- Describe object hierarchy structure and how to design such a hierarchy of related classes.
- Describe the concept of object polymorphism in theory and demonstrate this concept in practice.
- Identify and describe the task and issues involved in the process of developing interactive products for people, and the techniques used to perform these tasks.

Part I

Introduction

Note: This problem is inspired by a problem encountered on DataCamp's "Data Science with Python" career course. (http"//www.datacamp.com)

Random walks. We have previously encountered the idea of a random walk in this module in the course notes and practical sessions. We're going to explore this concept in this part of the assessment.

Figure 1: A reference to Franz Kafka's *The Metamorphosis*. Illustration by Jiřího Slívy.

Cockroaches at work (or play?). Consider two cockroaches that are moving up and down the staircase visiting the 102 floors of the Empire State Building (the iconic building in New York City). In case you don't know, (some) cockroaches can fly, so can sometimes move very quickly. We are going to assume that these cockroaches move (somewhat) at random, and we want to examine some aspects of their movement.

In particular, suppose that each cockroach starts on the ground floor of the Empire State Building, which we will label as "floor 1". During a single timestep of movement, each cockroach could move up or down one or more floors, or could possibly stay stationary. Their movement is governed by a random process, which can be different for each cockroach. (These details are given below for the two cockroaches we will consider here.)

The types of questions we are usually interested in when considering random walks include:

1. How long does it take a cockroach to reach the top floor?
2. After walking for a certain number of timesteps, what is the highest floor that has been reached during that period?

3. How often are the two cockroaches on the same floor during their random walks (assuming they both start in the same place)?

4. How long is it necessary for the cockroach to walk before it is at a "random floor" of the building? (This depends upon trying to more precisely define what we mean by this statement.)

Since we're dealing with random processes, we really want to consider *averages* for these answers, i.e. if we repeat the process a large number of times (with different random inputs), what is, say, the *average* maximum height reached during a fix time period?

Even the use of the word "average" should really be clarified. Does this word refer to the "mean", the "mode", or the "median" (or some other concept)? If you aren't aware of these different meanings, you should really learn what they are as sometimes the word "average" is used in (say) news reports without really specifying which one is meant, or the word "average" is used in a misleading way. If I were to tell you that the mean salary of employees in a company is £60,000 and the mode of the salaries is £25,000, what might this suggest about the salary of the "average employee" of that company?

In any event, the goal of this part of the assignment is to write code to answer these types of questions posed about the random walks of the cockroaches in question. (Don't worry, I will define what "average" refers to when we get to the questions we want to answer...)

Ideally we can use Java's object-oriented capabilities (and maybe some polymorphism) to help us write our code as we are dealing with cockroaches all the time, but ones with some slightly different behavior in how they move about the Empire State Building.

Figure 2: The Empire State Building. Photo by Sam Valadi https://www.flickr.com/photos/132084522@N05/17339180506, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=62752443

Terminology. In what follows, I will use the word "timestep" or simply "step" to refer to one "unit of time", which is the amount of time necessary for a cockroach to move exactly once. It is possible that during this single move the cockroach could traverse several floors, or possibly remain stationary, but at the beginning of a time step the cockroach is on some floor, and at the end of the timestep it has moved once (or possibly remained stationary) and is again on some floor of the building (i.e. not "in between" floors).

The floors of the Empire State Building building are numbered sequentially starting from the "ground floor" (with value 1) to the "top floor" (numbered 102). (Note that while many tall American buildings don't have a floor numbered "13", the builders of the Empire State Building weren't superstitious, so it does have a "13th floor". Note also that the cockroaches don't have access to the secret 103rd floor of the Empire State Building.)

A cockroach can never go below the ground floor, nor above the top floor, so if it was to try a move that would take them too low or too high, it can instead move to the ground floor or top floor, but not beyond. (For example, if a cockroach was on floor 101 and was going to move up 3 floors, it would instead move to floor 102 and stop. If it started the timestep on the ground floor and wanted to move down, it would instead remain stationary for that step.)

Our subjects. We are going to consider two cockroaches, named Don and Bella. Don likes to show off his big wings and, thus, generally likes to move up more than down, but unfortunately Don is also a bit clumsy. In any single time step, this is how Don moves:

(1) With a 0.1% chance, Don flies into the center of the stairwell and ends up falling all the way back to the ground floor.

(2) Otherwise, Don rolls his 6-sided die (since we all know cockroaches love to carry around and use dice) and does the following:

   (a) On a 1 or a 2, Don moves down one floor.

   (b) On a roll of a 3, 4, or 5, Don moves up one floor.

   (c) On a roll of a 6, Don re-rolls his six-sided die and moves up the number of floors equal to this second roll of the die.

Figure 3: An artist's rendition of Don (or maybe Bella, I forget...).

Our other cockroach, Bella, has wings that are just as big as Don's and she also generally likes to move up rather than down, but Bella is lazy and doesn't usually move as fast as Don. Fortunately for Bella, she isn't clumsy like Don, but she does like to stop at the observation deck on the 86th floor to enjoy the view. Bella moves as follows:

(1) If Bella is pausing for a view on the 86th floor, she doesn't move at all for one step, and in the next step will roll her die (after all, she is a cockroach too!) to (possibly) move.

(2) Otherwise, Bella rolls her 6-sided die and does the following:

   (a) On a 1, 2, or 3, she moves down one floor.

   (b) On a 4, she moves up 2 floors.

   (c) On a 5, she moves up 3 floors.

   (d) On a roll of a 6, Bella doesn't move.

   (e) *However*, if Bella moves onto (or tries to move through) the 86th floor, she will stop on floor 86, and will pause for a view on the next step. (So if, for example, Bella was on floor 85 and rolled a 4, she would stop on floor 86 for a view on the next step, instead of progressing to floor 87.) If Bella rolled a 6, and was already on floor 86, she will still pause for a view on the next step. (Note that this means Bella will spend two time steps on floor 86, one for the step when she rolled a 6, and another when she pauses for the view, as described in (1) above.)

Remember that neither Don nor Bella can ever go below the ground floor (floor 1) nor above the top floor (floor 102), so if they try to do that, they will move as far as they are able and then stop.

Requirements

Your goal is to implement Java classes to represent the cockroaches, so that we can gather some emprical knowledge about the random walks in the Empire State Building. You should note the following:

(1) **(Requirement)** Develop an abstract class called "Cockroach.java". This class should have an abstract method called `takeStep()` (taking no parameters), which will be implemented by subclasses. This method defines what a cockroach does in a single timestep of movement. The Cockroach class should have an attribute for a `name` of the cockroach (a `String`), and can have other class attributes, constants, and methods as you deem appropriate. You can, for example have an attribute that keeps track of the number of steps the cockroach has taken since it started its random walk. You will, of course, need an attribute that keeps track of its current location (floor) in the Empire State Building, which the `takeStep()` method should update.

Be sure to include appropriate "get" or "set" methods as you need them and/or other methods you want to include in the Cockroach class.

(2) **(Requirement)** Each individual cockroach should be a subclass of "Cockroach.java". Only include additional attributes in the subclasses as you need them (i.e. in good object-oriented programming style, put as many of the attributes, methods, and constants into the base "Cockroach.java" class as you can).

Obviously each subclass will need to implement the `takeStep()` method above which determines how the cockroach moves in a single timestep of its movement. As mentioned, you can include additional atributes, constants, or methods beyond those in "Cockroach.java", but these should be integeral to a *particular* cockroach and not ones that could be included in that base class.

(3) **(Requirement)** You are going to output some summary statistics that you gather after repeating some experiments.

DO NOT USE an external library to compute these statistics! Write some (short) methods to compute these numbers as appropriate. Make certain the values that you output are **`double`** values.

Do the following experiments:

(a) For each of our two cockroaches, starting from the ground floor, have them each walk for 100 timesteps. (By "walk" I mean execute their `takeStep()` method.) Record the **maximum floor** that each cockroach reaches during the course of their walk.

Repeat this experiment 2000 times (for each cockroach) and report the average maximum height each cockroach reaches. By "average", I am referring to "arithmetic mean" (i.e. sum the values and divide by the total number of values).

(b) For each of our two cockroaches, starting from the ground floor, have them walk until they reach the top floor for the first time. Record the number of steps needed for them to do this.

Repeat this experiment 2000 times and report the arithmetic mean of these numbers (i.e. give the average time needed for each cockroach to reach the top floor).

(c) With Don and Bella starting on the ground floor, have them simultaneously execute their random walks for 2000 steps each, and count the number of times that they are on the same floor. This count has a minimum value of 1, since they are both on the ground floor (floor 1) at the start of time step 1.

Record this number of "co-occurrences". Repeat this experiment 2000 times and report the arithmetic mean of the number of co-occurrences.

(4) **(Requirement)** Name your main application program (the one reporting the statistics) as "RW.java".

Sample output

Here is some sample output for part (a) of the experiments. Obviously since there are random numbers involved, you should not expect your answers to exactly agree with mine. But with 2000 repetitions of the experiment, your answer and my answer should be close to each other.

```
$ java RW

----------
Don
----------
2000 experiments, walking 100 steps, the maximum height achieved
has average value: 74.971.

----------
Bella
----------
2000 experiments, walking 100 steps, the maximum height achieved
has average value: 39.7005.
```

A question to consider

Suppose that on a very windy day, there is a strong cross breeze on floor 95 due to some open windows. For each cockroach, again starting from the ground floor, they begin their random walk. If they land exactly on floor 95, they are blown out of the Empire State Building. If they make it to the top floor, they stop. What are the chances that they will make it to the top floor? (This probability will depend upon the particular cockroach in question...)

In your report, describe what kind of experiments you would run to determine this value, as well as giving the probability of reaching the top floor. Include your Java code in your submission.

What other kinds of interesting things could you report on?

Part II

Introduction

In Part II, you're again going to look at a random process, but this one is slightly different in nature. Since you have seen a little bit of graph theory in COMP108, we will talk about a random process on graphs here. The overall idea is that this random process models a series of interactions between people. When two people interact, they can "cooperate" for their mutual benefit, one could "cooperate" and the other "defect" (so the "defector" gets more benefit than the "cooperator"), or both can "defect" (meaning they get less than they would if they both cooperate).

This terminology comes from the so-called "Prisoner's Dilemma" which has been much studied in the world of game theory, showing how two rational people might not cooperate, even if it seems to be in their best interests to do so. (There are many explanations of the Prisoner's Dilemma online. See this Wikipedia article for an introduction https://en.wikipedia.org/wiki/Prisoner%27s_dilemma.)

As I said, we'll be modelling a series of interactions between people (the so-called "Iterated Prisoner's Dilemma"). People can't necessarily interact with just anyone, but only certain other people. The possible interactions between people are modelled by a graph. In particular, we're going to look at two different graphs, namely the *Path* and the *Cycle*.

A *path* on n vertices is just a collection of n vertices that form a "line". We can think of these as vertices labelled $v_0, v_1, \ldots, v_{n-1}$, where v_i is joined to v_{i+1} for $i = 0, 1, n-2$. A path with n vertices is often denoted as P_n.

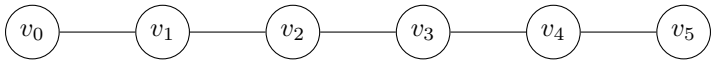


Figure 4: The path P_6.

A *cycle* (or ring) is a closed path, i.e. we add an edge between vertex v_{n-1} and v_0 to "close the loop". A cycle with n vertices is often denoted as C_n.

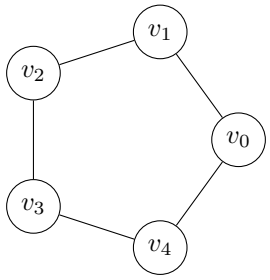


Figure 5: The cycle C_5.

Conveniently for us, each of these can actually be represented by a simple array in Java (provided we do this in the right way), so we don't need to do anything particularly complex to model this random process I am going to describe.

Each vertex in our graph (either P_n or C_n) is also going to have a label (or value) associated with it, either a 1 or a -1. (Note that I have deliberately chosen to label the vertices as I have, so that I can think of storing the vertex labels in a Java array, i.e. the label of v_i can be stored in the Java array element $v[i]$.)

Each vertex represents a person in our process. People can interact with each other if they are joined by an edge. For example, in the cycle, vertex v_i can interact with v_{i-1} and v_{i+1} and no one else. Note that these indices "wrap around" in the cycle, so that v_{n-1} can interact with v_0. What happens in our (random) process is that we select a random edge in the graph, and the two vertices on that edge interact with one another (described below). We can accomplish the same thing (instead of choosing a random edge) in the path by choosing a random vertex from the set $\{v_0, v_1, \ldots, v_{n-2}\}$, and have that vertex interact with the vertex "to their right". Note that we omit choosing the vertex v_{n-1} in this selection as they have no one "to their right").

In the cycle, we choose any vertex v_i and have them interact with the vertex "to their right" v_{i+1}, noting that the vertex "to the right" of v_{n-1} is v_0.

When two vertices interact in one round, it can cause them to update their labels. A label of 1 (or $+1$) means that someone will "cooperate", while a -1 means someone will "defect". If two cooperaters interact, they will each cooperate on their next interaction (with any person they can interact with). If one vertex cooperates and one defects, they will both be defectors in their next interaction (with anyone). The idea is that the "cooperator" feels "cheated" and will then defect the next time he/she interacts with someone. And if two defectors interact, they will both become cooperators. (They realize that they could do better by cooperating, so will try to cooperate in their next interaction.) See Figure 6 for a partial example on P_6.

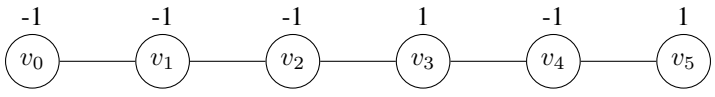
If you think about this process, what this means is that when two vertices interact, they replace their own label by the *product* of their label and the other label of the vertex they are interacting with. In some cases this means that the labels don't actually change (if they are two 1s, they will stay 1s), but we still count that as a step in our random process.

I hope it's clear that eventually (after some number of interactions), this process can end (and *will* end, given enough time), in the sense that every vertex will eventually have a label of 1, after which no further changes in labels can occur. Please convince yourself that this is true when the graph is a cycle or a path. (Indeed, for any *connected graph*, this process will eventually end, i.e. all labels will eventually become 1 and then never change.)

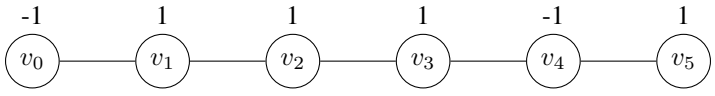
What we want to study is *how long* it will take for this random process to end, i.e. how many interactions will it take until every vertex in the graph has a label of 1? This could possibly depend upon how many -1 labels there are at the start of the process and/or where those -1s start in the graph. As you will see, it will also depend upon whether the graph is a path or a cycle.

Requirements

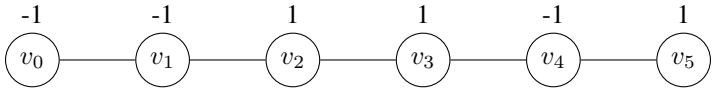
1. You're going to write a program (or a set of classes) to perform this random procedure above. As in Part I, you could write an abstract class for this process on a (generic) graph and then define subclasses



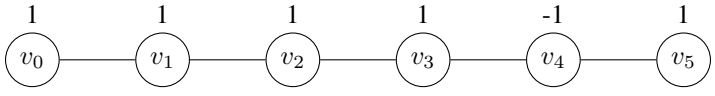
(a) Edge $\{v_1, v_2\}$ selected.



(b) Edge $\{v_0, v_1\}$ selected.



(c) Edge $\{v_0, v_1\}$ selected (again).



(d) Etc.

Figure 6: Sample interactions on P_6.

that would implement one move of the random process (i.e. selecting one edge and updating the labels on the two vertices).

While you are not required to do this, I would encourage you to do so to practice your object-oriented programming skills.

For example, you could implement an abstract "Graph" class which has an array (of length n) that stores the label of each vertex. A subclass could implement a "oneStep()" method that will perform one interaction on a randomly chosen edge in the graph (how this method works will depend upon the particular graph, hence that is why this method is implemented in each subclass). A method (called, say, "run()") in the "Graph" class could call the "oneStep()" method until all the labels are equal to 1, and then stop, returning the number of steps needed until all labels are equal to 1.

You would call this "run()" method many times and take the average of the values that are returned.

2. Your application program is going to run this process on both the cycle C_n and the path P_n. Call your application "CoopGame.java". (That's "co-op" as in "cooperative", not "coop".)

3. This program should be executed from the command line by giving the program either one or two parameters. The first parameter is the value of n, i.e. the number of vertices in the graph. This value of n should be an integer and at least 3.

   If there is a second parameter (this is optional!), that number is the number of times you will repeat the "run()" procedure on each type of graph. Like in Part I, you will take an average over a large number of repetitions to get a mean value for the number of steips until the process stops. This number should also be an integer and at least 1 and at most 5000.

   If the second parameter isn't given, you should assume a value of 2000 for the number of repetitions.

   You must check for these parameters as arguments to the program. There must be at least one parameter, and it is the value of n, which must be an integer and at least 3. If there is a second parameter, this must also be an integer and at least 1 and at most 5000. (If there is not a second parameter, take this value to equal 2000.) Other values (and double values and non-numeric input, etc.) should be rejected. Use Java exception handling methods to help with this.

4. For each graph type (path P_n and cycle C_n), you must run the random process for the given number of repetitions, either one specified by the user, or the default 2000 times. Furthermore, for each graph type, you must do this for two situations, (1) where each graph has a single -1 label to start with (at

one end of the path, if the graph is a path), and (2) when there are two -1 values in the graph (at either end if it's a path) and "opposite" each other in the cycle (one at v_0 and $v_{n/2}$ for the ring, where $n/2$ is rounded down if n is odd). All other labels in the graph should be initialized to 1.

5. For each graph type, and for each case of one/two initial -1 labels, give the (arithmetic) average number of time steps until the graph reaches the "all 1s" state, where the average is taken over the number of repetitions (either 2000, or the number specified by the user in the second parameter to the program).

   Clearly label your output, to show how many repetitions you are doing for each graph, what is the value of n, and whether there are one or two inital values of -1 in each case. As stated previously, in each case you want the arithmetic average (or mean) of the number of steps until the process stops. **DO NOT USE an external library to calculate this number!** Write a short method to do it yourself.

Sample Output

```
$ java CoopGame

     Usage: java CoopGame n [trials]

     n = number of vertices
     trials = number of trials (optional, defaults to 2000)

$ java CoopGame 100
Cycle of size 100 (2000 trials) with one starting -1 value: 381.377
Cycle of size 100 (2000 trials) with two starting -1 values: 590.746

Path of size 100 (2000 trials)i with one starting -1 value: 352.518
Path of size 100 (2000 trials)i with two starting -1 values: 510.8655

$ java CoopGame 250
Cycle of size 250 (2000 trials) with one starting -1 value: 932.9465
Cycle of size 250 (2000 trials) with two starting -1 values: 1466.311

Path of size 250 (2000 trials)i with one starting -1 value: 910.515
Path of size 250 (2000 trials)i with two starting -1 values: 1301.3205

$ java CoopGame 300 4250
Cycle of size 300 (4250 trials) with one starting -1 value: 1148.2235294117647
Cycle of size 300 (4250 trials) with two starting -1 values: 1797.2950588235294

Path of size 300 (4250 trials)i with one starting -1 value: 1051.341411764706
Path of size 300 (4250 trials)i with two starting -1 values: 1578.014588235294

$ java CoopGame -10

     Oops, check your parameter(s).  The first parameter specifies n,
   the number of vertices (at least 3).
     An optional second parameter specifies the number of trials to
   perform (between 1 and 5000).
```

A question to consider

How about if the graph is a star? The start, S_n, consists of a single "middle" vertex connected to n other vertices, so that it looks like a star. So the star S_n has $n + 1$ vertices, one at the center, and n vertices on the "spokes".

You can also represent a star for this random process just by using a Java array of the appropriate length. You can represent the interactions by choosing one of the outer vertices at random (choose a random integer

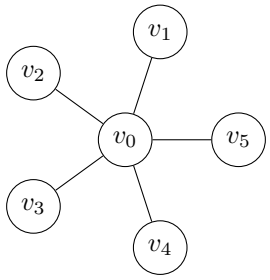


Figure 7: The star S_5.

between 1 and n inclusive), and having it interact with the center vertex v_0 to update their labels (as these are the only possible interactions).

How long does it take this process to converge? Note that if you go beyond about 20 or 25 vertices, your process will take a very long time to converge! It's very, very slow to finish on the star? Can you give any idea why it's so slow to converge on the star? Does it matter if, with one starting -1 if it starts on the center vertex or one of the others?

If you want, you can include calculations for the star in your main program you wrote for the path and the cycle. As I said, I would recommend not running the process on a star for much more than 20 to 25 vertices as it's very slow to converge!!

Submission Instructions

Your submission should consist of a report (a PDF file) and implementation (source code) files. Be certain to include all Java source files (i.e. the ".java" files) needed to run your application.

- Submit one compressed file, using only the "zip" format for compression, that includes all files (report and Java source code) for your submission.
- The report (a PDF file) should consist of

  Requirements: Summary of the above requirements statement in your own words. Do this for each part of the assessment.

  Analysis and Design: A short (one paragraph) description of your analysis of the problem including a Class Diagram outlining the class structure for your proposed solution, and pseudocode for methods. Again, do this for each part of the assessment. Pseudocode need not be a line-by-line description of what you do in each method, but an overview of how methods operate, what parameters they take as input, and what they return. Ideally, pseudocode should be detailed enough to allow me (or someone else) to implement a method in any programming language of their choosing, but not rely on language-specific constructs/instructions.

  For Part I and Part II you must submit your Java file(s). **Do not submit your compiled class files, only the source code!**

  Testing: A set of proposed test cases presented in tabular format including expected output, and evidence of the results of testing (the simplest way of doing this is to cut and paste the result of running your test cases into your report). You should describe carefully how you tested in your code in each part to determine that it is running correctly. (One way to test a random process is to actually could give a sequence of the "random values" that are used in the process, which are not actually random. For a small value of n (like $n = 3$) you could give a sequence of values that will result in the process stopping after a small number of steps. Alternatively, you can print out the random values that are generated, and the labels before and after the update step, and show that they are being updated correctly.)

- The implementation should consist of

  Your Java source files, i.e. the relevant .java files, not the class (.class) files.

Upload your files as a single compressed zip file.