An intuitive algorithm for finding strongly connected components

Sjoerd J. Henstra

Abstract

There are already several efficient algorithms for finding strongly connected components in directed graphs. These are usually based on a depth-first search, making them difficult to parallelize as depth-first searching is inherently sequential. In this paper I describe an intuitive, easily parallelizable algorithm for finding strongly connected components and compare the performance of several variations of the algorithm.

1 Introduction

There are several efficient algorithms for finding strongly connected components. Two well known examples are the Kosaraju-Sharir algorithm [Sha81] and Tarjan's algorithm [Tar72]. Both algorithms make use of a depth-first traversal of the graph. This complicates parallelization of the algorithms as depth-first tree traversals are inherently sequential. Others have used a divide-and-conquer approach to create slightly less efficient algorithms that are more easily parallelizable. [LFP00] The basis for this paper is a seminar in which we set out to formulate an easily parallelizable algorithm for finding strongly connected components and then finding a way to derive an existing, efficient algorithm from it.

while *not done* **do** :
    pick edge(u, v)
    if$(v \rightarrow^* u)$:
        merge(u, v)
    else :
        propagate data(u, v)
    fi
od

Given this base, there are several decision that can significantly impact the algorithm's performance, namely:

- How do we pick edges?
- How do we determine that there is a path from u to v?
- What data do we propagate and in which direction?

2 Identifying cycles

The first proposed idea was to propagate a single mark forward along a single path. If there is an edge (u, v) at some point in that path where node v has already been marked, there is apparently a path from v to u and the two nodes are part of the same strongly connected component. However, this is not enough to fully identify all strongly connected components. Since we can only follow a single path, this method can only visit a single source node and a single sink node. Visiting multiple disjoint subgraphs is also impossible.

Thus it seems we need to follow multiple paths and use multiple distinct marks which are propagated along all edges. Rather than somehow deciding which nodes need an initial mark, we choose to give all nodes a unique initial mark.

If, given an edge (u, v), we propagate all markings from u to v, the set of marks on a node u identifies all nodes from which we have found a path to u. We call this the ancestor set of u and use the notation u^-. If, given an edge (u, v), v is part of u^- there is apparently some path from v to u, meaning u and v are part of the same strongly connected component. Because marks are no longer ambiguous we can visit edges in any order we like and as often as we like. To avoid unnecessary edge visits we mark an edge (u, v) as visited after adding u^- to v^-. We should not pick this edge again unless at some point u^- changes. To do this, when we visit some edge (u, v) and v^- changes, we removed any visited marks from all edges (v, w).

Analogous to ancestor set, nodes also have descendant sets, written as u^+. Each node can have a separate set of marks which are propagated backward. This set then identifies all nodes to which we have found a path from a given node. If, given an edge (u, v), u is part of v^+ there is a path from v to u, so once again we know u and v are part of the same strongly connected component.

Using both ancestor sets and descendant sets allows more information to be passed whenever an edge is picked. This option was considered, but it turned out to be less than optimal. Because information is propagated both forward and backward along an edge (u, v), both edges leading to u and edges leading from v need to be marked as unvisited. This makes it much more difficult to come up

with an efficient edge picking method and the number of iterations of the main loop can become much larger. For this reason, only using the ancestor set and only propagating information forward is preferable.

Whenever two nodes u and v are identified as belonging to the same strongly connected component they are immediately merged. Merging nodes not only identifies two nodes as belonging together, but decreases the number of edges and increases the potential information propagated by each edge. In many cases it is possible to identify all strongly connected components without visiting all edges in the graph. Merging nodes u and v is done by moving edges connected to node u to node v instead. So any edge (w, u) is changed into (w, v) unless $v = w$ or (w, v) already exists. Likewise any edge (u, w) is changed into (v, w) unless $v = w$ or (v, w) already exists. Any edges that are not moved can be deleted. Edges that are moved are marked as not visited.

There are also costs involved in merging nodes as edges may need to be moved or deleted. The precise cost of these operations depends on the data structures used to represent the graph. In addition to minimizing the number of edge moves and deletions it may be worth reducing the cost of these operations by using certain data structures.

3 Picking edges

There are three important considerations for edge picking methods. This choice can significantly impact the required number of iterations of the main loop. Additionally the performance of the method itself matters as it is performed in each iteration of the main loop. The latter depends largely on the data structure used for the graph. Finally this choice may result in more or fewer edges being moved. The cost of this operation also depends on the data structures used.

As such, the focus here lies on the number of edges visited and moved on several types of graphs using the various methods. These metrics can be reliably measured and compared. The precise implications of these metrics depends on the cost of each of these operations, but all three should obviously be kept as low as possible.

We use the small graph from Figure 1 as an example to illustrate the way edge picking methods work.

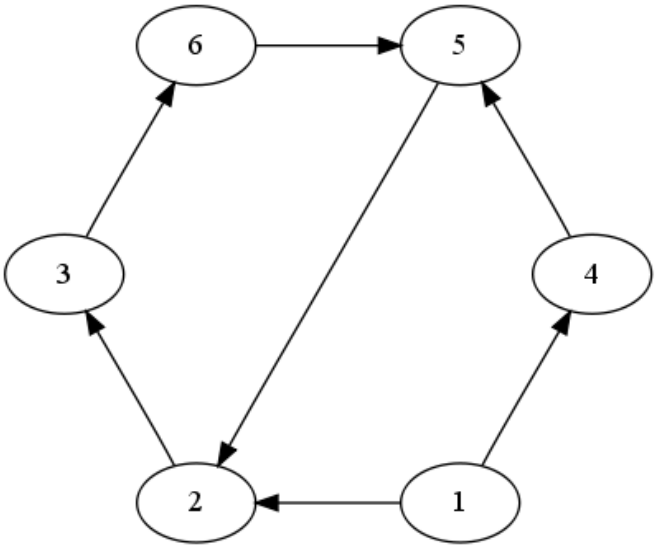

Figure 1: sample graph

3.1 Random edges

Picking edges at random is our control method. One does not expect this method to perform especially well, so other methods should be considered to be interesting if they perform significantly better than this random method for some types of graphs.

3.2 Follow a path

Another fairly simple method involves following some path through the graph for as long as possible. After visiting some edge (u, v), one may try to follow an edge (v, w), followed some edge leading from w to yet another node. A potential advantage associated with this method is that as long as we follow a path, we increase the amount of information that is passed forward during each edge visit. This method can be implemented very efficiently for some graph implementations as we always have some node from which we would prefer to follow an edge. If no such edge is available we may fall back to any other selection criterium. For testing purposes we used two different fallback methods. One picks the first available edge in numerical order and the other picks a random edge.

When this method is applied to the sample graph using the numerical fallback method, our algorithm performs the following steps:

- visit $(1, 2)$
- visit $(2, 3)$
- visit $(3, 6)$
- visit $(6, 5)$
- visit $(5, 2)$
- merge 5 and 2
- visit $(1, 4)$
- visit $(4, 2)$
- visit $(2, 3)$
- merge 2 and 3
- visit $(1, 3)$
- visit $(4, 3)$
- visit $(6, 3)$
- merge 6 and 3

A possible extension of this method uses a stack of visited nodes to create a pattern of visited edges resembling a depth-first search.

3.3 Minimal in-neighborhood

Our final edge picking method makes use of the number of unvisited incoming edges each node has. The fewer such edges a node has, the more complete the current information in the node is likely to be. If a node has no incoming edges whatsoever we can safely visit all outgoing edges, knowing that those edges will never have to be visited again. Other than that particular case, there is no guarantee that an edge selected this way will not be visited again repeatedly. For each unvisited edge (u, v) we count the number of unvisited edges (w, u). All edges for which this count is minimal are candidate edges. One of those edges may be picked A naive implementation of this method is likely to be costly, but it need not be if

the number of unvisited incoming edges to each node is tracked in a separate data structure.

When this method is applied to the sample graph, one possible run of our algorithm would perform the following steps:

- visit $(1, 2)$
- visit $(1, 4)$
- visit $(4, 5)$
- visit $(2, 3)$
- visit $(3, 6)$
- visit $(6, 5)$
- visit $(5, 2)$
- merge 5 and 2
- visit $(4, 2)$
- visit $(6, 2)$
- merge 6 and 2
- visit $(3, 2)$
- merge 3 and 2

4 Results

To test the different methods we run the algorithm on some randomly generated graphs with the following characteristics:

- A graph with many large strongly connected components which are connected to each other with many edges.
- A graph with many large strongly connected components which are connected to each other with very few edges.
- A graph with many small strongly conencted components.
- A graph consisting of a single large strongly connected component.

For each of these graphs we will compare the number of edge visits and edge moves performed using the different methods. Methods involving some randomness are run three times. Tarjan's algorithm is also included for comparison. It does not create or delete edges and visits each edge exactly once.

We will also run the algorithm on a large number of randomly generated graphs with a fixed number of nodes and an increasing number of edges and plot the number of required operations for three edge picking methods: random picking, following a random path and selecting minimal in-neighborhoods.

4.1 Test 1

For this test we use a graph consisting of 20 strongly connected components, each made up of 50 nodes. These strongly connected components are connected to each other with up to 20 nodes per pair. These edges are chosen to ensure the graph is weakly connected and do not form additional cycles.

This graph contains 1000 nodes and 2708 edges.

Edge picking method	Edges visited	Edges created	Edges deleted
Random 1	7541	15552	18186
Random 2	7520	16999	19633
Random 3	7465	16662	19296
Follow path	26819	17389	20023
Follow random path 1	7557	17338	19972
Follow random path 2	7382	17006	19640
Follow random path 3	7644	16222	18856
Minimal in-neighborhood	11863	5829	8463
Tarjan	2708	0	0

4.2 Test 2

For this test we use a graph consisting of 20 strongly connected components, each made up of 50 nodes. These strongly connected components are connected to each other with at most one edge per pair. These edges are chosen to ensure the graph is weakly connected and do not form additional cycles.

This graph contains 1000 nodes and 2039 edges.

Edge picking method	Edges visited	Edges created	Edges deleted
Random 1	4802	9816	11782
Random 2	4879	10167	12133
Random 3	4912	10151	12117
Follow path	6410	10937	12903
Follow random path 1	4905	10242	12208
Follow random path 2	5062	8914	10880
Follow random path 3	5034	9842	11808
Minimal in-neighborhood	5323	3307	5273
Tarjan	2039	0	0

4.3 Test 3

For this test we use a graph consisting of 100 strongly connected components, each made up of 1 to 5 nodes. These strongly connected components are connected to each other with at most one edge per pair. These edges are chosen to ensure the graph is weakly connected and do not form additional cycles.

This graph contains 303 nodes and 1197 edges.

Edge picking method	Edges visited	Edges created	Edges deleted
Random 1	4255	1378	1780
Random 2	4456	1284	1686
Random 3	4160	1390	1792
Follow path	37256	1157	1559
Follow random path 1	4228	1328	1730
Follow random path 2	4205	1341	1743
Follow random path 3	4530	1324	1726
Minimal in-neighborhood	8474	1243	1645
Tarjan	1197	0	0

4.4 Test 4

For this test we use a graph consisting of a single strongly connected component made up of 100 nodes.

This graph contains 100 nodes and 198 edges.

Edge picking method	Edges visited	Edges created	Edges deleted
Random 1	544	1750	1948
Random 2	551	2007	2205
Random 3	552	1085	1283
Follow path	198	2087	2285
Follow random path 1	569	1958	2156
Follow random path 2	545	1375	1573
Follow random path 3	537	1980	2178
Minimal in-neighborhood	581	374	572
Tarjan	198	0	0

4.5 Comparison

Tarjan's algorithm clearly performs better than the various versions of our intuitive algorithm. It visits each edge only once and, more importantly, does not have to create and delete edges.

Picking random edges is our basis for comparison. The number of edge visits with this method is several times the number of edges in the graph. The number of edge creations and deletions varies strongly depending on the type of graph. Graphs with large strongly connected components result in a large number of creations and deletions, while small graphs with small strongly connected components require few such operations.

The performance with the path following method depends largely on the fallback method that is used. With random picking as our fallback method, the algorithm performs similarly to the completely random picking version. It seems that trying to follow a path through the graph is not useful.

When the fallback method is picking the first available edge in numerical order, the algorithm actually performs significantly worse on most graphs. However, when the entire graph forms a single strongly connected component, the number of edges visited is equal to the number of edges in the graph, just as with Tarjan's algorithm. Even then, the number of edge creations and deletions is much larger and the algorithm cannot be said to be particularly efficient.

Selecting nodes based on the size of their in-neighborhood tends to increase the number of edge visits, more so for graphs in which many edges exist which are not part of a cycle. In return, the number of edge creations and deletions is significantly reduced for graphs with large strongly connected components.

4.6 Performance trends

To see how each method's performance changes as graphs become more dense, we create a large number of random graphs with 100 nodes and a varying number of random edges. We start at 200 edges and go up to 1000 edges in 50 edge increments. At each interval we generate 100 different random graphs with 100 nodes and the given number of edges. These graphs are generated by creating edges completely at random. Unlike previous tests graphs, these graphs have no predictable structure.

Figures 2, 3 and 4 show the progression of the average number of edge visits, moves and deletions respectively for increasingly dense graphs for three picking methods. The three picking methods used are: following randomly selected paths, picking edges at random and picking edges whose source node has a minimal in-neighborhood.

Figure 2 shows that for these completely randomly generated graphs, following random paths results in a smaller number of edge visits in graphs with more edges. As all edges need to be visited at least once or be deleted before the algorithm finishes, this picking method apparently leads to more edges being deleted before they can be visited in dense graphs.

The number of edge visits using random picking remains roughly the same as the number of edges increases. Apparently this method is also able to delete more unvisited edges in denser graphs, but the number of edge visits does not decrease as the number of edges increases like with the previous method.

When picking edges based on in-neighborhoods, performance doesn't improve much for denser graphs.

Figure 3 shows that in terms of the number of edges that are moved (i.e. have their source or sink node changed), completely random edge picking is the best method and following random paths the worst. However, as graphs become more and dense, the difference in edge moves between following random paths and edge picking based on in-neighborhoods shrinks.

Figure **??** shows the average number of edge deletions for graphs of varying density if very similar for random picking and picking based on in-neighborhoods. Following random paths results in more edge deletions, though the difference with the other two methods doesn't seem to change much for denser graphs.

It is interesting to compare this graph with Figure 2 as all edges in a graph need to be deleted or visited at least once during the course of the algorithm. Following random paths leads to more edge deletions and fewer edge visits in denser graphs, while picking edges at random leads to a steady number of edge visits and a smaller and more steadily growing number of edge deletions in graphs of increasing density.

Which of these two is the better performer then depends on the cost of edge

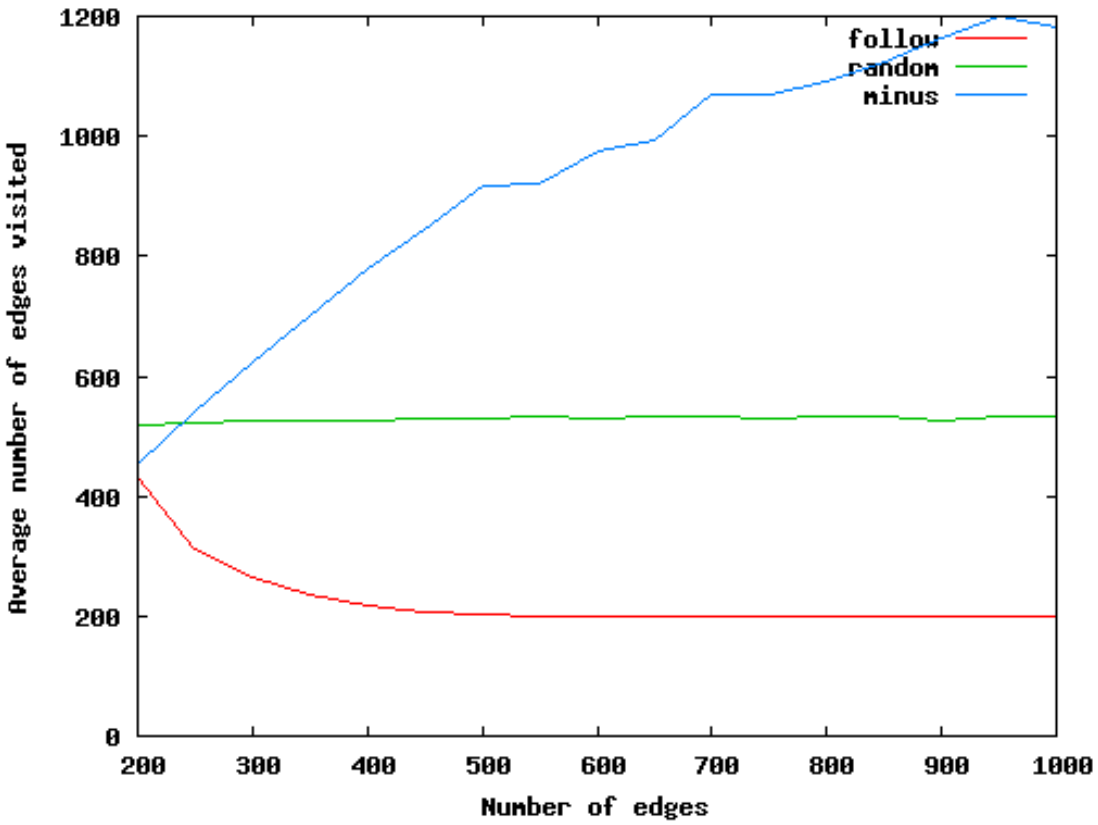

Figure 2: Average number of edge visits for random graphs with 100 nodes

deletion and the cost of the picking functions themselves. If deleting an edge is a costly operation, picking random edges is preferred. On the other hand, if edge deletions and moves are cheap compared to other operations, following random paths may be more efficient.

Edge picking based on the size of in-neighborhoods results in consistently worse performance than random edge picking. The large number of edge visits is particularly bad if we don't have an efficient way to pick edges based on their in-neighborhoods. Even if we do have an efficient way to do this, the large number of edge visits also means an even large number of in-neighborhood updates.

Note that all these results are based on completely randomly generated graphs without any predictable structure. In cases where something is known in advance about the structure of the graph, for example that the graph contains many small strongly connected components results from the earlier tests may be more applicable.

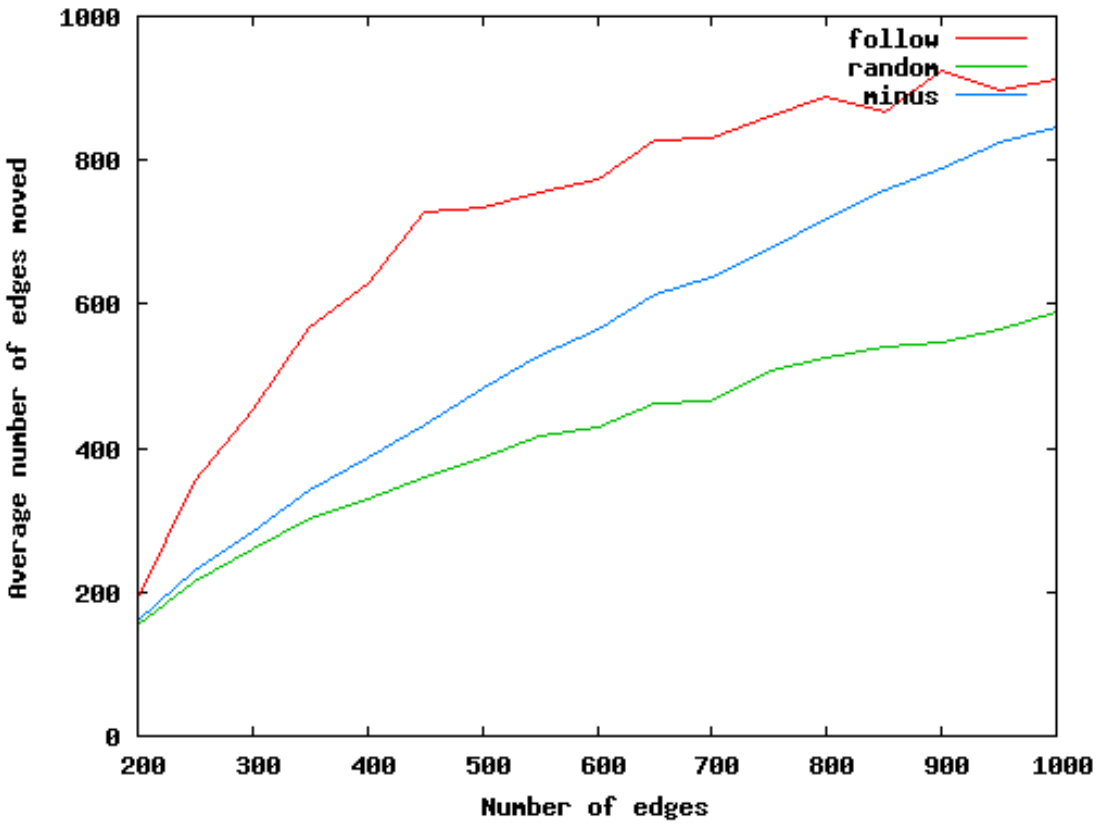

Figure 3: Average number of edge moves for random graphs with 100 nodes

5 Conclusions

We have formulated a very simple algorithm that identifies strongly connected components in a directed graph and which should be easier to parallelize than algorithms based on depth-first searches. We have investigated a few edge picking possibilities and observed that while selecting edges at random or following random paths is likely to provide the best performance in general, different edge picking methods can lead to very good performance on some graphs and very bad performance on others. Additionally, the algorithm's performance determines on the cost of visited, moving and deleting edges. The cost of these operations depends on the choice of data structures and edge picking method, which could be an interesting topic for further research. Other possibilities for future work include deriving a more efficient algorithm without explicitly performing a depth-first search and parallelizing the algorithm.

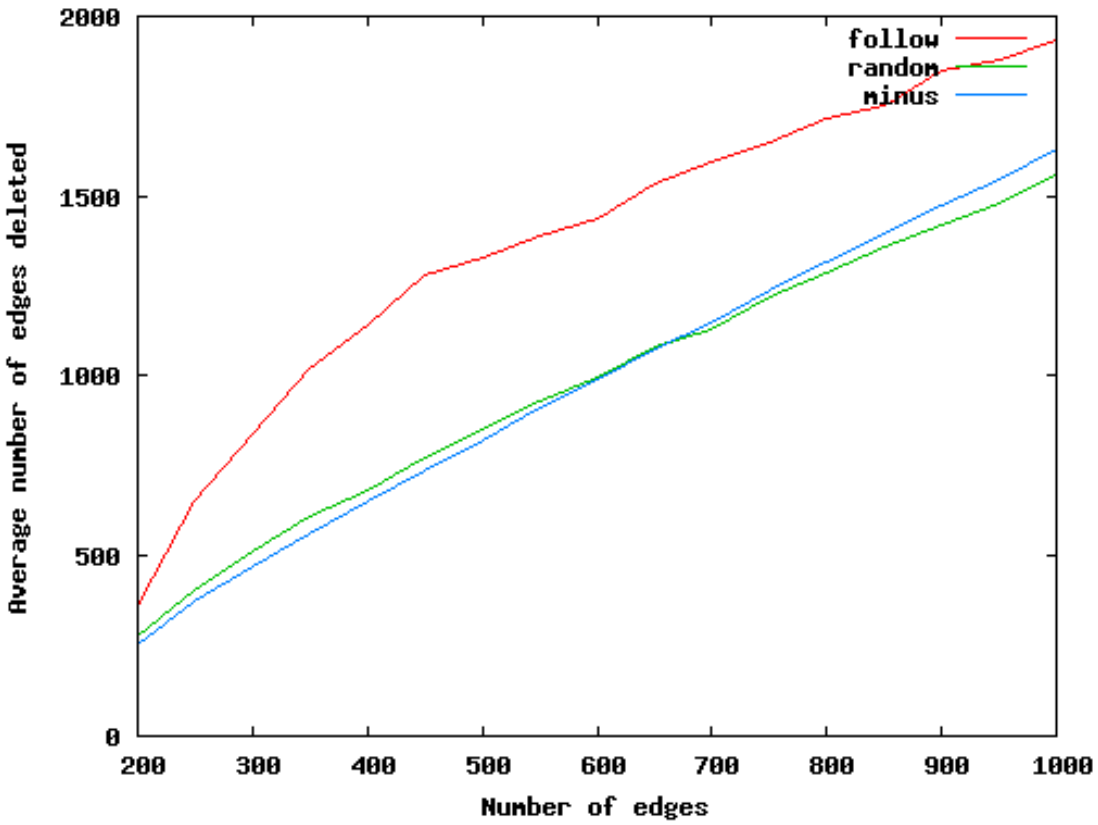

Figure 4: Average number of edge deletions for random graphs with 100 nodes

References

[LFP00] B. Hendrickson L. Fleischer and A. Pinar. On identifying strongly connected components in parallel. *Proc. Irregular '2000, Lecture Notes in Computer Science*, 1800:505–512, 2000.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–71, 1981.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.