A review of Afforest, and SV derivative

Christopher Blackman

December 14, 2019

Abstract

Graph Connectivity algorithms play an important part in many of today's infrastructure. This topic has been thoroughly studied by many introducing distributed algorithms, parallel algorithms, GPU-based algorithms. Many of these algorithms focus on the number of edges, however optimal work is done when running time approaches $O(|V|)$. We approach an algorithm called Afforest based on the well known Shilovach-Vishkin connected components algorithm. We prove the average case of Afforest, and demonstrate performance in large connected graphs.

1 Literary Review

As we approach the limit of De Morgan's law, we come to an age in computing where sequential computing is limited by processing speed. Using RAM (Random Access Memory) as our base model of computation we create optimal sequential algorithms. However, in the age of big data, these sequential algorithms are not powerful enough to process all the data in reasonable time. Thus, we turn to parallel computing to where we have two models of computation which attempt to divide work. The first model assumes that all knowledge is known about the problem, then divides work base accordingly; this is known as the field of parallel computing. The second approach assumes that only local information is known about the problem, and through a series of steps, protocol definitions, and message passing an answer can be derived/approximated; this is known as the field of distributed computing. However, there are trade-offs to both approaches like : speed, run-time, correctness, optimality. Most algorithms take a trade-off between these two fields as computation gets further away from processors.

In this age being able to extract information form graphs becomes more prevalent in analyzation of web link graphs, social networks, and many other graphs types. One property that we can extract is graph connectivity. In graph theory graph connectivity is the minimum number of vertices, or elements that need to be removed to separate nodes into isolated sub-graphs [7]. A very similar definition of this is the minimum cut of a graph where a minimum cut is defined as the minimum partition of vertices, and edges into two disjoint sets. For simple unweighted graphs David R. Krager et al [8] created a randomized sequential algorithm based on properties preserved when edge contraction is applied. The papers by David R. Krage et al [9], and Barbara Geissmann et al. [10] further improve the algorithm by adding parallelisation. The downside of parallel algorithms introduced is that they are still based on the Parallel Random Access Memory (PRAM) model which can be used to prove theoretical bounds on algorithm. But the algorithms introduced disregarded hardware limitations where CPU cache is finite, and message passing to start up threads/processes are costly. It is noted that the paper in Barbara Geissmann et al. [10] attempts to improve cache misses by creating monotonic sequences to be read, however, the paper still relies heavily on parallelism of subroutines.

Another version of this problem comes from weighted undirected graphs. The goal is to find a cut where the sum of the edge weights are minimum. The Stoer-Wagner algorithm by Mechthid Stoer et al. [11] is a good sequential recursive algorithm for solving undirected weighted graphs with non-negative weights in $O(|V||E| + |V|^2 \log |V|)$ time. Usually the min-cut with respect to weighted undirected graphs is linked with the maximum flow problem as the minimum cut is often the bottleneck of the network [12,13].

A subfield of graph connectivity is connected components. Connected components can be divided into two categories: strongly connected components (SCC), and weakly connected components (WCC). A graph is said to be strongly connected if every vertex is reachable from every other vertex. Then a SCC is the same definition, but the maximal sub-graph that keeps this property. The SCCs of a directed graph form partitions of sub-graphs where they themselves are strongly connected. WCCs of a directed graph form a sub-graph for every pair in the sub-graph there exists a path, but not necessarily a bijective path.

One of the earliest algorithms introduced is Tarjan's algorithm which uses the properties of depth first search (DFS) to find SCC given by Robert Tarjan [16]. Later Edsger W. Dikstra came up with the path-based strong component algorithm seen in [17]. A simpler algorithm was also found after named Kosaraju's algorithm and published by Micha Sharir [17]. However, all of these algorithms abuse depth first search to find SCC, yet for parallel algorithms DFS is hard to paralyze. In fact, it was proved by John H. Reif that DFS is P-complete [19]. Therefore parallel algorithms do not focus on DFS when computing strongly connected components seen in Lisa K. Fleischer et al. [20], and Sungpack Hong et al. [15].

There is another version of Connected Components in undirected graphs. Unlike the directed version, if there is a path, then that path is a SCC since edges are bidirectional. Therefore the undirected version of SCC focuses on finding disjoint sub-graphs in the original graph, since each disjoint sub-graph is itself a SCC of the original graph. One may also relate this to finding WCC in a directed graph, since a path represents a bijective edge in undirected graphs. However, by doing so the WCC may contain a SCC in the original graph. A simple sequential algorithm can be done to find this where you take a DFS, or a breadth first search (BFS). Once you have exhausted your search, you select a non-traversed vertex to find the next SCC [21]. There have also been algorithms for dynamically changing graphs researched by Yossi Shiloach et al. [22].

As graphs increase in size, so does the amount of processing power required. In the field of SCC on undirected graphs there have been several subfields in which parallelization has taken affect. The first parallel algorithms dealing with SCC on undirected graphs deal with spanning forests [23,24,25,26,27,28,29,30,31,32,33,34,35,36,37]. The algorithms by Uzi Vishkin et al. [36,4], and Awerbuch et al. [37] work by combining vertices into trees, such that a constant number of vertices are deleted every iteration, but does not guarantee a constant fraction of edges. Thus they require $O(|E|log|V|)$ work. The random algorithm by Reif, and Phillips work by contracting vertices in the same component, which guarantees a constant number of vertices every iteration, however, it again does not guarantee a constant fraction of edges. Thus the expected work is $O(|E|log|V|)$.

Work-efficient poly-logarithmic-depth parallel connectivity algorithms have been designed in theory [38,39,40,41,42,43]. These algorithms are based on random edge sampling, or filtering edges. However, these have complicated structures which may be impractical to implement in practice.

There have also been CC algorithms focused in the field of distributed computing, where labels are propagate throughout nodes. A certain rule is imposed at each node describing when, and how to propagate a label. When no more labels are propagated the algorithm completes. Min-Label Propagation [3] algorithms generally have work done in $O(D|E|)$.

Another kind of algorithm tries to emulate a Breath First Search (BFS). Where agating in parallel from a single vertex until a single component is traversed. The downside is that, each component must be search in sequential order, thus what we obtain in lack of conflict resolution opposed to distributed algorithms, we also get serialization. Moreover, more algorithms have taken a 'bottom up' approach reducing work to sub-linear in $|E|$[2].

The algorithm which we will be focusing on is based on Shilovach-Vishkin[4] which has $O(\log n)$ parallelism, an uses a maximum of $2|E| + |V|$ processors. Shilovach-Vishkin represent their CC as a forest where each tree represents a connected component. They have two key phases *hook*, and *shortcut*. *Hook* is responsible for connecting processed components together where it hooks new nodes with their respective component, or connected two components together given the current knowledge know of the graph. They prove that this process results in tree that have height at most $\lfloor \log_{3/2} n \rfloor + 2$. The *shortcut* phase results in a compression of the trees in the forest such that they have depth at most one.

2 Statement of Problem

The key problem with Vishkin-Shilovach is that work was done on a per-node basis, such that there may be duplicate reads to the same edge. Furthermore, if there is a large component in the graph, then the algorithm would read all edges that exist within that component. Thus, if we have a large complete graph, then all $|E| = |V|^2$ edges in this complete graph would be traversed, when only $|V|$ edges are needed in-order to determine a CC.

A solution to this is to sample for the largest component in the graph, and skip any node not processed in this component. This algorithm was proposed by Michael Sutton et al.[1] where they attempt to do super linear work on the number of edges in the graph, such that no edges are visited twice. The algorithm that they present is called *Afforest*, which is similar in nature to the algorithm presented by Shilovach-Vishkin. We will be performing a review on *Afforest* gauging performance, and filling in missing proofs not fully described in the original paper.

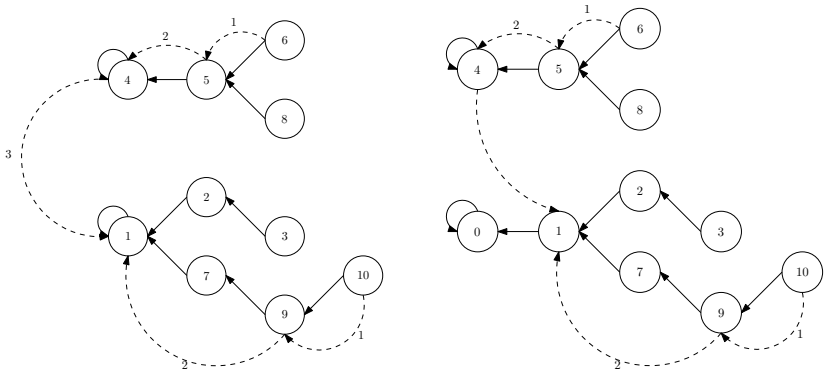
Afforest

Similar to Shilovach-Vishkin, Afforest's underlying data structure is a forest. In this forest each tree is represented as a connected component found thus far in the view of the processed graph. For all $v \in V$, v has a unique identifier. We denote the symbol $\pi(v)$ to represent the parent of v, such that we hold an invariant in which $\pi(v) \leq v$. We note that if v is the root then $\pi(v) = v$.

The basis of Afforest is contingent on two operations : link, and compress. Link is seen in Algorithm 1. Link ensures that two verticies will be in the same component tree. Link searches the tree(s) for a root with the larger identifier. Upon finding the root with a larger identifier link attempts a Compare and Swap (CAS) operation in order to connect the larger tree to the smaller identifier. The smaller identifier does not necessarily need to be the root of the tree. We can see that our invariant still holds.

Algorithm 1 Link Operation

```
1: procedure LINK(edge(u, v) , π)
2:     p1 ← π(u)
3:     p2 ← π(v)
4:     while p1 ≠ p2 do
5:         h ← max{p1, p2}
6:         h ← min{p1, p2}
7:         if compare_and_swap(π(h), h, l) then return
8:         p1 ← π(π(h))
9:         p1 ← π(l)
```



(a) Case where CAS happens at both roots

(b) Case where CAS happens at one root

Figure 1: Link operation traversing up component trees linking edge(6,10)

In figure 1, we see the operation of adding the edge with $u = 6$, and $v = 10$. We see that in both cases we traverse up to the largest root 4, and we connect the tree rooted at 4, to the tree rooted at 1. It can be seen that not matter where we connect root 4, there is always a path from from every element in the tree rooted at 4 to the new component tree root 1.

Lemma 1. If the endpoints of link u, and v are within the same component tree, then the ancestors of the component tree do not change. Since a CAS operation is only triggered when the largest node is the root, and since our invariant proves that the root is the smallest value in the tree, the CAS operation is never triggered. ■

Lemma 2. If the endpoints of link u, and v are not within the same component tree, then the link operation will ensure that both trees are merged into one component tree. There are three possible cases that we need to cover to prove this.

1. If the processors' h remains the root, and CAS succeeds, then we know that h, and l are directly connected. Resulting in one connected component
2. If another processor connects h to the tree, then we apply lemma 1. This results in one connected component.
3. If another processor connects h to l' such that l' does not live in the tree of u, nor the tree of v. Then $h \neq \pi(h)$. We then repeat searching up the tree, and we end up with the same problem as described in Lemma 2. Since there are a finite number of vertices, then there are also a finite number of components. Therefore the operation will complete results in one connected component containing both u and v.

■

Lemma 3. If we apply link to all edges, then the resulting component tree represents the CC of the original graph. $\forall edges(u, v)$, u and v will be in the same component since we apply link to all edges. Then by Lemmas' 2, and 3 all edge pairs will be in the same component.

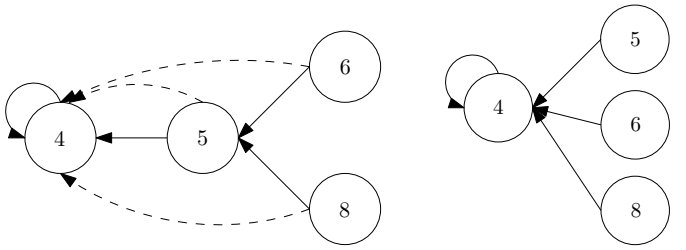
By definition a CC is connected if there exists a path between all pairs in the component. If we take one such path $x_1, x_2, ...x_{n-1}, x_n$ then link would be applied to : $(x_1, x_2), ..., (x_{n-1}, x_n)$. Which means all the nodes in the path will be in the component tree. If we take take the set of all paths and convert it to the set of all edges, then every vertex that is an element of a CC C_i, is also an element of a unique component tree corresponding to C_i. Since we apply link to all edges, then we cover the set of all possible paths. Therefore the resulting component tree represents a connected component in the original graph. ■

The second most important operation of Afforest is the *compress* operation seen in Algorithm 2. Compress makes a single node search for the root of its component tree. If we apply compress to all edges, then this results in a depth one tree. This is seen in figure 2 where a tree is converted to a depth one tree.

A depth one tree is obtained when all nodes apply the compress operation. At each iteration in the while loop a node reduces it's depth by at least one. We say at least one since at any time another processor could be modifying the path, however, any modification to the path results in a shorter path. Once a node has reached the root of the tree we stop, thus create a depth one node. If we apply this to all nodes then all nodes will have depth one.

Algorithm 2 Compress Operation

```
1: procedure COMPRESS(v , π)
2:     while π(v) ≠ π(π(v)) do
3:         π(v) ← π(π(v))
```



(a) Before Compress Operation

(b) After Compress Operation

Figure 2: Example of a Compress Operation

One might notice that we traverse that we may traverse the whole tree in link, and that we may traverse the whole tree in compress. This would result in a sequential span of the maximum height of one of these tree. If under the assumption that only the roots of the tree change, we propose an exact expected value for the height of the tree. If our assumption is false, then we still would have obtained a lowerbound on the expected height of the tree. In order to prove this we must look at Union-Find data structure invented by Kruskal[46].

Union-Find is an algorithm similar to Afforest. The data structure consists of a forest of trees, such that each tree represents components in a union. If we want to find the union of $\{x_1, x_2\}$ then the algorithm finds the root of x_1, and the root of x_2. If the roots are in the same tree then no work is needed to be done since they are in the same set. If the roots are not in the same tree, then

smallest root is chosen as the new root of the tree such that the largest root is a child of the smallest root. In other words if $x_1 < x_2$, then set $\pi(x_2) = x_1$. Under the assumption we made that only roots change similar to figure 1 a), then this data structure exhibits the same properties as Afforest.

Obviously the worst case results in a tree with length n, where n is the number of elements in the set. Therefore let us consider the expected case. First we assume that the addition of union-find requests are uniformly random. Second, we only focus on the largest tree during formation. By doing so we get an upper bound on the expected height, and we can apply backwards analysis.

Thus, let X be the height of the largest tree. Let $X_i = 1$ when we pick a node at iteration i that increases the height, and let $X_i = 0$ otherwise. There is no bound on the number of iterations, but we choose a bound n as we can form any permutation of sets, thus each iteration results in a union. Now we consider the largest tree T. At any iteration i, T grows at most by 1 as we add an edge, or we add a tree which is at most the height of T. If we attempt to build our tree with $i = 0$, it is hard to count. Therefore, let us consider T at iteration i. Between iteration i and $i - 1$ we choose one pair out of the $(n - (i))$ possible choices that will increase the height of the tree. Therefore the probability that we choose that pair between one iteration is : $\frac{1 \cdot (n-(i+1))!}{(n-i)!} = \frac{(n-i+-1)!}{(n-i)!} = \frac{1}{(n-i)}$. Therefore if we apply this over all iterations then the expected maximum height is :

$$E(X) = E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{1}{n-i} = \sum_{i=1}^{n} \frac{1}{i} \leq Hn(n) \leq ln(n) + 1$$

Therefore we have showed that the expected height is not more than $O(ln(n))$, but how much does this deviate from the expected value. We can use Markov's Inequality to calculate with high probability that this is true. We assume that $a = n^{-d}$, where d is a constant. We can then use Markov's Inequality to prove that :

$$P(X \geq a) = P(X \geq n^{-d}) \leq \frac{E(X)}{a} = \frac{ln(n)}{n^{-d}}$$
$$lim_{n\to\infty} \frac{ln(n)}{n^{-d}} = lim_{n\to\infty} \frac{n^{-1}}{(-d)(n^{-d-1})} = 0$$

Therefore as n approaches infinity, then we can say with hight probability that our tree does not exceed a height of n^{-d} since $1 - P(X \geq n^{-d}) = P(X \leq n^{-d}) = 1$. This is reassuring since the Taylor series of log contains a series of additions, and subtraction of exponentials. Therefore we can say that this is a good approximation to logarithmic height.

If we did not make the assumption that addition of nodes is only at the roots, then the algorithms would not be the same. However, if we replace the double

jump, with a single jump in our link operation on line 8, then we can remove our assumption. By removing the line, we ensure that when we connect two components, the height increases by at most one, and we can use the same backwards analysis technique that we devised for union find. Therefore, with slight modification the maximum depth of the algorithm is at most $ln(n)$.

Furthermore, although not specified in the original paper of Afforest[1], it may be possible to remove the sequential span of the algorithm by traversing up the tree by performing a parallel find head of link list. However, one would be doing $O(n)$ work, instead of $O(ln(n))$ work in order to get a span of one, which may more expensive. This is due to the fact that we would be searching in a forest of linked lists, instead of a single linked list.

Sampling is second key component of Afforest. By sampling, Afforest is able to able to reduce the amount of work that it performs, thus speeding up the algorithm. Afforest does this by taking a linear amount of nodes to sample for the largest CC. Afforest then processes all the vertices that are not an element of the largest CC, and processes all the edges associated with them. Since we are performing work on an undirected graph, edges are bidirectional therefore if we missed a component that is a subset in the largest CC, there is always an edge that we do not skip that is still connected to the component. If we look into figure 3 we can see that we skip a majority of the edges in the largest component, and process the rest.

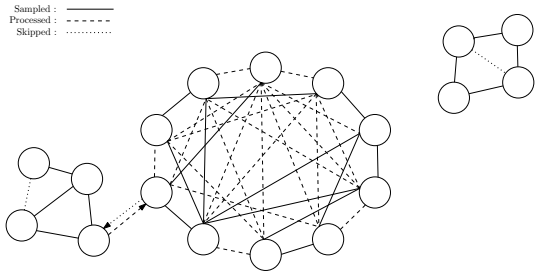


Figure 3: Component skip : where we skip edges in the largest component, and process the rest of the graph

The general reason sampling might be a good idea comes from from the work seen in Frieze et al.[47]. Frieze et al. showed that given a d-regular graph G, and a subset of the graph G' or the sampled graph they showed that by sampling edges with $p \geq \frac{\alpha}{d}$ for some $\alpha > 1$, then G' contains a component of size $\Theta(|V|)$.

By doing so we can prove that the expected number of edges in G' is $O(|V|)$. By definition of G $|E| = \frac{d}{2}|V|$. Each edge is sampled with probability $p = \frac{\alpha}{d}$, then the expected number of edges in G' is $p \cdot |E| = \frac{\alpha}{2} = O(n)$. This can be

generalized to multiple component regular d graphs such that we still obtain the same $O(n)$ bound on the sample size.

We note that this does not generalize to none d-regular graphs, but gives us an opportunity to apply a type of model onto the problem. Furthermore, uniformly sampling from G creates a bias towards vertices with with high degree. Instead, we sample on neighbourhoods such that the number of edges we sample in the neighbourhood of v have probability of $\frac{\alpha}{|N(v)|}$, such that $N(v)$ is the neighbourhood of vertex v. If our graph is d-regular, then the proof still applies.

We now present the Afforest algorithm in algorithm 3. We first obtain a sample graph, by sampling a number of rounds. We then compress all the nodes to depth one, and find the most frequent component in our sample. We then skip any vertex in our sample, and process the remaining edges. With everything discussed above, this results in every vertex knowing which component it belongs in. A key different of our algorithm compared to the algorithm provided by Michael Sutton et al.[1] is that we perform our compress after sampling the graph allowing us to perform all sampling in parallel.

Algorithm 3 Link Operation

```
1:  procedure AFFOREST(V, E, rounds)
2:      for v ∈ V do
3:          π(v) ← v
4:      for i ← 1 to rounds do
5:          for {v ∈ V : i ≤ |N(v)|} do
6:              link(v, N_i(v), π)
7:      for v ∈ V do
8:          compress(v, π)
9:      c = most_frequent_element(π)
10:     for {v ∈ V : π(v) ≠ c} do
11:         for {v ∈ V : rounds + 1 ≤ i ≤ |N(v)|} do
12:             link(v, N_i(v), π)
13:     for v ∈ V do
14:         compress(v, π)
        return π
```

There was one detail in the original paper by Michael Sutton et al.[1] where they mention that finding the most frequent element is easily obtained with random sampling. However, we could not find any research which indicated this thus we shall prove that with a delta between the most frequent element, and the second most frequent element, we can calculate an upperbound on the number of samples needed for finding the most frequent element.

Let X_1 be the number of elements in our sample containing the most frequent element. Let X_2 be the number of elements in our sample containing the second most frequent element. Let m_i be the number of elements in a set pertaining to the ith most frequent element, k be the total number of samples chosen, and n the total population. Then if we use random indicator variable with (1 : an element is in sample, and 0 otherwise); we get and expected value of $E(X_i) = k\frac{m_i}{n} = kf_i$, such that f_i is the frequency of the ith most frequent element.

If we took a k size sample of elements and choose them most frequent element. Then what we are looking for is the probability that $X = X_1 - X_2 \geq 0$, since $X_1 \geq X_2 \geq ... \geq X_l$. Furthermore, due to linearity of expectation $E(X) = E(X_1 - X_2) = E(X_1) - E(X_2) = k(f_1 - f_2)$. Since we are dealing with a binomial distribution, we can use a Chernoff bound, which is an over approximation of the Taylor series of the binomial distribution.

$$P(X \leq (1-\sigma)\mu) \leq \exp(-\frac{\sigma^2 u}{2}) \; for \; 1 \leq \mu \leq 0$$

Where σ is a value that we choose to obtain the tail end of a distribution, and μ is the expected value of our random variable. If we choose a $\sigma = 1$ then we obtain the following formula:

$$P(X \leq 0) \leq \exp(-\frac{u}{2})$$

We also want to find values when $X \geq 0$ so we solve for the following :

$$\begin{aligned} 1 - P(X \leq 0) \geq & 1 - \exp(-\frac{u}{2}) \\ P(X_1 - X_2 \geq 0) \geq & 1 - \exp(-\frac{k(f_1 - f_2)}{2}) \\ k \geq & \frac{-2(1 - P(X \geq 0))}{(f_1 - f_2)} \end{aligned}$$

Therefore, within a certain delta frequency we can always calculate the most frequent element in constant time with probability $0 < p < 1$. If we look at table 1 we see the exact calculations.

P(X>0)	f1-f2	k
0.9999	0.01	1842
0.9999	0.02	921
0.9999	0.05	368
0.9999	0.1	184
0.9999	0.2	92
0.9999	0.4	46
0.9999	0.5	37
0.9999	0.7	26
0.9999	0.9	20
0.9999	1	18

Table 1: Showing the sample size needed for each difference in frequency for the worst case

3 Analysis

At the core, Afforest is composed of two main procedures : link, and compress. The amount of sequential work that these procedures do is proportional to the height of their trees. Thus, in the worst case these procedures do $|V|$ amount of work traversing, however, we proved that with high probability that the expected height of these trees is at most $ln(n) + 1$.

The amount of work that the Afforest does is bounded by the number of edges that we visit, and vertices that we visit. For all link, and compress operations we perform $|V| + |E|$ work. However, if we sample a very large CC, then we perform $|V| + |V|$ work, which can be nice in graphs that have approximately $|E| \approx |V|^2$.

Therefore, the parallelism of Afforest is equal to $\frac{|Work|}{|Span|} = \frac{|V|+|E|}{ln(n)}$. If we sample a big enough component, then this may also be $\frac{|V|}{ln(n)}$. As mentioned before, one may be able to implement an algorithm for finding the head of a linked list in a forest an bring down the parallelism to $\frac{|V|}{1}$, however doing so results in $|V|^2$ processors being used in compress/link, when only $|V|$ are needed to traverse a list of size $ln(n)$. The maximum amount of processors needed is $\max(|V|, |E|)$ as only $|E|$ are needed for link operations, and $|V|$ are needed for compress operations.

4 Implementation Details

In Afforest the number of rounds that we choose to sample was 2 since it satisfies the property that $\alpha > 1$. For the most frequent element we choose a sample size of approximately 2000 as that gave a probability of 0.9999 for a delta

frequency of 0.01 for the most frequent element, and the second most frequent element. If you know a close approximation for the size of the component, on can always use the specified above to calculate the sample size. Used OpenMp for our as our work stealing scheduler for scheduling threads. Furthermore we used C++ for implementation, and used the standard library atomic header for CAS operations, as well as memory access patterns. The graph we used was a USA roadmap with approximately 90% connectivity. We used a machine with 32 cores, and 64 threads which ran an Intel Broadwell 2697Av4 at 2.6Ghz. The storage format that we choose was to represent out graph as an adjacency list (as indexing was not specifically needed). There have been other implementations that use *csr* matrices for storing redundant vertices. For storing the component graph, we choose to use an array of indices, such that each index represents a specific node. Our component graph represents a directed graph to the root.

5 Experimental Evaluation

In our experiment we we only used one data set (USA road maps), and we tested Afforest based on itself with random sampling, and itself without random sampling as a baseline. In the experiment we saw that Afforest with random sampling was faster than its counter part with a speedup of about 2-3 times its counterpart. However, both level off fairly quickly compared to optimal. This could be due to the fact that a CAS operation might be focusing on only a couple nodes in the component tree. The results can be seen below in figures 4, and 5.

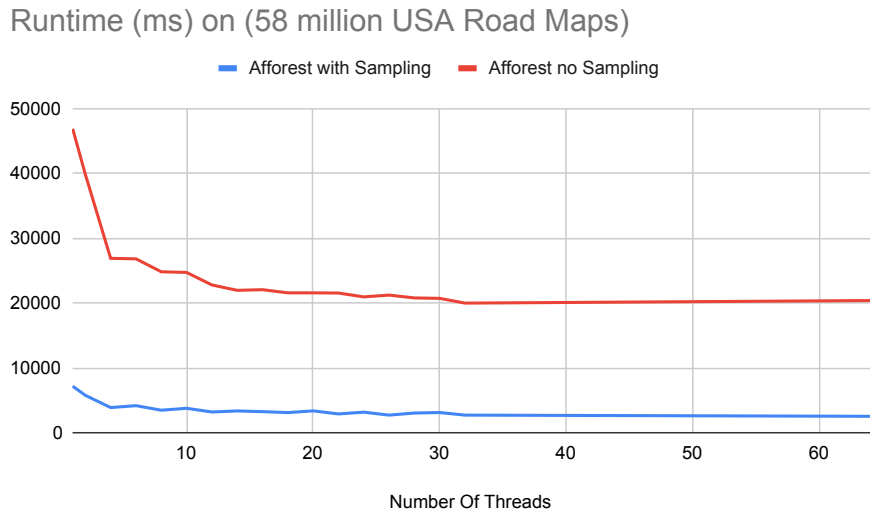


Figure 4: Runtime of both algorithms in milliseconds

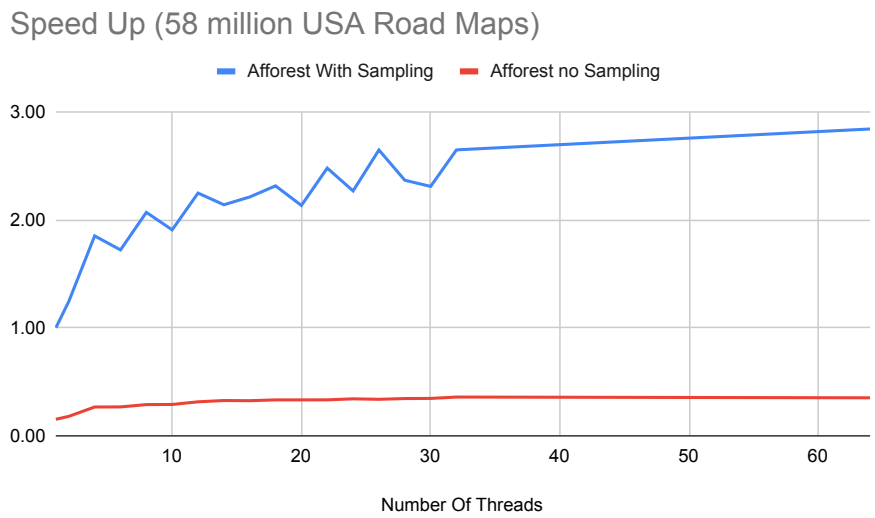


Figure 5: Speedup of both algorithms such that $T(1) = \textit{Afforest with Sampling}$

6 Conclusions

In conclusion, through this review we have filled in some of the missing theoretical knowledge of Afforest. However, it has opened up more topics about Afforest. What is the key limiting factor for the experiment done. Does it pertain to a couple nodes having excessive CAS writes to certain roots, and is there a way to distribute if that is the case. Have we created a processor efficient algorithm for finding a small list in a list of lists in unsorted order. If we implemented a fast parallel pointer jumping algorithm for this algorithm, would it speed up our algorithm, or slow it down since we already know that the expected size of our list is $ln(n)$. Is there a relation of the diameter of a connected component, with the height of our trees in Afforest. It is obvious that if depth is two, then the maximum height is going to be two. But is there a way of proving that fact. Knowing this, does the algorithm perform differently on different type of graphs. These are question that are still left unanswered after this review.

7 Bibliography

[1]https://ieeexplore-ieee-org.proxy.library.carleton.ca/document/8425156/references#refere
[2]https://ieeexplore-ieee-org.proxy.library.carleton.ca/document/6877288/references#refere
[3]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2733089
[4]https://www-sciencedirect-com.proxy.library.carleton.ca/science/article/pii/019667748290
[5]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=2612692
[6]https://link.springer.com/chapter/10.1007/3-540-45591-4_68
[7]http://diestel-graph-theory.com/index.html
[8]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=234534
[9]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=331608
[10]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=3210393
[11]https://dl.acm.org/citation.cfm?id=263872
[12]https://link.springer.com/chapter/10.1007/978-0-8176-4842-8_15
[13]https://dl.acm.org/citation.cfm?id=1942094
[14]https://dl.acm.org/citation.cfm?doid=362248.362272
[15]https://dl.acm.org/citation.cfm?id=2503246
[16]https://ieeexplore.ieee.org/document/4569669
[17]https://dl.acm.org/citation.cfm?id=550359
[18]https://www.sciencedirect.com/science/article/pii/0898122181900080
[19]https://www.sciencedirect.com/science/article/pii/0020019085900249
[20]https://dl.acm.org/citation.cfm?id=663154
[21]https://dl.acm.org/citation.cfm?id=362272
[22]https://dl.acm.org/citation.cfm?id=322235
[23]https://www.sciencedirect.com/science/article/pii/0020019082901314
[24]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=739745
[25]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=305744
[26]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=254195
[27]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=182530
[28]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=359141
[29]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=214077
[30]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=106163
[31]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=203622
[32]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=358650
[33]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=72952
[34]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=586952
[35]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=1398907
[36]https://www.sciencedirect.com/science/article/pii/0166218X84900192
[37]B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and
[38]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=237563
[39]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=123143
[40]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=247524
[41]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=377897
[42]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=586952
[43]https://dl-acm-org.proxy.library.carleton.ca/citation.cfm?id=686427
[44]S. Hambrusch and L. TeWinkel. A study of connected component labeling algorithms on the

[45]S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh-connect
[46]https://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/home.html
[47]https://arxiv.org/abs/1605.06643