COSC201 Assignment exemplar
Lakes, Islands and Union Find

Brendan McCane

April 2026

1 Introduction

This report documents experiments and the performance of union find algorithms [3] for the purpose of performing connected components on a grid-based height map. The context is that we are given a grid-based height map of a geographical area and are to find the lakes and islands in the area for a given water-level height. To find these components of the map we are to evaluate the performance of three different union find algorithms. Union find is a data structure and algorithm for keeping track of disjoint sets and hence is ideal for this problem. Each disjoint set is represented by a single representative. Union find has three basic operations:

make(n) creates a union find instance with n elements,

find(a) returns the representative for an element of the set,

union(a,b) joins the two subsets of which a and b are members,

We are given three implementations of union find to evaluate, here named as per the lecture notes [1, 2]: UF1, UF2 and UF3. UF1 is a naive implementation that has theoretical complexity of $\Theta(1)$ for *find* and $\Theta(n)$ for *union*. UF1 stores the representative for each set directly in a *reps* array. UF2 alters UF1 by allowing the representative of an element to be reached through a local representative chain - the *reps* array is now implicitly organised into a tree. This hopefully improves *union* because it no longer needs to scan the entire *reps* array, but *find* becomes a bit more costly. Unfortunately, the chains can become as long as the number of elements and therefore the worst case performance for both *find* and *union* is $O(n)$. Finally, UF3 improves on UF2 by keeping track of the length of the chains in a *rank* array. In UF3, *union* is forced to keep the longer chain representative as the representative and thus limits the length of the chain to $O(\log(n))$. Both *union* and *find* are $O(\log(n))$ in UF3.

I hypothesise that UF3 will outperform UF2, and UF2 will outperform UF1 in all practical experiments. I also hope to confirm that the performance of each algorithm will match its theoretical efficiency. However, I think UF2 will perform better than $O(n)$ because on average it should build chains that are much shorter than the maximum - and indeed, I think UF2 will be close to $O(\log(n))$ performance.

2 Method

To find the connected components (the lakes and islands), we must scan the grid and check neighbours using a 4-neighbour topology. If two neighbours are both below or above the water-height, then the sets they belong to should be unioned. It is important to note that the number of cells in a grid is $n = w \times h$ where w is the width and h the height of the grid. Since we are scanning the entire grid, we potentially perform n unions, and therefore the complexities for *union* will be: $\Theta(n^2)$, $O(n^2)$, and $O(n \log(n))$ for UF1, UF2, UF3 respectively.

I report on two experiments.

2.1 Experiment 1: grid sizes

In experiment 1, I test the performance of the algorithms as the grid size changes. For this experiment I fix the height of the grid to $h = 100$ and vary the width from 100 to 1000 in steps of 100 (n ranges from 10000 to 100000) so that data points are equally spaced in n. The height maps are generated randomly with heights varying from 0 to 1 and smoothing using a Gaussian-like smoothing function of size 19. Each trial is repeated 10 times to account for variations in the state of the machine although I don't explicitly account for JVM warmup or JIT effects. The

Java system utility *nanoTime* is used to time the use of the union find algorithms which includes scanning the grid, applying union find when needed, and rescanning the grid to create the lakes and islands sets.

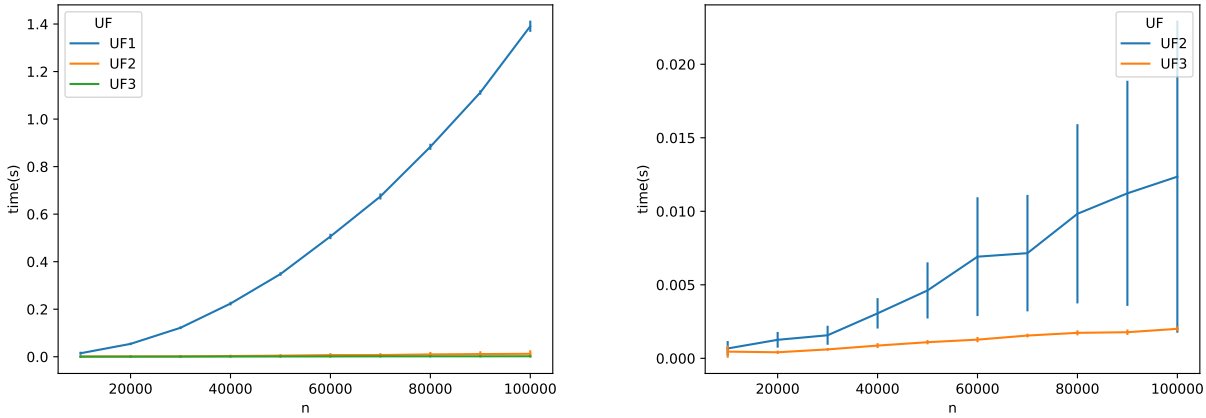
2.2 Experiment 2: water heights

In experiment 2, I test the performance of the algorithms on a single height map (dunedin_large thinned by a factor of 3), for 11 different sea height levels spaced evenly between the minimum and maximum heights in the map. The map was thinned by a factor of 3 to allow UF1 to finish in reasonable time. As in experiment 1, each trial was repeated 10 times to measure variation across runs. For experiment 2, I expect some dependence on sea height for all algorithms, as the number of unions will depend on the number of islands and lakes in the data. For this reason, I also measure the number of islands and lakes in each experiment.

3 Results

3.1 Experiment 1

Figures 1a and 1b shows the timing results for experiment 1 split into two figures. On the left are all algorithms for which UF1 dominates. To better visualise the performance of UF2 and UF3, these are shown on the right.



(a) Timing of all algorithms.

(b) Timing of UF2 and UF3 only.

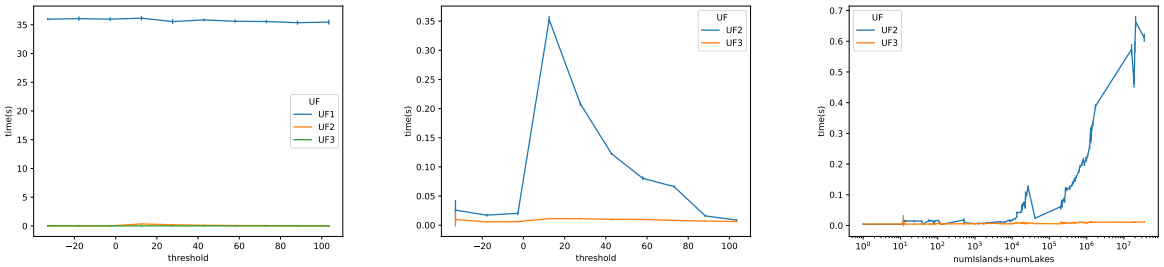
Figure 1: Scaling of algorithms with grid size

Figures 2a and 2b show the timing results on the dunedin_large thinned dataset varying with sea level. Figure 2c shows how UF2 and UF3 vary with the number of lakes and islands this time with 100 thresholds and with a log scale on the x-axis to better show the relationship. This third graph clearly highlights the reason for the jump in cost for UF2 for middle thresholds.

4 Discussion and Conclusion

It is clear from experiment 1 that UF3 is the fastest algorithm, followed by UF2, with UF1 a distant third. UF1 timing is consistent with $\Theta(n^2)$ - the graph looks quadratic and the variation between runs is very low. For UF2 and UF3, it is more difficult to tell if they are scaling as $O(n)$ or $O(n\log(n))$. The graphs are consistent with both linear and $n\log(n)$ scaling but given the theoretical analysis, it is reasonable to assume $n\log(n)$. Perhaps not surprisingly, UF2 has a large variance which is likely because the length of chains generated is dependent on the data.

For experiment 2, the data is very interesting. Again, UF1 is the clear loser and is much less efficient than UF2 and UF3. Somewhat surprisingly UF1 had little dependence on the sea level compared to the overall cost of UF1. But, on further thought this makes sense - the efficiency of UF1 depends on the number of items in the set and is completely independent of the number of disjoint sets. For UF2, there is a clear dependence on the sea level, but the relationship is not simple as shown in Figure 2b. Figure 2c shows how UF2 varies with the number of islands



(a) Timing with sea level of all algorithms.

(b) Timing with sea level of UF2 and UF3 only.

(c) Timing with number of lakes and islands of UF2 and UF3.

Figure 2: Timing for experiment 2

and lakes. Note the log-scale on the x-axis for better visualisation. This shape of curve is consistent with $x \log(x)$ and x^2 curves (for $x =$ the number of disjoint sets), and according to the theoretical analysis, UF2 lies somewhere between these two. Although not evident from Figure 2c, UF3 is almost certainly growing at $x \log(x)$, albeit at a slower rate that UF2.

Given the performance of the three implementations, there are few reasons to choose anything other than UF3 for any application. The only case would be if *find* was used much more frequently than *union*, in which case UF1 would be the preferred choice due to its $\Theta(1)$ performance on *find*. However, there is one further optimisation that could be applied to UF2 or UF3 that would improve the performance of *find* in those implementations — path compression [3]. Path compression updates the representatives of all cells during a call to *find* and maximally flattens the tree structure in the process. As a result, the complexity of both *find* and *union* becomes very close to constant. We leave the implementation of path compression for another time.

References

[1] Brendan McCane and Michael Albert. *Basic Union Find*. COSC201 Algorithms and Data Structures lecture notes, University of Otago. Accessed: 2026-04-26. 2026. URL: `https://cosc201.cspages.otago.ac.nz/lectures/L02_BasicUnionFind.pdf`.

[2] Brendan McCane and Michael Albert. *Improving Union Find*. COSC201 Algorithms and Data Structures lecture notes, University of Otago. Accessed: 2026-04-26. 2026. URL: `https://cosc201.cspages.otago.ac.nz/lectures/L03_ImprovingUnionFind.pdf`.

[3] Wikipedia contributors. *Disjoint-set data structure*. Accessed: 2026-04-26. 2026. URL: `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`.