Survey of Strongly Connected Components Algorithms

Andrew Gamble (20703630)

CS 466 Spring 2020

Abstract

In this paper we will be comparing and contrasting algorithms for finding strongly connected components in directed graphs. First we will consider sequential algorithms built upon techniques such as depth first search and topological sorting. These algorithms give linear time, $O(n)$ run-time but due to the sequential nature of a depth first search are hard to run in parallel. This prompted the creation a new form of reach-ability based algorithms, whose implementations take advantage of parallelism. Although this trade of comes at the expense of a total run-time of $O(n \log n)$. We will explain both types of algorithms and show the advantages and disadvantages of both, and discuss situations which where either would be favourable.

Contents

1 Introduction

Finding strongly connected components (SCCs) in a graph is a very natural graph problem with many applications in computing. A strongly connected component within a graph is a maximal set of vertices where there exists a path from ever vertex in the set to every other vertex in the set. In the course we have seen the usefulness of the property of strongly connectedness in the definition of a irreducible Markov chain [5]. Another more applied example of strongly connected components is finding cyclical elements within a graph. This is useful for many types of Electronic Aided Design (EDA) software which lays out digital circuits. For instance, within a section of logical elements, loops must be detected to properly analyze the timing of a circuit. Since digital circuit complexity is growing rapidly in size EDA software compile time requires fast and efficient solutions to find SCCs within a graph. EDA software is just one example of the importance of finding SCCs effectively, but SCC algorithms are also important to a variety of other spaces such as DNA sequencing [6].

1.1 Definitions

We will now define some definitions that will be use throughout the paper.

- Let $G = (V, E)$ be a directed graph.
- There is a path from vertex v to u: $v \to u$
- Let C be a SCC defined as a maximal set $C \subseteq V$ s.t $\forall\ (v, u) \in C,\ v \to u$ and $u \to v$. In other words all vertices in C have a path to one another.
- Let G^T be the graph G, where the direction of all edges are flipped.
- Let $\delta(S)$ be the set of all edges leaving or entering a set of vertices S.
- Let G^C be the component graph of G, where all SCCs have been replaced with a supernode. For each SCC: $C \in G$ replace it with a supernode with incident edges $\delta(C)$.
- For $v \in V$, let v_d, v_f be the discover and finish time respectively.
- For some $S \subseteq V$ define $f(S)$ and $d(S)$ as: $f(S) = \max_{v \in S}(v_f)$, $d(S) = \min_{v \in S}(v_d)$. These are the minimum discover time and maximum finish time of a vertex in the set S.

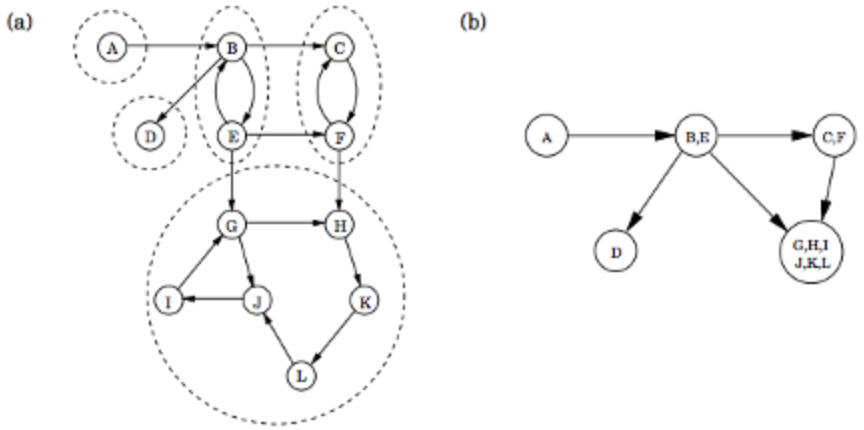

Figure 1: Showing going from G to G^C

2 Sequential Approach

The sequential algorithms for finding SCCs can be viewed as a solved problem, with much of the research being accepted as optimal. There are now formal proofs for many of the algorithms verifying there run-time and correctness [2]. Although there are many sequential algorithms that solve in $O(|V| + |E|)$ time complexity, we will focus on two of the most popular algorithms, Kosaraju's algorithm [4] and Tarjan's algorithm [8].

Kosaraju's algorithm has a simpler implementation than Tarjan's, but requires two passes of the graph while Tarjan's only requires a single pass.

2.1 Kosaraju's Algorithm

Kosaraju's Algorithm is one of the simplest ways to find the SCCs in a graph in linear time. It does this using the observation that the SCCs in G and G^T are identical. The algorithm requires two passes of the graph, the first to do a topological sort and the second to find the SCCs. In each DFS we will mark nodes that we have previously visited and mark them un-visited after searching all their outgoing edges.

Consider the following psuedo-code:

Algorithm 1 Kosaraju's Algorithm

```
1: procedure KOSARAJU(G, b)
2:     topologically sort G                          ▷ calculate v_d, v_f | ∀v ∈ V
3:     calculate G^T
4:     for v ∈ V by decreasing v_f do
5:         C ←DFS (G^T, v)                  ▷ start DFS at v, put all nodes visited in C
6:         add C as a SCC and remove from G^T
7:     end for
8:     return all SCCs found
9: end procedure
```

This algorithm is based upon the component graph G^C, as defined in the definitions. A property of the component graph is that it is a directed acyclic graph (DAG). Therefore if we traverse G^T by the decreasing finish time we will be always search only the deepest SCC in G^T, since G^C is a DAG. Thus by doing a graph search from v_f with the greatest finish time we will reach only vertices in the deepest SCC. By doing this in order of greatest v_f we will find all the SCCs in order. Here deepest refers to the SCC such that there are no outgoing edges to another SCC that has not been removed.

We will now prove the above statements formally. The proofs have been derived from the work in [4].

Lemma 1. *Let C and C' be distinct strongly connected components in directed graph G let $u, v \in C$ and $u', v' \in C'$. If G has a path $u \to u'$, then there cannot exist a path from $v' \to v$.*

Proof. By definition we know that C and C' are SCCs, therefore there are paths between all their own vertices respectively. Assume to reach a contradiction that we have a path from $v' \to v$. Thus if we have $u \to u'$ and $v' \to v$ we also have $u \to u' \to v' \to v \to u$. Thus we have created a bigger SCC from C and C', this contradicts the fact that C and C' are maximal. Therefore there must not be a path from $v' \to v$ if there is a path from $u \to u'$. □

This lemma implies that G^C is a DAG.

For the proof of the following lemmas refer to [4].

Lemma 2. *Let C and C' be arbitrary distinct SCC's in G. If $\exists$ $uv \in E$ such that $u \in C$ and $v \in C'$ then $f(C) > f(C')$*

Lemma 3. *Let C and C' be arbitrary distinct SCC's in G. Let $uv \in E^T$, edges of G^T, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.*

Theorem 4. *Kosaraju's Algorithm computes the strongly connected components of a input directed graph G.*

Proof. We will prove the correctness of Kosaraju's Algorithm by inducting on the number of trees found by the DFS of G^T on line 5 of the algorithm and that each tree's vertices make up those of a SCC within G. Let k be the number of trees found.

Base Case: $k = 0$.
Since there are no trees, the vertices in no trees trivially make up a SCC of size 0.

Induction Hypothesis:
Assume that for some k that the algorithm produces the first k trees of the DFS each contain the vertices of an SCC.

Inductive Conclusion:
We will now show that it holds for $k + 1$ trees of the DFS search return the vertices of each $k + 1$ SCCs.

Let us consider the $(k+1)$th tree and let the root of this tree be v. Also let C be the SCC that contains v. By the construction of our algorithm v_f must be the largest of the nodes not yet removed from the graph. Therefore since v_f is the largest we get $v_f = f(C) > f(C')$ by definition of f, where C' is any SCC in the graph yet to be removed. From the induction hypothesis we have that all vertices in C have yet to be visited by the algorithm. In addition since C is a SCC all the DFS of v should reach all vertices in C as there is a path between all vertices in C.

We also note that we are processing G^T, thus vertices incident to any outgoing edges of C must have already been visited in the algorithm. This is because if we consider some outgoing edge from C to some other SCC C' we know from Lemma 3 that $f(C) < f(C')$ therefore C' must have already been processed and the DFS from v will not leave C on an outgoing edge. Thus in any DFS from v will only reach nodes also within C and the vertices of the $(k+1)$th tree visited by the DFS search will correspond to the vertices of the $(k+1)$th SCC.

By the mathematical induction we have show that Kosaraju's Algorithm successful computes all the strongly connected components within an input directed graph G. □

The analysis of the running time of Kosaraju's Algorithm can be simply calculated as $O(|V| + |E|)$ because calculating G^T and topologically sorting are both $O(|V| + |E|)$. In addition, the loop on line 4 will run once per vertex again giving $O(|V|+|E|)$. Thus we have the total running time is $O(|V| + |E|)$.

Kosaraju's Algorithm is a great start to finding the SCCs in a directed graph, and highlights some of the interesting properties of DFS and the topological sorting of the graph. Although when considering how to run this algorithm in parallel there is no easy transition

due to the dependence on the state of the DFS and the progression of searching through nodes on G^T by finish time. Therefore there can be little to no gain for adding more processors.

2.2 Tarjan's Algorithm

Tarjan's Algorithm improves upon the main drawback of Kosaraju's Algorithm which is the requirement of multiple passes of the graph. Instead Tarjan's Algorithm can compute SCCs with a single DFS pass while maintain state upon a separate stack. Tarjan's Algorithm is the most popular sequential algorithm for finding SCCs due to it's increased speed and relatively straight forward implementation.

The strengths of Tarjan's algorithm are also echoed by Turing Award winner Donald Knuth in his article "Twenty Questions for Donald Knuth" upon release alongside his book the Art of Computer Programming. Where Knuth says "my favorite is the implementation of Tarjan's beautiful algorithm for strong components." [9]. Followed by him commenting that "the data structures that [Tarjan] devised for [the] problem fit together in an amazingly beautiful way, so that the quantities you need to look at while exploring a directed graph are always magically at your fingertips. And his algorithm also does topological sorting as a byproduct" [9].

Further outlining how Tarjan's Algorithm is one of the optimal algorithm for finding SCCs sequentially within a directed graph.

We will now consider the pseudo-code of Tarjan's Algorithm:

Algorithm 2 Tarjan's Algorithm

```
1: i ← 0
2: create new stack
3: procedure DFS(v)
4:     v_low ← i
5:     v_d ← i
6:     i ← i + 1
7:     place v on stack
8:     for w ∈ δ(v)^out do                          ▷ δ(v)^out is outgoing edges of v
9:         if w_d is undefined then                 ▷ w not yet visited
10:            DFS(w)
11:            v_low = min(v_low, w_low)
12:        else if w_d < v_d then                   ▷ w
13:        end if
14:    end for
15:    if v_low = v_d then                          ▷ v is the root of the component
16:        create new SCC
17:        while u = Top(stack) s.t u_d ≥ v_d do    ▷ We have the answer if r is 0
18:            remove u from the stack
19:            add u to new SCC
20:        end while
21:    end if
22: end procedure
23: while v ∈ V where v_d is undefined do
24:    DFS(v)                                       ▷ Apply method on any unvisited vertices
25: end while
```

Now let us explain the definition used in the above algorithm. These have been derived from [8].

- Let v_d be the discover time of a vertex and v_{low} be the the be the smallest v_d reached by traversing zero or more edges following not in the DFS tree (i.e edges to previously visited vertices).
- Let $v \rightarrow^* w$ denote a path from v to w.
- Let $v \rightarrow^- w$ denote an edge from v to w where w is in the DFS search tree, i.e previously visited.

The algorithm formally defines the low property of a vertex as:

$$v_{low} = \min(\{v_d\} \cup \{w | v \rightarrow^* \rightarrow^- w \text{ and } \exists\ u \text{ such that } (u \rightarrow^* v \text{ and } u \rightarrow^* w \text{ and } u, w \text{ are both in the same SCC})\} \quad (1)$$

Tarjan goes on to prove the following theorem which is used to prove the correctness of the algorithm [8].

Theorem 5. *Let G be a directed graph. Based upon the above definition for v_{low}, a vertex v is the root of SCC if and only if $v_d = v_{low}$.*

Here root is defined as the vertex that separates the strongly connected component from the rest of the graph. It can also been seen from the definition of v_{low} that if $v_d = v_{low}$ then v must have the smallest v_d out of all the vertices within it's SCC. Then it can be inductively argued that by considering the vertices it reaches in a DFS, that were not previously put into SCC, are within it's own SCC.

Therefore on line 15 of the algorithm we can consider all the vertices added to the stack in the DFS and can construct the SCC by pulling them off the stack. In addition, we will not need to worry about vertices in an SCC connected to the SCC of v by an outgoing edge as they will be previously popped off the stack and added to their respective SCC.

The full proof of correctness can be found in Tarjan's original paper. In summary the paper is very well written and I found it highlight the strength true of and expand upon the usefulness of depth first search when designing sequential graph algorithms. Reading his paper can provide a deeper understanding of the power of DFS beyond simply searching. [8].

For the run time of the Tarjan's algorithm we can see that it is also $O(|V| + |E|)$, but now only requires one pass of the graph. Therefore Tarjan's algorithm will run much faster than Kosaraju's Algorithm.

But again as in we saw in Kosaraju's Algorithm, Tarjan's Algorithm is a purely sequential algorithm. It is highly dependant on the DFS and the current state of the stack. This again makes it hard to run in parallel as the work of the algorithm is hard to subdivide.

3 Parallel Approach

Due to the previous algorithms sequential nature of their approaches to solving the SCCs problem makes it hard to gain from distributed computing. Therefore a new paradigm of algorithms have been devised, these algorithms are commonly referred to as reachability based algorithms. The approach of these algorithms is to divide up the problem into solvable sub problems allowing for multiple processors to be taken advantage of. Although they do not hope to reduce the total work of finding the SCCs, rather trade of doing more work but being able to run concurrently [7]. The trade off of time and work of these algorithms is a key consideration when comparing them to their sequential counter parts.

A reachability based SCC algorithm is based on the idea that all the vertices that are both reachable from v and can reach v form a SCC by definition. Therefore by doing two reachability queries through the graph, the first following edges normally and the second following edges in reverse, we can take their intersection of vertices to find one SCC in the graph. By repeating this process all the SCCs in the graph can be found. In addition to finding the SCCs, the sets created through the reachability queries can be used to sub-divide the problem, a useful property for divide and conquer algorithms.

Through this approach reachability based approach Coppersmith et. al have shown that with a random pivot selection that this algorithm runs in $O(m \log n)$ time when ran in a sequentially fashion [3]. The random pivot selection analysis is very similar to that done in the quick-sort sorting algorithm.

In general parallel algorithms and divide and conquer go hand in hand but if we consider this reachability based approach the subsets created after finding one SCC are not always optimally balanced. One example given by Guy E. Blelloch et. al is a spare graph [1]. In this case a limited number of vertices would be found in each iteration, resulting in unbalanced partitions which will produce a recursive depth of $O(|V|)$.

To combat the depth of recursion Warren Schudy developed an algorithm to better partition of the subsets and limiting the number of reachability queries is $O(\log^2 |V|)$. Although by doing so he reduces the speed of the algorithm from the original by a factor log factor.

More recently in 2018 Guy E. Blelloch et. al have devised an process of analyzing sequential randomized incremental algorithms and showing they can be run in parallel [1]. We will use their definitions and analysis of a reachability algorithm to show that the following reachability based sequential algorithm can be run in parallel efficiently.

Psuedocode from [1].

Algorithm 3 Reachability based SCC

1: $V \leftarrow$ vertices of the graph in random order
2: $S_{SCC} \leftarrow \emptyset$
3: **for** $i \leftarrow 1$ to n **do**
4: $\quad S \in V$ ▷ V sub graph containing v_i
5: $\quad$ **if** $S \neq \emptyset$ **then**
6: $\quad\quad R^+ \leftarrow$Forward-Reachability$(S, v_i)$
7: $\quad\quad R^- \leftarrow$Backward-Reachability$(S, v_i)$
8: $\quad\quad V_{SCC} \leftarrow R^+ \cap R^-$
9: $\quad\quad V \leftarrow V \setminus S \cup \{R^+ \setminus V_{SCC}, R^- \setminus V_{SCC}, S \setminus (R^+ \cup R^-)\}$
10: $\quad\quad S_{SCC} \leftarrow S_{SCC} \cup V_{SCC}$
11: $\quad$ **end if**
12: **end for**

This algorithm can be transformed into a parallel algorithm randomized (Las-Vegas) parallel algorithm to solved the SCC problem. A Las-Vegas randomized algorithm means that the algorithm will always produce the correct answer and our running time will be bounded with high probability.

In order to transform this sequential incremental algorithm we will use the work done by Guy E. Blelloch et. al [1].

The main idea is to run batches of the iterations of the sequential algorithm in parallel and show that when randomized they have low dependency on one another. With regards to the SCC problem this means that we will run the *for loop* as a *parallel for loop* for batches of vertices. By doing so Guy E. Blelloch et. al shows that the overlapping work will not effect the correctness of the algorithm by assigning a priority to each parallel processor.

First in their description Guy E. Blelloch et. al divide up the problem into a set of $O(\log n)$ rounds. Within these rounds they run parallel versions of the sequential algorithm, i.e computing the reachability for each v_i within the batch. While doing this across processors the reachability is stored on the vertices, by the priority of the vertex running the reachability queries. This means that if two vertices v_a, v_b in the random ordering are both running in a parallel batch and they both reach a vertex v_i, then v_i will store only the reachability of v_a as $v_a \leq v_b$ within the random ordering (lower having a higher priority as it was earlier in the ordering).

Since if v_a and v_b's reachability queries reach v_i, they will be in the same SCC anyways and thus the collision is resolved by using the order of the vertices.

Through this dependency ordering argument Guy E. Blelloch et. al are able to prove the following theorem.

Theorem 6. *For a random order of the vertices in a graph, when running the random incremental SCC algorithm in parallel it does* $O(W_R(|V|, |E|) \log |V|)$ *expected work and has* $O(D_R(|V|, |E|) \log |V|)$ *depth on a priority concurrent read, concurrent write PRAM (CRCW PRAM) model .*

Here W_R refers to the work needed to be done by a reachability query and D_R is the depth of the reachability queries. We leave these as a black box algorithm as there are many different algorithms for performing reachability such as a using a breath first search.

We also note that a CRCW PRAM model is used by considering the priorities we set on the vertices. Therefore in our example above of two vertices reaching the same vertex in the same step, the vertex with priority will be write able to write to the vertex and the other will be ignored. This alleviates synchronization issues with running the queries from the verities in parallel.

Therefore by consider a simple implementation for the execution of the reachability queries such as BFS we get $W_R(|V|, |E|) = O(|E|)$ and $D_R(|V|, |E|) = O(|E| \log |V|)$ as well. Therefore the total work of the parallel algorithm would be $O(|E| \log |V|)$ and the depth to be . If we can run this work optimally across P processors we can approximate the running time as $O(|E| \log |V|/P + D_R(|V|, |E|)) = O(|E| \log |V|/P + |E| \log |V|))$, when using a breath-first search. [1].

Here we see that if the depth of the graph is high the resulting depth of the reachability queries is also increased, and the use of parallel processors deteriorates. Although for graphs with a low diameter using a breath first search will work find as it can have a tighter bound than $|E|$ for the depth of the graph.

4 Conclusions

We have looked at both the sequential and parallel techniques for finding strongly connected components withing a graph. The sequential approaches provide a fast $O(|V| + |E|)$ running time, while the parallel approach sacrifices total work and uses an ordering argument to sub-divide the problem.

After analyzing the different approaches it can be seen that their is no optimal choice for the algorithm and instead it depends on the types of input graphs to be analyzed. For instance, for graphs that are not massive the sequential linear approach is fast enough for most applications. In addition, if one simply wishes to determine if there are a few small SCCs or no SCCs in a graph it would be clearly superior. This is because in the case of only a few SCCs SCCs the amount of parallelism available is small, as most vertices would be in

the trivial SCC of just them self. This would result in all the reachability queries only ever returning one vertex. While on the other hand for the sequential approach the solution can be determined in $O(|V| + |E|)$ in a single traversal through Tarjan's algorithm. Thus when there are none or few small SCCs the sequential approach is cleary better.

Now if we consider the opposite case of there being large SCCs within the graph the parallel algorithm will able to take advantage of this sub-division and find large SCC within one step of the algorithm (i.e doing reachability queries for one vertex).

Relating back to the problem of finding SCCs in circuit designs for EDA software the sequential approach is still the best way to go. Most digital circuit designs today are made of synchronous logic (using registers between sections of logic). This leads to there being very few un-synchronized (not register separated) cycles within a circuit design. This is the case as having cycles in a circuit creates asynchronous logic making timing analysis much more difficult and the circuit more unpredictable. As cyclic logic is unfavourable in digital circuit designs the detection of SCCs is motivated by ensuring none exists in the design, and highlighting any SCCs found to the designer to fix. Therefore as described previously a sequential algorithm such as Tarjan's will work best in this case, as SCCs are rare within the input graphs. Therefore in EDA software a sequential SCC algorithm approach is much preferred due to the properties of the possible input graphs that digital circuits, as they contain few SCCs and are manageable size graphs.

References

[1] Guy E. Blelloch et al. *Parallelism in Randomized Incremental Algorithms*. 2018. arXiv: 1810.05303 [cs.DS].

[2] Ran Chen et al. "Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq and Isabelle". In: 2019.

[3] Don Coppersmith et al. "A divide-and-conquer algorithm for identifying strongly connected components". eng. In: (2003).

[4] T. H. et al Cormen. *Introduction to algorithms*. MIT Press, 2003.

[5] Lap Chi Lau. "CS 466 Coures Notes". In: (2020).

[6] Sina Majidian, Mohammad Hossein Kahaei, and Dick de Ridder. "Hap10: reconstructing accurate and long polyploid haplotypes using linked reads". eng. In: *BMC bioinformatics* 21.1 (June 2020). PMC7302376[pmcid], pp. 253–253. ISSN: 1471-2105. DOI: 10.1186/s12859-020-03584-5. URL: https://doi.org/10.1186/s12859-020-03584-5.

[7] Thomas H. Spencer. "Time-Work Tradeoffs for Parallel Algorithms". In: *J. ACM* 44.5 (Sept. 1997), pp. 742–778. ISSN: 0004-5411. DOI: 10.1145/265910.265923. URL: https://doi-org.proxy.lib.uwaterloo.ca/10.1145/265910.265923.

[8] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[9] *Twenty Questions for Donald Knuth*. https://www.informit.com/articles/article.aspx?p=2213858&WT.mc_id=Author_Knuth_20Questions. Accessed: 2020-08-10.