Marie Cho, Jason Xie
mhc2323, jx4334
September 20, 2024
CS314H

Prog 3 Report

1 Overview

Goals

This lab was focused on developing code to simulate critters in a 2D world, where each critter follows specific behaviors and interacts with other species. Through the implementation of algorithms for critter movement and interaction, explored reading and executing behavior instructions, utilizing API documentation, and testing if the methods are runnable/running as intended in a controlled simulation. By the end of the lab, we successfully developed a fully functioning simulation of diverse critter behavior.

Insights and Surprises

A few insights we gained from this lab were how specific conditions, such as the angle of movement or the presence of allies and enemies, significantly affect the critter's behavior as well as the result of others. For example, a critter's movement in a certain direction might not just depend on its current state (like being hungry or starved) but also on the surrounding environment, including the positions of other critters and their bearings. This complexity indicates how each unique condition, which consists of a multitude of external factors (including the 50% probability of the *ifrandom()* method), influences the critter's decisions and interactions. It can also lead to interesting scenarios where a critter might behave unexpectedly based on its environment, revealing the depth of the simulation. This lab also pushed us to consider edge cases, such as handling the critters reaching the grid boundaries or interacting with critters moving in the direction of their current position, as it could lead to unexpected and unrealistic behavior if unaccounted for. For instance, a critter moving beyond the grid's edge might need to turn around or stop, and interactions with such critters could cause conflicts in decision-making processes.

2 Solution

Assumptions

(a) Methods in the *CritterSimulator* class work as intended.
(b) Register numbers are within the range of 1 to 10.
(c) The behavior file has the correct format with the first line being the species name (index 0) followed by a list of instructions (index starting from 1).

(d) All instructions are written line by line and a blank line or null marks the end of the behavior file (no off-by-one index errors).

Design Considerations

When designing the solution for the Interpreter Class, I prioritized efficiency due to the fact that we needed to be able to handle an extensive set of repetitive commands. This is evident in our use of a switch-case structure, which offers a more efficient performance compared to a series of if-else statements. This allowed the interpreter to process commands rapidly, contributing to a smoother simulation experience.

Furthermore, we had to consider the data structures to use to optimize both performance and usability. We chose to use the *String[]* data type to contain the current line's behavior command and its parameters, if any, as it enabled a straightforward way to store and access the data.

Additionally, we incorporated error handling to properly manage potential failures and edge cases, including the wrong number parameters, null cases, and unparsable integers. By implementing error messaging and try-catch blocks in methods that check for these things, the program can smoothly handle these issues without crashing unexpectedly.

Finally, we created more helper functions for each behavior than initially given in the lab to not only increase code readability but also make it easier to test or add features to individual components. This modularity ensures that the codebase is both scalable and maintainable for future extensions or improvements.

Algorithms and Implementation

Each effect was designed with edge cases in mind and was addressed in *loadSpecies()* or in other helper methods, such as *hasNulls()* and *isNotInteger()*. Unexpected or uncommon scenarios, such as insufficient or invalid arguments (for any method) were handled through try-catch blocks, printing out an error statement, and stopping further execution of a behavior if an error was encountered. Implementations for each method are described below:

(a) executeCritter(): The code for the behavior commands of a critter is retrieved. Each line in the code, except for the first, is stored in a *String[]* array called *behaviorLine*, where the first element of this array represents the behavior name. If either the line or the behavior name is null, the current command execution is stopped. The method retrieves the current command from the critter's behavior line and calls the corresponding helper method. If none of the method names match, the command execution is terminated.

(b) loadSpecies(): This method reads species data from a file, including the name, behaviors, and the parameters associated with each behavior while ignoring any extra white space. This information is stored in a *List<String[]>* data structure to make it accessible during

execution. This sets up the *executeCritter()* method to correctly call the behavior methods using the parameters.

(c) hasNulls(): This method iterates through each element in the *behaviorLine* array. It checks each element for null values. If found, it prints an error message and returns *true*, signaling an error. If no nulls are detected, it returns *false*. This makes sure that the interpreter processes only valid commands and prevents runtime errors.

(d) isNotInteger(): This method takes a string input and attempts to parse it into an integer. If the parsing succeeds, it returns *false*, indicating that the string is a valid integer. However, if a NumberFormatException is thrown, either due to an empty string or non-numeric characters, it prints an error message and returns *true*, indicating that the string is not a valid integer. This makes sure that the interpreter processes only valid commands and prevents runtime errors.

(e) go(): This method checks for invalid arguments and, if none, calls the sub-helper method *goToLine()*.

(f) goToLine(): This method sets the next behavior command to a specific line number in the behavior sequence. If a '+' or '-' is present, the line number is adjusted by adding or subtracting from the current line number, respectively. If an 'r' is present, the line number is updated to the value set in the register number. If neither is the case, the line number is set to the provided parameter, if parsable.

(g) infect(): This method calls *CritterSimulator*'s *infect()* function, passing in the provided parameter, if any. If none are passed in, the parameter is defaulted to 1, meaning executing the commands again from the beginning. The *infect()* method changes an enemy's critter's state to reflect the infected status and becomes a member of the infecting critter's species.

(h) ifrandom(): This method calls *CritterSimulator*'s *ifRandom()* function. If it returns *true*, the critter will jump to a specified line, as provided by the parameter; if *false*, the critter continues to follow the behavior sequence.

(i) ifhungry(): This method calls *CritterSimulator*'s *ifHungry()* function. If it returns *HUNGRY* or *STARVING*, the critter will jump to a specified line, as provided by the parameter; otherwise, the critter continues to follow the behavior sequence.

(j) ifstarving(): This method calls *CritterSimulator*'s *ifStarving()* function. If it returns *STARVING*, the critter will jump to a specified line, as provided by the parameter; otherwise, the critter continues to follow the behavior sequence.

(k) ifempty(): This method calls *CritterSimulator*'s *getCellContent()* function, passing in the 1st provided parameter as the bearing. It checks whether the adjacent cell at the bearing angle is unoccupied, indicated by a return value of 0. If empty, the critter will jump to a specified line, as provided by the 2nd parameter; otherwise, the critter continues to follow the behavior sequence.

(l) ifally(): This method calls *CritterSimulator*'s *getCellContent()* function, passing in the 1st provided parameter as the bearing. It checks whether the adjacent cell at the bearing

angle is occupied by an ally, which returns ALLY. If so, the critter will jump to a specified line, as 2nd provided by the parameter; otherwise, the critter continues to follow the behavior sequence.

(m) ifenemy(): This method calls *CritterSimulator*'s *getCellContent()* function, passing in the 1st provided parameter as the bearing. It checks whether the adjacent cell at the bearing angle is occupied by an enemy, which returns ENEMY. If so, the critter will jump to a specified line, as provided by the 2nd parameter; otherwise, the critter continues to follow the behavior sequence.

(n) ifwall(): This method calls *CritterSimulator*'s *getCellContent()* function, passing in the 1st provided parameter as the bearing. It checks whether the adjacent cell at the bearing angle is occupied by a wall, which returns WALL. If so, the critter will jump to a specified line, as provided by the 2nd parameter; otherwise, the critter continues to follow the behavior sequence.

(o) ifangle(): This method calls *CritterSimulator*'s *getOffAngle()* function, passing in the 1st provided parameter as the bearing. It checks whether the bearing of the critter occupying that cell is equal to the 2nd provided parameter. If so, the critter will jump to a specified line, as provided by the 3rd parameter; otherwise, the critter continues to follow the behavior sequence.

(p) write(): This method sets the value of the register number, as indicated by the 1st parameter, to the 2nd parameter. The critter continues to follow the behavior sequence.

(q) add(): This method sets the value of the register number, as indicated by the 1st parameter, to the sum of the values in the registers indicated by the 1st and 2nd parameters. The critter continues to follow the behavior sequence.

(r) sub(): This method sets the value of the register number, as indicated by the 1st parameter, to the difference of the values in the registers indicated by the 1st and 2nd parameters. The critter continues to follow the behavior sequence.

(s) inc(): This method increments by 1 the value of the register number, as indicated by the 1st parameter. The critter continues to follow the behavior sequence.

(t) dec(): This method decrements by 1 the value of the register number, as indicated by the 1st parameter. The critter continues to follow the behavior sequence.

(u) iflt(): This method checks if the value of the 1st register number, as indicated by the 1st parameter, is less than the value of the 2nd register number, as indicated by the 2nd parameter. If so, the critter will jump to a specified line, as provided by the 3rd parameter; otherwise, the critter continues to follow the behavior sequence.

(v) ifeq(): This method checks if the value of the 1st register number, as indicated by the 1st parameter, is equal to the value of the 2nd register number, as indicated by the 2nd parameter. If so, the critter will jump to a specified line, as provided by the 3rd parameter; otherwise, the critter continues to follow the behavior sequence.

(w) ifgt(): This method checks if the value of the 1st register number, as indicated by the 1st parameter, is greater than the value of the 2nd register number, as indicated by the 2nd

parameter. If so, the critter will jump to a specified line, as provided by the 3rd parameter; otherwise, the critter continues to follow the behavior sequence.

3 Further Discussion

Scope

This project has real-world applications in fields such as robotics and artificial intelligence. These methods might be useful in building autonomous programs that need to check their surrounding environment, recognize allies vs enemies, and make decisions based on that information. Modern technologies that might implement this kind of logic are robot vacuum cleaners, autonomous cars, and surveillance drones or cameras. For example, Roombas need to know to turn when they run into a wall, cars need to avoid obstacles and control their speed, and cameras need to detect weapons in case of emergency.

However, the program faces limitations regarding scalability and model simplicity. While it handles calling helper methods for actions like infecting, turning, or executing specified behaviors for a small number of critters, the system can become computationally expensive to manage as the number of critters increases (to infinity). This highlights the need for scalability and optimization in larger simulations, including using different data structures and algorithms employed to ensure efficiency. Additionally, this lab does not capture the intricacies of real-life decision-making processes, particularly those that require quick and unambiguous responses like the examples mentioned above. Advancements that involve incorporating more advanced models, such as neural networks and machine learning techniques, could allow for more accurate and adaptive behaviors in such scenarios, but developing such models is beyond the scope of this project. Exploring this further in future work could lead to significantly improved results and applicability.

Solution Quality

The implemented methods succeeded in the following:

(a) Outputs had consistent results with no errors.
(b) Methods were able to handle edge/unique cases, including null or invalid inputs.
(c) Methods performed within a reasonable time frame.
(d) Code is clean, simple, and implements encapsulation for flexible changes in the future.
(e) The simulator displays the critters and their interactions correctly (changing color for *infect()*, disappearing for *eat()*, moving forward for *hop()*).

Interesting Results

One interesting result was the efficiency of the interpreter in managing multiple critters within the simulation. Even as the world became crowded with a wide variety of critters running distinct instruction sets, the interpreter maintained relatively quick processing times. This was a positive

indication of the efficiency of our code, showing that even in scenarios where many critters were performing complex behaviors simultaneously, the system performed within a quick time frame.

While the rover proved to be the most powerful critter, it often struggled with finding food, particularly toward the end of the simulation when food supplies were limited. Early on, it was able to locate food quickly, but as resources became scarce, it took an increasingly long time to locate and eat food. This result suggests that the simulation could be improved by implementing a more sophisticated method for critters to locate food, rather than relying on somewhat random commands at somewhat random times. This struggle highlights the potential for refining the way critters navigate and interact with their environment.

Our Critter

Our critter (under *Xie.cri*) works similarly to the given *FlyTrap*, but instead of spinning in place, it is always against a wall and hops only when there's an immediate threat. This predictable movement in response to threats is more efficient than *Rover's* use of random movement which is less focused and makes it more vulnerable to being infected. Furthermore, by only infecting enemies rather than eating them, our critter can neutralize enemies and spread its influence. Infecting converts enemies into allies, giving it a clear advantage. *Rover*, by alternating between infecting and eating, doesn't offer the same benefits. This strategy allowed our critter to overtake *Rover* even when the numbers were substantially unbalanced (ex: 30 *Xie* vs 300 *Rover*).

Problems Encountered

Initially, we used a *List<String>* data structure for the commands read during the *loadSpecies()* method and a *String* data structure for each behavior line being executed during the *executeSpecies()* method. However, this proved inefficient due to the variability in both the number and types of parameters being passed. Using a *String* necessitated splitting it into an array later anyway, which added unnecessary processing time.

Additionally, we encountered off-by-one errors with the line numbers in the behavior files that led to incorrect behavior execution. For example, while simulating the *Rover* critter, the program started executing with the *ifempty()* method (line 3) instead of the *ifenemy()* method (line 2). This mistake resulted from a misunderstanding that line numbers were based on a one-indexed system. This error required careful debugging as we had to trace the simulation line by line to ensure that the correct behavior commands were executed in the proper sequence, especially when the critter had to jump commands from line to line.

4 Testing

Test Procedure

Using JUnit, we implemented test methods for each behavior to verify whether the called method was executed correctly at the appropriate time. Using the same logic, we also checked to see if the methods would take care of invalid or insufficient arguments (ex: out of bounds, unparsable int) by checking if they printed an error message (white box testing). Finally, our test harness built a simple 3x3 grid to simulate the environment and check if the methods were actually running within the simulated world (black box testing). This holistic testing approach helped validate both individual method logic and the overall system behavior under various conditions.

Test Cases and Rationale

In our *InterpreterTest* file, we built a test method for each corresponding behavior in the *Interpreter* class (ex: *testHop()*), where we simulated the specific behavior and determined its functionality by either returning *true* or *false*. For methods that required specific parameters, we passed in various appropriate values (ex: 8 bearings, conditions HUNGRY vs. STARVING) and checked whether the method ran as intended. If the statement was asserted *true*, it confirmed that the *Interpreter* class was functioning as intended for that particular behavior.

In our `InvalidParameterTest` file, we tested various scenarios where the critter's behavior methods were given invalid inputs, such as too many parameters, not enough parameters, non-integer values, out-of-bounds line numbers, and invalid registers. The program would be then expected to print an error statement instead of crashing. This ensured that the program could handle faulty input and that our error-handling mechanisms were functioning correctly.

In our *CritterTest* file, we implemented a simplified version of the *Critter* interface to simulate how a critter would behave within a 3x3 grid environment. We created mock versions of actions like hopping, turning, eating, and infecting to verify whether these methods were being called appropriately without needing actual movement or infection. The environment was hardcoded as a 2D array, with the critter starting in the middle, allowing us to simulate how it interacted with adjacent cells. This setup enabled us to test how the critter's behavior methods worked in practice, ensuring that they executed properly and returned the expected results, such as detecting cell content and making directional choices.

4 Pair Programming

Programming Log

Sunday

(a) Jason drives for 30 minutes
(b) Marie drives for 30 minutes
(c) Jason drives for 15 minutes
(d) Marie drives for 15 minutes

Tuesday

(a) Jason drives for 45 minutes
(b) Marie drives for 45 minutes
(c) Jason works alone for 1 hour

Wednesday

(a) Jason works alone for 30 minutes

Thursday

(a) Jason drives for 30 minutes
(b) Marie drives for 30 minutes
(c) Jason works alone for 5 hours
(d) Marie works alone for 5 hours 30 minutes

Friday

(a) Jason works alone for 2 hours 20 minutes
(b) Marie works alone for 3 hours 30 minutes

Saturday

(a) Jason works alone for 2 hours
(b) Marie works alone for 2 hours 30 minutes

Reflection

In this lab, pair programming was an effective method for writing correct code with fewer bugs the first time. Collaborating allowed us to catch mistakes early, think of different edge cases, and discuss various approaches to solving problems. It was different from working alone in that since we were looking at and reviewing each other's code, we each suggested improvements that helped minimize logical errors from the start. Some difficulties we encountered were finding a solid amount of time to work on the project together as we had different commitments and schedules, but this improved our communication skills even when working remotely.

5 References and Citations

(a) Thank you to Andy Li for helping us with our project.