HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Fall 2018

T-301-REIR, Reiknirit

S1: Percolation

Nökkvi Karlsson & Atli Gíslason
kt. 150896-3639 & 051299-2649
Group X-S1 47

September 9, 2018

TA: Sigurður Helgason

1 Introduction

The objective of this exercise is to calculate the average number of sites to be removed at random in an N by N grid for water to run through the grid so it percolates. We performed this experiment to further our knowledge of built-in union-find data structures and algorithms and to find the average number of sites that need to be opened so that an N by N grid percolates.

We implemented this first by using QuickFindUF and then WeightedQuickUnionUF. These are both union-find data structures. A union-find data structure is a data structure that tracks a set of elements partitioned into a number of non-overlapping subsets. The difference between QuickFindUF and WeightedQuickUnionUF is that WeightedQuickUnionUF takes in consideration the size of the subsets when it connects them while QuickFindUF doesn't. So most of the time WeightedQuickUnionUF is faster and in this exercise it's always faster, we calculate the difference in this report.

We ran multiple tests of the project and developed a consistently accurate model for statistics regarding the subject such as the average number of sites removed. In the first part, we constructed a Java class named Percolation whose function was simply to model the N by N grid and allow us to remove sites. This class was used to visualize the process of percolation and give us a model which was easy to test and manipulate. Secondly, we worked on the PercolationStats class which was constructed to easily repeat large numbers of tests and map data and comparisons between tests.

1.1 Setup and Methods

We started by reading about about union-find data structures and how they would help us implement the exercise. We started implementing the Percolation datatype, it had a boolean 2D array which modeled a grid of size N by N and a QuickFindUF data structure with the size N * N + 2 that we initialize as private variables.

Our QuickFindUF data structure uses a 1D array and all our functions take in 2D coordinates so we must convert the coordinates to 1D. That's why we created the private helper function xyTo1d() which takes in 2D coordinates and returns the same coordinates in 1D. Most of the functionality in our Percolation datatype is in the open() function which takes in two parameters that act as coordinates in our grid and open the site at that index and convert the coordinates to 1D so that we can connect the corresponding site with other open sites around it in the QuickFindUF data structure. But if the site is in the first row we connect it to a virtual top site that has the index N * N in our QuickFindUF and if it was in the bottom row we connected it to our virtual bottom site with the index N * N + 1.

This made it easy to see if the site was full, the function isFull() took 2D coordinates converted them to 1D and checked if the site was connected to the virtual top. If it was, that meant that water would reach the site and fill it. We also used this method to check if our grid percolates, all we had to do was to check if our virtual bottom site was connected to our virtual top site.

But now we noticed that when we tested the data type and a grid percolated, the sites that had no open path to the top but were in the bottom row also return true when checked. This backwash occurs because when we open a site in the bottom row we connect it to the virtual bottom site so every site that is connected to the virtual bottom site is also connected to the top site when the grid percolates.

We fixed this by having two union-find data structures that mirrored each other except one did not have a virtual bottom site so it is initialized with the size N * N + 1. Every time that we open a site and connected it to another open site nearby we do it in both data structures except we don't connect a site in the bottom row to a virtual bottom site in one of the data structures because it does not have any. So now when we check if our grid percolates we check if the virtual bottom site in our first union-find data structure is connected to it's virtual top site, but use our other union-find data structure if we are checking if a site is full.

Next we implemented our PercolationStats class which we used to test our Percolation datatype. The PercolationStats constructor takes in two integers as parameters and T and N and performs T independent experiments of a Percolation(N). It generates two random numbers using StdRandom from the algs4 library and they act as coordinates in our Percolation grid when we call the open function. The class calculates the mean, the standard deviation and the confidence high and low of the percolation threshold as well the the average time that a grid takes to percolate. Later we changed our union-find data structures in Percolation from QuickFindUF to WeightedQuickUnionUF to test the average time difference between those variations when running our percolation tests with PercolationStats.

We worked on this exercise on a MacBook Pro computer with a 2.6 GHz intel Core i7 processor and a 16GB 2133 MHz LPDDR3 memory running Oracle Java 1.8 on macOS High Sierra operating system.

1.2 Implementation

We started by implementing the Percolation datatype. It has 8 private member variables:

1. A boolean grid[][] which we use to keep track of which sites are open.
2. A private int openSites which we use to keep track of how many sites are open.
3. A private int numberOfRows which we use to know how many rows the grid has.
4. A private int numberOfCol which we use to know how many columns our grid has.
5. A QuickFindUF uf that is our union-find data structure that we use to connect open sites, we later changed it to WeightedQuickUnionUF to calculate the time difference between QuickFindUF and WeightedQuickUnionUF.
6. A QuickFindUF uf_noBackwash is another union-find data structure which is identical to the QuickFindUF uf but it has no virtualBottom site.
7. A private int virtualTop which has the value of the index of the virtual top site.
8. A private int virtualBottom which has the value of the index of the virtual bottom site.

The constructor for our Percolation datatype takes in an integer N as a parameter and initializes grid with N rows and N columns, the uf with N * N + 2 because the virtual top and bottom take the last 2 indexes. We initialize a second union-find data structure uf_noBackwash with the size N * N + 1 because we only need to make room for the virtual top. We initialize numberOfRows and numberOfCol as N and give virtualTop the value N * N and virtualBottom N * N + 1.

Then we implemented the private helper function xyTo1d(int row, int col) which takes 2D coordinates as parameters and returns them as 1D coordinates, that uses the formula (numberOfCols * row) + col.

Next we implemented isOpen(int row, int col) which only returns the value of the grid at location [row][col].

open(int row, int col) has the most code in our Percolation datatype, we start by checking if the site at these coordinates is already open and if it is, we return from the function. If it's not open we open the site by changing the value of the index in our grid from false to true and add 1 to openSites. Next we change the 2D coordinates which were passed as parameters to 1D using our xyTo1d function so we can use the coordinates in uf and uf_noBackwash.

Now we have an open site and there are a few cases that we need to check. If there are other sites open next to our newly opened site we need to connect them together in both uf and uf_noBackwash, if our newly opened site is in the top row of our grid we connect it to the virtual top index in both uf and uf_noBackwash or if it's in the bottom row we connect it to our virtual bottom index only in uf because uf_noBackwash has no virtual bottom.

isFull(int row, int col) is supposed to check if a site is connected to a path of open sites that connect to the top. We do this by checking if the site is connected to the virtual top site in uf_noBackwash by calling uf_noBackwash(the coordinates of the site, virtualBottom).

numberOfOpenSites() this function returns the value of openSites.

percolates() this function checks if there is a path of open sites from the top row to the bottom row. We do this by checking if there is a path by checking if the virtual top site is connected to the virtual bottom site by calling uf.connected(virtualTop, virtualBottom).

After finishing the Percolation datatype, we begun implementing the PercolationStats class. PercolationStats was a much more different task than the Percolation datatype as we included more libraries and had generally simpler functions.

PercolationStats has 7 public member variables.

1. A public double mean which we use to store the mean of the percolation threshold in the tests run by PercolationStats.

2. A public double stddev which we use to store the standard deviation of the percolation threshold in the tests run by PercolationStats.

3. A public double confidenceLow which we use to store the lowest value of the 95% confidence interval. This value has a z-score of -1.96.

4. A public double confidenceHigh which we use to store the highest value of the 95% confidence interval. This value has a z-score of +1.96.
5. A public double meanOfTime which we use to store the mean of the average time it took to percolate a grid of size N by N in our tests run by PercolationStats.
6. A public double percyThreshold[] stores the percolation threshold of each individual test in an array, which we can later manipulate and analyze.
7. A public double timeElapsed[] stores the time it took for a grid to percolate in each individual test in an array, which we can later manipulate and analyze.

The constructor of the PercolationStats class takes in two integer parameters N and T and performs T independent experiments of a Percolation(N) datatype that we call percy. We do this by using a for loop that loops T times and in each loop we create a variable of the type Stopwatch and a Percolation(N) datatype. Stopwatch is a datatype from the algs4 library which we use to measure time. In the for loop is a while loop that loops until the Percolation instance has been percolated (while(!percy.percolates())) and there we generate two random integers between 0 and N-1 and call the open function for percy using these random integers as coordinates. When the while loop stops we take the time it took to percolate the grid and store it in our timeElapsed array at index i and the number of site that were open divided by the size of the grid and store it in our percyThreshold[] array at index i.

mean() imports StdStats from the algs4 library and uses the library's mean function which takes in an array and returns the mean of that array.

stddev() also imports StdStats from the algs4 library and uses the library's standard deviation function that takes in an array as a parameter and returns the standard deviation of that array.

confidenceHigh() and confidenceLow() had to be implemented manually since StdStats does not have a function that calculates confidence intervals. We are searching for the 95% confidence interval which means we needed values with a z-score of +1.96 and -1.96. This is found using formulas used in statistics: mean + (1.96*stddev)/$\sqrt{n}$ and mean - (1.96*stddev)/$\sqrt{n}$ where n is the length of the array percyThreshold[]. In order to perform square root operator, we used the sqrt() function in the Math library.

2 Results

The objective of this project was to calculate the percolation constant. In addition, we wanted to see if this constant would change depending on the size of the grid, the number of trials, and the type of data structure. We ran numerous tests and trials, and filled in the graphs below in section 2.1 and 2.2.

In the trials where we used Quick-Find as our data structure and used the biggest possible grid (350x350), we found that the percolation constant was found to be between 0.5913 and 0.5936.

In the trials where we used Quick-Find as our data structure and repeated the trials 350 times, the constant was between 0.5922 and 0.5957. This was a much larger gap than in the other, size-based trial.

We ran additional trials with the Weighted Quick-Union data structure with size-focused and repeat-focused trials. Those gave us the intervals 0.5905-0.5932 and 0.5911-0.5933 respectively.

Percolation constant

Although we ran many different types of experiments with different data structures and sizes, all of them do share similarities in the constant. All intervals fell in between the gap 0.5905-0.5957. The mean of all tests concludes that the constant is 0.5931.

2.1 Quick-Find

Results

In the tables below are the running times and confidence intervals of different values of N with a fixed T, and different values of T with a fixed N. In the first table (fixed T), and N a multiple of 50. When N is 350, the confidence interval becomes 0.5913-0.5936. In addition, when N gets larger, the interval shrinks. When N is 100, the difference between the highest and lowest interval is 0.006468, but when N is 350, the difference shrinks to 0.002292 (3 times less).

With regards to the running time, we see a sharp increase in time compared to the rate of growth for N. When N is 100, the running time is 0.03677, but when N is doubled to 200, the running time becomes 1.28964, 35 times larger. When we graph N vs. Avg Running time, we see that the graph grows exponentially, which alludes that the Time Complexity will have O(n) of some type of exponential function.

In the second table, N is at a constant and T is changed. Average Running time does not change when T is increased, and therefore we can assume from here that T will have a linear relationship with Time Complexity.

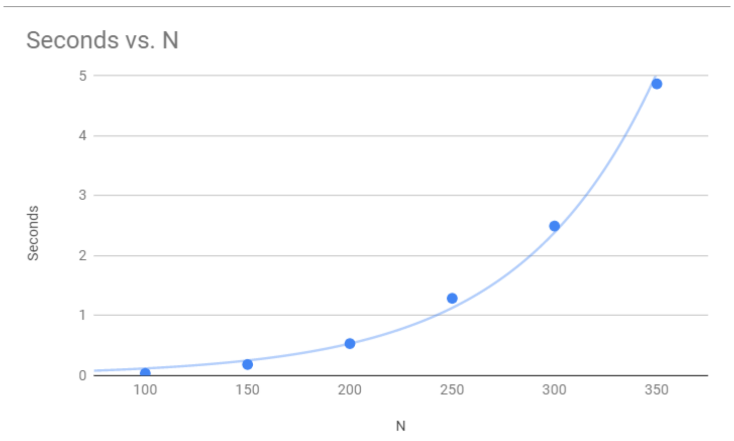

Figure 1: Size of grid N vs. Avg Running Time

N	Avg Running Time	Confid.(low)	Confid.(high)
100	0.036770	0.588250	0.594718
150	0.185523	0.592031	0.596591
200	0.533900	0.589070	0.592793
250	1.289640	0.591011	0.594268
300	2.495380	0.590303	0.592999
350	4.867550	0.591278	0.593570

(a) T is fixed at 100

T	Avg Running Time	Confid.(low)	Confid.(high)
100	0.034121	0.587714	0.594495
150	0.033567	0.589701	0.594383
200	0.033265	0.591115	0.595447
250	0.033092	0.589016	0.592834
300	0.033947	0.590392	0.594262
350	0.034726	0.592239	0.595721

(b) N is fixed at 100

Table 1: QuickFind

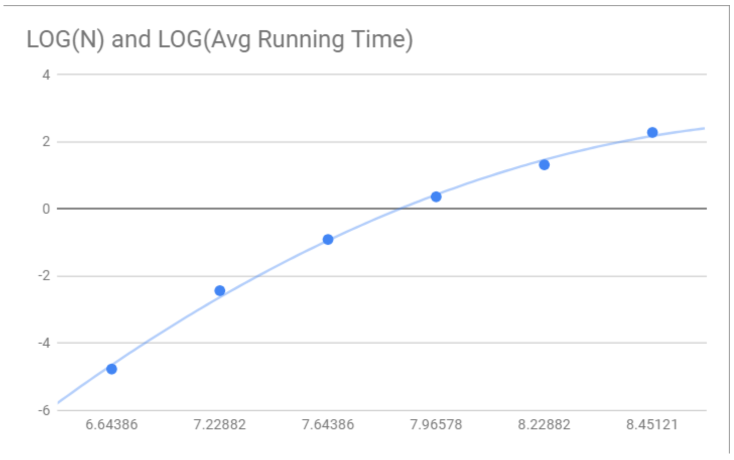

Figure 2: Size of grid N vs. Avg Running Time

Time complexity Running time as a function of N and T: $6.4 \cdot 10^{-10} \cdot N^{3.9} \cdot T$. In tilde notation, that is $\sim N^{3.9} \cdot T$

Reasoning: In the log-log plot, the time complexity can easily be predicted using a polynomial best-fit line. We found that the formula $y = -2.1316 \cdot 10^{-3} \cdot x^2 + 3.8962x - 30.538$. Since the graph is a log-log plot, y is the log of Avg. Running Time, and x is the log of N. Therefore we can rewrite the equation as $log(ART) = -2.1316 \cdot 10^{-3} \cdot log(N)^2 + 3.8962log(N) - 30.538$. We want to solve for ART, so we perform the following steps:

- $ART = 2^{-2.1316 \cdot 10^{-3} \cdot log(N)^2 + 3.8962 log(N) - 30.538}$
- $ART = 2^{-2.1316 \cdot 10^{-3} \cdot 2 \cdot log(N) + 3.8962 log(N) - 30.538}$

- $ART = 2^{-4.2632 \cdot 10^{-3} \cdot log(N) + 3.8962 log(N)} \cdot 2^{-30.538}$
- $ART = 2^{3.892 log(N)} \cdot 2^{-30.538}$
- $ART = 2^{-30.538} \cdot N^{3.892}$
- $ART = 6.4 \cdot 10^{-10} \cdot N^{3.892}$

We have found a formula to calculate Time for one experiment, but we repeat each experiment T times, so we multiple this with T to find the total time for T experiments. Our final answer is $6.4 \cdot 10^{-10} \cdot N^{3.892} \cdot T$

2.2 Weighted Quick-Union

Results

In the tables below are the running times and confidence intervals, in the same format as the previous tables. Here the confidence intervals shrink as well when N and T get larger.

In regard to the running time, the average running time increased at a much smaller rate than the QuickFind algorithm. When N doubled from 100 to 200, the average running time only increased from 0.00126 to 0.00309, which is less than threefold (less than 35 like QuickFind).

In the second table, N is at a constant and T is changed. Average Running time does not change when T is increased, and therefore we can assume from here that T will have a linear relationship with Time Complexity.

N	Avg Running Time	Confid.(low)	Confid.(high)
100	0.001260	0.590248	0.596640
150	0.002210	0.590211	0.594456
200	0.003090	0.590828	0.594539
250	0.004620	0.591484	0.594144
300	0.006490	0.591214	0.594124
350	0.009040	0.590525	0.593167

(a) T is fixed at 100

T	Avg Running Time	Confid.(low)	Confid.(high)
100	0.003990	0.591965	0.595961
150	0.002780	0.591729	0.594940
200	0.002720	0.589997	0.592349
250	0.002716	0.590242	0.592643
300	0.002687	0.592077	0.594285
350	0.002694	0.591140	0.593306

(b) N is fixed at 200

Table 2: WeightedQuickUnion

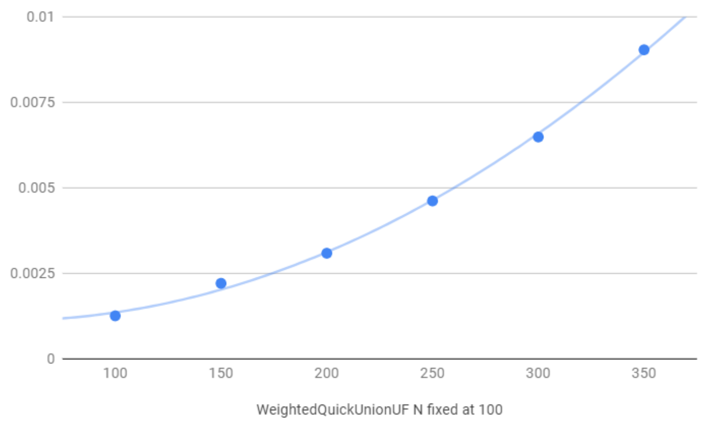

Figure 3: Size of grid N vs. Avg Running Time

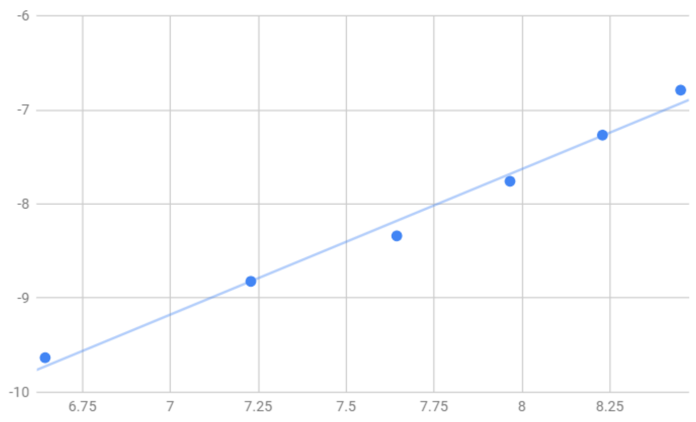

Figure 4: Size of grid N vs. Avg Running Time

Time complexity Running time as a function of N and T: $7.4 \cdot 10^{-10} \cdot N^{2.1} \cdot T$. In tilde notation, that is $\sim N^{2.1} \cdot T$

Reasoning: In the log-log plot, the time complexity can easily be predicted using a linear best-fit line. We found that the formula $y = 2.14x - 24.872$. Since the graph is a log-log plot, y is the log of Avg. Running Time, and x is the log of N. Because of this, we can use the following rules to find the Time Complexity $a \cdot n^b$

· b is the slope

· log(a) is the y-intercept

The slope of the graph is 2.14 and the y-intercept is -24.872. b is therefore 2.14, and a is $2^{-24.872}$. The time complexity is thus $7.4 \cdot 10^{-10} \cdot N^{2.1}$. Since we do each trial T times, we multiply by T to get the total time $7.4 \cdot 10^{-10} \cdot N^{2.1} \cdot T$.

2.3 Memory Usage

Our Percolation datatype contains:

1. A 2D boolean grid which takes mn bytes.
2. Two WeightedQuickFindUF objects and references to them each taking 8 + (8n + 88) bytes of memory.
3. 5 integer variables which take 4 bytes each adding up to 20 bytes together.
4. Plus 16 extra bytes for the object overhead and 4 for padding.

So our program Percolation uses ∼ mn + 16n + 232 bytes of memory, or ∼ mn. But because the m and n will always be the same number in our Percolation datatype the memory usage of it can also be written like this $\sim n^2$.

2.4 Discussion/Conclusions

The outcomes matched what we expected. When we ran the numbers for QuickFind, it took a lot longer than when we used WeightedQuickUnion. The time complexity for the two is very different: n^4 and n^2. When solving for the time complexity we also found an easier method to calculate the time complexity using slope and y-intercept. The experiment went well and we believe it is successful.

3 About This Solution

Have you taken (part of) this course before: No we have not taken part of this course before.

3.1 Quiz on Collaboration

1. How much help can you give a fellow student taking REIR?
   (a) None. Only the TAs can help.
   (b) You can discuss ideas and concepts but students can get help debugging their code only from a TA, or student who has already passed REIR.
   (c) You can help a student by discussing ideas, selecting data structures, and debugging their code.
   (d) You can help a student by emailing him/her your code.

   Answer: b

2. What is the expectation when partnering?
   (a) You and your partner split the assignment between you and solve the parts separately.
   (b) You and your partner discuss all the problems together, but code separately.
   (c) You and your partner discuss the problems and write all the code together.

Answer: c

3.2 Known Bugs / Limitations.

We did not learn of any bugs while implementing this exercise and we did not notice any limitations.

3.3 Help Received

We discussed the basic concept of the virtual top and bottom sites with Sigurður Kalman Oddsson our classmate before starting to implement Percolation.That furthered our understanding of union-find data structures and helped us in the implementation of this exercise.

3.4 Problem Encountered

Our virtual top site was defined first at another index than the percolationVisualizer expected and that led to the percolationVisualizer to draw an incorrect image of our grid. We first had our virtual top site at the beginning of our uf but then we changed it to the next to last index (N * N).

3.5 Comments

We really enjoyed this project and found the subject matter very interesting. Our understanding of union-find data structures has greatly improved.