Analyzing Percolation Algorithm Efficiencies with Various UnionFind Methods

Percolation Problem Set

Elven Isaac Shum

October 10 2019

1 Introduction

"Given a porous landscape with water on the surface, under what conditions will the water be able to drain through to the bottom?"[1] To model this scenario, an abstract process called percolation" was developed.

In this percolation model, a grid is created to simulate the landscape. Via "opened" sites, or creating holes, in the grid, we can determine whether a path exists from the top to the bottom. To do so requires a method to determine whether sites are "connected." In addition to modeling the scenario, we must discover, what types of algorithms are the fastest? Can we develop efficient ones? Several algorithms were presented at a very high level.

In this assignment, I develop the percolation model and test 4 different algorithms to determine their speed and computation efficiency.

[1] From Percolation Assignment introduction on Canvas

2 Brief Algorithm Explanations

Note that for all these algorithms are designed for a 1 dimensional array. As is specified by the problem statement, our grid will be an 1 dimensional array instead of a 2 dimensional array. This is likely for understanding and translatability with our prescribed algorithms. There are two fundamental operations with the algorithms: Finding and Unioning.

Unioning can be thought of as "connecting" different sites of the array. While Finding is simply checking whether two cites are connected.

2.1 QuickFind

With QuickFind, to be "connected" means for given indexes to hold the same value. Stated another way: for index p and q to be connected, $arr[p] = arr[q]$ As described in the algorithm's name, this quick at Finding. In turn, its significantly slower at Unioning. The finding is quick since only one comparison needs to be made $arr[p] == arr[q]$. But for Unioning, in the event connections have already been made, the algorithm must search the entirety of the array to determine which spots must be altered. See Figure 1 for a visual representation. [2]

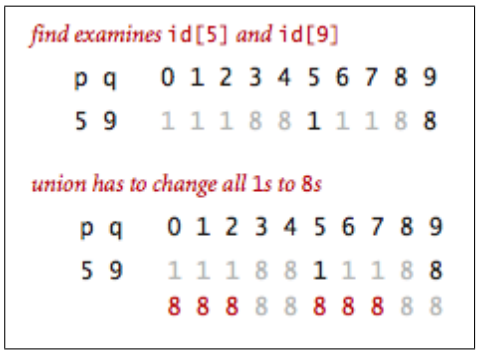

Figure 1: Intuitive Diagram of QuickFind Algorithm [2]

[2]Image courtesy of *https* : *//algs4.cs.princeton.edu/15uf/*

2.2 QuickUnion

The "dual" of QuickFind, QuickUnion's speeds are reversed: Fast at Unioning, slower at Finding. The algorithm defines site-unions as links, effectively creating a tree-like structure. When unioning p and q, the value at p's index gets changed into q; in this way, the value within p links to q. Successive connections result in long strings. One could image that, when these long strings are attached, they create trees. To find some p and q, the algorithm works its way up both p and q's trees until it reaches their "root". If their roots's values are equal, then p and q are connected. See Figure 2 for a visual representation.

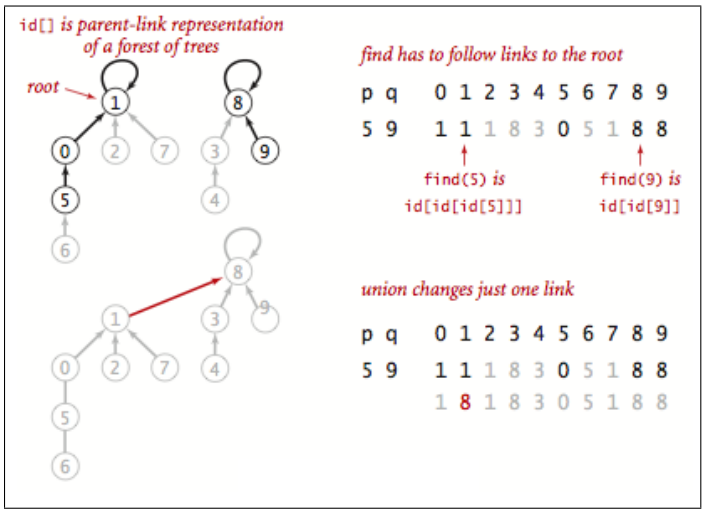

Figure 2: Intuitive Diagram of QuickFind Algorithm [2]

2.3 WeightedQuickUnion

An improvement on QuickUnion. While the definition of a connection stays the same, the difference is that trees are no longer arbitrarily connected. Rather, the algorithm determines which tree is smaller and the smaller tree to the larger, thus increasing the efficiency. See Figure 3 for a visual representation.

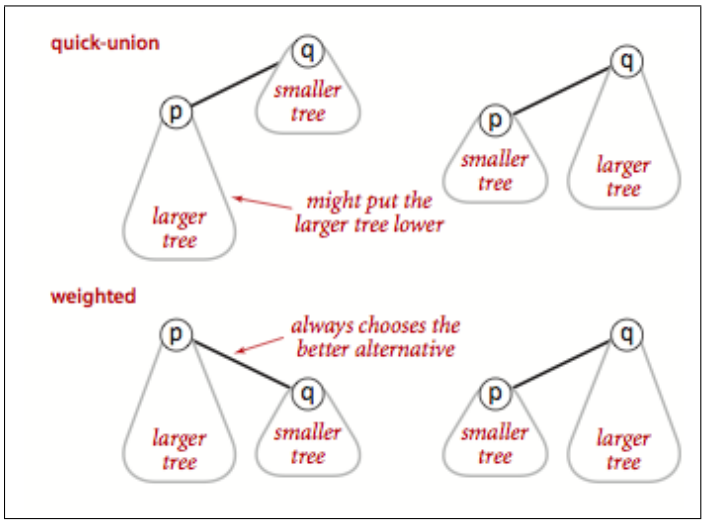

Figure 3: Intuitive Diagram of WeightedQuickFind Algorithm [2]

2.4 WeightedQuickUnion with Path Compression

A further improvement to QuickUnion and WeightedQuickUnion. Now, the goal is to have each site directly connected to it's root–no more long trees to search through. To achieve this, the algorithm immediately connects all nodes it examines to the root. The diagram is left as an exercise to the reader.

3 Data and Results

After running several Monte Carlos simulations for grid lengths intervals of 100, the following table 1 of results was produced. Each Algorithm was tested and is noted in milliseconds. These values were plotted in Figure 4.

Table 1: Gathered Data of Percolation Runner

Grid Length	QFind(ms)	QUnion(ms)	Weighted(ms)	PathComp(ms)
100	29	4	2	3
200	362	13	8	9
300	2453	28	22	10
400	8338	50	37	18
500	20117	135	35	26
600	41840	135	73	39
700	71342	196	86	50
800	121536	274	95	67
900	216334	421	127	77
1200	291951	949	262	185

The table's algorithm order was determined by the hypothesized speeds in decreasing order. As the data demonstrates, the hypothesies were correct.

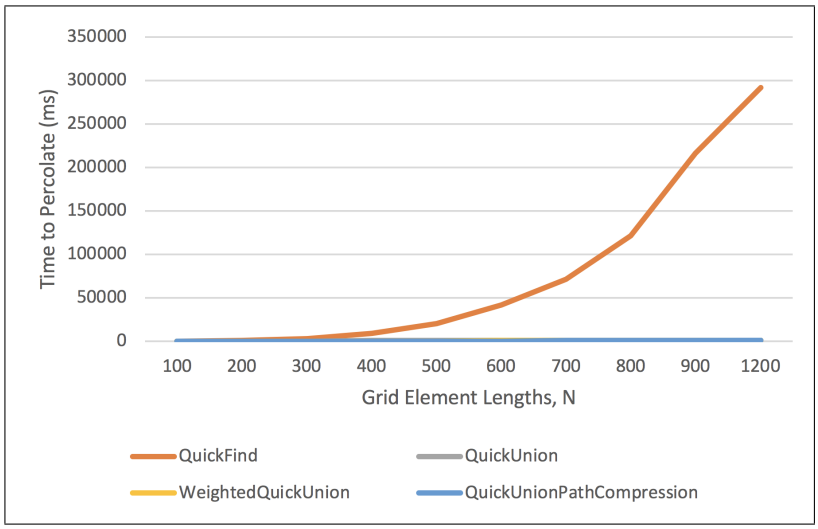

Figure 4: Plot of Grid Element Length vs Percolation Time in milliseconds

As is clear from Figure 4 and Table 1, QuickFind is incredibly slow relative to the other algorithms. To get a better representation of the other speeds, Figure 5 has removed QuickFind.

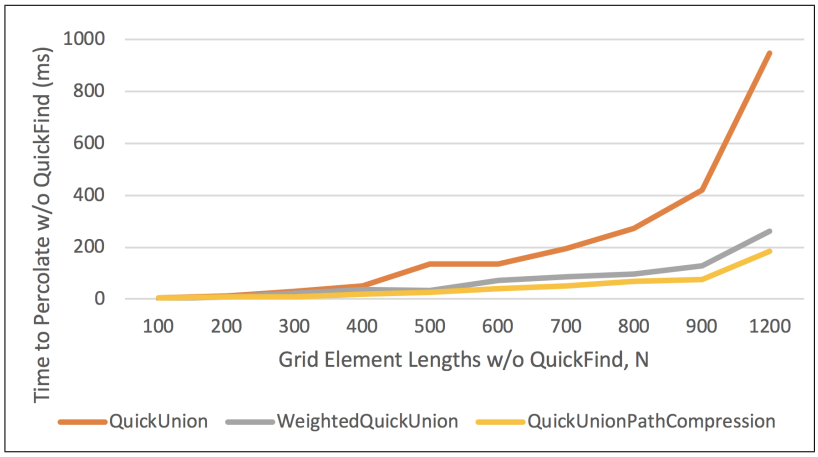

Figure 5: Plot of Grid Element Length vs Percolation Time W/O QuickFind

To see the type of N-speed growth of each algorithm, a log-log plot was generated. It culminates in Figure 6.

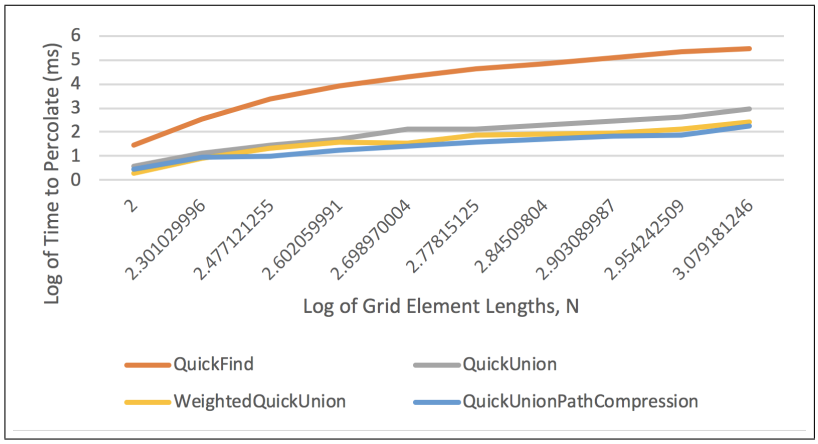

Figure 6: Log-Log Plot of Grid Length vs Percolation Times

4 Appendix

In the below figure 7 are the worst case N-speeds for each algorithm from Princeton's Algs4. Each of the speeds accurately corroborates the data found from the Percolation simulation.

Figure 7: Algorithm Worst Case N-Speed [2]

algorithm	order of growth for N sites (worst case)		
	constructor	union	find
quick-find	N	N	1
quick-union	N	*tree height*	*tree height*
weighted quick-union	N	$\lg N$	$\lg N$
weighted quick-union with path compresson	N	*very, very nearly, but not quite* 1 (*amortized*) (see EXERCISE 1.5.13)	
impossible	N	1	1