COSC201 Assignment 1: Forming a pool

Due: 11:59 p.m. Friday, March 31, 2023

Introduction

Imagine a large warehouse floor with water dripping from random points overhead. As puddles form and grow larger, they merge one by one into pools until there is only one giant pool. Assuming the puddles grow at a uniform rate, then we know the order in which they will merge: the two closest centres first, then the next closest pair, and so on. In this assignment you'll investigate the following issue:

> *Up to the time when all the puddles have merged into a single pool one what proportion of the merges are essential in the sense that the two centres involved are not already part of the same larger puddle?*

However, probably the more significant part of this assignment is to investigate the efficiency of the various union-find implementations we've considered, and to learn just how large a union-find instance can be handled smoothly on the hardware you have available.

The experiment

Initialisation

A given number, n, of points are randomly distributed with x and y-coordinates between 0 and 1. The distance between each pair of points is computed and the pairs are sorted in increasing order of distance between them.

Computation

Using a union-find instance and the sorted list of pairs, a sequence of unions are performed representing two puddles merging. Each union is either *essential* if the two puddles were previously in two different pools, or *superfluous* if they already belonged to the same pool. At the point where the number of pools reduces to one, so that all further merges would be superfluous, we are interested in knowing what the ratio is between the number of superfluous merges and essential merges.

For example, if there were 1001 puddles to begin with, then there are 1000 essential merges that have to happen since each essential merge reduces the number of pools by one. If, at the time the 1000th essential merge takes place, there had been 4500 superfluous merges then for that run of the experiment, the ratio we're interested in would be 4.5.

Ideally, we'd like to see how this ratio behaves over many runs of the experiment for a given value of n and also how it changes when n changes. So, an experimental framework that allows you to collect that data conveniently is what you need to put together.

Raw materials

You will be provided with a number of Java classes. These include:

- All the union-find implementations we've looked at.
- A class `Point2D.java` that represents points in the plane.
- A class `Puddles.java` that can be used to represent scenarios of the type we're interested in.
  - Its constructor just takes an integer parameter describing the number of puddle centres and places them uniformly at random inside a square.
  - A convenience method returns a list of pairs of centres, in increasing order of distance between them.

Using these classes, you are asked to produce some more code and to conduct and report on some experiments.

Code submission (5 points)

The code you submit for this assignment will be a single class called `PoolAnalyser`. Details to follow.

Written submission (10 points)

Please submit the answers to the following questions as a single **PDF** document There are no formal requirements for the format of your submission, but presentation, spelling and grammar are all important elements which will account for roughly 40% of the marks for this part of the assignment.

The written answers to questions one and three could be a couple of paragraphs each (or a bit longer) along with supporting data. A single paragraph plus data is enough for question two, while a single paragraph plus possibly some pseudocode is enough for question four.

1. An individual run of the experiment can be divided up into three phases:
   - Generating the points,
   - Generating the merge-ordering list,
   - Carrying out the union operations until there is only one pool.

   In theory, how *should* these operations scale with the number of points? Conduct timing experiments to determine whether this actually seems to be the case. In the event that bottlenecks arise that prevent you from running these experiments over a wide range of point counts, report that rather than trying to over-interpret the data.
2. Only the third part of the run depends on the union-find implementation used. How does the choice of union-find implementation affect this run time? In particular, determine (on your hardware) what value of n produces a wall-clock time of (roughly) 1 second for this part for each of the different union-find implementations. In the event that bottlenecks arise (typically, memory bottlenecks in building the merge order), report the time taken for each type of union-find implementation on the largest (common) size that you can run instead.
3. The *result* of an individual run of the experiment does not depend on which union-find implementation is used. Using the most efficient one available (which, having done the previous question, you should now know) describe how the ratio between superfluous unions and essential unions behaves as the number of pools changes.

4. In creating the sorted list of pairs for input to union-find we could have chosen to first compute the distance between every pair of points and stored it, then simply accessed the precomputed distances for comparison purposes. How would you suggest doing this in Java?