reaction} = 0.001$ for Simulation 2, and for Simulation 3 $p_{point} = 0.1$ with $k_{reac} = 0.01$. The populations again consisted of 100 cells each, giving the expected number of mutations per generation to be 1 and 10 for Simulations 2 and 3 respectively.

Diagnostic plots for 3 typical populations are shown on Figure 12 and Figure 14. Both simulations evolved the populations of MN-s for 6×10^7 generations. This meant that Simulation 3 had ~ 10 times more mutations than Simulation 2.

Simulation 2 evolved the populations through approximately 600000 mutations, and of the populations that completed the run $\sim$ 28% had no improvement in their fitness $\sim$ 55% had an improvement of ~ 0.5, and 16% have improved by ~ 1. The discovery of a new pathway to produce ATP increases the fitness function by 0.5, but after such an improvement the MN-s usually gather a large amount of unused reactions, in this case about 70 reactions. This lowers the fitness function by $70 \times k_{reac} = 0.07$. The number of unused reactions fluctuates wildly around ~ 110 for a network with fitness ~ 1, and around ~ 150 for a cell with fitness ~ 1.4. The number of reactions necessary for a fitness increase is 6 (11 reactions in total), and the other increase could be achieved using an additional 4 reactions (15 reactions in total). Overall $\sim$ 10% of the reactions are used within a MN. The entropy of the population fluctuates around ~ -360 which can correspond to a population of ~ 80 identical cells, and a few mutants that are different from them. This can be observed on the similarity matrix of the final networks shown on Figure 13. We can see that the populations are homogeneous, with very few different cells even when every reaction is considered (below diagonal). The used reactions are similar between populations 1 and 3. The similarity considering all reactions is very low between the populations. The IPR plots of the populations are similar to those shown for Simulation 1, with even less variation in the case of used reactions. An IPR plot similar to this simulation's is shown on Figure 19.

When visualizing the final networks of this simulation pathways similar to those at the end of Simulation 1 emerge. The networks have again "rediscovered" reaction 633, and found alternative pathways to produce ATP. The alternative pathways are longer than the original glycolysis, and in the fittest networks they run in pairs. These pairs of pathways run from DHAP to pyruvate, but they meet at intermediate compounds, in some cases

more than once. In some cases we find one of these pathways not producing any net ATP, but they contribute to the other part of the pair by providing nutrients to it (eg. phosphate). When considering the node degree distribution of the resulting networks the results are similar to those shown on Figure 9.

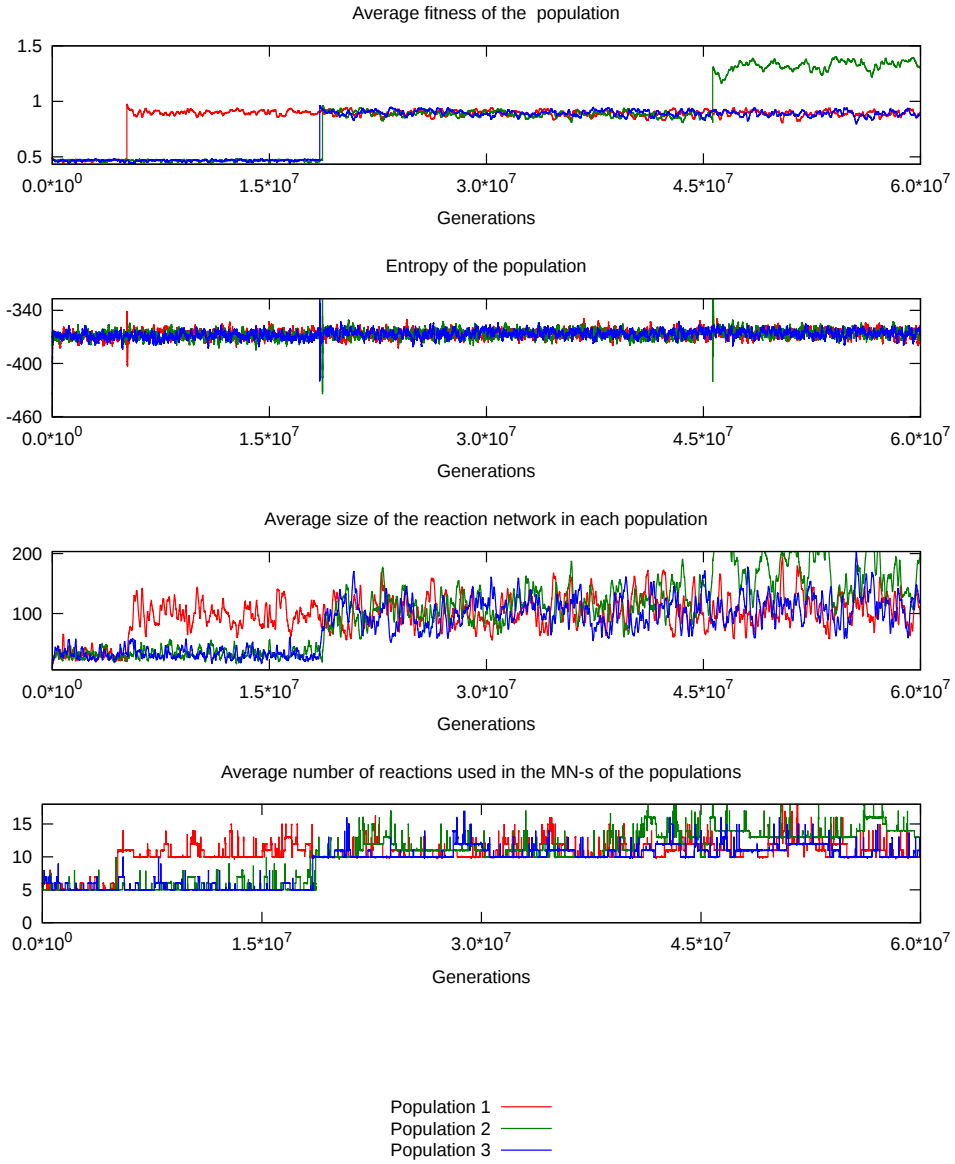

Figure 12: Diagnostic plots of 3 typical populations of Simulation 2.

Simulation 3 had 10 times more mutations than Simulation 2. Even though the number of reactions in the MN-s was smaller, due to a higher reaction cost, k_{reac} the populations of Simulation 3 improved further than those of Simulation 2. Here every simulation made advanced their ATP production, $\sim$ 16% made 1 improvement, 79% made 2, and 5% (1 population) made 3 improvements in their ATP production. After an improvement the networks gain reactions here too, but they only gain $\sim$ 15 networks after a single

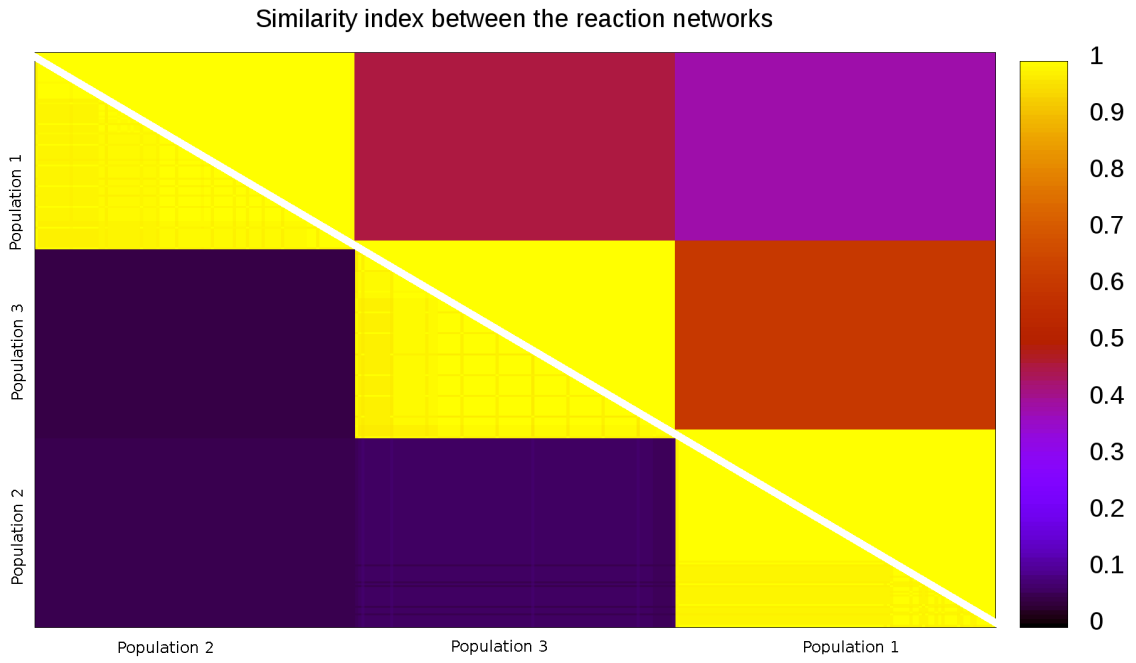

Figure 13: Similarity matrix of the 3 populations of Simulation 2

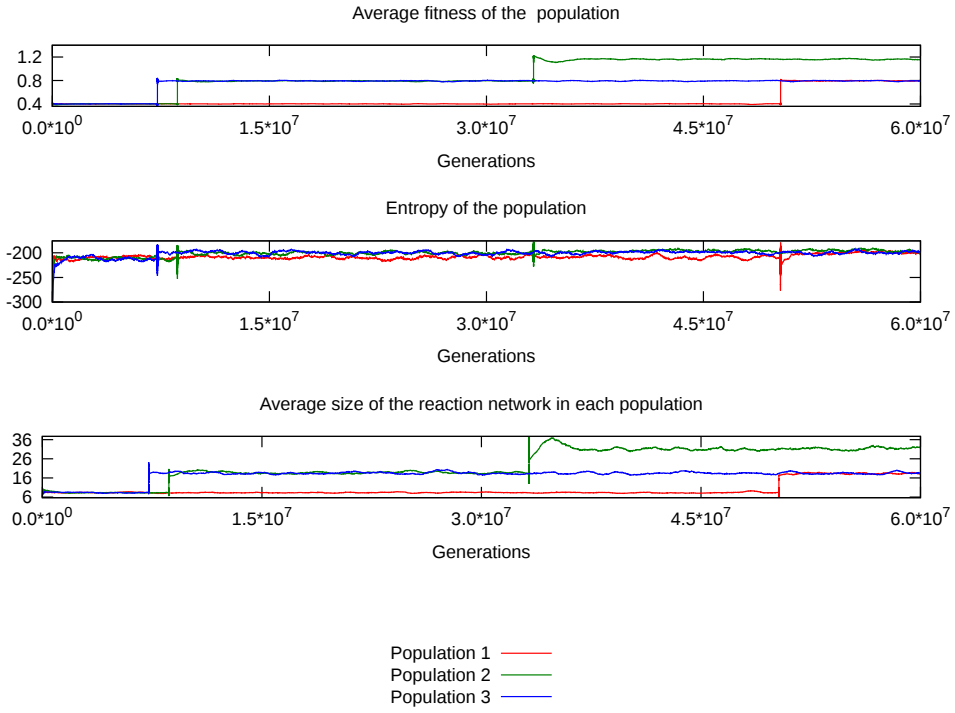

Figure 14: Diagnostic plots of 3 typical populations of Simulation 3. This simulation was performed before the number of used reactions was included in the simulation's log

improvement. The total number of used reactions required for an improvement is similar to that of Simulation 2, but in this case the proportion of reactions used by the MN is $\sim 50\%$. The entropy of the populations fluctuates less wildly than that of Simulation 2, around -200. This can correspond to ~ 7 approximately equal sized subpopulations, and a few lone mutants. This can be observed in the similarity matrix of the simulation shown on Figure 15. The used reactions are identical within almost every reaction within a population, and they show a rather large degree of similarity between populations too. When considering the whole metabolic network we can see that the populations themselves are diverse, but within a population networks only differ from each other by a handful of reactions.

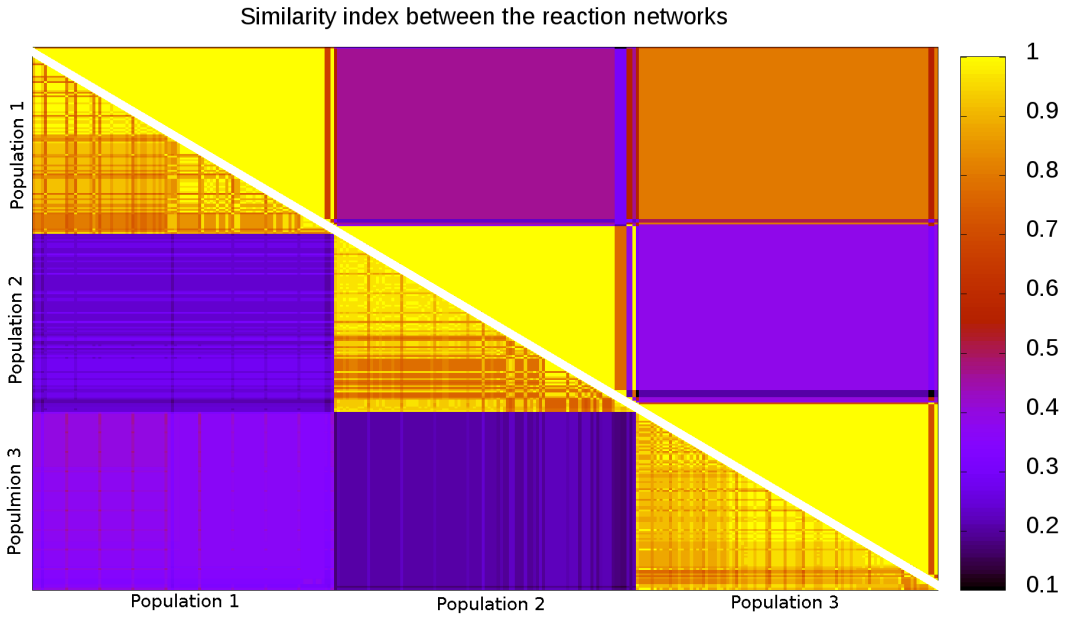

Figure 15: Similarity matrix of 3 populations of Simulation 3.

In an other simulation we have allowed the cells to exchange genetic material (reactions) with each other in a process resembling horizontal gene transfer [31]. We chose the reproducing cell (A) as before, but we also chose an other one uniformly(B) that will attempt to transfer a gene to the reproducing cell. We select a reaction of the donor cell (B) uniformly, and add this reaction to the reproducing cell's genome, if not already present. A transfer is deemed successful if the donated reaction is not present in A before the transfer. After the transfer this reaction is then present in both A and B. When exchanging genetic material we observed that the number of successful horizontal gene transfers was very low ~ 0.01. The small success rate for this transfer is due to the high degree of similarity within populations, as shown on Figure 15. The effects of horizontal gene transfer could be examined in further studies in a simulation setup where two populations evolve separately, with the only interaction between them being a small possibility of horizontal gene transfer between cells in the different populations.

We also considered a simulation with $p_{point} = 0.01$ and $k_{reac} = 0.01$ but none of the 20 populations produced any improvement after a similar simulation time as used above.

3.2 Multiple sources

Real cells import a variety of nutrients from their environment, and most of their MN-s can use multiple similar compounds to produce their end-products [1]. To test how the presence of multiple sources influences the process of evolution of our networks, we performed simulations of networks that had access to 2 nutrients. To be able to compare the resulting networks with our previous simulations, the initial network was, as previously, the modified trunk of glycolysis, shown on Figure 4. The two sources used for this network were DHAP, and G3P. These two molecules are the end-products of the upper glycolytic pathway. They enter the lower part by DHAP being converted into G3P, and the two G3P molecules undergo the same trunk pathway. The sink of the simulation was pyruvate, as before, with the objective function rewarding ATP production only. The simulation was run using $p_{point} = 0.01$ and $k_{reac} = 0.001$. We relaxed the steady state condition on the flux of phosphate, allowing the cell to take phosphate up, or dispose of it. This allows the cell to remove phosphate by other means than the ADP $\rightarrow$ ATP conversion. Interestingly this was not used by the cells. We call this run Simulation 4.

Diagnostic plots for 3 populations are shown on Figure 16. All of the populations that finished the simulation made at least 2 improvements in their ATP production. These first two improvements appeared rather early in the simulation, with the last population to obtain them among all 20 populations being Population 2. These two improvements increase the number of used reactions to 10, and the networks acquire ~ 200 reactions in total. Approximately 22% of the simulations made 3 improvements, and 11% made 4 improvements. The number of reactions necessary for the 3 improvement network is 14, and with 17 reactions the system can produce 4 improvements. The entropies of the populations are similar to the previous simulations.

We show one of the fittest networks at the end of this simulation on Figure 17. The network has abandoned the DHAP source, instead using only G3P. This is a phenomena appearing in all the populations. The flux through the reactions is shown on the figure by the different thickness of the arrows. The thinner ones symbolize reactions with flux 0.5, while the thick ones mean a flux of 1. It is important to note how interlinked the network is. While there are 3 pathways leading away from G3P, and there are also 3 leading in to pyruvate, these pathways are very connected. The pathways starting with reaction 883 meets with the one starting with 879 at phosphoenolpyruvate, and the paths starting with 879 and 877 diverge and meet at glycerate.

When comparing the similarity matrices of the populations we find that the diversity within a population is small, much like on Figure 15. The networks with 3 improvements show a large degree of similarity in the used reactions ~ 0.8. Of those with only 2 improvements, there is a pair where the used reactions are identical, and many of them have a moderate amount of similarity between them ~ 0.6. If we look at the whole list of reactions, any pair of networks show a very small degree of similarity $\sim 0.1 - 0.2$. This is expected, as the unused reactions are random, their presence or absence provides no fitness advantage or disadvantage to the cell other than the reaction cost for each reaction. The IPR plots of this simulations are not shown here, as they are very similar to those shown for the next simulation on Figure 19.

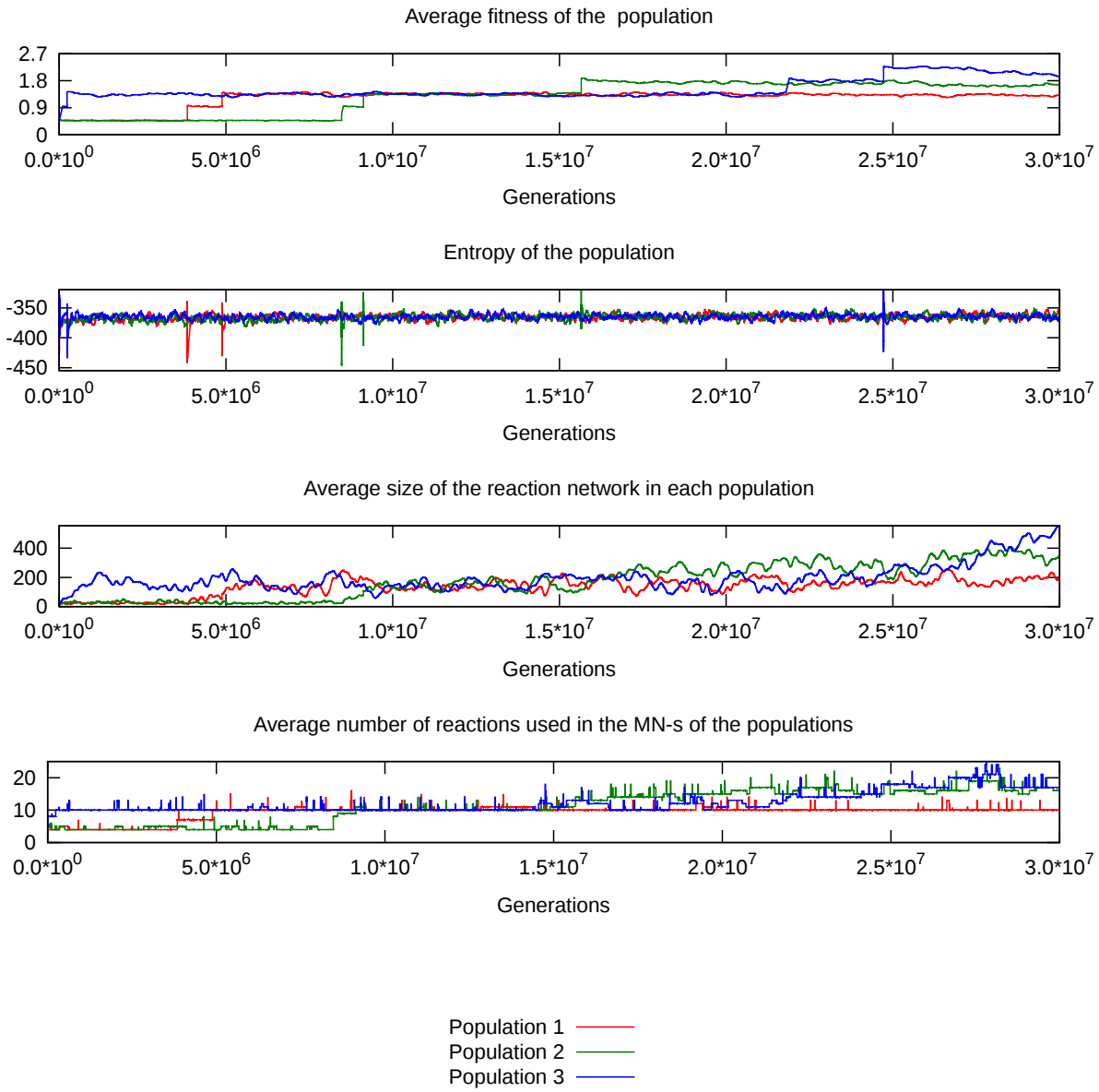

Figure 16: Diagnostic plots for 3 populations of Simulation 4

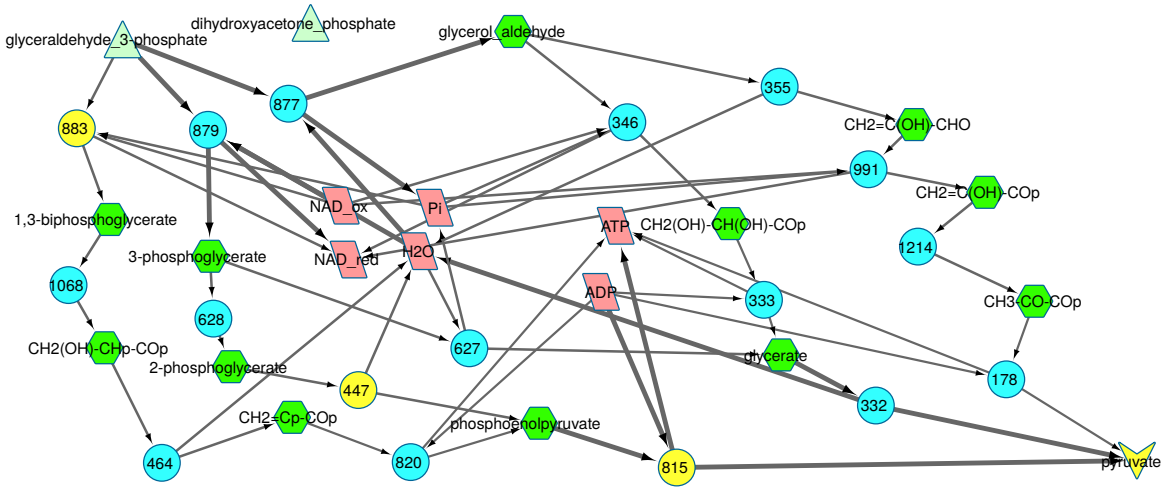

Figure 17: The best performing network at the end of Simulation 4. The reactions used by real glycolysis have been coloured yellow.

3.3 The effect of objective functions

The fitness of real cells is influenced by a variety of factors, including their energy and biomass output. To examine the effects of different fitness functions on their evolution we consider simulations where we reward the cells for a combination of their energy output and their biomass production. Due to the limitations of the linear programming method used in flux balance analysis this combination can only be linear, $f_{goal} = av_{energy} + bv_{biomass}$ for some constants a and b. A more realistic approach could be implemented using non-linear objective functions of the form $f_{goal} = \min(v_{energy}, v_{biomass})$. In this setup the cell would have to produce both energy and biomass, both of which are requirements of reproduction. Such a objective function would be more realistic as the cell needs both energy and biomass for the reproduction. If has a large amount of either, but none of the other reproduction would not happen.

We performed simulations with equal weights for ATP and biomass (pyruvate) production, $a = b = 0.5$ in the above formula, and also ones with only rewarding biomass production ($a = 0$, $b = 1$). We run these simulations with DHAP and G3P as nutrients separately, and combined too. The results for all three nutrient cases were very similar for the pyruvate only objective function, but the combined nutrient case showed more interesting results for the composite objective function, therefore we present the results of the combined nutrient cases. As the nutrients of the network are phosphate containing, and the sink is not, we allow the network to take up or dispose of phosphate, as in the previous simulation.

Simulation 5 used $p_{point} = 0.01$, and $k_{reac} = 0.001$ as before. The nutrient of the network was G3P and DHAP, and its sink pyruvate, with the rewarded flux in the objective function being pyruvate production only. We show diagnostic plots for 3 populations of this simulation on Figure 18.

Even though this simulation only considered 10^7 generations, the networks have made significant improvements very rapidly. Every population achieved a fitness value ~ 2 in

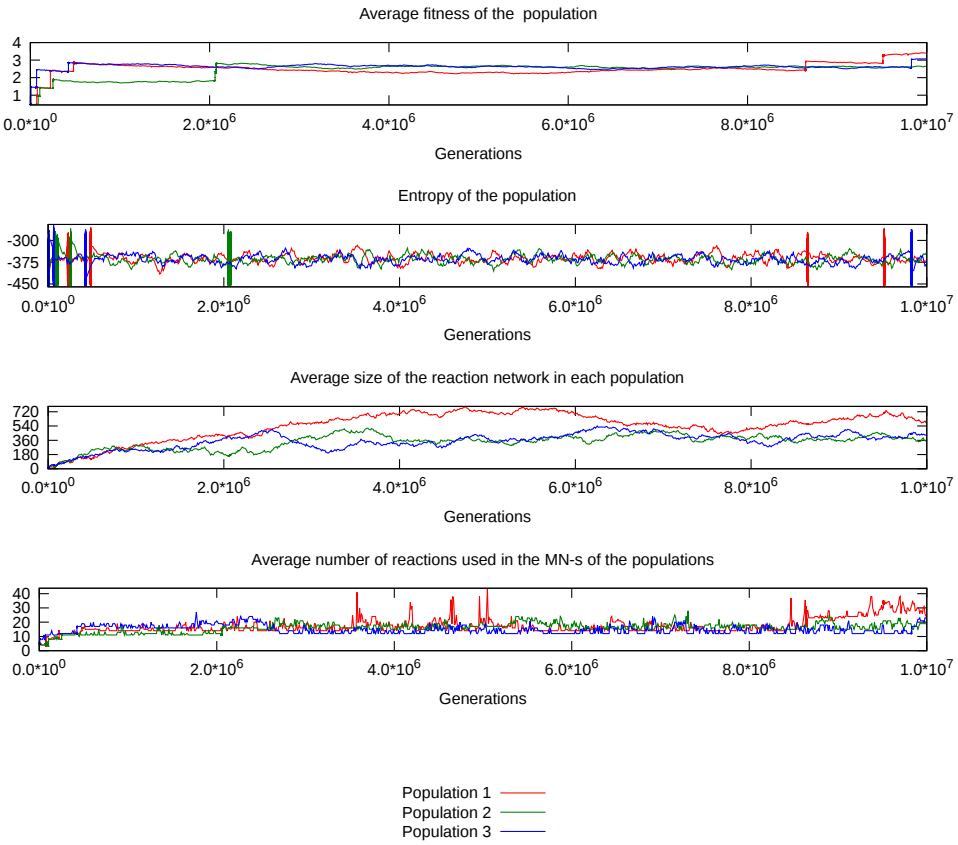

Figure 18: Diagnostic plots for 3 populations of Simulation 5

the first $\sim$ 10% of the simulation time. Approximately 15% of the populations made 6 improvements, producing a pyruvate flux of 4, compared to the original 0.5. The reaction networks gain unused reactions after each improvement as seen on the previous diagnostic plots, the networks with the most improvements have $\sim$ 600 reactions at the end of the simulation, giving them a fitness value of $\sim$ 3.5. The MN-s rate of obtaining new reactions appears to be smoother than for the simulations before, and the number of reactions in the networks fluctuates less. The fittest networks use $\sim$ 30 reactions at the end of the simulation, but Population 1 uses up to 40 reactions at multiple stages of its evolution. It is interesting to note that the increase in the number of used reactions does not come with a fitness advantage in this case, meaning the network most likely lost a reaction (due to a mutation) and it could find an alternative, but longer path to produce the same pyruvate flux as before. Such a mutation has spread in a population multiple times during the simulation as we can see on the last graph of Figure 18. Closer examination of the log file produced by the population shows that the phenomena doesn't always spread in the population, sometimes it only affects one cell that by losing this reaction becomes the best cell, this allele then spreads to part of the population. We show the IPR plots for the 3 populations on Figure 19. The populations are more similar than those with a higher mutation rate. This can be observed in the left and middle graphs. The populations are almost completely homogeneous in therms of used reactions, and they are more similar (smaller IPR value) in terms of all the reactions. The IPR values of the fitness are higher in this plot than on Figure 11; this is due to the method of calculating the IPR values addressed in Section 2.5.

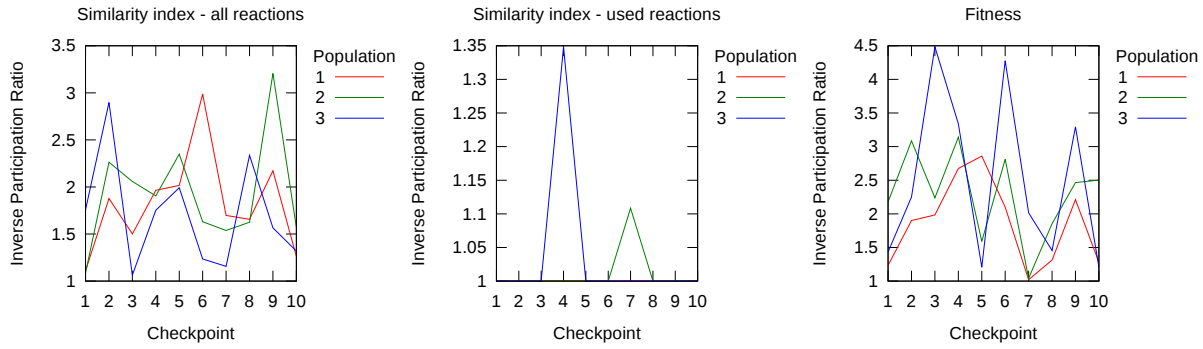

Figure 19: IPR plots of 3 populations of Simulation 5

It is interesting to note that the shortest reaction, discovered by most networks very early, providing twice the original pyruvate flux is only 3 reactions long. One such path is G3P $\rightarrow$ 3-phosphoglycerate $\rightarrow$ glycerate $\rightarrow$ pyruvate, with the phosphate being discarded at the middle reaction. As in this case we did not reward ATP production, the networks rather discard the phosphate groups they shed from the nutrient, but some of the networks produce a small amount of ATP, while other consume a similar amount.

Most of the populations abandon the DHAP source early in the simulation, and use G3P, as in Simulation 4. Towards the end of the run, the fittest networks return to it, and use a small amount of DHAP. The pathways starting from DHAP are joining into the G3P paths quickly, within $1-2$ reactions. The metabolite where the two pathways join (eg. CHO-CO-CH2p in some networks) can be considered a precursor molecule, as it is

processed the same way regardless of the source.

Simulation 6 was setup using the same parameters as Simulation 5 except for its objective function, which in this case was an equal weighted combination of ATP and pyruvate production. The simulation has again been run for 10^7 generations. Even though this is shorter than Simulation 2, the networks have shown rapid improvement, and the runtime of the two simulations was approximately equal. This is due to the networks of Simulation 6 growing much larger than those of Simulation 2.

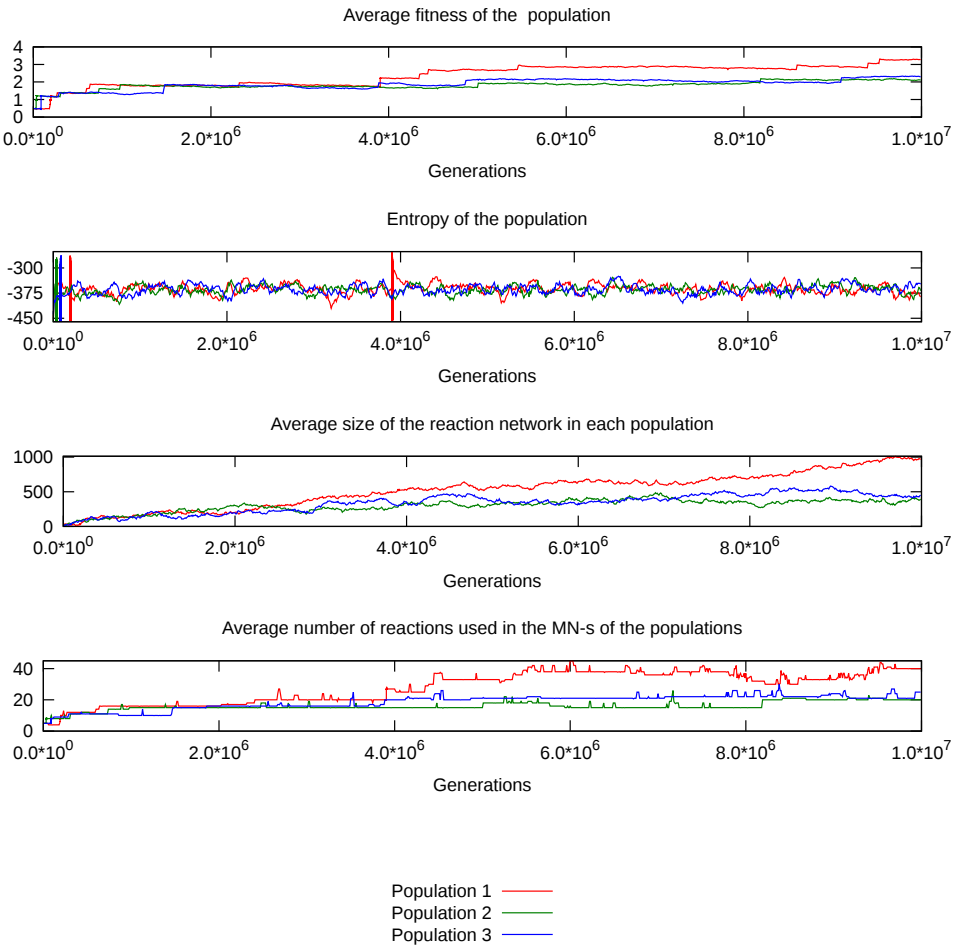

Figure 20: Diagnostic plots for 3 populations of Simulation 6

We show diagnostic plots of 3 populations of this simulation on Figure 20. The networks start to improve very early on the simulation, some in the first few thousand generations. The steps of improvement were generally smaller than those for Simulation 5, usually bringing an improvement of$\sim$ 0.25, meaning either ATP or pyruvate production was increased by 0.5. This wasn't always the case, for example Population 1 has a larger jump visible on Figure 20 near the beginning of the simulation. The fittest network of the simulation (in Population 1) had a final fitness value of 3.28. The value of the objective function of this network is 4.25, which is lowered by the network having 967 reactions in total at the end of the simulation. After an improvement we don't see the

sudden increase in the number of total reactions, as for the previous simulations, but the networks keep obtaining new reactions throughout the whole run. Approximately 11% of the populations improved to a fitness value above 3, $\sim 35\%$ of them to above 2, with the rest of them remaining around 1.8. The fittest reactions collect $\sim 800 - 1000$ reactions, those above 2 have ~ 500, while the rest $300 - 400$. The number of used reactions in the fittest network is 40, those with fitness above 2 have ~ 25 used reactions and the ones with fitness ~ 1.8 have 14. The entropies of the populations show fewer spikes then before. The lack sudden increases in the number of reactions, and the spikes in entropy are most likely the result of the smaller improvements. In this simulation the summary statistics shown on Figure 20 were written out every 10000 generations, unless a larger increase in fitness was observed. Since most of the improvements were ~ 0.25 these have not triggered the writing out of intermittent statistics. The simulation could be run again with lowering the threshold for this trigger, but unfortunately we didn't have enough time for this.

The similarity matrix of the populations show a large degree of similarity within the populations, as before. The inter-population similarity is higher than before, often above 0.7 for the used reactions, and there is 3 populations (of fitness ~ 1.8 that have identical used reactions. These populations have been stagnant at this fitness level for approximately the last third of the simulation, therefore the configuration obtained by them might constitute a local optimum. These networks have $14 - 15$ used reactions, while most of those with fitness above 2 have ~ 25. The networks unable to improve above the fitness value of 1.8 might have a difficulty in obtaining enough useful reactions to reach the next fitter state.

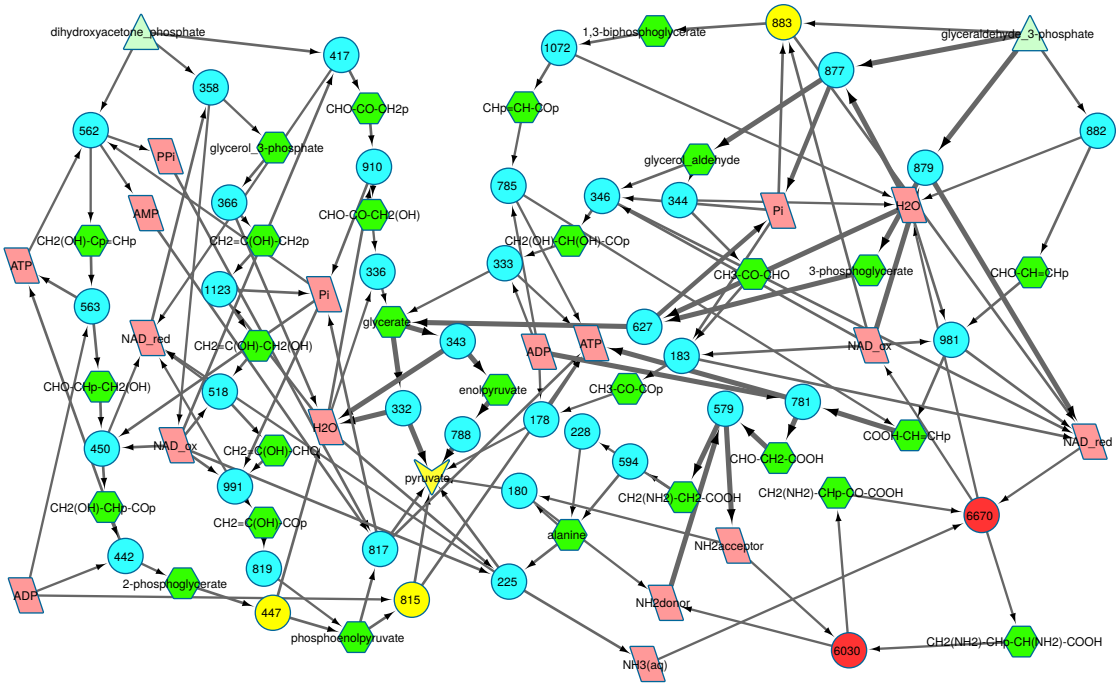

Figure 21: Fittest network of Simulation 6. Reactions from the real glycolytic network are marked with yellow

We show the fittest network of this simulation on Figure 21. This network uses both of the sources it was provided, with the networks connecting the two sources to the sink, connected by only a few reactions. Note that there are multiple instances of certain internal metabolites on the figure (ATP, ADP, H_2O, NAD^+ and NADH). This was created solely for the purpose of better showing the network, the cell received the same nutrients as before. Should we not have included multiple instances of these, the image would be difficult to interpret due to the large amount of arrows crossing. We show the reactions of the real glycolytic pathway with yellow. An interesting feature of this MN is the cycle marked by red. This pair of reaction takes ammonium and converts it into a source of amidogen (NH_2). This is then inserted on a compound that did not contain nitrogen previously. This amidogen group is removed only at the last step of the pathway, when converting to pyruvate. This set of reactions is a very interesting way of allowing the cell to produce more pyruvate.

4 Discussion

In this work we have considered the effects of the rate of mutation and the objective function on the patterns of evolution in our model. We have found similar evolutionary patterns when we modified the rates of mutation, but improvements appeared later when we decreased it. We found more diverse networks appeared if we used multiple sources, but this might be a result of a poor choice of our original source. When we considered more complex objective functions the number of improvements increased and the resulting networks became even more complex.

The effect of mutation rate

The authors of [36] note that crucial parameters for modelling evolution, such as the frequency of beneficial mutations are not known. The rate of spontaneous mutations per genome per genome replication is approximately 0.01 for higher eukaryotes [10]. We have considered mutation rates comparable to this, but it is important to note that the genome of our organism are very short compared to that of real organisms. High per gene mutation rates such as those we considered can be relevant to very early stages of evolution. Decreasing the mutation rate to a value lower than 0.01 while keeping the population size at $N = 100$ yields no improvements in similar scale simulations as the ones considered above. This is because decreasing the mutation probability (eg. to $p = 0.001$) means the expected number of mutations per generation will be less than one, making the populations too homogeneous for mutations to spread fast enough. If these simulations were run for a considerably longer time than the ones we performed they might show improvements.

We observed similar networks emerging in most of the populations of any given simulation. In the case of Simulations 1, 2, and 3 the emerging networks were similar between simulations too. We found that the expected times until an improvement appears within a network increases as the mutation rate decreases. This effect might appear because of the small population size we used. In real organisms the rates of mutations are smaller

than in our simulations, but the size of their populations are vastly greater than those we considered, eg. 1 ml liquid sample of bacterial colonies can contain bacteria in excess of 10^9 [37]. It would be interesting to see whether every population would arrive to the same network if we allowed the simulations to run for a long enough time.

We find that the fitness cost of reactions also influences the improvement patterns. These costs must allow the network some "experimental" reactions, that are not themselves used, but form a pool of reactions from which improvements can emerge. If the cost of having a reaction is high, the number of reactions is too tightly bound, the network takes more mutations to improve, as we have seen in Simulations 2, 3 and the unsuccessful $p_{point} = 0.01 = k_{reac}$ simulation.

In all of our simulations with $k_{reac} = 0.001$ we observed the large number of unused reactions in our metabolic networks. Some of these unused networks are part of the next potential improvement of the network. If the mutation rate is low, and the cost of reactions high the expected time until an improvement becomes very large. We observed this phenomena in the simulation with $p = 0.01$ and $k_{reac} = 0.01$ that produced no improvements during a similar time that was used for the other simulations. The genetic information of our molecules consists of the reactions they can produce. In a real cell the genetic information serves a variety of functions, including the coding of proteins that catalyse reactions. A large proportion of the DNA of higher eukaryotes does not code proteins, the estimate for the proportion of human DNA that codes proteins is $\sim 3\%$ [39]. Non-coding DNA is sometimes referred to as junk DNA, as its function is not well understood. Its purpose can be redundancy, in case of the coding parts of the DNA are damaged. In our network some unused reactions can potentially serve as "backups" in case some of the currently used ones are removed, as seen in Simulation 5.

We would like to have tested this phenomena on the evolved networks, by calculating the fitness of the networks if one or more randomly chosen used reactions are removed. By finding synthetic lethal genes one could map out how modular the networks are following the approach of [20]. Unfortunately we couldn't do this due to the lack of time.

In our model we found no difference in the evolution of the networks depending on the presence or absence of horizontal gene transfer. This is most likely the effect of our very homogeneous populations.

The effect of multiple sinks and different objective functions

When we allowed our organisms to use multiple molecules as sources while keeping the objective function as before we observed the networks abandoning the initial source DHAP in favour of G3P. This influenced the evolutionary patterns shown by the populations by allowing more and faster improvements in the fitness of the cells. This is most likely the effect of G3P being better suited to serve as a nutrient than DHAP, but further simulations with different nutrients would need to be performed to confirm this.

The fittest organisms of Simulation 5 returned to the DHAP source by the end of the simulation. This could mean that the network has reached the maximal flux possible while using G3P only. These networks use DHAP to produce a compound that was already produced by the G3P only network with a smaller flux, using this compound

similar to a precursor molecule.

It would be interesting to see whether similar networks using precursor molecules would emerge if we allowed the network to use different nutrients. Precursors would likely emerge in dynamic environments, where the availability of nutrient fluctuates with time. Simulations considering this could be performed using our code with slight modifications to it.

An important difference between the first and second three simulations is that for the second three we allowed our cells to take up or dispose of phosphate. Doing so we observed more diverse improvement patterns; this is a result of our nutrients containing phosphate, while the sinks of the network not. If we do not allow phosphate to be disposed of, the only way for the cell to rid itself of the phosphate of the nutrient is via the auxiliary reaction ATP $\rightarrow$ ADP [8]. We have deliberately not included the phosphate in this reaction to allow the networks to dispose of phosphate. The populations of Simulation 4 did not use the offered phosphate sink however, as in that case the only reaction rewarded in the objective function was the ATP producing one. If the network disposed of phosphate in any other way, it would not produce ATP and therefore the phosphate disposing reaction path would not constitute a fitness advantage.

When we reward multiple processes in the objective function of cells their evolution can take smaller steps at one time. By allowing them to exchange phosphate with their environment we decouple the process of ATP production from biomass (pyruvate) production. This allows the cells to develop a pathway that does only one of the two goals giving the process greater flexibility. The resulting networks are more complex than previously, with two lightly connected sub-networks, for the two sources. Such a network could function as a module within the glycolytic pathway providing a high degree of redundancy to it. The upper part of glycolysis produces DHAP and G3P, these could be processed using the network shown on Figure 21. In case of the loss of reactions in either substrates network a larger flux could be used by the other network by using the reaction DHAP $\rightleftharpoons$ G3P, keeping the performance of the cell intact.

The pair of reactions shown in red on Figure 21 play a role somewhat similar to the citric acid cycle, in the sense that they conserve the carbon containing molecules within the cycles and use a circular pathway. In contrast to the citric acid cycle that is driven by the energy releases during the oxidization of pyruvate, our cycle is fuelled by the reduction of the partaking compounds via NAD^+ (Nad_ox on the figure).

Limitations of our approach

The main limitations of our model arise due to the small population size, the restrictions on the chemistry and the simplifications introduced by using flux balance analysis.

The populations of simple organisms, that our MN-s model are usually substantial. The probability of the fixation of a neutral mutation is inversely proportional to the population size, thus for such large populations it is very unlikely [37]. In our populations with size

[8] In the real world this reaction works by consuming a water molecule and producing a phosphate, ATP + H_2O $\rightarrow$ ADP + Pi

$N = 100$ approximately one in every hundred mutations fix in the population. This aides the emergence of beneficial mutations, since the acquiring of no single reaction yields a fitness increase. For example in Simulation 2 the MN-s need 4 additional reaction to produce more ATP. 3 of these reactions must fix in the population as a neutral one, and stay dormant until the last reaction can fix too. Thus smaller populations might be able to increase their fitness faster than large ones. This claim is supported by [21]. Although we have not examined the effect of population size in our model the program could easily be extended to accommodate a larger population. Existing research suggests that for a larger populations the patterns of evolution become more deterministic with larger populations [36].

We restricted our cells to use a small but realistic set of possible molecules in their MN-s. As the lower part of the glycolytic pathway only utilizes molecules that are present within our network, this has most likely not served as a limiting factor in modelling the evolution of the networks. We have performed test simulations using a larger set of molecules, using CHOPN molecules containing up to 5 carbon atoms. The number of such compounds in our list is 12606, and there are 100835 reactions between them. The results of these simulations were similar to those discussed in Section 3. This is likely the result of our nutrients being 3 carbon molecules. In order for the networks to use molecules containing 4 or 5 carbon molecules, they would have to obtain this carbon from the only source of carbon other than the nutrients, CO_2. The release of CO_2 is an exothermic reaction, therefore its capture would be endothermic, in other words requiring a large investment of energy. While the networks could invest this energy using ATP[9], the resulting network would probably not produce a net flow of ATP.

A more important restriction was necessary to make in order to keep the flux calculations a linear programming problem. In such a problem one needs to provide boundaries for the parameters that are to be optimized, in our case the fluxes through reactions. These boundaries can be infinite, and can contain other fluxes, such as $v_1 + v_2 < c$, but can only be linear functions of them. This forced us to determine the direction of every reaction depending only on its Gibbs free energy change. In real chemical pathways only the overall change of the free energy needs to be negative, in other words parts of the path may have a positive ΔG, if the rest of the path compensates for it by having a large enough negative ΔG value. Such a constraint would look like $\sum_{i=1}^{N} \Delta G(v_i) < 0$, where $\Delta G(v_i) = \frac{-v_i}{|v_i|}\Delta G_i$, with ΔG_i being the free energy change of reaction i. Here $\Delta G(v_i)$ simply flips the sign of ΔG_i if the reaction is happening in the reverse direction, compared to the direction in the list of reactions. Such a constraint is non-linear due to the sign-taking, and therefore incompatible with the linear programming approach. In contrast our path has to have every reaction's ΔG as negative, (or not very positive for a reduced flux). Paths that require the investment of energy are not impossible in our network. The energy investment usually comes in the form of an ATP $\rightarrow$ ADP conversion, such as in case of the reaction shown on Figure 1. Such investments are used mostly in Simulation 5, where we do not reward ATP production. In the simulations where ATP production is used as a component of the objective function a network with an ATP investment would have to produce at least 2 ATP molecules in order for the network to be considered a

[9]An example for this is reaction 182 in our list of 4 carbon reactions: pyruvate (CH3-CO-COOH) + ATP + CO_2 $\rightarrow$ oxaloacetate (COOH-CH2-CO-COOH) + ADP + Pi with $\Delta G_0 = -8.825$ eV

fitness advantage.

Flux balance analysis as implemented here also assumes that the maximal flux through each reaction can only be one of three cases, as described in Section 2.2. Using smoother maxima here, eg. calculating the maximal flux as a continuous function of the current free energy change could lead to more accurate results. The likelihood of a reaction happening not only depends on the difference of free energy between the initial compounds and the end-products, but also on the free energy landscape between these two states. A reaction with a high potential barrier will be less likely to happen than one where the free energy change is a roughly monotonic function.

Further research in the area could consider the effects of different environmental variables, such as temperature, acidity and the concentration of metabolites. Our code can be easily adapted to use different environments. The effects of sample size could be analysed by considering larger populations, with the same mutation rates as we did, however in case of a larger population the mutation rates can be further decreased to closer approximate that of real cells. The evolution of the networks could be examined in a dynamically changing environment, for example varying nutrient availability or physiological conditions. Simple examples of such dynamic environments could be simulated using our software with slight modifications.

Acknowledgements

I would like to thank my supervisor Dr. Bartek Waclaw, for his time and guidance throughout the project; Máté Nagy who provided help and advice in setting up the program, and continued aiding me in resolving difficult bugs; Rozália Nagy for her help with figures and support throughout the project; and Ioan-Bogdan Magdau for his code calculating moving averages.

References

[1] Aditya Barve and Andreas Wagner. A latent capacity for evolutionary innovation through exaptation in metabolic systems. *Nature*, 500(7461):203–206, 2013.

[2] Aditya Barve, Sayed-Rzgar Hosseini, Olivier C Martin, and Andreas Wagner. Historical contingency and the gradual evolution of metabolic properties in central carbon and genome-scale metabolisms. *BMC Systems Biology*, 8(1):48, 2014.

[3] Richard A Blythe and Alan J McKane. Stochastic models of evolution in genetics, ecology and linguistics. *Journal of Statistical Mechanics: Theory and Experiment*, 2007(07):P07018, 2007.

[4] Béla Bollobás. *Random graphs*. Springer, 1998.

[5] Bálint Borgulya. Evolution of Metabolic Networks. `https://github.com/HoneyMonster7/frankenstein/`, 2015.

[6] Steven J Court, Bartlomiej Waclaw, Rosalind J Allen, et al. Lower glycolysis carries a higher flux than any biochemically possible alternative. *Nature Communications*, 6, 2015.

[7] Charles Darwin. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. 2009.

[8] J Arjan GM de Visser and Joachim Krug. Empirical fitness landscapes and the predictability of evolution. *Nature Reviews Genetics*, 15(7):480–490, 2014.

[9] Peter Dittrich, Johannes C Ziegler, and Wolfgang Banzhaf. Artificial chemistries - a review. *Artificial Life*, 7(3):225–275, 2001.

[10] John W Drake and John J Holland. Mutation rates among RNA viruses. *Proceedings of the National Academy of Sciences*, 96(24):13910–13913, 1999.

[11] Jeremy L England. Statistical physics of self-replication. *The Journal of chemical physics*, 139(12):121923, 2013.

[12] Nathan Entner and Michael Doudoroff. Glucose and gluconic acid oxidation of Pseudomonas saccharophila. *Journal of Biological Chemistry*, 196(2):853–862, 1952.

[13] W.J. Ewens. *Mathematical Population Genetics 1: Theoretical Introduction*. Interdisciplinary Applied Mathematics. Springer New York, 2004. ISBN 9780387201917.

[14] Iwona J Fijalkowska, Roel M Schaaper, and Piotr Jonczyk. DNA replication fidelity in Escherichia coli: a multi-DNA polymerase affair. *FEMS Microbiology Reviews*, 36(6):1105–1121, 2012.

[15] Ronald Aylmer Fisher. *The genetical theory of natural selection: a complete variorum edition*. Oxford University Press, 1930.

[16] Christoph Flamm, Alexander Ullrich, Heinz Ekker, Martin Mann, Daniel Högerl, Markus Rohrschneider, Sebastian Sauer, Gerik Scheuermann, Konstantin Klemm, Ivo L Hofacker, et al. Evolution of metabolic networks: A computational framework. *Journal of Systems Chemistry*, 1(4), 2010.

[17] Errol C Friedberg, Graham C Walker, Wolfram Siede, and Richard D Wood. *DNA repair and mutagenesis*. American Society for Microbiology Press, 2005.

[18] C Gyles and P Boerlin. Horizontally transferred genetic elements and their role in pathogenesis of bacterial disease. *Veterinary Pathology Online*, page 0300985813511131, 2013.

[19] Leland H Hartwell, John J Hopfield, Stanislas Leibler, and Andrew W Murray. From molecular to modular cell biology. *Nature*, 402:C47–C52, 1999.

[20] Arend Hintze and Christoph Adami. Evolution of complex modular biological networks. *PLoS Comput Biol*, 4(2):e23, 2008.

[21] Kavita Jain, Joachim Krug, and Su-Chan Park. Evolutionary advantage of small populations on complex fitness landscapes. *Evolution*, 65(7):1945–1955, 2011.

[22] Hawoong Jeong, Bálint Tombor, Réka Albert, Zoltan N Oltvai, and A-L Barabási. The large-scale organization of metabolic networks. *Nature*, 407(6804):651–654, 2000.

[23] Judith Kleinfeld. Could it be a big world after all? The six degrees of separation myth. *Society, April*, 12:5–2, 2002.

[24] Albert Lehninger, David L. Nelson, and Michael M. Cox. *Lehninger Principles of Biochemistry*. W. H. Freeman, Fifth edition, June 2008.

[25] Richard E Lenski, Charles Ofria, Robert T Pennock, and Christoph Adami. The evolutionary origin of complex features. *Nature*, 423(6936):139–144, 2003.

[26] Andrew Makhorin. GNU linear programming kit. *Moscow Aviation Institute, Moscow, Russia*, 38, 2001.

[27] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[28] Patrick Alfred Pierce Moran. Random processes in genetics. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 54, pages 60–71. Cambridge Univ Press, 1958.

[29] Nasif S Nahle. Irreversibility.

[30] Jeffrey D Orth, Ines Thiele, and Bernhard Ø Palsson. What is flux balance analysis? *Nature Biotechnology*, 28(3):245–248, 2010.

[31] Csaba Pál, Balázs Papp, and Martin J Lercher. Adaptive evolution of bacterial metabolic networks by horizontal gene transfer. *Nature Genetics*, 37(12):1372–1375, 2005.

[32] Juli Peretó. Embden-Meyerhof-Parnas Pathway. In *Encyclopedia of Astrobiology*, pages 485–485. Springer, 2011.

[33] Nathan D Price, Jennifer L Reed, and Bernhard Ø Palsson. Genome-scale models of microbial cells: evaluating the consequences of constraints. *Nature Reviews Microbiology*, 2(11):886–897, 2004.

[34] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13(11):2498–2504, 2003.

[35] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Boost C++ Libraries. http://www.boost.org/, June 2000.

[36] Ivan G Szendro, Jasper Franke, J Arjan GM de Visser, and Joachim Krug. Predictability of evolution depends nonmonotonically on population size. *Proceedings of the National Academy of Sciences*, 110(2):571–576, 2013.

[37] Bartek Waclaw. The physics of biological evolution. Lecture notes and problem solutions, 2015.

[38] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440–442, 1998.

[39] Gane Ka-Shu Wong, Douglas A Passey, Ying-zong Huang, Zhiyong Yang, and Jun Yu. Is "junk" DNA mostly intron DNA? *Genome Research*, 10(11):1672–1678, 2000.

Appendix A. main.cpp

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <cstdlib>
#include <boost/random.hpp>
#include <boost/graph/bipartite.hpp>
#include <string>
#include <iostream>
#include <fstream>
#include <utility>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <queue>
//for the command line flags
#include <unistd.h>
#include "reaction.h"
#include "cell.h"
using namespace boost;
int main(int argc, char **argv)
{
  int seedForGenerator=1;
  int c;
  std::string jobName;
  double probOfPointMut=1.0;
  double probOfHorizGene=0;
  double smallK=1e-3;
  double probOfSinkMutation=0;
  while ((c = getopt(argc,argv,"hs:j:p:g:k:a:")) != -1)
    switch(c)
    {
      case 's':
        seedForGenerator=std::stoi(optarg);
        break;
      case 'p':
        probOfPointMut=std::stod(optarg);
        break;
      case 'g':
        probOfHorizGene=std::stod(optarg);
        break;
      case 'k':
        smallK=std::stod(optarg);
        break;
      case 'a':
        probOfSinkMutation=std::stod(optarg);
        break;
      case 'h':
```

```
48          std::cout<<"Accepted options:"<<std::endl;
49          std::cout<<"\t -s [intSeed] seed for the MersenneTwister, default is 1, max is
             ↪  2147483647"<<std::endl;
50          std::cout<<"\t -p [probOfPointMut] probablity of point mutations (double) should be beteen 0 and
             ↪  1. Default:1"<<std::endl;
51          std::cout<<"\t -g [probOfHorizGene] probablity of horzontal gene transfer (double) should be
             ↪  beteen 0 and 1, preferably lower than probOfPointMut. Default:0"<<std::endl;
52          std::cout<<"\t -k [smallKforFitness] cost of a reaction within the reaction network.
             ↪  Default:0.001"<<std::endl;
53          std::cout<<"\t -a [probOfSinkMutation] probability of a sink mutation (addition or deletion with
             ↪  equal probability). Default:0"<<std::endl;
54          std::cout<<"\t -j [jobName] name for the folder to output the results into. If skipped a
             ↪  timestamped directory will be used for this."<<std::endl;
55          exit(0);
56          break;
57        case 'j':
58          jobName=optarg;
59          break;
60        case '?':
61          if (optopt =='s')
62            std::cout<<"Option -s requires an integer argument"<<std::endl;
63          else if (optopt =='j')
64            std::cout<<"Option -j requires the desired folder name (jobName)"<<std::endl;
65          else if (optopt =='p')
66            std::cout<<"Option -p requires the desired probality of point mutations (double)"<<std::endl;
67          else if (optopt =='g')
68            std::cout<<"Option -g requires the desired probality of horizontal gene transfer"<<std::endl;
69          else if (optopt =='k')
70            std::cout<<"Option -k requires the desired cost for reactions"<<std::endl;
71          else if (optopt =='a')
72            std::cout<<"Option -a requires the desired probability"<<std::endl;
73          else
74            std::cout<<"Unknown option, try -h for allowed options."<<std::endl;
75          return 1;
76        default:
77          exit(1);
78      }
79    RandomGeneratorType generator(seedForGenerator);
80    //for automatically creating an output file named from current time
81    time_t now=time(0);
82    tm *ltm = localtime(&now);
83    std::ostringstream forDateTime;
84    forDateTime<<1900+ltm->tm_year<<"."<<1+ltm->tm_mon<<"."<<ltm->tm_mday<<"_"<<1+ltm->tm_hour<<":"<<1+ltm-
       ↪  >tm_min<<":"<<1+ltm->tm_sec;
85    std::string dateForFileName=forDateTime.str();
86    std::string actualFilename;
87    if (!jobName.empty()){
88      actualFilename=jobName;
89    }
90    else {
91      actualFilename=dateForFileName;
92    }
93    std::string dirCommand="mkdir "+actualFilename;
94    const int dir_err = system(dirCommand.c_str());
95    if (-1 == dir_err)
96    {
97      std::cout<<"Error creating directory!"<<std::endl;
98      exit(1);
99    }
100   std::cout<<"The chosen probablity for point mutations is: "<<probOfPointMut<<" and for horizontal gene
       ↪  transfer it is: "<<probOfHorizGene<<std::endl;
101   std::ofstream improvementlog;
102   improvementlog.open(actualFilename+"/"+actualFilename+".fitt");
103   //set the number of internalmetabolites here:
104   int nrOfInternalMetabolites=13;
105   reaction::nrOfInternalMetabolites=nrOfInternalMetabolites;
106   std::cout<<"Tests begin."<<std::endl;
107   std::vector<reaction> reacVector;
108   std::vector<substrate> substrateVector;
109   try
```

```
110     {
111       reaction::readCompounds(DATA_PATH "fullnewcompounds.dat", substrateVector);
112       std::cout<<"length of the reaction vector is: "<<reacVector.size()<<std::endl;
113       reaction::readReactions(DATA_PATH "fullnewreactions.dat", reacVector, substrateVector);
114       std::cout<<"length of the reaction vector is: "<<reacVector.size()<<std::endl;
115       std::cout<<"length of the substrate vector is: "<<substrateVector.size()<<std::endl;
116     }
117     catch(std::runtime_error& e)
118     {
119       std::cout << e.what() << std::endl;
120       return 1;
121     }
122     if(reacVector.size() < 300)
123     {
124       std::cout << "not enough data read." << std::endl;
125       return 1;
126     }
127     std::cout<<"Building the neighbour list..."<<std::endl;
128     substrate::buildNeighbourList(reacVector,substrateVector);
129     std::cout<<"Neighbour list built."<<std::endl;
130     environment currentEnvironment;
131     currentEnvironment.atpCont=1e-1;
132     currentEnvironment.adpCont=1e-2;
133     currentEnvironment.ampCont=1e-4;
134     currentEnvironment.nadoxcont=1e-1;
135     currentEnvironment.nadredcont=1e-4;
136     currentEnvironment.piCont=1e-3;
137     currentEnvironment.ppiCont=1e-3;
138     currentEnvironment.co2cont=1e-5;
139     currentEnvironment.nh3aqCont=1e-5;
140     currentEnvironment.glutCont=1e-2;
141     currentEnvironment.oxo2Cont=1e-3;
142     for (reaction& current : reacVector){
143       current.recalcEchange(currentEnvironment);
144     }
145     std::vector<int> subset= {417,884,1070,448,816,629};
146     //setting static member variables
147     cell::nrOfInternalMetabolites=nrOfInternalMetabolites;
148     cell::reactionVector=reacVector;
149     cell::substrateVector=substrateVector;
150     //this is the k value for the fitness function
151     cell::smallKforFitness=smallK;
152     //setting the probabilities for mutations
153     cell::probabilityOfMutation=probOfPointMut;
154     cell::probabilityOfHorizontalGenetransfer=probOfHorizGene;
155     cell::sourceSubstrate.push_back(104);
156     cell::sinkSubstrate.push_back(54);
157     //creating the original cell here
158     cell trialcell(subset);
159     //Restructuring the population mutations
160     int numberOfCells=100;
161     std::vector<int> populationIndex(numberOfCells,0);
162     std::vector<int> howManyOfEach(numberOfCells,0);
163     //cell type vector
164     std::vector<cell> cellVector(numberOfCells);
165     cellVector[0]=trialcell;
166     howManyOfEach[0]=100;
167
168     std::string fileName="initial";
169     trialcell.printXGMML(fileName);
170     std::cout<<"Initial performance is; "<<trialcell.getPerformance()<<std::endl;
171     // this is the pathfinding algorithm, not used now
172     //cell::findThePaths(needMore, needLess, currentReactions, targetCompound, reacVector, substrateVector,
        ↪   actualFilename);
173       double probabilityOfAnyMutation=cell::probabilityOfMutation+cell::probabilityOfHorizontalGenetransfer;
174       int numberOfMutationsToSimulate=60000000;
175       double dcheckpointLenght=(numberOfMutationsToSimulate/probabilityOfAnyMutation)/(10*numberOfCells);
176       int NRofCheckpoints=10;
177       int checkPointLength=(int)dcheckpointLenght;
178       const int generationsPerWriteout=1000;
```

```
179     double previousAvgFittness=cellVector[0].getPerformance();
180     std::cout<<"To simulate "<<numberOfMutationsToSimulate<<" mutations, the checkPointLength has been
         ↪   set to "<<checkPointLength<<std::endl;
181     //defining the queues here for the intermittent steps between logging
182     double maxFittQueue [generationsPerWriteout];
183     double avgFittQueue [generationsPerWriteout];
184     double entropyQueue [generationsPerWriteout];
185     int bestNetSizeQueue [generationsPerWriteout];
186     double avgNetSizeQueue [generationsPerWriteout];
187     int bestUsedReacsQueue [generationsPerWriteout];
188     double avgUsedReacsQueue [generationsPerWriteout];
189     int arrayPos=0;
190     std::time_t startTime=std::time(nullptr);
191     //outer loop is there in order to save networks every 10% of the simulation
192     for (int outerLoop=0; outerLoop<NRofCheckpoints; outerLoop++){
193       for (int k=0; k<checkPointLength; k++){
194         for (int iter=0; iter<numberOfCells; ++iter){
195           //running a generation here - as many mutations as many cells there are in the population
196           cell::mutatePopulation(populationIndex,howManyOfEach,cellVector,generator);
197         }
198         cell::printProgressFile(populationIndex,cellVector,howManyOfEach,k,outerLoop,generationsPerWriteo
           ↪   ut,checkPointLength,improvementlog,previousAvgFittness,maxFittQueue,avgFittQueue,entropyQue
           ↪   ue,bestNetSizeQueue,avgNetSizeQueue,bestUsedReacsQueue,avgUsedReacsQueue);
199       }
200       std::time_t currentTime=std::time(nullptr);
201       double timeElapsed=std::difftime(currentTime,startTime);
202       std::cout<<"Time for Checkpointing! "<<timeElapsed<<" seconds have passed since the start, that's
           ↪   "<<(outerLoop+1)*checkPointLength/timeElapsed<<" generations per seconds."<<std::endl;
203       cell currentBest=cell::printNFittest(populationIndex,cellVector,numberOfCells);
204       std::vector<cell> bestCells=cell::getBestNCells(populationIndex,cellVector,cellVector.size());
205       if (outerLoop != NRofCheckpoints){
206         //the checkpoints will be saved into separate folders
207         //creating a directory for saving the checkpoints into
208         std::ostringstream forCheckpointFolder;
209         forCheckpointFolder<<actualFilename<<"/CP"<<outerLoop+1;
210         std::string dirCommand="mkdir "+forCheckpointFolder.str();
211         const int dir_err = system(dirCommand.c_str());
212         for (int i=0; i<bestCells.size(); i++){
213           std::ostringstream forFileName;
214           forFileName<<forCheckpointFolder.str()<<"/"<<actualFilename<<"CP"<<outerLoop+1<<"NR"<<i+1<<"cel
             ↪   l";
215           bestCells[i].printXGMML(forFileName.str());
216         }
217       }
218     }
219     improvementlog.close();
220   std::cout<<"Tests completed."<<std::endl;
221 }
```

Appendix B. cell.h

```
1  #pragma once
2  #include <boost/graph/adjacency_list.hpp>
3  #include <boost/graph/visitors.hpp>
4  #include <boost/graph/breadth_first_search.hpp>
5  #include <boost/generator_iterator.hpp>
6  #include <boost/random/mersenne_twister.hpp>
7  #include <boost/random.hpp>
8  #include <boost/random/uniform_real.hpp>
9  #include <boost/random/uniform_int.hpp>
10 #include <cstdlib>
11 #include <boost/graph/bipartite.hpp>
12 #include <string>
13 #include <iostream>
14 #include <fstream>
15 #include <utility>
```

```
16  #include <algorithm>
17  #include <array>
18  #include <math.h>
19  #include <queue>
20  #include <unordered_set>
21  #include "reaction.h"
22  typedef boost::mt19937 RandomGeneratorType;
23  typedef boost::uniform_int<> UniIntDistType;
24  typedef boost::variate_generator<RandomGeneratorType&, UniIntDistType> Gen_Type;
25  typedef boost::uniform_real<> UniRealDistType;
26  typedef boost::variate_generator<RandomGeneratorType&, UniRealDistType> RealGenType;
27  class cell{
28    std::vector<int> availableReactions;
29    double performance;
30    std::vector<int> additionalSinks;
31    private: double firstPerformance;
32         std::vector<double> fluxThroughReacs;
33    public:
34         static std::vector<int> sourceSubstrate,sinkSubstrate;
35         static std::vector<reaction> reactionVector;
36         static std::vector<substrate> substrateVector;
37         static int nrOfInternalMetabolites;
38         static double smallKforFitness;
39         static double probabilityOfMutation;
40         static double probabilityOfHorizontalGenetransfer;
41         static double probabilityOfSinkMutation;
42         cell(std::vector<int>& availableReactons);
43         //other constructor that works similarily but won't calculate fittness automatically
44         cell(std::vector<int>& availableReactons, int notUsed);
45         cell();
46         bool operator<(const cell& other) const;
47         inline std::vector<int> getReacs() {return availableReactions;}
48         inline double getPerformance() const {return performance;}
49         inline std::vector<double> getFluxes() {return fluxThroughReacs;}
50         inline std::vector<int> getAddSinks() {return additionalSinks;}
51         cell  mutateAndReturn(RandomGeneratorType& generator);
52         void printReacs();
53         std::vector<int> canBeAdded();
54         void setFluxes(std::vector<double>& fluxVector);
55         double calcThroughput();
56         static void mutatePopulation(std::vector<int>& population,std::vector<int>& howManyOfEach,
            ↪   std::vector<cell>& cellVector, RandomGeneratorType& generator);
57         static std::vector<double> getPopulationFittness(std::vector<int>& population, std::vector<cell>&
            ↪   cellVector);
58         static void printPopulationFittnesses(std::vector<int>& population, std::vector<cell>& cellVector);
59         static cell printNFittest(std::vector<int>& population, std::vector<cell>& cellVector, int N);
60         static std::vector<cell> getBestNCells(std::vector<int>& population, std::vector<cell>&
            ↪   cellVector, int N);
61         static void findThePaths(std::vector<int> needMore, std::vector<int> needLess, std::vector<int>
            ↪   currentReactions, int TargetCompound, std::vector<reaction>& ReactionVector,
            ↪   std::vector<substrate>& SubstrateVector, std::string fnameString);
62         void printHumanReadable();
63         void printXGMML(std::string fileName);
64         static void printProgressFile(std::vector<int>& population, std::vector<cell>& cellVector,
            ↪   std::vector<int>& howManyOfEach, int k, int outerloop,const int generationsPerWriteout, int
            ↪   checkPointLength, std::ofstream& fileToWrite,double& previousFittness, double maxFittQueue
            ↪   [], double avgFittQueue [], double entropyQueue [], int bestNetSizeQueue [], double
            ↪   avgNetSizeQueue [], int bestUsedReacsQueue [], double avgUsedReacsQueue []);
65         void deleteThisSink(int thisSinkIsToBeGone);
66         void addThisSink(int thisSinkIsNowYours);
67         void theseAreYourAddSinks(std::vector<int> theseSinksAreNowYours);
68         //Only use in mutations. take care with this, remember to calc throughput
69         void setReacs(std::vector<int> theseAreYourNewReacs);
70    private: static int randomIntInRange(RandomGeneratorType& generator, int maxNumber);
71         static double randomRealInRange(RandomGeneratorType& generator, double maxNumber);
72  };
```

Appendix C. cell.cpp

```cpp
#include "cell.h"
#include <tuple>
// static class variables
std::vector<reaction> cell::reactionVector;
std::vector<substrate> cell::substrateVector;
int cell::nrOfInternalMetabolites;
std::vector<int> cell::sourceSubstrate;
std::vector<int> cell::sinkSubstrate;
double cell::smallKforFitness;
double cell::probabilityOfMutation;
double cell::probabilityOfSinkMutation;
double cell::probabilityOfHorizontalGenetransfer;
  cell::cell(std::vector<int>& tmpAvailReacs)
:availableReactions(tmpAvailReacs)
{
  double initialPerformance=calcThroughput();
  std::vector<double> fluxThroughReacs(availableReactions.size());
  performance=initialPerformance;
  firstPerformance=initialPerformance/22;
  std::vector<int> additionalSinks;
}
//this constructor just creates the vector availablereacs and initializes the other variables
  cell::cell(std::vector<int>& tmpAvailReacs, int notUsed)
:availableReactions(tmpAvailReacs)
{
  double initialPerformance=0;
  std::vector<double> fluxThroughReacs(availableReactions.size());
  performance=initialPerformance;
  firstPerformance=initialPerformance/22;
  std::vector<int> additionalSinks;
}
cell::cell() {}
void cell::addThisSink(int thisSinkIsNowYours) {
  additionalSinks.emplace_back(thisSinkIsNowYours);
  performance=calcThroughput();
}
void cell::theseAreYourAddSinks(std::vector<int> theseSinksAreNowYours){
  additionalSinks=theseSinksAreNowYours;
  performance=calcThroughput();
}
void cell::deleteThisSink(int thisSinkIsToBeGone){
  //thisSinkIsToBeGone needs to be the index of the sink to remove
  additionalSinks.erase(additionalSinks.begin()+thisSinkIsToBeGone);
}
void cell::printReacs() {
  std::cout<<"Current reactions: ";
  for (auto i : availableReactions){
    std::cout<<i<<", ";
  }
  std::cout<<" END"<<std::endl;
}
bool cell::operator<(const cell& other) const {
  //orders such that the largest performance wil be [0]
  return getPerformance()>other.getPerformance();
}
void cell::printXGMML(std::string filename){
  std::ofstream outfile,typesfile,edgeFile,xgmmlFile;
  //for the node_types
  std::set<int> reacNumbers;
  std::set<int> compoundIDs;
  std::set<int> internalMetIDs;
  std::vector<std::tuple<int,double,int>> edgeVector;
  std::string fileToOpen=filename + ".xgmml";
  xgmmlFile.open(fileToOpen);
  //xml header as cytoscape needs it
  xgmmlFile<<"<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>"<<std::endl;
```

```
67    xgmmlFile<<"<graph label=\""<<filename<<"\""<<std::endl<<" xmlns:cy=\"http://www.cytoscape.org\"
       ↪   "<<std::endl<<"xmlns=\"http://www.cs.rpi.edu/XGMML\" "<<std::endl<<"directed=\"1\">"<<std::endl;
68    xgmmlFile<<"\t <att name=\"Fittness\" type=\"real\" value=\""<<getPerformance()<<"\"/>"<<std::endl;
69    for (int j=0; j<availableReactions.size();j++){
70      int i=availableReactions[j];
71      double fluxOfCurrentReaction=fluxThroughReacs[j];
72      //in order to get rid of the 1e-300 kind of fluxes
73      if (std::abs(fluxOfCurrentReaction)<1e-6){fluxOfCurrentReaction=0;}
74      reaction currentReac= reactionVector[i-1];
75      std::vector<int> currentsubs=currentReac.getsubstrates();
76      std::vector<int> currentproducts=currentReac.getproducts();
77      int reacNR=currentReac.getListNr();
78      //adding the current reaction to a set, will be used to generate the node-type file
79      reacNumbers.insert(reacNR);
80      for (int sub:currentsubs){
81        std::string substrateName=substrateVector[sub+nrOfInternalMetabolites].niceSubstrateName();
82        compoundIDs.insert(sub);
83        //-1 here because that will be the ID of the reactions
84        edgeVector.emplace_back(sub+nrOfInternalMetabolites,fluxOfCurrentReaction,-1*reacNR);
85      }
86      for (int prod:currentproducts){
87        std::string productName=substrateVector[prod+nrOfInternalMetabolites].niceSubstrateName();
88        compoundIDs.insert(prod);
89        //-1 here because that will be the ID of the reactions
90        edgeVector.emplace_back(-1*reacNR,fluxOfCurrentReaction,prod+nrOfInternalMetabolites);
91      }
92    }
93    for (int i=0; i<nrOfInternalMetabolites; i++){
94      internalMetIDs.insert(i-nrOfInternalMetabolites);
95    }
96    for (auto sourceSubstrate_one:sourceSubstrate){
97      std::string
         ↪   sourceName=substrateVector[sourceSubstrate_one+nrOfInternalMetabolites].niceSubstrateName();
98      compoundIDs.erase(sourceSubstrate_one);
99      xgmmlFile<<"<node label=\""<<sourceName<<"\"
         ↪   id=\""<<sourceSubstrate_one+nrOfInternalMetabolites<<"\">"<<std::endl;
100     xgmmlFile<<"\t <att name=\"Type\" type=\"string\" value=\"Source\"/>"<<std::endl;
101     xgmmlFile<<"</node>"<<std::endl;
102   }
103   for (auto sinkSubstrate_one:sinkSubstrate){
104     std::string sinkName=substrateVector[sinkSubstrate_one+nrOfInternalMetabolites].niceSubstrateName();
105     compoundIDs.erase(sinkSubstrate_one);
106     xgmmlFile<<"<node label=\""<<sinkName<<"\"
         ↪   id=\""<<sinkSubstrate_one+nrOfInternalMetabolites<<"\">"<<std::endl;
107     xgmmlFile<<"\t <att name=\"Type\" type=\"string\" value=\"Sink\"/>"<<std::endl;
108     xgmmlFile<<"</node>"<<std::endl;
109   }
110   for (auto sinkSubstrate_one:additionalSinks){
111     std::string sinkName=substrateVector[sinkSubstrate_one+nrOfInternalMetabolites].niceSubstrateName();
112     compoundIDs.erase(sinkSubstrate_one);
113     xgmmlFile<<"<node label=\""<<sinkName<<"\"
         ↪   id=\""<<sinkSubstrate_one+nrOfInternalMetabolites<<"\">"<<std::endl;
114     xgmmlFile<<"\t <att name=\"Type\" type=\"string\" value=\"Sink\"/>"<<std::endl;
115     xgmmlFile<<"</node>"<<std::endl;
116   }
117   //looping through all the internalMet names writing them into the type file, removing them from the
       ↪   substrate list
118   while(!internalMetIDs.empty()){
119     compoundIDs.erase(*internalMetIDs.begin());
120     std::string currentName=substrateVector[*internalMetIDs.begin()+nrOfInternalMetabolites].niceSubstrat⌋
         ↪   eName();
121     xgmmlFile<<"<node label=\""<<currentName<<"\"
         ↪   id=\""<<*internalMetIDs.begin()+nrOfInternalMetabolites<<"\">"<<std::endl;
122     xgmmlFile<<"\t <att name=\"Type\" type=\"string\" value=\"InternalMet\"/>"<<std::endl;
123     xgmmlFile<<"</node>"<<std::endl;
124     internalMetIDs.erase(internalMetIDs.begin());
125   }
126   //doing the same with reacnubmers
127   while(!reacNumbers.empty()){
128     int currentReacNumber=*reacNumbers.begin();
```

```
129       xgmmlFile<<"<node label=\""<<currentReacNumber<<"\" id=\""<<-1*currentReacNumber<<"\">"<<std::endl;
130       xgmmlFile<<"\t <att name=\"Type\" type=\"string\" value=\"Reaction\"/>"<<std::endl;
131       xgmmlFile<<"</node>"<<std::endl;
132       reacNumbers.erase(reacNumbers.begin());
133     }
134     //now with the normal substrates
135     while(!compoundIDs.empty()){
136       std::string
            ↪   currentName=substrateVector[*compoundIDs.begin()+nrOfInternalMetabolites].niceSubstrateName();
137       xgmmlFile<<"<node label=\""<<currentName<<"\"
            ↪   id=\""<<*compoundIDs.begin()+nrOfInternalMetabolites<<"\">"<<std::endl;
138       xgmmlFile<<"\t <att name=\"Type\" type=\"string\" value=\"Compound\"/>"<<std::endl;
139       xgmmlFile<<"</node>"<<std::endl;
140       compoundIDs.erase(compoundIDs.begin());
141     }
142     for (std::tuple<int,double,int> currentEdge:edgeVector){
143       int source, sink;
144       double flux;
145       std::tie (source,flux,sink) = currentEdge;
146       xgmmlFile<<"<edge label=\""<<source<<" - "<<sink<<"\" source=\""<<source<<"\"
            ↪   target=\""<<sink<<"\">"<<std::endl;
147       xgmmlFile<<"\t <att name=\"flux\" type=\"real\" value=\""<<flux<<"\"/>"<<std::endl;
148       xgmmlFile<<"</edge>"<<std::endl;
149     }
150     xgmmlFile<<"</graph>"<<std::endl;
151     xgmmlFile.close();
152   }
153   std::vector<int> cell::canBeAdded(){
154     int nrOfAvailableReactions=availableReactions.size();
155     std::set<int> canAdd;
156     for (int i=0; i<nrOfAvailableReactions;i++){
157       std::vector<int> neighboursOfCurrentReac=reactionVector[availableReactions[i]-1].getNeighbours();
158       for (int j:neighboursOfCurrentReac){
159         canAdd.insert(j);
160       }
161     }
162     for (int i:availableReactions){
163       //getting rid of reactions that are already in the network
164       int alreadyInReacNr=reactionVector[i-1].getListNr();
165       canAdd.erase(alreadyInReacNr);
166     }
167     std::vector<int> tobereturned(canAdd.begin(),canAdd.end());
168     return tobereturned;
169   }
170   cell cell::mutateAndReturn( RandomGeneratorType& generator ){
171     //this function is for the population mutation. in this case there is no acceptance
172     //probability, every mutation is accepted, as survival of the fittest is dealt with
173     //in chosing the cells to reproduce
174     int deleteThisOne,addThisOne;
175     std::vector<int> trialNewCell = availableReactions;
176     double addProb=0.5;
177     double doWeAdd=randomRealInRange(generator, 1);
178     //calculating current throughput
179     bool areWeAdding= doWeAdd<=addProb;
180     if(areWeAdding){
181       std::vector<int> whatCanWeAdd = canBeAdded();
182       int whichOneToAdd=randomIntInRange(generator,whatCanWeAdd.size()-1);
183       addThisOne=whatCanWeAdd[whichOneToAdd];
184       trialNewCell.push_back(addThisOne+1);
185     }
186     else{
187       int whichOneToDel=randomIntInRange(generator,trialNewCell.size()-1);
188       deleteThisOne=whichOneToDel;
189       trialNewCell.erase(trialNewCell.begin()+deleteThisOne);
190     }
191     //not calculating flux just here
192     cell tryIfWorks(trialNewCell,0);
193     return tryIfWorks;
194   }
195   int cell::randomIntInRange(RandomGeneratorType& generator, int maxNumber){
```

```
196   Gen_Type intgenerator(generator, UniIntDistType(0,maxNumber));
197   boost::generator_iterator<Gen_Type> randomInt(&intgenerator);
198   int currentRandomNumber=intgenerator();
199   return currentRandomNumber;
200 }
201 double cell::randomRealInRange(RandomGeneratorType& generator, double maxNumber){
202   RealGenType realgenerator(generator, UniRealDistType(0,maxNumber));
203   double currentRandomNumber=realgenerator();
204   return currentRandomNumber;
205 }
206 void cell::mutatePopulation(std::vector<int>& population,std::vector<int>& howManyOfEach,
    ↪   std::vector<cell>& cellVector,  RandomGeneratorType& generator){
207   bool gotOneToMutate=false;
208   double maxPossibleFittness=10;
209   int whichOneToMutate;
210   bool areWeMutating=false;
211   bool areWeHorizontalTransferring=false;
212   bool wasThereAChange=false;
213   while(!gotOneToMutate){
214     //implementing the moran process selection
215     whichOneToMutate=cell::randomIntInRange(generator,population.size()-1);
216     double compareFittnesWithThis=cell::randomRealInRange(generator,maxPossibleFittness);
217     double fittnessOfCurrentCell=cellVector[population[whichOneToMutate]].getPerformance();
218     if(fittnessOfCurrentCell>compareFittnesWithThis){gotOneToMutate=true;}
219   }
220   int whichCellDies=cell::randomIntInRange(generator,population.size()-1);
221   //now figure out if we are mutating
222   double luckyToMutate=randomRealInRange(generator,1);
223   double luckyToHorizontalGene=randomRealInRange(generator,1);
224   if (luckyToMutate<cell::probabilityOfMutation) {
225     areWeMutating=true;
226   }
227   if (luckyToHorizontalGene<cell::probabilityOfHorizontalGenetransfer) {
228     areWeHorizontalTransferring=true;
229   }
230   //now mutate the selected cell, and put it in the place of the dying cell
231   std::vector<int> additionalSinks=cellVector[population[whichOneToMutate]].getAddSinks();
232   std::vector<int> originalReacs=cellVector[population[whichOneToMutate]].getReacs();
233   cell tmpCellWithoutPerfCalc(originalReacs,0);
234   if (areWeHorizontalTransferring){
235     //chosing the cell that transfers a reaction to our current cell
236     int whichCellDonatesGenes=randomIntInRange(generator,population.size()-1);
237     std::vector<int> GenesOfDonor=cellVector[population[whichCellDonatesGenes]].getReacs();
238     int whichReacToDonate=randomIntInRange(generator,GenesOfDonor.size()-1);
239     std::vector<int> GenesOfCurrent=tmpCellWithoutPerfCalc.getReacs();
240     //converting to unordered set to avoid double reactions
241     std::unordered_set<int> setOfGenesOfCurrent(GenesOfCurrent.begin(), GenesOfCurrent.end());
242     setOfGenesOfCurrent.insert(GenesOfDonor[whichReacToDonate]);
243     //if the donated reaction was already present in the current cell don't bother creating a new cell
244     if (setOfGenesOfCurrent.size() != GenesOfCurrent.size()){
245       //std::cout<<"There was a successful horziontal gene transfer."<<std::endl;
246       //converting back to vector for initialization of the new cell
247       std::vector<int> GenesWithTransferred(setOfGenesOfCurrent.begin(), setOfGenesOfCurrent.end());
248       tmpCellWithoutPerfCalc.setReacs(GenesWithTransferred);
249       wasThereAChange=true;
250     }
251   }
252   if (areWeMutating) {
253     //creating the mutatnt cell
254     cell offspringOfChosenCell=tmpCellWithoutPerfCalc.mutateAndReturn(generator);
255     tmpCellWithoutPerfCalc.setReacs(offspringOfChosenCell.getReacs());
256     wasThereAChange=true;
257   }
258   if(wasThereAChange){
259     //since there was a change now we'll have to recalculate the performance
260     //this is triggered by re-add the additional sinks (if any)
261     tmpCellWithoutPerfCalc.theseAreYourAddSinks(additionalSinks);
262     //now we need to find a place to store this mutatnt cell in cellVector
263     //decrease the number of those cells that die
264     --howManyOfEach[population[whichCellDies]];
```

```
265       //if the for loop doesn't find any unused places in cellVector, the statement after will exit with an
           ↪   error (this should never happen under normal circumstances)
266       int whichIsUnused=population.size();
267       for (int i=0; i<howManyOfEach.size();i++){
268         if (howManyOfEach[i]==0) {
269           whichIsUnused=i;
270           break;
271         }
272       }
273       //we have an unused element in cellVector now
274       cellVector[whichIsUnused]=tmpCellWithoutPerfCalc;
275       //create the population element that points to the newborn cell
276       population[whichCellDies]=whichIsUnused;
277       //increase the number corresponding to this newborn cell
278       ++howManyOfEach[whichIsUnused];
279     }
280     else{
281       //if there's only reproduction without mutation just change the pointer of the dying cell to the
           ↪   reproducing one and adjust the numbers in howManyOfEach
282       --howManyOfEach[population[whichCellDies]];
283       population[whichCellDies]=population[whichOneToMutate];
284       ++howManyOfEach[population[whichOneToMutate]];
285     }
286   }
287   std::vector<double> cell::getPopulationFittness(std::vector<int>& population, std::vector<cell>&
       ↪   cellVector){
288     std::vector<double> toReturn(population.size());
289     for (int k=0; k<population.size();k++){
290       toReturn[k]=cellVector[population[k]].getPerformance();
291     }
292     return toReturn;
293   }
294   void cell::printPopulationFittnesses(std::vector<int>& population, std::vector<cell>& cellVector){
295     std::vector<double> fittnesses=getPopulationFittness(population, cellVector);
296     for (auto i:fittnesses){
297       std::cout<<i<<", ";
298     }
299     std::cout<<std::endl;
300   }
301   cell cell::printNFittest(std::vector<int>& population, std::vector<cell>& cellVector,int N){
302     std::sort(population.begin(),population.end(),[&,cellVector,population](int first, int second) {return
         ↪   cellVector[first].getPerformance() > cellVector[second].getPerformance();});
303     std::cout<<"The best "<<N<<" are:";
304     for (int i=0;i<N;i++){
305       std::cout<<cellVector[population[i]].getPerformance()<<", ";
306     }
307     std::cout<<std::endl;
308     return cellVector[population[0]];
309   }
310   std::vector<cell> cell::getBestNCells(std::vector<int>& population, std::vector<cell>& cellVector, int N){
311     std::vector<cell> toReturn(N);
312     //doesn't matter if population is sorted even when the Moran process is running
313     //random selection doesn't care for ordered vector
314     //using inline lambda to do the sorting in the indexed population
315     std::sort(population.begin(),population.end(),[&,cellVector,population](int first, int second) {return
         ↪   cellVector[first].getPerformance() > cellVector[second].getPerformance();});
316     for (int i=0; i<N; i++){
317       toReturn[i]=cellVector[population[i]];
318     }
319     return toReturn;
320   }
321   double cell::calcThroughput(){
322     //first we find how many substrates actually participate in the network
323     std::vector<int> substrateIndex(substrateVector.size());
324     for(int i=0; i<nrOfInternalMetabolites; i++){
325       substrateIndex[i]=i+1;
326     }
327     //counter to keep track of which row of the GLPK problem the next substrate will belong to
328     int nextRowNumber=14;
329     std::set<int> substrateSet;
```

```
330     for (int i=0; i<availableReactions.size();i++){
331       reaction currentReac=reactionVector[availableReactions[i]-1];
332       for (int j:currentReac.getsubstrates()){
333         substrateSet.insert(j+nrOfInternalMetabolites);
334       }
335       for (int j:currentReac.getproducts()){
336         substrateSet.insert(j+nrOfInternalMetabolites);
337       }
338     }
339     //erasing internal metabolites from the set, as we already have those at the beginning of the list
340     for(int metab=0; metab<nrOfInternalMetabolites; metab++){substrateSet.erase(metab);}
341     //in order to always have the source and sink nodes
342     for(int sinkSubstrate_one:sinkSubstrate){
343       substrateSet.insert(sinkSubstrate_one+nrOfInternalMetabolites);
344     }
345     for(int sinkSubstrate_one:additionalSinks){
346       substrateSet.insert(sinkSubstrate_one+nrOfInternalMetabolites);
347     }
348     for(int sourceSubstrate_one:sourceSubstrate){
349       substrateSet.insert(sourceSubstrate_one+nrOfInternalMetabolites);
350     }
351     while(!substrateSet.empty()){
352       substrateIndex[*substrateSet.begin()]=nextRowNumber;
353       nextRowNumber++;
354       substrateSet.erase(substrateSet.begin());
355     }
356     int nrOfNormalSinks=sinkSubstrate.size();
357     int nrOfNormalSource=sourceSubstrate.size();
358     int nrOfAdditionalSinks=additionalSinks.size();
359     glp_prob *lp;
360     lp=glp_create_prob();
361     glp_set_prob_name(lp,"network_throughput");
362     glp_set_obj_dir(lp,GLP_MAX);
363     //silencing GLPK output
364     glp_term_out(GLP_OFF);
365     glp_add_rows(lp,nextRowNumber-1);
366     for (int i=1; i<nextRowNumber; i++){
367       //specifying that the rows must sum to zero (flux vector in the nullspace of S matrix)
368       glp_set_row_bnds(lp,i,GLP_FX,0.0,0.0);
369     }
370     //extra columns for the auxiliary reactions
371     int listSize=availableReactions.size();
372     glp_add_cols(lp,listSize+4+nrOfNormalSinks+nrOfNormalSource+nrOfAdditionalSinks);
373     std::vector<int> ia,ja;
374     std::vector<double> ar;
375     for (int i=1;i<=listSize;i++){
376       //preparing the sparse matrix's values
377       //final -1 because the availableReactions stores the id form the file
378       //starting with 1, as opposed to the vector starting at 0
379       reaction tmpreac=reactionVector[availableReactions[i-1]-1];
380       double freeChange=tmpreac.getCurrentFreeEChange();
381       if(freeChange<-10){
382         glp_set_col_bnds(lp,i,GLP_DB,0.0,1.0);
383       }
384       else if(freeChange>10){
385         glp_set_col_bnds(lp,i,GLP_DB,-1.0,0.0);
386       }
387       else {
388         glp_set_col_bnds(lp,i,GLP_DB,-0.5,0.5);
389       }
390       std::vector<int> tmpsubs=tmpreac.getsubstrates();
391       std::vector<int> tmpprods=tmpreac.getproducts();
392       for (int j: tmpsubs){
393         //using i+nrOfInternalMetabolites as the column numbering starts from 1
394         //and internalMets have negative substrate indices
395         int rownumber=substrateIndex[j+nrOfInternalMetabolites];
396         ia.push_back(rownumber);
397         ja.push_back(i);
398         ar.push_back(-1.0);
399       }
```

```
400       for (int j: tmpprods){
401         //using i+nrOfInternalMetabolites as the column numbering starts from 1
402         //and internalMets have negative substrate indices
403         int rownumber=substrateIndex[j+nrOfInternalMetabolites];
404         ia.push_back(rownumber);
405         ja.push_back(i);
406         ar.push_back(1.0);
407       }
408     }
409     //bounding the imaginary reactions, only certain amount of material can be taken at once
410     //water
411     glp_set_col_bnds(lp,listSize+1,GLP_DB,0.0,40.0);
412     //co2
413     glp_set_col_bnds(lp,listSize+2,GLP_DB,0.0,40.0);
414     //adp->atp auxilliary
415     glp_set_col_bnds(lp,listSize+3,GLP_DB,-10.0,40.0);
416     //nad_red->nad_ox
417     glp_set_col_bnds(lp,listSize+4,GLP_DB,-40.0,40.0);
418     //add imaginary reactions here:
419     //adding water
420     ia.push_back(substrateIndex[-1+nrOfInternalMetabolites]);  ja.push_back(listSize+1); ar.push_back(1.0);
421     //adding or removing co2
422     ia.push_back(substrateIndex[-2+nrOfInternalMetabolites]);  ja.push_back(listSize+2); ar.push_back(-1.0);
423     //adding ADP
424     ia.push_back(substrateIndex[-6+nrOfInternalMetabolites]);  ja.push_back(listSize+3); ar.push_back(1.0);
425     //removing ATP
426     ia.push_back(substrateIndex[-7+nrOfInternalMetabolites]);  ja.push_back(listSize+3); ar.push_back(-1.0);
427     //adding NadOx
428     ia.push_back(substrateIndex[-3+nrOfInternalMetabolites]);  ja.push_back(listSize+4); ar.push_back(1.0);
429     //removing Nad_red
430     ia.push_back(substrateIndex[-4+nrOfInternalMetabolites]);  ja.push_back(listSize+4); ar.push_back(-1.0);
431     //removing the sink substrates
432     for (int sinkIndex=0; sinkIndex<nrOfNormalSinks; ++sinkIndex){
433       ia.push_back(substrateIndex[sinkSubstrate[sinkIndex]+nrOfInternalMetabolites]);  ja.push_back(listSiz⌋
        ↪   e+4+sinkIndex+1);
        ↪   ar.push_back(-1.0);
434       //bounding the sinks
435       glp_set_col_bnds(lp,listSize+4+sinkIndex+1,GLP_DB,0.0,10.0);
436     }
437     //the additional sinks (if any)
438     for (int sinkIndex=0; sinkIndex<nrOfAdditionalSinks; ++sinkIndex){
439       ia.push_back(substrateIndex[additionalSinks[sinkIndex]+nrOfInternalMetabolites]);  ja.push_back(listS⌋
        ↪   ize+4+sinkIndex+1+nrOfNormalSinks);
        ↪   ar.push_back(-1.0);
440       //bounding the sinks
441       glp_set_col_bnds(lp,listSize+4+nrOfNormalSinks+sinkIndex+1,GLP_DB,0.0,10.0);
442     }
443     //adding source substrates
444     for (int sourceIndex=0; sourceIndex<nrOfNormalSource; ++sourceIndex){
445       ia.push_back(substrateIndex[sourceSubstrate[sourceIndex]+nrOfInternalMetabolites]);  ja.push_back(lis⌋
        ↪   tSize+4+nrOfNormalSinks+nrOfAdditionalSinks+sourceIndex+1);
        ↪   ar.push_back(1.0);
446       //bounding the sources
447       glp_set_col_bnds(lp,listSize+4+nrOfNormalSinks+nrOfAdditionalSinks+sourceIndex+1,GLP_DB,0.0,10.0);
448     }
449     //target is to maximize the auxiliary reactions throughput of the ADP->ATP reaction
450     //glp_set_obj_coef(lp,listSize+3,1.0);
451     //target is to maximize the auxiliary reaction removing pyruvate
452     glp_set_obj_coef(lp,listSize+5,1);
453     //creating the arrays now
454     int length=ia.size();
455     int iarray[length+1],jarray[length+1];
456     double ararray[length+1];
457     std::copy(ia.begin(),ia.end(),iarray+1);
458     std::copy(ja.begin(),ja.end(),jarray+1);
459     std::copy(ar.begin(),ar.end(),ararray+1);
460     glp_load_matrix(lp,length,iarray,jarray,ararray);
461     glp_simplex(lp,NULL);
462     double goodness=glp_get_obj_val(lp);
463     std::vector<double> tmpFluxes(availableReactions.size());
```

```
464     //storing the optimal fluxes
465     for (int i=0;i<availableReactions.size();i++){
466       tmpFluxes[i]=glp_get_col_prim(lp,i+1);
467       //  std::cout<<tmpFluxes[i]<<", ";
468     }
469     //WARNING! this is ALL the fluxes, also the ones for the auxilliary reactions
470     //care must be taken to extract the fluxes you actually want
471     setFluxes(tmpFluxes);
472     //need to delete glp object, otherwise memory consumption just increases without bounds
473     //it is not cleaned up automatically
474     glp_delete_prob(lp);
475     return goodness-smallKforFitness*availableReactions.size();
476   }
477   void cell::printHumanReadable(){
478     for (int i: availableReactions){
479       int id=reactionVector[i].getListNr();
480       std::vector<int> products = reactionVector[i].getproducts();
481       std::vector<int> substrates = reactionVector[i].getsubstrates();
482       std::cout<<id<<": ";
483       for (int j:substrates){
484         //figuring out whether we have a meaningful name
485         std::string toPrint=substrateVector[j].niceSubstrateName();
486         std::cout<<toPrint<<" + ";
487       }
488       std::cout<<" > ";
489       for (int j:products){
490         //figuring out whether we have a meaningful name
491         std::string toPrint=substrateVector[j].niceSubstrateName();
492         std::cout<<toPrint<<" + ";
493       }
494       std::cout<<std::endl;
495     }
496   }
497   void cell::setFluxes(std::vector<double>& fluxVector){
498     fluxThroughReacs=fluxVector;
499   }
500   void cell::findThePaths(std::vector<int> needMore, std::vector<int> needLess, std::vector<int>
      ↪   currentReactions, int TargetCompound, std::vector<reaction>& ReactionVector,
      ↪   std::vector<substrate>& SubstrateVector, std::string fnameString){
501     int writeoutcounter=1;
502     std::vector<int> doesntHaveToBalance = {-3, -4, -1, -2, 94};
503     doesntHaveToBalance.emplace_back(TargetCompound);
504     std::set<int> canWeAdd;
505     //first figure out what can be added
506     for (int i=0; i<currentReactions.size();i++)
507     {
508       std::vector<int> neighboursOfCurrentReac=ReactionVector[currentReactions[i]].getNeighbours();
509       for (int j:neighboursOfCurrentReac){
510         canWeAdd.insert(j);
511       }
512     }
513     for (int j:currentReactions){
514       canWeAdd.erase(j);
515     }
516     std::vector<int> PossibleReactionsToAdd(canWeAdd.begin(),canWeAdd.end());
517     for (int j:canWeAdd){
518       std::vector<int> substrates, products;
519       if (reactionVector[j].getCurrentFreeEChange()>0){
520         substrates=ReactionVector[j].getproducts();
521         products=ReactionVector[j].getsubstrates();
522       }
523       else{
524         substrates=ReactionVector[j].getsubstrates();
525         products=ReactionVector[j].getproducts();
526       }
527       bool doesItSolveANeed=false;
528       bool doesItSolveALess=false;
529       for (int currProduct:products){
530         doesItSolveANeed= doesItSolveANeed || (std::find(needMore.begin(),
            ↪   needMore.end(),currProduct)!=needMore.end());
```

```
531         if (std::find(needMore.begin(), needMore.end(), currProduct) != needMore.end()){
532           doesItSolveANeed=true;
533         }
534       }
535       for (int currSubs:substrates){
536         if (std::find(needLess.begin(), needLess.end(), currSubs) != needLess.end()){
537           doesItSolveALess=true;
538         }
539       }
540       //now if adding the reaction solves a need or a too much problem, we add it
541       if (doesItSolveALess || doesItSolveANeed){
542         std::vector<int> currentReactionsInLoop=currentReactions;
543         std::vector<int> needMoreInLoop=needMore;
544         std::vector<int> needLessInLoop=needLess;
545         currentReactionsInLoop.emplace_back(j);
546         // and we note the problems it creates, and solves
547         for (int noBalance:doesntHaveToBalance){
548           substrates.erase(std::remove(substrates.begin(), substrates.end(), noBalance), substrates.end());
549           products.erase(std::remove(products.begin(), products.end(), noBalance), products.end());
550         }
551         for (int k:substrates){
552           if (std::find(needLessInLoop.begin(), needLessInLoop.end(), k)!=needLessInLoop.end()){
553             needLessInLoop.erase(std::remove(needLessInLoop.begin(), needLessInLoop.end(), k),
                ↪   needLessInLoop.end());
554           }
555           else{
556             needMoreInLoop.emplace_back(k);
557           }
558         }
559         for (int k:products){
560           if (std::find(needMoreInLoop.begin(), needMoreInLoop.end(), k) != needMoreInLoop.end()){
561             needMoreInLoop.erase(std::remove(needMoreInLoop.begin(), needMoreInLoop.end(), k),
                ↪   needMoreInLoop.end());
562           }
563           else{needLessInLoop.emplace_back(k);
564           }
565         }
566         //now writing out the current state, and calling it again
567         int thingsInNeed=needMoreInLoop.size()+needLessInLoop.size();
568         if(thingsInNeed<2){
569           std::cout<<std::endl<<"Current state of the system:"<<std::endl;
570           std::cout<<"Reactions: ";
571           for (int k:currentReactionsInLoop){std::cout<<k<<", ";}
572           std::cout<<std::endl<<"Needmore: ";
573           for (int k:needMoreInLoop){std::cout<<k<<", ";}
574           std::cout<<std::endl<<"Needless: ";
575           for (int k:needLessInLoop){std::cout<<k<<", ";}
576           std::cout<<std::endl;
577           if (thingsInNeed<1){
578             std::ostringstream forFileName;
579             forFileName<<fnameString<<"/NR"<<writeoutcounter<<"cell";
580             cell tmpcell(currentReactionsInLoop);
581             tmpcell.printXGMML(forFileName.str());
582             writeoutcounter++;
583           }
584         }
585         if (currentReactionsInLoop.size()<7){
586           cell::findThePaths(needMoreInLoop,needLessInLoop, currentReactionsInLoop, TargetCompound,
                ↪   ReactionVector, SubstrateVector, fnameString);
587         }
588       }
589     }
590   }
591   void cell::printProgressFile(std::vector<int>& population, std::vector<cell>& cellVector,
       ↪   std::vector<int>& howManyOfEach,int k, int outerloop,const int generationsPerWriteout, int
       ↪   checkPointLength, std::ofstream& fileToWrite,double& previousAvgFittness, double maxFittQueue [],
       ↪   double avgFittQueue [], double entropyQueue [], int bestNetSizeQueue [], double avgNetSizeQueue [],
       ↪   int bestUsedReacsQueue [], double avgUsedReacsQueue []){
592     double maxFittness=0;
593     double totFittness=0;
```

```
594     int bestNetworkSize;
595     int totalNetworkSize=0;
596     double enthropy=0;
597     int bestUsedReacs=0;
598     int totUsedReacs=0;
599     //now runnign through the population to find the desired quantities
600     //only running through the vector containing how many of each cells there are in the population
601     //in a population containing many similar cells this can be much faster than running through each of the
          ↪  cells (in the population vector)
602     for (int i = 0; i < population.size(); ++i) {
603       int element=howManyOfEach[i];
604       if(element!=0){
605         enthropy+=element*std::log(element);
606         double currFittness=cellVector[i].getPerformance();
607         int currReactionSize=cellVector[i].getReacs().size();
608         int fluxcounter=0;
609         std::vector<double> currentFluxes=cellVector[i].getFluxes();
610         //the resizing is to make sure that we only count the nonzero real reactions not the auxilliary ones
611         currentFluxes.resize(currReactionSize);
612         for (double inspectedFlux:currentFluxes)
613         {
614           if (std::abs(inspectedFlux)>1e-5) {
615             ++fluxcounter;
616           }
617         }
618         totUsedReacs+=fluxcounter*element;
619         totFittness+=currFittness*element;
620         totalNetworkSize+=currReactionSize*element;
621         if (maxFittness<currFittness) {
622           maxFittness=currFittness;
623           bestNetworkSize=currReactionSize;
624           bestUsedReacs=fluxcounter;
625         }
626       }
627     }
628     int remainderOfK=k%generationsPerWriteout;
629     //now writing the required stuff in the file
630     if (remainderOfK==0){
631       double avgFittness=totFittness/(double)cellVector.size();
632       //if there was a large enough improvement since the last writeout, write out the steps inbetween
633       if (avgFittness>0.3+previousAvgFittness){
634         for (int i=1; i<generationsPerWriteout; ++i){
635           long long int number=k-generationsPerWriteout+i+outerloop*checkPointLength;
636           fileToWrite<<number<<" "<<maxFittQueue[i]<<" "<<entropyQueue[i]<<" "<<avgFittQueue[i]<<"
              ↪  "<<bestNetSizeQueue[i]<<" "<<avgNetSizeQueue[i]<<" "<<bestUsedReacsQueue[i]<<"
              ↪  "<<avgUsedReacsQueue[i]<<std::endl;
637         }
638       }
639       long long int generationNr=k+outerloop*checkPointLength;
640       fileToWrite<<generationNr<<" "<<maxFittness<<" "<<-1*enthropy<<" "<<avgFittness<<"
            ↪  "<<bestNetworkSize<<" "<<totalNetworkSize/(double)cellVector.size()<<" "<<bestUsedReacs<<"
            ↪  "<<totUsedReacs/(double)cellVector.size()<<std::endl;
641       //setting the previous avg network fittness to the current value
642       previousAvgFittness=avgFittness;
643     }
644     //Popoulating the next positions of the queue
645     maxFittQueue[remainderOfK]=maxFittness;
646     avgFittQueue[remainderOfK]=totFittness/cellVector.size();
647     entropyQueue[remainderOfK]=-1*enthropy;
648     bestNetSizeQueue[remainderOfK]=bestNetworkSize;
649     avgNetSizeQueue[remainderOfK]=totalNetworkSize/(double)cellVector.size();
650     bestUsedReacsQueue[remainderOfK]=bestUsedReacs;
651     avgUsedReacsQueue[remainderOfK]=totUsedReacs/(double)cellVector.size();
652   }
653   void cell::setReacs(std::vector<int> theseAreYourNewReacs){
654     //Note: no performance recalc was done here. We do that after all the mutations are done when re-adding
          ↪  the additional sinks
655     availableReactions=theseAreYourNewReacs;
656   }
```

Appendix D. reaction.h

```
#pragma once
// header file for the reaction class
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <cstdlib>
#include <boost/graph/bipartite.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/uniform_int.hpp>
//for the glpk library
#include "stdio.h"
#include "stdlib.h"
#include "glpk.h"
#include <string>
#include <iostream>
#include <fstream>
#include <utility>
#include <algorithm>
#include <array>
typedef boost::mt19937 RandomGeneratorType;
typedef boost::uniform_int<> UniIntDistType;
typedef boost::variate_generator<RandomGeneratorType&, UniIntDistType> Gen_Type;
typedef boost::uniform_real<> UniRealDistType;
typedef boost::variate_generator<RandomGeneratorType&, UniRealDistType> RealGenType;
struct environment
{
  double nh2acceptorCont=1;
  double nh2donorCont=1;
  double conh22Cont=1;
  double nh3aqCont=1;
  double ppiCont=1;
  double piCont=1;
  double atpCont=1;
  double adpCont=1;
  double ampCont=1;
  double nadredcont=1;
  double nadoxcont=1;
  double co2cont=1;
  double h2ocont=1;
  double temperature=293; //temperature in kelvins
  double glutCont=1;
  double oxo2Cont=1;
};
class reaction;
class substrate
{
  //properties of each substrate
  int index;
  double freeOfCreation;
  std::string molecule;
  std::string name;
  int charge;
  std::vector<int> involvedInReacs;
  public:
  substrate(int index, double freeOfCreation,std::string molecule, std::string name, int charge);
  substrate();
  inline std::vector<int> getinvolved() { return involvedInReacs;}
  void addInvolved(int addThisReac);
  static void buildNeighbourList(std::vector<reaction>& reacList, std::vector<substrate>&
   ↪   substrateVector);
  void printInvolved();
  std::string niceSubstrateName();
};
typedef std::array<int, 13> InternalMetsT;
class reaction
```

```
{
  int listNR;
  std::vector<int> substrates;
  std::vector<int> products;
  InternalMetsT internalMets = {};
  double freeEChange = 0.0;
  double currentFreeEChange=0;
  std::vector<int> neighbourOf;
  public: static int nrOfInternalMetabolites;
public:
  reaction(int listNR,double tmpfreechange, const std::vector<int>& tmpsubstrates, const
   ↪   std::vector<int>& tmproducts, const InternalMetsT& tmpInternalMets);
  reaction();
  static void readReactions(std::string fileName, std::vector<reaction>& reacPointer,
   ↪   std::vector<substrate>& substrateVector);
  static void readCompounds(std::string fileName,  std::vector<substrate>& substrateVector);
  void printReaction();
  double freeEchange();
  void recalcEchange(const environment& env);
  void printNeighbours();
  // inlining the return of big vectors may help the compiler
  // to get rid of some needless copying
  inline std::vector<int> getsubstrates() { return substrates; }
  inline std::vector<int> getproducts() { return products; }
  inline std::vector<int> getNeighbours() { return neighbourOf; }
  inline double getCurrentFreeEChange() {return currentFreeEChange;}
  int getListNr();
  void setNeighbours( std::vector<int>& neighbourList);
};
```

Appendix E. reaction.cpp

```
#include "reaction.h"
int reaction::nrOfInternalMetabolites;
  reaction::reaction(int reacNr,double tmpfreechange, const std::vector<int>& tmpsubstrates, const
   ↪   std::vector<int>& tmproducts, const InternalMetsT& tmpInternalMets)
  : listNR(reacNr)
  , substrates(tmpsubstrates)
  , products(tmproducts)
  , internalMets(tmpInternalMets)
    , freeEChange(tmpfreechange)
{}
reaction::reaction() {}
void reaction::printReaction()
{
  std::cout<<"From: ";
  for(auto i =substrates.begin(); i!=substrates.end(); ++i) std::cout<<*i<<' ';
  std::cout<<std::endl;
  std::cout<<"To: ";
  for(auto i=products.begin(); i!=products.end(); ++i) std::cout<<*i<<' ';
  std::cout<<std::endl;
  std::cout<<"standard conditions free energy change: "<<freeEChange<<std::endl;
}
void reaction::readCompounds(std::string fileName,  std::vector<substrate>& substrateVector)
{
  std::ifstream inFile(fileName);
  std::string line, nameChem, namenorm;
  if(!inFile)
  {
    throw std::runtime_error("Can't open input file " + fileName);
  }
  while(std::getline(inFile, line))
  {
    int tmpindex, tmpcharge;
    double tmpfreeEofcreation;
    std::string tmpname, tmpmolecule;
```

```
34      std::stringstream iss(line);
35      iss>>tmpindex;
36      iss>>tmpfreeEofcreation;
37      iss>>tmpmolecule;
38      iss>>tmpname;
39      iss>>tmpcharge;
40      //creating substrate and putting it into the vector
41      substrate tmpsubstrate(tmpindex,tmpfreeEofcreation,tmpmolecule,tmpname, tmpcharge);
42      substrateVector.emplace_back(tmpsubstrate);
43    }
44  }
45  int reaction::getListNr(){ return listNR;}
46  void reaction::readReactions(std::string fileName, std::vector<reaction>& reacPointer,
     ↪   std::vector<substrate>& substrateVector)
47  {
48    std::ifstream inFile(fileName);
49    std::string line,tmpsubs,tmpprods;
50    double tmpfreeE;
51    if(!inFile)
52    {
53      throw std::runtime_error("Can't open input file " + fileName);
54    }
55    int counter=0;
56    while(std::getline(inFile, line))
57    {
58      InternalMetsT tmpinternalMets = {};
59      std::vector<int> tmpsubstrates, tmproducts;
60      std::stringstream iss(line);
61      iss>>tmpfreeE;
62      std::getline(iss,tmpsubs, '>');
63      std::getline(iss,tmpprods);
64      std::istringstream subss(tmpsubs);
65      std::istringstream prodss(tmpprods);
66      for(int tmp; subss>>tmp;)
67      {
68        tmpsubstrates.push_back(tmp);
69      }
70      for(int tmp; prodss>>tmp;)
71      {
72        tmproducts.push_back(tmp);
73      }
74      //finding if any of the internal metabolites appear on any side of the reaction
75      for(int i=-nrOfInternalMetabolites; i<0; i++)
76      {
77        if(std::find(tmpsubstrates.begin(), tmpsubstrates.end(), i) != tmpsubstrates.end())
78        {
79          tmpinternalMets[i+nrOfInternalMetabolites]--;
80        }
81        if(std::find(tmproducts.begin(), tmproducts.end(), i) != tmproducts.end())
82        {
83          tmpinternalMets[i+nrOfInternalMetabolites]++;
84        }
85      }
86      reacPointer.emplace_back(counter,tmpfreeE, tmpsubstrates, tmproducts, tmpinternalMets);
87      for(int i : tmpsubstrates)
88      {
89        if(i>=0 ){
90          substrateVector[i+nrOfInternalMetabolites].addInvolved(counter);
91        }
92      }
93      for(int j: tmproducts)
94      {
95        if(j>=0 ){
96          substrateVector[j+nrOfInternalMetabolites].addInvolved(counter);
97        }
98      }
99      counter++;
100   }
101 }
102 double reaction::freeEchange() { return freeEChange; }
```

```
103 void reaction::recalcEchange(const environment& env)
104 {
105   //recalculate the freeEchang using the formula G=G0+RT*ln(([C]^c*[D]^d)/([A]^a*[B]^b))
106   //need the reactions database changed for this
107   double insideLog=(std::pow(env.nh2acceptorCont,internalMets[0])*std::pow(env.nh2donorCont,internalMets[
       ↪   1])*std::pow(env.conh22Cont,internalMets[2])*std::pow(env.nh3aqCont,internalMets[3])*std::pow(env
       ↪   .ppiCont,internalMets[4])*std::pow(env.piCont,internalMets[5])*std::pow(env.atpCont,internalMets[
       ↪   6])*std::pow(env.adpCont,internalMets[7])*std::pow(env.ampCont,internalMets[8])*std::pow(env.nadr
       ↪   edcont,internalMets[9])*std::pow(env.nadoxcont,internalMets[10])*std::pow(env.co2cont,internalMet
       ↪   s[11])*std::pow(env.h2ocont,internalMets[12]));
108   currentFreeEChange=freeEChange+8.3144598e-3*env.temperature*std::log(insideLog);
109 }
110 void reaction::setNeighbours(std::vector<int>& neighbourList){
111   neighbourOf=neighbourList;
112 }
113 void reaction::printNeighbours(){
114   std::cout<<listNR<<" has neighbours: ";
115   for (const int & i:neighbourOf){
116     std::cout<<i<<", ";
117   }
118   std::cout<<std::endl;
119 }
```

Appendix F. substrate.cpp

```
1  #include "reaction.h"
2    substrate::substrate(int tmpindex, double tmpfreeOfCreation, std::string tmpmolecule, std::string
       ↪   tmpname, int tmpcharge)
3    : index(tmpindex)
4    , freeOfCreation(tmpfreeOfCreation)
5    , molecule(tmpmolecule)
6    , name (tmpname)
7      , charge(tmpcharge)
8  {std::vector<int> involvedInReacs;}
9  substrate::substrate() {}
10 void substrate::addInvolved(int addThisReac){
11   int tmpadd=addThisReac;
12   involvedInReacs.emplace_back(tmpadd);
13 }
14 void substrate::printInvolved(){
15   std::cout<<niceSubstrateName()<<" is involved in: ";
16   for (int i: involvedInReacs){
17     std::cout<<i<<", ";
18   }
19   std::cout<<std::endl;
20 }
21 void substrate::buildNeighbourList(std::vector<reaction>& reacList, std::vector<substrate>&
       ↪   substrateVector){
22   for (reaction& currentReac : reacList){
23     std::set<int> neighbourSet;
24     std::vector<int> currSubs=currentReac.getsubstrates();
25     std::vector<int> currProds=currentReac.getproducts();
26     for (int i:currSubs)
27     {
28       std::vector<int> involvedIn=substrateVector[i+reaction::nrOfInternalMetabolites].getinvolved();
29       for (int j:involvedIn){
30         neighbourSet.insert(j);
31       }
32     }
33     for (int i:currProds)
34     {
35       std::vector<int> involvedIn=substrateVector[i+reaction::nrOfInternalMetabolites].getinvolved();
36       for (int j:involvedIn){
37         neighbourSet.insert(j);
38       }
39     }
```

```
40      std::vector<int> toBeNeighbours(neighbourSet.begin(), neighbourSet.end());
41      currentReac.setNeighbours(toBeNeighbours);
42    }
43  }
44  std::string substrate::niceSubstrateName(){
45    std::string emptyName=("---");
46    std::string tobereturned;
47    if (name.compare(emptyName) ==0)   {tobereturned=molecule;}
48    else   {tobereturned=name;}
49    return tobereturned;
50  }
```