Towards an Automatic Derivation of Tarjan's Algorithm for Detecting Strongly Connected Components in Directed Graphs

H.L.A. van der Spek

October 30, 2006

Abstract

Ideally, algorithms should be easy to understand and perform efficiently. However, these two requirements are often contradicting. In this thesis, by describing a semi-automatic derivation of an efficient algorithm for detecting strongly connected components, we argue that efficiency may be derivable, thereby satisfying both requirements. First, some basic graph theory will be reviewed. Then we will focus on some existing algorithms, among which Tarjan's algorithm is the most well known. Some attention is given to parallel algorithms for detecting strongly connected components. Next, we start with a simple but inefficient algorithm for detecting strongly connected components. This algorithm will be transformed step-by-step into a more efficient algorithm. Finally, we will present some test results and compare the efficiency of the resulting algorithm to Tarjan's algorithm.

Acknowledgements

I would like to thank all people that have contributed in any manner to this thesis. Special thanks go to Harry A.G Wijshoff and Erwin M. Bakker. We have spent many hours discussing this work and they were a great support when I was about to give up. I would like to thank all participants of the spring 2006 seminar "data structures revisited". This is where my initial interest in the problem of detecting strongly components started. Many of the discussions there have been helpful in understanding the problem. I also would like to thank my family. I have been able to work at home without worrying about anything else. There is no place like home.

Contents

Chapter 1

Introduction

As of today, many pieces of complex softwarehave been produced. Often, much effort is put into writing efficient algorithms. It is not uncommon that simplicity and readability are sacrificed in order to increase performance. Although this may be needed sometimes, this situation is far from ideal. Hand-optimized algorithms are generally hard to understand and all clever tricks employed obscure the fundamental workings of the algorithm. Ideally, software should be written such that the problem is unambiguously defined and the real implementation should be generated by a compiler. Such a compiler should be able to translate concise, formal descriptions into efficient implementations. Advantages are that the source code remains small, simple and correct.

For many problems, efficient algorithms have been written. In [13], R. Tarjan calls for the need of a language that is both easy to understand and efficient to execute. In order to achieve this, a compiler must be able to derive alternatives that are more complex, but yield lower execution times. This process involves generating alternative code and selection of appropriate data structures. In [7], R. Paige writes about their achievements in transformational methodology. He also mentions the use of simple rules to derive faster implementations. The

main contribution he and his group made is about finite differencing applied to computable expressions, which will be treated shortly.

In this thesis, the main focus will be on derivation of an efficient algorithm for detecting strongly connected components from a high-level, intuitive specification. In order to get a better understanding of the problem domain, some basic graph theory is given. Two existing algorithms for identifying strongly connected components are reviewed in order to get a tighter grip on the subject. Chapter 4 shows and explains the derivation of an algorithm that computes the strongly connected components of a graph from a high-level specification. Finally, the entire derivation chain is reviewed and some questions for future research are posed.

Chapter 2

Preliminaries

In order to specify algorithms, some notation is needed. In this chapter some basics of graph theory are presented. A lot of material covered here is not strictly necessary to understand the later chapters. It is however practical to introduce it, as many transformational steps can be linked to graph theoretic concepts. A compiler might not need this knowledge, but for the human reader it can be a great aid in understanding the structure of the code generated by a transformation. Another section treats the concept of finite differencing.

2.1 Graph Theory

A set of nodes V together with a set of edges E is called a graph. Edges can either be directed or undirected. If the edges of a graph are undirected, a graph is called an undirected graph, otherwise it is referred to as a directed graph, often abbreviated as *digraph*, Instead of nodes, the term vertices is often used. Another term used for directed edges is *arcs*. Figure 2.1 shows an example of a digraph.

Definition 2.1 *An undirected graph is a graph with a set nodes V and a set of*

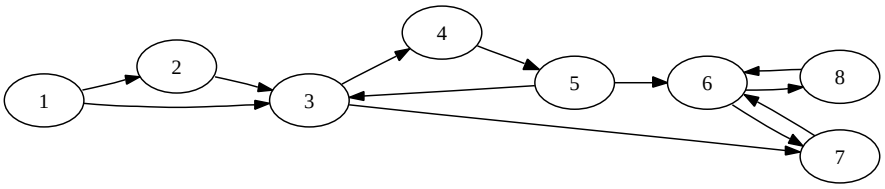

Figure 2.1: Example digraph

undirected edges $E \subseteq \{\{a, b\} \mid a, b \in V \wedge a \neq b\}$.

Definition 2.2 *A digraph G is a graph with a set nodes V and a set directed edges $E \subseteq V \times V$.*

Note that any undirected graph can be represented by a digraph, just by replacing any undirected edge that connects two nodes a and b by two directed edges (a, b) and (b, a). From this point on, when a graph is mentioned, it always will be a digraph, as all algorithms in this thesis operate on digraphs.

Consider a graph $G = (V, E)$. If there exists a sequence of edges

$$((v_0, v_1), (v_1, v_2), \ldots, (v_{k_1}, v_k),$$

where each edge is in E, then there exists a path from v_0 to v_k which is denoted by $v_0 \xrightarrow{*} v_k$. $v \xrightarrow{*} v$ always holds, as v can be reached from v by a path of length zero. If two nodes v_0 and v_k are connected via a path with length greater than zero, $v_0 \xrightarrow{+} v_k$ holds as well. More formally:

Definition 2.3 $v_0 \xrightarrow{*} v_k$ *iff there exists a path from v_0 to v_k.*

Definition 2.4 $v_0 \xrightarrow{+} v_k$ *iff there exists a path from v_0 to v_k with length greater than zero.*

Example In Figure 2.1, the sequence $((1, 2), (2, 3), (3, 7))$ defines a path. Therefore, $1 \xrightarrow{*} 7$ holds. This path has length 3, therefore $1 \xrightarrow{+} 7$ holds as well.

Definition 2.5 *A cycle is a path with length greater than zero where the first node and the last node are the same.*

Definition 2.6 *$v \xrightarrow{+} v$ iff there is a cycle containing v.*

Example In Figure 2.1, the sequence $((3,4),(4,5),(5,3))$ ($3 \xrightarrow{+} 3$ holds) defines a path of length greater than zero. The starting point and end point are identical, and therefore this sequence is a cycle.

Definition 2.7 *Let $G = (V, E)$ be a graph. The transitive closure of this graph G is a graph $G^+ = (V, E^+)$ such that E^+ contains an edge (a, b) if $a \xrightarrow{+} b$.*

Definition 2.8 *Let $G = (V, E)$ be a graph. The reflexive transitive closure of this graph G is a graph $G^* = (V, E^*)$ such that E^* contains an edge (a, b) if $a \xrightarrow{*} b$.*

Example In Figure 2.1, $1 \xrightarrow{*} 1$ holds, but $1 \xrightarrow{+} 1$ does not hold. Therefore the edge $(1, 1)$ is in the reflexive transitive closure of graph G, but not in the transitive closure of graph G.

Note that from this point on, the term transitive closure is used for reflexive transitive closure.

Definition 2.9 *Two nodes a and b are strongly connected iff $a \xrightarrow{*} b$ and $b \xrightarrow{*} a$ holds.*

Definition 2.10 *Let $G = (V, E)$ be a graph. C is a strongly connected component (SCC) of G if C is a maximal subgraph of G in which all nodes are reachable from each other (that is, for all nodes $v, w \in C$, $v \xrightarrow{*} w$ and $w \xrightarrow{*} v$ holds). Maximal means that adding any other node will break the mutual reachability property.*

Lemma 2.11 *If $G = (V, E)$ is a graph, then there exists a unique partition $C = \{C_1, \ldots, C_n\}$, where C_i is the i^{th} SCC of G.*

Proof In order to prove that a partition of G exists in its SCCs it must proved that every node is only member of exactly one SCC. Every node must at least be member of one component, as $v \xrightarrow{*} v$ holds for every node v. It only need to be proved now that there exists no node v such that v is member of more than

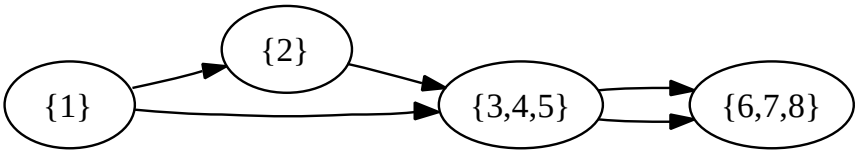

Figure 2.2: Condensed graph with nodes representing SCCs

one SCC. Assume such a node v does exist. Let C_1 and C_2 be two *different* components that contain v. Consider the arbitrary nodes $v_1 \in C_1$ and $v_2 \in C_2$. Then $v_1 \xrightarrow{*} v$, $v \xrightarrow{*} v_2$, $v_2 \xrightarrow{*} v$ and $v \xrightarrow{*} v_1$ hold, and therefore v_1 and v_2 are in the same SCC. This is a contradiction. Therefore any node v belongs exactly to one SCC.

Let us consider two partitions of the same graph in SCCs, $C = \{C_1, \ldots, C_n\}$ and $C' = \{C'_1, \ldots, C'_m\}$. Let $C_i \in C$ be an arbitrary element from C. Let $v \in C_i$ be an arbitrary node from SCC C_i. As C' is a partition in SCCs as well, there must exist a $C'_j \in C'$, such that $v \in C'_j$. Then $C_i = C'_j$, as both sets share the same definition, that is, they both contain nodes that are reachable from v and can reach v themselves. Therefore, a $1-1$ mapping from C to C' exists, making C unique. ∎

Example Consider the graph in Figure 2.1. The SCCs of this graph are $C_1 = \{1\}, C_2 = \{2\}, C_3 = \{3, 4, 5\}$ and $C_4 = \{6, 7, 8\}$. Note that these components are a partition of the nodes of the graph under consideration. The resulting *condensed graph*, the graph where nodes belonging to the same SCC are collapsed into a single node, is shown in Figure 2.1. Also note that this graph is acyclic. This is always the case for a graph whose nodes represent SCCs (See Lemma 2.14).

Definition 2.12 *The condensed graph G_c of a graph G is defined by $G_c = (V_c, E_c)$, where $V_c = \{C(v) | v \in V\}$ and*
$E_c = \{(C(v), C(w) | (v, w) \in E, C(v) \neq C(w)\}$. C is a function that maps a node to its corresponding SCC.

Definition 2.13 *A directed acyclic graph (DAG) is a directed graph that does not contain cycles.*

Lemma 2.14 *A condensed graph is a directed acyclic graph.*

Proof Assume there exists a condensed graph with a cycle. Let v and w be two nodes on that cycle. Then, $v \xrightarrow{*} w$ and $w \xrightarrow{*} v$ hold. Therefore, v and w are in the same SCC and the graph under consideration is not a condensed graph. This is a contradiction. Therefore, any condensed graph is acyclic. ∎

Definition 2.15 *In a DAG, a node without outgoing edges is called a* sink node*. Likewise, a node without incoming edges is called a* source node*. A SCC that has no outgoing edges in the condensed graph is called a source strongly connected component. A SCC that has no incoming edges in the condensed graph is called a sink strongly connected component.*

Definition 2.16 *Let $G = (V, E)$ be a graph. Then the reversed graph G^{-1}, which is graph G with the direction of the edges reversed, is given by $G^{-1} = (V, E^{-1})$, where $E^{-1} = \{(a, b)|(b, a) \in E\}$.*

Definition 2.17 *Let v be a node from a graph. Then v^* denotes the set of nodes reachable from v which is the set of target nodes of v in the transitive closure of the graph. This is also called the* successor set *of v. v^- denotes the set of nodes from which v can be reached, which is the set of nodes reachable from v in the reversed graph. This is also called the* predecessor set *of v.*

Lemma 2.18 *Let $G = (V, E)$ be a graph. For each node v, the SCC that it belongs to is defined by $C_v = v^* \cap v^-$.*

Proof First it must be proved that all elements in the set C_v are in the same SCC as v. For all $w \in v^*$, $v \xrightarrow{*} w$ holds. For all $w \in v^-$, $w \xrightarrow{*} v$ holds. Therefore, for all $w \in v^* \cap v^-$, $v \xrightarrow{*} w$ and $w \xrightarrow{*} v$ holds. Every $w \in v^* \cap v^-$ is thus in the same SCC as v.

Next, it must be proved that this set defines a complete SCC. Assume that there

exists a node x that is not in C_v but belongs to the same SCC. Then, $v \xrightarrow{*} x$ and $x \xrightarrow{*} v$ must hold. But then $x \in v^* \cap v^-$. This is a contradiction. Therefore, the set C_v defines the complete SCC that v belongs to. ∎

Lemma 2.19 *Two nodes v and w are in the same SCC iff $v^* = w^*$.*

Proof $\rightarrow$) Assume v and w are in the same SCC. Because $v \xrightarrow{*} w$ holds, $w^* \subseteq v^*$. (Everything that is reachable from w is reachable from v because w is reachable from v.) Because $w \xrightarrow{*} v$ holds, $w^* \supseteq v^*$. From $w^* \subseteq v^*$ and $w^* \supseteq v^*$ it follows that $v^* = w^*$.

($\leftarrow$) Assume $v^* = w^*$ holds. Then ($v \xrightarrow{*} w$ and $w \xrightarrow{*} v$ hold. Because w is reachable from v, $w^- = w^- \cup v^-$. Also, v is reachable from w, and therefore $v^- = w^- \cup v^-$ holds as well. As a direct consequence, $v^- = w^-$ holds. The SCCs C_v and C_w can now be expressed as $C_v = v^* \cap v^-$ and $C_w = w^* \cap w^-$. Because $v^* = w^*$ and $v^- = w^-$ hold, this can be rewritten as $C_v = C_w = v^* \cap v^-$. Therefore v and w are in the same SCC. ∎

Lemma 2.20 *For any graph G, G and its reversed graph G^{-1} have the same SCCs.*

Proof All possible paths in G can be followed in G^{-1} as well, only in reverse order. Therefore, nodes that are mutually reachable in G, remain mutually reachable in G^{-1} and nodes that are not mutually reachable in G are not mutually reachable in G^{-1}. ∎

Lemma 2.21 *Let $G = (V, E)$ be a graph. For any two nodes v and w, $v \in w^*$ and $w \in v^*$ iff v and w are in the same SCC.*

Proof ($\rightarrow$) Because $w \in v^*$, $v \xrightarrow{*} w$ holds. Likewise, because $v \in w^*$, $w \xrightarrow{*} v$ holds. This exactly defines mutual reachability. Therefore v and w are in the same SCC.

($\leftarrow$) v and w are in the same SCC. Therefore, they are mutually reachable. $v \xrightarrow{*} w$ implies $w \in v^*$ and $w \xrightarrow{*} v$ implies $v \in w^*$. ∎

Lemma 2.22 *Let $G = (V, E)$ be a graph. For any two nodes v and w, $v \in w^-$ iff $v^- \subseteq w^-$.*

Proof ($\rightarrow$) $v \in w^-$ holds. So $v \xrightarrow{*} w$ holds. Also, $\forall x(x \in v- \rightarrow x \xrightarrow{*} v)$ holds. By transitivity, $\forall x(x \in v- \rightarrow x \xrightarrow{*} w)$ holds as well. Therefore, $v- \subseteq w-$ is true.

($\leftarrow$) $v- \subseteq w-$ is true. Specifically, $v \in v-$ and therefore $v \in w-$.

```
1  procedure dfs(v: node)
2  begin
3  |  S = S ∪ {v}
4  |  precount = precount + 1
5  |  prenum[v] = precount
6  |  forall (v, w) ∈ E do
7  |  |  if w ∉ S then
8  |  |  |  dfs(w)
9  |  |  end
10 |  end
11 |  postcount = postcount + 1
12 |  postnum[v] = postcount
13 end
   /* Main procedure                                          */
14 begin
   |  /* S keeps track of visited nodes                       */
15 |  S = ∅
   |  /* Counters to keep track of preorder and postorder
   |     numbering                                            */
16 |  precount = 0
17 |  postcount = 0
18 |  forall v ∈ V do
19 |  |  if v ∉ S then
20 |  |  |  dfs(v)
21 |  |  end
22 |  end
23 end
```

Algorithm 1: Depth-First Search

Graphs often need to be searched. There exist many different ways though to do this. One of the most common search methods is the depth-first search DFS. The algorithm for a DFS is given by Algorithm 1. If a graph is explored using DFS starting at a node v, an edge (v, w) is chosen, where w can be any child node of v. The next edge is chosen from w. This continues until all outgoing

edges of a node have been traversed or an already visited node is encountered. In short, a DFS always tries to expand the path followed so far until it cannot proceed any further. In that case, it backtracks until there is a new branch to an unvisited node.

During the graph traversal, nodes can be numbered in the order they are visited. Two different numberings exists, namely the preorder numbering and the postorder numbering. In the preorder numbering, the next ordinal number is assigned as soon as a node is encountered. In the postorder numbering, the next ordinal number is assigned after all children of a node have been visited. Another term often used for postorder number is *completion number*. These completion numbers will turn out to be an important property later in this thesis.

2.2 Finite Differencing

A very interesting method to optimize high-level code into faster lower-level code is the application of finite differencing of computable expressions, which is presented in [8]. The basics of this method will be covered based on examples taken from [8].

Finite differencing is a method that has been used in mathematics to evaluate function values by using difference polynomials. The first difference polynomial is given by

$$p_1(x) = p(x+h) - p(x)$$

If a function is a polynomial of degree n, the n^{th}-degree difference polynomial is a constant. By only evaluating $p(x), p_1(x), \ldots, p_n(x)$, these values can be used to calculate the rest of the values $p(x+h), p(x+2h)$ and so on.

The easiest way to understand this method is by looking at an example. Table 2.1 shows the method applied to the function $p(x) = x^3$. The numbers that

x	x^3	Δ	Δ^2	Δ^3	Δ^4
0	**0**				
		1			
1	**1**		**6**		
		7		**6**	
2	**8**		**12**		**0**
		19		**6**	
3	**27**		**18**		0
		37		6	
4	**64**		24		0
		61		6	
5	125		30		0
		91		6	
6	216		36		0
		127		6	
7	343		42		0
		169		6	
8	512		48		0
		217		6	
9	729		54		
		271			
10	1000				

Table 2.1: Finite differencing applied to $p(x) = x^3$

are shown in boldface are numbers that have been explicitly calculated. The last column contains a zero. Knowing that this is a polynomial of degree 3, this entire column must be zero. Now the rest of the table can be filled in from right to left by using the rule

$$p_{i-1}(x+h) = p_{i-1}(x) + p_i(x).$$

This finite differencing method can be beneficial if evaluating derivatives is cheaper than evaluating the function itself. In [8], finite differencing is applied to computable expressions. Simply put, finite differencing on computable expressions looks at expressions and how they change if one of their variables changes. This is analogous to computing the change to $p(x+h)$ when h changes, which is what is actually done in the example above. When programming, statements that concisely describe their contents are often easy to understand and easy to

proof. But reevaluating such an expression over and over again can be a very expensive operation. This is where finite differencing becomes useful. By finding the initial value of an expression and updating the result of the expression incrementally, the same result is achieved. This is actually how efficient algorithms are often organized. Instead of recomputing results over and over again, results are constantly updated, which makes the code more efficient, but harder to understand. Ideally, steps in generating algorithms that perform incremental computations instead of complete evaluation of expressions should be taken by a compiler and not by a programmer.

A simple example illustrates best how finite differencing can be applied. This example is taken from [8]. It is presented here in a shorter form, because the focus here is on the concept, not on all the underlying theory. The following code reads a sequence of integers and prints the even numbers from this sequence.

```
1 a := {}
2 while eof = False do
3     read(i)
4     a := a ∪ {i}
5 end
6 print({x ∈ a|a  mod 2 = 0})
```

Algorithm 2: Algorithm printing even numbers from a sequence

The idea is to create a new variable that keeps track of the set $\{x \in a|a \mod 2 = 0\})$ incrementally. The variable E is used to keep track of this set. This leads to the algorithm shown in Algorithm 3. This code does

```
1 ∂E < a := {} >
2 a := {}
3 while eof = False do
4     read(i)
5     ∂E < a := a ∪ {i} >
6     a := a ∪ {i}
7 end
8 print({x ∈ a|a  mod 2 = 0})
```

Algorithm 3: Introducing the differentially computed variable E

not change anything yet, as only redundant code is inserted but eventually it

will provide a more efficient alternative. We will not go into depth on how to evaluate the differential computations of E. [8] provides some basic derivation rules, but for the sake of simplicity an intuitive approach is used here instead of a formal one. It is obvious that if the statement $a := \{\}$ is executed, $E := \{\}$ must be executed as well. The other derivative is with respect to the statement $a := a \cup \{i\}$. Checking whether this is an even number and adding i to the set E preserves the definition of E. In [8], Paige and Koenig state the following:

> By using a partial-correctness inference system similar to Hoare's [4] and based on Gerhart [3] and Schwartz [10], assertions can be propagated throughout Algorithm 4 and eliminate redundant **achieve** statements.

```
1  E := {}
2  a := {}
3  assert E := {x ∈ a|a mod 2 = 0}
4  while eof = False do
5      read(i)
6      achieve E := {x ∈ a|a mod 2 = 0}
7      if i mod 2 = 0 then
8          E := E ∪ {i}
9      end
10     a := a ∪ {i}
11     assert E := {x ∈ a|a mod 2 = 0}
12 end
13 print({x ∈ a|a mod 2 = 0})
```

Algorithm 4: Algorithm that incrementally computes result

Note that the *achieve* statement informs the compiler which expression must be computed incrementally. The step of propagating assertions is ommitted in this example, as the places where they hold are quite obvious. Now the expression that is the argument of the print function can be replaced by E. This leads to code where many other statements become redundant. The final result is shown in Algorithm 5.

The technique of finite differencing is just briefly introduced in this thesis. For a more thorough treament of this subject we refer to [8].

```
1 E := {}
2 while eof = False do
3     read(i)
4     if i  mod 2 = 0 then
5         E := E ∪ {i}
6     end
7 end
8 print(E);
```

Algorithm 5: Final result after replacing the function argument of print by E and dead code elimination

Chapter 3

Finding Strongly Connected Components in Directed Graphs

The concept of strongly connected components is rather simple: find maximal subsets of nodes that are mutually reachable. Lemma 2.18 (the SCC that a node v belongs to is defined by $v^* \cap v^-$) suggests a very simple algorithm to calculate the SCCs of a graph G: This procedure is quite easy to understand,

1 Pick an arbitrary node v from G
2 Calculate v^* and v^-
3 Output SCC $v^* \cap v^-$
4 Remove the found SCC from G
5 Repeat until no nodes are left

Algorithm 6: Intuitive algorithm calculating SCCs

but it is very inefficient as well. Therefore, other algorithms have been devised, which find SCCs in linear time. We will treat several different algorithms here. First Kosaraju-Sharir's algorithm [11] will be discussed, which is an algorithm that is conceptually the easiest to grasp. Then we will move on to Tarjan's

algorithm [12], which is even more efficient, but unfortunately not very easy to understand. Finally, some practical issues regarding the detection of SCCs are discussed and other work in this field is discussed. This is mainly about finding SCCs is parallel.

3.1 Kosaraju-Sharir's Algorithm

As we have seen in Chapter 2, any condensed graph is a directed acyclic graph (DAG). This also means that there must be at least one node in this graph that has no outgoing edges. This is quite easy to see, because if this would not be the case, we could always follow an outgoing edge from a node. If a graph contains n nodes and every node has at least one outgoing edge, then the longest path we can potentially find without revisiting a node is $n - 1$ edges long. Then, choosing another edge necessarily visits a node that has already been visited before (the pigeonhole principle, one cannot put n pigeons into $n - 1$ distinct pigeonholes). This would break the graph's property of being acyclic. The same holds for incoming edges. At least one node does not have an incoming edge. This is easy to see, as it corresponds to the reversed graph having at least one node with no outgoing edges.

The general idea behind Kosaraju-Sharir's algorithm is the following:

1. Find a node v that is in a source SCC.
2. Perform DFS on G^{-1} starting from v.
3. When the search cannot proceed any further, explored nodes are in the same SCC.
4. Remove SCC found from G^{-1}.
5. Repeat until G^{-1} is empty.

Finding a node that is in a source SCC turns out to be possible in linear time.

For now, we take this for granted, we will get back to this later. Figure 3.1 shows what happens if this algorithm is applied to a graph. Every node is labeled with its node number and its completion number (between parentheses). The first graph is the original graph. The other graphs are the reverse graph. Black nodes indicate nodes whose associated SCC have already been determined and thus have been removed from the graph. The node with the highest completion number has a double border, and all nodes that can be reached from that node are made grey. As the node with the highest completion number corresponds to a sink SCC in the reverse graph, a DFS (in this case a BFS does the job as well) gets stuck exactly when all nodes belonging to this SCC have been visited.

Lemma 3.1 *A DFS started in a sink SCC will exactly visit the entire SCC.*

Proof Let v be a node in a sink SCC. A DFS starting at v will exactly find v^*. By definition, any node of a SCC has to be in v^*. But also, any node that is in v^* has to be in the SCC, because we cannot leave a sink SCC in forward direction. ∎

The only remaining question is how to identify a node belonging to a sink SCC. Another strategy is to identify a node belonging to a source SCC. For finding SCCs this option is just as good, as a source SCC is a sink SCC in the reverse graph. And the SCCs of a graph and its reverse graph are the same (see Lemma 2.20). This is where the completion numbers come into play. The completion numbers require one DFS, and are therefore computable in linear time. In [6], a formal definition of the properties of nodes and their completion numbers is given. Lemma 3.2 is taken from this book, but a correction is included here along with some extra clarification.

Lemma 3.2 *Let $C_i = (V_i, E_i), 1 \leq i \leq k$, be the SCCs of $G = (V, E), n = |V|$, and let r_i be the node with largest completion number in C_i. Let us also assume that $compnum[r_1] < compnum[r_2] < \cdots < compnum[r_k]$. Then the following properties hold:*

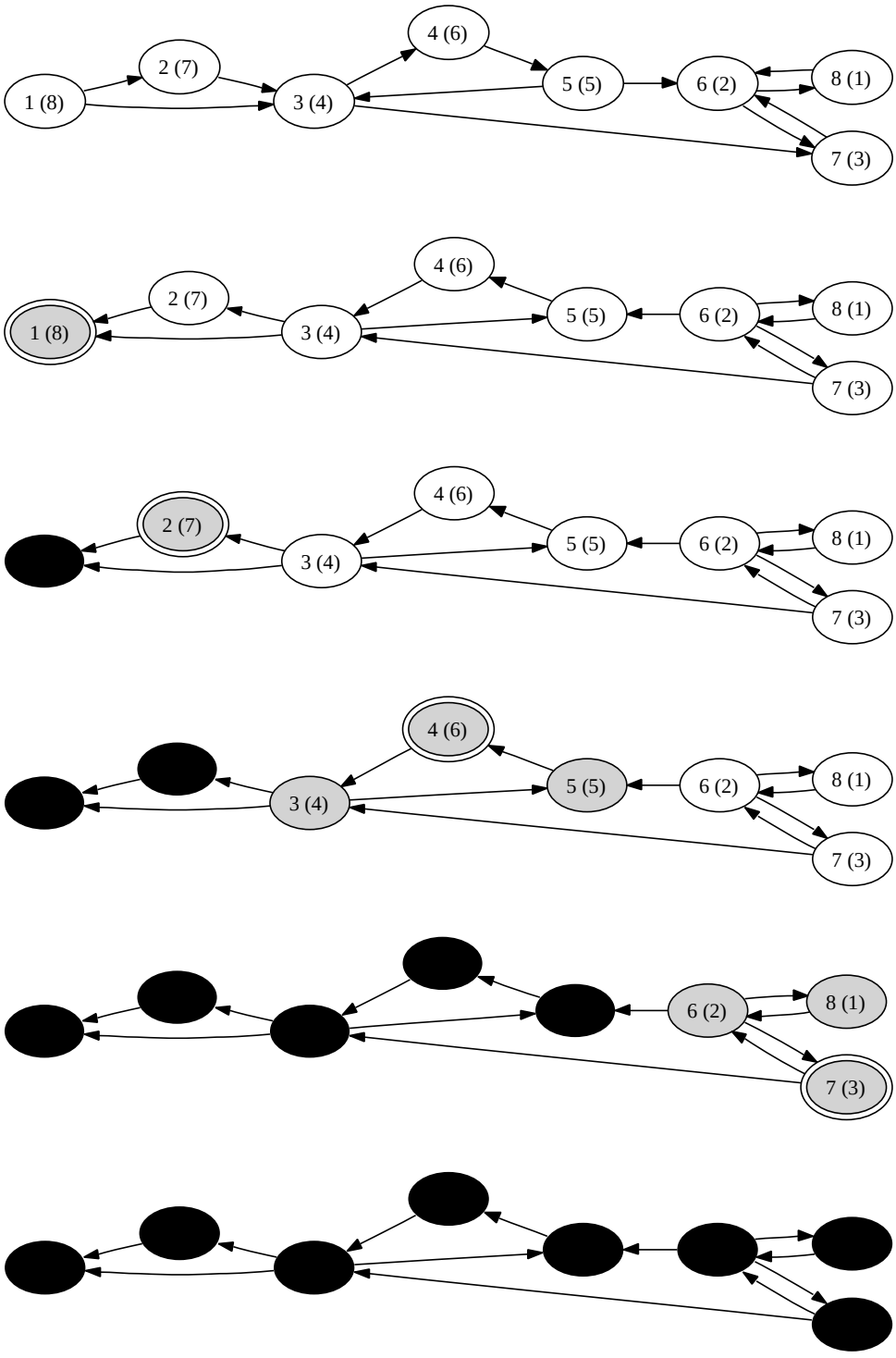

Figure 3.1: Finding SCCs in a directed graph

1. $compnum[r_k] = n$
2. *If* $v \xrightarrow[E]{*} r_i$ *then* $v \in \bigcup_{j \geq i} V_j$
3. *If* $r_i \xrightarrow[E^{-1}]{*} v$ *then* $v \in \bigcup_{j \geq i} V_j$
4. $r_i \xrightarrow[E^{-1}]{*} v$ *for all* $v \in V_i$

Proof 1) The node with completion number n belongs to some SCC.

2) Let $v \in V_j$ and $v \xrightarrow[E]{*} r_i$. Then $r_j \xrightarrow[E]{*} r_i$ and hence $dfs(r_i)$ is started before $dfs(r_j)$ is completed. Thus, either $compnum[r_i] \leq compnum[r_j]$ and hence $i \leq j$ by assumption or call $dfs(r_j)$ is nested within call $dfs(r_i)$ and hence $r_i \xrightarrow[E]{*} r_j$ and hence $i = j$. (This last statement actually is incorrect. If $i = j$, then $dfs(r_j)$ cannot be nested within call $dfs(r_i)$ as they both refer to the same function call.)

This proof is not trivial. It is easier to understand it if the following cases are considered.

1. $r_i = r_j$. Then $i = j$, making 2) true.
2. $r_i \neq r_j$ and $dfs(r_j)$ is called first. Because $r_j \xrightarrow[E]{*} r_i$ holds, $dfs(r_i)$ is nested within $dfs(r_j)$ and $dfs(r_i)$ will complete before $dfs(r_j)$, making $compnum[r_i] < compnum[r_j]$ and hence $i < j$ by assumption.
3. $r_i \neq r_j$ and $dfs(r_i)$ is called first. Because $i \neq j$, r_i and r_j are in distinct SCCs. As $r_j \xrightarrow[E]{*} r_i$ holds, $r_i \xrightarrow[E]{*} r_j$ cannot hold. Therefore, $dfs(r_j)$ can never be nested in the call $dfs(r_i)$. As a result, $dfs(r_i)$ completes before $dfs(r_j)$, making $compnum[r_i] < compnum[r_j]$ and hence $i < j$ by assumption.

3) Follows immediately from part 2) and the observation that $v \xrightarrow[E]{*} r_i$ iff $r_i \xrightarrow[E^{-1}]{*} v$.

4) Let $v \in V_i$. Then $v \xrightarrow[E]{*} r_i$ and the claim follows. ∎

If a DFS is started at r_k in the reverse graph, we know from Lemma 3.2, part 3

	Simple algorithm	Kosaraju-Sharir's Algorithm
1.	Pick an arbitrary node from G	Pick node with highest completion number
2.	Calculate v^* and v^-	Calculate v^* with respect to G^{-1}
3.	Output SCC $v^* \cap v^-$	Output SCC v^*
4.	Remove the found SCC from G	Remove the found SCC from G^{-1}
5.	Repeat until no nodes are left	Repeat until no nodes are left

Table 3.1: Comparison of a straightforward algorithm and Kosaraju-Sharir's Algorithm

if v is in r_k^* then $v \in V_k$. So every node that is reachable in the reverse graph from r_k is in SCC V_k. Furthermore, every node that is in SCC V_k is reachable from r_k. Next, a DFS from r_{k-1} can be started (nodes from the found SCCs so far should not be revisited), which finds the next SCC.

In Table 3.1 Algorithm 6 is compared to Kosaraju-Sharir's algorithm. It is interesting to see that actually these two algorithms do not differ that much. Apparently, choosing the node with the highest completion number instead of an arbitrary node leads to a faster algorithm. Now the question rises if it is possible to let a compiler automatically transform the intuitive algorithm into Kosaraju-Sharir's algorithm.

3.2 Tarjan's Algorithm

Kosaraju-Sharir's algorithm needs two DFS passes, one to assign completion numbers and one on the reverse graph to find the SCCs. Computing the reverse graph does not come for free as well. All these steps take linear time, but it would be nice to achieve the same results with less effort. Tarjan proposed an algorithm in [12] which computes SCCs in linear time by using only one DFS on a graph.

An important notion is that a SCC always contains a cycle, because if two nodes v and w are in the same SCC, $v \xrightarrow{*} w$ and $w \xrightarrow{*} v$ hold, thus forming a cycle. This is pretty obvious, but nevertheless a very important property. Together

with the property that the condensed graph is a DAG, this suggests another simple algorithm to find SCCs. This algorithm is given by Algorithm 7.

Input: Graph $G = (V, E)$
Output: Graph $G_c = (V_c, E_c)$, the condensed graph of G

1 $V_c := \{\{v\} : v \in V\}$
2 $E_c := \{(\{v\}, \{w\}) : (v, w) \in E\}$
3 $G_c := (V_c, E_c)$
4 **while** $has\_cycle(G_c) =$ ***true*** **do**
5 $\quad C := find\_cycle(G_c)$
6 $\quad V_c := V_c - C \cup \bigcup_{x \in C} x$
7 $\quad E_c := \{(v, w) : v, w \in V_c, v \neq w | \exists (x, y) \in E_c | x \subseteq v, y \subseteq w\}$
8 $\quad G_c := (V_c, E_c)$
9 **end**

Algorithm 7: Intuitive algorithm calculating SCCs

Lemma 3.3 *After execution of Algorithm 7, V_c contains nodes that define the SCCs of G.*

Proof After execution of Algorithm 7, G_c is a DAG. Furthermore, if $C \in V_c$, $v, w \in C$ then $v \xrightarrow{*} w$ and $w \xrightarrow{*} v$ hold in graph G and v and w belong to the same SCC. Also, if $v, w \in V$ belong to the same SCC then they will end up in the same node in G_c. Assume that nodes $v, w \in V$ exist that belong to the same SCC, but that end up in two different nodes in G_c, namely C_1 and C_2. Then, $C_1 \xrightarrow{*} C_2$ and $C_2 \xrightarrow{*} C_1$ would hold in G_c. But then G_c is not a DAG. This is a contradiction. Therefore, every node $C \in V_c$ contains the nodes belonging to exactly one SCC. ∎

Tarjan's algorithm does roughly the same as Algorithm 7, only in a more subtle way. It discovers cycles and because it uses DFS, some important properties can be proved which ensure that edges are only traversed once. The complete algorithm is shown in Algorithm 8. It is taken from [6]. A complete proof of this algorithm is given there as well. Here we will try to develop an intuitive feeling about how Tarjan's algorithm works by executing the algorithm on an example graph. This graph is shown in Figure 3.3.

Input: Graph $G = (V, E)$
Output: SCCs of Graph G

```
1  procedure dfs(v : node)
2  begin
3      count1 := count1 + 1
4      dfsnum[v] := count1
5      S := S ∪ v
6      push v onto unfinished
7      in_unfinished[v] := true
8      push v onto roots
9      forall w with (v, w) ∈ E do
10         if w ∉ S then
11             dfs(w)
12         else
13             if in_unfinished[w] then
14                 while dfsnum[top(roots)] > dfsnum[w] do
15                     pop(roots)
16                 end
17             end
18         end
19     end
20     if v = top(roots) then
21         repeat
22             w := pop(unfinished)
23             in_unfinished[w] := false
24         until v = w
25         pop(roots)
26     end
27 end

   /* Main function                                                  */
28 begin
29     unfinished := roots := empty_stack
30     S := ∅
31     count1 := 0
32     forall v ∈ V do
33         in_unfinished[v] := false
34     end
35     forall v ∈ V do
36         if v ∉ S then
37             dfs(v)
38         end
39     end
40 end
```

Algorithm 8: Tarjan's Algorithm for Computing SCCs

```
dfs(1)
 | dfs(2) - edge (1,2)
 |  | dfs(3) - edge (2,3)
 |  |  | dfs(4) - edge (3,4)
 |  |  |  | dfs(5) - edge (4,5)
 |  |  |  |  | (3 already visited) - edge (5,3)
 |  |  |  |  | dfs(6) - edge (5,6)
 |  |  |  |  |  | dfs(8) - edge (6,8)
 |  |  |  |  |  |  | (6 already visited) - edge (8,6)
 |  |  |  |  |  | dfs(7) - edge (6,7)
 |  |  |  |  |  |  | (6 already visited) - edge (7,6)
 |  |  | (7 already visited) - edge (3,7)
 | (3 already visited) - edge (1,3)
```

Figure 3.2: Calls to dfs when Tarjan's algorithm is applied to the graph from Figure 3.3

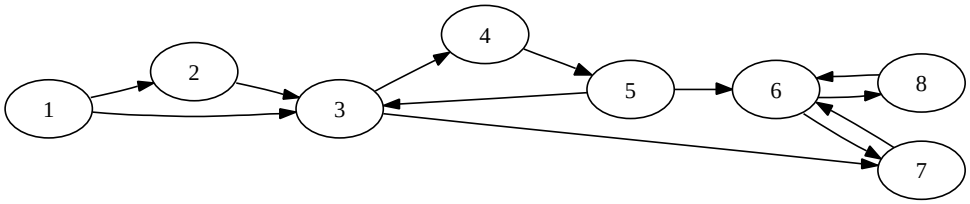

Figure 3.3: Example graph with 3 SCCs

Let us assume that the edges are traversed in the following depth-first order: $(1,2),(2,3),(3,4),(4,5),(5,3),(5,6),(6,8),(8,6),(6,7),(7,6),(3,7),(1,3)$. The corresponding call trace of calls to dfs is shown in Figure 3.2. It also shows all edges that cause these calls and indicates when an edge does not result into a call to dfs, as an other call to dfs with this target node has been made.

Traversal of the first few edges until we end up in node 5 is not very exciting.

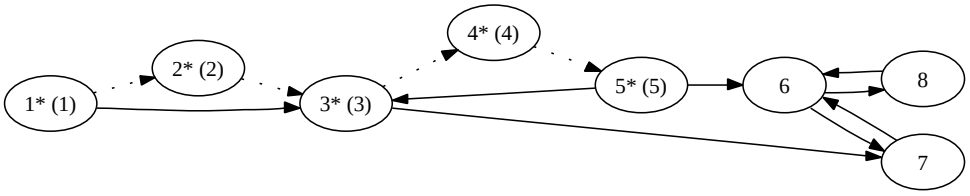

Figure 3.4: State after visiting edge $(4,5)$

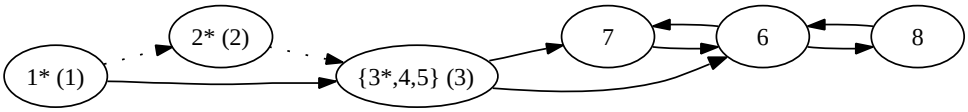

Figure 3.5: State after visiting edge $(5, 3)$

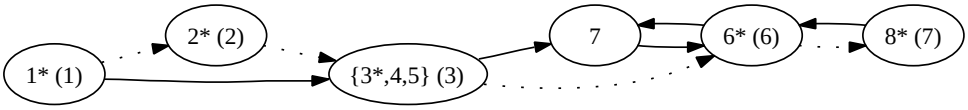

Figure 3.6: State after visiting edge $(6, 8)$

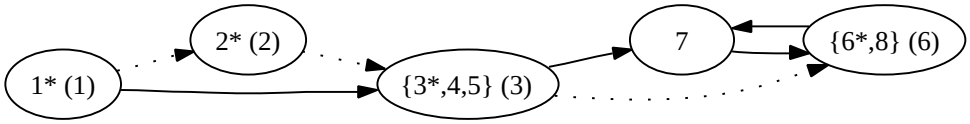

Figure 3.7: State after visiting edge $(8, 6)$

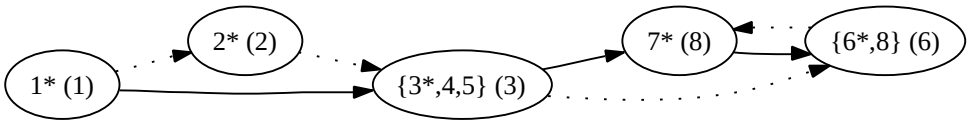

Figure 3.8: State after visiting edge $(6, 7)$

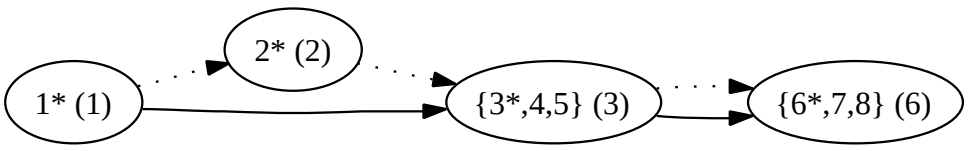

Figure 3.9: State after visiting edge $(7, 6)$

Each node visited is assigned a preorder number and is pushed onto the stacks $roots$ and $unfinished$. Their contents are as follows now (in $unfinished$, nodes that appear in $roots$ as well are marked with a $*$):

$roots = (1, 2, 3, 4, 5)$

$unfinished = (1*, 2*, 3*, 4*, 5*)$

A few things can be said. Up to now, all nodes visited form a path. This path is stored on both stacks at this time. Any node on this path can reach the node stored on top, namely 5. Now we traverse edge $(5, 3)$. Node 3 has already been visited before. We also know that node 3 is on the current path we are exploring. This property is being stored in the array $in\_unfinished$, which tracks whether a node is on the path we are currently exploring. We will consider this property in more detail later on. Anyway, visiting node 3 again creates a cycle in the graph that we have explored so far. By popping nodes off the stack $roots$ until we find node 3, we pop off the entire cycle except for node 3. Now the stacks $roots$ and $unfinished$ are not the same anymore. The stacks look as follows now:

$roots = (1, 2, 3)$

$unfinished = (1*, 2*, 3*, 4, 5)$

This leads to the following observation: If a node in $unfinished$ is also in $roots$, then all nodes on the right up to the next node that appears in $roots$ belong to the same SCC.

Next, we traverse $(5, 6)$, $(6, 8)$. Again, not much is happening. The current path we are exploring is extended with two nodes, which are SCCs consisting of one node each, namely themselves. The two stacks look as follows now:

$roots = (1, 2, 3, 6, 8)$

$unfinished = (1*, 2*, 3*, 4, 5, 6*, 8*)$

The next edge traversed is $(8, 6)$, which visits a node that has already been visited. This node is still unfinished as well, and therefore we can collapse this

cycle, which results in the following stacks:

$roots = (1, 2, 3, 6)$

$unfinished = (1*, 2*, 3*, 4, 5, 6*, 8)$

The edges $(6, 7)$ and $(7, 6)$ are visited next, forming another cycle. Again, this cycle is collapsed, leaving the stacks in the following state: $roots = (1, 2, 3, 6)$

$unfinished = (1*, 2*, 3*, 4, 5, 6*, 8, 7)$

The next edge $(3, 7)$ is different. Before this edge is traversed, the calls $dfs(8)$, $dfs(7)$, $dfs(6)$ and $dfs(5)$ are completed first. This can be seen in Figure 3.2. Looking at Algorithm 8, we see at line 19 that a check is made if the node on which dfs was called is on top of the stack $roots$. If so, we pop nodes off $unfinished$ until we find the root node we are looking for on top of unfinished. This node is popped off as well. Also, the top element of $roots$ is popped off. As stated above, in $unfinished$ a root node and all nodes on top of it are in the same SCC. Therefore we have exactly popped off an entire SCC. Note that no node could ever be added to this SCC anymore. This is not too hard to see, as whenever we reach a node belonging to this SCC again by dfs, there is no edge out of this SCC anymore, as all edges out of this SCC have been explored already. Therefore, a cycle containing any element from this SCC will never be found anymore.

$dfs(8)$ and $dfs(6)$ do not find their node on top of $roots$. $dfs(6)$ does and all nodes that appear on top of 6 on the stack $unfinished$ are popped off and they define the SCC $\{6, 7, 8\}$. Next, the calls $dfs(5)$ and $dfs(4)$ are completed as well. This results in the following stacks:

$roots = (1, 2, 3)$

$unfinished = (1*, 2*, 3*, 4, 5)$

Now, the edge $(3, 7)$ is traversed, which ends up in a node which already has been visited, namely node 7. In addition, node 7 is not on the stack $unfinished$. Therefore, its SCC must have been determined before and therefore it will not

change anymore. Therefore, nothing should be done for this node anymore.

Next, $dfs(3)$ is being completed. Node 3 is on top of the stack roots and therefore another SCC is popped off from the stack $unfinished$. This results in another SCC, namely $\{3, 4, 5\}$. The stacks contain the following now:
$roots = (1, 2)$
$unfinished = (1*, 2*)$

Likewise, $dfs(2)$ is finished now, which is a SCC as well. Next, $(1, 3)$ is explored, ending up in a node whose SCC is already determined. At last, $dfs(1)$ is completed, which is a SCC itself.

Summarizing, Tarjan's algorithm works because the following invariants hold, which are described in [6].

- There are no edges (x, y) with x belonging to a completed component and y belonging to an uncompleted component. That is because for any node in a completed SCC, the call to dfs is completed and all outgoing edges have been traversed already.
- The uncompleted components form a path and we are currently exploring edges out of the last component of this path. This component is identified by the top element from the stack $roots$.
- The nodes of each uncompleted SCC form a contiguous subsequence of the sequence $unfinished$. And whenever a node which is the top of $roots$ is finished, all nodes on top of this root in $unfinished$ are in one SCC that will not change anymore.

3.3 Finding Strongly Connected Components in Parallel

The algorithms described above are quite elegant and efficient. They are however not always practical, as they are sequential algorithms. Ideally, these algorithms should be parallelized. But unfortunately, both algorithms are based on DFS, which seems impossible to parallelize [9]. Knowing this, it is better to look at approaches that avoid DFS.

In order to discover potential parallelization opportunities, let us repeat Lemma 2.18:

> Let $G = (V, E)$ be a graph. For each node v, the SCC that it belongs to is defined by $C_v = v^* \cap v^-$.

Of course, this also implies for any node $w \in C_v$ that $C_v = w^* \cap w^-$ holds as well. We also observe that $C_v \subseteq v^*$ and $C_v \subseteq v^-$ hold. This also means that given any node x, that there does not exists any SCC C such that $C \not\subseteq x^*$ $C \not\subseteq V - x^*$, that is, any SCC is either completely contained within x^* or completely contained in its complement $V - x^*$. The same holds for x^- as well. In [14], such sets are referred to as SCC-closed sets. They list the following sets as being SCC-closed:

- Forward sets and backward sets. That is, v^* and v^- are SCC-closed, for any node $v \in V$. This is easy to see: for all $w \in v^*$, $w^* \subseteq v^*$ holds. By Lemma 2.18, $C_w = w^* \cap w^-$ and therefore $C_w \subseteq w^* \subseteq v^*$.
- The difference for any two backward sets (and likewise the difference for any two forward sets). That is, for any $v, w \in V$, $v^- - w^-$ is SCC-closed.

This suggests a divide-and-conquer approach for finding SCCs. In [14] such an approach is presented. They pick a random node, determine its backward set and recursively compute SCCs on the backward set and its complement. In

principle, any partition into sets which are SCC-closed works. In [2], a SCC detection algorithm is used in a heat transport simulation algorithm. They use a slightly different partition. This algorithm is shown in Algorithm 9. In [5],

```
1  procedure scc(V, E)
2  begin
3      if V = ∅ then
4          return;
5      end
6      Pick an arbitrary node v from V
7      Calculate v* and v⁻
8      Output SCC C := v* ∩ v⁻
9      V1 := v* − C
10     V2 := v⁻ − C
11     V3 := V − (V1 ∪ V2)
12     scc(V1, E)
13     scc(V2, E)
14     scc(V3, E)
15 end
```

Algorithm 9: Parallel algorithm detecting SCCs

McLendon et al. improve on this algorithm by trimming all nodes that are not on a cycle or a descendant of a cycle. This is done on both the normal graph and the reverse graph. They discard these nodes, but they could be regarded as SCCs consisting of a single node as well.

Chapter 4

A Stepwise Derived Solution

All algorithms for SCC detection described in previous chapters might be efficient, elegant or whatever positive qualification one can think of, but they share a common problem: they are not easily understood. And moreover, each problem might need a different implementation, tailored to the problem under consideration. All this extra effort obscures what the algorithm is all about: finding SCCs. Why not have a compiler make those decisions?

This is exactly what will be investigated here. How can a simple specification of the problem be written, namely finding SCCs, and let a compiler transform this into an efficient implementation.

4.1 Derivation of a More Efficient Algorithm

We consider an algorithm that computes SCC. From Lemma 2.19, it is known that two nodes v and w are in the same SCC iff $v^* = w^*$ iff $v^- = w^-$. Algo-

rithm 10 is based on this property. Its principle is very simple: Keep track of the transitive closure in reverse (called the predecessor set) of every node while edges are visited.

At first, every node can only be reached by itself. When an edge (a, b) is picked from U, the set of unvisited edges, every node that can reach a can now reach b as well. This is recorded in the statement $b^- := b^- \cup a^-$. In addition, the predecessor sets of the successors of b should be updated as well as all nodes from a^- can reach those nodes as well.

The final while loop calculates the SCCs by picking an arbitrary node and finding all nodes with an equivalent predecessor set. The set found is a SCC. This SCC is stored, and then subtracted from the set of remaining nodes. This process continues until all nodes have been put in a SCC.

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 while U ≠ ∅ do
 6     Pick an edge (a, b) ∈ U
       /* Update b⁻ and all successors of b                    */
 7     for i ∈ {x ∈ V | b ∈ x⁻} do
 8         i⁻ := i⁻ ∪ a⁻
 9     end
10 end
11 V₂ := V
12 count := 1
13 while V₂ ≠ ∅ do
14     pick random v from V₂
15     for w ∈ {x ∈ V | x⁻ = v⁻} do
16         SCC[count] := SCC[count] ∪ {w}
17     end
18     V₂ := V₂ − SCC[count]
19     count := count + 1
20 end
```

Algorithm 10: Basis algorithm calculating SCCs

In many compilers, dependencies between different loop iterations are identified

and the code is optimized such that those dependencies are not violated. In our code, there potentially exist interesting cases that depend on code that is executed in previous iterations. This information is however not directly available at compile time, which suggests that such information should be tracked at run time. We are going to track which predecessor sets have been used in an assignment statement, both on the right-hand side and the left-hand side of the assignment. In our example, these variables are put into set V'.

Input: A graph $G = (V, E)$
1 **forall** $v \in V$ **do**
2 $v^- := \{v\}$
3 **end**
4 $U := E$
5 $V' := \emptyset$
6 **while** $U \neq \emptyset$ **do**
7 Pick an edge $(a, b) \in U$
  `/* Update` b^- `and all successors of` b `*/`
8 **for** $i \in \{x \in V | b \in x^-\}$ **do**
9 $i^- := i^- \cup a^-$
10 $V' := V' \cup \{i, a\}$
11 **end**
12 **end**
13 $V_2 := V$
14 $count := 1$
15 **while** $V_2 \neq \emptyset$ **do**
16 pick random v from V_2
17 **for** $w \in \{x \in V | x^- = v^-\}$ **do**
18 $SCC[count] := SCC[count] \cup \{w\}$
19 **end**
20 $V_2 := V_2 - SCC[count]$
21 $count := count + 1$
22 **end**

Algorithm 11: Keeping track of nodes that are part of an assignment statement

Initially, for any predecessor set $v^- = \{v\}$ holds and V' is empty, as no node has been part of an assignment statement yet. Therefore, at the start of a loop iteration the following statements hold:

1. $\forall v \notin V'(v^- = \{v\})$
2. $\forall v \notin V' \nexists w(v \neq w \wedge v \in w^-)$

This suggests splitting the loop body in two separate cases, namely $v \in V'$ and $v \notin V'$. Splitting a loop into two different cases while preserving the same loop body within these cases is always valid. But different properties may hold due to the conditions implied by the condition of the if-statement. In our case this is b's membership of V'. Splitting is done in Algorithm 12.

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 while U ≠ ∅ do
 7     Pick an edge (a, b) ∈ U
 8     if b ∈ V' then
           /* Update b⁻ and all successors of b                */
 9         for i ∈ {x ∈ V | b ∈ x⁻} do
10             i⁻ := i⁻ ∪ a⁻
11             V' := V' ∪ {i, a}
12         end
13     else
           /* Update b⁻ and all successors of b                */
14         for i ∈ {x ∈ V | b ∈ x⁻} do
15             i⁻ := i⁻ ∪ a⁻
16             V' := V' ∪ {i, a}
17         end
18     end
19 end
20 V_2 := V
21 count := 1
22 while V_2 ≠ ∅ do
23     pick random v from V_2
24     for w ∈ {x ∈ V | x⁻ = v⁻} do
25         SCC[count] := SCC[count] ∪ {w}
26     end
27     V_2 := V_2 − SCC[count]
28     count := count + 1
29 end
```

Algorithm 12: Splitting loop body by membership of V'

If the branch $v \notin V'$ is taken, the iteration set of the for loop can be evaluated at compile-time, as $v \notin V' \rightarrow \nexists w(v \neq w \wedge v \in w^-)$. Therefore, the for loop in this branch can be fully unrolled. This is shown in Algorithm 13.

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 while U ≠ ∅ do
 7     Pick an edge (a, b) ∈ U
 8     if b ∈ V' then
           /* Update b⁻ and all successors of b                    */
 9         for i ∈ {x ∈ V | b ∈ x⁻} do
10             i⁻ := i⁻ ∪ a⁻
11             V' := V' ∪ {i, a}
12         end
13     else
           /* Update b⁻ and all successors of b                    */
           /* ∀v ∉ V'(v⁻ = {v})                                    */
           /* ∀v ∉ V' ∄w(v ≠ w ∧ v ∈ w⁻)                           */
           /* By applying the second rule, it is known that no
              x⁻ exists that contains b, except for potentially
              b⁻ itself.  By the first rule, b does qualify.
              Therefore, the loop can be unrolled completely.  */
14         b⁻ := b⁻ ∪ a⁻
15         V' := V' ∪ {b, a}
16     end
17 end
18 V_2 := V
19 count := 1
20 while V_2 ≠ ∅ do
21     pick random v from V_2
22     for w ∈ {x ∈ V | x⁻ = v⁻} do
23         SCC[count] := SCC[count] ∪ {w}
24     end
25     V_2 := V_2 − SCC[count]
26     count := count + 1
27 end
```

Algorithm 13: Unrolling for loop when b has never been on right hand side of an assignment

Another useful property of the for loop considered is that the order in which the different iterations are executed is not important. Therefore, one is completely free to split the iteration set into multiple disjoint sets. A compiler can use the statement within the loop to determine how to split the loop. In our case, it could detect that if $i^- \subseteq a^-$ then $i^- := i^- \cup a^-$ can be transformed to $i^- := a^-$, because all elements of i^- were already members of a^-. Such an optimization is desirable, as it eliminates the use of a union, which is more expensive than a simple assignment. An assignment might even be implemented by using references and copy-on-write, but such techniques are beyond the scope of this text. In order to create this situation, the iteration set of the for loop must be splitted in two cases, one in which all elements are subsets of a^- and one in which all elements are not subsets of a^-. This is done in Algorithm 14.

Next, the union in the for loop where $i^- \subseteq a^-$ holds can be transformed to $i^- := a^-$. Algorithm 15 incorporates this change.

Let us have a closer look at the iteration sets. It is known that within this if-branch $b \in V'$ holds. Therefore, if $b \in x^-$, then either $x = b$ or $x \neq b$. In the first case, $x \in V'$. In the latter case, $\{b, x\} \subseteq x^-$ and therefore $x \in V'$, as if $x \notin V'$, it would violate the property $\forall v \notin V'(v^- = \{v\})$. As a consequence, the iteration sets can be restricted to V'. This is shown in Algorithm 16.

As the iteration sets are restricted to V' now, it becomes clear that most of the updates to V' are redundant. In every iteration, i refers to a node that is from V'. Therefore, adding this node i to V' is a redundant operation. Furthermore, adding a within each iteration over and over again is not efficient as well. As $b \in b^-$ holds always, and either $x^- \subseteq a^-$ or $x^- \not\subseteq a^-$ at least one statement exists in which a is added to V'. As a result, redundant updates of V' can be removed and a needs only to be added to V' once. This is done in Algorithm 17.

If the split up iteration sets of the for loop ($iSet1$ and $iSet2$) are compared, it is favorable to maximize the size of $iSet1$, because then unions are prevented. The

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 while U ≠ ∅ do
 7     Pick an edge (a, b) ∈ U
 8     if b ∈ V' then
           /* Update b⁻ and all successors of b                    */
 9         iSet1 := {x ∈ V | b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
10         iSet2 := {x ∈ V | b ∈ x⁻ ∧ x⁻ ⊈ a⁻}
11         for i ∈ iSet1 do
12             i⁻ := i⁻ ∪ a⁻
13             V' := V' ∪ {i, a}
14         end
15         for i ∈ iSet2 do
16             i⁻ := i⁻ ∪ a⁻
17             V' := V' ∪ {i, a}
18         end
19     else
20         b⁻ := b⁻ ∪ a⁻
21         V' := V' ∪ {b, a}
22     end
23 end
24 V2 := V
25 count := 1
26 while V2 ≠ ∅ do
27     pick random v from V2
28     for w ∈ {x ∈ V | x⁻ = v⁻} do
29         SCC[count] := SCC[count] ∪ {w}
30     end
31     V2 := V2 − SCC[count]
32     count := count + 1
33 end
```

Algorithm 14: Split iteration set on subset condition

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 while U ≠ ∅ do
 7     Pick an edge (a, b) ∈ U
 8     if b ∈ V' then
           /* Update b⁻ and all successors of b                    */
 9         iSet1 := {x ∈ V | b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
10         iSet2 := {x ∈ V | b ∈ x⁻ ∧ x⁻ ⊈ a⁻}
11         for i ∈ iSet1 do
12             i⁻ := a⁻
13             V' := V' ∪ {i, a}
14         end
15         for i ∈ iSet2 do
16             i⁻ := i⁻ ∪ a⁻
17             V' := V' ∪ {i, a}
18         end
19     else
20         b⁻ := b⁻ ∪ a⁻
21         V' := V' ∪ {b, a}
22     end
23 end
24 V2 := V
25 count := 1
26 while V2 ≠ ∅ do
27     pick random v from V2
28     for w ∈ {x ∈ V | x⁻ = v⁻} do
29         SCC[count] := SCC[count] ∪ {w}
30     end
31     V2 := V2 − SCC[count]
32     count := count + 1
33 end
```

Algorithm 15: Elimination of useless union

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 while U ≠ ∅ do
 7     Pick an edge (a, b) ∈ U
 8     if b ∈ V' then
           /* Update b⁻ and all successors of b                    */
           /* b ∈ x⁻ implies that x ∈ V'.  Two cases:
             1. x = b.  b ∈ b⁻ is always true, as b is initially added to b⁻.
                And b ∈ V' for sure, because of the if-branch that is
                executed currently.
             2. x ≠ b.  Then if b ∈ x⁻, then x ∈ V', because if x ∉ V',
                x⁻ = {x} and x ≠ b.
                                                                    */
 9         iSet1 := {x ∈ V' | b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
10         iSet2 := {x ∈ V' | b ∈ x⁻ ∧ x⁻ ⊈ a⁻}
11         for i ∈ iSet1 do
12             i⁻ := a⁻
13             V' := V' ∪ {i, a}
14         end
15         for i ∈ iSet2 do
16             i⁻ := i⁻ ∪ a⁻
17             V' := V' ∪ {i, a}
18         end
19     else
20         b⁻ := b⁻ ∪ a⁻
21         V' := V' ∪ {b, a}
22     end
23 end
24 V2 := V
25 count := 1
26 while V2 ≠ ∅ do
27     pick random v from V2
28     for w ∈ {x ∈ V | x⁻ = v⁻} do
29         SCC[count] := SCC[count] ∪ {w}
30     end
31     V2 := V2 − SCC[count]
32     count := count + 1
33 end
```

Algorithm 16: Restrict iteration sets to V'

Input: A graph $G = (V, E)$
1 **forall** $v \in V$ **do**
2 $v^- := \{v\}$
3 **end**
4 $U := E$
5 $V' := \emptyset$
6 **while** $U \neq \emptyset$ **do**
7 Pick an edge $(a, b) \in U$
8 **if** $b \in V'$ **then**
9 $V' := V' \cup \{a\}$
`/* Update` b^- `and all successors of` b `*/`
10 $iSet1 := \{x \in V' | b \in x^- \wedge x^- \subseteq a^-\}$
11 $iSet2 := \{x \in V' | b \in x^- \wedge x^- \not\subseteq a^-\}$
12 **for** $i \in iSet1$ **do**
13 $i^- := a^-$
14 **end**
15 **for** $i \in iSet2$ **do**
16 $i^- := i^- \cup a^-$
17 **end**
18 **else**
19 $b^- := b^- \cup a^-$
20 $V' := V' \cup \{b, a\}$
21 **end**
22 **end**
23 $V_2 := V$
24 $count := 1$
25 **while** $V_2 \neq \emptyset$ **do**
26 pick random v from V_2
27 **for** $w \in \{x \in V | x^- = v^-\}$ **do**
28 $SCC[count] := SCC[count] \cup \{w\}$
29 **end**
30 $V_2 := V_2 - SCC[count]$
31 $count := count + 1$
32 **end**

Algorithm 17: Remove redundant updates to V'

only thing that is different between the sets is the subset condition $x^- \subseteq a-$. As a rule of thumb, maximizing $|a^-|$ causes as many as possible other predecessor sets to meet the condition $x^- \subseteq a^-$, thereby maximizing $iSet1$. In order to maximize $|a^-|$, instead of picking an arbitrary edge from U, an edge (a, b) is chosen such that $|a^-|$ is maximal (with respect to the domain of U, not V).

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 while U ≠ ∅ do
 7     Pick an edge (a, b) ∈ U with |a⁻| maximal
 8     if b ∈ V' then
 9         V' := V' ∪ {a}
           /* Update b⁻ and all successors of b                    */
10         iSet1 := {x ∈ V' | b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
11         iSet2 := {x ∈ V' | b ∈ x⁻ ∧ x⁻ ⊈ a⁻}
12         for i ∈ iSet1 do
13             i⁻ := a⁻
14         end
15         for i ∈ iSet2 do
16             i⁻ := i⁻ ∪ a⁻
17         end
18     else
19         b⁻ := b⁻ ∪ a⁻
20         V' := V' ∪ {b, a}
21     end
22 end
23 V2 := V
24 count := 1
25 while V2 ≠ ∅ do
26     pick random v from V2
27     for w ∈ {x ∈ V | x⁻ = v⁻} do
28         SCC[count] := SCC[count] ∪ {w}
29     end
30     V2 := V2 − SCC[count]
31     count := count + 1
32 end
```

Algorithm 18: Pick edge (a,b) with $|a^-|$ maximal

Knowing that finally the contents of the predecessor sets is not important but only the question whether they are equal (the statement for $w \in \{x \in V | x^- =$

$v^-\}$), the idea rises to find nodes whose predecessor sets will never be made equal to another node's predecessor set.

In order to do so, a set of nodes for which this is the case must be kept track of. This set will be called $Sink$. This name is used because of the insight that any update done to a node from $Sink$ is only propagated to nodes within $Sink$ and will never cause nodes to become equivalent (as no node from $Sink$ is a subset of a^-).

A node that is member of $Sink$ is a node whose predecessor set will never become a subset of any predecessor set belonging to a node not in $Sink$. Initially, this set is empty. Every time an edge is chosen, all nodes that have a bigger predecessor set are added to $Sink$, because they are not a subset of any other predecessor set that is not in $Sink$ currently. These nodes also do not have outgoing edges anymore, because if that had, they would have been chosen because their predecessor set is bigger. Specifically, the following rule holds for nodes in the set $Sink$:

$$if\ x \in Sink\ then\ \forall y \notin Sink(x \notin y^-)$$

That is, whenever a node becomes a member of $Sink$, this node will never be part of any predecessor set of a node outside of $Sink$.

Currently, we have not been able to find a formal method to derive the $Sink$ set automatically. The same holds for the rule presented above. The rule itself can be proved valid by induction on the code though. Further research is needed in order to find a formal method to make usage of the set $Sink$ a logical consequence of the code.

In order to maintain the set $Sink$, another restriction is needed when picking an edge to visit. This restriction is that an edge must be chosen such that $a \in V'$ preferably holds. With this extra restriction, it is known that when $a \notin V'$

holds, all nodes that are currently in V' should be added to $Sink$, as they do not have any unvisited outgoing edges left.

The introduction of the set $Sink$ is shown in Algorithm 19.

Next, the iteration sets $iSet1$ and $iSet2$ are splitted on set membership of $Sink$. This results in the code shown in Algorithm 20. The postfixes S and Sc in the new iteration sets stand for $Sink$ and $Sink$ complement.

$iSetS$ and $iSet2c$ are both marked as being empty. For $iSetS$, this follows from the rule that is stated above which is repeated here:

$$if\ x \in Sink\ then\ \forall y \notin Sink(x \notin y^-)$$

This implies

$$if\ x \in Sink\ then\ \forall y \notin Sink(x^- \not\subseteq y^-)$$

and more specifically, for $a \notin Sink$

$$if\ x \in Sink\ then\ x^- \not\subseteq a^-$$

Applied to $iSet1$, this rule adds the condition $x^- \not\subseteq a^-$, resulting in the definition

$$iSet1S := \{x \in V' | x \in Sink \wedge b \in x^- \wedge x^- \subseteq a^- \wedge x^- \not\subseteq a^-\}$$

which contains contradicting conditions, producing an empty set.

We prove that $iSet2c$ is empty by proving that $x \in V' \wedge x \notin Sink$ is a totally ordered set by $\subseteq$ in Algorithm 20.

Basis Let $S = \{x \in V' | x \notin Sink\}$, evaluated just after line 13 of Algorithm 20. Initially this set is empty making the statement above true.

Inductive step Let S_i be the set S in the i^{th} iteration. Assume the statement above holds for S_i. The set S_i is evaluated at the same position in the loop as above. It is known that in S_i, the node a has the biggest predecessor set. All

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 Sink := ∅
 7 while U ≠ ∅ do
 8     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
 9     if a ∈ V' then
10         Sink := Sink ∪ {x ∈ V' | |x⁻| > |a⁻|}
11     else
12         Sink := Sink ∪ {x ∈ V'}
13     end
14     if b ∈ V' then
15         V' := V' ∪ {a}
           /* Update b⁻ and all successors of b                    */
16         iSet1 := {x ∈ V' | b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
17         iSet2 := {x ∈ V' | b ∈ x⁻ ∧ x⁻ ⊈ a⁻}
18         for i ∈ iSet1 do
19             i⁻ := a⁻
20         end
21         for i ∈ iSet2 do
22             i⁻ := i⁻ ∪ a⁻
23         end
24     else
25         b⁻ := b⁻ ∪ a⁻
26         V' := V' ∪ {b, a}
27     end
28 end
29 V2 := V
30 count := 1
31 while V2 ≠ ∅ do
32     pick random v from V2
33     for w ∈ {x ∈ V | x⁻ = v⁻} do
34         SCC[count] := SCC[count] ∪ {w}
35     end
36     V2 := V2 − SCC[count]
37     count := count + 1
38 end
```

Algorithm 19: Introduction of sink nodes

nodes with a bigger predecessor set have been moved to $Sink$ before. As S_i is totally ordered by $\subseteq$, it must be proved that S_{i+1} is totally ordered by $\subseteq$ as well. This is done by tracing the execution of one iteration. Two cases exist:

1. $b \in V'$: As S_i is totally ordered, $iSet2Sc$ is empty because a is in S_i, and within this set $|a^-|$ is maximal. Therefore, $\forall x \in S_i(x^- \subseteq a^-)$.

   Nodes in $iSet2S$ are elements that are in $Sink$. As $Sink$ only grows, these elements can never end up in S_j, where $j \geq i$.

   The predecessor sets of the nodes in $iSet1Sc$ are being made equal to a^-. These nodes now have a maximal predecessor set as well and therefore S_i still is totally ordered by $\subseteq$.

   In the next iteration, a new edge is chosen. For such an edge $a \notin Sink$ holds, otherwise this node was maximal before and not chosen, contradicting the selection condition. All nodes with a bigger predecessor set are added to $Sink$, and if $a \notin V'$, then all nodes from V' are added to $Sink$. This removes elements from S_i, which does not change the property of being totally ordered.

2. $b \notin V'$: Because of the statement $b^- := b^- \cup a^-$, $a^- \subseteq b^-$ holds (in this case even $a^- \subset b^-$ holds, because $\forall v \notin V'(\nexists w(v \in w^-))$. After b is added to V' the total order property is preserved, because the predecessor sets of all other elements in S_i are a subset of b^-.

   Picking the next edge and proving that the total order is preserved is the same as in the other case.

As the total order property of S is preserved during execution, the set $iSet2c$ will always be empty. ∎

Algorithm 21 shows the results of removing the for loops with empty iteration sets.

Now two loops are left. Looking at the second loop shows that every member

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 Sink := ∅
 7 while U ≠ ∅ do
 8     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
 9     if a ∈ V' then
10         Sink := Sink ∪ {x ∈ V' | |x⁻| > |a⁻|}
11     else
12         Sink := Sink ∪ {x ∈ V'}
13     end
14     if b ∈ V' then
15         V' := V' ∪ {a}
           /* Update b⁻ and all successors of b                    */
16         iSet1S := {x ∈ V' | x ∈ Sink ∧ b ∈ x⁻ ∧ x⁻ ⊆ a⁻}(:= ∅)
17         iSet1Sc := {x ∈ V' | x ∉ Sink ∧ b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
18         iSet2S := {x ∈ V' | x ∈ Sink ∧ b ∈ x⁻ ∧ x⁻ ⊈ a⁻}
19         iSet2Sc := {x ∈ V' | x ∉ Sink ∧ b ∈ x⁻ ∧ x⁻ ⊈ a⁻}(:= ∅)
20         for i ∈ iSet1S do
21             i⁻ := a⁻
22         end
23         for i ∈ iSet1Sc do
24             i⁻ := a⁻
25         end
26         for i ∈ iSet2S do
27             i⁻ := i⁻ ∪ a⁻
28         end
29         for i ∈ iSet2Sc do
30             i⁻ := i⁻ ∪ a⁻
31         end
32     else
33         b⁻ := b⁻ ∪ a⁻
34         V' := V' ∪ {b, a}
35     end
36 end
37 V2 := V
38 count := 1
39 while V2 ≠ ∅ do
40     pick random v from V2
41     for w ∈ {x ∈ V | x⁻ = v⁻} do
42         SCC[count] := SCC[count] ∪ {w}
43     end
44     V2 := V2 − SCC[count]
45     count := count + 1
46 end
```

Algorithm 20: Split iteration sets on *Sink* membership

Input: A graph $G = (V, E)$
1 **forall** $v \in V$ **do**
2 $v^- := \{v\}$
3 **end**
4 $U := E$
5 $V' := \emptyset$
6 $Sink := \emptyset$
7 **while** $U \neq \emptyset$ **do**
8 Pick an edge $(a, b) \in U$ with $|a^-|$ maximal and preferably $a \in V'$
9 **if** $a \in V'$ **then**
10 $Sink := Sink \cup \{x \in V' || x^-| > |a^-|\}$
11 **else**
12 $Sink := Sink \cup \{x \in V'\}$
13 **end**
14 **if** $b \in V'$ **then**
15 $V' := V' \cup \{a\}$
 `/* Update` b^- `and all successors of` b `*/`
16 $iSet1Sc := \{x \in V' | x \notin Sink \wedge b \in x^- \wedge x^- \subseteq a^-\}$
17 $iSet2S := \{x \in V' | x \in Sink \wedge b \in x^- \wedge x^- \not\subseteq a^-\}$
18 **for** $i \in iSet1Sc$ **do**
19 $i^- := a^-$
20 **end**
21 **for** $i \in iSet2S$ **do**
22 $i^- := i^- \cup a^-$
23 **end**
24 **else**
25 $b^- := b^- \cup a^-$
26 $V' := V' \cup \{b, a\}$
27 **end**
28 **end**
29 $V_2 := V$
30 $count := 1$
31 **while** $V_2 \neq \emptyset$ **do**
32 pick random v from V_2
33 **for** $w \in \{x \in V | x^- = v^-\}$ **do**
34 $SCC[count] := SCC[count] \cup \{w\}$
35 **end**
36 $V_2 := V_2 - SCC[count]$
37 $count := count + 1$
38 **end**

Algorithm 21: Removal of loops with empty iteration set

of the iteration set is element of $Sink$. As $Sink$ is a set that only grows, any node that has been in iteration set $iSet2S$ will never be member of $iSet1Sc$ anymore, as such a node cannot be removed from $Sink$ anymore. Additionally, as $i^- \not\subseteq a^-$, $(i^- \cup a^-) \supset a^-$ and therefore such a statement never establishes equivalence between predecessor sets.

It can be concluded that equivalence is only established when nodes are not in $Sink$. As soon as nodes end up in $Sink$, their predecessor set will never be made equivalent to a predecessor set of another node anymore, as both i and a in the first loop (with $iSet1Sc$ as iteration set) are not member of $Sink$. Therefore, updates on nodes from $Sink$ are redundant and can be skipped. This is shown in Algorithm 22.

At this point, the only loop that is left is the loop with iteration set $iSet1Sc$. Computation of this iteration set is an expensive operation. Therefore, this set is going to be kept track of incrementally by using finite differencing. The notion of finite differencing will be used a bit informally though, but the thought behind the transformation remains the same.

Two variables will be used, named H and EQ, which stands for *History* and *Equivalence* respectively. Both keep track of the set

$$\{x \in V' | x \notin Sink\}$$

but both will use a different data structure to track this set. H stores these nodes in an ordered list, ordered by $\subseteq$ on their predecessor sets. EQ stores ordered sets of nodes, such that within such a set, nodes are equivalent, and if two nodes a, b are in different sets where a appears left of b, without loss of generality, then $a^- \subseteq b^-$.

Example $H := (1, 2, 3, 4, 5, 6)$ and $EQ := (\{1\}, \{2, 3, 4\}, \{5, 6\})$
Then $1^- \subseteq 2^- \subseteq 3^- \subseteq 4^- \subseteq 5^- \subseteq 6^-$ and additionally $2^- = 3^- = 4^-$ and

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 Sink := ∅
 7 while U ≠ ∅ do
 8     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
 9     if a ∈ V' then
10         Sink := Sink ∪ {x ∈ V' | |x⁻| > |a⁻|}
11     else
12         Sink := Sink ∪ {x ∈ V'}
13     end
14     if b ∈ V' then
15         V' := V' ∪ {a}
           /* Update b⁻ and all successors of b                    */
16         iSet1Sc := {x ∈ V' | x ∉ Sink ∧ b ∈ x⁻ ∧ x⁻ ⊆ a⁻}
17         for i ∈ iSet1Sc do
18             i⁻ := a⁻
19         end
20     else
21         b⁻ := b⁻ ∪ a⁻
22         V' := V' ∪ {b, a}
23     end
24 end
25 V2 := V
26 count := 1
27 while V2 ≠ ∅ do
28     pick random v from V2
29     for w ∈ {x ∈ V | x⁻ = v⁻} do
30         SCC[count] := SCC[count] ∪ {w}
31     end
32     V2 := V2 − SCC[count]
33     count := count + 1
34 end
```

Algorithm 22: Eliminate updates of predecessor sets of nodes from *Sink*

$5^- = 6^-$.

Algorithm 23 shows the two variables. The definition of the variables is surrounded by brackets, indicating that a specific data structure is used to store the set. This data structure is not visible in the code, but in this text the knowledge about the underlying data structure will be used.

Instead of reevaluating the variables H and EQ in each iteration again, these variables will be computed incrementally. In order to do this, any change to the variables H and EQ depend on must be detected. These variables are V' and $Sink$. Additionally, the effect of any statement that changes the contents of a predecessor set must be taken into account as the data structures must be updated accordingly. This especially applies to EQ.

Algorithm 24 shows the code after finite differencing applied to H and EQ. Each step will be explained in detail. Initially, H and EQ are empty, as V' is empty.

The first change of a variable H and EQ depend on is the change of $Sink$ at line 11. It is known that any node having a bigger predecessor set than a will appear in sets right of the set that a belongs to in EQ. Therefore, removing elements from the end of the list until $a \in EQ(end)$, where end is the index of the last element of the list, removes exactly those nodes whose predecessor sets are bigger than a^-. These elements should also be removed from H. Every time an element X is removed from EQ, the corresponding number of nodes are removed from H ($\#X$ is the number of elements in X).

The next position where a variable is changed on which H and EQ depend is at line 20. This time, all element from V' must be added to the set $Sink$. But this is exactly the same as adding the set $\{x \in V' | x \notin Sink\}$ to $Sink$. Therefore, all nodes from EQ and H must be removed. This is done in a similar way as in the first branch of the if-statement, only now the algorithm does not stop until EQ is empty.

Input: A graph $G = (V, E)$
1 **forall** $v \in V$ **do**
2 $v^- := \{v\}$
3 **end**
4 $U := E$
5 $V' := \emptyset$
6 $Sink := \emptyset$
7 **while** $U \neq \emptyset$ **do**
8 Pick an edge $(a, b) \in U$ with $|a^-|$ maximal and preferably $a \in V'$
9 **if** $a \in V'$ **then**
10 $Sink := Sink \cup \{x \in V' || x^-| > |a^-|\}$
11 **else**
12 $Sink := Sink \cup \{x \in V'\}$
13 **end**
14 $H := [x \in V' \wedge x \notin Sink]$
15 $EQ := [x \in V' \wedge x \notin Sink]$
16 **if** $b \in V'$ **then**
17 $V' := V' \cup \{a\}$
/* Update b^- and all successors of b */
18 $iSet1Sc := \{x \in V' | x \notin Sink \wedge b \in x^- \wedge x^- \subseteq a^-\}$
19 **for** $i \in iSet1Sc$ **do**
20 $i^- := a^-$
21 **end**
22 **else**
23 $b^- := b^- \cup a^-$
24 $V' := V' \cup \{b, a\}$
25 **end**
26 **end**
27 $V_2 := V$
28 $count := 1$
29 **while** $V_2 \neq \emptyset$ **do**
30 pick random v from V_2
31 **for** $w \in \{x \in V | x^- = v^-\}$ **do**
32 $SCC[count] := SCC[count] \cup \{w\}$
33 **end**
34 $V_2 := V_2 - SCC[count]$
35 $count := count + 1$
36 **end**

Algorithm 23: Introduce variable that keeps track of iteration set

At line 32, V' changes. To be sure that V' changes, the statement adding a is guarded by $a \notin V'$. If the element is added to V', it should be added to H and EQ as well. If not, they do not need to be updated. In case a is added to V', we add a to the end of H and $\{a\}$ to the end of EQ. Note that at this point both H and EQ are empty and adding these elements to H and EQ is valid because of the proof stated above that $\{x \in V' | x \notin Sink\}$ can be totally ordered by $\subseteq$. As $|a^-|$ is maximal and $a \in V' \wedge a \notin Sink$, its valid position in the list it at the end.

The assignment statement at line 45 in the for loop is the next statement that changes EQ. The iteration $iSet1Sc$ can be easily computed by using EQ. In general, EQ looks as follows:

$$EQ = (\ldots, \{\ldots, b, \ldots\}, X_1, \ldots, X_n, \{\ldots, a, \ldots\})$$

The condition $b \in x^-$ is true for all x appearing in the same set in EQ as b and all sets right of this set. In the sequence above, that is the set containing b, X_1 through X_n and the set containing a. By the assignment statement at line 45, all the nodes contained in these sets become equivalent. This change in EQ is reflected by merging the two rightmost nodes until $b \in EQ(end)$. Updating EQ is guarded by $b \notin Sink$, because if $b \in Sink$, then b certainly cannot be found in EQ, as it only contains nodes that are not member of $Sink$. Now $EQ(end)$ is equal to $iSet1Sc$ and $EQ(end)$ can be substituted for $iSet1Sc$.

At line 49, it cannot be decided whether $a \in V'$ holds or not. Therefore, a guarding if-statement is created, and H and EQ are updated accordingly. $b \notin V'$ holds for sure, so the updates for b to H and EQ do not have to be guarded.

This entire process of finite differencing of H and EQ results in the code shown in Algorithm 24.

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 V' := ∅
 6 Sink := ∅
 7 H := ∅
 8 EQ := ∅
 9 while U ≠ ∅ do
10     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
11     if a ∈ V' then
12         Sink := Sink ∪ {x ∈ V' | |x⁻| > |a⁻|}
13         while a ∉ EQ(end) do
14             X := EQ(end)
15             remove_end(EQ)
16             for i ∈ [1, #X] do
17                 remove_end(H)
18             end
19         end
20     else
21         Sink := Sink ∪ {x ∈ V'}
22         while EQ ≠ ∅ do
23             X := EQ(end)
24             remove_end(EQ)
25             for i ∈ [1, #X] do
26                 remove_end(H)
27             end
28         end
29     end
30     if b ∈ V' then
31         if a ∉ V' then
32             V' := V' ∪ {a}
33             H := H + [a]
34             EQ := EQ + [{a}]
35         end
           /* Update b⁻ and all successors of b                    */
36         if b ∉ Sink then
37             while b ∉ EQ(end) do
38                 X := EQ(end)
39                 remove_end(EQ)
40                 Y := EQ(end)
41                 remove_end(EQ)
42                 EQ := EQ + [X ∪ Y]
43             end
44             for i ∈ EQ(end) do
45                 i⁻ := a⁻
46             end
47         end
48     else
49         b⁻ := b⁻ ∪ a⁻
50         if a ∉ V' then
51             V' := V' ∪ {a}
52             H := H + [a]
53             EQ := EQ + [{a}]
54         end
55         V' := V' ∪ {b}
56         H := H + [b]
57         EQ := EQ + [{b}]
58     end
59 end
60 V2 := V
61 count := 1
62 while V2 ≠ ∅ do
63     pick random v from V2
64     for w ∈ {x ∈ V | x⁻ = v⁻} do
65         SCC[count] := SCC[count] ∪ {w}
66     end
67     V2 := V2 − SCC[count]
68     count := count + 1
69 end
```

Algorithm 24: Apply finite differencing to H and EQ

As can be seen, applying finite differencing generates quite a lot of code. In principle, this code can be used to directly optimize further. But in order to minimize code size some common code is moved and redundant statements are eliminated.

In Algorithm 24, the if-statements at line 32 and line 49 are the same and can be moved in front of the containing if-statement.

At line 35, $Sink$ is used. As $b \in V'$ holds, the condition of the if-statement that follows can be expanded with $b \in V'$. This results in the condition $b \in V' \wedge b \notin Sink$, which can be replaced by $b \notin H$. Now $Sink$ is not used anymore in statements other than updating itself it has become redundant. Therefore, any statement in which $Sink$ occurs can be removed.

The result of these steps is shown in Algorithm 25.

It is known that each element of EQ contains nodes which have the same predecessor set. An other observation is that only nodes that occur in EQ can become equivalent to other nodes. Only nodes from the last element are made equivalent. Therefore, when a node is removed from EQ, such a node will never be on the right hand side of an assignment of a predecessor set. As a consequence, the iteration set of the for loop at line 55 in Algorithm 25 is determined for any $v \in EQ(end)$ if the last element of EQ is removed. Therefore, if an element is removed from EQ, then for any v from this set, the iteration set evaluates to exactly the set which is removed from EQ. Therefore, the execution of the for loop at line 55 of Algorithm 25 can be moved to any position where an element is removed from EQ, and the iteration set is exactly the set removed from EQ. Finally, after all edges have been visited, all remaining elements from EQ can be removed, as the contents of these elements will not change anymore.

The result of this transformation is shown in Algorithm 26.

All nodes that have at least one incoming or outgoing edge have been added to V' at the end of the algorithm. Therefore, these nodes have been added to EQ

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 U := E
 5 H := ∅
 6 EQ := ∅
 7 while U ≠ ∅ do
 8     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
 9     if a ∈ H then
10         while a ∉ EQ(end) do
11             X := EQ(end)
12             remove_end(EQ)
13             for i ∈ [1, #X] do
14                 remove_end(H)
15             end
16         end
17     else
18         while EQ ≠ ∅ do
19             X := EQ(end)
20             remove_end(EQ)
21             for i ∈ [1, #X] do
22                 remove_end(H)
23             end
24         end
25     end
26     if a ∉ V' then
27         V' := V' ∪ {a}
28         H := H + [a]
29         EQ := EQ + [{a}]
30     end
31     if b ∈ V' then
32         if b ∉ H then
33             while b ∉ EQ(end) do
34                 X := EQ(end)
35                 remove_end(EQ)
36                 Y := EQ(end)
37                 remove_end(EQ)
38                 EQ := EQ + [X ∪ Y]
39             end
40             for i ∈ EQ(end) do
41                 i⁻ := a⁻
42             end
43         end
44     else
45         b⁻ := b⁻ ∪ a⁻
46         V' := V' ∪ {b}
47         H := H + [b]
48         EQ := EQ + [{b}]
49     end
50 end
51 V₂ := V
52 count := 1
53 while V₂ ≠ ∅ do
54     pick random v from V₂
55     for w ∈ {x ∈ V | x⁻ = v⁻} do
56         SCC[count] := SCC[count] ∪ {w}
57     end
58     V₂ := V₂ − SCC[count]
59     count := count + 1
60 end
```

Algorithm 25: Common code movement and removal of redundant code

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 V2 := V
 5 count := 1
 6 U := E
 7 H := ∅
 8 EQ := ∅
 9 while U ≠ ∅ do
10     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
11     if a ∈ H then
12         while a ∉ EQ(end) do
13             X := EQ(end)
14             remove_end(EQ)
15             for i ∈ [1, #X] do
16                 remove_end(H)
17             end
18             SCC[count] := X
19             V2 := V2 − SCC[count]
20             count := count + 1
21         end
22     else
23         while EQ ≠ ∅ do
24             X := EQ(end)
25             remove_end(EQ)
26             for i ∈ [1, #X] do
27                 remove_end(H)
28             end
29             SCC[count] := X
30             V2 := V2 − SCC[count]
31             count := count + 1
32         end
33     end
34     if a ∉ V' then
35         V' := V' ∪ {a}
36         H := H + [a]
37         EQ := EQ + [{a}]
38     end
39     if b ∈ V' then
40         if b ∉ H then
41             while b ∉ EQ(end) do
42                 X := EQ(end)
43                 remove_end(EQ)
44                 Y := EQ(end)
45                 remove_end(EQ)
46                 EQ := EQ + [X ∪ Y]
47             end
48             for i ∈ EQ(end) do
49                 i⁻ := a⁻
50             end
51         end
52     else
53         b⁻ := b⁻ ∪ a⁻
54         V' := V' ∪ {b}
55         H := H + [b]
56         EQ := EQ + [{b}]
57     end
58 end
59 while EQ ≠ ∅ do
60     X := EQ(end)
61     remove_end(EQ)
62     for i ∈ [1, #X] do
63         remove_end(H)
64     end
65     SCC[count] := X
66     V2 := V2 − SCC[count]
67     count := count + 1
68 end
69 while V2 ≠ ∅ do
70     pick random v from V2
71     for w ∈ {x ∈ V | x⁻ = v⁻} do
72         SCC[count] := SCC[count] ∪ {w}
73     end
74     V2 := V2 − SCC[count]
75     count := count + 1
76 end
```

Algorithm 26: Evaluate SCC when removing element from EQ

as well. At line 69 in Algorithm 26, EQ is empty and all nodes that have been added EQ before, must have ended up in a SCC and these nodes have been removed from V_2. All nodes that have ever been added to EQ are the nodes in V'. Therefore, at line 69, $V_2 = V - V'$ holds before the first iteration of this loop. The result is that all previous updates to V_2 can be eliminated.

Now, for all $v \in V_2$, $v \notin V'$ holds. Again, the two rules stated before can be used:

1. $\forall v \notin V'(v^- = \{v\})$
2. $\forall v \notin V' \nexists w(v \neq w \wedge v \in w^-)$

As a consequence, the set $\{x \in V | x^- = v^-\}$ evaluates to $\{v\}$.

Finally, the predecessor sets are not used anymore to find SCCs. Therefore, the statements updating these predecessor sets can be eliminated.

The resulting code is shown in Algorithm 27.

The last thing that remains is the function that picks an edge. This function and the modifications that must be made to the code in order to make picking an edge efficient are treated separately. *pick_edge* will be treated as a separate function which has access to all variables found in Algorithm 27. The main reason this is done is because of a practical reason: the entire algorithm would not fit on one page anymore if we did not.

Algorithm 28 shows the function *pick_edge*. It uses a variable M (maximal), which keeps track of nodes with outgoing edges, ordered by $\subseteq$. From this point of view, it is very similar to EQ. When an edge is picked, it starts by trying to find an outgoing edge from the node with the biggest predecessor set. If this is not succesful, it removes this node from M, as it apparently has not got any outgoing edges. It stops when it finds such a node, or when there are no nodes left in M. Then, in the case a maximal node is found, it picks an arbitrary outgoing edge from this node, otherwise is picks an arbitrary edge.

```
   Input: A graph G = (V, E)
 1 forall v ∈ V do
 2     v⁻ := {v}
 3 end
 4 count := 1
 5 U := E
 6 H := ∅
 7 EQ := ∅
 8 while U ≠ ∅ do
 9     Pick an edge (a, b) ∈ U with |a⁻| maximal and preferably a ∈ V'
10     if a ∈ H then
11         while a ∉ EQ(end) do
12             X := EQ(end)
13             remove_end(EQ)
14             for i ∈ [1, #X] do
15                 remove_end(H)
16             end
17             SCC[count] := X
18             count := count + 1
19         end
20     else
21         while EQ ≠ ∅ do
22             X := EQ(end)
23             remove_end(EQ)
24             for i ∈ [1, #X] do
25                 remove_end(H)
26             end
27             SCC[count] := X
28             count := count + 1
29         end
30     end
31     if a ∉ V' then
32         V' := V' ∪ {a}
33         H := H + [a]
34         EQ := EQ + [{a}]
35     end
36     if b ∈ V' then
37         if b ∉ H then
38             while b ∉ EQ(end) do
39                 X := EQ(end)
40                 remove_end(EQ)
41                 Y := EQ(end)
42                 remove_end(EQ)
43                 EQ := EQ + [X ∪ Y]
44             end
45         end
46     else
47         V' := V' ∪ {b}
48         H := H + [b]
49         EQ := EQ + [{b}]
50     end
51 end
52 while EQ ≠ ∅ do
53     X := EQ(end)
54     remove_end(EQ)
55     for i ∈ [1, #X] do
56         remove_end(H)
57     end
58     SCC[count] := X
59     count := count + 1
60 end
61 V2 := V − V'
62 while V2 ≠ ∅ do
63     pick random v from V2
64     SCC[count] := {v}
65     V2 := V2 − SCC[count]
66     count := count + 1
67 end
```

Algorithm 27: Calculation of SCCs for nodes from $V - V'$

M contains all nodes from V' that potentially have an outgoing edge, ordered by $\subseteq$. Therefore, it can fulfill the condition "pick edge with $|a^-|$ maximal and preferably $a \in V'$".

```
1  function pick_edge
2  begin
3      i := #M
4      while i ≠ 0∧ ∄x ∈ V((M(i), x) ∈ U) do
5          remove_end(M)
6          i := i − 1
7      end
8      if i = 0 then
9          (a, b) :=arbitrary U;
10     else
11         S := {(a, b)inU|a = M(i)}
12         (a, b) := arbitrary S
13     end
14     return [a, b]
15 end
```

Algorithm 28: Function that picks the next edge to visit

The original algorithm was impractical to use. Only on very small graphs it would terminate within reasonable time. By using the transformations explained in this chapter, an algorithm is derived whose workings are not obvious, but whose performance is much better than the original algorithm.

Chapter 5

Results and Conclusions

In this chapter the effect of the optimizations on the execution time of the algorithm will be considered. Next, the properties of the optimized algorithm are compared to Tarjan's algorithm. Finally, the achievements made in this paper are reviewed, and some shortcomings will be considered.

5.1 Results

In order to assess the effectiveness of the proposed transformation, the initial program (Algorithm 10 is compared to the derived program (Algorithm 27). These programs are implemented in SETL[1], a set oriented language. The initial algorithm can be found in Appendix A, the optimized version in Appendix B. We could not find a suitable test set of graphs. Therefore, randomly generated graphs are used. One test consists of a graph with cycles of size 10. Additionaly, 0.25 times the number of edges are added randomly. The other tests consist of graphs which are generated randomly, with the number of edges being $0.25, 0.5, 1, 2$ or 4 times the number of nodes. These numbers will be referred to as the graph density. These graphs are used as input to both al-

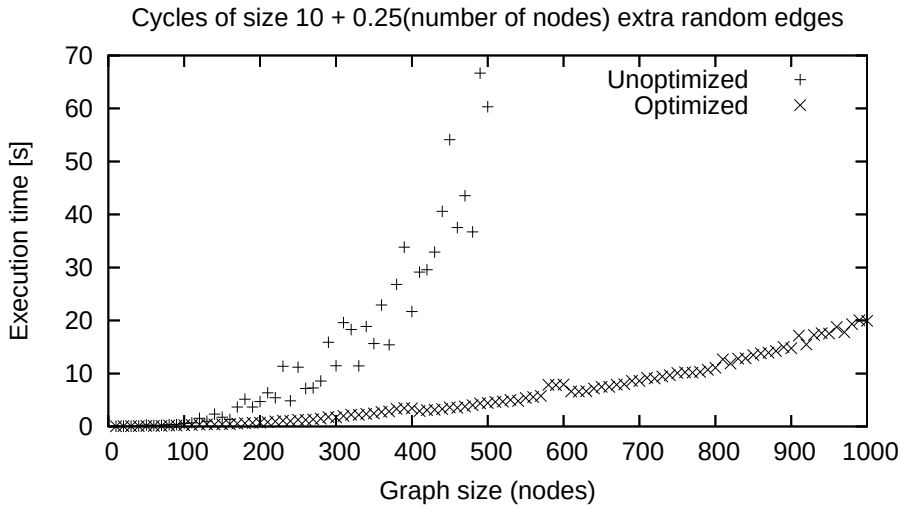

Figure 5.1: Graphs with cycles of size 10, with extra edges added randomly $(0.25 * (number of nodes))$

gorithms and their execution time is measured. The graphs have sizes ranging from 10 to 1000 nodes. For various different graph densities, the graph size is plotted against execution time for both algorithms.

The results of the benchmarks are shown in Figure 5.1, 5.2, 5.4, 5.5 and 5.6. All figures show that the optimized version performs better. A closer look at Figure 5.1 shows that for larger graphs, the unoptimized algorithm is performing slower and slower. In fact, measurements for graphs larger than 500 nodes have not been made, as execution times exceeded our patience. On the contrary, the optimized algorithm seems to scale much better. It has no difficulty in solving the problem for a graph of 1000 nodes.

From the other figures, it is clear that the optimized especially outperforms the old algorithm as the input graph gets denser. This can be explained by the fact that dense graphs contain more edges, and therefore the predecessor sets tend to be larger on average. This slows down the algorithm as more edges also means that updates of predecessor sets have to be propagated to more nodes on average.

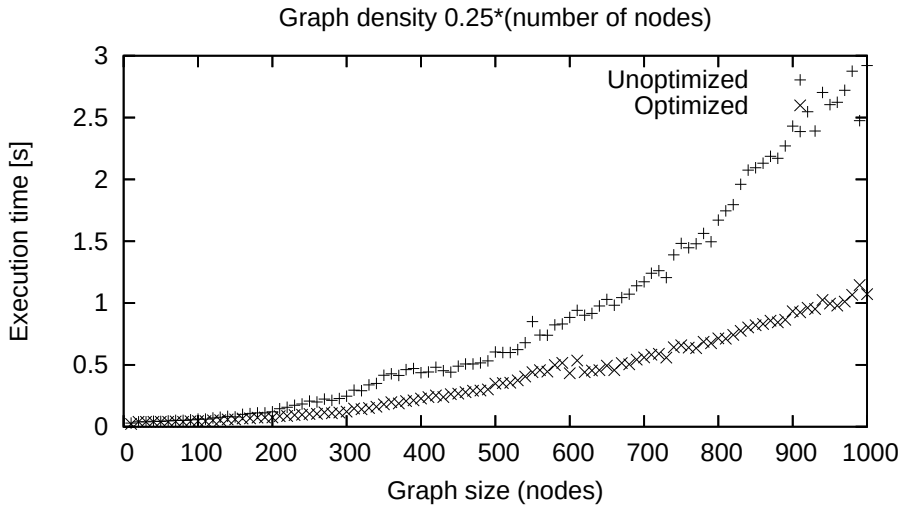

Figure 5.2: Graphs with a density of 0.25

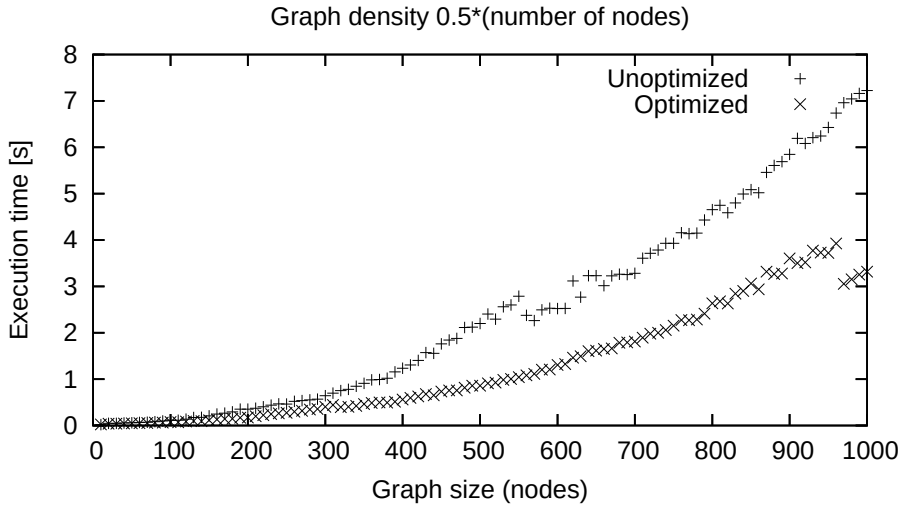

Figure 5.3: Graphs with a density of 0.5

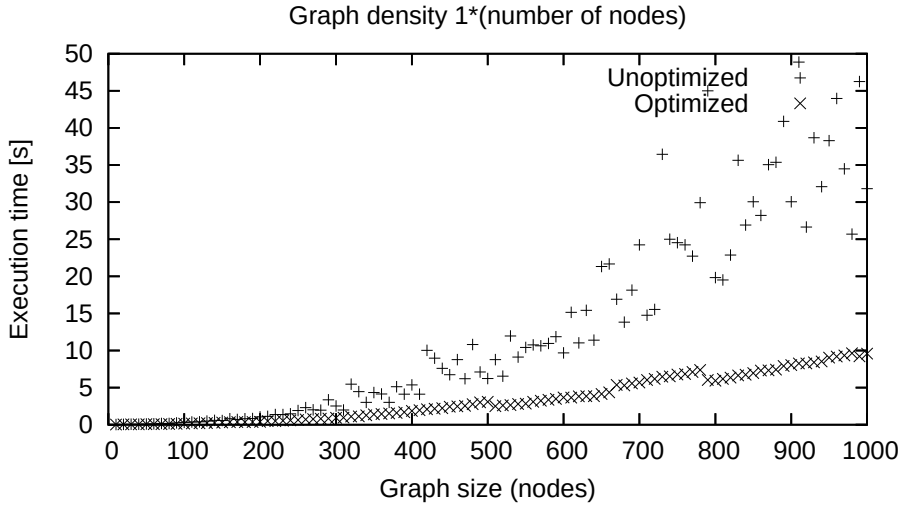

Figure 5.4: Graphs with a density of 1

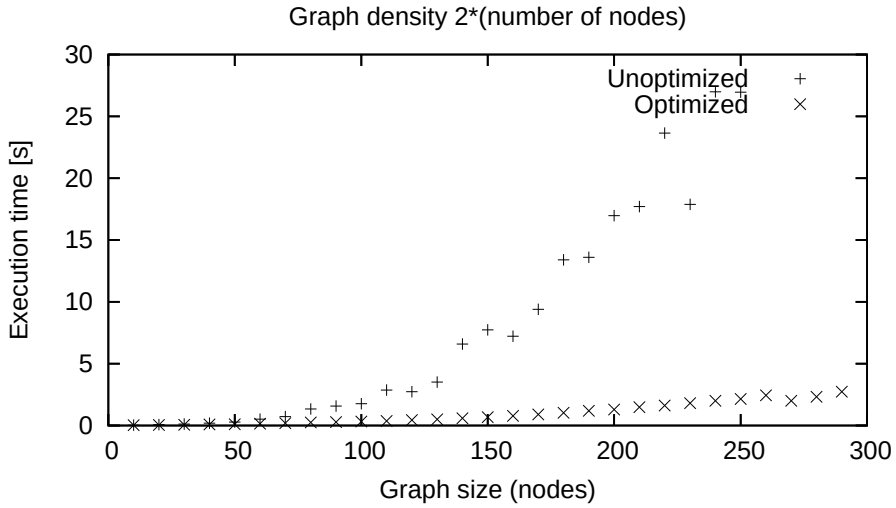

Figure 5.5: Graphs with a density of 2

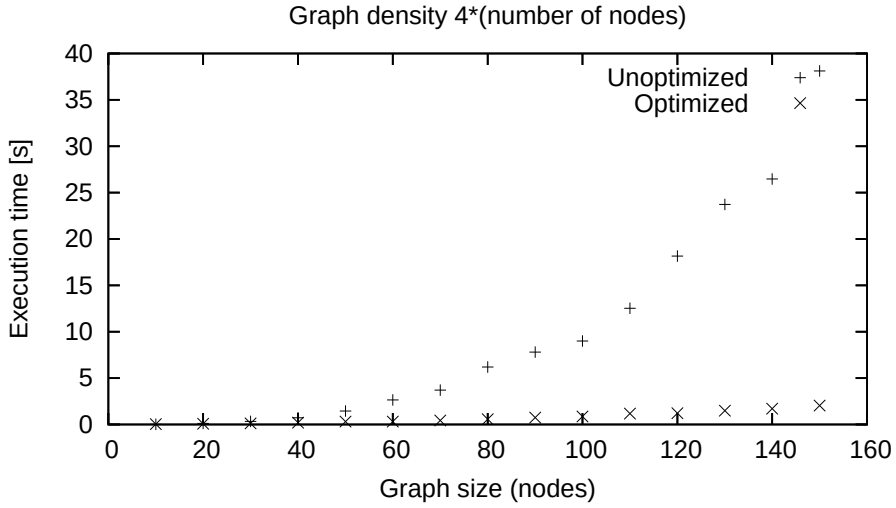

Figure 5.6: Graphs with a density of 4

It can thus be concluded that the optimized version performs better on these random examples. These results are influenced by the way SETL evaluates expressions. Therefore, by using different underlying data structures, performance could potentially be improved.

5.2 Comparison with Tarjan's Algorithm

Many of the concepts found in the derived algorithm can be linked to the workings of Tarjan's algorithm. Tarjan's algorithm can be found in Algorithm 8 on page 25. The algorithm derived from the simple algorithm is shown in Algorithm 27 on page 60.

In the derived algorithm, the variable EQ is very similar to the stack *roots* combined with the stack $unfinished$. EQ stores sets of nodes which are in the same SCC, whereas *roots* has the same purpose in Tarjan's algorithm. It indicates a position in $unfinished$, and all nodes on top of this node in $unfinished$, until the next root node or the top of the stack is met, belong to the same SCC.

Example $EQ = (\{1, 2, 3\}, \{4\}, \{5, 6\})$ could correspond to

$roots = (1, 4, 5)$ and $unfinished = (1, 2, 3, 4, 5, 6)$ in Tarjan's algorithm. Note however that the nodes on $unfinished$ and $roots$ are ordered by the *preorder* number assigned to them. Nodes within an element of EQ are not ordered. In principle, it is possible to choose any outgoing edge from the last SCC found in EQ. Tarjan's algorithm uses DFS, and would certainly choose an edge from a node that is most recently visited.

The next step considered is the recursive call to dfs in Tarjan's algorithm. This statement is guarded by $w \notin S$. This corresponds to $b \notin V'$ in our algorithm. This causes node b to be added to V', which is similar to S, in that it effectively keeps track of nodes that have been visited. When a node is not visited, our algorithm adds it to the lists H and EQ. As H is ordered by $\subseteq$, this information can be used to pick the node with the biggest predecessor set. H corresponds to unfinished nodes that appear on the call stack of Tarjan's algorithm.

The guard $in\_unfinished[w]$ in Tarjan's algorithm corresponds to $b \notin H$ in our algorithm. Being a member of V' but not of H means that the SCC of such a node is determined, which is tracked in the array $in\_unfinished$ in Tarjan's algorithm.

If a node w is in S and in $in\_unfinished[w]$, then a cycle is found in Tarjan's algorithm and this cycle is collapsed. This is done by popping nodes from $roots$. In our algorithm, this is done by merging the two topmost sets of EQ, until the target node of the edge is found.

Finally, SCCs must be calculated. In Tarjan's algorithm, this is done whenever the DFS of a root node completes. Then, all the nodes on top of this node in the stack $unfinished$ must belong to the same SCC. Our algorithm removes elements from the end of EQ as long as a is not in the last SCC. This is correct because the equivalence of their predecessor sets cannot change anymore. This procedure is very similar to unwinding the call stack in Tarjan's algorithm.

5.3 Conclusions

Traditionally optimizing compilers perform only relatively straightforward optimizations of the generated code. A compiler that optimizes code by using restructuring techniques is potentially more powerful. Another advantage of this approach is that the code written by the programmer can remain simple, which greatly aides in productivity and program correctness.

In this thesis, we have used basic set theory as a basis to write algorithms. Next, we have reviewed existing algorithms that determine the strongly connected components of a directed graph. In Chapter 4, we started with a simple, easy to proof algorithm. By using relatively simple transformations, we ended up with an algorithm which is much more efficient and in fact is quite close to Tarjan's algorithm.

There are some shortcomings of course of this approach. Some of the transformational steps are not properly formalized at this time. It should be regarded as an example of what potentially is possible. We hope this chain of transformations can be caught into a more formal model such that they can be applied to other algorithms as well.

Appendix A

Original algorithm in SETL

```
E := {};

read(nrVertices);

loop
    read(a,b);
    if eof then quit; end;
    E{a} with:= b;
end loop;

V := domain(E) + range(E);
minset := { [a,a]: a in V };
U := E;

while U /= {} loop
    [a,b] := pickEdge(U);
    U less:= [a,b];
    minset{b} := minset{b} + minset{a};
    (for i in { x in V | b in minset{x} })
        minset{i} := minset{i} + minset{a};
    end;
end;

V2 := V;
count := 1;
SCC := [];

(while V2 /= {})
    x from V2;
    SCC(count) := {};
    (for i in { y in V | minset{x} = minset{y} } )
        SCC(count) with:= i;
    end;
```

```
    V2 := V2 - SCC(count);
    count := count + 1;
end;

print(SCC);

proc pickEdge( pick_U );
    [a,b] := arb pick_U;
    return [a,b];
end proc;
```

Appendix B

Optimized algorithm in SETL

```
var Vp;
var V;
var H;
var M;
var U;

E := {};

read(nrVertices);

loop
    read(a,b);
    if eof then quit; end;
    E{a} with:= b;
end loop;

V := domain(E) + range(E);
U := E;
count := 1;
Vp := {};
H := [];
M := [];
EQ := [];
SCC := [];

while U /= {} loop
    [a,b] := pickEdge();
    U less:= [a,b];

    if a in H then
        (while a notin EQ(#EQ))
```

```
            X frome EQ;
            (for i in [ 1 .. #X ])
                tmp frome H;
            end;
            SCC(count) := X;
            count := count + 1;
        end;
    else
        (while EQ /= [])
            X frome EQ;
            (for i in [ 1 .. #X ])
                tmp frome H;
            end;
            SCC(count) := X;
            count := count + 1;
        end;
    end;

    if a notin Vp then
        Vp with:= a;
        H with:= a;
        M with:= a;
        EQ with:= {a};
    end;

    if b in Vp then
        if b in H then
            (while b notin EQ(#EQ) )
                X frome EQ;
                Y frome EQ;
                EQ with:= (X + Y);
            end;
        end;
    else
        Vp with:= b;
        H with:= b;
        M with:= b;
        EQ with:= {b};
    end;
end;

(while EQ /= [])
    X frome EQ;
    (for i in [ 1 .. #X ])
        tmp frome H;
    end;
    SCC(count) := X;
    count := count + 1;
end;
```

```
V2 := V - Vp;
(while V2 /= {})
    x from V2;
    SCC(count) := {x};
    V2 := V2 - SCC(count);
    count := count + 1;
end;

print(SCC);

proc pickEdge();
    i := #M;
    (while i /= 0 and not exists x in V | [M(i),x] in U )
        tmp frome M;
        i := i - 1;
    end;
    if i = 0 then
        [a,b] := arb U;
    else
        S := { [a,b] : [a,b] in U | a = M(i) };
        [a,b] := arb S;
    end;
    return [a,b];
end proc;
```

Bibliography

[1] R.B.K. Dewar. *The SETL Programming Language*. 1973.

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

[3] S.L. Gerhart. Correctness-preserving program transformations. *In conference record of the 2nd ACM symposium on principles of programming languages*, pages 54–66, 1975.

[4] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 583, 1969.

[5] W. McLendon, B. Hendrickson, and S. Plimpton. Finding strongly connected components in parallel in particle transport sweeps. *Technical Report TR01-005, Texas A&M University*, 2001.

[6] K. Mehlhorn. *Data Structures and Algorithms*. Springer-Verlag, 1984.

[7] R. Paige. Research retrospective. *Higher-Order and Symbolic Computation*, 16:7–13, 2003.

[8] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4:402–454, 1982.

[9] J.H. Rief. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[10] J.T. Schwartz. Correct program technology. *Courant Computer Science Report 12*, 1977.

[11] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7:67–71, 1981.

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

[13] R. Tarjan. Algorithm design. *Communications of the ACM*, 30:204–212, 1987.

[14] A. Xie and P.A. Beerel. Implicit enumeration of strongly connected components. *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19, 2000.