Study of Four Union-Find Algorithms

By Xuheng Duan, Bojun Li, Yingjian Wu, and Shuyue Zhou

Abstract

Our goal is to implement different Union-Find Algorithms as well as testing running times in synthesized settings. Specifically, we learned and implemented Quick-Union, Quick-Find, Weighted Quick-Union, and Path-Compression versions. For each different algorithm, we used three languages: Java, Python, and C++. We tested the running time of them respectively. We constructed 500 nodes and performed 10,000 Union operations on them. We used three different languages and four versions of Union-Find Algorithms to achieve the data respectively. Then we changed the number of nodes to 5,000 and Union operations to 100,000, testing the total time for each trial again.

Introduction

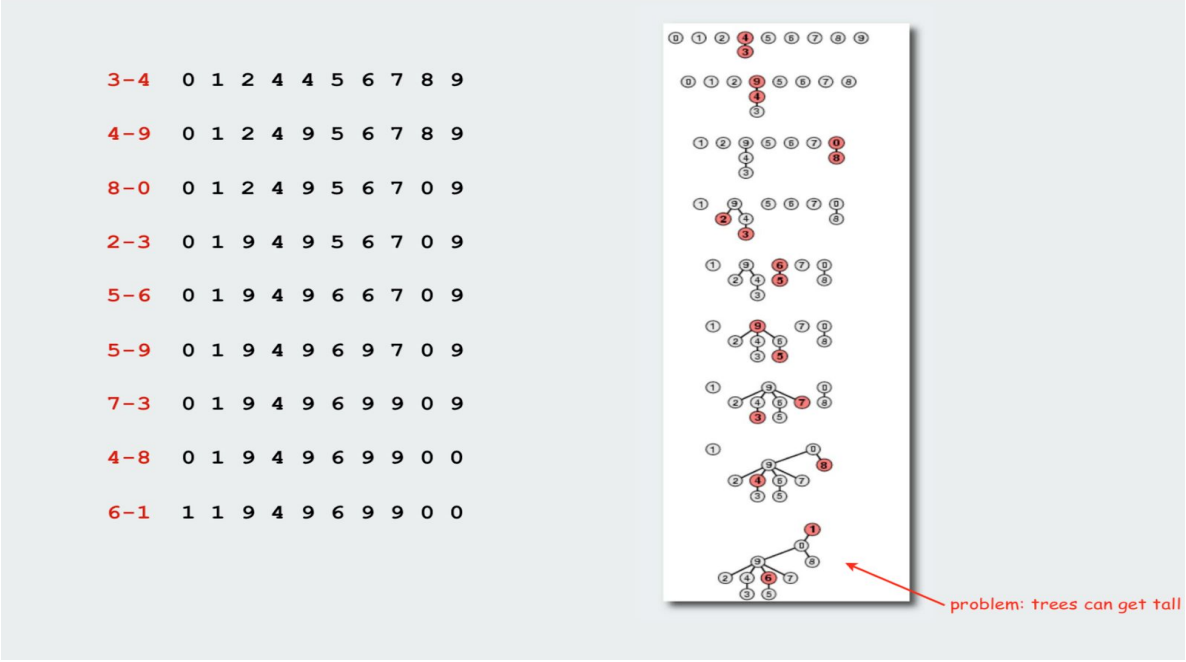
Union-Find Algorithm is widely used for maintaining a disjoint-set data structure, which is a data structure tracks a set of elements partitioned into some non-overlapping subsets. With the two basic operations, Union and Find, the disjoint data structure provides near-constant time to add new sets, merge existing sets and check whether elements are in the same set. Researchers have well studied and Union-Find Algorithms and have proposed some different improvements in different directions such as faster speed and more method. For instance, the algorithm developed by Rem in 1976 is generally considered the fastest (Patwary), and Ben-Amram attempted to add more features, like deletion, to classic Union-Find Algorithm. Thus we think Union-Find is a vital algorithm closely relates to our class and worth to study.

Summary of implementation

We implemented four Union-Find Algorithms in three languages: Python, Java, and C++. Among the four algorithms, two of them are basic versions, and the others are improvements. Namely, Quick-Union, Quick-Find, Weighted Quick-Union and Path-Compression, with the latter two being improved versions. We used an array to represent the disjoint data structure, with the index of the array representing the element, and the value of the array representing the set information. For each Union-Find Algorithm, we implemented Union operation and Find operation, respectively.

The Quick-Find algorithm marks every object in the set to the root node. When executing the Find operation, the algorithm checks for values of the two indexes (each representing an object). When performing the Union command, the project would use Find operation to check if they are in the same set already, and if not, the algorithm changes the value under the second index to the same value as under the first index.

The Quick-Union algorithm is similar to the Quick-Find algorithm, but instead of mapping to the root node, this time, the algorithm marks each new element of the set to its parent node. When performing Union operation in Quick-Union, after checking the elements are not in the same group, the algorithm changes the element's value to its corresponding parent. When performing Find operation, the algorithm will trace back to the root node via the series of parent nodes of two elements. For instance, in the first row, elements under index three and four both have value four, which means element four's parent is element four (which is itself), and element three has parent element four.



However, Quick-Find can be slow because its Union operation requires iterating through all of the elements, and Quick-Union can be slow since it may forge an extremely tall (linear) tree, which takes the Find operation a long time to trace all the way back to the root node. Hence we also explored two improvements.

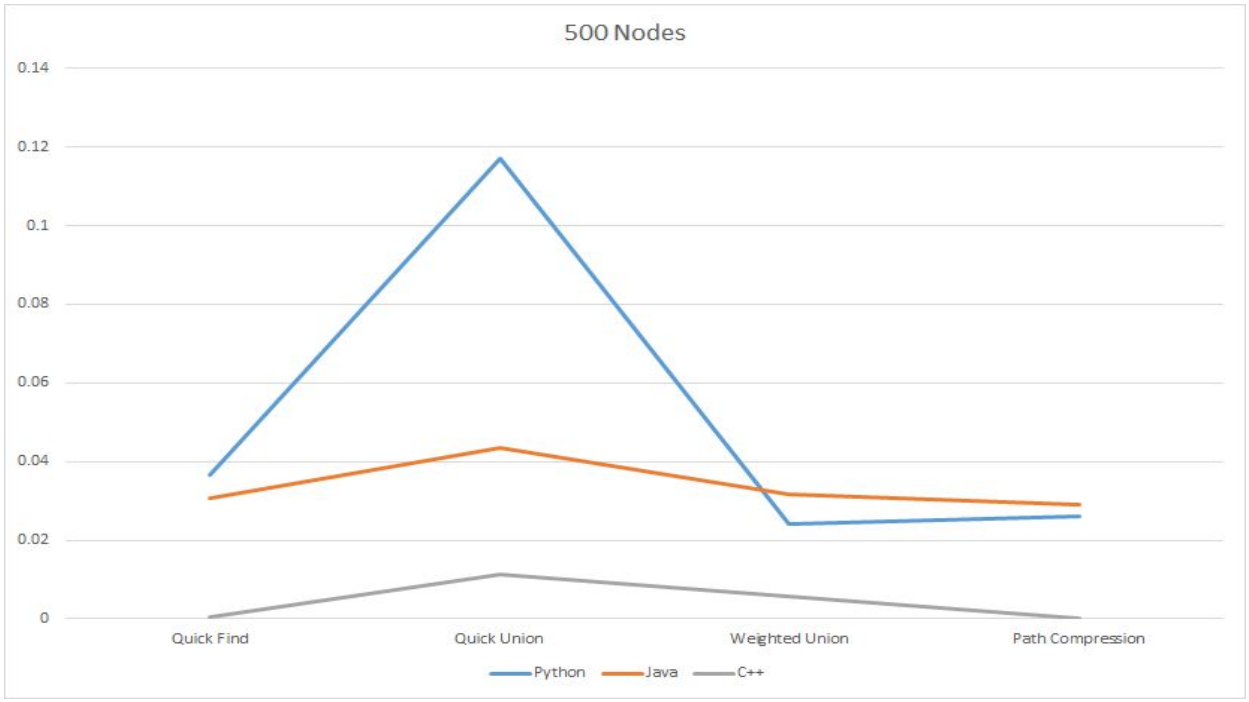
The first one is Weighted Quick-Union, which is built off Quick-Union but avoids the tall-tree problem. When a Union command is called in Weighted Quick-Union, the algorithm check the size of each sub-tree and only adds the smaller tree to the larger tree. This improvement ensures that the depth of the tree will be O(log(N)).

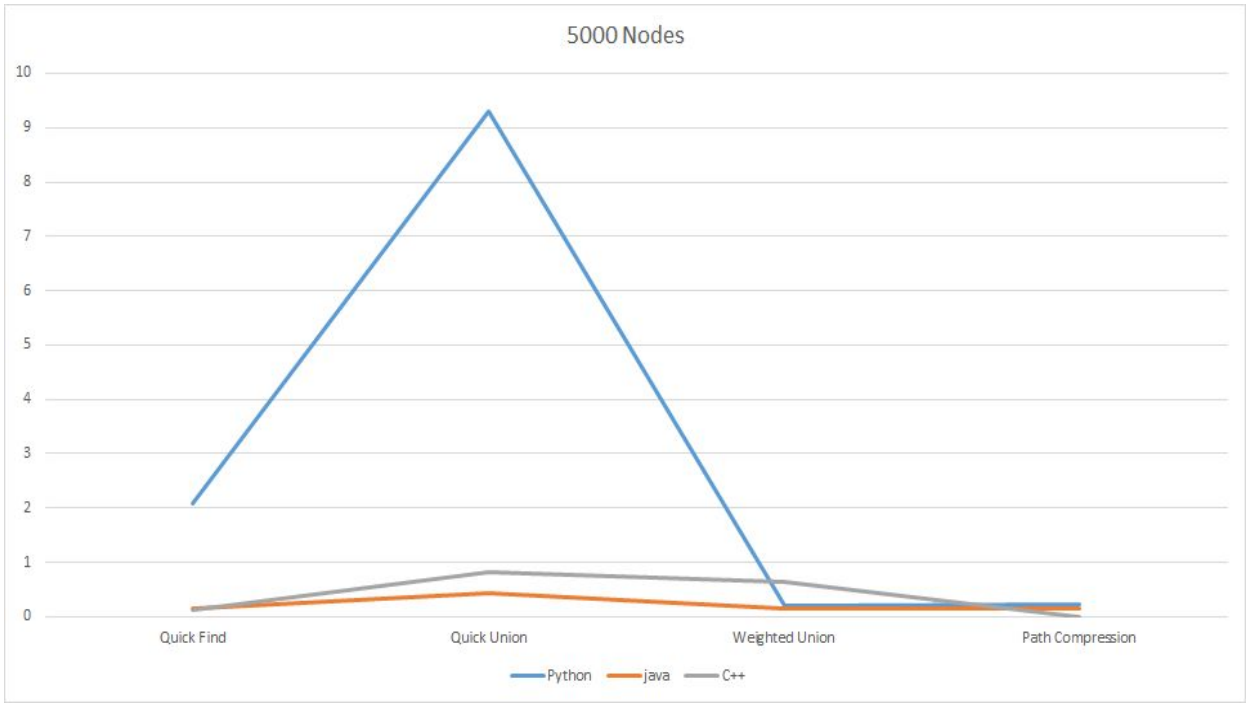
The second improvement is called Path Compression, in which a for loop is used to travel through the tree and set the value of each node to its root. It is easy to repeat the for loop multiple times until the depth of the tree becomes O(1). However, in real life, two to three times of compression is good enough.

To test the performances of the four algorithms, we built a random test case generator. It takes in two values: the number of nodes and the number of Union commands. For instance,

when the generator is called with 500 nodes and 10,000 Union cases, it adds 500 elements to an empty disjoint-set data structure and randomly unioned two elements together for 10,000 times.

Results & Discussion





In the graphs above, the X-axis represents four kinds of algorithms we implemented, and the Y-axis represents the time used in the millisecond. The first graph shows the resulted time

spent of performing 10,000 Union commands on 500 nodes, while the second graph shows that of performing 100,000 Union commands on 5000 nodes. From our tests, we find that the performance of each algorithm across all three languages agrees with the following order, from the best to the worst:

Path Compression, Weighted Quick-Union, Quick-Find, Quick-Union.

As for the languages we used, Python gives the slowest overall running time, and C++, Java the faster. The time differences between Python and other languages are more evident in Quick-Find and Quick-Union and not so much in Weighted Quick-Union and Path Compression. We suspect that something abnormal happened to python's Quick-Union function since the leap was so huge that we could not give any reason to justify it. Thus, unfortunately, we have to omit that from our discussion.

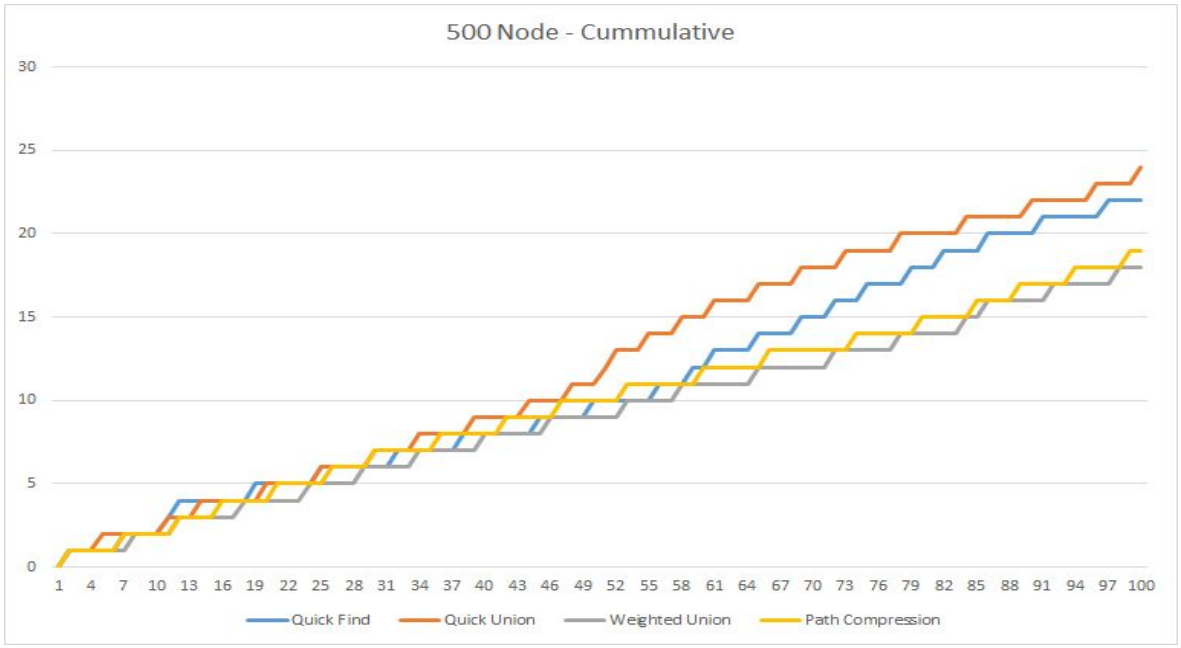
Below are two graphs drew from the same set of testing data that we collected but in a different style. The X-axis represents the recorded cumulative time cost for each 100 Union operations. For example, 3 in X-axis means the point of completing 300 Union operations. The Y-axis still represents the time in millisecond. Notably, Quick-Find outperformed Quick-Union after about 10,900 Union operations. To explore this intersection, we added a counter was added to the program to record the number of operations it took until all elements are in one set. The counter showed that for 500 nodes trials, it took on average 1,300 Union operations to unite all elements into one set, and for 5,000 nodes trials, it took about 18,000 Union operations. This result suggests that as the number of disjoint sets becomes relatively small, the Union operation is no longer the determining factor. Instead, the Find operation dominates the running time, which cost O(1) in Quick-Find and O(N) in quick-Union. Thus Quick-Union became much slower than Quick-Find after all elements are in one set. Furthermore, in the long run, Quick-Find also outperforms weighted quick union in 5000 nodes.

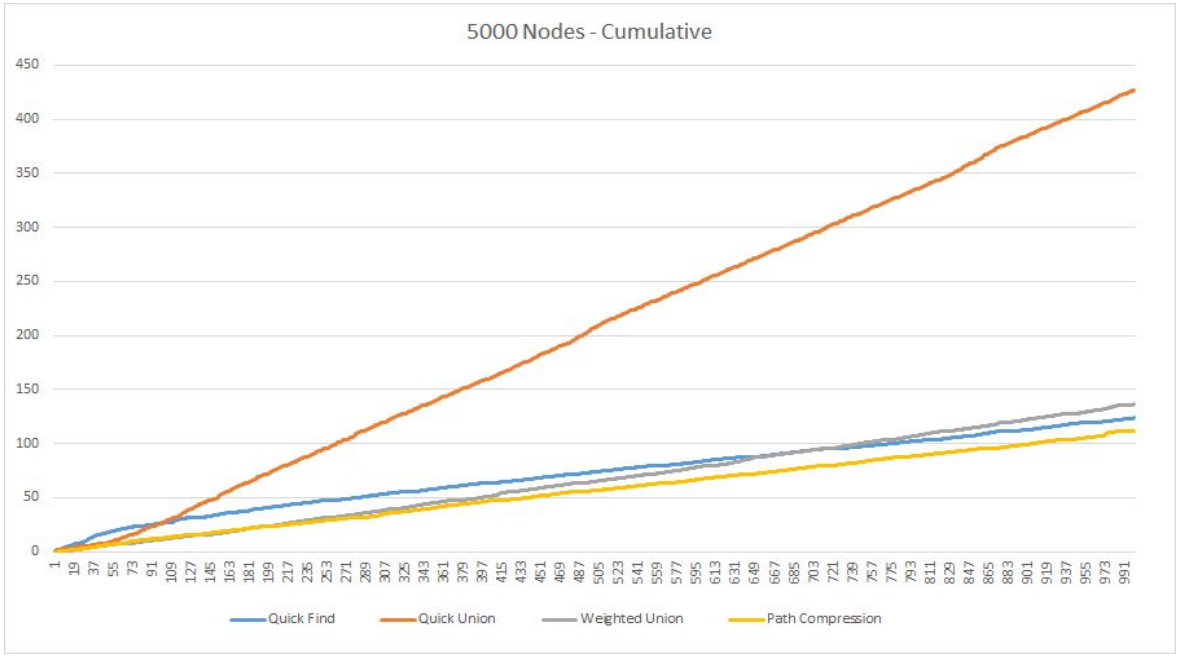
For the same reason above, as the number of disjoint sets becomes small, find operation dominates, in which Quick-Find find takes O(1)time, and Weighted-Quick-Union takes O(log n)time. Lastly, Path Compression, which also includes the idea of Weighted-Quick-Union outperforms all other three because, for each union, it takes O(log*n) times. O(log*n) can be approximated to some small constants.

The running times of the four algorithms under our synthesized setting generally agree with our expectations. From fastest to slowest are:

Path-Compression, Weighted Quick-Union, Quick-Union, Quick-Find.

Our results show that Python is relatively slow compares to C++ and Java. Quick-Union and Quick-Find in Python are slower than those in Java by twenty folds. We suspect that Python is slower because of its dynamically typed nature that Python needs to generate dynamic dispatch for almost all operations. On the other hand, JVM converts Java code to codes that are at a reasonably low level, which enhance java performance.



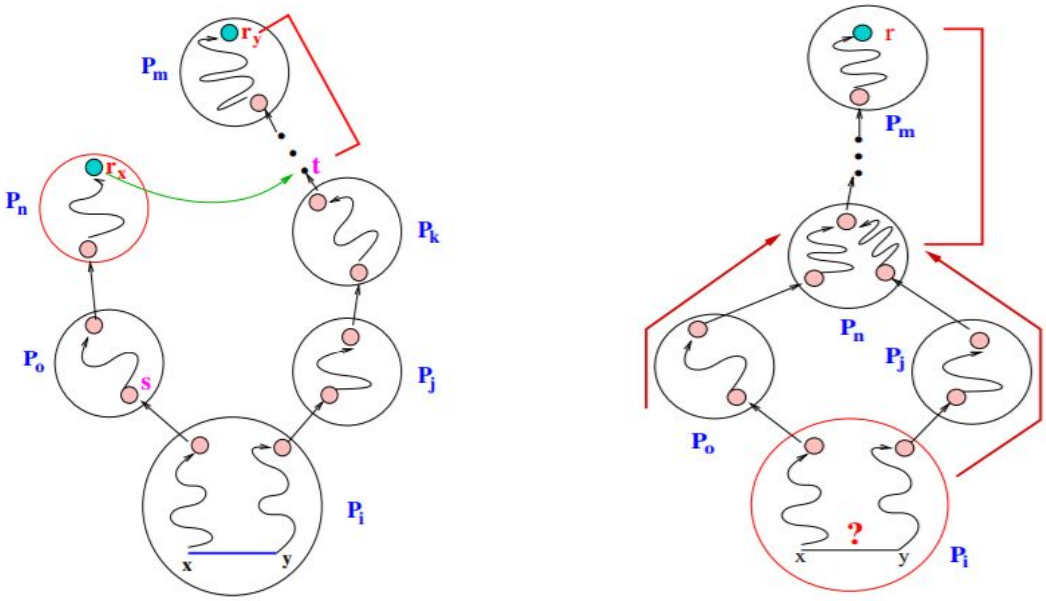


Summary on parallel computing

Another way of improvement that we have learned from reading the literature is called parallelization. Here we will illustrate it with an example. Suppose that we have a distributed memory computer which has multiple processors and we want to find the spanning forest of a graph G. We can start with partitioning the graph G into smaller subgraphs and assigning them to each processor. For each processor, we will delete the crossing edges from each subgraph and find the spanning forests of the remaining subgraph (F1) and the graph formed by deleting

crossing edges and ghost nodes (F2). Then for each edge in F2, we will execute the "Zigzag Union-Find" Algorithm to construct a new spanning forest for each subgraph. Note that during the Zigzag Union-Find process, processors are not working independently. Instead, each processor receives and sends information (to keep or discard the edge) to other processors. Finally, the global-spanning forest of G is just the Union of all spanning forests of the subgraphs.

About the "Zigzag Union-Find" algorithm:

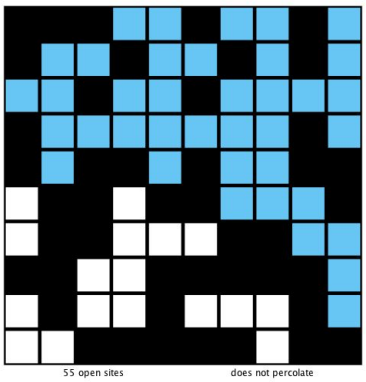


As one observes from the graphs above, each circle P represent a processor and the nodes and edges within the circle represent the spanning forest derived from subgraph after deleting the crossing edges, and the goal is to find out whether the edge (x, y) need to be added to the graph or not. The algorithm will traverse backward from x and y alternatively. If it has already reached the root of x, but with the distance traversed, it has not reached the root of y yet, then it will simply make the parent of the node on the path started from y to be the parent of the root of x, as shown in the graph on the left. If instead, after traversing through the paths starting from both x and y, it has reached the same spanning forest, as shown in the graph on the right, then it would be safe to say that (x, y) need not be added since x and y are already in the same spanning forest. The advantage of this algorithm is that in either case, time traversing the edges that we do not need to care about is saved — either the edges from t to Pm on the left or the edges from Pn to Pm on the right.

Summary on Percolation Project:

A matrix percolates if there exists a path that can connect top and bottom.

As the figure shown below:



If only focus on the blue entries, the matrix does not percolate.

The **basic rule** is the following:

If an entry is not open, color it black.

If an entry is open and connects with the top row, color it blue.

If an entry is open and not connect with the top row, color it white.

Initially, all entries are color in black because none of them are open when we open an entry, then it is going to union with all its' neighbors who are also open as well. The input is a text file. Each line in the text file contains two numbers indicating row and column. These two numbers indicate the entry that is going to be open.

Why Union-Find:

Union-Find algorithm is useful in checking whether a matrix percolates because the Union-Find algorithm can detect percolation in O(1) time. On the other hand, other naive methods will take $O(n^2)$.

To achieve O(1) time, we added two more dummy rows: one dummy top row on top of the matrix, and another dummy bottom row at the bottom of the matrix. Then, we union all entries for dummy top row, and do the same thing for the entries inside the dummy bottom row as well. After we built these two dummy rows, to check whether matrix percolates, we only need to arbitrarily pick one of the entries from the top dummy row and one of the entries from the bottom dummy row by checking whether these two entries are in the same set or not. As mentioned before, using quick-Find's find method can help detect percolation in a matrix in O(1) time. Or if union time also needs to be considered, we can choose the Path-Compression version but in this case, the running time for checking percolation becomes O(log*n).

Supplements

Did you meet your milestone? If not, what made you fail to achieve it?

Yes, we meet our milestone. In the mid-term report we planned to do the followings:

1. Implement Union-by-rank & path compression demo; come up with other ways to further improve Union by rank & path compression and implement them

2. Build a workload generator that tests the performance of the implementation.
3. Learn about parallel computing paper, write a summarization on the report.
4. Present Union-Find Algorithm & concept of parallel computing to the class.

We included 1, 2, and 3 in the final report and we presented 3 and 4 in the class.

What did you learn from this project?:

First of all, we learned a new algorithm that is commonly used in the industrial world. Steven encountered an interview question which the interviewer asked him to solve the problem (number of islands) using Union-Find Algorithm. By carefully studying and implementing such an algorithm, he felt more confident having this algorithm as a tool for future interviews. Also, we learned how to better collaborate with busy schedules. It was quite a struggle to find a time that works for all of us to meet, but the experience of working together had been rewarding. Lastly, this project pushed us to read research papers that we tended to avoid before. Even though we had a hard time understanding the papers, we started to get used to reading these papers after finishing reading a few.

Do you agree to share your code/report with other students, say in other classes or future Algorithms class?

Yes. You can find the source code for our project at GitHub: https://github.com/SpicyGoose/2018_CSCI3383Algo_UnionFind

[Bonus: up to 5pt] Do you have any suggestions on final project? E.g., more help? More resources? Format of poster session? Workload?...etc. Please be specific.

1. For the final presentation, we thought it could be better if we are given some background information about other group's project beforehand. Given the 15 minutes time slot of presentation, it was tough for the audience to follow and understand the details of what the group was doing. We could only manage to get a general idea.
2. We think that doing a poster session may be fun so that we can share our project with students and professors out of the class.

References

Patwary M.M.A., Blair J., Manne F. (2010) Experiments on Union-Find Algorithm for the Disjoint-Set Data Structure. In: Festa P. (eds) Experimental Algorithms. SEA 2010. Lecture Notes in Computer Science, vol 6049. Springer, Berlin, Heidelberg

Ben-Amram, Amir, and Simon Yoffe. “A Simple and Efficient Union–Find–Delete Algorithm.” Theoretical Computer Science, 2009.

CS, Princeton, director. Princeton Union Find Algorithm. YouTube, YouTube, 24 Aug. 2014, www.youtube.com/watch?v=gfSpPbJWzVs&list=PL5iJcUfx7xTdBIAUg4pco1qUK-VOLekNo.

Manne, Fredrik, and Md. Mostofa Ali Patwary. “A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers.” *Parallel Processing and Applied Mathematics Lecture Notes in Computer Science*, 2010, pp. 186–195., doi:10.1007/978-3-642-14390-8_20.

Simsiri, Natcha, et al. “Work-Efficient Parallel Union-Find.” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 4, Sept. 2017, doi:10.1002/cpe.4333.