Universiteit Leiden
The Netherlands

Opleiding Informatica

Depth First Search Characterizations

Martijn Wester

Supervisors:
Dr. A.W. Laarman and L.T. Vinkhuijzen, MSc

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 22/08/2018

Abstract

Depth First Search (DFS) algorithms are ubiquitous, yet difficult to understand. In this thesis, mathematical characterizations of DFS orders are studied. We provide novel characterizations for preorder and postorder DFS, in order to simplify proving correctness of DFS algorithms. These characterizations happen to be symmetrical. The characterization applies to both connected and disconnected graphs that can be both directed or undirected, making it more general than previous characterizations.

Contents

Chapter 1

Introduction

1.1 Applications of Depth First Search

Depth First Search is a fundamental algorithm in computer science, that has many applications. The order in which DFS visits a graph's vertices is often used by algorithms to solve problems.

Some important uses of DFS are summarized below.

- Tarjan found an algorithm for finding a graph's strongly connected components in linear time which makes use of DFS [12].
- Hopcroft and Karp used DFS for maximum cardinality matching in bipartite graphs [6].
- Eve and Kurki-Suonio found an algorithm based on Tarjan's algorithm to compute the transitive closure of a relation [2].
- Tarjan used DFS in an algorithm to find dominators in directed graphs with a time bound of $O(V \log V + E)$ [13].

Depth First Search is a simple algorithm but difficult to understand. For a long time, people thought DFS could only be computed sequentially [5]. But recently parallel algorithms were discovered, for example:

- Aggarwal provided a $\mathcal{RNC}$ algorithm for constructing a DFS tree for undirected graphs [1].
- Laarman et al. introduced a multi-core nested DFS algorithm to detect accepting cycles with considerable speedups compared to other parallel cycle detection algorithms. [10]

Proving correctness of new DFS algorithms however, is difficult. To aid in this problem, a DFS characterization is very useful, since it brings another perspective on what output DFS produces. When an algorithm satisfies such a characterization, it can be concluded that the algorithm is a correct version of DFS. Especially for proving correctness of parallel algorithms, this can save effort.

1.2 Research Questions

In 2005, Krueger introduced characterizations to verify Lexicographic Breadth-First Search, Lexicographic DFS, Breadth-First Search and DFS [9]. His work about DFS covered preorder DFS for undirected connected graphs. Some questions are still left open, namely: in practice, most graphs are not that simple, and besides preorder, postorder is also often utilized. The different DFS variants are displayed in Table 1.1.

	Undirected sequential	Directed sequential	Undirected parallel	Directed parallel
Preorder	Krueger [9]	See chapter 3	Future work	Future work
Postorder	See chapter 4	See chapter 5	Future work	Future work

Table 1.1: The DFS variants to research.

Apart from preorder undirected sequential DFS, there are no known characterizations for the DFS variants, so questions rise:

"What characterizations can be found and proved to verify correctness of variants of Depth First Search algorithms? And how do these properties affect the form of the characterizations?"

An answer to these questions will be found in this thesis. To each different DFS variant that is researched in this thesis, a chapter has been dedicated, see Table 1.1. In Chapter 6 we compare their characterizations to see what differences the variants yield.

1.3 Contributions

As shown in Table 1.1, we cover different DFS variants in this thesis. We will give a mathematical prove for a characterization for directed preorder DFS. This characterization applies on both connected and disconnected graphs to make it generally useful. A possible characterization for undirected postorder will be given but not proved completely. We will discuss that for directed postorder DFS, a characterization is really difficult to find when based on the vertex order only.

Chapter 2

Background

2.1 Graphs

Graphs are mathematical structures that are used to represent relations between data points. A graph consists of a set of vertices and a set of edges. Vertices represent the data points and edges are connections between them. A concatenation of edges is called a path. The children of a vertex a are the vertices that a has an edge to. Then a is called the parent of these children.
Graphs can represent all kinds of real world data. For example, Figure 2.1 exhibits a social network where the vertices represent persons and edges represent the social connections between them. To discover e.g. the maximum amount of matchings, DFS can be used [11]. That is, forming two person groups in which the persons are connected to eachother. In this example, a perfect matching would be the pairs {Anna, Thomas}, {George, Richard}, and {Lily, Sofia}.

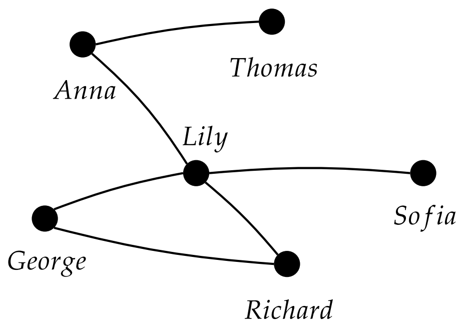


Figure 2.1: Example of a graph exhibiting a social network.

Graphs are also used to represent many other things, such as Euler's bridges of Königsberg, see [4]. And for example, to represent mazes or geographical maps. The edges between vertices can have weights that represent for example distance, time or (fuel) costs.
By using DFS on graphs, many graph problems can be solved. DFS is often used in algorithms for finding connected components, topological sorting, planarity testing and solving mazes or puzzles [7, 8, 12, 14].

There are different kinds of graphs:

- Directed graphs: edges are unidirectional, an edge from *a* to *b* only allows to go from *a* to *b*.
- Undirected graphs: edges are bidirectional, for each edge from *a* to *b*, it is allowed to go from *b* to *a*.
- Connected graphs: for each pair of vertices there is a path between them.
- Disconnected graphs: there are vertices that are unreachable from other vertices.

Combinations of these variants are visualized in Figure 2.2 and Figure 2.3.

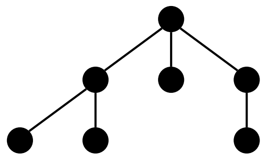


Figure 2.2: Example of a connected undirected graph.

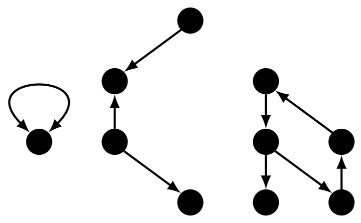


Figure 2.3: Example of a directed disconnected graph.

2.2 Depth First Search

Depth First Search is a greedy algorithm. After choosing a start vertex, the algorithm travels as far as possible into depth before backtracking. An example of an order in which the vertices are visited is illustrated in Figure 2.4. For each split, the algorithm can choose any unexplored path to visit next. So there are multiple orders of visitation possible.

In this thesis we discuss two variants on how DFS numbers the vertices, namely preorder and postorder. With preorder numbering, the vertices are numbered in the same order in which they are visited (as in Figure 2.4). With postorder numbering, a vertex is numbered while backtracking, which means a vertex is numbered if it's subtree is fully explored. An example of a postorder ordering is illustrated in Figure 2.5.

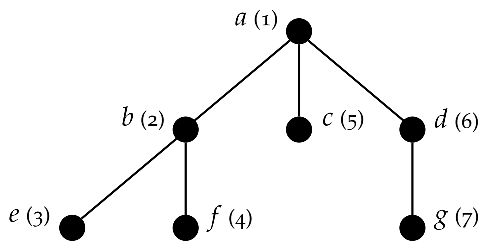


Figure 2.4: An example of DFS's order of visitation / a **preorder** numbering, with *a* as start vertex.

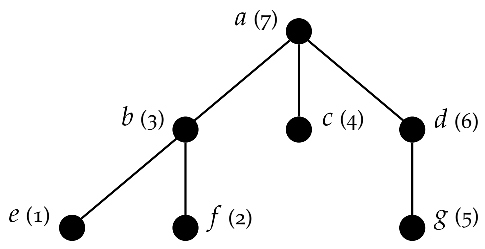


Figure 2.5: An example of a **postorder** numbering, with *a* as start vertex.

2.3 Notation and definitions

In this section, we introduce some terminology we use in this thesis.

Definition 1. *A graph $G \triangleq (V, E)$ consists of a (finite) set of vertices V and a set of edges $E \subseteq V^2$.*

We write vw for $(v, w) \in E$.
We let $E(v) \triangleq \{w | vw\}$,
$E(A) \triangleq \{w | \exists v \in A : vw\}$,
and $n = |V|$ is the number of vertices in V.

Definition 2. *Sequence $\sigma = (v_1, ..., v_g)$ represents an ordering of vertices.*

We let $\forall i : 1 \leq i \leq g$, $\sigma(i) = v_i$ and $\sigma^{-1}(v_i) = i$.
For $\bowtie \in \{<, >, \leq, \geq\}$, let $a \bowtie_\sigma b \triangleq \sigma^{-1}(a) \bowtie \sigma^{-1}(b)$.

Definition 3. *Sequence $S = (v_1, ..., v_m)$ represents a stack of vertices.*

We let $\forall j : 1 \leq j \leq m$, $S(j) = v_j$ and $S^{-1}(v_j) = j$.
For $\bowtie \in \{<, >, \leq, \geq\}$, let $a \bowtie_S b \triangleq S^{-1}(a) \bowtie S^{-1}(b)$.
We write $top(S) = S(|S|)$ as the top element of the stack.

Sequences σ, S are interpreted dually as sets:
$\sigma = \{v_1, ..., v_g\}$
$S = \{v_1, ..., v_m\}$

Hoare logic

We use Hoare logic [3] as a formal proof system to prove the correctness of the algorithms. With Hoare logic, we make use of Hoare triples. A triple consists of $\{P\}C\{Q\}$ where P is an assertion about the precondition and Q is an assertion about the postcondition of programming code C. When P is satisfied, C can be executed resulting in Q. In other words, for every C that is executed in the algorithm, if the precondition P holds then the postcondition holds afterwards.

Proof technique

We use induction over the lines of code and invariants as logic assertions that hold during the entire execution. These assertions are deduced by reasoning over the pre- and postconditions of programming code. For Hoare logic statements, we prove that the precondition is met and contingent on the precondition holding that the postcondition holds.

2.4 DFS Algorithms

Algorithm 1: (Un)directed Pre and Postorder DFS

```
Input: A graph $G \triangleq (V, E)$
Output: An ordering of $\sigma$ of $V$
1  $S \leftarrow \emptyset$;                                  ▷ $S$ starts empty
2  $\sigma \leftarrow \emptyset$;                             ▷ $\sigma$ starts empty
3  while $\exists u \in V \backslash \sigma$ do                ▷ until all vertices are in $\sigma$
4      $S \leftarrow \{u\}$;                                  ▷ select an unnumbered vertex $u$
5      $\sigma(|\sigma|+1) \leftarrow u$;                     ▷ this adds $u$ to the end of $\sigma$ in preorder
6      while $S \neq \emptyset$ do                            ▷ while $S$ is not empty
7          $v \leftarrow top(S)$;                             ▷ peek top of $S$
8          while $E(v) \backslash (\sigma \cup S) \neq \emptyset$ do   ▷ $v$ has unvisited children
9              $v \leftarrow w \in E(v) \backslash (\sigma \cup S)$;  ▷ choose an unvisited child $w$ of $v$
10             $S(|S|+1) \leftarrow v$;                       ▷ push $v$ to top of $S$
11             $\sigma(|\sigma|+1) \leftarrow v$;             ▷ this adds $v$ to the end of $\sigma$ in preorder
12         end while                                          ▷ no more unvisited children
13         $S \leftarrow S \backslash \{v\}$;                 ▷ pop $v$ from top of $S$
14         $\sigma'(|\sigma'|+1) \leftarrow v$;               ▷ this adds $v$ to the end of $\sigma'$ in postorder
15     end while
16 end while
17 return $\sigma$;                                           ▷ return $\sigma$ when every vertex is in $\sigma$
```

Algorithm 1 is a correct DFS algorithm that terminates on a finite set of vertices.

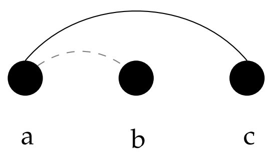
The algorithm differs from Krueger's version [9]. The changes were made to support the different DFS variants but preserve the DFS characteristics. Contrary to Krueger's algorithm which has two loops, the DFS algorithms in this thesis have three loops. The extra outer loop was added to the algorithm to support disconnected graphs.

A preorder ordering is obtained by adding vertices to the ordered sequence σ while creating the DFS tree. This order σ functions dually as a visited set, since each visited vertex is immediately added to σ. The postorder ordering is made while backtracking, and is represented by the ordered sequence σ' in Algorithm 1. The sequence S represents a stack that is used to keep track of which vertices are visited but not fully explored.

The outer loop iterates until every $v \in V$ is in order σ and ensures the algorithm continues after completing one of the connected components. The loop of line 6-15 continues until the current subgraph is added to σ by backtracking when a vertex's children are already in $\sigma \cup S$. The inner loop does the forward search of the DFS algorithm.

2.5 Preorder Undirected DFS by Corneil and Krueger

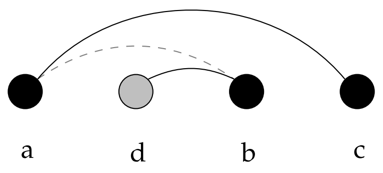
In 2005, Krueger introduced a new perspective on graph searching with DFS [9]. He described a characterization for undirected connected graphs. Krueger's approach on the subject was as follows:



Given an ordering σ of V, where $a <_\sigma b <_\sigma c$. DFS travels greedily through vertices, step by step. When $ac \in E$, but $ab \notin E$, how could DFS have chosen vertex b before vertex c?
The answer to this question lies in characterization (D$_\sigma$), which states roughly that there must be a path between vertex a and vertex b, via this path b could be reached. In the following paragraph, this characterization is explained.

2.5.1 Characterization

(D$_\sigma$):



Given a ordering σ of V, if $a < b < c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d with $a < d < b$ such that $db \in E$.

The characterization satisfies (D$_\sigma$) $\Longleftrightarrow$ $DFS_{UndirPre}()$ of algorithm 1. It describes the nested structure of a DFS tree. For every d between a and c for which $ad \notin E$, the characterization is once again applicable. By applying (D$_\sigma$) repeatedly until $ad \in E$, a path forms from a to b. So every b is reachable from a. Every vertex a can be seen as a vertex with more than one child, since there must be another child on the path to b. For a visualisation of this property, see Figure 2.6.
Each b is added to σ after a but before c. So when every b is not connected to c and is in σ, DFS can backtrack and continue with c as an other child of a. If a b is connected to c, there is no need for backtracking.

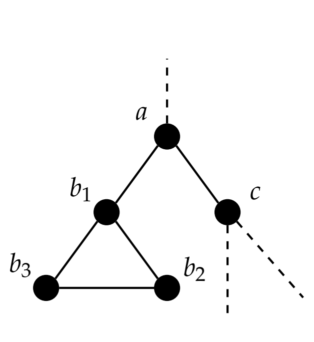


Figure 2.6: Graph example with vertex labels corresponding with those of characterization (D$_\sigma$).

Characterization (D_σ) is applicable on simple undirected connected graphs. In practice, most graphs are not that simple. In our research, we go further on the subject, with the work of Krueger as basis and will provide a mathematical proof for a characterization for preorder DFS orderings of directed possibly disconnected graphs.

Chapter 3

Directed Preorder DFS

Algorithm 2: Directed Preorder DFS

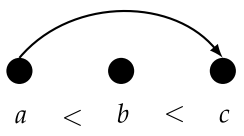
```
Input: A graph $G \triangleq (V, E)$
Output: An ordering of $\sigma$ of $V$
1  $S \leftarrow \emptyset$;  $\sigma \leftarrow \emptyset$;  } atomic                    ▷ $S$ starts empty; $\sigma$ starts empty
2  while $\exists u \in V \backslash \sigma$ do                                    ▷ until all vertices are in $\sigma$
3      $S \leftarrow \{u\}$;  $\sigma(|\sigma| + 1) \leftarrow u$;  } atomic       ▷ select an unnumbered vertex $u$; add $u$ to the end of $\sigma$ in preorder
4      while $S \neq \emptyset$ do                                                  ▷ while $S$ is not empty
5          $v \leftarrow top(S)$;                                                   ▷ peek top of $S$
6          while $E(v) \backslash (\sigma \cup S) \neq \emptyset$ do                ▷ $v$ has unvisited children
7              $v \leftarrow w \in E(v) \backslash (\sigma \cup S)$;                ▷ choose an unvisited child $w$ of $v$
8              $S(|S| + 1) \leftarrow v$;  $\sigma(|\sigma| + 1) \leftarrow v$;  } atomic   ▷ push $v$ to top of $S$; and $v$ to the end of $\sigma$ in preorder
9          end while                                                               ▷ no more unnumbered children
10         $S \leftarrow S \backslash \{v\}$;                                      ▷ pop $v$ from top of $S$
11     end while
12 end while
13 return $\sigma$;                                                                ▷ return $\sigma$ when every vertex is in $\sigma$
```

Algorithm 2 is a correct DFS algorithm that terminates on a finite set of vertices. The code fragments on line 1, line 3 and line 8 are treated as atomic operations to facilitate the proof.

Adding the vertices in the order they are first visited is characteristic to preorder. This is done at line 3 and 8, right after the vertices were added to the stack. The outer loop iterates until every $v \in V$ is in order σ and ensures that the algorithm continues after completing an unconnected subgraph. The loop of line 4-11 continues until the current subgraph is added to σ by backtracking when a vertex's children are already in $\sigma \cup S$. The inner loop does the forward search of the DFS algorithm.

3.1 A directed preorder characterization

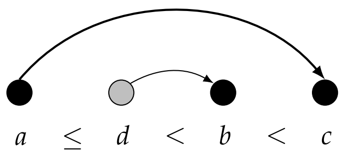
In this section, we explain the intuition behind the correctness of our characterization for the directed preorder DFS algorithm. This characterization was found with a similar approach to the one used in [9], as follows:



Given an ordering σ of V, where $a <_\sigma b <_\sigma c$. DFS travels greedily through vertices, step by step. When $ac \in E$ and vertex b is added to σ between a and c, for some reason b is chosen over c.
How can this b be reached by DFS before c is chosen? The answer to this question lies in characterization (Q$_\sigma$), which states that there must be a path between vertex a and vertex b, so that b could be reached. In the following paragraph, this characterization is explained.

3.1.1 Characterization

Definition 4.
(Q$_\sigma$):



An ordering σ of the nodes V satisfies our characterization (Q$_\sigma$) when:
$\forall a, b, c \in V : a <_\sigma b <_\sigma c \wedge ac \in E \Rightarrow \exists d : a \leq_\sigma d <_\sigma b \wedge db \in E$

Characterization (Q$_\sigma$) is fairly similar to Krueger's undirected preorder characterization (D$_\sigma$). The difference between the undirected and directed characterization obviously is that in our case the edges are directed.

Both characterizations make use of an edge between vertices a and c. This defines the characteristic of DFS to backtrack and choose another path when a subgraph was fully explored. Each vertex b between a and c represent the vertices of a subtree underneath a in the DFS tree. All these vertices are interconnected and reachable from a, which is represented by the vertex d existing left of every b with $db \in E$. Together, all these edges form a path from a to each b. Vertex c represents the first vertex of a new subtree to explore, but can also be a vertex of the current subtree that a has an edge to.

In section 3.2 and 3.3, we will prove that (Q$_\sigma$) is a correct characterization. We will show that $DFS_{DirPre}() \iff$ (Q$_\sigma$) by firstly proving that (Q$_\sigma$) follows from executing algorithm 2, thus $DFS_{DirPre}() \Rightarrow$ (Q$_\sigma$). This is done in section 3.2. Then in section 3.3, the reverse direction: (Q$_\sigma$) $\Rightarrow DFS_{DirPre}()$ is proved, completing the proof for $DFS_{DirPre}() \iff$ (Q$_\sigma$).

3.2 Proof of soundness

In this proof, we prove soundness i.e. $DFS_{DirPre}() \Rightarrow (Q_\sigma)$. In other words, we prove that every order produced by the DFS in algorithm 2 satisfies the characterization (Q_σ).

While doing this, we frequently use the sets σ and $\sigma \backslash S$, which represent the set of visited vertices and the set of backtracked vertices respectively. Namely, each visited vertex is added to σ and S. While backtracking, the vertices are only removed from S. All unvisited vertices will be represented by $V \backslash \sigma$.

To prove characterization (Q_σ) in Theorem 1, a couple of lemmas are used. Lemma 2 and Lemma 3 describe the pre and postconditions of code lines in algorithm 2 with Hoare logic. Lemma 1, Corollary 1 and Lemma 4 describe invariants that are deduced from the code in algorithm 2.

Lemma 1. *$E(\sigma \backslash S) \subseteq \sigma$ is an invariant of algorithm 2.*

Put another way, vertices are removed from the stack only when all their children have been visited.

Proof. The atomic operations on line 3 and line 8 ensure that every vertex that is added to S is also added to σ. Starting with $S = \emptyset$ and $\sigma = \emptyset$ by line 1, since no line removes vertices from σ, every $v \in \sigma \backslash S$ must be removed from S and $S \subseteq \sigma$.

Line 10 is the only line where the vertices v are removed from S. On this line, by the postcondition of the loop on line 6-9, $E(v) \backslash (\sigma \cup S) = \emptyset$ and because $S \subseteq \sigma$, the statement $E(\sigma \backslash S) \subseteq \sigma$ holds. □

Corollary 1. *$\forall a \in \sigma : E(a) \backslash (\sigma \cup S) \neq \emptyset \Rightarrow a \in S$ is an invariant of algorithm 2.*

That is, if a visited vertex has any unvisited children, then it must still be on the stack.

Proof. This can be derived from Lemma 1. When $a \in \sigma \wedge a \notin S$, a must be removed from S on line 10 and thus $a \in \sigma : E(a) \backslash (\sigma \cup S) = \emptyset$ by the postcondition of the loop on line 6-9. This is a contradiction, so $a \in S$. □

Lemma 2. $\{S = \emptyset,\ \sigma' := \sigma,\ E(\sigma) \subseteq \sigma\}$

Line 3:

```
        $S \leftarrow \{u\}$;
        $\sigma(|\sigma| + 1) \leftarrow u$;
```

$\{S = \{u\},\ \sigma = \sigma' + u\}$

As preconditions, S must be empty and there are no edges from the vertices in σ to $V \backslash \sigma$.

Proof. We first prove that the individual preconditions always hold:

- $S = \emptyset$. Initially $S = \emptyset$ from line 1. Just before completion of the outer loop body, $S = \emptyset$ from the postcondition of the loop of line 4-11.
- $E(\sigma) \subseteq \sigma$. From Lemma 1, we have $E(\sigma \backslash S) \subseteq \sigma$. Since $S = \emptyset$, we obtain $E(\sigma) \subseteq \sigma$.

As the postconditions hold trivially, the proof is complete. □

Lemma 3. $\{v' := v = top(S),\ \exists w \in E(v)\backslash(\sigma \cup S)\}$

Line 7-8: $v \leftarrow w \in E(v)\backslash(\sigma \cup S)$;

$S(|S|+1) \leftarrow v$;

$\sigma(|\sigma|+1) \leftarrow v$;

$\{v' = S(|S|-1),\ v = S(|S|) = top(S),\ v'v \in E\}$

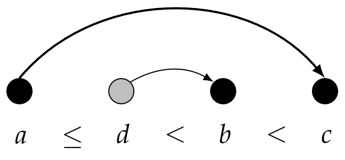
Before line 7-8, $v' = v$ is $top(S)$ and must have an unvisited child. This child is set as v and is placed on $top(S)$.

Proof. The lemma's precondition that $v = top(S)$ holds on loop entry by virtue of line 5, and on reentry by virtue of line 8.
We now prove that the postconditions also hold. Line 7 sets $v \leftarrow w \in E(v')\backslash(\sigma \cup S)$, so there is an edge $v'v$. Then line 8 puts v on the top of the stack, so $v = top(S)$ and $v' = S(|S|-1)$. $\square$

Lemma 4. $\forall v, w \in S : v <_S w \Rightarrow v <_\sigma w$ *is an invariant of algorithm 2.*
The order of vertices in S is the same as in σ.

Proof. Because the atomic operations on line 3 and line 8 assure that every vertex that is added to the top of S is also added to the end of σ, the order of the vertices is the same in both sequences. Line 10 removes vertices from sequence S, but removing vertices preserves the order of S and also preserves $S \subseteq \sigma$.
Since in no other lines, S or σ are modified, the proof is complete. $\square$

Theorem 1.
(Q_σ):



$\forall a, b, c \in V : a <_\sigma b <_\sigma c \wedge ac \in E \Rightarrow \exists d : a \leq_\sigma d <_\sigma b \wedge db \in E$ *is an invariant of algorithm 2.*

Proof. Vertices can be added to σ at line 3 or line 8. Let $\sigma = \sigma' + v$, where σ' is the order before executing these lines and $V\backslash\sigma$ are the remaining vertices in V. Before these lines, we assume σ' already satisfies ($Q_{\sigma'}$). This still holds after adding v to σ. Thus we only have to show each added v satisfies (Q_σ) to prove that σ satisfies (Q_σ). We will show for each of these lines that if $v = b$, the theorem is either satisfied trivially because $\neg(ac \in E)$, or is satisfied because there is an edge $db \in E$.

Line 3 Before a vertex v is added to σ at line 3, the precondition $E(\sigma') \subseteq \sigma'$ holds of Lemma 2.
After adding v to σ', with $v = b$ of characterization (Q_σ), there cannot be an $a \in \sigma : a <_\sigma b <_\sigma c \wedge ac \in E$. Namely, regardless of the rest of the ordering, there are no other edges from a than the edges pointing to σ. Without $ac \in E$, the characterization applies to every v added at line 3.

Line 8 Before $v \in E(v') \backslash (\sigma \cup S)$ is added to σ at line 8, we assume (Q$_{\sigma'}$) holds. With no $ac \in E$, characterization (Q$_\sigma$) applies to every v added at line 8. We will now show (Q$_\sigma$) holds afterwards when there is an edge ac over b.

After v is added to σ, according to Lemma 3, vertex $v = top(S)$ has a parent $v' = S(|S| - 1)$ with $v'v \in E$. Let again be $v = b$, let $v' = d$ and let a be a vertex before v with an edge to a vertex $c \in V \backslash \sigma$. According to Corollary 1, vertex $a \in S$.
With $b = top(S)$ clearly $a <_S b$, and because $d = S(|S| - 1)$, there can be concluded that $a \leq_S d <_S b$ since there is no vertex between $S(|S| - 1)$ and $top(S)$.

Because the order of the vertices in S is the same as in σ (see Lemma 4), the characterization also holds after adding v to σ on line 8, by the existence of vertex $d : a \leq_\sigma d <_\sigma b \wedge db \in E$.

Characterization (Q$_\sigma$) applies on both line 3 and line 8, these are the only lines that alter σ, so Theorem 1 holds. $\square$

3.3 Proof of completeness

In this section, we prove completeness i.e. (Q$_\sigma$) $\Rightarrow$ $DFS_{DirPre}()$. In other words, we prove that every order allowed by (Q$_\sigma$), is indeed obtainable by an execution of algorithm 2. We will do this by showing, algorithm 2 can always add the vertices in this order to σ.

Theorem 2.
Let $\tau = (v_1, ..., v_n)$ be an order of V respecting (Q$_\sigma$). Suppose $\sigma = (v_1, ..., v_{i-1})$ can be produced by execution of algorithm 2. We show that the algorithm can always extend σ with v_i. Then τ can be produced by an execution of algorithm 2.

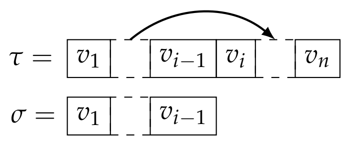


Figure 3.1: We visualize order τ and σ as arrays, where arcs are edges.

Proof. The theorem follows by induction. Vertices can only be added to σ on line 3 or 8, we explain both situations.

Line 3 We have v_i in $V \backslash \sigma$, therefore v_i can be selected as u at line 2. Thus v_i can be added to σ.

Line 8 With $v' = top(S)$, assume $v \in V \backslash \sigma$ with $v'v \in E$ is a vertex that can be selected by the algorithm on line 7, (see Lemma 3). There are two cases:

$v = v_i$ In this case, v_i is added to σ at line 8. So we are done.

$v \neq v_i$ Then after adding v to σ at line 8, we know by Lemma 3 that $v' = S(|S| - 1)$, $v = top(S)$ and edge $v'v \in E$. Also $v' <_\tau v_i <_\tau v$, as also illustrated in Figure 3.2.

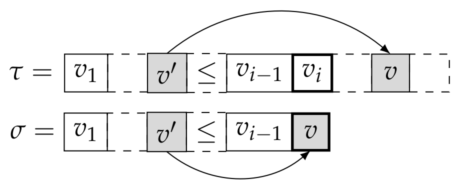


Figure 3.2: Visualisation with edges from v' over v_i in τ, where $v' \leq_\tau v_{i-1}$. Grey vertices are in S.

Applying (Q$_\sigma$) on the triple (v', v_i, v) in τ, gives us the existence of a d so that $v' \leq_\tau d <_\tau v_i$ and $dv_i \in E$. Thus $v' \leq_\sigma d \leq_\sigma v_{i-1}$ in σ and with $dv_i \in E$ and $v_i \in V \backslash \sigma$, therefore vertex $d \in S$ according to Corollary 1.

Since $v' = S(|S| - 1)$, $v = top(S)$ and $v' \leq_\sigma d <_\sigma v$, vertex d must be equal to v', so $v'v_i \in E$ which makes v_i reachable from v'. This tells us that v_i could also have been selected by the algorithm on line 7. Thus vertex v_i could also be added to σ in this case.

For both lines, v_i can be added to σ. There are no other lines where the algorithm adds vertices to σ, so we can say that (Q_σ) holds. $\square$

Conclusion

With $DFS_{DirPre}() \Rightarrow$ (Q_σ) proved in the soundness proof of section 3.2 and (Q_σ) $\Rightarrow DFS_{DirPre}()$ proved in the completeness proof of section 3.3, we can conclude that $DFS_{DirPre}() \iff$ (Q_σ). So characterization (Q_σ) is a correct characterization for directed preorder Depth First Search. This characterization applies for possibly disconnected graphs. $\square$

Chapter 4

Undirected Postorder DFS

Algorithm 3: Undirected Postorder DFS

Input: A graph $G \triangleq (V, E)$

Output: An ordering of σ of V

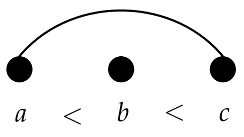
```
1  $\left.\begin{array}{l} S \leftarrow \emptyset; \\ \sigma \leftarrow \emptyset; \end{array}\right\}$ atomic     ▷ S starts empty; σ starts empty
2  while $\exists u \in V \backslash \sigma$ do                  ▷ until all vertices are in σ
3      $S \leftarrow [u \in V \backslash \sigma]$;               ▷ select an unnumbered vertex u
4      while $S \neq \emptyset$ do                        ▷ while S is not empty
5          $v \leftarrow top(S)$;                          ▷ peek top of S
6          while $E(v) \backslash (\sigma \cup S) \neq \emptyset$ do     ▷ v has unvisited children
7              $v \leftarrow w \in E(v) \backslash (\sigma \cup S)$;       ▷ choose an unvisited child w of v
8              $S(|S|+1) \leftarrow v$;                  ▷ push v to top of S
9          end while                                 ▷ no more unvisited children
10         $\left.\begin{array}{l} S \leftarrow S \backslash \{v\}; \\ \sigma(|\sigma|+1) \leftarrow v; \end{array}\right\}$ atomic   ▷ pop v from top of S; and v to the end of σ in postorder
11     end while
12 end while
13 return $\sigma$;                                   ▷ return σ when every vertex is in σ
```

Algorithm 3 is a correct DFS algorithm that terminates on a finite set of vertices. The code fragments on line 1 and line 10 are treated as atomic operations to facilitate the proof.

Adding the vertices while backtracking is characteristic to postorder. This is done at line 10, right after the vertices were removed from the stack. The outer loop iterates until every $v \in V$ is in order σ and ensures that the algorithm continues after completing one of the connected components. The loop of line 4-11 continues until the current connected component is added to σ by backtracking when a vertex's children are already in $\sigma \cup S$. The inner loop does the forward search of the DFS algorithm.

4.1 An undirected postorder characterization

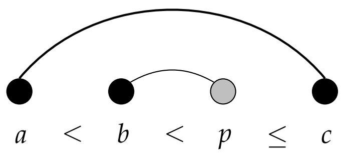
In this section, we explain the intuition behind our characterization for the undirected postorder DFS algorithm. For this characterization, only soundness is proved. We leave the proof of completeness as future work. So we can now only assume this characterization is correct. This characterization was found with a similar approach to the one used in [9], as follows:



Given an ordering σ of V, where $a <_\sigma b <_\sigma c$. DFS travels as far as possible into depth before backtracking. With $ac \in E$, why was c not chosen before a? And how can vertex b be added to σ between a and c?
The answers to these questions lie in characterization (P$_\sigma$), which states that there must be a path between vertex c and vertex b, so that b could be reached. In the following paragraph, this characterization is explained.

4.1.1 Characterization

Definition 5.
(P$_\sigma$):



An ordering σ of the nodes V satisfies our characterization (P$_\sigma$) when:
$\forall a, b, c \in V : a <_\sigma b <_\sigma c \wedge ac \in E \Rightarrow \exists p : b <_\sigma p \leq_\sigma c : bp \in E$

Characterization (P$_\sigma$) is fairly similar to Krueger's undirected preorder characterization (D$_\sigma$) [9]. The only difference between the preorder and postorder characterization is that the order is mirrored.

Both characterizations make use of an edge between vertices a and c. This defines the characteristic of DFS to backtrack and choose another path when a subgraph was fully explored. Each vertex b between a and c represents the vertices of a subtree underneath c in the DFS tree. All these vertices are interconnected and reachable from c, which is represented by the vertex d existing right of every b with $db \in E$. Together, all these edges form a path from c to each b. Vertex a represents the first vertex of a other subtree that is explored, but can also be a vertex of the current subtree that c has an edge to.

We will show that $DFS_{UndirPost}() \Rightarrow$ (P$_\sigma$) by proving that (P$_\sigma$) follows from executing algorithm 3. This is done in section 4.2. In this thesis we do not prove completeness ((P$_\sigma$) $\Rightarrow DFS_{UndirPost}()$) thus the proof is not complete for $DFS_{UndirPost}() \iff$ (P$_\sigma$).

4.2 Proof of soundness

In this proof, we prove soundness i.e. $DFS_{UndirPost}() \Rightarrow (\mathrm{P}_\sigma)$. In other words, we prove that the output of algorithm 4 always satisfies (P_σ). All visited vertices will be in $\sigma \cup S$. All unvisited vertices will be represented by $V \backslash (\sigma \cup S)$.

To prove the soundness of characterization (P_σ) in Theorem 1, a couple of lemmas are used. Lemma 1 and Lemma 2 describe the pre and postconditions of code lines in algorithm 3 with Hoare logic. Lemma 3 and Lemma 4 describe invariants that are deduced from the code in algorithm 3.

Lemma 1. $\{v' := v = top(S),\ \exists w \in E(v) \backslash (\sigma \cup S)\}$

Line 7-8: $v \leftarrow w \in E(v) \backslash (\sigma \cup S)$;

$S(|S|+1) \leftarrow v$;

$\{v' = S(|S|-1),\ v = S(|S|) = top(S),\ v'v \in E\}$

Before line 7-8, $v' = v$ as top of S must have an unvisited child. This child is set as v and is placed on $top(S)$.

Proof. The lemma's precondition that $v = top(S)$ holds on loop entry by virtue of line 5, and on reentry by virtue of line 8.
We now prove that the postconditions also hold. Line 7 sets $v \leftarrow w \in E(v') \backslash (\sigma \cup S)$, so there is an edge $v'v$. Then line 8 puts v on the top of the stack, so $v = top(S)$ and $v' = S(|S|-1)$. □

Lemma 2. $\{v = top(S),\ S' := S,\ \sigma' := \sigma,\ E(v) \subseteq (\sigma \cup S)\}$

Line 10: $S \leftarrow S \backslash \{v\}$;

$\sigma(|\sigma|+1) \leftarrow v$;

$\{S = S' - v,\ \sigma = \sigma' + v\}$

The precondition states that vertex v as $top(S)$ has no unvisited children. The postcondition posits that vertex v is popped from the stack and added to σ.

Proof. We first prove that the individual preconditions always hold:

- $v = top(S)$. This holds by virtue of line 5 or the postcondition of line 7-8 (see Lemma 1).
- $E(v) \subseteq (\sigma \cup S)$. This holds because of the postcondition of the inner loop (line 6-9).

As the postconditions can easily be seen to follow from the preconditions, the proof is complete. □

Lemma 3. *$\forall a \in \sigma : E(a) \subseteq \sigma \cup S$ is an invariant of algorithm 3.*
Put another way, vertices are added to σ only after all their children have been visited.

Proof. Starting with $\sigma = \emptyset$ on line 1, the atomic operation on line 10 is the only place where vertices are added to σ. For line 10, the lemma holds because it is the postcondition of the loop on line 6-9, since it is the negation of the loop condition on line 6. $\square$

Lemma 4. *$\forall v, w \in S : S^{-1}(v) = S^{-1}(w) - 1 \Rightarrow vw \in E$ is an invariant of algorithm 3.*
All stacked vertices are interconnected by the vertices directly beneath them, as visualised in figure 4.1.
In other words, the stack is a path.

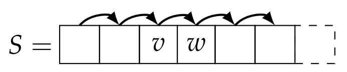


Figure 4.1: We visualize order S as an array, where arcs are edges.

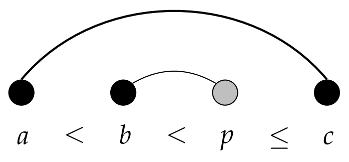
Proof. Vertices can be added to S at line 3 or line 8. We will show for each of these lines that the statement is satisfied.

Line 3 Before this line $S = \emptyset$, initially from line 1 and just before completion of the outer loop body $S = \emptyset$ from the postcondition of the loop of line 4-11.
So, for every vertex u added to S on line 3, there is no vertex v before w in S because $S = \emptyset$ before line 3, so the statement holds vacuously.

Line 8 After a v is added to S on line 8, the vertex v' just below v on the stack has an edge to v (see Lemma 1). So for every vertex $v = S(|S|)$ added to S, the vertex $v' = S(|S| - 1)$ underneath w has an edge to w. Thus, for this line the statement also holds.

The lemma holds for every line where vertices are added to S. Removing vertices from the top of the stack as is done at line 10 does not affect the assertion. So the lemma holds. $\square$

Theorem 1.
(P$_\sigma$):



$\forall a, b, c \in V : a <_\sigma b <_\sigma c \wedge ac \in E \Rightarrow \exists p : b <_\sigma p \leq_\sigma c : bp \in E$ is an invariant of algorithm 3.

Proof. Vertices can only be added to σ at line 10, so it suffices to consider this line. Let $\sigma = \sigma' + v$, where σ' is the order before executing these lines, and $V \backslash \sigma$ are the remaining vertices in V. Before line 10, we assume σ' already satisfies (P$_\sigma$). This obviously still holds after adding v to σ. Thus we only have to show each added v satisfies (P$_\sigma$), to prove that σ satisfies (P$_\sigma$).

We will show for line 10 that if $v = b$ in (P$_\sigma$), the theorem is either satisfied trivially because $\neg(ac \in E)$, or is satisfied because there is an edge $db \in E$.

Let $v = b$ of characterization (P$_\sigma$), and b has just been added to σ at line 10. Let a and c be vertices such that $a <_\sigma v <_\sigma c$.
Suppose there is an edge ac over b. Then a is a vertex before b with an edge to a vertex $c \in V \backslash \sigma$, because c is not in σ yet, since b is the last vertex added to σ. According to Lemma 3, vertex $c \in S$, since $c \in E(a)$ and $E(a) \backslash \sigma \subseteq S$.
With $S \neq \emptyset$ after b was removed from S, let p be the new $top(S)$. Then p must be on top of or equal to c in S, i.e. $c \leq_S p$. And since b was on top of p in the stack, the edge pb exists by Lemma 4.

Since $c \leq_S p$, after vertex p is added to σ at line 10, vertex c must be equal to p or vertex c must still be on the stack. This is because the atomic operation on this line is the only place where vertices are removed from the stack, and the removed vertices come from $top(S)$, (see Lemma 2). So in order σ, vertex $p \leq_\sigma c$ since p is added to σ first. So $b <_\sigma p \leq_\sigma c$ and $pb \in E$, thus characterization (P$_\sigma$) holds after adding v to σ on line 10.

In case there is no $ac \in E$, characterization (P$_\sigma$) applies vacuously to every v added at line 10.

Characterization (P$_\sigma$) applies in all cases on line 10. This is the only line that alter σ, so Theorem 1 holds. $\square$

Chapter 5

Directed Postorder DFS

Algorithm 4: Directed Postorder DFS

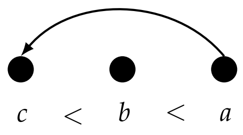
```
   Input: A graph $G \triangleq (V, E)$
   Output: An ordering of $\sigma$ of $V$
1  $S \leftarrow \emptyset$;
2  $\sigma \leftarrow \emptyset$;
3  while $\exists u \in V \backslash \sigma$ do                         ▷ until all vertices are in $\sigma$
4      $S \leftarrow [u \in V \backslash \sigma]$;                    ▷ select an unnumbered vertex $u$
5      while $S \neq \emptyset$ do                                     ▷ while $S$ is not empty
6          $v \leftarrow top(S)$;                                      ▷ peek top of $S$
7          while $E(v) \backslash (\sigma \cup S) \neq \emptyset$ do   ▷ $v$ has unvisited childs
8              $v \leftarrow w \in E(v) \backslash (\sigma \cup S)$;   ▷ an unvisited child $w$
9              $S(|S| + 1) \leftarrow v$;                              ▷ push $v$ to top of $S$
10         end while                                                   ▷ no more unvisited children
11         $S \leftarrow S \backslash \{v\}$;                          ▷ pop $v$ from top of $S$
12         $\sigma(|\sigma| + 1) \leftarrow v$;                        ▷ this adds $v$ to the end of $\sigma$ in postorder
13     end while
14 end while
```

This is a correct DFS algorithm that terminates on a finite set of vertices.

Adding the vertices while backtracking is characteristic to postorder. This is done at line 10, right after the vertices were removed from the stack. The outer loop iterates until every $v \in V$ is in order σ and ensures that the algorithm continues after completing one of the connected components. The loop of line 4-11 continues until the current connected component is added to σ by backtracking when a vertex's children are already in $\sigma \cup S$. The inner loop does the forward search of the DFS algorithm.

5.1 A directed postorder characterization

In this section, the intuition behind a characterization for the directed postorder DFS algorithm is explained. It turns out, it is really difficult to find a characterization when only examining the order that is produced by algorithm 3. We conjecture that a characterization for directed postorder DFS is not even possible. We will explain why. We started with a similar approach to the one used in [9], as follows:



Given an ordering σ of V, where $c <_\sigma b <_\sigma a$. DFS travels as far as possible into depth and then backtracks. With $ac \in E$, how can vertex b be added to σ between a and c? The answer to this question is that there must be a path between vertex a and vertex b, so that b could be reached. In the following paragraph, we will explain what makes it difficult to discern the start of such a path within a directed postorder DFS ordering.

5.1.1 Problems with the directed characterization

When characterizing DFS based on the order it produces, the only tools you have are the existence of vertices, their order and the direction of their edges. To distinguish separate cases, the cases must differ in these. In characterizations (D_σ) and (Q_σ), the property of DFS to continue searching after a subgraph has been explored, is defined by an edge $ac \in E$ going over the vertices b of the subgraph. See Figure 5.1 for an illustration of such a graph with directed edges.

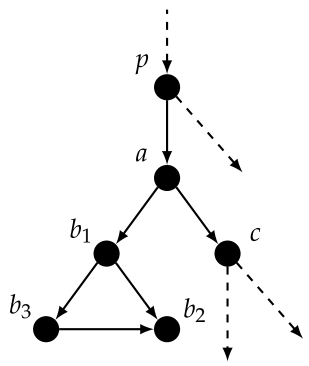


Figure 5.1: A directed graph example with vertex labels corresponding with those of characterization (D_σ) and (Q_σ).

For a directed postorder characterization, splits like the one of vertex a must presumably also be distinguished. Intuitively, these splits would also have a characteristic edge $ac \in E$ going over vertices b in postorder. It however is impossible to distinguish the splitting vertices from other vertices based on their "shape", because every other vertex that is added to σ can have the same appearance, therefore the nested structure of DFS cannot be detected easily. We will show this in the following paragraph.

For every vertex v added to order σ on line 10, $E(v) \subseteq (\sigma \cup S)$ because of the postcondition of the inner loop. So for every v in σ, the edges $E(v)$ go to the left (to a vertex in σ) or to the right (to an already visited vertex in S). Notice that if an edge goes to the right, it must be part of a cycle, since the algorithm already visited this vertex.

For every vertex v that is added to σ, it is possible for v to have an edge to each other vertex already in σ. For example, in Figure 5.1, vertex c could have edges to any vertex that is already numbered. Because $E(c) \subseteq (\sigma \cup S)$ then still applies, these edges do not affect if c can be added to σ. Moreover, any vertex that is added to the end of σ, can have edges to the same vertices as any other vertex in σ has edges to. So the vertices in σ just before a can have the same outgoing edges as a. For example, this applies to the vertices of the subgraph starting at vertex c. The vertices added to σ right after a can also have the same outgoing edges as a. For example, vertex p and his children that were numbered after a.
So vertices added to σ both before and after a splitting vertex a, can have exactly the same "shape" as a itself, therefore vertex a cannot be distinguished. However, the distinction of these splitting vertices was necessary to characterize the preorder DFS characterizations. We presume this is also necessary for characterizing directed postorder.

Concluding, we conjecture that no characterization can be found that holds for all cases of directed postorder DFS, since the nested structure of DFS cannot be distinguished.

Chapter 6

Conclusions

In this thesis, we proved a characterization for directed preorder DFS. A possible characterization for undirected postorder is given but not proved completely. We conjecture that a characterization for directed postorder DFS is probably not possible when based on the order of the vertices only.

All proven characterizations describe the nested structure of a DFS tree. When this property is satisfied by an order of vertices, there is a DFS execution possible that results in this order. In other words, the characterizations assure that all vertices are in the right order to form a DFS tree for each connected subgraph. The characterizations can be helpful for future work with testing correctness of parallel algorithms. There are still subjects unexplored, especially the parallel variants of DFS, as described as future work in Table 1.1 in section 1.2. When examining these DFS variants, the characterizations in this thesis can be a useful starting point.

Bibliography

[1] A. Aggarwal, R. Anderson, and M. Kao. Parallel depth-first search in general directed graphs. *SIAM Journal on Computing*, 19(2):397–409, 1990.

[2] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8(4):303–314, October 1977.

[3] R. W. Floyd. *Assigning Meanings to Programs*. Springer Netherlands, Dordrecht, 1993.

[4] B. George. *Graph Theory, Konigsberg Problem*. Springer International Publishing, Cham, 2017.

[5] John H. Reif. Depth-first search is inherently sequential. 20:229–234, 06 1985.

[6] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[7] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.

[8] A. Jonasson and S. Westerlind. *Genetic algorithms in mazes*. B.s. thesis, School of Computer Science and Communication (CSC), 2016.

[9] R.M. Krueger. *Graph Searching*. PhD thesis, University of Toronto, 2005.

[10] A.W. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. *Multi-core Nested Depth-First Search*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[11] Jang-Ping Sheu, Nan-Ling Kuo, and Gen-Huey Chen. Graph search algorithms and maximum bipartite matching algorithm on the hypercube network model. 13:245–251, 02 1990.

[12] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[13] R. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 03 1974.

[14] R. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, Jun 1976.