CS2112—Spring 2012

Final Project

Simulating Evolving Artificial Life

Version of April 30, 2012

The final programming assignments for this course make up a final project that you will work on as part of a two-person team. In this project, you will simulate a simple world of animals that wander around, eat food, reproduce, and evolve. This world will include a graphical visualization that enables a user to take control of individual animals. Different species can also interact in this simulated world. Because of evolution, there will be multiple species eventually even if initially there was only one.

The animals, called critters, are located on a regular, hexagonal grid. Plant-like food grows on the surface of the world. Critters must find and eat it to stay alive. Critters may also attack each other; when critters run out of energy, they die and become food. If they get enough energy, they can reproduce.

The genome of a critter is actually an event-driven program that determines what it does on a given time step. When critters reproduce, the genome is copied from the parent or parents to the new critter, with possibly some mutations applied. This means that critter programs may change over time and perhaps evolve to make them more effective.

The critter simulation will be implemented as a networked Java service with a graphical front end that is a Java applet. This design permits multiple users to view and interact with the same virtual world from their web browsers.

As part of this project, you will build a simple parser and interpreter for the critter language. You will build a graphical user interface. You will design your own abstractions and data structures. You will become very familiar with the Java libraries. You will learn how to build Java applets and distributed programs using Java.

0 Changes to the spec

The following changes have been made since HW6.

- 4/16: Clarified sensing of food (see HW7).
- 4/19: Clarified plant growth.
- 4/22: Added `random` expression.
- 4/23: Changed what happens if no rule is satisfied on a given pass.
- 4/23: Specified that mutation can change attributes too, not just rules.
- 4/25: Plant growth slowed down.
- 4/26: Clarified use of `MAX_SMELL_DISTANCE`.
- 4/30: `BUD_COST` and `MATE_COST` were both increased significantly to prevent energy creation.

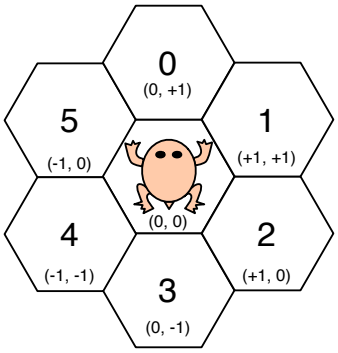

Figure 1: Tiles and directions

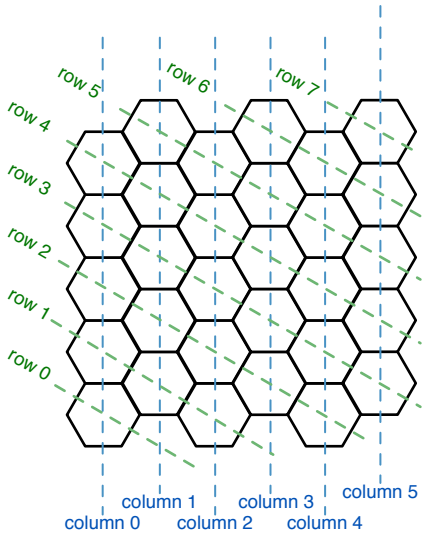

Figure 2: Rows and columns of hexes

1 The world

The world that critters live in is a large array of hexagonal tiles called *hexes*. A given critter is at any moment located on one of these hexes and facing in one of the six possible directions, as shown in Figure 1. The world advances in discrete time steps. On each time step, a critter may perform one of several possible actions, or simply update its internal state and wait for the next time step.

Each hex may contain up to one object, where an objects is either a critter or a rock. Rocks are inert obstacles that critters cannot move onto or over. Hexes that do not contain rocks may contain a plant (possibly in addition to a critter). It may also contain food, which is produced when a critter dies.

Plant hexes can provide food to critters. When a critter eats the plant on its hex, the plant is destroyed. However, every hex containing a plant will, with a small probability on each turn and for each of its neighboring hexes that does not already contain a plant, generate a new plant. Note that a plant may generate a plant on more than one adjacent hex on a given turn, but generation of plants on its adjacent hexes occurs independently. Plants provide a fixed amount of energy when eaten (`ENERGYPERPLANT` = 100). In addition, a few plants spontaneously appear at random on non-rock hexes on each turn.

Hex coordinates. Each hex is identified by a coordinate (c, r) where c is the column and r is the row on which the hex is located. The lower-left (southwest corner) hex is at coordinate (0,0). Figure 2 shows the row and column coordinates of various hexes. Moving in one of the six possible directions changes the row and column coordinate of the critter. The corresponding adjustments to row and column coordinates are shown in Figure 1.

Every hex has coordinates (c, r) where c and r are nonnegative integers. However, some such coordinates lie outside the world. A coordinate such that $c \not\leq$ `MAX_COLUMN` lies off the east edge

of the world. A coordinate where $c \not\leq 2r$ lies off the south edge of the world. And a coordinate where $c + (2 \cdot \texttt{MAX\_ROW} - \texttt{MAX\_COLUMN}) \not\geq 2r$ lies off the north edge of the world. For example, if Figure 2 showed the entire world, we would have $\texttt{MAX\_COLUMN} = 5$ and $\texttt{MAX\_ROW} = 7$.

You might notice that there is a third potential "coordinate" we could specify, corresponding to lines of hexes slanting upward toward the right. We can define the "slice" s of a given hex as $(r - c)$, increasing as we move upward along a column or "northwest" along a row. Using slices, we can compactly represent the length of the shortest path between two hexes:

$$\max\{|\Delta c|, |\Delta r|, |\Delta s|\} = \max\{|\Delta c|, |\Delta r|, |\Delta c - \Delta r|\}$$

A coordinate that lies outside the world acts in all ways as though it is a hex containing a rock. Critters cannot fall off the edge of the world, and they see rock when they look off the edge.

Time The simulation proceeds in time steps. During each time step, each critter is allowed to take one turn. These turns are taken sequentially, so each critter sees the updates cause by all critters that have taken a turn during the current time step. The order in which critters take turn is fixed. Newly created critters are added to the end of the order.

Being early in the ordering is not a significant advantage because other behavior of the simulation occurs throughout a given time step. In particular, plant growth occurs randomly between critter turns rather than all at once at the beginning of the time step. However, since simulated time is measured in time steps rather than in critter turns, the probability that plants grow in a given critter turn must be scaled down according to the number of critters taking turns.

2 Critter actions

The allowed actions are the following:

- **Wait.** The critter waits until the next turn without doing anything.
- **Move forward or backward.** A critter may move forward to the hex in front of it or backward to the hex behind it. If it attempts to move and there is a critter or rock in the destination hex, the move fails (but still takes energy)
- **Turn.** It may rotate 60 degrees right or left. This takes little energy.
- **Eat.** A critter may eat all food that is sitting on its current hex. It gains energy from any food that it eats.
- **Attack.** It may attack the critter in front of it, assuming there is one. The attack removes an amount of energy from the attacked critter that is determined by the size and offensive ability of the attacker and the defensive ability of the victim.
- **Grow.** A critter may use energy to increase its size by one unit.
- **Bud.** A critter may use a large amount of its energy to produce a new, smaller critter with the same genome (modulo mutation).

- **Mate.** A critter may attempt to mate with another critter in front of it. For this to be successful, the critter in front must also be facing toward it and attempting to mate in the same time step. If mating is successful, both critters use energy to create new size 1 critter containing a merge of their genomes.

3 Critter state

Critter state consists of several attributes in addition to the program that drives the critter. The critter has a current location in the world and a current direction, represented as an integer in 0..5, as shown in Figure 1. It also has a current size and energy. The critter also has some fixed attributes: its offensive and defensive ability, the size of its memory, and the rules governing its behavior.

Each critter has a derived attribute, its *complexity*. This is a weighted sum of the number of its rules, the size of its memory, and its offensive and defensive abilities. The energy of certain actions the critter performs depends on its complexity. The formula for complexity is given below.

4 Critter memory

Each critter has a memory called `mem`, which is an array of fixed length containing integers. The first few entries in this array have a meaning that is the same for all critter species:

- `mem[0]`: the length of the critter's memory (immutable)
- `mem[1]`: defensive ability (immutable)
- `mem[2]`: offensive ability (immutable)
- `mem[3]`: size (variable, but cannot be assigned directly)
- `mem[4]`: energy (variable, but cannot be assigned directly) other critters. Must be a positive number.)
- `mem[5]`: appearance (assignable but must be positive; affects appearance to other critters).
- `mem[6]`: rule counter, explanation below (variable, but cannot be assigned directly).

There are three kinds of memory entries: immutable entries that never change, variable entries that reflect the current state of the critter but that the critter's rules cannot assign to, and general-purpose mutable entries that can be both read from and assigned to by critter rules.

The entry `mem[5]` contains the critter's appearance, which defines how it looks to other critters. The appearance can be used as a way to signal to other nearby critters what the current critter is up to. It might also be exploited by predators to learn what the current critter is up to!

The size of the critter's memory must be at least 7 to accommodate these entries. If the size is larger, the remaining entries, indexed starting at 7, are general-purpose entries.

program → *rule**

rule → *condition* `-->` *command* `;`

command → *update** *update-or-action*

update-or-action → *update* | *action*

update → `mem [` *expr* `] :=` *expr*

action → `wait` | `forward` | `backward` | `left` | `right`
| `eat` | `attack` | `grow` | `bud` | `mate`

condition → *conjunction* (`or` *conjunction*)*

conjunction → *relation* (`and` *relation*)*

relation → *expr rel expr* | `{` *condition* `}`

rel → `<` | `<=` | `=` | `>=` | `>` | `!=`

expr → *term*

term → *factor* (*addop factor*)*

factor → *atom* (*mulop atom*)*

atom → `<number>` | `mem [` *expr* `]` | `(` *expr* `)` | *sensor*

sensor → `nearby [` *expr* `]` | `ahead [` *expr* `]`
| `damage` | `food` | `random [` *expr* `]`

addop → `+` | `-`

mulop → `*` | `/` | `mod`

Figure 3: Grammar for critter rules

5 Rule language

The grammar for the rules is given as a context-free grammar in so-called EBNF (Extended Backus-Naur Form[1]) in Figure 3). In EBNF grammars, the right-hand side may be a regular expression. EBNF does not add any real expressive power to context-free grammars, but makes them easier to express concisely. Terminal symbols are shown in typewriter font. Terminal symbols whose lexical representation is not fixed are shown using angle brackets: for example, `<number>`. Non-terminals are shown in italics, like this: *program*.

When it is a critter's turn, it finds the first rule in its list of rules whose condition is true. It then performs all of the updates on the right-hand-side of the rule, along with the action, if any. If the command for the rule contains no action, the process repeats: it again finds the first rule whose condition holds and performs its command. If no rule's condition holds, nothing is done on this pass through the rule. This process is performed up to 1000 times, after which the critter automatically performs a `wait` action. If on any pass no rule has a satisfied condition, the critter

[1]Despite its name, EBNF is apparently not due to either Backus or Naur!

stops performing passes and selects `wait` as its action for this turn.

The special memory location `mem[6]` reports which pass through the rules is being done. It has the value 0 on the first pass through the rules, 1 on the second pass (if any), then 2, and so on. It starts over again at 0 on the critter's next turn.

It may be possible to accelerate running the rules on later passes by only checking rules whose condition could possibly have become true. However, this optimization is not required.

6 Sensing

A critter can sense its immediate surroundings using *sensor* expressions as described in the grammar.

- The expression `nearby[`dir`]` reports the contents of the hex in direction dir, where $0 \leq dir \leq 5$. Here the direction is relative to the critter's current orientation, so 0 is always immediately in front, 1 is 60 degrees to the right, and so on. (If d is out of bounds, its remainder when divided by 6 is used.) The contents are reported as a number n, as follows:
  - 0: the hex is completely empty
  - $n > 0$: the hex contains a critter with appearance n.
  - $n < -1$: the hex contains no critter, and total food value is $-n$ (including the plant on the hex, if any).
  - $n = -1$: the hex contains a rock.
- The expression `ahead[`$dist$`]` reports the contents of hex that is directly ahead of the creature at distance $dist$, using the same scheme as `nearby`. Thus, `ahead[0]` reports on the hex under the critter, ignoring the critter itself.
- The expression `damage` reports on the total damage to another critter in the previous time step. (Note that it is possible to keep track of the damage to oneself by monitoring `mem[4]`.)
- The expression `food` uses the critter's sense of smell to report the direction and distance to the nearest food or plant, up to a distance of `MAX_SMELL_DISTANCE` (= 10) hexes. The result of the expression is $6 \cdot distance + direction$, where direction is relative to the critter's current orientation. If the food is not precisely in one of the six directions, the direction closest to the direction to the food is used in this expression. Ties between two directions are broken in an implementation-defined manner. If there is no food within `MAX_SMELL_DISTANCE` hexes, the result is 0. If there is food on the critter's current hex, the value is 1.

  (Note that the notion of the direction and distance to the nearest food or plant has been refined as part of HW7.)
- The `random` expression generates a random number from 0 up to the value of the given expression, exclusive. Thus, `random[2]` gives either 0 or 1 randomly. For $n < 2$, `random[`n`]` always is zero.

7 Energy and size

A critter has an initial size of 1 but can increase this through the `grow` action. Size affects energy expenditure but also the critter's effectiveness at some actions. Size also determines the maximum energy of the critter. For each point of size, the critter can hold `ENERGY_PER_SIZE` = 500 points of energy. Attempts to increase energy beyond this point result in excess energy being discarded.

If energy ever goes to (or below) zero, the critter dies. Its death adds to the food on its hex a number of food points equal to `FOOD_PER_SIZE` (= 200) points per point of size.

8 Attacking and defending

When one critter attacks another, some damage is done to the defending critter (the victim). This subtracts energy from the victim. If the victim's energy goes to zero (or lower), the victim dies and is turned into an amount of food proportional to its size.

When one critter attacks another, the damage done depends on the sizes of the two critters. If critter 1 attacks critter 2, and S_1 and S_2 are the sizes of the corresponding critters, O_1 is the offensive ability of critter 1, and D_2 is the defensive ability of critter 2, the energy removed from critter 2 is:

$$\texttt{BASE\_DAMAGE} \cdot S_1 \cdot P(\texttt{DAMAGE\_INC} \cdot (S_1 \cdot O_1 - S_2 \cdot D_2))$$

where `BASE_DAMAGE` $= 100$, `DAMAGE_INC` = 0.2, and $P(x)$ is the *logistic function*:

$$P(x) = \frac{1}{1 + e^{-x}}$$

This formula means that critters do damage proportional to their size, but that they do only half their maximum damage if they are evenly matched against the defending critter. Damage falls off quickly to zero when attacking a critter with a higher effective defense.

9 Mutation

When a critter's genome is copied to a new critter, the copy may be perfect. But with probability $p = 0.5$ there will be at least one mutation. When copied, a random number in $[0, 1]$ should be generated. If it is less than p, a mutation should be applied. The mutation process should then repeat, until a number greater than p is randomly generated.

A mutation is either a change to an attribute or a change to the rule set, with each equally probable. The attributes that may change are the size of the memory and the offensive and defensive abilities. A change to an attribute is an increase or decrease, chosen with equal probability, to one of these three attributes, chosen with equal probability. One exception is that changes to attributes never reduce them below their minimal legal value (7 for memory size, 1 for offense and defense).

A mutation to the rule set is performed by choosing randomly among the following:

- removing a rule
- duplicating a rule

- interchanging the order of two rules
- mutating a rule

A mutation to a rule is chosen randomly with equal probability between changing its condition and changing its command.

A mutation to a condition or a command is applied to one of the nodes in the AST of the condition or command. The choice of node to which the change is applied is made randomly with equal probability for all nodes. Once the node is chosen, once of the following mutations is applied, with equal probability:

- reversing the order of operands
- replacing an expression with one of its operands (if possible)
- changing the operation or action to another operation or action that is legal in the current position
- for numeric literal constants, changing to another literal constant
- replacing an operand with a copy of another randomly chosen expression found in the program
- replacing an operand with a randomly generated expression whose subexpressions, if any, are either 0 or a copy of another randomly chosen expression already in the program.

Numeric literals are adjusted by adding a randomly chosen value to the existing constant. The constant is adjusted up or down by the value (where legal) of `java.lang.Integer.MAX_VALUE/r.nextInt()`, where `r` is a `java.util.Random` object.

10 Budding and mating

When a new critter is created by budding, it appears directly behind the critter doing the budding. When two critters mate, it appears directly behind one of the two critters, chosen at random.

When a new critter is created by budding, its rules are copied from its parent, modulo possible mutation. Its attributes are also copied from the parent, with the exception of energy, size, and appearance. Energy is set to a constant `INITIAL_ENERGY` = 250, size is always set to 1, and appearance is always set to 1. All memory locations above index 6 are set to zero in the newly created critter.

When two critters mate, however, they exchange genetic material to form the new critter. Attributes 0-2 are chosen from between the two critters randomly. Attributes 3–5 are chosen as for budding. The new rule sequence is chosen by picking the corresponding rule in sequence from either the 'mother' or the 'father', at random. If the mother or father have different-sized rule sets, the new rule set either has the size of the mother or the father, randomly chosen. Thus, if the mother and father have identical genomes, and there are no mutations, the child will have the same genome too.

11 Energy

Different actions take different amounts of energy, even waiting for a turn. The energy cost of different actions is as follows:

- `wait`, `turn`, `eat` : energy equal to the critter's size.
- `forward` and `backward`: energy equal to the critter's size times `MOVE_COST` where `MOVE_COST` = 3.
- `eat` : the same energy as `wait`.
- `attack` : energy equal to the critter's size times `ATTACK_COST` = 5.
- `grow` : energy equal to $size \cdot complexity \cdot$ `GROW_COST` where `GROW_COST` $= 1$.
- `bud` : `BUD_COST` $\cdot\, complexity$ energy, where `BUD_COST` = 9, and
- `mate` : `MATE_COST` $\cdot complexity$ energy, where `MATE_COST` = 5.

Several of these energy costs depend on the critter complexity. If r is the number of rules in the critter program and *offense* and *defense* are the critter's offensive and defensive abilities, the critter complexity is equal to:

$$r \cdot \texttt{RULE\_COST} + (offense + defense) \cdot \texttt{ABILITY\_COST}$$

Most actions take the same energy whether they are successful or unsuccessful. One exception is the `mate` action, which only costs as much as `wait` if it is unsuccessful.

12 Constants

Numbers used in this document are mostly parameters, with symbolic names. We may fiddle with the values of these parameters to make the simulation more interesting, so you should always use the symbolic names rather than hard-coding them into your program. We will provide a file containing the current values of these constants; your program should parse the file at run time to set them to the correct values.

The current values of the simulation constants are shown in Figure 4.

13 Challenges

This project has several very different kinds of subsystems. One of the major challenges will be to keep the different parts of the system separate, so that, for example, your simulation code is entirely separate from your graphics code. This particular separation will be crucial for making an applet version of the program, since the simulation and the displays will be done on a completely different machines. Similarly, you will want to separate different parts of the simulation into different modules. The interpretation of critter rules should be kept separate from the mechanics of the world and even of the critter itself.

Name	Value	Description
BASE_DAMAGE	100	The multiplier for all damage done by attacking
DAMAGE_INC	0.2	Controls how quickly increased offensive or defensive ability affects damage
ENERGY_PER_PLANT	100	How much energy is obtained by eating a plant
ENERGY_PER_SIZE	500	How much energy a critter can have per point of size
FOOD_PER_SIZE	200	How much food is created per point of size when a critter dies
MAX_SMELL_DISTANCE	10	The maximum distance at which food can be sensed
ROCK_VALUE	-1	The value reported when a rock is sensed
MAX_COLUMN	99	The maximum column index in the world map
MAX_ROW	139	The maximum row index in the world map
MAX_RULES_PER_TURN	1000	The maximum number of rules that can be run per critter turn
PLANTS_CREATED_PER_TURN	2	How many plants are created from nothing on each simulation time step
PLANT_GROW_PROB	0.005	The probability of a plant sprouting an adjacent plant on a given simulation time step
MOVE_COST	3	Energy cost of moving (per unit size)
ATTACK_COST	5	Energy cost of attacking (per unit size)
GROW_COST	1	Energy cost of growing (per size and complexity)
BUD_COST	9	Energy cost of budding (per unit complexity)
MATE_COST	5	Energy cost of successful mating (per unit complexity)
RULE_COST	2	Complexity cost of having a rule
ABILITY_COST	25	Complexity cost of having an ability point
INITIAL_ENERGY	250	Energy of a newly birthed critter

Figure 4: Constants

Thoughtful design up front along with your partner will save you a tremendous amount of time later on. Meet early with your partner and decide on how you will structure your project and agree on interfaces and specifications that connect the different parts of the code.

14 Extensions

You may add extensions to the critter simulation, but you need not. Possible extensions might be additions to the critter language (abbreviations? function definitions?), to the critter model (better sensory capabilities?), changes to the world (volcanos? water? climate gradients?), better user control over the world view (zooming and panning? high-level critter commands?). Feel free to be creative. If your extensions might interfere with our testing, for example by making our critter programs invalid, be sure to support a command-line flag `-compatible` that turns off your extra features. We recommend being backward compatible to our specification in any case.