Universiteit Leiden
The Netherlands

Towards Parallel Generation of DFS Orders using Verifiers

Yannik Marchand

Supervisors:
dr. Alfons Laarman
Lieuwe Vinkhuijzen, MSc

31/07/2020

Bachelor Thesis

Informatica

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

Abstract

Many algorithms depend on depth-first search (DFS). Yet, this computational problem is difficult to parallelize. In this thesis, we investigate whether we can efficiently generate DFS orders from random permutations of vertices. We show that the efficiency of our algorithm depends on the number of DFS orders in the input graph. To determine on which graphs our algorithm is efficient we empirically investigate the number of DFS orders in undirected graphs. We find that the efficiency of our algorithm depends on the number of edges and the shape of the input graph.

Contents

1 Introduction

For many years, the number of transistors in a processor has been increasing exponentially. This is most famously described in Moore's law [Moo65]. As a consequence, the speed of processors has also been growing exponentially. However, in recent years, this increase has been declining as physical limits are getting closer. Producing even smaller transistors has become both expensive and impractical. An increasing amount of effort is put into other means for increasing the computing power, the most prominent of which involves parallelism: instead of increasing the speed of a *single* core, the *number* of cores is increased. However, most of our algorithms are designed to run on a single core. To make use of multiple cores, we need to design parallel algorithms, which adds considerable complexity to their design and implementation.

A depth-first search (DFS) algorithm systematically traverses all nodes in a graph. Many algorithms for fundamental computational problems use DFS as a subroutine, such as detection of strongly connected components [Tar72] and planarity testing [HT74]. Thus, if we can find an efficient parallel algorithm for DFS, we can apply it to many different problems.

Alok Aggarwal, Richard Anderson and Ming-Yang Kao showed that DFS order generation is in $\mathcal{RNC}$, but compared to sequential DFS algorithms their parallel algorithm is relatively complex. In this thesis, we investigate the efficiency of a very simple algorithm. As explained in Section 2.7, DFS orders can be efficiently verified in parallel. It is also known that random permutations can be efficiently generated in parallel. This thesis therefore asks the following question:

- Can parallel verification be used to efficiently generate DFS orders?

To answer this question, we investigate the following subquestions:

- What is the ratio between the number of DFS orders and the total number of vertex permutations in a graph?
- Can randomization be used together with parallel verification to find a DFS order efficiently?
- Are there subclasses of graphs where the above questions can be answered positively?

As explained in Section 3.2 there is no easy way to count the number of DFS orders in a graph. In this thesis, we therefore investigate the subquestions experimentally, by generating all possible graphs with a certain number of vertices or edges and generating all possible DFS orders in each graph. We also show that recognizing a DFS order on a directed graph is in $\mathcal{NC}$.

We first describe some general concepts in Section 2. In Section 3 we describe our randomized parallel algorithm and the problems we need to solve. Section 3.2 describes the experiments that we performed and which kinds of graph lend themselves for our parallel algorithm. We then give a rough overview of existing literature in Section 4. Finally, we draw a conclusion in Section 5.

2 Background

This section describes general concepts about parallel computing and depth-first search.

2.1 Problems and Algorithms

The primary goal of computation is to obtain an answer for a problem for a given input string. A computational problem is defined as a binary relation over input and output strings with an arbitrary alphabet. Problems are often classified into categories, among which:

1. *Decision problem*: a problem that maps each input string to either 'yes' or 'no'. For example: given an integer x, is x prime?
2. *Function problem*: a problem that maps each input string to a *single* output string. For example: given an integer x, how many prime numbers exist that are lower than x?
3. *Search problem*: a problem that maps each input string to any number of output strings. For example: given an integer x, find a prime number that is lower than x.

An algorithm describes (in precise steps) *how* a specific problem can be solved within a specific model of computation. In this thesis, we are only interested in parallel algorithms. In the next section, we explain what it means for an algorithm to be parallel.

2.2 Parallel Models of Computation

To define parallel algorithms we need to have an appropriate model of computation. In 1978, Steven Fortune and James Wyllie introduced the parallel random access machine model (PRAM) [FW78]. Like the regular RAM model, the PRAM model provides an unbounded global memory and its programs consist of finite sequences of instructions. However, unlike the regular RAM model, which only describes a single processor, the PRAM model provides an unbounded number of processors, each of which has its own program counter and local memory. The original model that was presented by Fortune and Wyllie allows concurrent reads but rejects any input that leads to concurrent writes. Roughly at the same time however, other parallel models of computation were proposed with a different way of dealing with read/write conflicts [Coo81].

PRAMs are commonly classified into three categories:

- EREW (exclusive read, exclusive write): concurrent reads and writes are not allowed.
- CREW (concurrent read, exclusive write): concurrent reads are allowed, but concurrent writes are rejected. Fortune's PRAM fits into this category.
- CRCW (concurrent read, concurrent write): both concurrent reads and concurrent writes are allowed. The behavior of concurrent writes depends on the model. For example, one model might allow an arbitrary write to succeed, whereas another model might use a priority scheme in which only the write with the highest priority succeeds.

A fourth model could be described as ERCW (exclusive read, concurrent write), but this model is hardly ever considered. For our purposes, we do not have to worry about concurrent reads or writes, because each step in a CRCW algorithm can be simulated in logarithmic time by an EREW algorithm.

2.3 Parallel Algorithms

In general, developing a parallel algorithm is more difficult than developing a sequential algorithm. Because each processor operates independently, one has to divide the problem into independent units of work (subproblems). At the same time, it is important to synchronize the processors such that they do not interfere when combining the the solutions of the subproblems. Consider the analogy of building a house. Increasing the number of simultaneously active workers could speed up the process, but at some point they will start laying stones at the same place at the same time, ruining the end-result even if this happens only once. There is a limit to the amount of parallelism that can be introduced here.

The amount of work that is performed by a parallel algorithm is generally described by the product of the number of processors and its runtime. The amount of work that is required to solve a problem in parallel is always greater than or equal to the amount of work that is required to solve it sequentially, as otherwise this would imply the existance of a faster sequential algorithm. Each parallel algorithm can be simulated in a sequential manner without increasing the amount of work after all: simply execute each step of the algorithm for each processor one by one.

Thus, the primary goal of a parallel algorithm is not to reduce the amount of work, but to reduce the amount of time. Unfortunately, not all problems can be spread across multiple processors efficiently. The primary reason for this is that steps that are taken later in the sequential algorithm may be dependent on earlier steps. These steps can not be executed in parallel, regardless of the number of processors. Problems that have this property are often called *inherently sequential*.

2.4 Complexity Classes

To analyze the complexity of an algorithm, we usually consider the asymptotic behavior of its resource consumption, which for instance relates how much the required time and number of processors increase with respect to the input size. The most commonly used notation for asymptotic complexity is the big $\mathcal{O}$ notation. This notation describes an upper bound to the asymptotic growth of the resource consumption of an algorithm. Consider a function $f(n)$ that describes the resources that are required by an algorithm for input size n, and another function $g(n)$. We say that f is in $\mathcal{O}(g)$ if there exist constant $c > 0$ and $n_0 > 0$, such that for all $n > n_0$ the inequality $0 \leq f(n) \leq cg(n)$ holds. In other words, f is in $\mathcal{O}(g)$ if $f(n)$ is not larger than a constant multiple of $g(n)$ for large enough n. Of course, the goal of an efficient algorithm is to keep $f(n)$ and $g(n)$ as small as possible. Figure 1 illustrates this with a few examples. Even on small input sizes the difference between linear and exponential algorithms is dramatic.

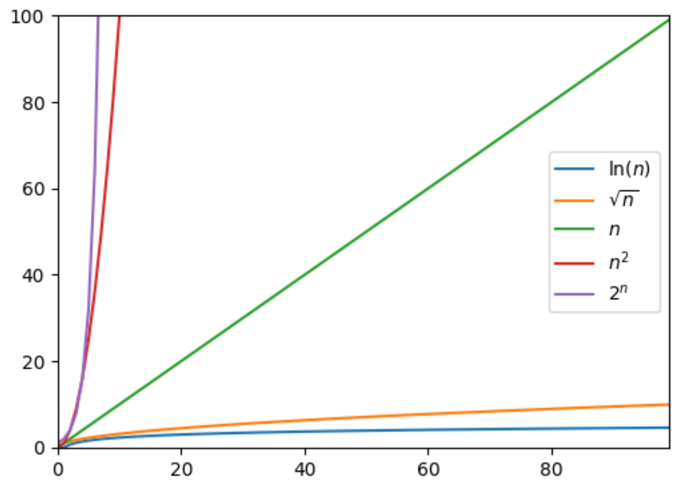

Figure 1: Asymptotic complexity

Now that we have a formal way to describe the complexity of an algorithm we can define some complexity classes:

- $\mathcal{L}$: the class of decision problems that can be solved with $\mathcal{O}(\log n)$ space.
- $\mathcal{P}$: the class of decision problems that can be solved by a sequential algorithm in $\mathcal{O}(n^c)$ time for a constant $c \geq 0$. These problems are generally considered to be efficiently solvable or tractable.
- $\mathcal{NC}$ (Nick's Class): the class of decision problems that can be solved in $\mathcal{O}(\log^k n)$ time with $\mathcal{O}(n^l)$ processors, for a constant $k, l \geq 0$. Similar to $\mathcal{P}$, this class contains the problems that are considered to be efficiently solvable in parallel.
- $\mathcal{RNC}$ (randomized $\mathcal{NC}$): the class of decision problems that can be solved with high probability in $\mathcal{O}(\log^k n)$ time with $\mathcal{O}(n^l)$ processors with access to randomness, for a constant $k, l \geq 0$.

It is known that all problems in $\mathcal{L}$ are also in $\mathcal{NC}$, and all problems in $\mathcal{NC}$ are also in $\mathcal{P}$, but a longstanding open question is whether the reverse is true, which would imply that $\mathcal{L} = \mathcal{NC}$ or $\mathcal{NC} = \mathcal{P}$. The general belief is that this is not the case, but no one has been able to give a proof so far. The relationship between these classes is shown in Figure 2.

Formally, we have defined these complexity classes as a set of decision problems. However, the notion of complexity classes can be applied to other types of problems as well. In the remainder of this thesis we sometimes use the name of a complexity class to refer to a set of function or search problems, rather than decision problems.

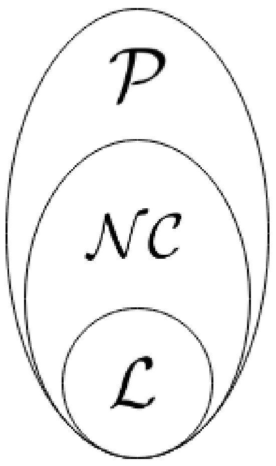

Figure 2: Relation between complexity classes

2.5 Complexity Reductions

To prove that a problem can be solved within a given amount of time as a function of the input size, it is enough to provide an algorithm that solves this problem and prove that it always terminates quickly enough. To prove that an problem can *not* be solved within the given time however, one has to take a different approach. For some problems it is easy to find a lower bound. For example, to find the maximum of a set of n integers each integer has to be examined somehow, so every algorithm that finds the maximum integer in this set performs at least n steps.

For many problems it is difficult to tell if they are in a given complexity class, however. Instead, one can show a relationship between problems by providing a reduction, that is, solving one problem using a hypothetical algorithm that solves another problem as a subroutine.

Consider two problems P and Q that are known to be in $\mathcal{P}$, but we do not know whether they are in $\mathcal{L}$. Imagine an algorithm that solves P with logarithmic space using a hypothetical algorithm that solves Q with logarithmic space. If, at some point, we find an algorithm that solves Q with logarithmic space we now know that we can also solve P with logarithmic space.

This allows us to define another complexity class:

- $\mathcal{P}$-Complete: the set of decision problems in $\mathcal{P}$ that every problem in $\mathcal{P}$ can be reduced to with an $\mathcal{NC}$-reduction. This class contains the decision problems that are known to be not in $\mathcal{NC}$, unless $\mathcal{NC} = \mathcal{P}$.

If someone discovers an $\mathcal{NC}$-reduction from a $\mathcal{P}$-Complete problem to a problem in $\mathcal{NC}$, then we know that all other problems in $\mathcal{P}$ are also solvable in $\mathcal{NC}$, that is, $\mathcal{NC} = \mathcal{P}$. This would mean that all problems in $\mathcal{P}$ are also efficiently parallelizable. However, the general belief is that this is not the case, which implies that all $\mathcal{P}$-Complete problems are inherently sequential. See Figure 3.

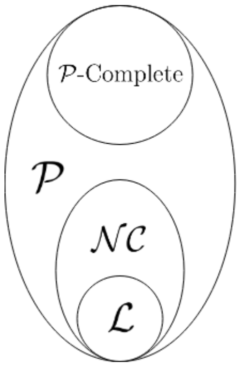

Figure 3: Relation between complexity classes, assuming $\mathcal{NC} \subset \mathcal{P}$

2.6 Depth-First Search

Consider a directed graph $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of edges. A depth-first search algorithm starts at an arbitrary vertex and follows edges until it hits a vertex from which it can only visit vertices that it has visited before (or no vertices at all). At this point the algorithm backtracks until it reaches a vertex from which it can travel to an unvisited vertex again. When the algorithm has visited all vertices that are reachable from the starting vertex it picks an arbitrary unvisited vertex as the new starting vertex and continues the search. This process is repeated until all vertices are visited. As an example, consider the graph in Figure 4. Figure 5 shows the path that a DFS might take through this graph.

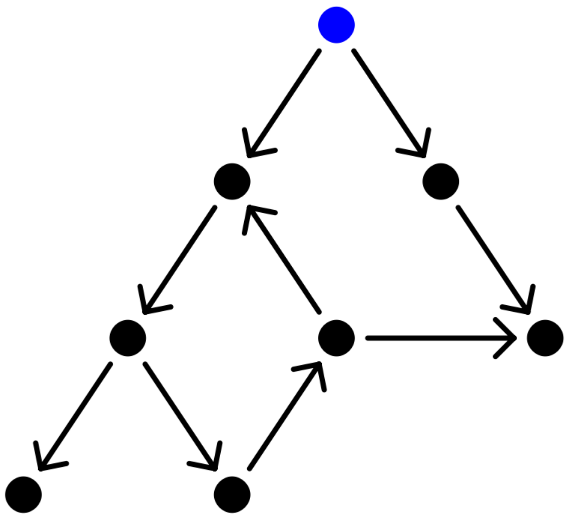

Figure 4: An example graph

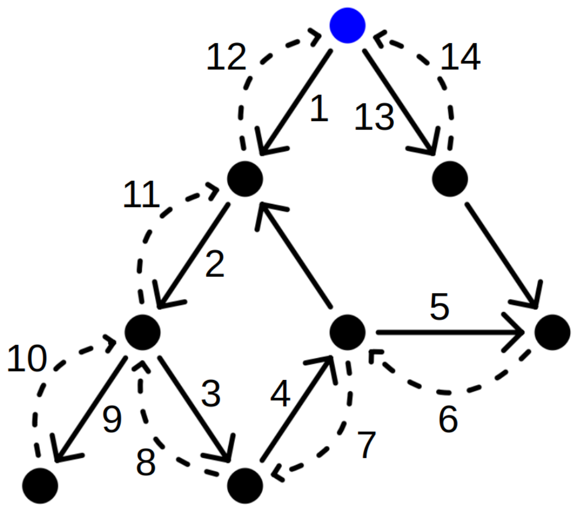

Figure 5: A DFS on the graph of Figure 4

The path in Figure 5 is only one example of a DFS order. Another valid DFS order could (for example) travel along the right edge before the left edge at the vertex that is marked in blue. Most sequential DFS algorithms follow the edges in the graph in the order in which they appear in the adjacency list, but many applications of DFS do not depend on this behavior. We define two variants of the DFS problem:

1. Ordered DFS: given a graph $G = (V, E)$, find the order in which the vertices are visited by a DFS, where the edges are traversed in the order in which they appear in the adjacency lists.

2. Unordered DFS: given a grah $G = (V, E)$, find an order in which the vertices could be visited by a DFS, where the edges can be traversed in any order.

Algorithm 1 implements ordered DFS sequentially and assigns each vertex a number that describes the order in which the vertices are first visited. The stack s contains the vertices that the algorithm needs to process. It has two purposes:

1. At the start of the algorithm all vertices are pushed onto the stack. This ensures that all vertices are processed at least once, even those that are not reachable from the initial starting vertex.

2. When an unvisited vertex is popped from the stack, all of its neighbours are pushed onto the stack. These are the next vertices to be visited. This also allows the algorithm to backtrack because of the 'first in, first out' nature of the stack.

If the graph is implemented with an adjacency list this algorithm requires linear time.

Algorithm 1: Sequential DFS without recursion

```
$s$ = **new** Stack;
$i = 0$; // the dfs order
**foreach** vertex $v$ in $G$ **do**
    $v$.visited = **false**; // mark all vertices as 'unvisited'
    $s$.push($v$); // push all vertices onto the stack
**end**
**while not** $s$.empty() **do**
    $v = s$.pop(); // pop the next vertex from the stack
    // skip vertices that we have visited before
    **if not** ($v$.visited) **then**
        $v$.visited = **true**; // mark vertex as 'visited'
        $v$.index = $i$;
        $i = i + 1$;
        **foreach** vertex $w$ in neighbours($v$) **do**
            $s$.push($w$); // push all neighbours onto the stack
        **end**
    **end**
**end**
```

2.7 A Characterization of DFS

In 2005, Richard Krueger described a simple characterization of unordered DFS in his PhD thesis [Kru05]. His characterization is based on the following idea:

- Given an undirected graph $G = (V, E)$ and an ordering of vertices σ, where $a <_\sigma b <_\sigma c$ and $(a, c) \in E$ but $(a, b) \notin E$. How can b be visited before c in a depth-first search? The answer is simple: there must be a vertex d such that $a <_\sigma d <_\sigma b$ and $(d, b) \in E$.

This is illustrated in Figure 6.

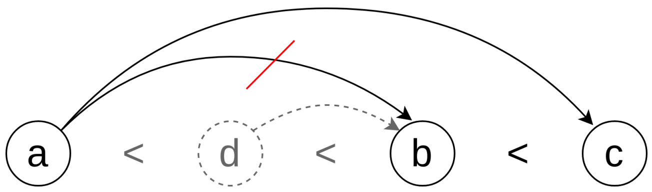

Figure 6: Krueger's DFS characterization

Krueger proved that the characterization works on undirected graphs. In 2018, Martijn Wester proved that the characterization also works on directed graphs [Wes18]. Krueger's characterization can be used to efficiently check if an ordering of vertices σ is a valid DFS order in parallel. In short, σ is a valid DFS order of a graph $G = (V, E)$ if and only if:

$$\forall_{a,b,c}(a <_\sigma b <_\sigma c \land (a, c) \in E \land (a, b) \notin E \rightarrow \exists_d(a <_\sigma d <_\sigma b \land (d, b) \in E))$$

By applying the logical equivalence $(A \rightarrow B) \iff (\neg A \lor B)$ we get:

$$\forall_{a,b,c}(a \geq_\sigma b \lor b \geq_\sigma c \lor (a, c) \notin E \lor (a, b) \in E \lor \exists_d(a <_\sigma d <_\sigma b \land (d, b) \in E))$$

Algorithm 2 is almost a direct translation of the characterization. The algorithm takes a DFS ordering σ and a graph $G = (V, E)$ and returns a boolean that indicates whether σ is a valid DFS ordering of G. The algorithm independently checks for each combination of vertices a, b and c whether the predicate in the universal quantification holds. The result is stored in the temporary boolean array 'validity'. The existential quantification is solved by exhaustively checking if the predicate holds for any vertex d. If at least one such vertex exists the predicate is true for the given a, b and c. Finally, the universal quantification is solved by merging the entries of the temporary array into a single answer.

All loops are executed in parallel. Assuming membership of E can be tested in constant time (for example, if G is provided as an adjacency matrix), the algorithm requires n^4 processors, $\mathcal{O}(1)$ time and $\mathcal{O}(n^3)$ space on a CRCW PRAM.

Algorithm 2: DFS verification algorithm

```
bool validity$[n][n][n]$;
parfor $a, b, c = 1$ to $n$ do
    validity$[a][b][c]$ = false;
end

parfor $a, b, c, d = 1$ to $n$ do
    if $a \geq_\sigma b \vee b \geq_\sigma c \vee (a, c) \notin E \vee (a, b) \in E \vee (a <_\sigma d <_\sigma b \wedge (d, b) \in E)$ then
        validity$[a][b][c]$ = true;
    end
end

bool result = true;
parfor $a, b, c = 1$ to $n$ do
    if $\neg$ validity$[a][b][c]$ then
        result = false;
    end
end

return result;
```

3 Approach

In this thesis, we investigate a simple randomized parallel algorithm that generates a DFS order from a directed graph G. Our algorithm works roughly as follows:

1. Generate a random permutation σ of the vertices of G.
2. Check if σ is a valid DFS order of G.
3. If yes, output σ. Otherwise, go back to Step 1.

The efficiency of our algorithm depends on the efficiency of Step 1 and 2, and on the number of permutations we need to examine on average until a valid DFS order is generated. We know that Step 1 is in $\mathcal{RNC}$. For example, Laurent Alonso and René Schott described an algorithm that generates a random permutation in $O(\log^2(n))$ time with $O(n)$ processors [AS96]. In Section 2.7 we showed that DFS order validation is also in $\mathcal{NC}$. In Section 3.1 and 3.2 we discuss the number of permutations that we need to examine on average.

3.1 Analysis of DFS Order Counts

Consider a graph $G = (V, E)$ and $n = |V|$. We write $\delta(G)$ for the number of different DFS orders that could be produced by an unordered DFS algorithm in G. There are $n!$ different vertex permutations in G. Thus, our algorithm has to examine $\frac{n!}{\delta(G)}$ vertex permutations on average to discover a valid DFS order with high probability. Because we are given a polynomial number of processors we can examine a polynomial number of permutations in parallel. This means that our algorithm is within the resource limits of $\mathcal{RNC}$ if $\frac{n!}{\delta(G)}$ is polynomial, that is, $\frac{n!}{\delta(G)} \leq n^k$ for a constant $k \geq 0$.

It is easy to see that there are graphs that have this property, but it is also easy to see that there are graphs that do not. Consider the graph that only consists of one big cycle (Figure 7). If $n \geq 3$, this graph has exactly n valid DFS orders (one for each starting vertex), so $\frac{n!}{\delta(G)} = \frac{n!}{n} = (n-1)!$, which is obviously not polynomial. In a complete graph on the other hand (Figure 8), any permutation of vertices is also a valid DFS order, so $\frac{n!}{\delta(G)} = \frac{n!}{n!} = 1$.

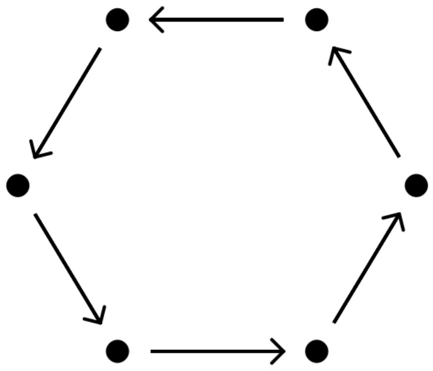
Figure 7: A cyclic graph

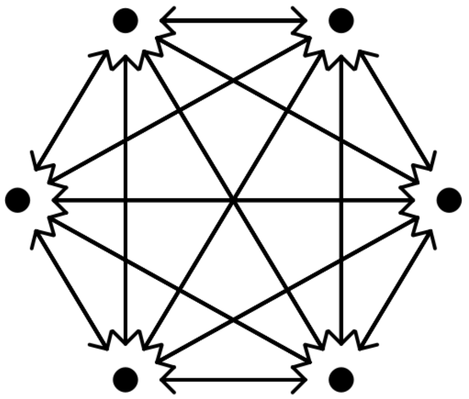
Figure 8: A complete graph

We can see that the efficiency of our algorithm strongly depends on the shape of the graph.

3.2 Empirically Counting DFS Orders

Counting the number of DFS orders in a graph is $\mathcal{NP}$-Hard, even in directed acyclic graphs [KT89], so there is no easy way to determine if a graph has enough different DFS orders for our algorithm to be efficient. In this section we try to find graphs that lend themselves for our algorithm and graphs that do not. To get an intuitive feeling of the number of DFS orders in a graph, we exhaustively generated all undirected graphs with certain properties and counted the number of DFS orders with brute force.

We first generated all possible undirected graphs with $n \leq 7$ vertices and determined the minimum, average and maximum number of DFS orders for a given n. These represent the worst case, average case and best case for our algorithm respectively. The results are shown in Figure 9. Figure 10 shows the ratio between $n!$ and the average number of DFS orders in graphs with n vertices. All numbers are rounded to 6 decimal digits.

Between $n = 4$ and $n = 7$, the ratio between $n!$ and the average number of DFS orders rougly doubles at every step. We suspect that this pattern continues for higher n, which would mean that our algorithm has to try at least an exponential number of DFS orders on average for a given n.

n	min	max	avg	#graphs
0	1	1	1	1
1	1	1	1	1
2	2	2	2	2
3	4	6	4.5	8
4	6	24	10.5	64
5	8	120	26.71875	1024
6	10	720	78.925781	32768
7	12	5040	273.683167	2097152

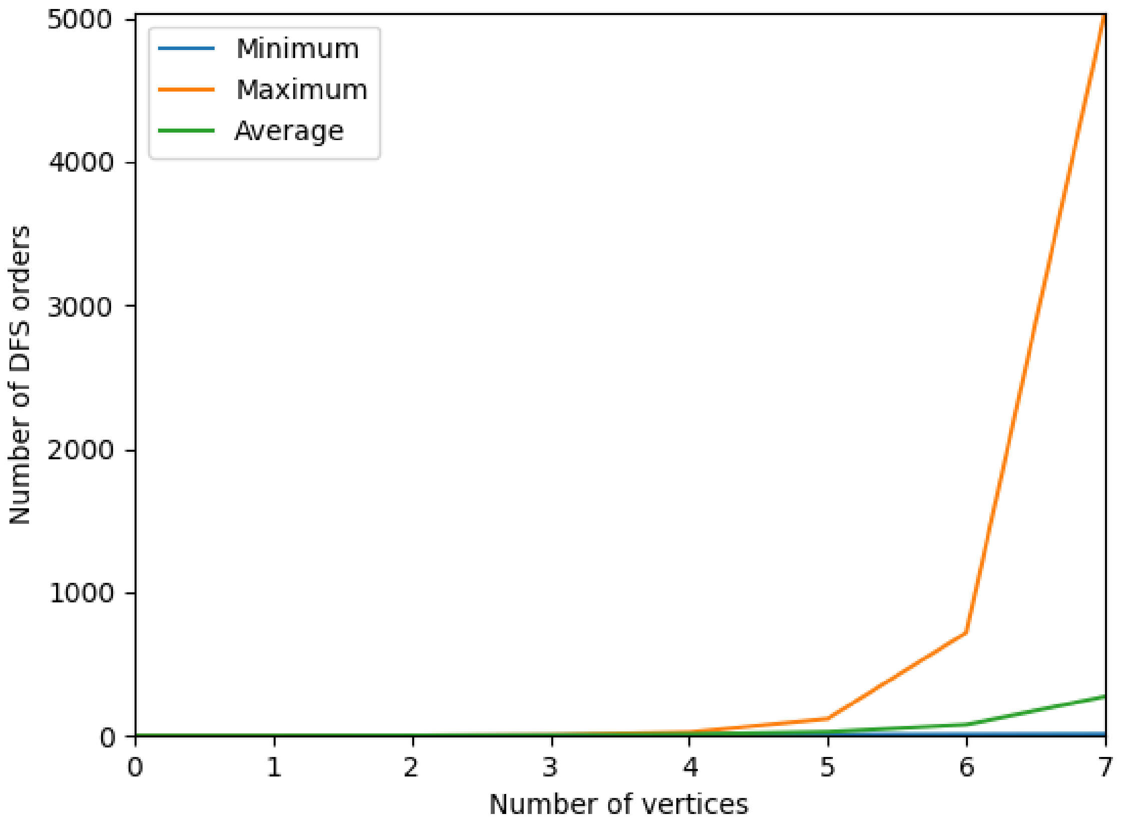

Figure 9: The minimum, maximum and average number of DFS orders for a given n

n	ratio
0	1
1	1
2	1
3	1.333333
4	2.285714
5	4.491228
6	9.122494
7	18.415455

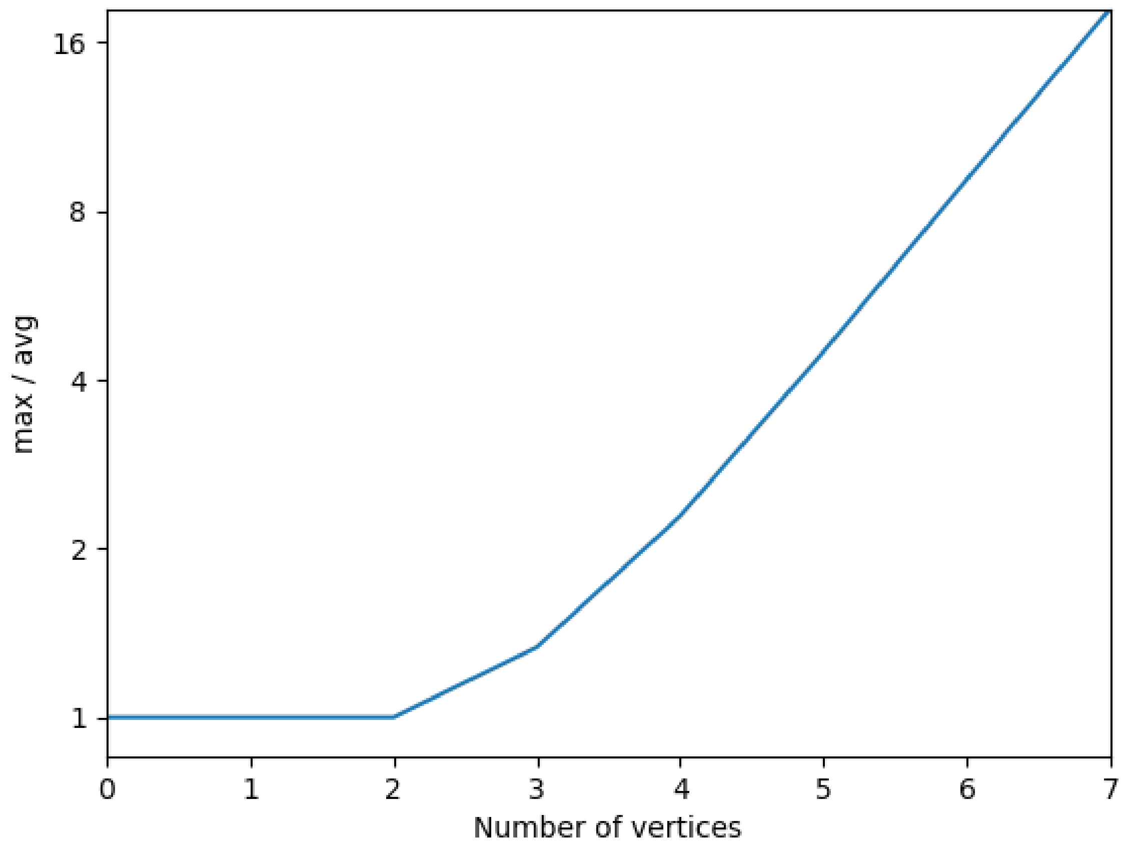

Figure 10: Ratio between $n!$ and avg

We also explored the relationship between the number of edges in an undirected graph and its number of DFS orders. Figure 11, 12 and 13 show the results for $n = 5$, $n = 6$ and $n = 7$ respectively.

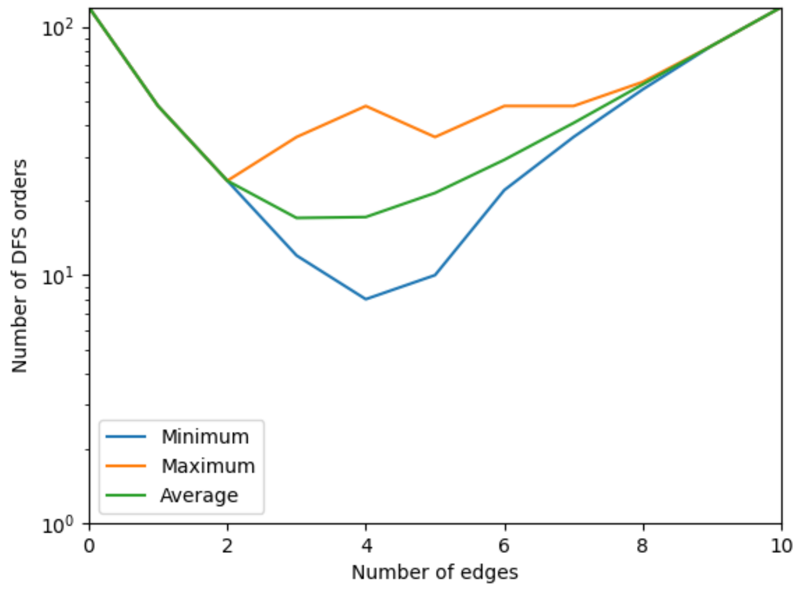

Figure 11: The number of DFS orders for $n = 5$

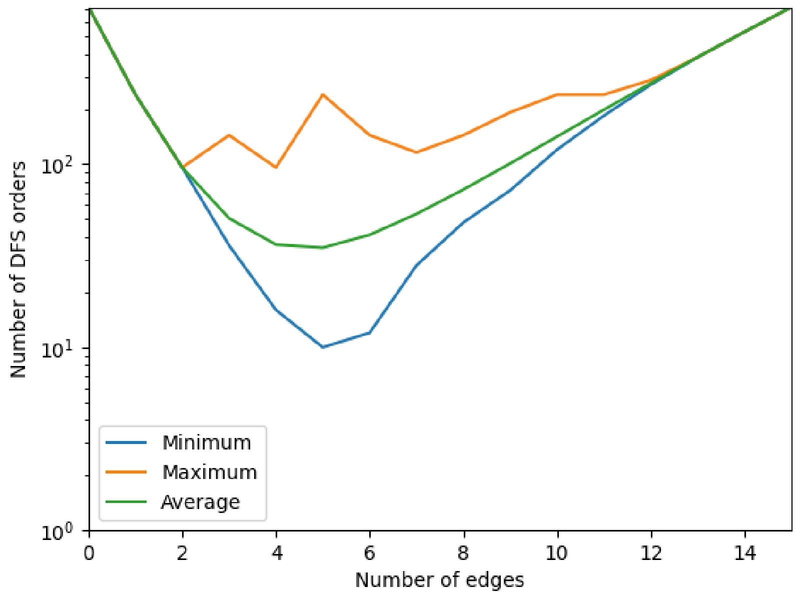

Figure 12: The number of DFS orders for $n = 6$

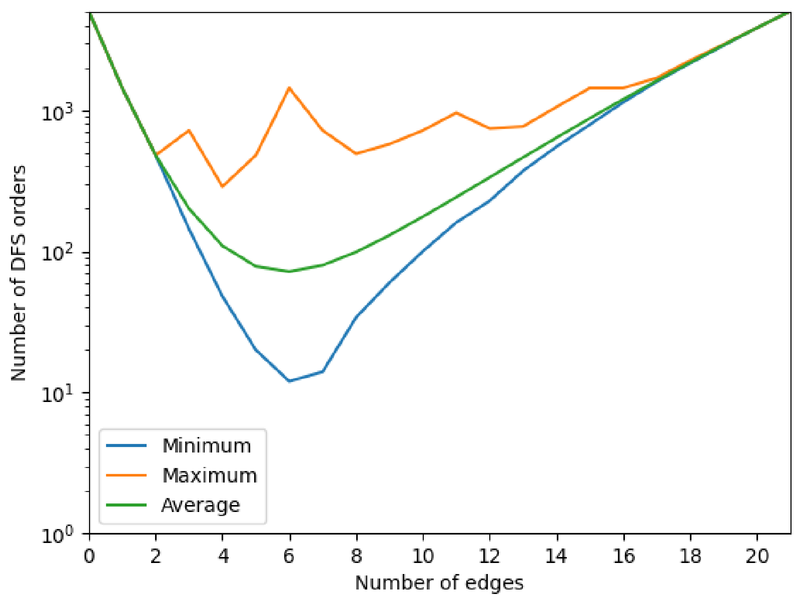

Figure 13: The number of DFS orders for $n = 7$

The orange line (which indicates the maximum number of DFS orders) has an interesting shape. This suggests that some graphs may have significantly more DFS orders than others. We found that the peak at $n - 1$ edges comes from the star graph (Figure 14).

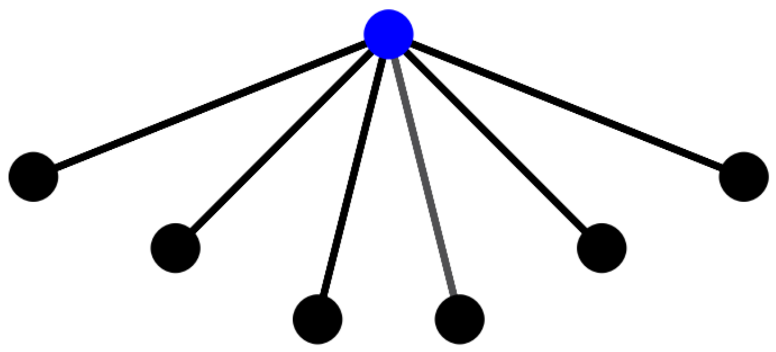

Figure 14: Graph with a high number of DFS orders

If the DFS starts at the blue vertex it can visit the other vertices in any order, which gives $(n-1)!$ different DFS orders. Any edge that is added to this graph imposes restrictions on the order in which the vertices are visited, which reduces the number of possible DFS orders.

We can also see that graphs with a low or high number of edges have significantly more DFS orders than others. It is easy to reason about graphs that have exactly one edge (Figure 15 shows an example). In such a graph, the vertices can be visited in any order in a DFS (regardless of the starting vertex), until the DFS hits one of the two vertices that is connected to the edge. At that point, it is forced to visit the other vertex that is connected to the edge, after which it can visit the remaining vertices in any order. This means that a graph with exactly one edge always has exactly $2(n-1)!$ different DFS orders.

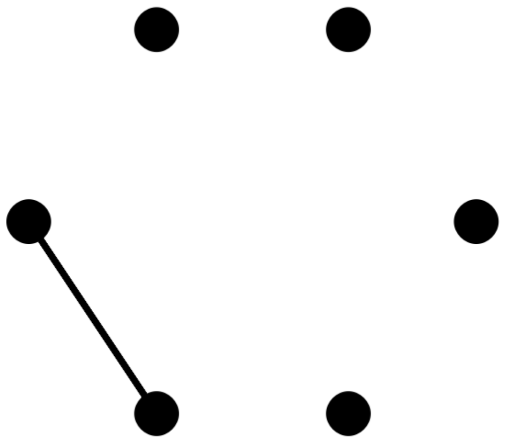

Figure 15: Graph with exactly one edge

Let us take a look at undirected graphs that are almost complete: graphs in which there is only one pair of vertices between which there is not an edge. Figure 16 shows an example. We counted the number of DFS orders in a graph that misses exactly one edge for all $n \leq 12$. The results are in Figure 17.

Note that Figure 17 uses a logarithmic scale. The number of DFS orders seems to grow exponentially, or perhaps even faster. For our purposes, this is a good sign, as it means that we can find a DFS order with a relatively small number of guesses.

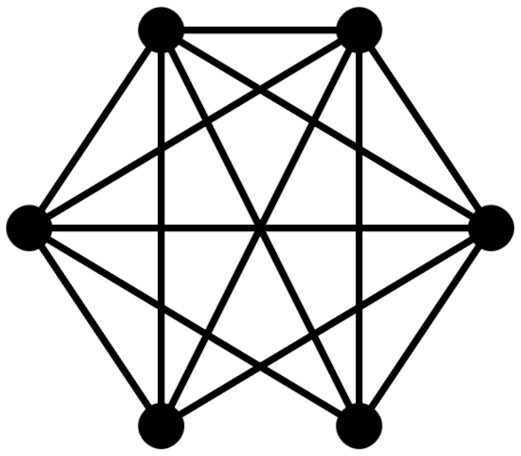

Figure 16: Graph that is almost complete

n	$\delta(G)$
2	2
3	4
4	16
5	84
6	528
7	3840
8	31680
9	292320
10	2983680
11	33384960
12	406425600

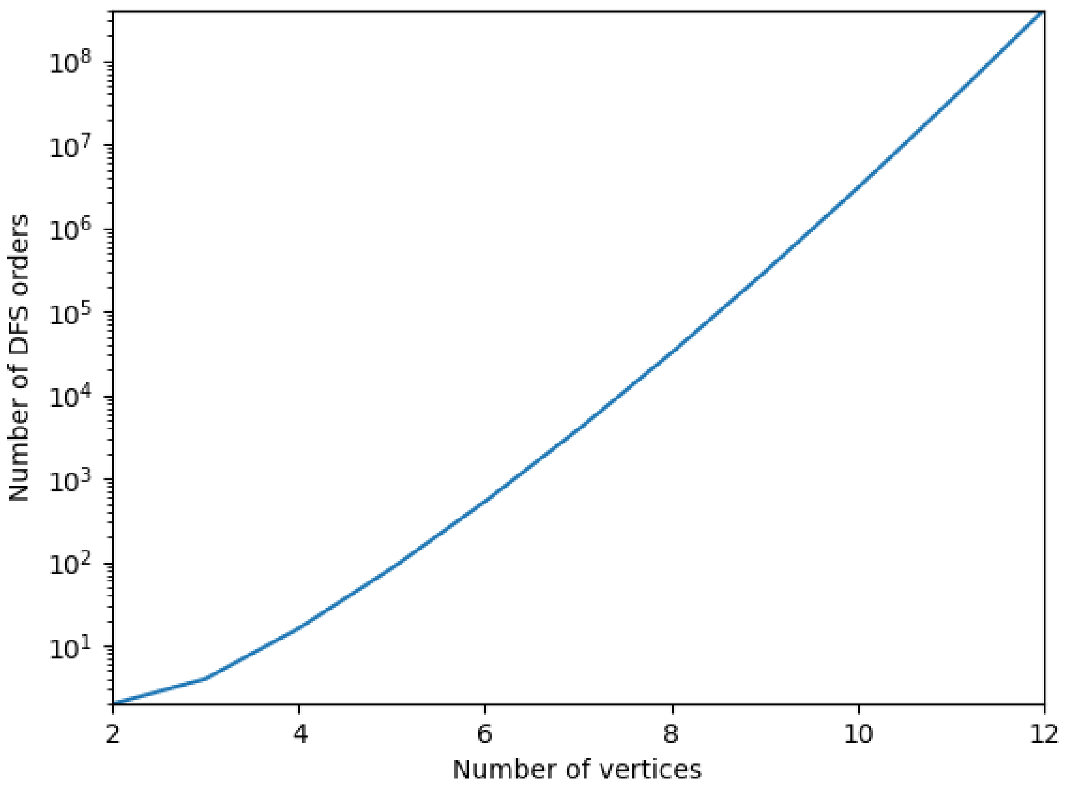

Figure 17: Number of DFS orders in graphs that are almost complete

4 Related Work

This section discusses existing parallel algorithms and insights into DFS. The fastest sequential DFS algorithms (such as Algorithm 1) run in linear time, so a parallel algorithm must run in sublinear time to be useful.

An important insight into the parallelization of ordered DFS was given by John Reif in 1985 [Rei85]. His paper provides a log-space reduction from the circuit value problem (which is $\mathcal{P}$-Complete) to ordered DFS. This implies that ordered DFS itself is also $\mathcal{P}$-Complete, which means that it can probably not be parallelized efficiently (only if $\mathcal{NC} = \mathcal{P}$). Reif has shown that this holds for both directed and undirected graphs.

Whether the unordered version of DFS is $\mathcal{P}$-Complete is an open question, and Reif's discovery does not rule out that DFS in specific types of graphs (for example, acyclic graphs) can be parallelized efficiently. Many advancements have been made in the past decades. To name a few:

- In 1979, James Wyllie gave an $\mathcal{NC}$ algorithm that can construct the preorder, inorder and postorder of a given binary tree [Wyl79]. His algorithm first transforms the tree into a linked list that represents the path of the DFS traversal. Then he constructs the DFS order by assigning a number to each node in the order in which they appear in the linked list.

- In 1986, Justin Smith gave an $\mathcal{NC}$ algorithm that can construct a DFS tree from a planar undirected graph [Smi86].

- In 1988, Ming-Yang Kao showed that finding a DFS forest for a planar directed graph is also in $\mathcal{NC}$ [Kao88].

- In 1987, Alok Aggarwal and Richard Anderson provided an $\mathcal{RNC}$ algorithm that construct a DFS tree of an undirected graph [AA87].

- In 1989, Alok Aggarwal, Richard Anderson and Ming-Yang Kao were able to provide an $\mathcal{RNC}$ algorithm for unordered DFS in general directed graphs [AAK89].

- In 2016, Stephen Fenner, Rohit Gurjar and Thomas Thierauf showed that constructing a DFS tree in in quasi-$\mathcal{NC}$ [FRT19].

To summarize:

- Ordered DFS in general graphs is $\mathcal{P}$-complete, but it is unknown whether this holds for unordered DFS.

- Deterministic $\mathcal{NC}$ algorithms have been developed for DFS in specific types of graph (both ordered and unordered), but so far no one has been able to provide a deterministic $\mathcal{NC}$ algorithm for general directed graphs.

- Randomized $\mathcal{NC}$ algorithms exist for unordered DFS in general directed graphs. However, it is unknown whether these algorithms can be derandomized, that is, whether these algorithms can be turned into a deterministic algorithm.

5 Conclusions and Further Research

In this thesis we investigated an algorithm that generates DFS orders based on random vertex permutations. Compared to other randomized DFS algorithms, our algorithm is simple to understand and implement. However, because it only seems efficient on a small number of graphs its usefulness is limited.

We were only partially able to answer our research questions. We did find graphs on which our algorithm can find a DFS order efficiently. We also found graphs on which it is not efficient. We only considered a small number of different graphs, however. It might be useful to gain more insight into the number of DFS orders in general. For example, how does the number of DFS orders behave on higher n? How does it behave on other types of graphs than the ones we investigated, such as trees or planar graphs? These are all questions whose answer could turn out useful.

References

[AA87] Alok Aggarwal and Richard Anderson. A random nc algorithm for depth first search. *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 1987.

[AAK89] Alok Aggarwal, Richard Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 1989.

[AS96] Laurent Alonso and René Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 1996.

[Coo81] Stephen Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique*, 1981.

[FRT19] Stephen Fenner, Gurjar Rohit, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. *SIAM Journal on Computing*, 2019.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. *Proceedings of the tenth annual ACM symposium on Theory of computing*, 1978.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 1974.

[Kao88] Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in dnc. *Aegean Workshop on Computing*, 1988.

[Kru05] Richard Krueger. Graph searching. *University of Toronto*, 2005.

[KT89] Henry Kierstead and William Trotter. The number of depth-first searches of an ordered set. *Order*, 1989.

[Moo65] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 1965.

[Rei85] John Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 1985.

[Smi86] Justin Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM Journal on Computing*, 1986.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[Wes18] Martijn Wester. Depth first search characterizations. *Leiden Institute of Advanced Computer Science*, 2018.

[Wyl79] James Wyllie. The complexity of parallel computations. *Cornell University*, 1979.