Simulating the evolution of the ability to metabolize a novel nutrient-source

CSE 561 Final project report

by

Vivin Suresh Paliath

Abstract: *In 2008 Richard Lenski's long-term E. Coli evolution-experiment yielded an important result: a particular strain of the bacterium was able to metabolize citrate under aerobic conditions. This is an ability not found in wild-type E. Coli and hence marks the evolution of a novel ability; one that was observed under controlled conditions for the very first time. In this paper I describe an agent-based model in DEVSJAVA that uses artificial evolution in an attempt to replicate Lenski's results, albeit in a radically-simplified form. I will also describe and explain the experiments conducted and the results observed through the simulation of this model.*

Contents

List of Figures

List of Tables

List of Algorithms

1 Background

In the natural world, evolution is driven by two forces: mutation and natural selection[1]. Mutation is a process by which random changes are introduced into an organism's genome due to transcription errors when DNA (or RNA) is copied. The resulting changes in the genotype can have a significant impact on the organism's phenotype (observed characteristics, morphology, development etc.). Natural selection is a process which gradually causes certain traits to become more or less common based on selective pressures in the environment. In this manner, mutations that cause an organism to display traits that cause it to be more successful in its environment will be selected for and passed onto offspring, whereas less-favorable traits will be slowly weeded out.[1][2]

In 2008, Dr. Richard Lenski and his collaborators reported that their long-term *E. Coli* evolution-experiment yielded a strain of the bacterium which displayed the ability to metabolize citrate under aerobic conditions[3]. The metabolysis of citrate is not an ability found in wild-type *E. Coli* and hence is the evolution of a novel mutation. Although this ability has been observed in other organisms, such as a strain of *Flavobacterium* that is capable of digesting certain byproducts of nylon manufacturing[4], this was the first time that the evolution of a novel ability was observed in controlled conditions.

2 Problem Description

My project involves replicating Lenski's finding (on a vastly-simpler scale) through artificial-evolution, using an agent-based model developed in DEVSJAVA. One feature of agent-based models is "emergent complexity"; i.e., complex behavior that arises from simple rules and conditions. I believe that this feature can be exploited to simulate complex-behavior like the evolution of novel abilities among a population of agents.

My hypothesis is that my agent-based model will also display the behavior observed by Lenski among his population of *E. coli* bacteria: the evolution of the ability to metabolize a novel nutrient-source.

3 Approach

In general, the model is an abstraction of a petri-dish with two nutrient-sources, and a population of bacteria. Of the two nutrient-sources, the bacteria are only able to metabolize one. The bacteria metabolize the nutrient source to gain energy for their metabolic processes. If at some point, the bacteria have enough energy, they will be able to reproduce asexually through mitosis. It is at this stage that changes to the bacterium's genetic code can occur, since transcription is not an error-free process. These changes are completely random, but can have a significant effect on the properties of the bacterium. Furthermore, an innocuous change can have a significant impact on the survival of the bacterium. For example, the bacterium may become more efficient at metabolizing nutrients, or may have a longer lifespan. These changes, combined with the selective pressures of the environment that the bacteria live in, will allow certain strains to be more successful than others. If a transcription error causes a bacterium to have a change that causes it to be, for example, less efficient at metabolizing a nutrient, or lowers its lifespan, then it has a greater chance of dying out, which means that the particular strain does not survive. In this way, selective pressures of the environment decide which strains of the artificial bacteria are more successful than others.

The model neither attempts to faithfully replicate or simulate the actual metabolic-processes inside an *E. Coli* bacterium, nor the actual chemical-processes involved in the metabolysis of nutrients like citrate or glucose. The reason is that such a model would be prohibitively complex to implement given the time constraints for this class. Hence, I have made suitable abstractions that preserve the spirit of the experiment without being overly complex.

In succeeding sections I will go over the individual components of the model in detail, covering the abstractions made and the rationale for those abstractions.

3.1 Model Development

As mentioned before, bacteria in this model will be living inside an abstraction of a petri-dish. The petri-dish itself is represented as a toroidal grid (i.e., cells at edge wrap around). Hence the basic unit of space in this model is the cell. Each cell has eight neighbors: north, south, east, west, northeast, northwest, southeast, and southwest. This kind of neighborhood is called a *Moore* neighborhood (versus the alternative, which is the *von Neumann* neighborhood). Both neighborhoods are shown in Figure 1.

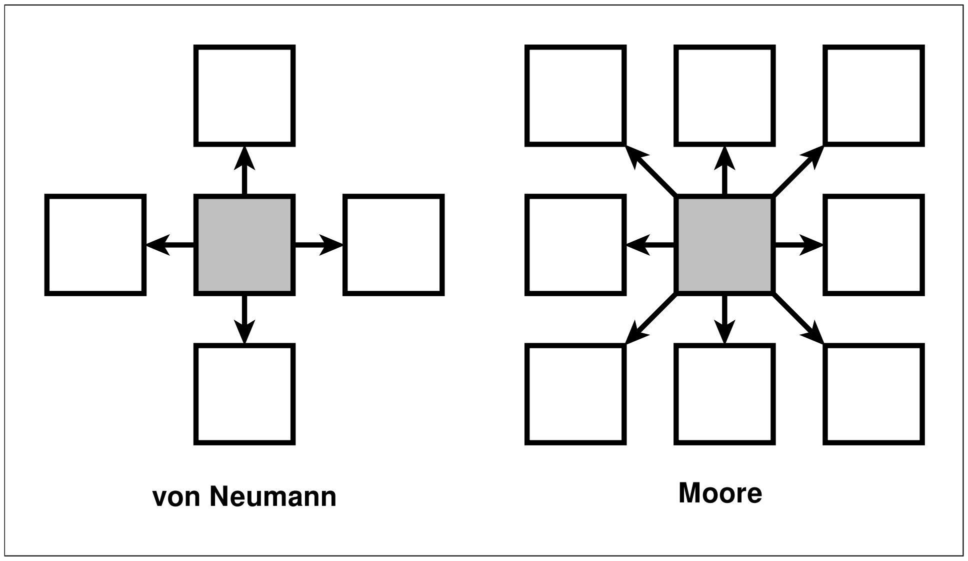


Figure 1: von Neumann vs. Moore neighborhoods

Since the basic unit of space is a cell, this agent-based model can also be considered to be a cellular-automata-like model. Each cell will hold an inexhaustible nutrient-source and can hold at most one agent at a time. My reason for making the nutrient-source inexhaustible is for two reasons: *a)* the focus of this model is on the evolution of novel abilities to metabolize new nutrients and *b)* keeping track of exhausted nutrients and then replenishing them would make the model a little more complicated.

Cells will be connected to each other in a *Moore* network. Additionally, each cell will be connected to a transducer as well. The transducer keeps track of the status of each cell and of the grid as a whole. The cell itself will implement the agent's (bacterium's) lifecycle and send updates to the transducer. Another important function of the transducer is to respond to queries from cells regarding vacancies in neighboring cells. When an agent in a cell wishes to reproduce, the cell can query the transducer about vacancies in neighboring cells. The transducer can then "reserve" that cell for the requesting cell. When the requesting cell's agent has finished reproducing, the child agent can then be sent to the reserved cell. Since multiple cells could be making requests at the same time, the transducer will not ignore requests, but will queue them up instead.

A diagram of the model can be seen in Figure 2.

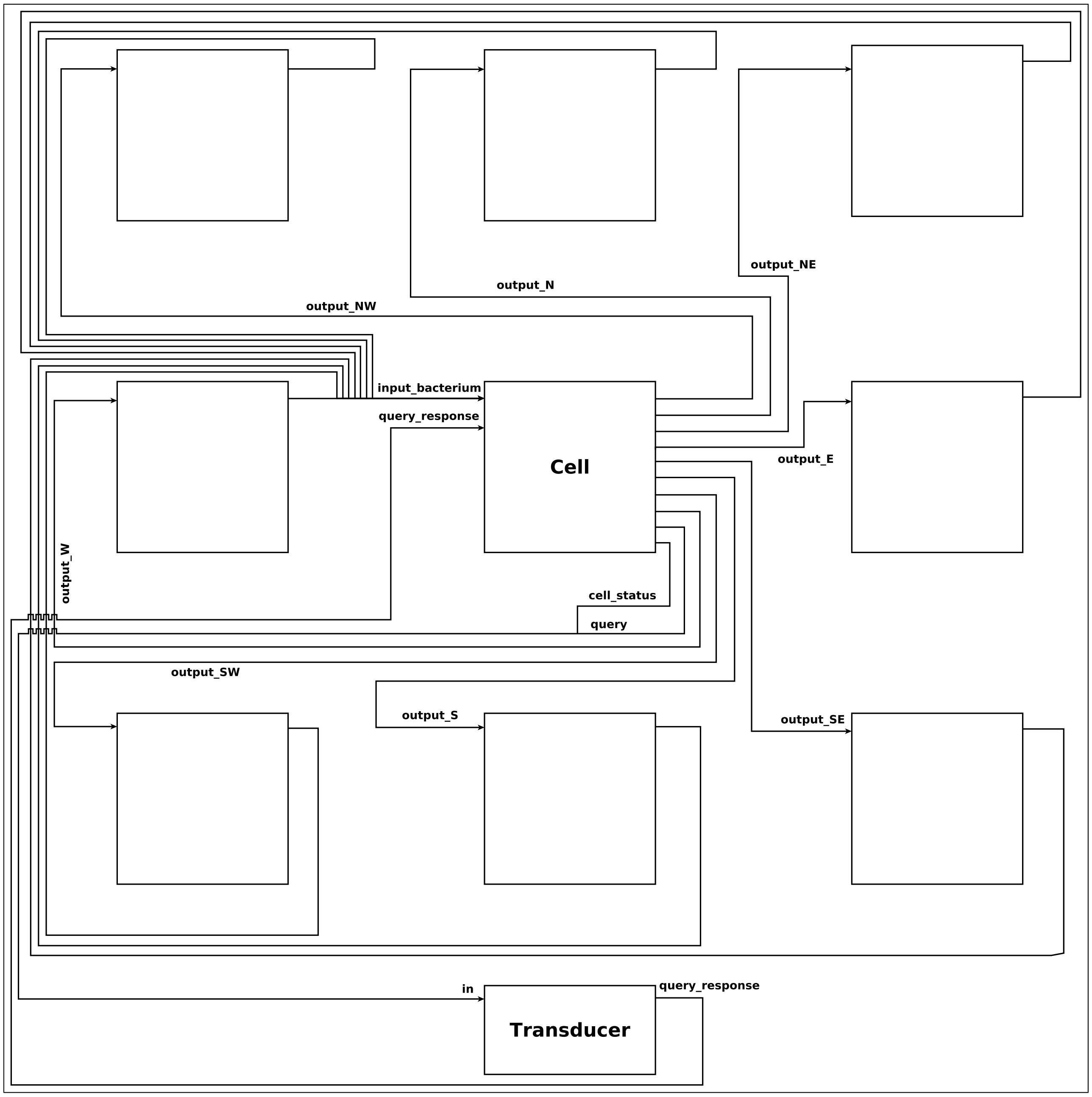


Figure 2: Model showing one cell connected to its neighbors and the transducer

Notice that the cell has inputs and outputs connecting it to its neighbors, as well as to the transducer. Additionally, the cell can also receive inputs from the transducer as well as its neighbors. The transducer itself has a single input and a single output. However, the input is connected to the `query` output port of every cell, and the output is connected to the `query_response` input port of every cell.

3.1.1 Cell Model

The main participant in this agent-based model is the bacterial agent, which is an abstraction of the *E. coli* bacterium. Before we look at the cell model itself, it is helpful to learn some details about the agent itself. The bacterial agent has a very simple life-cycle, which involves the following phases:

- **Feed**: During this phase, the agent is feeding. This phase involves metabolizing nutrients. When an agent metabolizes a nutrient, it gains energy (G_n).

- **Rest**: During this phase, the agent is not involved in any activity, but simply uses energy necessary to maintain its metabolic processes.
- **Reproduce**: During this phase, the agent reproduces asexually, and splits into two agents.
- **Die**: During this phase, the agent is dead.

The transitions between these states depend on various properties of the agent. The properties of an agent are summed up as follows:

- **Lifespan** (L): Each agent has a lifespan. The lifespan of the agent determines how long it lives. One unit of the lifespan corresponds to one iteration of the agent's life-cycle. When the lifespan of the agent reaches zero, the agent will die.
- **(Initial) free energy** (G): Each agent will be instantiated with some amount of initial free-energy that it can use for its metabolic processes. The amount of energy can be increased through feeding. If the amount of energy ever reaches zero, the agent dies.
- **Reproduction energy-threshold** (G_r): When an agent's energy level exceeds this threshold, it means that it has enough energy to reproduce asexually. The amount of energy used during reproduction is $0.5G_r$, or half of the reproduction energy-threshold.
- **Metabolic energy used** (G_m): This is the energy used by the agent in the resting-phase to maintain its metabolic processes. The actual amount of energy used by the agent is also dependent on the number of enzymes its genome supports. Therefore, the total amount of energy used by an agent during its resting-phase is $N_e \times G_m$.
- **Enzymes**: An agent has a one or more enzymes which it can use to metabolize nutrients in the cell.

Using the information above, we can represent the lifecycle of the agent diagrammatically (see Figure 3).

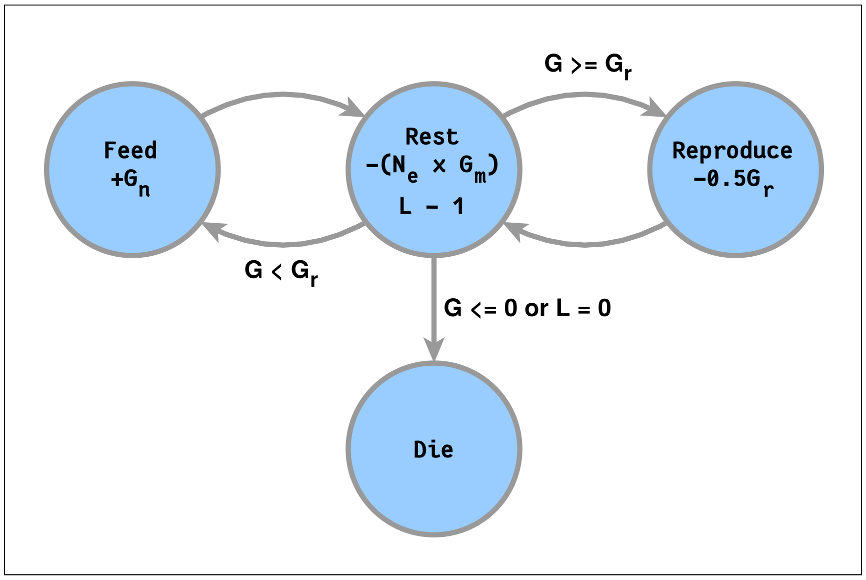


Figure 3: Life-cycle of the bacterial agent

The **Reproduce** and **Feed** states (and associated behaviors) are the most important. At the heart of any artificial-evolution model lies some mechanism to introduce random mutations. In this model, this occurs in the **Reproduce** state. In addition, artificial-evolution models have some sort of fitness function that determines how successful (with regard to some goal) a particular mutation is. In this model, the fitness function involves the metabolysis of a nutrient by an agent, using the enzymes in its genome. It is important to note that I do not explicitly select for any offspring using the fitness function. The point of the model is to see whether some sort of natural selection occurs, since it is only by having efficient enzymes (for a nutrient) that an agent can accumulate enough energy to produce offspring. Now let us take a look at both of these states in some detail so that it is clear exactly how reproduction and nutrient-metabolysis happens.

Reproduction

When an agent gains enough energy such that its free-energy is over the reproduction threshold ($G \geqslant G_r$), it can reproduce asexually into two agents. Reproduction involves the copying of the parent's genome into the child. The parent's genome is represented as a bit pattern which encodes the properties described earlier (lifespan, initial free-energy, etc.). In addition, the genome also encodes the enzymes that the agent has at its disposal to metabolize nutrients.

The genome of an agent has the following format:

$$\overbrace{0000\ 0111}^{L}\ \overbrace{0000\ 0011\ 1011\ 0001}^{G}\ \overbrace{0000\ 1100\ 1011\ 0001}^{G_r}\ \overbrace{0000\ 1100}^{G_m}\ \overbrace{0110\ 0101\ 1000\ 1110}^{\substack{\text{First}\\ \text{"enzyme"}}}\ \overbrace{1100\ldots 1000}^{\substack{\text{Second}\\ \text{"enzyme"}}}$$

Note that even though two enzymes are displayed above in the genome, it does not imply that every agent will always have two enzymes. Depending on the number of bits, an agent can have any number of enzymes. Each 16-bit "chunk" defined in the genome after G_m codes for an enzyme. This implies that it is possible to have "partial" enzymes as well. How these enzymes impact metabolysis is explained later.

Transcription of the parent's genome into the child is not an error-free process and will include transcription errors. This mimics transcription-errors that occur naturally, when DNA is transcribed. There are four types of transcription errors that can occur:

- **Deletion**: Deletion is when one or more consecutive-bits are not copied over from the parent to the child's genome.
- **Repetition**: Repetition is when a bit from the parent is repeated one or more times in the child's genome.
- **Inversion**: Inversion is when one or more consecutive-bits from the parent's genome are inverted and then copied into the child's genome.
- **Insertion**: Insertion is when one or more random-bits are inserted into the child's genome.

By including these types of errors, I am able to introduce random changes into the genomes of the agents as they reproduce. These changes, coupled with selective pressures, should decide which changes are beneficial and which are deleterious. Over time, beneficial changes should persist and win out over others. Currently the rates of mutation are hardcoded into the model as follows: there is an overall 1% mutation rate. If there is a mutation, there is an equal chance that it can be any one of the types described above (deletion, repetition, inversion, or insertion). A maximum of 4 bits can be mutated at a time, and this number is calculated at random with the following frequencies:

- 55% of the time, 1 bit is mutated.
- 30% of the times, 2 bits are mutated.
- 10% of the time, 3 bits are mutated.

- 5% of the time, 4 bits are mutated.

Feeding

An agent gathers energy through feeding, which involves the metabolysis of a nutrient source. In the real world, *E. Coli* metabolize glucose through a process known as glycolysis[5]. Furthermore, there are numerous other factors that affect the efficiency of metabolysis of a nutrient in general, such as the structure and nature of the enzyme, the energy content of the nutrient, and terminal electron-acceptors. As mentioned before, it is not feasible to model or replicate all of these factors. So what would be a good way abstract this complex process? The main question regarding an enzyme is to see how efficient it is at metabolizing a nutrient. Therefore, the two things we care about are the total energy-content of the nutrient, and how much of that energy an enzyme can successfully extract. For this model, I use a scheme involving binary-strings that addresses both these concerns nicely.

Both nutrients and enzymes are represented by 16-bit binary numbers. The base-10 value of the 16-bit number that represents a nutrient, is also the energy-content of the nutrient, and represents the maximum energy that can be extracted from it by an enzyme. Hence, a nutrient represented by the 16-bit value 0111101010001111_2 has an energy content of $31,375$ joules.

Metabolysis of a nutrient is performed by an enzyme, which is also a 16-bit value. The actual metabolic process is represented by the XOR operation. The result of the XOR operation is then used to calculate the percentage of the total energy-content of the nutrient, that can be extracted by the enzyme. In this manner, I am able to take into account the enzyme's efficiency at metabolizing a nutrient. The entire process is best described through an example:

Assume we have a nutrient 0111101010001111_2 and an enzyme 0110011110011100_2. The result of XORing the two values is 0001110100010011_2. The efficiency of the enzyme is then defined as the percentage of ones in the resulting 16-bit binary-value. In this particular case, we are left with 7 ones. Since the maximum-possible number of ones in a 16-bit value is 16, the efficiency of the enzyme is 43.75%. This means that the enzyme is able to extract 43.75% of $31,375$ joules, or approximately $13,727$ joules from the nutrient. It follows thus, that the most-efficient enzyme for any particular nutrient, is its bitwise inverse.

From the genome structure that I have described, we can see that it is possible for an agent to have more than one enzyme. If so, which enzyme will the agent end up using? In that case, the agent will use the enzyme that can extract the highest amount of energy out of the nutrient.

Another thing to consider is the case of "partial" enzymes. Due to the nature of the transcription process, it is possible to end up with enzymes that do not have all 16 bits. In this case, the XOR operation proceeds as normal, but will only use the same number of bits from the nutrient as the number of bits in the enzyme, starting with the least-significant bit. In the result, the unmatched bits are set to 0 as well. For example, assuming we have the nutrient 0111101010001111_2 and the enzyme 10011100_2, the XOR operation is only performed between 10001111_2 and 10011100_2. This gives us a result of 00010011_2 and thus an efficiency of only 18.75%.

Within the model, the bacterial agent is realized as a class, which implements all of the behaviors described above.

Having looked at the agent, let us now focus on the cell model itself. While the cell itself does not implement any of the behaviors of the bacterial agent, it does maintain and transition through the states that represent the agent's lifecycle. The cell does this based on the current state of the agent, any inputs it receives, and on certain outputs that it sends out. These states that the cell goes through are its primary states, and are as follows:

- **`FEED`**: In this state, the cell invokes the "feed" behavior on the agent occupying it.
- **`SEND_FEED_MESSAGE`**: This state is used by the cell to send results to the transducer regarding the results of the agent's feeding action. I do this because I am interested in how well an agent metabolizes a nutrient and therefore I would like to send this information to the transducer.
- **`REST`**: In this state, the cell invokes the "rest" behavior on the agent.
- **`SEND_REST_MESSAGE`**: This state is used by the cell to send results of the resting behavior to the transducer. It is during this phase that an agent's lifespan is reduced by one, and that it uses up some of its own energy for metabolic processes. If the lifespan ever reaches zero, or if the current energy is lesser than zero, the agent is dead. The transducer can use this information to clear out an occupied cell.
- **`QUERY`**: In this state, the cell queries the transducer to see if any of the surrounding cells are empty. The cell enters this state only if the occupying agent's energy is above the reproductive threshold.
- **`WAIT`**: In this state, the cell waits for a response from the transducer in regard to its query.
- **`REPRODUCE`**: In this state, the cell invokes the "reproduce" behavior on the agent. The cell will transition to this state only in response to a successful query from the transducer (i.e., there is enough room available for a child).
- **`SEND_CHILD`**: In this state, the cell sends a child agent to a vacant neighbor.
- **`DEAD`**: In this state, the cell's occupying agent is dead which means that the cell is unoccupied. This is a passive state. All cells initially start in this state until they are told to "start" by an input to the coupled model. If a cell does not have an occupying agent, it will remain in the **`DEAD`** state.

The cell maintains other necessary information as well. These are its secondary states, and are as follows:

- **`row`**: This is the row that the cell is on, in the toroidal grid that represents the petri-dish.
- **`column`**: This is the column that the cell is on, in the toroidal grid that represents the petri-dish.
- **`nutrient`**: The nutrient source that is in this cell.
- **`bacterium`**: The bacterial agent that currently occupies this cell.
- **`metabolysisResult`**: The result of a feeding behavior performed by the occupying bacterial-agent, which contains information such as the enzyme used for nutrient-metabolysis and the efficiency of said enzyme (for that nutrient).
- **`queryResponse`**: The query response from the transducer, which contains information regarding the address (row and column) of a vacant cell. This state also lets the cell know if no vacant neighbors are available.
- **`childGenome`**: The genome of the child agent. This state is a result of the agent reproducing. The child genome is what is sent to vacant neighbors.

The cell also needs to communicate with its neighbors and the transducers. It does this through the following ports:

- `START` **(inbound)**: This is an input from the coupled model that tells a cell to start implementing the agent's lifecycle. The cell will only do so if there is an occupying agent. Otherwise, it remains in the `DEAD` state.
- `INPUT_BACTERIUM` **(inbound)**: This is an input that can come from any of the cell's neighbors. It will contain the genome of the offspring from the sending cell's occupying agent.
- `QUERY_RESPONSE` **(inbound)**: This is an input that comes from the transducer, in response to a cells query about vacancies in any of its neighbors.
- `QUERY` **(outbound)**: This is the port the cell uses to query the transducer about vacancies in its neighbors.
- `CELL_STATUS` **(outbound)**: This is the port the cell uses to send information to the transducer regarding the results of resting or feeding behavior by the occupying agent.
- `OUTPUT_N` **(outbound)**: This is the port the cell uses to send a child to its northern neighbor.
- `OUTPUT_NW` **(outbound)**: This is the port the cell uses to send a child to its northwestern neighbor.
- `OUTPUT_NE` **(outbound)**: This is the port the cell uses to send a child to its northeastern neighbor.
- `OUTPUT_S` **(outbound)**: This is the port the cell uses to send a child to its southern neighbor.
- `OUTPUT_SW` **(outbound)**: This is the port the cell uses to send a child to its southwestern neighbor.
- `OUTPUT_SE` **(outbound)**: This is the port the cell uses to send a child to its southeastern neighbor.
- `OUTPUT_E` **(outbound)**: This is the port the cell uses to send a child to its eastern neighbor.
- `OUTPUT_W` **(outbound)**: This is the port the cell uses to send a child to its western neighbor.

Now let us take a look at the external-transition function for the cell. This function describes how the cell reacts to incoming inputs:

Algorithm 1 Cell External-transition function

```
function DELTEXT($e, message$)
    if $phase = $ DEAD then                                              ▷ If we are in the DEAD phase
        if $message$ is on port START and $bacterium \neq$ null then
            $phase \leftarrow$ REST                                      ▷ Start lifecycle because cell is occupied
        else if $message$ is on port INPUT_BACTERIUM then
            $bacteriumMessage \leftarrow message$                        ▷ Initialize new bacterium from child genome
            $bacterium \leftarrow createBacterium(bacteriumMessage)$
            $phase \leftarrow$ REST
    else if $phase = $ WAIT then                                         ▷ If we are in the WAIT phase
        if $message$ is on port QUERY_RESPONSE then
            $queryResponse \leftarrow message$
            if $queryResponse.row = row$ and $queryResponse.column = column$ then
                if $queryResponse.emptyCellAvailable$ then
                    $phase \leftarrow$ REPRODUCE                         ▷ There is room available to reproduce
                else
                    $phase \leftarrow$ REST
```

Here we can see a few things happening:

- In the **DEAD** state, if we receive a message on the **START** port, and the cell is currently occupied, we can initiate the lifecycle of the occupying bacterium (transition to the **REST** phase). Otherwise we ignore the message on the **START** port. Also, if we are in the **DEAD** state, and we receive a message on the **INPUT_BACTERIUM** port, it means that a neighboring cell has reproduced and has sent us the child's genome. Using the genome we can instantiate a new instance of the agent and then initiate its lifecycle (transition to the **REST** phase).
- If we are in the **WAIT** state, it means that we are waiting for a response from the transducer. So, if we receive a message on the **QUERY_RESPONSE** port, we first check to see if the message applies to this cell. This is because the transducer has a single output-port that is connected to the **QUERY_RESPONSE** port of all cells. If the address in the message matches the address of the current cell, we check to see if the message says that an empty cell is available. If it is, we transition to the **REPRODUCE** state. If no empty cell is available, we will transition to the **REST** state.

Now let's take a look at the internal-transition function:

Algorithm 2 Cell Internal-transition function

```
function DELTINT
    if $phase = $ REST then
        $bacterium.rest()$
        $phase \leftarrow$ SEND_REST_MESSAGE
        $\sigma \leftarrow 0$
    else if $phase = $ SEND_REST_MESSAGE then
        if $bacterium.lifespan = 0$ or $bacterium.freeEnergy \leq 0$ then
            $bacterium \leftarrow$ null
            $phase \leftarrow$ DEAD            ▷ Lifespan is over or agent has no energy so transition to DEAD state
            $\sigma \leftarrow \infty$
        else if $bacterium.freeEnergy \geq reproductionThreshold$ then
            $phase \leftarrow$ QUERY           ▷ We need to see if we any vacancies in neighboring cells
            $\sigma \leftarrow 0$
        else
            $phase \leftarrow$ FEED            ▷ We need more energy
    else if $phase = $ FEED then
        $metabolysisResult \leftarrow bacterium.feed(nutrient)$
        $phase \leftarrow$ SEND_FEED_MESSAGE
        $\sigma \leftarrow 0$
    else if $phase = $ SEND_FEED_MESSAGE then
        $phase \leftarrow$ REST
    else if $phase = $ QUERY then
        $phase \leftarrow$ WAIT            ▷ We need to wait for a response from the transducer
        $\sigma \leftarrow \infty$
    else if $phase = $ REPRODUCE then
        $childGenome \leftarrow bacterium.reproduce()$
        $phase \leftarrow$ SEND_CHILD      ▷ We need to send the child genome to a vacant neighbor
        $\sigma \leftarrow 0$
    else if $phase = $ SEND_CHILD then
        $phase \leftarrow$ REST
```

Note: In the above pseudocode, I have mentioned values that σ can take only in cases where it is pertinent. In all other cases, assume that σ is equal to some pre-defined state-transition time (1 in the case of this model).

In this function, we see the following things happening:

- If we are in the `REST` phase, we invoke the `rest()` method on the bacterial agent. We then immediately transition to the `SEND_REST_MESSAGE` phase.
- If we are in the `SEND_REST_MESSAGE` phase, we have to check a few things. We first check to see if the lifespan of the agent is zero, or if its free energy is lesser than or equal to zero. In both cases, it means that the agent is dead. So we set the state variable `bacterium` to `null` and then transition into the `DEAD` phase, which is a passive state. On the other hand, if the free energy of the agent is greater than the reproduction threshold, we need to see if we even have room for a child agent. Hence, we will immediately transition into the `QUERY` phase. Finally, if none of the above-two cases apply, we will transition to the `FEED` phase.
- If we are in the `FEED` phase, we invoke the `feed()` method on the bacterial agent. We assign the result of that method to the `metabolysisResult` state variable, which we can later send out to the transducer through the output function. We then immediately transition to the `SEND_FEED_MESSAGE` phase.
- If we are in the `SEND_FEED_MESSAGE` phase, it means that we finished sending our message to the transducer and now we can transition into the `REST` phase.
- If we are in the `QUERY` phasee, it means that we have sent a query to the transducer and need to wait for a response. So we will transition into the `WAIT` phase, which is a passive phase.
- If we are in the `REPRODUCE` phase, it means that we have enough room for a child agent, and so we invoke the `reproduce()` method on the bacterial agent. We assign the result of that method to the `childGenome` state variable, which we can later send out to the vacant neighbor using the output function. We then immediately transition to the `SEND_CHILD` phase.
- If we are in the `SEND_CHILD` phase, it means that we have finished sending the child genome to a vacant neighbor. Hence, we transition into the `REST` phase.

The next function we need to look at is the output function:

Algorithm 3 Cell Output function

```
function OUT
    if $phase =$ QUERY then
        $port \leftarrow$ QUERY
        $message \leftarrow createQueryMessage(row, column)$
    else if $phase =$ SEND_CHILD then
        $port \leftarrow queryResponse.neighborPort$
        $message \leftarrow createBacteriumMessage(row, column, childGenome)$
    else if $phase =$ SEND_REST_MESSAGE then
        $port \leftarrow$ CELL_STATUS
        $message \leftarrow createRestMessage(row, column, bacterium.lifespan, bacterium.freeEnergy)$
    else if $phase =$ SEND_FEED_MESSAGE then
        $port \leftarrow$ CELL_STATUS
        $message \leftarrow createFeedMessage(row, column, metabolysisResult)$
    return $(port, message)$
```

In the output function, the following things happen:

- If the current phase is **`QUERY`**, it means that we need to query the transducer regarding any vacancies in the neighboring cells. To do this, we create a query-message and then send it through the **`QUERY`** port. The query-message contains information about the address of the requesting cell (i.e., **`row`** and **`column`**).

- If the current phase is **`SEND_CHILD`**, it means that we need to send the child genome to a neighboring cell. The information regarding the vacant neighbor is contained within the **`queryResponse`** state variable (whose value comes from the transducer in response to a query, and is set in the external-transition function). We create a bacterium-message and then send value of the **`childGenome`** state variable to the appropriate neighbor through the appropriate port.

- If the current phase is **`SEND_REST_MESSAGE`**, it means that we need to send a rest-message to the transducer. We create a new instance of the message, which includes information such as the address of the requesting cell (i.e., **`row`** and **`column`**) and the current lifespan and free-energy of the agent that occupies this cell. We then send this message through the **`CELL_STATUS`** port.

- If the current phase is **`SEND_FEED_MESSAGE`**, it means that wee need to send a feed-message to the transducer. We create a new instance of the message, which includes information such as the address of the requesting cell (i.e, **`row`** and **`column`**) and the result of the agent's feeding, which is contained in the **`metabolysisResult`** state variable. We also send this message through the **`CELL_STATUS`** port.

This more-or-less covers the cell model. Next, we will go over the transducer.

3.1.2 Transducer Model

The transducer performs two important-functions: it keeps track of the state of the grid so that it can respond to queries from cells regarding vacancies, and it collects results of nutrient metabolysis, as well as the states cells in general (i.e., how much lifespan and energy they have). We can understand how the transducer accomplishes these functions by looking at it in greater detail. First, we will go over the primary states of the transducer:

- **`PASSIVE`**: In this state, the transducer doesn't do anything; it merely waits for input.

- **`PROCESSING`**: In this state, the transducer is processing a message from its message-queue.

- **`SEND_QUERY_RESPONSE`**: In this state, the transducer send a response to a query from a cell.

As mentioned before, the transducer keeps track of some important information. These form its secondary states, and are as follows:

- **`cellStatus`**: This is a two-dimensional array where each location contains information regarding a cell on the grid. The information includes the status of the current cell (occupied or vacant) as well as the nutrient in that cell. This array is updated when the transducer receives information that changes the state of the grid. For example, if a cell dies the transducer marks the corresponding location in the array as vacant, whereas if a cell is available for a child agent, the transducer "reserves" that location by marking that cell as occupied.

- **`queryResponse`**: This query response from the transducer, that will be sent to a requesting cell. This contains information such as the row and column of a vacant cell, if one is available.

- **`cellMessages`**: This is a queue that contains messages from cells. Since numerous cells could be requesting information and sending statuses, the transducer queues up the messages and processes them one by one.

- **averageEfficiencies**: This is a structure that maintains information about the average efficiencies of enzymes (per sampling interval) attempting to metabolize the novel nutrient. Since numerous cells will be metabolizing nutrients and sending data regarding their efficiencies to the transducer, it is helpful to look at an average efficiency over some predefined interval. This helps us see how the average efficiency changes over time. It is important to note that I do not consider efficiency-data from every agent. I only consider the efficiency-data from an agent if its efficiency for metabolizing the target nutrient is greater than zero, or if it is the offspring of an agent that was already in a cell with the novel nutrient. Otherwise, the reported efficiency becomes much lower, especially due to agents that move across the boundary from a non-novel nutrient-cell to a novel nutrient-cell; it is highly likely that the metabolic efficiency of these agents is zero, which would make the average efficiency seem to be much lower than it really is.

Now let us look at how the transducer communicates with cells in the grid. It does this through two ports:

- **IN (inbound)**: This is an input port through which the transducer receives messages from cells. These messages can include queries as well as other messages regarding the status of a cell.
- **QUERY_RESPONSE (outbound)**: This is an output port through which the tranducer responds to queries from cells regarding vacancies in neighboring cells.

Let's now take a look at the external-transition function for the transducer:

Algorithm 4 Transducer External-transition function

```
function DELTEXT($e, message$)
    if $phase$ = PASSIVE then
        if $message$ is on port IN then
            $cellMessages.enqueue(message)$
            $phase \leftarrow$ PROCESSING
    else if $phase$ = PROCESSING or $phase$ = SEND_QUERY_RESPONSE then
        if $message$ is on port IN then
            $cellMessages.enqueue(message)$
```

In the transducer's external-transition function, we first check to see if we are in the **PASSIVE** state. If we are in this state and we have a message on the **IN** input port, we enqueue the message into the **cellMessages** queue and then transition into the **PROCESSING** state. On the other hand, if we are already in the **PROCESSING** state and we get a message on the **IN** input port, we simply enqueue the message.

Now that the transducer has enqueued a message, it needs to process it. This happens in the internal-transition function (see Algorithm 5).

In the internal-transition function, we first check to see if we are in the **PROCESSING** phase. If we are in this phase, it means that we have messages to process, so we dequeue a message from the **cellMessages** queue. Depending on the type of the message, we do different things:

- If the message is a "query message", we check to see if the requesting cell has any vacant neighbors, and then assign that to the **neighbor** variable. If the cell does not have any vacant neighbors, **neighbor** is set to null. We then create a query response and transition into the **SEND_QUERY_RESPONSE** phase so that we can send our response to the requesting cell. We also update the **cellStatus** array and mark the neighbor's location as occupied (if **neighbor** is not **null**)

- If the message is a "rest message", we check to see if the agent from that cell is dead (i.e., lifespan is 0 and energy is less than or equal to zero). If the agent is indeed dead, we mark the agents location in the **`cellStatus`** array as vacant. We then check to see if we have any more messages left in the queue to process. If we do, we stay in the **`PROCESSING`** state. Otherwise, we transition into the **`PASSIVE`** state.
- If the message is a "feed message", we record information regarding the efficiency of nutrient metabolysis. Then we do the same thing we did in the case of the "rest message"; we check the queue to see if there are any more items left to process. If there are, we stay in the **`PROCESSING`** phase, otherwise we transition to the **`PASSIVE`** phase.

We can also be in the **`SEND_QUERY_RESPONSE`** state when the internal-transition function is called. In this phase we also have to check the status of the **`cellMessages`** queue to see if we have any more message to process. If we do, we transition to the **`PROCESSING`** phase; otherwise we remain in the **`PASSIVE`** phase.

Algorithm 5 Transducer Internal-transition function

```
function DELTINT
    if $phase = $ PROCESSING then
        $cellMessage \leftarrow cellMessages.dequeue()$
        if $cellMessage.isQueryMessage$ then
            $row \leftarrow cellMessage.row$
            $column \leftarrow cellMessage.column$
            $neighbor \leftarrow getEmptyNeighbor(row, column, cellStatus)$    ▷ neighbor is null if no vacancies
            $queryResponse \leftarrow createQueryResponse(row, column, neighbor)$
            if $neighbor \neq$ null then
                $cellStatus[row][column].occupied \leftarrow true$    ▷ Mark location as occupied
            $phase \leftarrow$ SEND_QUERY_RESPONSE
        else if $cellMessage.isRestMessage$ then
            $row \leftarrow cellMessage.row$
            $column \leftarrow cellMessage.column$
            if $cellMessage.lifespan = 0$ or $cellMessage.freeEnergy \leq 0$ then
                $cellStatus[row][column].occupied \leftarrow false$    ▷ Mark location as vacant
            if $cellMessage.empty = false$ then
                $phase \leftarrow$ PROCESSING    ▷ If queue is not empty we need to process remaining messages
            else
                $phase \leftarrow$ PASSIVE    ▷ If queue is empty we go back to the PASSIVE state
                $\sigma \leftarrow \infty$
        else if $cellMessage.isFeedMessage$ then
            $recordAverageEfficiency(averageEfficiencies, cellMessage.efficiency)$
            if $cellMessage.empty = false$ then
                $phase \leftarrow$ PROCESSING    ▷ If queue is not empty we need to process remaining messages
            else
                $phase \leftarrow$ PASSIVE    ▷ If queue is empty we go back to the PASSIVE state
                $\sigma \leftarrow \infty$
    else if $phase = $ SEND_QUERY_RESPONSE then
        if $cellMessage.empty = false$ then
            $phase \leftarrow$ PROCESSING    ▷ If queue is not empty we need to process remaining messages
        else
            $phase \leftarrow$ PASSIVE    ▷ If queue is empty we go back to the PASSIVE state
            $\sigma \leftarrow \infty$
```

We will now take a look at the transducer's output function (see Algorithm 6). This is where the transducer responds to queries from cells regarding vacancies in neighboring cells. In this function we check to see if we are in the **SEND_QUERY_MESSAGE** phase. If we are, we create a new message out of **queryResponse** and send it out on the **QUERY_RESPONSE** port.

Algorithm 6 Transducer Output function

```
function OUT
    if $phase = \texttt{SEND\_QUERY\_MESSAGE}$ then
        $port \leftarrow \texttt{QUERY\_RESPONSE}$
        $message \leftarrow createQueryResponseMessage(queryResponse)$
    return $(port, message)$
```

The transducer is concurrent and therefore requires a confluent function as well since it can be producing an output when an input comes in. In this case, we will always perform the internal transition first:

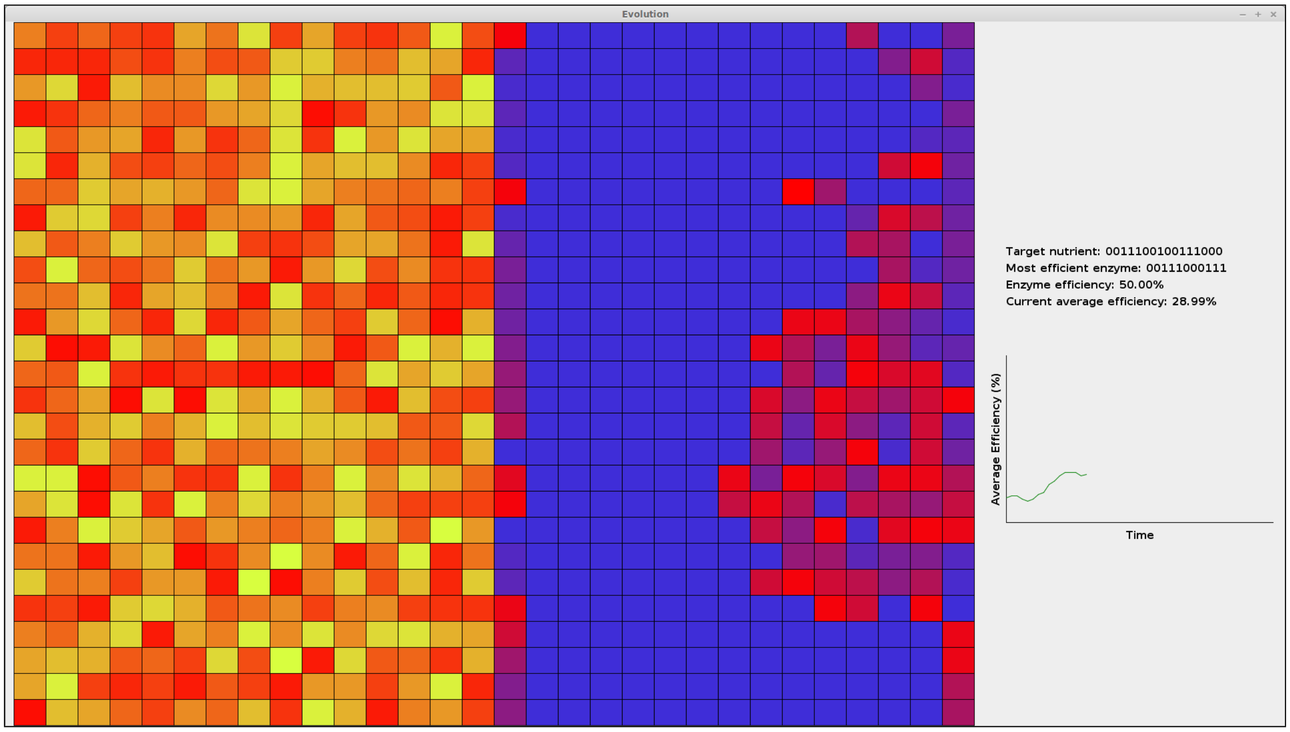
Algorithm 7 Transducer Confluent function

```
function DELTCON$(e, message)$
    DELTINT
    DELTEXT$(0, x)$
```

It should be clear now, how the transducer maintains the state of the grid, records data, and responds to queries from cells.

3.1.3 Visualizer

The visualizer is a class that isn't really part of the model. Instead, it is simply a way for us to visualize the agent-based model and its associated data as it is running. The visualizer uses Java's 2D graphics API to render data about the model. A screenshot is shown in Figure 4.



Target nutrient: 0011100100111000
Most efficient enzyme: 00111000111
Enzyme efficiency: 50.00%
Current average efficiency: 28.99%

Figure 4: Visualizer Screenshot

The visualizer uses data recorded by the transducer to render information on the screen. The majority of the visualizer is dedicated to displaying agents and nutrients on the grid, and the grid itself. Cells are color-coded according to the nutrients they possess. In the figure, you can see that one half of the grid has cells with nutrients of a specific type (yellow) whereas the other half has nutrients of a different type (blue). Individual agents are colored red, but their transparency is adjusted to reflect how much of their maximum lifespan they have left. So as the agents grow older, their transparency increases.

The right side of the visualizer shows a graph of the average efficiency over time. It also displays the target nutrient (i.e., the one we hypothesize that agents can evolve to metabolize) and the enzyme with the highest metabolic-efficiency for that nutrient, seen thus far. In this way we can see how the model changes over time.

3.2 Experimental Setup

The general idea behind the experimental setup was to determine if I could observe the evolution of a mutation that allowed an agent to metabolize a novel nutrient-source. Before I performed the experiment, I first had to establish the semantic validity of my model. Therefore my first experiment simply involved a 3x3 grid of cells connected to each-other and to the transducer (Figure 5).

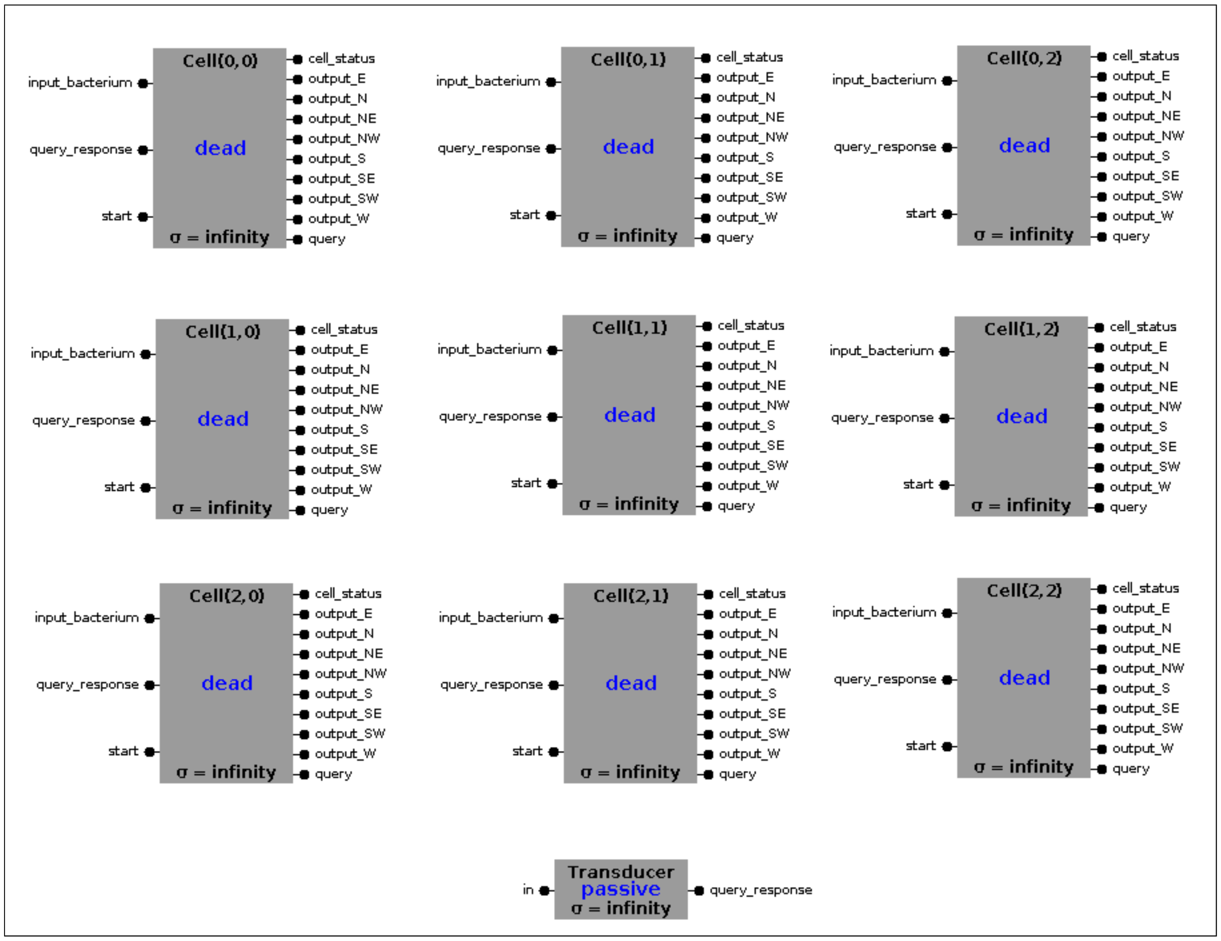


Figure 5: 3x3 grid model (couplings omitted for clarity)

By running this degenerate case, I was able to confirm that my cells transitioned through the right phases, and that the transducer was able to maintain consistent state, collect the right information, and respond to requests from the cells. Once I established that my model was valid, I was able to perform more-complex experiments. However, I soon noticed that I wasn't exactly getting the results I wanted.

For example, I found that agents were reproducing in the novel-nutrient area of the grid even though they weren't actually metabolizing the nutrient. As I examined the results further, I realized that agents were evolving to have a very high life-span, a high initial free-energy, low metabolic-energy, and low reproduction-threshold. This meant that agents could proliferate in the novel-nutrient area even without metabolizing the nutrient. That is, there wasn't any selective pressure to metabolize the nutrient itself since mutations to other attributes in the genome could still ensure that an organism could successfully survive and reproduce. To correct for this, I restricted mutations only to the section of the genome that codes for enzymes. This meant that all other attributes are held constant through reproduction, and it is only the section of the genome that codes for enzymes that can mutate. This means that the only thing that guarantees survival and reproduction for an organism is how well it can metabolize nutrients with the enzymes that it possesses.

One issue I faced was deciding what things I actually wanted to measure, and what parameters I should change. Since the model has non-trivial number of parameters, I could run numerous experiments to see how each parameter affects the evolution of agents. Since the focus is on whether the agents are actually able to metabolize the novel nutrient-source, most of my data pertains to that area. The parameters I ended up modifying to study their effects on agent evolution include initial free-energy, reproduction threshold, enzymes and nutrients, and mutation rates.

My first experiment had the following parameters:

- **Grid dimensions**: 30x30
- **Initial population of bacteria**: 50
- **Non-novel nutrient**: 1100011011000111_2
- **Novel nutrient**: 0011100100111000_2
- **Lifespan**: 20
- **Initial free-energy**: 20,000
- **Reproduction threshold**: 50,000
- **Metabolic energy**: 255
- **Enzymes**: 0011100100111000_2
- **Mutation rate**: 1% with equal chance for deletion, repetition, inversion, or insertion. 55% of the time one bit is modified, 30% of the time, two bits are modified, 10% of the time three bits are modified, and 5% of the time four bits are modified.

My aim here was simply to see whether I would see a mutant enzyme with the ability to metabolize the novel nutrient. I deliberately chose a nutrient that is the bitwise inverse of the existing enzyme since that ensures that the initial population of agents cannot metabolize the the novel nutrient at all. With the above parameters, an agent needs an enzyme with an efficiency of only 1.7% to survive. However, if it needs enough energy to reproduce, then it needs an enzyme with an efficiency of at least 10%.

My second experiment has the same parameters as the first experiment. However, I changed the mutation rates as follows:

- **Mutation rate**: 1% with 50% chance for inversion, 30% chance for addition, 15% chance for repetition, and 5% chance for deletion. 75% of the time one bit is modified, 15% of the time, two bits are modified, 7% of the time three bits are modified, and 3% of the time four bits are modified.

In this experiment, I specifically wanted to see how modifying the mutation rates would affect the evolution of the agent. In particular, I wanted to see how the modifying the chances of different types of mutation (i.e., inversion, addition, repetition, or deletion), affected the evolution of the agent.

4 Results

The results from both my experiments confirmed my hypothesis that random mutations and selective pressures led to the evolution of an agent that was able to metabolize the novel nutrient-source. The mutated enzyme was not extremely efficient, but the point is that it worked well enough that it allowed the agent to survive and reproduce. However, I did observe some unexpected results which I'll go over in detail in the final section of the paper.

4.1 Experiment 1

I ran the first experiment a total of 5 times and collected data from each run. I was mainly interested in how the efficiency of enzymes changed over time. My expectation was that the average efficiency would increase. This was what I saw, but only up to a point (see Figure 6). The average efficiency would peak and then slowly decline over time, with the end result that all agents eventually died. This was something that I did not expect. Furthermore, although some individual enzymes (see Table 3) had efficiencies as high as 56.25%, the peak average-efficiency never really crossed 40%. This was interesting, since I was hoping to see efficiency increase and at least stay constant instead of peaking and then declining.

Run	**Simulation Time**	$\overline{\eta}_{\textbf{peak}}$	$\overline{\eta}_{\textbf{peak}}$ **interval**	$\overline{\eta}_{\textbf{final}}$
1	425	37.5%	1	11.03%
2	430	28.9%	1	8.33%
3	476	22.48%	9	12.5%
4	475	30.78%	9	12.5%
5	427	25.94%	5	8.33%

Table 1: Simulation time, peak average-enzyme-efficiency, and final average enzyme-efficiency over 5 runs (Experiment 1)

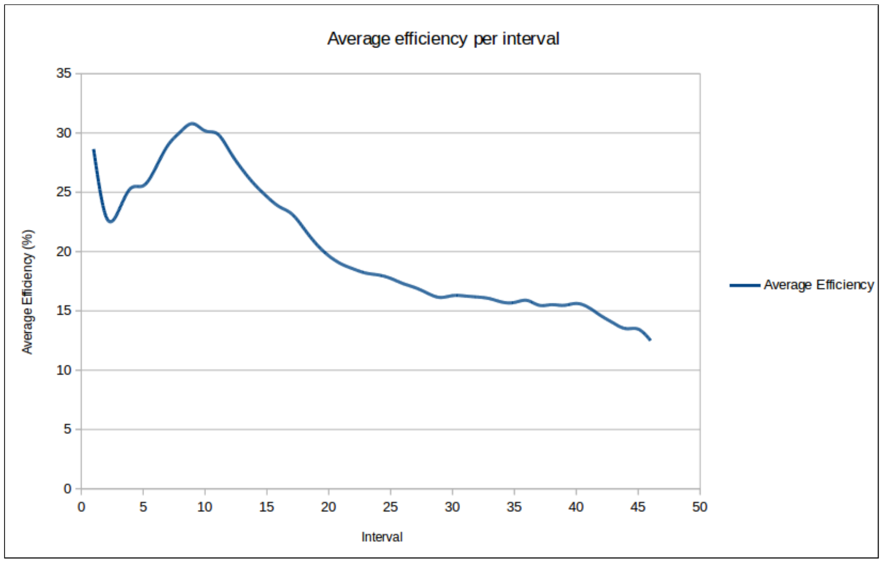


Figure 6: Average efficiency per interval; data is from run 4. (Experiment 1)

Another statistic I was interested in, was the time when the first functional-enzyme appeared (see Table 2). A functional enzyme is any enzyme that is able to metabolize the novel-nutrient with an efficiency greater than zero. For the most part I was seeing useful mutations emerge within 15 time-units, i.e., a functional mutation seemed to appear rather quickly. This isn't too surprising since only a small change is required to make the original enzyme function well-enough to metabolize the new nutrient. For example in run 4, the first functional-enzyme we see is 111100100111000_2 with an efficiency of 6.25% (see Table 2). The original enzyme for the agent is 0011100100111000_2. The mutations necessary to convert the first enzyme into the functional one are: deletion at bit 15 and inversion at bit 16. The deletion doesn't contribute in any way, but with a single inversion we are able to get a new enzyme that while not very efficient, is at least functional.

Run	**Time first functional-enzyme appeared**	**Enzyme**	$\eta_{\textbf{enzyme}}$
1	12	1110010011100_2	31.25%
2	5	01111100111000_2	18.75%
3	8	10100100111000_2	6.25%
4	8	111100100111000_2	6.25%
5	8	1010100111000_2	12.5%

Table 2: Time of appearance of first functional-enzyme and its efficiency over 5 runs (Experiment 1)

I also kept track of the most-efficient enzyme overall for a run (see Table 3), and noticed that in most cases the most-efficient enzyme had an efficiency between 30% to 50% percent. In previous experiments that I ran, I observed enzymes with efficiencies as high as 62.5%, but this was rather uncommon. I believe this is because the enzyme for the novel nutrient is the bitwise inverse of the original enzyme. Hence the number of mutations required to reach 100% are quite large which means that it is quite rare to see a mutation with a high efficiency. However, I was hoping that through natural selection we would end up seeing enzymes with higher and higher efficiencies. However, we have to realize that evolution doesn't necessarily have to result in the most-efficient result; we only need a result that is "good enough" and in most cases it looks like enzymes with efficiencies between 30% and 50% are "good enough".

Run	**Most-efficient enzyme**	$\eta_{\textbf{enzyme}}$
1	1110010011111_2	43.75%
2	1001001111_2	50%
3	11001001000_2	37.5%
4	1100100111_2	56.25%
5	10011100100_2	43.75%

Table 3: Most efficient-enzyme and its efficiency over 5 runs (Experiment 1)

4.2 Experiment 2

In my second experiment, I decided to see how mutation rates affect the evolution of agents. As it turned out, the results weren't all that different from my first experiment. I still saw enzyme efficiency peaking, and then declining over time (see Figure 7). The end result was also similar to experiment one; agents would eventually end up dying out in both regions of the grid. Similar to the results in experiment one, I observed enzymes with individual efficiencies as high as 56.25% (see Table 6) and peak average-efficiency between 30% and 20% (see Table 4). This suggests that perturbing the mutation rates didn't seem to change the outcome significantly, and that there must be other factors that are making the model behave the way it is.

Run	Simulation Time	$\overline{\eta}_{\text{peak}}$	$\overline{\eta}_{\text{peak}}$ interval	$\overline{\eta}_{\text{final}}$
1	370	33.03%	1	12.5%
2	382	32.16%	2	12.5%
3	348	31.25%	1	11.72%
4	333	21.5%	5	12.5%
5	455	24.65%	7	6.25%

Table 4: Simulation time, peak average-enzyme-efficiency, and final average enzyme-efficiency over 5 runs (Experiment 2)

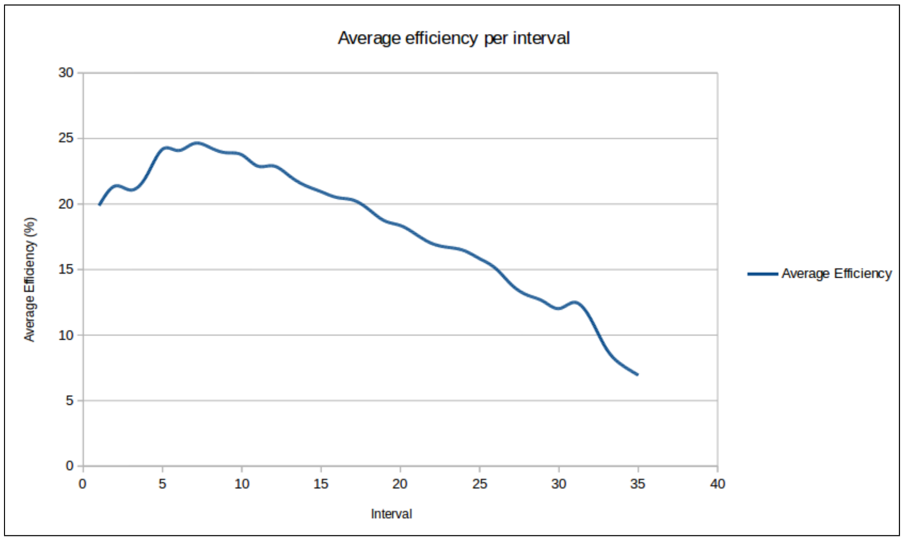


Figure 7: Average efficiency per interval; data is from run 5. (Experiment 2)

Results regarding the appearance of the first functional-enzyme didn't seem to differ very much from those in experiment one either (see Table 5). This is again due to reasons similar to those in experiment one; a single mutation is enough to make an enzyme functional.

Run	Time first functional-enzyme appeared	Enzyme	η_{enzyme}
1	5	01110010011100_2	37.5%
2	12	111000111000_2	18.75%
3	13	110010011100_2	31.25%
4	8	01000100111000_2	12.5%
5	8	01100111000_2	6.25%

Table 5: Time of appearance of first functional-enzyme and its efficiency over 5 runs (Experiment 2)

The most-efficient enzymes in the second experiment also had efficiencies similar to those in experiment one (see Table 6). Here also, I observed most efficiencies to be between 30% and 50%. As with experiment one, I was able to observe one enzyme with an efficiency of 56.25%.

Run	Most-efficient enzyme	$\eta_{\textbf{enzyme}}$
1	011100100110_2	56.25%
2	0001110010011100_2	37.5%
3	110010011100_2	31.25%
4	1001001100_2	37.5%
5	0010011100000_2	50%

Table 6: Most efficient-enzyme and its efficiency over 5 runs (Experiment 2)

5 Conclusions

While the ability to metabolize novel-nutrients was observed in both experiments, enzyme efficiency decreased over time. On examining the enzymes in the later generations of agents, I noticed that most of them had shrunk to only a few bits in length and were mostly non-functional. This was surprising since I had expected fitness to increase over time. Essentially, transcription errors had accumulated to a point where the agents were no-longer viable. After a performing a few more ad-hoc experiments and examining my model in greater detail, I believe I know the reasons for these results:

- **Competition**: The model does not allow competition for resources by agents. Currently, an agent monopolizes the nutrient source in the cell that it inhabits. In addition, the nutrient source is inexhaustible. These two factors ensure that an agent with an inefficient enzyme can still reproduce because the enzyme is "good enough".
- **Accumulation of errors**: The reproductive process in the model basically involves cloning the parent genome. Over time, we are making copies of copies of copies, due to which errors eventually build up to make the agent non-viable. A good example is the observation that enzymes in later generations were vestigial forms of the original enzymes; they had degraded to the point that they were no-longer useful.
- **Crowding**: A cell can contain only one agent. This means that agents are routinely crowded out and cannot reproduce even if they have enough energy. This hinders the passing-on of beneficial mutations to offspring; an agent becomes a victim of its own success.

Now that I know the reasons for the unexpected results, I have a few ideas to refine the model. With these refinements, I believe I will be able to observe results more in line with what I was expecting:

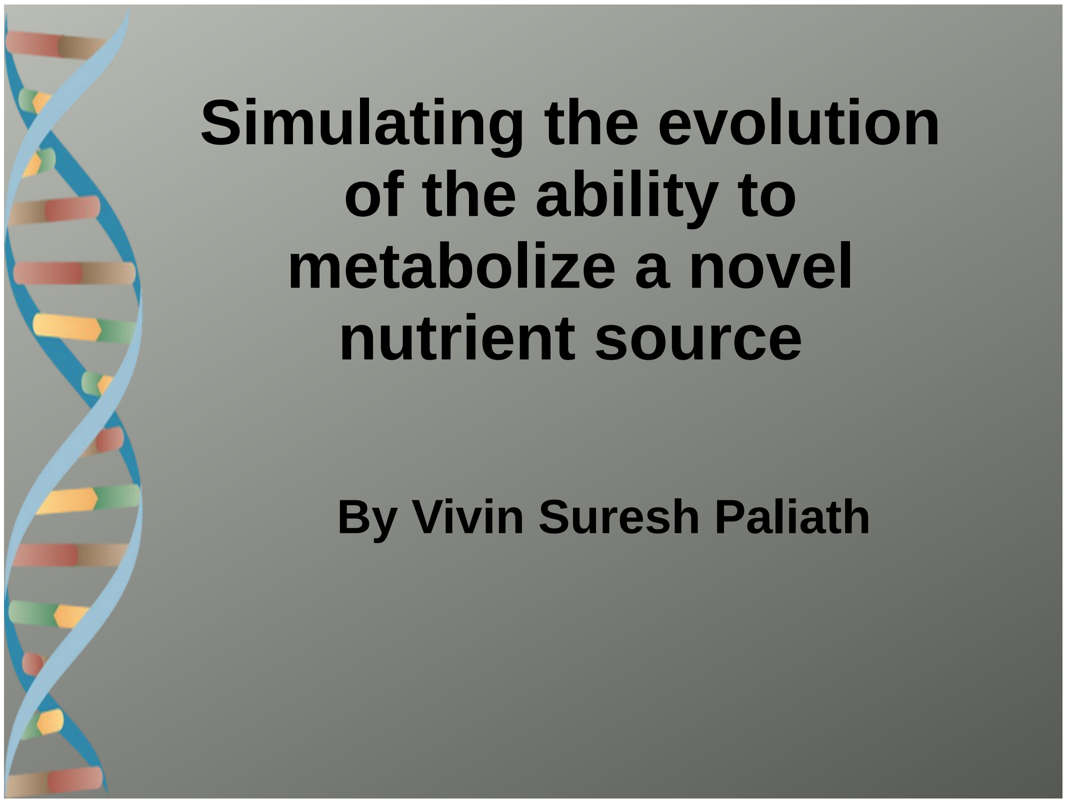
- **Exhaustible nutrient-source**: Nutrient sources in cells should be exhaustible and periodically refreshed. Agents with more-efficient enzymes will be able to consume a larger share of the nutrients whereas those with less-efficient enzymes will consume less.
- **Multiple occupancy**: Cells should be able to hold more than one agent. This change, coupled with an exhaustible nutrient source, forces agents in a cell to compete with each other for resources. This increases the selective pressure for more-efficient enzymes.
- **Agent movement**: Agents should be able to move from one cell to the other to forage for nutrients. This behavior allows more-successful agents to move and proliferate, and also forces competition for resources.
- **Sexual reproduction**: Agents should be able to sexually reproduce with other agents. This reduces the chances of deleterious mutations building up over time and increases overall fitness.

Overall, the experiments were a success and I was able to verify my initial hypothesis that an agent-based model is able to replicate the Lenski experiment through artificial evolution. Developing the model was an interesting and stimulating exercise that was also made easy by the DEVSJAVA framework. Though initially hard to comprehend, once I was able to understand the basic concepts of the framework, I was able to implement my model quickly. In particular, DEVSJAVA makes the development of models easy by abstracting away time-management and concurrency. For future work, I hope to refine this model further with the ideas described above.

6 Appendix A: Sample Console Output

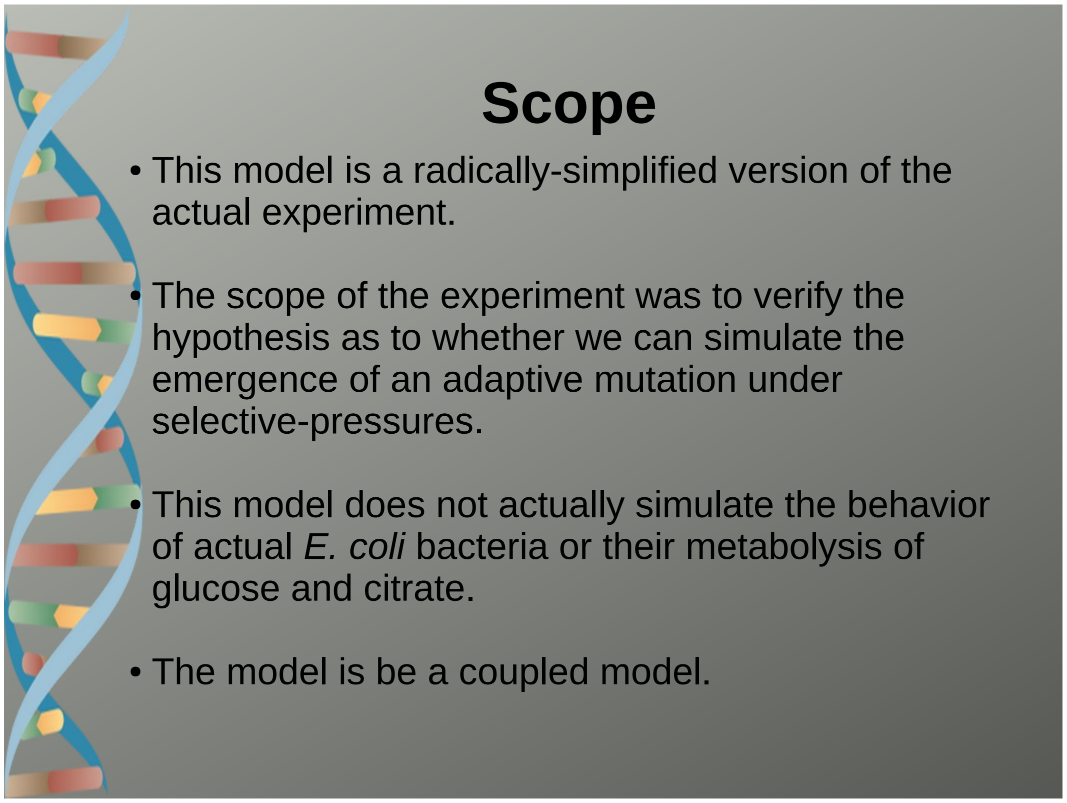
```
First functional enzyme found at clock 5.0. Enzyme is 01100111000 and efficiency was 0.0625
Peak average efficiency (thus far) of 0.19886363636363635 at time 10.0
###Average efficiency for interval 1: 0.19886363636363635
Peak average efficiency (thus far) of 0.21370967741935484 at time 20.0
###Average efficiency for interval 2: 0.21370967741935484
###Average efficiency for interval 3: 0.2105978260869565
Peak average efficiency (thus far) of 0.2215909090909091 at time 40.0
###Average efficiency for interval 4: 0.2215909090909091
Peak average efficiency (thus far) of 0.24182692307692308 at time 50.0
###Average efficiency for interval 5: 0.24182692307692308
###Average efficiency for interval 6: 0.24077868852459017
Peak average efficiency (thus far) of 0.24625576036866367 at time 70.0
###Average efficiency for interval 7: 0.24625576036866367
###Average efficiency for interval 8: 0.24305555555555566
###Average efficiency for interval 9: 0.2391304347826087
###Average efficiency for interval 10: 0.2375
###Average efficiency for interval 11: 0.22891923990498816
###Average efficiency for interval 12: 0.2290805785123967
###Average efficiency for interval 13: 0.22144112478031636
###Average efficiency for interval 14: 0.2141312893081761
###Average efficiency for interval 15: 0.2093937125748502
###Average efficiency for interval 16: 0.2049758953168044
###Average efficiency for interval 17: 0.2030590717299577
###Average efficiency for interval 18: 0.19595286885245916
###Average efficiency for interval 19: 0.18715034965034966
###Average efficiency for interval 20: 0.18359374999999997
###Average efficiency for interval 21: 0.17670863309352516
###Average efficiency for interval 22: 0.16981707317073175
###Average efficiency for interval 23: 0.16695205479452055
###Average efficiency for interval 24: 0.16452205882352947
###Average efficiency for interval 25: 0.1582733812949642
###Average efficiency for interval 26: 0.15091036414565834
###Average efficiency for interval 27: 0.13845802919708033
###Average efficiency for interval 28: 0.13050660792951535
###Average efficiency for interval 29: 0.12582781456953643
###Average efficiency for interval 30: 0.1201923076923077
###Average efficiency for interval 31: 0.125
###Average efficiency for interval 32: 0.11228813559322035
###Average efficiency for interval 33: 0.08928571428571429
###Average efficiency for interval 34: 0.07692307692307693
###Average efficiency for interval 35: 0.06944444444444445
Terminated Normally before ITERATION 254913 ,time: 455.0
```

7 Appendix B: Final Presentation Slides



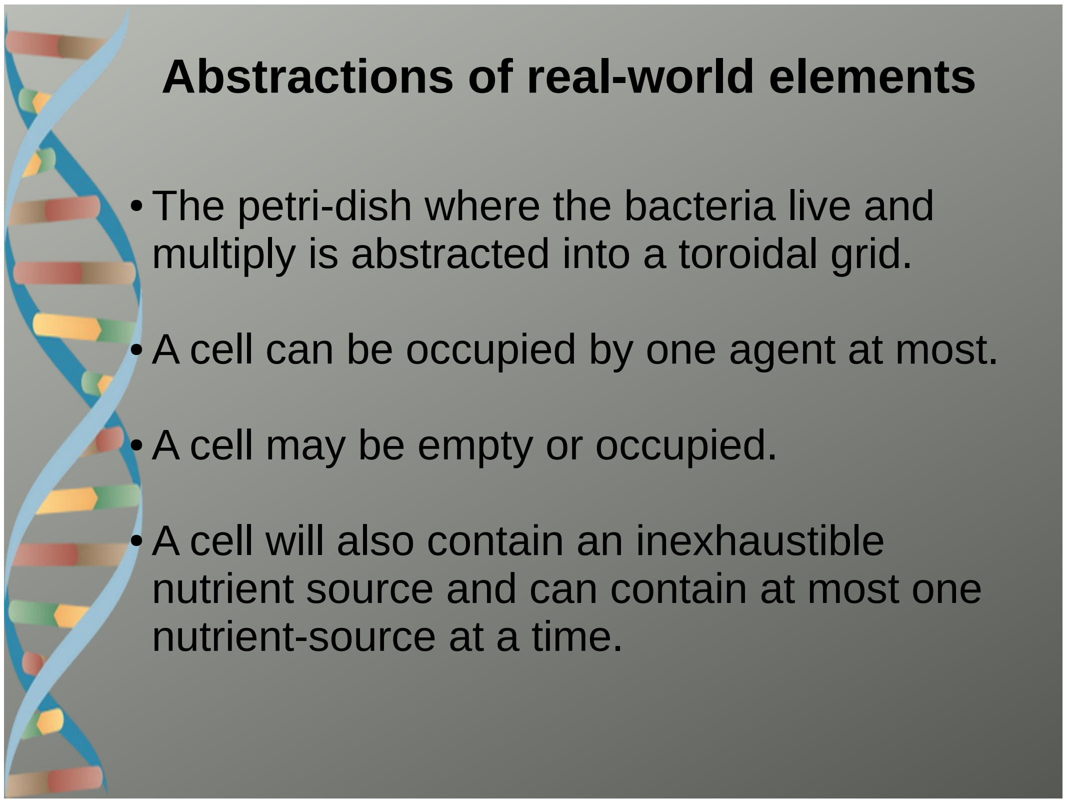
Simulating the evolution of the ability to metabolize a novel nutrient source

By Vivin Suresh Paliath



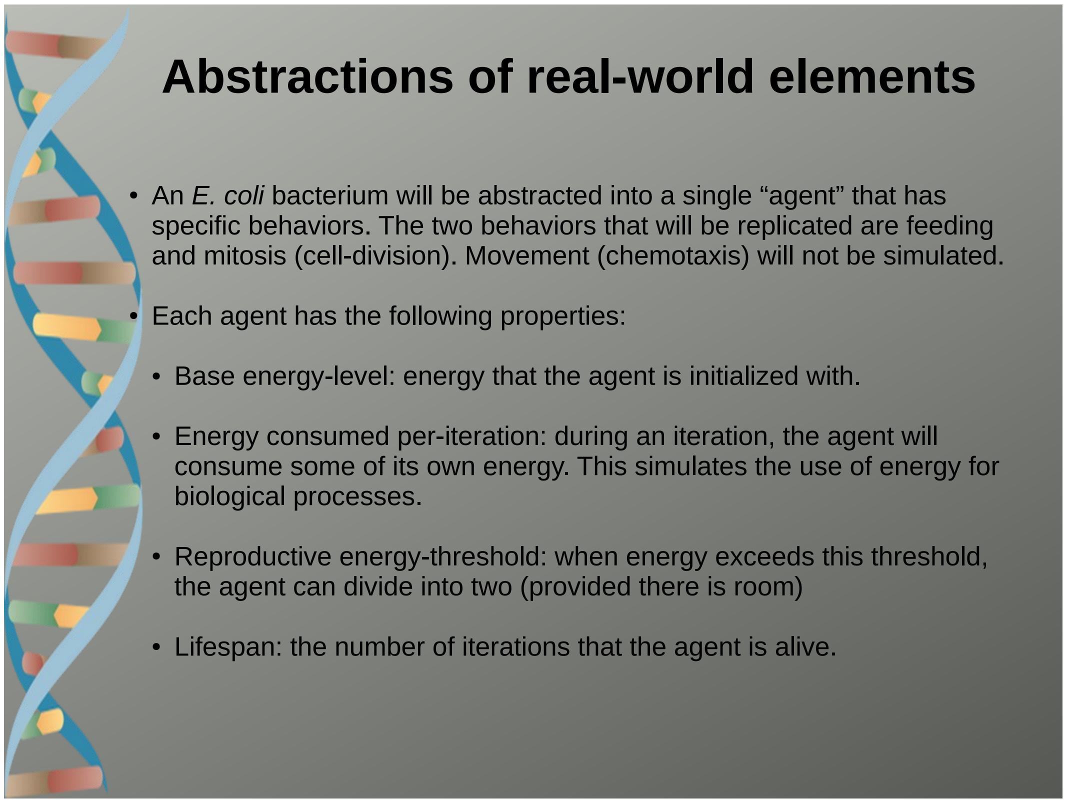
Scope

- This model is a radically-simplified version of the actual experiment.
- The scope of the experiment was to verify the hypothesis as to whether we can simulate the emergence of an adaptive mutation under selective-pressures.
- This model does not actually simulate the behavior of actual *E. coli* bacteria or their metabolysis of glucose and citrate.
- The model is be a coupled model.



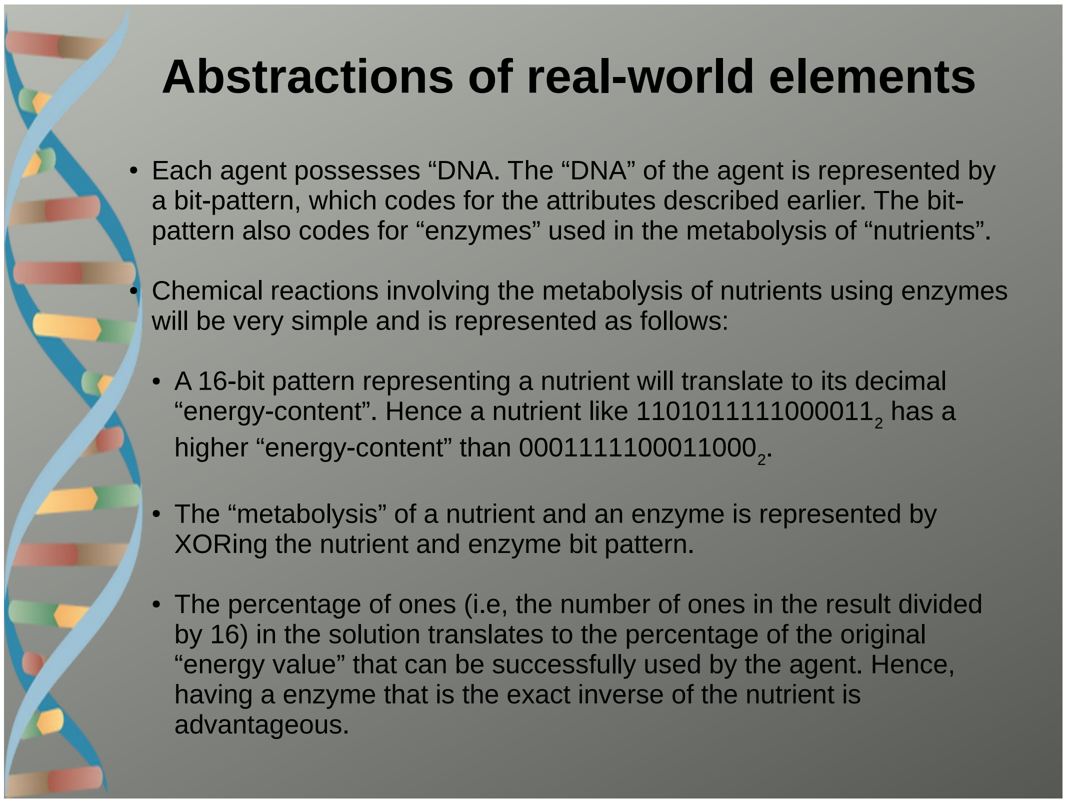
Abstractions of real-world elements

- The petri-dish where the bacteria live and multiply is abstracted into a toroidal grid.
- A cell can be occupied by one agent at most.
- A cell may be empty or occupied.
- A cell will also contain an inexhaustible nutrient source and can contain at most one nutrient-source at a time.



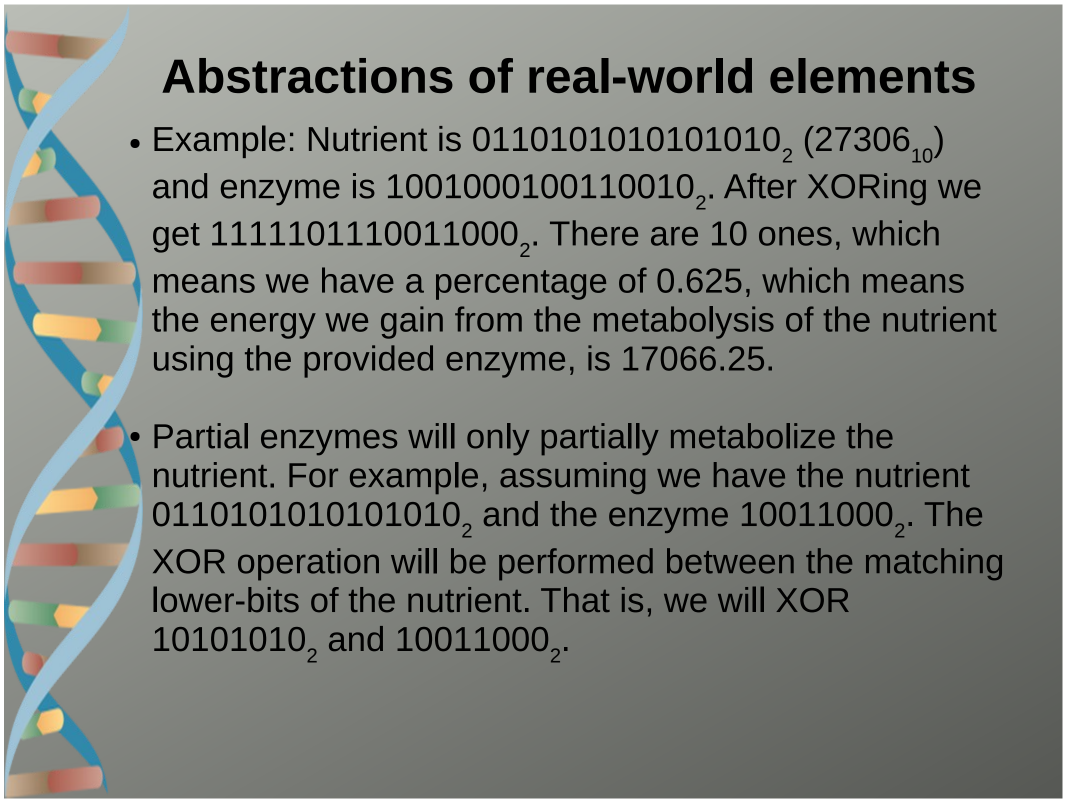
Abstractions of real-world elements

- An *E. coli* bacterium will be abstracted into a single “agent” that has specific behaviors. The two behaviors that will be replicated are feeding and mitosis (cell-division). Movement (chemotaxis) will not be simulated.
- Each agent has the following properties:
  - Base energy-level: energy that the agent is initialized with.
  - Energy consumed per-iteration: during an iteration, the agent will consume some of its own energy. This simulates the use of energy for biological processes.
  - Reproductive energy-threshold: when energy exceeds this threshold, the agent can divide into two (provided there is room)
  - Lifespan: the number of iterations that the agent is alive.



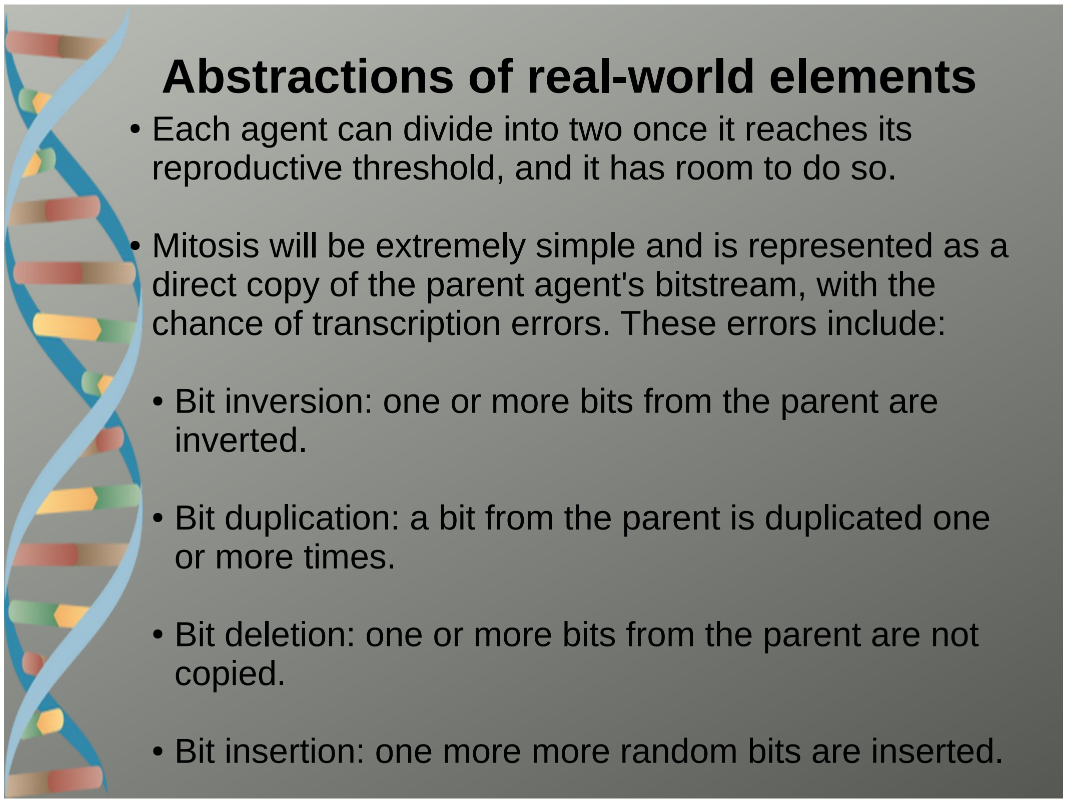
Abstractions of real-world elements

- Each agent possesses "DNA. The "DNA" of the agent is represented by a bit-pattern, which codes for the attributes described earlier. The bit-pattern also codes for "enzymes" used in the metabolysis of "nutrients".
- Chemical reactions involving the metabolysis of nutrients using enzymes will be very simple and is represented as follows:
  - A 16-bit pattern representing a nutrient will translate to its decimal "energy-content". Hence a nutrient like 1101011111000011_2 has a higher "energy-content" than 0001111100011000_2.
  - The "metabolysis" of a nutrient and an enzyme is represented by XORing the nutrient and enzyme bit pattern.
  - The percentage of ones (i.e, the number of ones in the result divided by 16) in the solution translates to the percentage of the original "energy value" that can be successfully used by the agent. Hence, having a enzyme that is the exact inverse of the nutrient is advantageous.



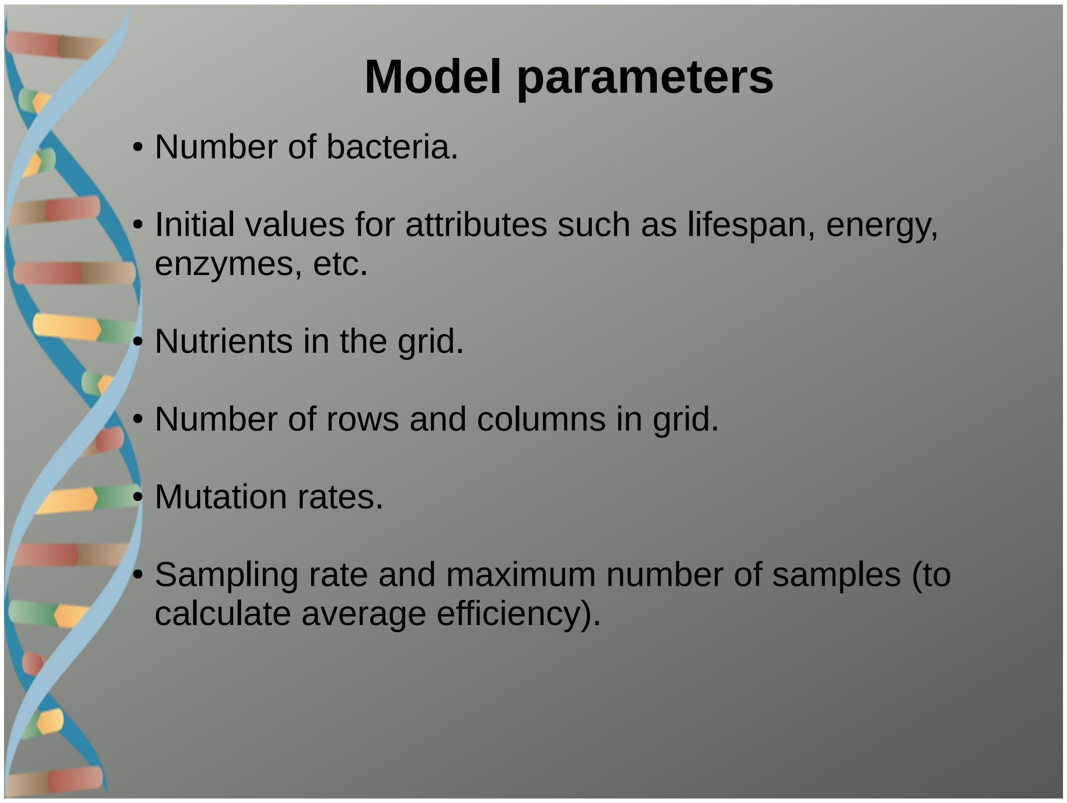
Abstractions of real-world elements

- Example: Nutrient is 0110101010101010_2 (27306_{10}) and enzyme is 1001000100110010_2. After XORing we get 1111101110011000_2. There are 10 ones, which means we have a percentage of 0.625, which means the energy we gain from the metabolysis of the nutrient using the provided enzyme, is 17066.25.
- Partial enzymes will only partially metabolize the nutrient. For example, assuming we have the nutrient 0110101010101010_2 and the enzyme 10011000_2. The XOR operation will be performed between the matching lower-bits of the nutrient. That is, we will XOR 10101010_2 and 10011000_2.



Abstractions of real-world elements

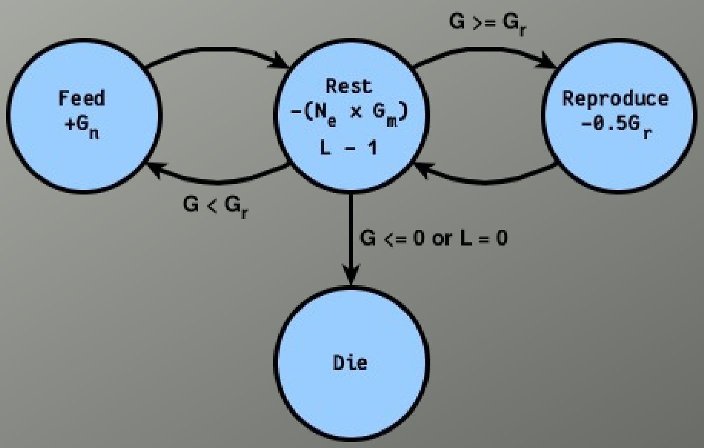
- Each agent can divide into two once it reaches its reproductive threshold, and it has room to do so.
- Mitosis will be extremely simple and is represented as a direct copy of the parent agent's bitstream, with the chance of transcription errors. These errors include:
  - Bit inversion: one or more bits from the parent are inverted.
  - Bit duplication: a bit from the parent is duplicated one or more times.
  - Bit deletion: one or more bits from the parent are not copied.
  - Bit insertion: one more more random bits are inserted.



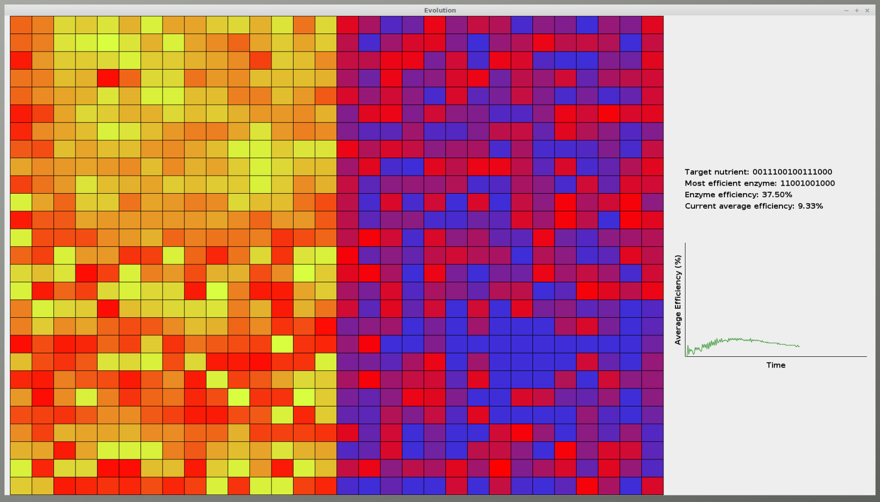
Model parameters

- Number of bacteria.
- Initial values for attributes such as lifespan, energy, enzymes, etc.
- Nutrients in the grid.
- Number of rows and columns in grid.
- Mutation rates.
- Sampling rate and maximum number of samples (to calculate average efficiency).

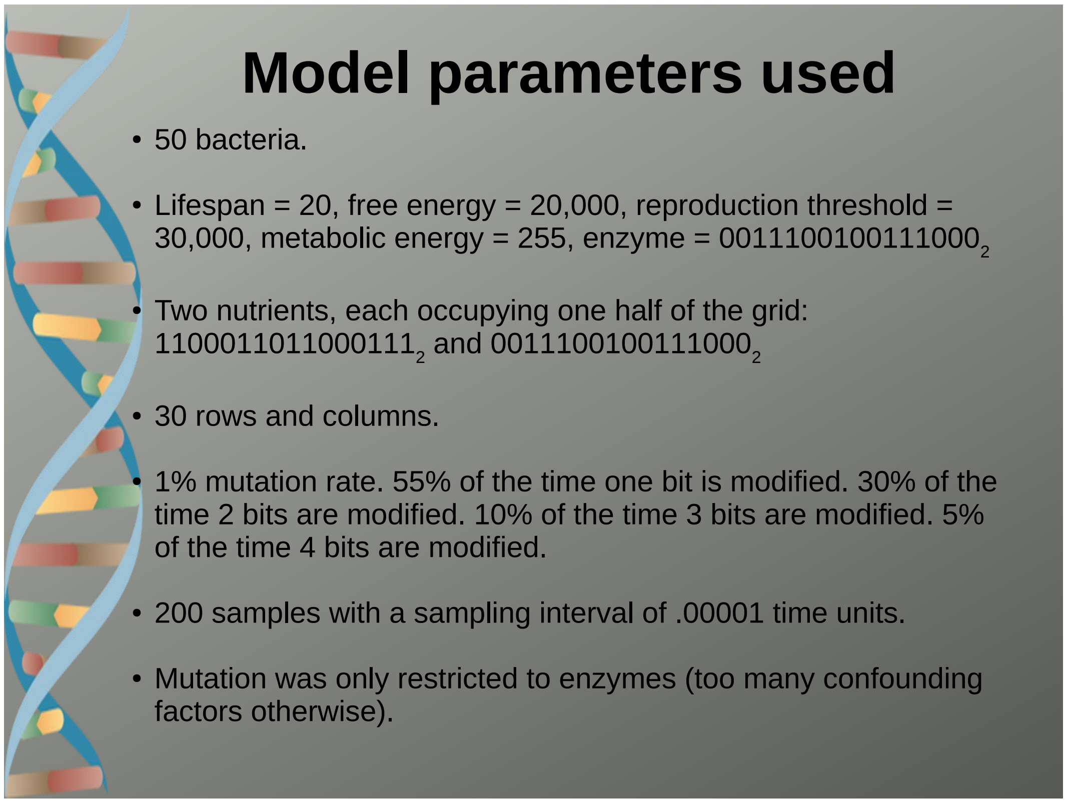
Cell life-cycle



Let's look at the model in action!

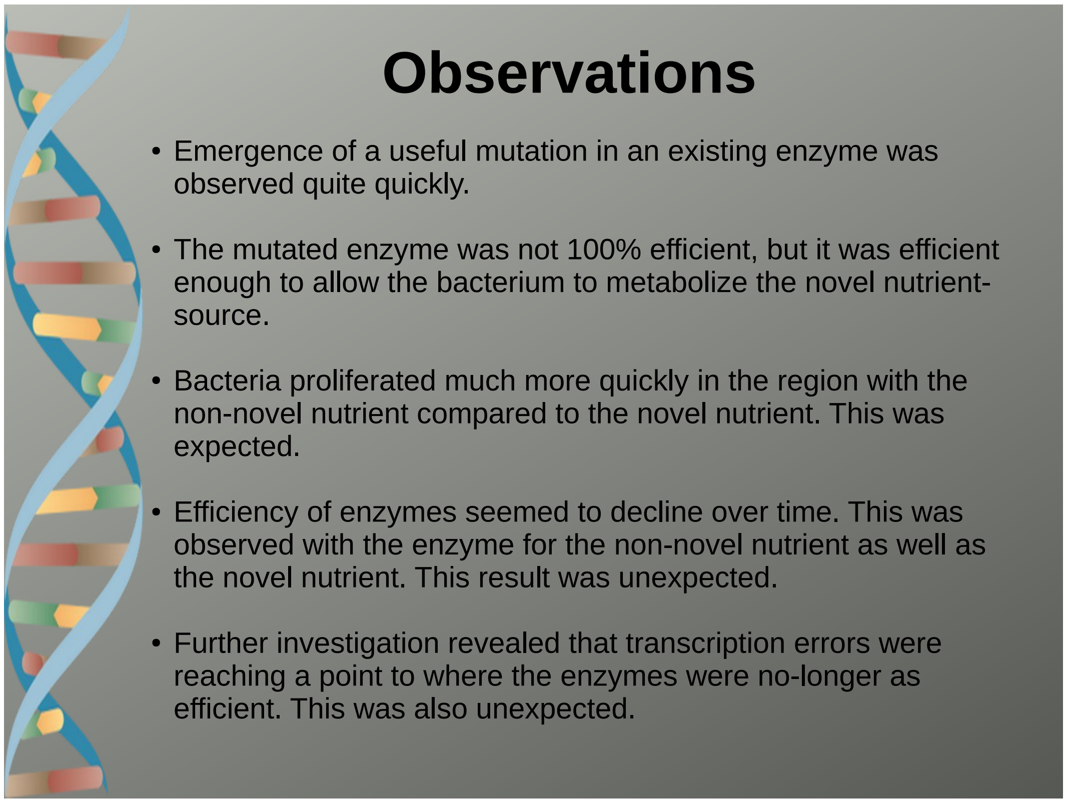


Target nutrient: 0011100100111000
Most efficient enzyme: 11001001000
Enzyme efficiency: 37.50%
Current average efficiency: 9.33%



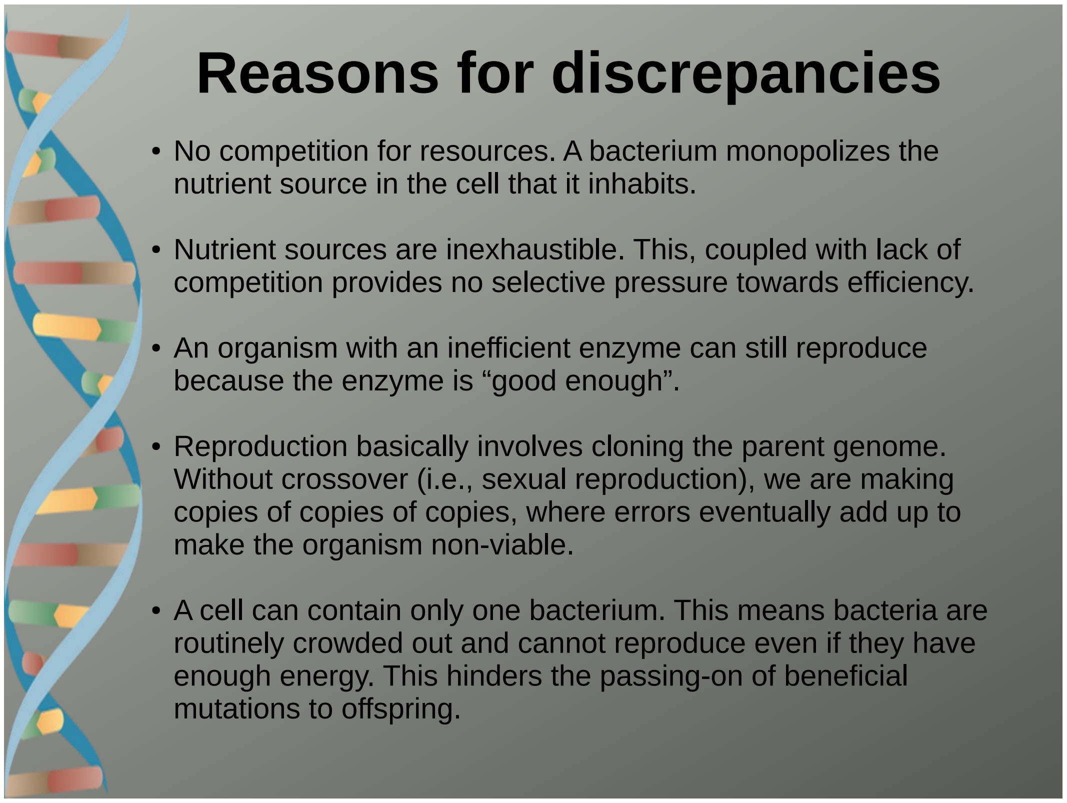
Model parameters used

- 50 bacteria.
- Lifespan = 20, free energy = 20,000, reproduction threshold = 30,000, metabolic energy = 255, enzyme = 0011100100111000_2
- Two nutrients, each occupying one half of the grid: 1100011011000111_2 and 0011100100111000_2
- 30 rows and columns.
- 1% mutation rate. 55% of the time one bit is modified. 30% of the time 2 bits are modified. 10% of the time 3 bits are modified. 5% of the time 4 bits are modified.
- 200 samples with a sampling interval of .00001 time units.
- Mutation was only restricted to enzymes (too many confounding factors otherwise).



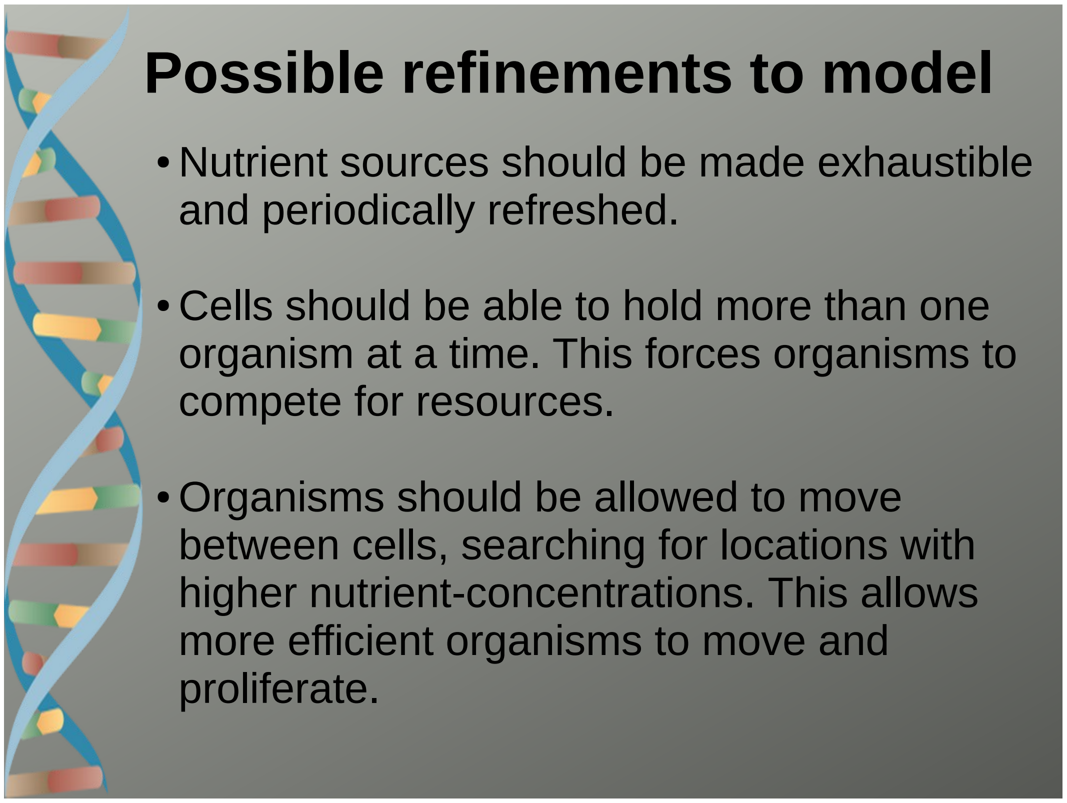
Observations

- Emergence of a useful mutation in an existing enzyme was observed quite quickly.
- The mutated enzyme was not 100% efficient, but it was efficient enough to allow the bacterium to metabolize the novel nutrient-source.
- Bacteria proliferated much more quickly in the region with the non-novel nutrient compared to the novel nutrient. This was expected.
- Efficiency of enzymes seemed to decline over time. This was observed with the enzyme for the non-novel nutrient as well as the novel nutrient. This result was unexpected.
- Further investigation revealed that transcription errors were reaching a point to where the enzymes were no-longer as efficient. This was also unexpected.



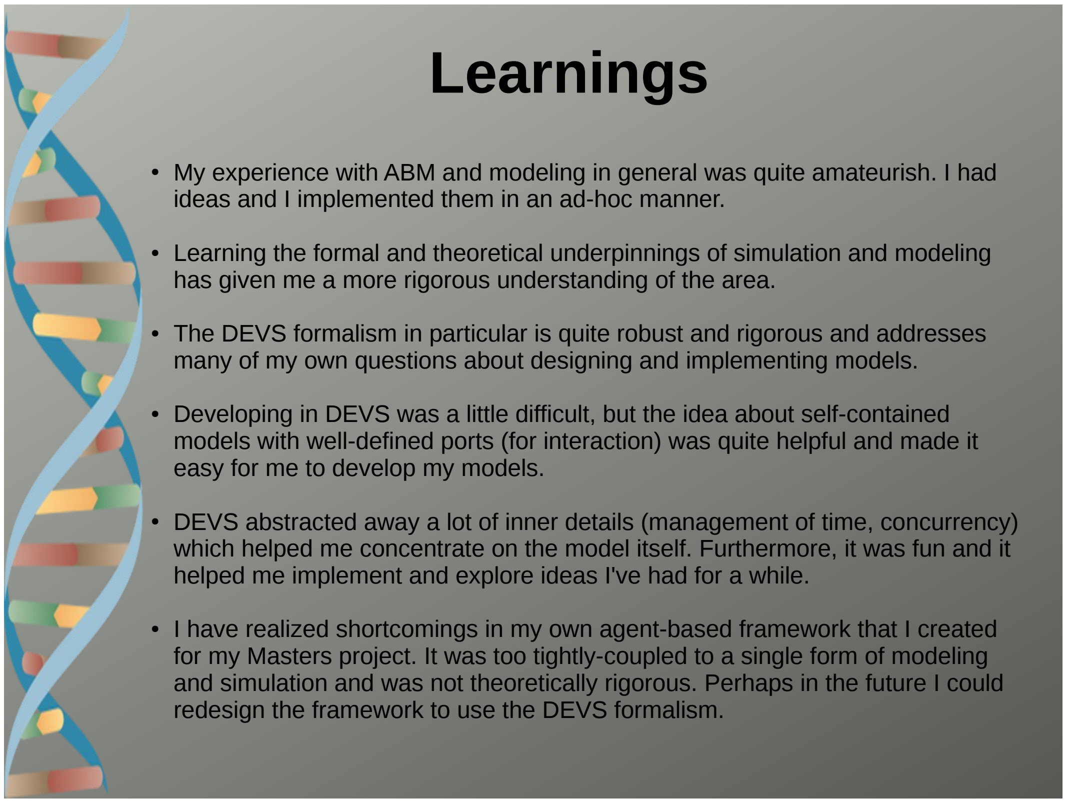
Reasons for discrepancies

- No competition for resources. A bacterium monopolizes the nutrient source in the cell that it inhabits.
- Nutrient sources are inexhaustible. This, coupled with lack of competition provides no selective pressure towards efficiency.
- An organism with an inefficient enzyme can still reproduce because the enzyme is “good enough”.
- Reproduction basically involves cloning the parent genome. Without crossover (i.e., sexual reproduction), we are making copies of copies of copies, where errors eventually add up to make the organism non-viable.
- A cell can contain only one bacterium. This means bacteria are routinely crowded out and cannot reproduce even if they have enough energy. This hinders the passing-on of beneficial mutations to offspring.



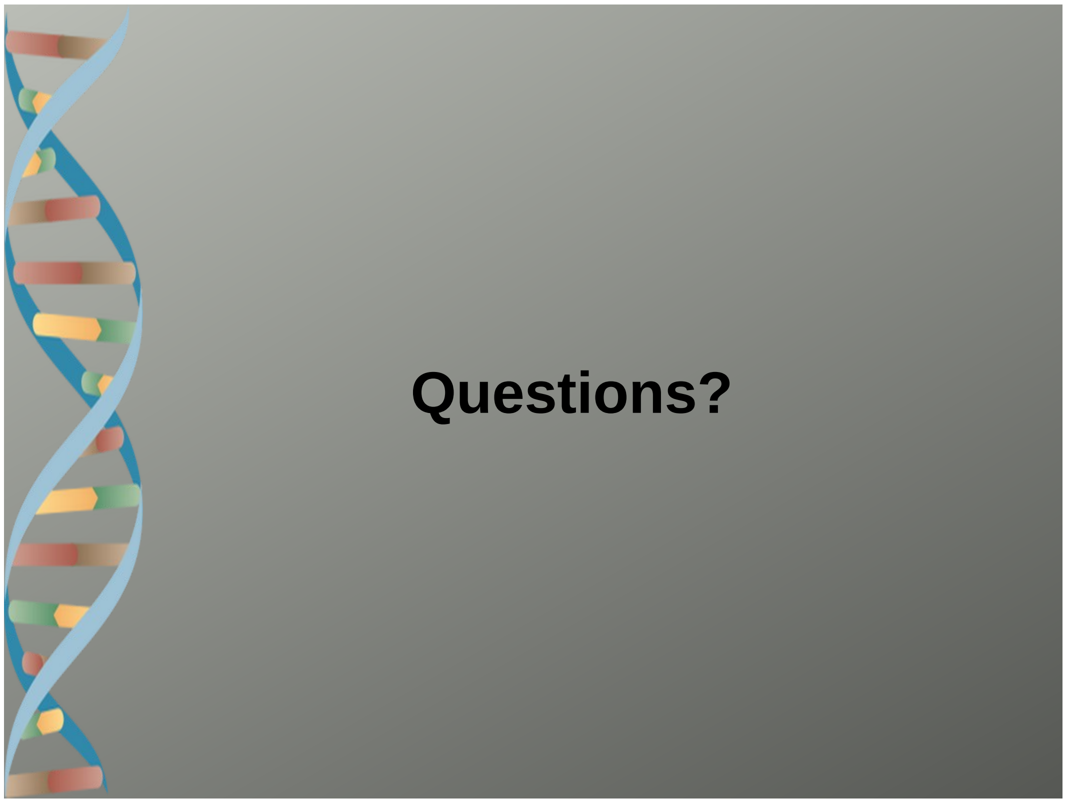
Possible refinements to model

- Nutrient sources should be made exhaustible and periodically refreshed.
- Cells should be able to hold more than one organism at a time. This forces organisms to compete for resources.
- Organisms should be allowed to move between cells, searching for locations with higher nutrient-concentrations. This allows more efficient organisms to move and proliferate.



Learnings

- My experience with ABM and modeling in general was quite amateurish. I had ideas and I implemented them in an ad-hoc manner.
- Learning the formal and theoretical underpinnings of simulation and modeling has given me a more rigorous understanding of the area.
- The DEVS formalism in particular is quite robust and rigorous and addresses many of my own questions about designing and implementing models.
- Developing in DEVS was a little difficult, but the idea about self-contained models with well-defined ports (for interaction) was quite helpful and made it easy for me to develop my models.
- DEVS abstracted away a lot of inner details (management of time, concurrency) which helped me concentrate on the model itself. Furthermore, it was fun and it helped me implement and explore ideas I've had for a while.
- I have realized shortcomings in my own agent-based framework that I created for my Masters project. It was too tightly-coupled to a single form of modeling and simulation and was not theoretically rigorous. Perhaps in the future I could redesign the framework to use the DEVS formalism.



Questions?

8 References

[1] Hurst, L. D., *Fundamental concepts in genetics: genetics and the understanding of selection.* Nature Reviews Genetics, 10(2):83–93, 2009

[2] Orr H. A., *Fitness and its role in evolutionary genetics.* Nature Reviews Genetics, 10(8):531–9, 2009

[3] Zachary D. Blount, Christina Z. Borland, and Richard E. Lenski, *Historical contingency and the evolution of a key innovation in an experimental population of Escherichia coli.* Proceedings of the National Academy of Sciences of the United States of America, 105(23):7899-7906, 2008

[4] Kinoshita, S.; Kageyama, S., Iba, K., Yamada, Y. and Okada, H., *Utilization of a cyclic dimer and linear oligomers of e-aminocaproic acid by Achromobacter guttatus.* Agricultural & Biological Chemistry, 39(6):1219–23, 1975

[5] Natarajan, A., Srienc, F., *Glucose uptake rates of single E. coli cells grown in glucose-limited chemostat cultures.* Journal of Microbiological Methods, 42(1):87-96, 2000