# Digiostelium Specification

W. Garrett Mitchener

## Contents

**1 Introduction** **2**

**2 Virtual machine and arena** **3**
2.1 Design overview . . . 3
2.2 Arena . . . 3
2.3 Arena state . . . 4
2.4 Colony state . . . 4
2.5 Cell state . . . 5
2.6 Initialization for a task . . . 5
2.7 Task processing phases . . . 5
2.8 Phase 1: Broadcast of input signals . . . 5
2.9 Phase 2: Synaptic channels . . . 6
2.10 Phase 3: Reactions (instruction evaluation) . . . 6
2.11 Phase 4: Signal emitters . . . 7
2.12 Stop conditions . . . 7
2.13 Efficacious analysis . . . 7

**3 Genome** **8**
3.1 Design overview . . . 8
3.2 Bit string chromosome . . . 8
3.2.1 Mutations . . . 8
3.2.2 Crossover . . . 9
3.2.3 Chromosome alignment . . . 10
3.2.4 Configuration file for chromosome alignment . . . 11
3.2.5 Standard interpretation . . . 11
3.2.6 Example instruction specification . . . 13
3.3 Genome architecture specification . . . 14
3.4 Haploid singleton genome architecture . . . 14
3.5 Diploid genome architecture . . . 16

**4 Selection** **17**
4.1 Agents and eggs . . . 17
4.2 Distinction between an `Agent` and a `Colony` . . . 18
4.3 Selection process . . . 18
4.4 Example configuration . . . 20

**5 Problems** **20**
5.1 Input-Output problems . . . 21
5.2 Singleton problems . . . 22

**6 Copy problem** **22**
6.1 Summary . . . 22

6.2 Implementation . . . . . . . . . . 23
6.3 Beacon subtask . . . . . . . . . . 24
6.4 Configuration . . . . . . . . . . 24
6.5 Details of the rating function . . . . . . . . . . 25
6.6 Example configuration . . . . . . . . . . 26
6.7 Example solution . . . . . . . . . . 28

**7 PairCopy problem** **30**

**8 Configuration files** **30**

# 1 Introduction

This document specifies the environment and behavior of simulated agents in experimental version 12 of Mitchener's Digiostelium[1] artificial life simulation. Not all features described here are used in all problem types. Some features are present for future use and are only partially implemented.

This purpose of this document is primarily to specify the design and operation of the Digiostelium simulation package. With that in mind, only the most essential references to other publications and software are included. This document is not intended to be an academic research article and therefore does not include a survey of the relevant literature or an extensive bibliography.

Be aware that this document has to explain the relationships between many data types and objects with mutually recursive definitions. Therefore, some features must be partially described when first mentioned, and further details are given later.

It should be noted that the simulation documented here is intended for experiments in *artificial life* (a-life) and *artificial biology*. It is a toolkit for simulating evolutionary processes in which biological phenomena such as synonymous codons, genetic diversity, species formation, and the advantages and disadvantages of different genome architectures are of primary interest. Although the vocabulary of "problem" and "solution" is used to describe what the simulation does, it should be understood that the emphasis is on identifying and understanding biologically plausible solutions, even if they are unsuitable for other applications. A single sample run may take what seems like a very long time, hours or months. However, the proper comparison is not to, say, optimization of a function by gradient descent that completes in a fraction of a second, but rather to the time scale of biological evolution, which can be years or millions of years.

An additional purpose is to fill the gap between mathematical models and actual biological experiments. Although mathematical modeling can serve as an excellent tool for understanding biological phenomena in a broad sense, a model can become intractable once more than a few details have been added, provided it is even possible to include the details of interest.

A particular type of experiment that this simulation was designed for is *evolutionary discovery*. To give an example, the well-known replicator equation can explain, in terms of abstract game theory, why members of a species engage in certain activities with various probabilities. However, there is no obvious way to use it to understand how an ancestor species that was not yet capable of performing these activities evolved the ability to do so, because a finite list of all possible behaviors must be specified when the system of differential equations is constructed. An agent-based simulation, such as the one described in this document, can be used for experiments in which agents that are capable of general-purpose computation evolve new behaviors, implemented by biologically reasonable mechanisms, that were not specified ahead of time. Only the consequences of an agent's actions must be specified. In many cases, performing this type of experiment on actual living organisms is unfeasible due to the time scale, resource requirements, or ethical considerations. In contrast, running many samples of a simulation is inexpensive and relatively easy to do, even if it takes hours or weeks.

[1] The name `Digiostelium` is a pun on *digital* and *Dictyostelium discoideum*, a widely-studied species of colonial amoeba, commonly called slime mould.

The simulation is *not* designed for solving general optimization problems, and it is not necessarily appropriate for such applications. Nor is it designed for data mining or other statistical applications. For an engineer or statistician, many of the biologically-inspired features of this simulation would be inefficient distractions. For such needs, other software would be more appropriate.

The simulation is *not* designed to produce black boxes. It includes features for probing agents to understand how they work.

It is designed to produce reproducible samples. In particular, numbers used to initialize the pseudorandom number generator are preserved as part of the data files for a sample run, so that it is possible to exactly re-create a sample run from its initial state. It utilizes the Mersenne Twister by Saito and Matsumoto, with a few modifications to make their `dSFMT` code callable from Haskell. Parts of the simulation run sequentially so as to properly pass around the PRNG state, which reduces opportunities to speed up the computation with parallelism. A future version may make use of a splittable PRNG, as in Steele et. al, *Fast splittable pseudorandom number generators*, ACM SIGPLAN Notices, October 15, 2014.

The simulation is a work in progress. Many features are partially implemented, and will be completed as time permits. While the author has made every effort to prevent and fix bugs, no guarantees are possible.

# 2 Virtual machine and arena

The code for this part of the simulation is in the `BaselMachine` package.

## 2.1 Design overview

The virtual machine (VM), named the *Basel Machine* in the source code, is designed as a discrete abstract chemistry system.[2] It takes the form of an *artificial reaction network* or *ARN*. A program consists of instructions that are executed simultaneously, as if by idealized biomolecules in solution.

The Basel Machine is extremely simple. It has one instruction type (op-code), designed to resemble the operation of a generic regulatory gene. Recall that within a strand of DNA, a gene consists of a *regulatory region*, consisting of *binding sites* for molecules called *transcription factors*, followed by one or more *coding regions* called *exons* separated by spacers called *introns*. Transcription factors are proteins that bind electrostatically to binding sites. They affect how readily *transcriptase* proteins bind to adjacent DNA, which is the molecular machinery that perform transcription of DNA into mRNA. Some repress transcription, and such a binding site is called an *operon*. Some enhance transcription, and such a binding site is called a *promotor*. Once mRNA is produced, the introns are edited out, resulting in one or more mRNAs that are then translated into peptide chains, that then fold and combine into proteins. The single VM instruction type effectively checks for whether enhancement minus repression exceeds some threshold, and if so, it creates units of abstract biochemicals, which in turn repress or enhance the activity of other instructions.

Consequently, Basel Machine programs take the form of an *artificial regulatory network* (ARN), analogous to a *gene regulatory network* (GRN) that might be found in living organisms.

## 2.2 Arena

Execution of VMs takes place in an *arena*. There are one or more *colonies* in the arena. A colony is, roughly, an instantaneous state of the phenotype of an agent. Within a colony, there are one or more *cells,* each of which has its own machine state, as well as synaptic channels between cells. Each colony is configured with a collection of output sources and sensors.

Each colony at a *location*. Depending on the problem being solved, the location may be:

[2] The predecessor was called the Utrecht machine, after the city of Utrecht in the Netherlands, because I first presented about it at a conference there. The current VM is named after the city of Basel, Switzerland, where I spent some time after that conference.

- `NullLocation`: This is used when the fitness of an agent is independent of any sense of location, as in the Copy problem.
- Either `Here` or `There`: This is used when an agent interacts with one other agent at a time, and only the distinction between the two agents is relevant.

Other types of locations may be added in the future.

## 2.3 Arena state

The state of an arena is a map from integer colony indices to colony states, and a list of signal broadcast states.

A *broadcast state* consists of:

- a signal, the type of which depends on the problem in question
- a location, indicating where it was emitted
- an integer colony index: This is `Just k` if it came from colony `k`, or `Nothing` if it came from the environment.
- a maximum age indicating how many time steps it continues: This is `Nothing` for a signal that continues indefinitely.

## 2.4 Colony state

A colony consists of one or more interconnected cells. Its state consists of:

- parameter specifications
- a name, which is a textual string used for display purposes
- a location
- a graph of cell states (nodes) and the states of synaptic channels between them (edges)

The parameters that must be specified include:

- which cells have synaptic connections and the details thereof
- which cells can broadcast output signals
- which cells have sensors and how they operate

A signal *source* is specified by:

- the index within the colony of the source cell
- the source pattern `j`
- the source threshold $\vartheta$ (the symbol is the Greek letter theta, `\vartheta`)
- the activation delta $\delta$ to be subtracted when the signal is emitted

A signal *target* is specified by:

- the index within the colony of the target cell
- the target pattern `t`
- the activation delta $\delta$ to be added when the signal is received

Any two cells within a colony may be connected by a *synaptic channel*. The channel is specified by:

- a source, indicating the pre-synaptic cell
- a target, indicating the post-synaptic cell
- an integer called the *synapse delay*

The state of a synaptic channel consists of a fixed-length queue (FIFO) of Boolean values. These travel through the queue as time advances. The synapse delay specifies the length of the queue. It is initialized to all `False`, indicating no activity. When a `True` is pulled from the front of the queue, that is interpreted as the arrival of an action potential at the post-synaptic cell.

A colony may have *sensors*, each of which is specified by:

- a signal to which it is sensitive
- a target, indicating which cell to modify when the signal is detected

A colony may generate *broadcasts*, each of which is specified by:

- the signal to produce
- a source, indicating the circumstances under which a signal is emitted
- a maximum age indicating how many time steps it continues: This is `Nothing` for a signal that continues indefinitely.

## 2.5 Cell state

Each cell state consists of:

- an array `A` of integers: The value of `A[j]` is the number of units of molecules of type `j` in solution.
- a counter for how many reactions were performed in completing a task
- a total of the absolute value of all changes made to the elements of `A` during completion of a task: This is called the *total of reaction deltas*.

*Design feature:* Negative values of `A[j]` are allowed, although they cannot be easily interpreted.

*Design feature:* The reaction count and total of reaction deltas are kept so that agents may be rated in part on efficiency.

*Terminology:* The integer indices `j` into `A` are called *patterns* as part of the metaphor of gene regulation. Proteins and nucleotides have positively and negatively charged protrusions, and the pattern of charges enables one to bond electrostatically to the other, which is how cells transcribe DNA to mRNA and translate mRNA to peptide chains. Additionally, the integer properties of these indices are not really used, just their distinctiveness.

## 2.6 Initialization for a task

An agent is usually rated on its performance on several tasks. For each such task, before the first time step, each agent in the arena is initialized to an all-zero state. That is, `∀j: A[j]=0`, the reaction count is set to `0`, and the total of reaction deltas is set to `0`.

## 2.7 Task processing phases

The execution process proceeds in discrete time steps by repeating the following phases:

1. Input signals for the current time step are broadcast to each colony.
2. Within each colony, synaptic channels between cells are advanced.
3. Within each cell, all instructions are evaluated.
4. Within each colony, all signal emitters are checked.

Along the way, output signals emitted by colonies are recorded.

## 2.8 Phase 1: Broadcast of input signals

A problem may specify a sequence of signals to be broadcast within the arena at particular time steps. If such a signal is to be sent on the current time step, a broadcast object with age `0` is created.

Each colony is checked, and any cells with sensors sensitive to any currently active broadcast signals are modified. For each sensor that is triggered, the target cell state is modified by `A[t] += δ`, where `t` is the target pattern, and `δ` is the target activation delta.

After all colonies have received all broadcast signals, 1 is added to each one's current age. Any that have reached their maximum age are discarded.

## 2.9 Phase 2: Synaptic channels

In each colony, each synaptic channel is checked as follows.

In the pre-synaptic cell, `A[j]` is compared to $\vartheta$, where `j` is the source pattern and $\vartheta$ is the threshold. If `A[j]` ≥ $\vartheta$, then `True` is put into the back of the synapse's queue, and the activation delta is subtracted from `A[j]`. Otherwise, `False` is put into the queue. The queue now contains one more item than its specified length.

One Boolean value `x` is pulled from the front of the synapse's queue, restoring it to its specified length. If `x == False`, nothing happens. If `x == True`, an action potential has arrived at the post-synaptic cell, and it is modified, `A[t] += δ`, where `t` is the target pattern and δ is the activation delta.

*Design feature:* If the synapse delay is 0, then an action potential arrives at the target cell during the same time step it was emitted.

## 2.10 Phase 3: Reactions (instruction evaluation)

Each cell has a list of instructions called its *program*. Each instruction consists of these fields:

```
data Instruction = Instruction {
  switchPattern :: Int,
  switchThreshold :: Int,
  targetsAndDeltas :: [(Int, Int)]
  }
```

During the instruction evaluation phase, the instructions in each cell of an agent are evaluated as follows.

The current state array `A` is copied to a mutable array `ANext` that represents the future cell state. Each instruction is interpreted as

```
    if A[switchPattern] ≥ A[switchThreshold]
    then
        for each (target, delta) in targetsAndDeltas
        do
            ANext[target] += delta
```

An instruction is *active* on a time step if its threshold condition is true. Each time the modification `ANext[target] += delta` is performed, 1 is added to the total reaction counter. The absolute value of each `delta` so used is added to the total of reaction deltas.

After all instructions are evaluated, `ANext` replaces `A` as the cell's state.

*Design feature:* The conditional part of each instruction is evaluated based on the *current* cell state represented by `A`. No changes are made to the original `A`. Instead, all modifications are made to the *future* cell state `ANext`. Since the evaluation process reads only from `A` and modifies only `ANext`, the final value of `ANext` is independent of the order in which instructions are evaluated.

*Design feature:* Reactions of this form suffice to perform all possible bit-wise calculations. Consequently, Basel Machine colonies are capable of general computation, given sufficient resources. For example, this program performs the NAND operation:

```
If A[0] ≥ 1: ..., A[3] -= 1
If A[1] ≥ 1: ..., A[3] -= 1
If A[2] ≥ 0: A[3] += 2, ...
```

Specifically, `p = (A[0] > 0)` and `q = (A[1] > 0)`. If `p ∧ q` then `A[3]` remains `0` at the end of instruction evaluation. If `¬p ∨ ¬q` then `A[3]` will exceed `0` after one time step. Thus, `¬(p ∧ q) = (A[3] > 0)`. The parts labeled `...` are no-ops that modify some `A[x]` that is not otherwise used.

*Design feature:* Since a Basel Machine program can be represented by a list of tuples of integers, there are many ways to interpret any bit string as encoding such a program. This greatly simplifies the implementation of the genome, compared to genetic programming using expression trees, as in the work of John Koza.

## 2.11 Phase 4: Signal emitters

Each colony's broadcast specification is checked. For each source, the relevant cell is examined. If `A[j] ≥ ϑ`, where `j` is the source pattern and `ϑ` is the threshold, then:

- the specified signal is emitted with age 0 and added to the arena state at the colony's current location
- `A[j] -= δ` where `δ` is the specified activation delta

The list of active broadcasts is recorded at each time step.

## 2.12 Stop conditions

The arena is evaluated repeatedly until either a maximum number of time steps has elapsed, or a specified signal has been broadcast by one of the colonies.

## 2.13 Efficacious analysis

It is possible to run an arena and keep track of all patterns `j` for which `A[j]` is ever non-zero. From that data, the *efficacious* instructions can be identified. These are the ones that actually have an impact on the output of a colony.

For this fine-grained analysis, each instruction is broken down into an equivalent set of individual *reactions* which consists of these fields:

```
data Reaction =
  Reaction {
    index :: Int, -- ^ location in program
    sPat :: Int, -- ^ switch pattern
    thr :: Int, -- ^ switch threshold
    tPat :: Int,  -- ^ target pattern
    delta :: Int -- ^ amount to add to A[tPat] when active
    }
```

The index refers to which instruction in a program the reaction is part of.

The reactions that are ever active form the edges of a directed graph where the vertices are patterns. All patterns used as sources for broadcasts or synaptic channels are marked. The graph is searched in a loop:

- Reactions that target a marked pattern are marked.
- Patterns read by a marked reaction are marked.

When no new marks can be made, the set of efficacious reactions is those that are marked. The instructions to which they belong can be identified, and these are also called efficacious.

Typically, only a fraction of reactions are efficacious. Some read a switch pattern `s` for which `A[s]` never reaches the instruction's threshold. Some only affect elements of `A` that no other instructions read.

Efficacious analysis is usually performed by combining data from runs of a colony on all tasks posed by a problem.

# 3 Genome

The code for this part of the simulation is part of the `Digiostelium` package.

## 3.1 Design overview

The code is designed to allow experiments to use a variety of genome architectures. It incorporates selected details of natural DNA; specifically, the following features all have direct analogs:

- Strands of DNA can re-combine, which can speed up evolution by constructing child genomes that include a mix of potentially useful genes from multiple ancestors.
- During genome duplication, DNA polymerase molecules can slip, resulting in duplication and deletion of whole genes.
- During meiosis in diploid species, unequal crossover of chromosomes can also result in duplication and deletion of whole genes.
- For the most part, the physical location of a gene on a chromosome is immaterial. If the same gene were transposed to another part of the genome, it would continue functioning normally.
- For species whose reproductive rate is not dominated by the speed of genome duplication, large parts of the genome may never be transcribed or interpreted in any way.[3]

## 3.2 Bit string chromosome

Chromosomes are implemented as bit strings. The `bitutil` package contains a custom high-performance bit string library. The module `Digostelium.Chromosome`, contains the implementation of chromosome-related functions.

### 3.2.1 Mutations

When operating on a single chromosome, the following mutations are available:

- Single bit toggle, analogous to a substitution point mutation in DNA.
- Block duplication, analogous to gene duplication.
- Block deletion, analogous to gene deletion.

The probability parameters for these mutations are given by a `SingleChromosomeMutationSpec`:

```
data SingleChromosomeMutationSpec =
  SingleChromosomeMutationSpec {
    mutationBlockSize :: Int,
    pToggle :: Double, -- ^ probability of toggling a bit
    pDuplicate :: Double, -- ^ probability of duplicating a block
    pDelete :: Double, -- ^ probability of deleting a block
    minimumBlocks :: Int -- ^ all chromosomes must have at least this many blocks
  }
```

[3] For a time, such non-coding regions were called "junk DNA," but since they can serve other purposes, this term is no longer used.

Since chromosomes don't have an inherent notion of gene, the block size must be explicitly given, and is normally the length of an instruction.

The number of bits and blocks can be quite large and the probability of any one of them experiencing a mutation is small. It would be very time-consuming to generate a random number for each bit and block to determine whether a mutation occurred there. Instead, a discretized Poisson process is used. Given $p$ = the probability of "success" in a Bernoulli trial, the number of trials until the next success is approximated by $X$ = an exponentially distributed random number with parameter $\lambda = p$, so the expected number of trials to the next success is $\mu = E[X] = 1/p$. A sequence of samples $(X_1, X_2, \dots)$ is pulled, and mutations occur at indices round($X_1$), round($X_1$) + round($X_2$), ...

It is possible that a chromosome could shrink to zero length due to block deletion mutations. To prevent that, any chromosome with up to `minimumBlocks` blocks does not experience block deletion mutations.

### 3.2.2 Crossover

The chromosome data types themselves do not have a concept of "gene" or "instruction." That is so the implementation allows for more complex interpretations than the Standard interpretation. However, there is a *block* concept that facilitates the standard interpretation of a bit string as a concatenation of bit string blocks, each of which encodes a single instruction.

Since crossover takes place at block boundaries, if the specified block size matches the instruction size, crossover won't split an instruction. This is usually how the simulation is configured, but there is no constraint that block size must match instruction size.

Alternatively, the block size could be set to 1, in which case crossover could take place at any bit index.

Crossover parameters are specified by a `CrossoverSpec` object:

```
data CrossoverSpec = CrossoverSpec {
  crossoverType :: CrossoverType,
  crossoverBlockSize :: Int
  }
```

Two chromosomes, referred to as `left` and `right` may be recombined through crossover to a chromosome `result`.[4] A `CrossoverType` object specifies one of the following options.

**`NoCrossover`** Return `left`, ignore `right`. This is so that experiments can be done comparing evolution with and without recombination.

**`SinglePointKeepTail`** Crossover at a single point, and preserve the "tail" of the longer piece. The chromosomes are aligned from their beginnings, and split at a crossover index $k$, which is on a block boundary, selected uniformly at random based on the length of `left`. That is, `1 ≤ k ≤ length left - 1`, using `0`-based indexing. The first part of `left` is concatenated to the second part of `right`:

```
               k
left   = [12345 67890]
right  = [abcde fghij]
result = [12345 fghij]
```

Since $k$ is picked based on the length of `left`, it is possible that $k$ is beyond the end of `right`. In such a case, the result is the first part of `left`:

```
                  k
left   = [12345678 90]
right  = [abc]
result = [12345678]
```

[4] One could extract two chromosomes from a single crossover action, but there has never been a need for it in this simulation.

The "tail" that is kept is the part of `left` past the end of `right`.

A problem with this crossover variant is that it causes the average length of the chromosome to increase over time. To see this, suppose that two chromosomes are under consideration, and are $m$ and $n$ blocks long, with $m \geq n$. Half the time, the $n$-block chromosome is `left`, in which case the length of `result` is $m$. Half the time, the $m$-block chromosome is `left`. With probability $n/(m-1)$, $k \leq n$, in which case the length of `result` is $m$. But with probability $(m-n)/(m-1)$, $k > n$, in which case the length of `result` is $k$, and the expected value of $k$ is some number $\hat{k} > n$. The average length of `result` across all cases is

$$\frac{1}{2}m + \frac{1}{2}\left(\frac{n}{m-1}m + \frac{m-n-1}{m-1}\hat{k}\right) > \frac{m+n}{2}$$

hence the growth.

**`SinglePointLengthNeutral`** Crossover at a single point. The chromosomes are aligned from their beginnings, and split at a crossover index $k$, which is on a block boundary, selected uniformly at random based on the length of the shorter chromosome. The first part of `left` is concatenated to the second part of `right`.

This variant does not cause the average length of the chromosome to change over time, hence the name `LengthNeutral`. To see this, suppose that two chromosomes are under consideration, and are $m$ and $n$ blocks long, with $m \geq n$. Then $1 \leq k \leq n - 1$. Half the time, the $n$-block chromosome is `left`, and the length of `result` is $m$ Half the time, the $m$-block chromosome is `left`, and the length of `result` is $n$. So the average length of `result` is

$$\frac{m+n}{2}$$

**`SinglePointAligned`** Crossover at a single point. The chromosomes are aligned based on a modified Smith-Waterman algorithm, as described in Chromosome alignment. A point between blocks is chosen uniformly at random from the result. The first part of `left` is concatenated to the second part of `right`.

### 3.2.3 Chromosome alignment

A modified Smith-Waterman algorithm is implemented. See module `Digostelium.SmithWaterman`.

A `BlockAlignmentSpec` consists of the following

```
data BlockAlignmentSpec = BlockAlignmentSpec {
      matchWeight :: Int, -- ^ should be positive; points per matching bit
      mismatchWeight :: Int, -- ^ should be negative; points per mismatching bit
      gapWeight :: Int, -- ^ should be negative; points per gap bit
      alignmentBlockSize :: Int -- ^ bit length of a block
      }
```

The default is

```
makeDefaultBlockAlignmentSpec alignmentBlockSize =
  BlockAlignmentSpec {
    matchWeight = 2,
    mismatchWeight = -1,
    gapWeight = -1,
    ..}
```

The `alignmentBlockSize` is normally set to the number of bits per instruction, and must be computed separately, as in the Standard interpretation.

Given bit strings `left` and `right`, they are divided into consecutive substrings of length `alignmentBlockSize`, which are aligned, possibly including *gaps*, as in this example where `alignmentBlockSize = 3`, and `---` denotes a gap:

```
left  = 101 111 --- 000
right = 100 111 011 ---
```

Given a proposed alignment, if two blocks are aligned, the weight function `w` of the blocks returns `matchWeight` times the number of bits at which they agree plus `mismatchWeight` times the number of bits at which they disagree. So for the default values,

```
w 111 111 = 2 × 3 + (-1) × 0 = 6
w 101 100 = 2 × 2 + (-1) × 1 = 3
```

If a block is aligned with a gap, the weight is the block size times `gapWeight`. Continuing the example,

```
w --- 011 = (-1) × 3 = -3
```

A dynamic programming algorithm is used to align the blocks to maximize the total weight. A workspace array `a` is built, such that `a[i,j]` represents the best alignment of `left[0..i]` with `right[0..j]`. That is, `a[i,j]` is determined by checking whether the total weight is greatest when

- the best alignment of `left[0..i-1]` with `right[0..j-1]` is extended by matching `left[i]` with `right[j]` (northwest)
- the best alignment of `left[0..i]` with `right[0..j-1]` is extended by appending a gap to `left` to match with `right[j]` (west)
- the best alignment of `left[0..i-1]` with `right[0..j]` is extended by appending a gap to `right` to match with `left[i]` (north)

The top row `a[0,..]` and left column `a[..,0]` are filled with `0`s. To complete `a`, go across row `i`, sweeping `j` through column indices, then go on to row `i+1` and so on. In addition to the partial weight sum, `a[i,j]` includes a mark indicating which option (northwest, west, or north) led there. Once `a` is completely filled in, backtracking from the bottom right corner yields the alignment information.

### 3.2.4 Configuration file for chromosome alignment

The configuration file for chromosome alignments is `Alignment.yaml`. The YAML structure is parallel to `BlockAlignmentSpec`. As an example:

```yaml
matchWeight: 2
mismatchWeight: -1
gapWeight: -1
alignmentBlockSize: 96
```

### 3.2.5 Standard interpretation

The bits of a chromosome are translated into instructions as follows. The implementation is in module `Digiostelium.Genome`.

The bits of a chromosome are called *nucleotide bits*. They are translated using a majority code to *codon bits*. A parameter called `expansionFactor` specifies how many nucleotide bits are considered for each codon bit; the usual value is 3. The expansion factor must be odd so that every group of bits has either a majority of `0`s or a majority of `1`s. The implementation is in package `bitutil`, `BitFiddle.c`.

*Design feature:* In DNA, triples of nucleotides are called *codons*, and they specify either an amino acid or stop transcription. Since there are $4^3 = 64$ possible codons and twenty or so amino acids, depending on the species, many *synonymous* codons map to the same amino acid. Some organisms show preferences for certain synonyms, called *codon bias*, for a variety of apparent reasons. For example, if a gene experiences

a single non-synonymous mutation that is favored by selection, the resulting codon cannot be one of the synonyms that differs by more than one substitution from its ancestor. Thus, certain synonyms may indicate that a gene has recently experienced selective pressure. The majority code from nucleotide bits to codon bits is incorporated into the simulation so that experiments on such phenomena may be conducted.

In living organisms, codon bias may not be perfectly selectively neutral. One synonym may yield faster transcription or translation, or cause the mRNA to adopt a different secondary shape, compared to other synonyms.

*Design feature:* The majority code makes each gene longer, and allows for synonymous mutations that are perfectly selectively neutral. That facilitates experiments involving neutral drift and genetic diversity.

In a standard interpretation, a chromosome is translated into a string of codon bits, which is split into blocks, each of which is translated into an instruction as specified by an InstructionSpec.

```
data InstructionSpec = InstructionSpec {
  numReactions :: Int,
  -- ^ how many actions are in an instruction
  patternSpec :: GenomeIntSpec,
  -- ^ how bit patterns in the genome are decoded
  thresholdSpec :: GenomeIntSpec,
  -- ^ how thresholds (positive or signed integers) in the genome
  -- are decoded
  deltaSpec :: GenomeIntSpec,
  -- ^ how deltas (signed integers) in the genome are decoded
  deltaSign :: DeltaSign
  -- ^ further options for the signs of reaction deltas
  } deriving (Read, Show, Eq, Ord, Typeable, Data, Generic)
```

A given InstructionSpec always requires a fixed number of codon bits, and that's where the block length comes from. Strings of bits are translated into integers as specified by a GenomeIntSpec; see below.

Given an InstructionSpec, an instruction is read from a string of codon bits as follows. Recall that a *pattern* is an index into a cell state array A. The switch pattern is read as specified by patternSpec from the first few bits. The threshold is read from the next few bits as specified by thresholdSpec. Then numReactions reactions are read from the remaining bits. The target pattern is read as specified by patternSpec. The value of delta is read according to deltaSpec, then modified according to deltaSign.

The possible values of deltaSign are:

```
data DeltaSign = JustDecode | ToggleByEvenOdd
```

- A value of JustDecode means to make no further modifications to delta.
- A value of ToggleByEvenOdd means that reactions are numbered, starting from 0, in the order they are read from the codon bits. For even numbered reactions, delta is exactly the value read according to deltaSpec. For odd numbered reactions, delta is minus that value.

A GenomeIntSpec specifies the encoding of an integer:

```
data GenomeIntSpec =
  Constant Int -- ^ always the same
  | TwosComplement Int -- ^ twos complement (no additional encoding)
  | ReflectedGrayEncoding Int -- ^ Gray code
  | PlusMinus
  | Offset Int GenomeIntSpec -- ^ add an offset
  | SumMany Int GenomeIntSpec -- ^ add several together
  deriving (Read, Show, Eq, Ord, Typeable, Data, Generic)
```

Here is what they mean.

**`Constant k`** An integer that occupies no bits. This option is available because one might want to restrict the δ of an instruction to ±k, in which case it doesn't need to be stored in the genome.

**`TwosComplement n`** Read n consecutive codon bits and interpret them as a standard twos-complement unsigned binary integer.

**`ReflectedGrayEncoding n`** Read n consecutive codon bits, convert them to a twos-complement unsigned binary integer k, then pick out the k-th item in the reflected Gray code (see below).

**`PlusMinus`** Read one bit, and map `0` to −1 and 1 to 1.

**`Offset d subEncoding`** Read j according to `subEncoding` and add d to the result. For example, `Offset (-4) (ReflectedGrayEncoding 3)` reads three bits, converts it to an integer in `[0,7]` through the Gray code, then translates it to a signed integer in `[-4, 3]`.

**`SumMany m subEncoding`** Read `m` integers using `subEncoding`, and add the result together.

*Design feature:* The reflected n-bit Gray code is a standard way of representing the integers 0 through $2^n - 1$ in such a way that the representation of $k$ and $k + 1$ differ by a single bit. For example, the 3-bit reflected Gray code is

```
 0   1   2   3   4   5   6   7
000 001 011 010 110 111 101 100
```

In evolutionary simulations, it is common to use a Gray code for integer magnitudes so as to avoid so-called Hamming barriers. For example, suppose that fitness is a function of a 3-bit integer $k$, specifically $f(k) = 20 - (k - 4)^2$, which has a maximum at $k = 4$. Using the twos-complement representation, if the population is saturated by genomes with $k = 3 = 011_2$, three simultaneous single-bit substitutions are required to improve them to $k = 4 = 100_2$, an extremely unlikely event. Using the Gray code, $3 = 010_G$, and a single substitution transforms it to $4 = 110_G$, so there is a much smaller valley in the fitness landscape, and evolution will locate the maximum much more quickly. There are many possible Gray codes, but the reflected one is standard and easily implemented.

*Design feature:* The `SumMany` encoding can be used when some range of values is required, but an n-bit encoding is not desired. For example, it may be that the desired range is 0 to 70-ish, and for some reason, a 7-bit encoding is problematic. An alternative would be to add up five 4-bit integers. The `SumMany` encoding can also smooth out the fitness landscape. Values near the middle of the resulting range have more possible representations, and are easier for evolution to locate.

*Design feature:* The reason for the `DeltaSign` feature is as follows. One could use `Offset` as described above to allow for positive and negative values of delta. However, there are single bit mutations that can change the sign of a delta. These mutations are nearly always very deleterious. Excessive potential deleterious mutations can slow down evolution. The use of `DeltaSign` allows for positive and negative deltas in reactions, but rules out these deleterious mutations, potentially speeding up the discovery of good genomes.

*Design feature:* There are many different ways of making the same range of instructions available. It isn't always clear which one is the "best" in a given situation, and some experimentation may be necessary. As noted, some encodings of integers may result in many possible deleterious mutations, which may be undesirable.

### 3.2.6 Example instruction specification

Here is an example of an `InstructionSpec`, expressed in YAML, as would be used as part of a genome configuration file.

```yaml
numReactions: 2
patternSpec:
  ReflectedGrayEncoding: 6
thresholdSpec:
  ReflectedGrayEncoding: 4
```

```
deltaSpec:
  Constant: 1
deltaSign: ToggleByEvenOdd
```

Each instruction encodes two reactions, the deltas of which are $\delta_0 = 1$ and $\delta_1 = -1$, respectively, and occupy no bits in the genome. The threshold is a 4-bit unsigned integer in the reflected Gray code. The switch pattern and the two target patterns $t_0$ and $t_1$ are 6-bit unsigned integer in the reflected Gray code.[5] So each instruction is of the form

```
If A[sp] ≥ ϑ then add 1 to A[t₀] and add -1 to A[t₁]
```

The 22 codon bits for a single instruction are laid out as follows

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
└sp──────────────┘└ϑ───────────┘└t₀──────────────┘└t₁──────────────┘
```

The majority-of-three code means there are 66 nucleotide bits per instruction, so a block size of 66 is used throughout.

## 3.3 Genome architecture specification

Genomes in the simulation are specified by a type gs that is an instance of the type class GenomeSpec. The essential features are as follows.

```
class GenomeSpec gs where
  type Genome gs :: Type
  type Mutation gs :: Type

  getInstructionSpec :: gs → InstructionSpec
  genomeLength :: gs → Genome gs → Integer
  applyMutations :: gs → [Mutation gs] → Genome gs → Genome gs
  ...
```

**Genome gs** A type for the concrete representation of a genome

**Mutation gs** A type for the concrete representation of a mutation

**getInstructionSpec** A function that returns the InstructionSpec for reading instructions from chromosomes.

**genomeLength gs genome** A function that returns the total length of a genome in nucleotide bits.

**applyMutations gs mutations genome** Apply a list of mutations to a genome and produce a new genome.

There are other members of type class GenomeSpec related to recording exactly where genes are located, enumerating all possible mutations, and converting genomes to and from JSON. See module Digiostelium.Genome for further details.

## 3.4 Haploid singleton genome architecture

This genome architecture is defined in module Digiostelium.HaploidSingletonGenome. It represents an organism with a single linear chromosome. Binary reproduction is accomplished by performing crossover on the chromosomes from each parent. There is no male-female distinction. It is not modeled on any specific species.[6]

The essential features of this genome architecture are as follows. Type HaploidSingletonGenomeSpec is an instance of type class GenomeSpec. It has these fields.

[5] Using a Gray code is not actually necessary for patterns since they are not treated as magnitudes, but I usually use it for all integers just to keep things consistent and to make it easier to summarize the encoding of an instruction.

[6] For reference: Bacteria usually have a single haploid chromosome and exchange genetic material routinely, though not usually by crossover. Liverworts have alternating generations, and the haploid gametophyte is the dominant form, but they have a male-female distinction.

```haskell
data HaploidSingletonGenomeSpec =
  HaploidSingletonGenomeSpec {
    expansionFactor :: Int,
    instructionSpec :: InstructionSpec,
    crossoverSpec :: CrossoverSpec,
    mutationSpec :: SingleChromosomeMutationSpec,
    initialNumGenes :: Int
    }
```

**expansionFactor** Expansion factor for the majority coding from nucleotide bits to codon bits. See Standard interpretation.
**instructionSpec** Specifies how to convert codon bits to instructions. See Standard interpretation.
**crossoverSpec** Specifies how to perform random crossovers (recombination). See Crossover.
**mutationSpec** Specifies how to perform random mutations. See Mutations.
**initialNumGenes** How many genes (instructions, blocks) are in a randomly generated genome for the first generation.

Here is an example of a complete `Genome.yaml` configuration file for this chromosome architecture. The YAML structure matches the `HaploidSingletonGenomeSpec` data structure. The instruction specification is the same as that in Example instruction specification.

```yaml
expansionFactor: 3
instructionSpec:
  numReactions: 2
  patternSpec:
    ReflectedGrayEncoding: 6
  thresholdSpec:
    ReflectedGrayEncoding: 4
  deltaSpec:
    Constant: 1
  deltaSign: ToggleByEvenOdd
crossoverSpec:
  crossoverBlockSize: 66
  crossoverType:
    SinglePointAligned:
      alignmentBlockSize: 66
      matchWeight: 2
      mismatchWeight: -1
      gapWeight: -1
mutationSpec:
  minimumBlocks: 4
  mutationBlockSize: 66
  pDelete: 1.0e-3
  pDuplicate: 1.0e-3
  pToggle: 5.0e-3
initialNumGenes: 32
```

The expansion factor of 3 means codon bits come from a majority-of-three decoding of the nucleotide bits. Since instructions are 22 codon bits long, a block size of 66 nucleotide bits is used throughout. Agents in the first generation are initialized with a single chromosome of 32 instructions, with all bits independently chosen, with equal chances for 0 or 1. Once a chromosome is 4 blocks long or shorter, no further block deletion mutations are applied.

## 3.5 Diploid genome architecture

The diploid architecture is designed to mimic the genome structure common to most higher organisms. See module `Digiostelium.PolyploidGenome` for the implementation.

The genome consists of several pairs of chromosomes called *bundles* of *strands*.

*Design feature:* The *di-* in *diploid* refers to the fact that there are *two* strands per bundle. Much of the code can handle arbitrary polyploidy, which occurs commonly in certain species of plants and is also important in cancer. Thus, the implementation uses *polyploid* in most cases. However, some code assumes two strands per bundle, and may have to be modified in the future if there is a need for true polyploidy.

Reproduction proceeds as follows. In each parent, a diploid germ cell is produced by applying random mutations to the parent's genome. A single chromosome is produce from each bundle through crossover, resulting in a haploid gamete. The two gametes are combined to produce a diploid zygote, which is the child.

A diploid genome is specified by a `PolyploidGenomeSpec` object.

```
data PolyploidGenomeSpec =
  PolyploidGenomeSpec {
    expansionFactor :: Int,
    instructionSpec :: InstructionSpec,
    crossoverSpec :: CrossoverSpec,
    mutationSpec :: SingleChromosomeMutationSpec,
    pGameteChromosomeUnrecombined :: Double,
    initialNumGenes :: [(Int, Int)],
    initialBundlesHomogeneous :: Bool
    }
```

**`expansionFactor`** Expansion factor for the majority coding from nucleotide bits to codon bits. See Standard interpretation.

**`instructionSpec`** Specifies how to convert codon bits to instructions. See Standard interpretation.

**`crossoverSpec`** Specifies how to perform random crossovers (recombination). See Crossover.

**`mutationSpec`** Specifies how to perform random mutations. See Mutations. In a future version, whole chromosome mutations may be implemented, such as duplication, deletion, and fusion.

**`pGameteChromosomeUnrecombined`** Probability that no recombination is performed when a gamete is produced from a germ cell.

**`initialNumGenes = [(m₀, n₀), (m₁, n₁), …]`** How many genes (instructions, blocks) are in a randomly generated genome for the first generation. $(m_j, n_j)$ means that bundle $j$ has $m_j$ strands, each of which has $n_j$ blocks.

**`initialBundlesHomogeneous`** `True` means the strands within a bundle are the same in randomly created genomes. `False` means they are created independently. In living organisms, the strands in a bundle are usually very similar, and can be aligned almost perfectly. However, some experiments were done in which the strands were initialized independently, hence this setting.

Here is an example of a complete `Genome.yaml` configuration file for this chromosome architecture. The YAML structure matches the `PolyploidGenomeSpec` data structure. The instruction specification is the same as that in Example instruction specification.

```
expansionFactor: 3
instructionSpec:
  numReactions: 2
  patternSpec:
    ReflectedGrayEncoding: 6
  thresholdSpec:
    ReflectedGrayEncoding: 4
  deltaSpec:
```

```
    Constant: 1
  deltaSign: ToggleByEvenOdd
crossoverSpec:
  crossoverBlockSize: 66
  crossoverType:
    SinglePointAligned:
      alignmentBlockSize: 66
      matchWeight: 2
      gapWeight: -1
      mismatchWeight: -1
pGameteChromosomeUnrecombined: 0
mutationSpec:
  minimumBlocks: 4
  mutationBlockSize: 66
  pDelete: 1.0e-3
  pDuplicate: 1.0e-3
  pToggle: 5.0e-3
initialBundlesHomogeneous: true
initialNumGenes: [[2,16], [2,16]]
```

The expansion factor of 3 means codon bits come from a majority-of-three decoding of the nucleotide bits. Since instructions are 22 codon bits long, a block size of 66 nucleotide bits is used throughout. Once a chromosome is 4 blocks long or shorter, no further block deletion mutations are applied. Agents in the first generation are initialized with two pairs of chromosomes, each of which has 16 instructions. Within each pair, the nucleotide bits in the first chromosome are chosen independently, with equal chances for 0 or 1, and the second chromosome is an exact duplicate of the first.

A third pair could be added by using

```
initialNumGenes: [[2,16], [2,16], [2,16]]
```

The chromosomes do not have to be the same length. For three pairs, with length 16, 17, and 20 blocks, use

```
initialNumGenes: [[2,16], [2,17], [2,20]]
```

In the future, it may be possible to specify a triploid genome, as in

```
initialNumGenes: [[3,16], [3,16]]
```

but such a genome architecture is not fully implemented.

It is typical to specify `pGameteChromosomeUnrecombined: 0` unless you are specifically experimenting with evolution without recombination.

It is typical to specify `initialBundlesHomogeneous: true` unless you specifically want the first generation to be initialized with non-identical chromosomes within pairs, in which case it is probably a bad idea to use an aligned crossover type.

# 4 Selection

## 4.1 Agents and eggs

Some types from module `Digiostelium.Agent` must be introduced here.

An `Agent` is an object with these fields

```
data Agent genome rating =
  Agent {
    identifier :: Integer,
```

```
    rating :: rating,
    lifespan :: Lifespan,
    egg :: Egg genome
    }
```

Agents are stored in database files as the simulation runs. Each agent has a unique identifier, a number assigned in sequence as agents are created. It has a rating, which comes from its performance on a problem. It has a lifespan, which is either the generation at which it was born, or a tuple of a birth generation and a death generation. Agents that haven't died yet only have a birth generation.

Each agent also has an `Egg`:

```
data Egg genome =
  Egg {
    parentIds :: [Integer],
    genome :: genome
    }
```

An `Egg` object is an incomplete agent, in that it has a genome, and a record of its parents, but no rating or other information. The list of parent IDs is empty for the egg of an agent that was created entirely at random in the initial generation of a population.

The term `Egg` was chosen instead of something like `Seed` or `Spore` because a purpose of this simulation is to understand the evolution of neuro-computation and audio-visual communication, both of which are more strongly associated with animals than other types of organisms.

## 4.2 Distinction between an `Agent` and a `Colony`

An `Agent` is essentially a genome that has been annotated with information about where it came from and how well it performs on the problem in question. During the rating process, the genome is used to build a `Colony` object within an `Arena`, so in this sense a `Colony` is the phenotype of the genome.

A complication to that terminology is that multiple `Colony` objects are usually created for each genome. Many problems effectively require the artificial organism to perform in an `Arena`, then be reset, then perform again in an `Arena`, and so on, with different input conditions in the `Arena` each time. The genome's rating is a sum of some function of those performances, and is eventually recorded in an `Agent` object and stored to a database file. The simulation is written in Haskell, which is a functional programming language, so it's more natural to create fresh `Colony` and `Arena` objects for each performance, than to create just one `Colony` and reset its state before each performance. So many `Colony` objects may be created for each genome, and it is perhaps more accurate to think of each of them as representing a state of the phenotype, incorporating its experience within a particular run of an `Arena`.

To give a biological analogy, imagine that an egg hatches in a specific environment and grows into an animal. That is parallel to producing a `Colony` from a genome and allowing it to run in an `Arena`, and the final `Colony` object represents the animal's final state. Now imagine that we rewind time, place the egg in a different environment, and again it hatches and grows into an animal. The final animal phenotype will be in a different state because the environment is different. That is parallel to producing a fresh `Colony` object, running it in a different `Arena`, extracting the final `Colony` object, and finding that it differs from the final `Colony` object from the previous `Arena`.[7]

## 4.3 Selection process

Mutations were described in the Mutations section. The other part of evolution is the selection process. See module `Digiostelium.Evolution`. Selection is configured with a `SelectionSpec` object:

[7] I apologize for any confusion the terminology causes. If it is any consolation, remember that biologists have trouble defining terms like "gene," "species," and "organism" with the kind of precision that computer science normally demands.

```haskell
data SelectionSpec =
  SelectionSpec {
    selectFromBlocks :: [SelectFromBlock],
    numToKeep :: Int
    }
```

Every problem comes with a way to assign a numerical rating to individual agents. Each generation is sorted by rating, then by age, with younger agents placed before older agents. The `numToKeep` field specifies how many agents from the top of the list are kept for the next generation, thereby implementing *elitism*.

*Design feature:* When `numToKeep > 0`, the maximum rating never decreases from one generation to the next. An agent that achieves the highest rating seen so far is a *landmark*, and does not die until enough other agents with at least as high of a rating have been born to push the landmark agent down past the `numToKeep`-th slot in the list. This makes it possible to specify that the simulation should continue until a landmark agent with a specified rating has died, which indicates that the population has found a highly rated solution, and that variants of it have spread through the population.

*Design feature:* If older agents are sorted before younger agents, the following undesirable behavior arises. An agent is born that achieves a higher rating than any other. It will be listed first until there is another innovation. Its successful descendants will be listed immediately after it, in decreasing order of age, until `numToKeep` of them have been born. Once that happens, the first `numToKeep` agents are fixed until the next innovation, which means that a large chunk of the population is fixed, limiting exploration of the fitness landscape. Younger agents must be listed before older agents to prevent this kind of stagnation. The first `numToKeep` agents will eventually all be highly rated, but they will eventually be replaced by their descendants, which gives the population more freedom to explore.

New agents are drawn according to the list of `SelectFromBlock` objects:

```haskell
data SelectFromBlock =
  UniformBlock {
    numOffspring :: Int,
    numToBreed :: Int
    }
  | Tournament {
    numOffspring :: Int,
    pTakeHigher :: Double
    }
```

**UniformBlock** From the first `numToBreed` agents in the sorted list, select `numToBreed` pairs of parents uniformly at random and produce an offspring agent from each pair. Selection is stronger when `numToBreed` is smaller.

**Tournament** Pick `numToBreed` pairs of parents through tournament selection and produce an offspring from each pair. The tournament is to pick two agents from the entire population, and with probability `pTakeHigher`, choose the one with the higher rating. Two such tournaments are required for each breeding pair. Selection is stronger when `pTakeHigher` is closer to 1. It must be between 0 and 1.

*Design feature:* It is possible, though generally unlikely, that the same parent agent is selected twice and serves as both parents. This is analogous to plants that can self-pollinate.

Child agents are produced according to each `SelectFromBlock` in `selectFromBlocks`. The resulting new agents are rated, mixed in with the ones kept from the previous generation, and re-sorted. The size of each generation is the total of `numToKeep` and the sum of the `numOffspring` field of each `SelectFromBlock`.

*Implementation details:* The main selection code is in module `Digiostelium.Evolution`, specifically,

```haskell
instance (...) Sample (SelectionSpec, genomeSpec, problem, Map Integer (Agent genome rating) where
    pullSample (...) = ...
```

The `pullSample` function is a method in the type class `Sample`, and pulls a random sample from a given

distribution. The distribution in this case is parameterized by the SelectionSpec, a genome spec, a problem, and the population of living agents expressed as a Map from agent identifiers to Agent objects. It sorts the population. It pulls random samples of Egg objects. It calls the rateEggs function for the particular problem to rate these eggs. The final result is a PopulationBirthDeathDelta object, that specifies which agents die at the end of this generation, and a list of eggs and their ratings. The nextGeneration function in module Digiostelium.Evolution applies the PopulationBirthDeathDelta object to the current generation to produce the next generation. Along the way, it does a lot of record keeping, such as completing the Egg objects into Agent objects, updating the time step of death for agents that die, and recording the clock time required to compute the next generation. It saves the new generation to database files and deletes dead agents from the current generation file.

### 4.4 Example configuration

Here is an example configuration file, Selection.yaml, the structure of which is parallel to a SelectionSpec object:

```yaml
selectFromBlocks:
- Tournament:
    numOffspring: 150
    pTakeHigher: 0.6
numToKeep: 100
```

Under this configuration, for each generation, the 100 highest rated, youngest agents are kept from the previous generation. A total of 150 new agents are produced from tournament selection. When choosing a parent, two living agents are picked, uniformly at random, and the one with the higher rating is selected with probability 0.6. Each generation therefore consists of 100 + 150 = 250 agents.

## 5 Problems

A *problem* specifies the environment in which evolution takes place. Agents are rated on their performance in an Arena as specified by a problem object.

There are several different problem types. For each problem type, the following related types must be defined; see Digiostelium.Problem.

```haskell
type family Rating problem :: Type
type family Location problem :: Type
type family Signal problem :: Type
```

The Rating must be an ordered semigroup whose binary operation is interpreted as combining two partial ratings. As an example, the rating could be a numeric type with the + operation.

*Future:* I plan to relax the conditions to being a partially ordered semigroup, which would enable Pareto-optimal problems.

The Location type specifies what kind of location information is needed in the Arena.

The Signal type specifies what kinds of signals agents can broadcast and receive.

The following must also be defined for each problem type:

```haskell
class ArenaProblem problem where
  stopSignal :: problem → Maybe (Signal problem)
  minLandmarkRating :: problem → Rating problem
```

The optional stopSignal is a signal that an agent can emit to indicate that its computation is complete; not every problem uses this feature. The minLandmarkRating is for record-keeping purposes, and indicates that agents with ratings below this value should not be recorded in the landmark database.

A problem must specify how to initialize the arena, which requires the following definitions:

```
type family ArenaInitializationSpec problem :: Type

class (ArenaProblem problem) ⇒ InitializeArena problem where
  initialArenaState ::
    problem
    → ArenaInitializationSpec problem
    → [ColonyInitialState (Location problem) (Signal problem)]

class (ArenaProblem problem) ⇒ RateEggs problem where
  rateEggs ::
    (GenomeSpec genomeSpec)
    ⇒
    genomeSpec
    → problem
    → Map Integer (Agent (Genome genomeSpec) (Rating problem))
    → [Egg (Genome genomeSpec)]
    → RandomST s [(Egg (Genome genomeSpec), Rating problem)]
```

The `ArenaInitializationSpec` includes compiled genomes and other information necessary to initialize an arena. The `initialArenaState` function produces the list of initial colony states. The `rateEggs` function takes the current generation in the form of a `Map` from agent identifiers to agent objects, and a list of `Egg` objects to be rated. It produces a list of tuples of each egg and its rating, possibly consuming random numbers. It is specified this way because some problems rate agents based on multi-agent interactions, so the entire population must be available.

*Implementation details:* The `RandomST s` in the return type is a Haskell abstraction called a *monad*. In this case, it serves as an idiom for calculations that pass around mutable state, and is required because `rateEggs` may need a pseudo-random number generator whose state changes after each number it generates.

## 5.1 Input-Output problems

An *input-output problem* is an abstract problem that supplies a stream of input to an `Arena`, and the ratings of agents in the arena are calculated entirely based on the signals they broadcast and the resources they use during the computation. See module `Digiostelium.InputOutputProblem`.

The key definitions required for an input-output problem are declared in a type class,

```
class InputOutputProblem problem where
  type Input problem
  type Output problem
  problemInputToArenaInput ::
    problem
    → Input problem
    → (Int, [[BroadcastState (Location problem) (Signal problem)]])
  arenaOutputToProblemOutput ::
    problem
    → Seq [BroadcastState (Location problem) (Signal problem)]
    → Output problem
  inputsForRating ::
    problem → [Input problem]
  ...
```

Related `Input` and `Output` types for the problem must be specified, along with these functions:

**problemToArenaInput problem input** Convert an Input object to (k, bs) where k is the maximum number of time steps the arena should be run, and bs is a list whose j-th item is a list of the broadcast states that should be in effect during time step j. That list can be fed into an Arena object as it runs.

**arenaOutputToProblemOutput problem bs** From a sequence (which is basically an optimized list) whose j-th item is a list of the broadcast states in effect during time step j, produce a problem Output object

**inputsForRating problem** Return a list of all Input objects that should be used to rate agents.

## 5.2 Singleton problems

A *singleton problem* is an abstract problem where each agent is rated in isolation. No interaction between agents is required. A singleton problem must implement everything for an InputOutputProblem, along with the following:

```
class (InputOutputProblem problem)
  ⇒  SingletonProblem problem where

  initialColonyState ::
    problem
    → (ProgramInformation, ReactionProgram)
    → ColonyInitialState (Location problem) (Signal problem)

  rateResult ::
    problem
    → Input problem -- ^ a given input
    → Output problem -- ^ the resulting output
    → Int -- ^ total number of reactions performed
    → Int -- ^ total of absolute value of all reaction deltas
    → Rating problem -- ^ resulting rating
```

**initialColonyState problem compilerOutput** From a compiled genome, build an initial colony state. The initial arena state will be just this one colony.

**rateResult input output numReactions totalReactionDeltas** Return the rating of a agent that was fed the given input, produced the given output, performed the given number of reactions, and produced totalReactionDeltas along the way

For a given genome, a fresh colony state and arena are created and run for each input object in the problem's training set. The rateResult function computes a rating for each of those runs, and the total is the agent's overall rating.

# 6 Copy problem

## 6.1 Summary

The Copy(m,n) problem is: Given an m-bit *input word*, transmit a message over time through a smaller n-bit-wide channel, and reconstruct the input word. Agents have two cells, one designated the *sender* and the other designated the *receiver*. They are connected by n parallel synaptic channels.

Agents are rated by testing them on each of the $2^m$ possible input words. For each word, a colony built from the genome is put into an arena in isolation. An *input bit signal* is turned on for each bit in the input word that is turned on. The arena runs until either the colony emits a *stop signal*, or the maximum number of time steps has elapsed. An *output word* is read, whose j-th bit is on or off depending on whether or not the colony is emitting the j-th *output bit signal*. Points are awarded if the colony is emitting a *beacon* signal when it stops, for each bit in the output word that matches the corresponding bit in the input word,

for stopping after fewer time steps. Points are deducted for using many reactions and for a high total reaction delta. The agent's rating is the total of points earned on all possible input words.

The configuration object allows various parts of this problem to be turned of and on. The points awarded for accomplishing parts of the task and the deductions can also be adjusted.

## 6.2 Implementation

The implementation is in `Digiostelium.CopyProblemBase` and `Digiostelium.CopyProblem`.

*Implementation details:* The implementation of the `Copy` problem is separated in this way so that parts of it can be shared with the implementation of the PairCopy problem.

The `Copy` problem is a singleton problem, and an input-output problem.

The associated input type is

```
type instance Input CopyProblem = CopyProblemInput

data CopyProblemInput = CopyProblemInput {
  inputBits :: [Int] -- ^ list of inputs bits set
  }
```

The `inputBits` are the bit indices `i` for which the `i`-th bit in the input word is set, so `InputSignal i` is broadcast the whole time the arena runs; see below.

The associated output type is

```
type instance Output CopyProblem = CopyProblemOutput

data CopyProblemOutput = CopyProblemOutput {
  numTimeSteps :: Int,
  -- ^ how many time steps until the arena stopped
  nonDataOutputs :: [CopyProblemSignal],
  -- ^ output signals active at the stop time, other than the data bits
  dataBits :: [Int]
  -- ^ data bit signals active at the stop time
  }
```

The `dataBits` are the bit indices `j` for which `OutputSignal j` was being emitted when the arena stopped; see below. The `j`-th bit in the output word is set for each such `j`.

The associated signal type is

```
data CopyProblemSignal =
  SenderRole
  | ReceiverRole
  | Stop
  | Beacon
  | InputBit Int
  | OutputBit Int
  | TransmitBit Int

type instance Signal CopyProblem = CopyProblemSignal
```

**`SenderRole`** This signal is on at all times and is detected by the sender cell.
**`ReceiverRole`** This signal is on at all times and is detected by the receiver cell.
**`Stop`** This signal is emitted by the receiver cell to indicated that it has finished reconstructing the message.
**`Beacon`** This signal is emitted by the receiver cell and should be on as much as possible.
**`InputBit i`** This signal is detected by the sender.

**`OutputBit j`** This signal is emitted by the receiver.
**`TransmitBit k`** This is only used in the PairCopy problem.

The sender and receiver cells have sensors that detect *role* signals. This feature enables each cell's program to detect whether it is the sender or the receiver. Not every solution takes advantage of these signals, as the receiver can identify itself from the activity of the synapse.

Given an input word with bits `p[0], p[1], …, p[m-1]`, if `p[i] = 1`, then the signal `InputBit i` is turned on in the arena for the entire run, otherwise it's turned off.

The receiver is responsible for emitting several signals. It should always emit the `Beacon` signal. It should emit the `Stop` signal to indicate that the appropriate `OutputBit` signals are turned on. The output word `q[0], q[1], …, q[m-1]` is formed by setting `q[j] = 1` if `OutputBit j` is turned on.

The location type is

```
type instance Location CopyProblem = NullLocation
```

because no particular location is involved. Signals are always at maximum strength.

The rating type is `Double`, wrapped in the `Sum` type, so that addition is used to combine scores for different input words.

## 6.3 Beacon subtask

The bit copying task is primary. Emitting the beacon signal is a secondary task. It can be solved easily, with just one instruction. It's included for experiments involving genetic diversity and phylogenetic calculations. If the genome encoding permits it, the population will usually (but not always) develop a single instruction that solves the beacon subtask, and most agents will have a variant of it. Since every genome includes one of these homologous genes, it can be used to measure genetic distance between individuals.

## 6.4 Configuration

The configuration object is defined as follows.

```
data CopyProblem = CopyProblem {
  wordSize :: Int,
  synapseSize :: Int,
  minSteps :: Int,
  maxSteps :: Int,
  correctnessFactor :: Double,
  inputWordIndexToPattern :: Int,
  inputActivation :: Int,
  synapseInputIndexToPattern :: Int,
  synapseOutputIndexToPattern :: Int,
  synapseThreshold :: Int,
  synapseCost :: Int,
  synapseDelay :: Int,
  synapseActivation :: Int,
  outputWordIndexToPattern :: Int,
  outputThreshold :: Int,
  outputCost :: Int,
  beaconPattern :: Int,
  beaconThreshold :: Int,
  beaconCost :: Int,
  beaconPoints :: Double,
  minReactions :: Double,
```

```
  reactionCostFactor :: Double,
  minTotalDeltas :: Double,
  deltaCostFactor :: Double,
  roleActivation :: Int
  }
```

**wordSize = m** How many bits are in an input word and an output word.

**synapseSize = n** How many synaptic channels there are from the sender to the receiver.

**maxSteps** Maximum number of time steps the arena runs on an input word.

**roleActivation** The sender and receiver have sensors that detect the `SenderRole` and `ReceiverRole` signals, respectively. The sender sensor is hardwired to add `roleActivation` to `A[1]` on each time step when the `SenderRole` signal is on. The receiver sensor is hardwired to add `roleActivation` to `A[2]` on each time step when the `ReceiverRole` signal is on. These sensors are hardwired to affect `A[1]` and `A[2]` because there hasn't been a reason to make their targets configurable.

**inputWordIndexToPattern, inputActivation** During a time step when the sender detects `InputSignal i`, `inputActivation` is added `A[s]`, where `s = i + inputWordIndexToPattern`.

**synapseInputIndexToPattern, synapseThreshold, synapseCost** A `True` bit is pushed into synaptic channel `x` when `A[s] ≥ synapseThreshold`, otherwise a `False` bit is pushed. The value of `s` is `synapseInputIndexToPattern + x`. When a `True` bit is sent, `synapseCost` is subtracted from `A[s]` to represent the energy required to activate the synapse.

**synapseDelay** This is how many bits long the synaptic channel is. It takes this many time steps for a bit to move from the sender to the receiver.

**synapseActivation, synapseOutputIndexToPattern** When a `True` bit in synaptic channel `x` arrives at the receiver, `synapseActivation` is added to `A[t]`, where `t = synapseOutputIndexToPattern + x`.

**outputWordIndexToPattern, outputThreshold, outputCost** The receiver has emitters for bits `0` through `m-1`. Emitter j emits the signal `OutputBit j` when `A[e] ≥ outputThreshold`, where `e = outputWordIndexToPattern + j`. When that happens, `outputCost` is subtracted from `A[e]` to represent the energy required to emit the signal. Furthermore, when `A[0] ≥ outputThreshold`, the `Stop` signal is emitted, and `outputCost` is subtracted from `A[0]`. The stop signal emitter is hardwired to read `A[0]` because there hasn't been a reason to make its source pattern configurable.

**beaconPattern, beaconThreshold, beaconCost** The receiver has an emitter that emits the `Beacon` signal when `A[beaconPattern] ≥ beaconThreshold`. When that happens, `beaconCost` is subtracted from `A[beaconPattern]`.

**minSteps, correctnessFactor, beaconPoints, minReactions, reactionCostFactor, minTotalDeltas, deltaCostFactor** Used to compute rating.

## 6.5 Details of the rating function

The rating of an agent on a single input word is computed by the `rateResult` function in module `Digiostelium.CopyProblem`:

```
instance SingletonProblem CopyProblem where

  rateResult ... = ...
```

For a given run of the arena with a particular input word:

- The *correctness score* is the number of bits in an output word that match the corresponding bit in the input word.
- The *time score* is `maxSteps - (max minSteps numTimeSteps)`. That is, the first `minSteps` time steps don't affect the colony's score. The time score counts the number of unused time steps beyond `minSteps`. If a colony doesn't emit the stop signal, all time steps are used, and the time score will be zero. If a colony emits the stop signal before `minSteps` have elapsed, it earns the maximum time score of `maxSteps - minSteps`.

- The *beacon score* is `beaconPoints` if the colony was emitting the beacon signal when the arena stopped.
- The *average reaction count* is the number of reactions performed, divided by the number of time steps.
- The *average total deltas* is the total reaction deltas divided by the number of time steps.
- The *reaction count cost* is `reactionCostFactor` multiplied by the average reaction count minus `minReactions`, except that a negative result is replaced by 0. A colony that averages fewer than `minReactions` per time step takes no penalty.
- The *reaction delta cost* is `deltaCostFactor` multiplied by the average total deltas minus `minTotalDeltas`. A colony that averages fewer than `minTotalDeltas` per time step takes no penalty.

The rating score from that run is

- `correctnessFactor` times `maxSteps` times the correctness score
- plus the time score
- plus the beacon score
- minus the reaction count cost
- minus the reaction delta cost

Multiplication of `correctnessFactor` by `maxSteps` is included for compatibility with an earlier version of this simulation, and may be changed in a future version as it isn't strictly necessary. The same effect could be achieved by using a different `correctnessFactor` and not multiplying by `maxSteps`.

## 6.6 Example configuration

Here is an example of a `Problem.yaml` configuration file.

```
beaconCost: 0
beaconPattern: 31
beaconPoints: 100000
beaconThreshold: 80
correctnessFactor: 100
deltaCostFactor: 0.001
inputActivation: 1
inputWordIndexToPattern: 8
maxSteps: 100
minReactions: 20
minSteps: 20
minTotalDeltas: 400
outputCost: 0
outputThreshold: 80
outputWordIndexToPattern: 12
reactionCostFactor: 0.001
roleActivation: 1
synapseActivation: 1
synapseCost: 0
synapseDelay: 0
synapseInputIndexToPattern: 3
synapseOutputIndexToPattern: 4
synapseSize: 1
synapseThreshold: 10
wordSize: 2
```

The word size is `m = 2` and the synapse size is `n = 1`, which means agents are expected to transmit the input words `00`, `10`, `01`, and `11` from the sender, through a single-bit synapse, to the receiver, and reconstruct it. The two bits in the input word, $p_0$ and $p_1$, are fed into `A[8]` and `A[9]`, respectively, because `inputWordIndexToPattern = 8`.

Suppose the arena is testing the agent on the input word `10`, so $p_0 = 1$ and $p_1 = 0$. On each time step, the arena broadcasts the `InputSignal 0`, but not `InputSignal 1`. Because that signal is detected by a sensor the sender cell, the `inputActivation` of 1 is added to `A[8]` in each time step.

> *Alternatively:* If the input word was `01`, the `InputSignal 1` would be broadcast, and 1 would be added to `A[9]` instead.

There is one synaptic channel, because `synapseSize = 1`. The synapse is configured so that if `A[3] ≥ 10` in the receiver, a `True` bit is pushed into the synaptic channel. The condition is

`A[synapseInputIndexToPattern + 0] ≥ synapseThreshold`

Since `synapseCost = 0`, nothing is subtracted from `A[3]`, which forces the genome to implement its own resetting mechanism. Since `synapseDelay = 0`, a `True` bit that gets pushed into the synapse arrives at the receiver at a later phase in the same time step. When that happens, `synapseActivation = 1` is added to `A[4]`

> *Alternatively:* If there were a second channel (`synapseSize = 2`), its condition would be `A[4] ≥ 10`, as in
>
> `A[synapseInputIndexToPattern + 1] ≥ synapseThreshold`
>
> and in the receiver, the arrival of a `True` in the channel would add 1 to `A[5]`, as in
>
> `A[synapseOutputIndexToPattern + 1]`

In the receiver, when `A[12] ≥ 80`, it emits `OutputSignal 0`, and when `A[13] ≥ 80`, it emits `OutputSignal 1`, as specified by

```
A[outputWordIndexToPattern + 0] ≥ outputThreshold
A[outputWordIndexToPattern + 1] ≥ outputThreshold
```

There is no change to `A[12]` or `A[13]` when this happens, since `outputCost = 0`.

During each time step, `roleActivation = 1` is added to `A[1]` in the sender and to `A[2]` in the receiver.

This configuration also enables the beacon subtask. Agents are awarded `beaconPoints = 100000` points on each input word when `A[31] ≥ 80 = beaconThreshold` when the arena stops running.

This problem configuration is intended to be used with a `Genome.yaml` configuration file that includes this instruction specification:

```yaml
instructionSpec:
  numReactions: 2
  patternSpec:
    ReflectedGrayEncoding: 6
  thresholdSpec:
    ReflectedGrayEncoding: 6
  deltaSpec:
    Offset:
    - 1
    - ReflectedGrayEncoding: 4
  deltaSign: ToggleByEvenOdd
```

This differs from the previous example instruction specification in several ways. The threshold is encoded by 6 bits instead of 4, so it takes on values from 0 to 63. The deltas are in the range 1 to 16, because they

are specified by 4 bits with an added offset of 1. The wider range of deltas permits a single instruction solution to the beacon subtask, such as this example from an actual sample run:

```
A[2] ≥ 4 ⇒ A[31]+=13 A[21]+=-9
```

This instruction reads the receiver role pattern A[2]. In the receiver, 1 is added to A[2] in each time step, so this instruction is active after 4 time steps. This came from a sample using a diploid genome architecture, and there are two copies (instructions 107 and 164), one on each sister chromosome. After a few more time steps, A[31] ≥ 80, and from then on, the beacon signal is turned on.

The skeleton of a colony under this configuration is displayed in the following diagram:

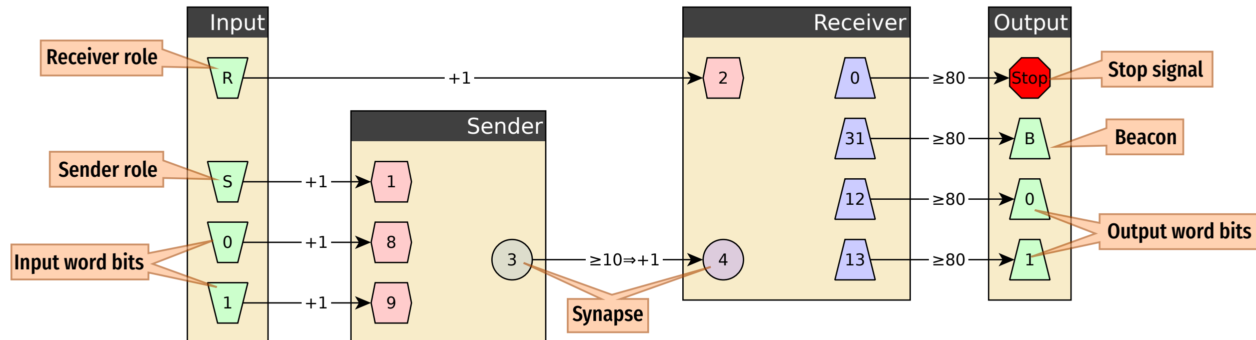


Figure 1: Diagram of the parts of a colony determined by the Copy problem configuration

The Input box contains a green trapezoid node for each signal that is broadcast as input to the colony. The Sender and Receiver boxes represent the two cells of the colony, and numbered nodes inside those boxes represent elements of the state array A. The shapes and colors of those nodes indicate elements of A that have some special function, such as being part of a sensor, emitter, or synapse. The Output box contains a green trapezoid node for each signal that is emitted by the colony, plus a red octagon node for the stop signal. Solid arrows with black heads between nodes are abbreviations of parts of the problem configuration. For example, the arrow from 3 in the Sender box to 4 in the Receiver box represents the synapse, and shows that if A[3] ≥ 10 in the sender, 1 is added to A[4] in the receiver during that time step.

The population must evolve genomes whose instructions connect the internal nodes of the sender and receiver and solve the Copy problem.

## 6.7 Example solution

Here is an example solution from an actual run (12/Problem-difficulty/Copy/Phase1/Samples/000, agent 4153408). The efficacious reactions are as follows:

```
Sender:
    110 : A[ 9]≥ 3 ⇒ A[ 3]+= 8
    124 : A[ 3]≥20 ⇒ A[40]+=12
    127 : A[ 8]≥ 1 ⇒ A[ 3]+=14
    147 : A[40]≥57 ⇒ A[ 3]+=-14
    147 : A[40]≥57 ⇒ A[ 8]+= 2
    181 : A[ 3]≥20 ⇒ A[40]+=12
    184 : A[ 8]≥ 1 ⇒ A[ 3]+=14
    184 : A[ 8]≥ 1 ⇒ A[ 8]+=-9
    204 : A[40]≥57 ⇒ A[ 3]+=-14

Receiver:
```

```
21  : A[ 4]≥ 5 ⇒ A[13]+=16
21  : A[ 4]≥ 5 ⇒ A[58]+=-2
58  : A[58]≥ 3 ⇒ A[ 4]+=15
63  : A[ 4]≥ 4 ⇒ A[51]+=16
64  : A[19]≥ 8 ⇒ A[12]+=16
80  : A[ 4]≥ 5 ⇒ A[13]+=16
80  : A[ 4]≥ 5 ⇒ A[58]+=-2
107 : A[ 2]≥ 4 ⇒ A[31]+=13
112 : A[ 2]≥11 ⇒ A[ 0]+=16
112 : A[ 2]≥11 ⇒ A[ 4]+=-14
164 : A[ 2]≥ 4 ⇒ A[31]+=13
169 : A[ 2]≥11 ⇒ A[ 0]+=16
169 : A[ 2]≥11 ⇒ A[ 4]+=-3
190 : A[51]≥34 ⇒ A[19]+=12
214 : A[19]≥44 ⇒ A[51]+=-4
214 : A[19]≥44 ⇒ A[58]+= 2
```

The solution can be displayed to show its nature as a reaction network.

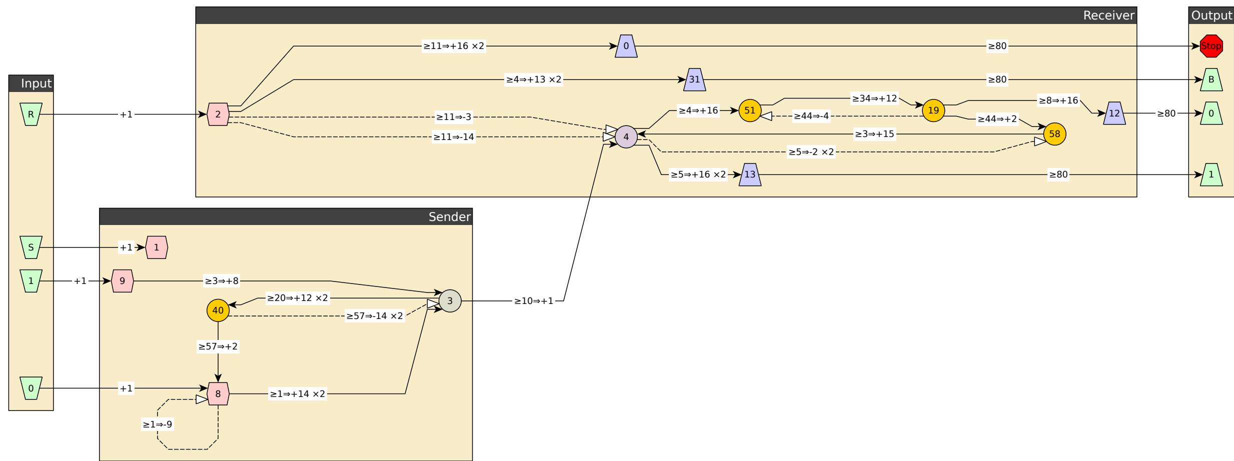


Figure 2: Reaction network diagram

Efficacious reactions are represented by arrows; other reactions are not drawn. As an example, consider instructions 147 and 204 of the sample solution:

```
147: A[40] ≥ 57 ⇒  A[8] += 2, A[3] += -14   bundle 1, strand 0, bits 3940-3936
cod: 001111 101001 001100 1000 010000 1101
nuc: 100010011110111111 111100011000000101 010010111111000010 110000100001 000111010010010001 101110000101
                                                                       ↑

204: A[40] ≥ 57 ⇒ A[55] += 2, A[3] += -14   bundle 1, strand 1, bits 3840-3936
cod: 001111 101001 001101 1000 010000 1101
nuc: 100000011110011111 111100011000000101 010010111011000011 110000100001 000101010010100000 101111001101
                                                                       ↑
```

The codon bits and nucleotide bits are listed under the instructions, and their locations in the genome are given. The two genes are twin alleles on sister chromosomes. They differ at a few bits. The only significant difference is the one marked with the arrow, which is what makes the difference between the 8 and the 55 in the first reaction of each instruction. These instructions are shown in isolation in the following figure.

Elements A[j] that have no special function are drawn as a j inside a yellow circle, such as the A[40] here.

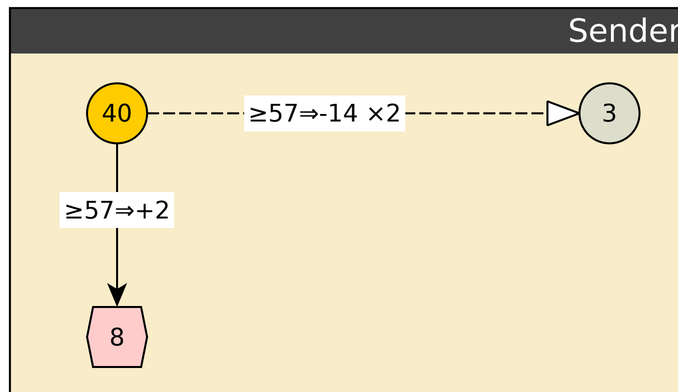


Figure 3: Instruction notation example

Other shapes and colors are used for elements of A[j] that do have some special function, as explained before. So in the abbreviated diagram, there are nodes for A[40], A[8], and A[3].

Both arrows involve instruction 147. The arrow to the node labeled 8 is marked ≥57⇒+2. It represents the part of instruction 147 that checks whether A[40] ≥ 57, and if so, adds 2 to A[8]. The 40 is not written on the label because it's implied by the node label at the foot of the arrow; likewise, the 8 is implied by the node label at the head of the arrow. The arrow is solid with a black head to indicate that the reaction it represents is excitatory, meaning it adds a positive number to the target element of A.

The arrow to the node labeled 3 is marked ≥57⇒−14 ×2. It represents the part of instruction 147 that checks whether A[40] ≥ 57, and if so, adds -14 to A[3]. It is dashed with a white head to indicate that the reaction it represents is inhibitory, meaning it adds a negative number to the target element of A. The ×2 on the end of its label means that there are two instructions that specify exactly the same reaction, the other one being 204. Duplicate reactions are common in diploid genomes, and rather than clutter the diagram with multiple parallel arrows, the duplication is shown in the label. This genome is heterozygous, in that the twin alleles (instructions 147 and 204) are not identical, and it happens that the first reaction in instruction 204 is not efficacious. That's why there's no ×2 on the excitatory arrow.

# 7 PairCopy problem

# 8 Configuration files