Linköping University | Department of Computer and Information Science
Bachelor thesis, 16 ECTS | Computer Engineering
2019 | LIU-IDA/LITH-EX-G--19/001--SE

# Detecting Cycles in GraphQL Schemas

**Jonas Lind**
**Kieron Soames**

Supervisor : Patrick Lambrix
Examiner : Olaf Hartig

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

## Abstract

GraphQL is a database handling API created by Facebook, that provides an effective alternative to REST-style architectures. GraphQL provides the ability for a client to specify exactly what data it wishes to receive. A problem with GraphQL is that the freedom of creating customized requests allows data to be included several times in the response, growing the response's size exponentially. The thesis contributes to the field of GraphQL analysis by studying the prevalence of simple cycles in GraphQL schemas. We have implemented a locally-run tool and webtool using Tarjan's and Johnson's algorithms, that parses the schemas, creates a directed graph and enumerates all simple cycles in the graph. A collection of schemas was analysed with the tool to collect empirical data. It was found that 39.73 % of the total 2094 schemas contained at least one simple cycle, with the average number of cycles per schema being 4. The runtime was found to be on average 11 milliseconds, most of which consisted of the time for parsing the schemas. It was found that 44 out of the considered schemas could not be enumerated due to containing a staggering amount of simple cycles. It can be concluded that it is possible to test schemas for cyclicity and enumerate all simple cycles in a given schema efficiently.

# Acknowledgments

We would like to thank Olaf Hartig, for the support and opportunity to take part in your research, we hope you find the results as interesting as we did. Another huge thank you goes out to Yun Wan Kim from Toronto for providing the collection of GraphQL schemas. We would also want to thank all friends and family that have supported us in our studies and the writing of this thesis. Finally, thank you to our opponents Daniel Dzabic and Jacob Mårtensson for the insightful critique, and for your patience.

# Contents

**Abstract** **iii**

**Acknowledgments** **iv**

**Contents** **v**

**List of Figures** **vii**

**List of Tables** **viii**

**1 Introduction** **1**
1.1 Background . . . . . 2
1.2 Motivation . . . . . 2
1.3 Research questions . . . . . 3
1.4 Delimitations . . . . . 3

**2 Theory** **4**
2.1 GraphQL . . . . . 4
2.2 Graph Theory . . . . . 7
2.3 Algorithms . . . . . 8

**3 Method** **18**
3.1 Issues to be addressed . . . . . 18
3.2 Comparing the algorithms . . . . . 18
3.3 Creating the cycle detecting tool . . . . . 19
3.4 Web application . . . . . 21
3.5 Testing the implementation . . . . . 23
3.6 Empirical evaluation . . . . . 23

**4 Results** **24**
4.1 Adjustments of the schema converter . . . . . 24
4.2 Runtime . . . . . 24
4.3 Distribution of Types . . . . . 25
4.4 Cycles and SCCs in schemas . . . . . 25

**5 Discussion** **33**
5.1 Implementation . . . . . 33
5.2 Cycles in schemas . . . . . 34
5.3 Evaluation of the Method . . . . . 35

**6 Conclusion** **36**
6.1 The work in a wider context . . . . . 36
6.2 Future work . . . . . 36

**Bibliography** **37**

**A Appendix** **38**

A.1 Git repositories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

# List of Figures

2.1 An example of some of the more important GraphQL schema types. . . . . . . . . 5
2.2 An example of a GraphQL query and the expected result. . . . . . . . . . . . . . . . . 7
2.3 The graph illustrated is an example of a directed multigraph. The edges are illustrated as arrows and are unidirectional, leading from a vertex to another. . . . . . . 7
2.4 An example of a directed multigraph that contains multiple *SCC's*. . . . . . . . . . 8
2.5 An example SCC that contains 2 cycles ($1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$). . . . 13
2.6 An example illustration the operation of the Weinblatt algorithm. . . . . . . . . . . 16

3.1 An example of a GraphQL schema. This schema is the schema for the graph shown in Figure 2.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2 An illustration of how the data is structured when converted to a graph. . . . . . . 21

4.1 Average and median runtime of the cycle detection tool for the collection of schemas. Note that none of the edge case schemas are represented in this graph. . 25
4.2 Runtime for each schema (without the edge case schemas), where the time is divided into the three phases of the cycle detection tool. . . . . . . . . . . . . . . . . . 26
4.3 The average distribution of vertex types in graphs. . . . . . . . . . . . . . . . . . . . 26
4.4 Proportion of schemas that contained cycles versus those that did not. . . . . . . . 27
4.5 The distribution of cycles in non-edge case schemas. Only the schemas that contain cycles are included. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.6 a) The distribution of cycles compared to the number of edges in schemas. b) The distribution of cycles compared to the number of vertices in schemas. . . . . . . . . 29
4.7 The distribution of cycles compared to the ratio between edges and vertices in schemas. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.8 The amount of cycles in schemas compared to the size of the largest SCC in each schema. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.9 Screenshot of the web application before the user have entered a GraphQL schema to be evaluated. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.10 Screenshot of the web application after the user entered a schema. The application evaluated the schema and produced a directed graph and all simple cycles are presented to the user in the leftmost text-area. . . . . . . . . . . . . . . . . . . . . . 32

A.1 An illustration of the general layout of the git repositories, branches, forks and how they are connected for the web application as well as for the locally run tool. . 39

# List of Tables

4.1 Average vertices, edges, average connectivity and the average size of the largest SCC in evaluated schemas, clustered by the number of cycles contained in the schemas. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

# 1 Introduction

Data transfers in a client-to-server relationship have been around since the first websites started to emerge in the 1990s. However, the way clients and servers communicate has been a topic of discussion and research for a long time. In the early 2000s the Representational State Transfer (REST) architectural style, coined by Fielding [4], started to become the most prevalent solution to how developers allowed clients to access websites and is the most common solution right up to the current day. REST-style architectures gave website developers powerful tools that would allow users to request, send or modify data from the server by sending GET, POST, PUT or DELETE requests over the predefined paths. Using a REST-style architecture would allow a client visiting a website to be provided with the information the client needed, when the client needed it. The client is also allowed to interact with the website in more ways than just retrieving information. For example, a client could enter a username and a password into a form to be able to create a profile on a website with persistent data of the client's information.

However, as web services started to increase in scope and with the rise of social media web applications, the amount of data that is sent to and from modern web servers increased drastically. The increase in data traffic would illuminate the limitations of using a REST style architecture, since it uses predefined static paths from the client to the server. In a REST style architecture, a datapoint is a node from which data can be received, e.g. *www.website.com/datapoint-name* where *datapoint-name* is one of the many datapoints that a website potentially could have.

A major limitation with REST style architectures lies in the concept of *over fetching* and *under fetching*. *Over fetching* occurs when the client requests data from a datapoint that in addition to new data that the client needs, provides some data that is already known to the client. The already known data will therefore be sent unnecessarily. This causes the client to be forced to gather a lot of unnecessary data. *Under fetching* occurs when there exist no datapoint that could in one request allow the client to retrieve all wanted data in a single request. This forces the client to request data from several more datapoints in order to access all data. A way to circumvent the problem of over/under fetching is to create more granular and specific datapoints, but at some point, the additional datapoints will become unwieldy and costly for the developers of the web application to construct and maintain.

Facebook realised the limitations of using a REST style architecture for large scale web applications and began their work on their own database handling API. The API was named

GraphQL and was meant to address the shortcomings of REST-style architectures. Every implementation of GraphQL contains a schema that contains information on how all data in the database is related to each other. GraphQL also allows the client to have more control over what data it requests and does this by working as an layer on top of the original database implementation. GraphQL collects all data from the databases, and then only sends the data that the client requested over a single HTTPS request [6]. However, the added freedom of formulating requests, or queries as they are called in GraphQL potentially leads to troublesome behaviour, as described by Hartig and Pérez [5]. A client can request data that, in the schema, is related to other data in a cyclic fashion, allowing the response from the server to include individual data more than once, and therefore it is possible for the response to grow exponentially when the request grows linearly.

There exist countermeasures, but these work more as damage control than trying to eliminate the root cause. Damage mitigation could either be time constrained, where GraphQL aborts the creation of a response if it would take longer than a certain amount of seconds. GraphQL can also abort a response if the response is more than a set number of levels deep, where depth is measured in the amount of references in the schema that have to be made [6]. However an aborted response is always resources spent in vain, especially if a server needs to spend several seconds on a discarded request.

## 1.1 Background

Since its public release in 2016, GraphQL has provided web application and website developers with a flexible client-to-server data transfer API that can easily be implemented on top of existing deployments. The main feature of GraphQL is the ability of the client to dynamically specify what data it wants the server to provide, ensuring that requests would not over or under fetch [6]. The server could also return a lot of data in a response that does not need to be directly related to each other, reducing the amount of requests the client needs to make, further saving some computational and processing resources. Another benefit GraphQL provides is the possibility of both requesting data and modifying other data in the same request, thereby reducing the total number of requests considerably.

## 1.2 Motivation

GraphQL provides significant improvements on how a client requests information from a server, but the possibility of creating queries prompting exponential sized responses is a major issue being studied. However currently there exists no major empirical investigations on how widespread the existence of cycles in GraphQL schemas are, or what schemas typically look like.

Alongside not knowing how widespread the problem of cyclic relationships in GraphQL schemas is [5], there exist at the time of writing no tools that can be used by developers to enumerate cycles in a GraphQL schema. Since no tools for enumerations exist, developers intending to use GraphQL to solve the communication with their database need to be very observant on what types of queries could create exponential responses. Developers also need to keep in mind what relationships in their GraphQL schemas allow cycles to be formed, by keeping track of the database manually.

Manually searching and eliminating cycles or implementing safety checks, is a valid way of working on small scale projects. But on large web applications with massive amounts of data it would be harder to get a good overview of the project and make it troublesome to pinpoint where the problem of cyclic relationships may appear.

## 1.3 Research questions

In this thesis one of the aims is to implement a tool to make it possible for developers to analyse their GraphQL schemas to be able to find cyclic relationships. The tool will be implemented both as a program run locally on a machine, and as a web application allowing schema developers to test their schemas for cyclicity. In the process of implementing this tool it is also intended to provide researchers with empirical data on how widespread the problem of cyclic relationships is. The aims outlined above can be summarized into these three questions:

1. Can a GraphQL schema efficiently be tested for cyclicity?
2. How can all cycles in a GraphQL schema be identified efficiently?
3. What are the characteristics of existing GraphQL schemas in terms of cyclicity and cycles?

## 1.4 Delimitations

Since GraphQL is a large framework, some limitations have to be set on the thesis in order to answer all three research questions.

1. Since the thesis does not have a primary focus on web design, the website will first and foremost be functional serving as the practical implementation of the thesis which will later be available as a tool for developers.
2. Because the focus is on detecting cycles in a GraphQL schema, the tool developed will assume that the syntax of the provided schema is correct and will not evaluate any syntax or other errors beforehand.
3. Since the tool is meant to be implemented in a web application, scripting languages such as JavaScript or Python will be used.

# 2 Theory

In this chapter we will discuss what GraphQL is and what its uses are as well as some of the theoretical concepts we need to take into consideration when developing our cycle detection tool. In the closing sections of the chapter we will introduce several algorithms, one or more of which we will later implement in our tool in subsequent chapters.

## 2.1 GraphQL

GraphQL started as Facebook's internal API for data requesting, manipulation and communication in a client-to-server relationship, GraphQL was developed to work with pre-existing databases such as SQL. It started to be used in operations in 2012 with a release to the general public in 2015 under a technical preview phase. GraphQL was finally available as open source in 2016 after it exited the technical preview phase [6].

### 2.1.1 GraphQL's purpose

The objective of GraphQL is to present an alternative to REST styled architectures for transmitting data in a client-to-server relationship. The main point to improve on is the way a client can request data from the server. Using REST-style interfaces allows a client to send static requests through predefined paths on the server, whereby the server would answer with all the data related to the path requested on. An example of this would be if a client requests information about a user. The client sends a GET request over a predefined path on the server, the server, if possible, responds with a static data structure containing all data related to the user. GraphQL improves on the cases where the client has some of the data it needs and wishes to only request the data it does not already have access to. This case is solved by allowing the client to customise what data it wishes to receive.

GraphQL allows a client to select and retrieve data from multiple data-structures in a single request, by doing a so called sub-selection [6].

The possibility to specify what data a client wishes to receive together with the ability to construct sub-selections allows the client to avoid over or under fetching of data. But by giving the client the ability to customize exactly what it wishes to receive, there exist edge cases where the client can create server responses that can grow exponentially in size by exploiting cyclic relationships in the schema.

### 2.1.2 GraphQL Schema

Every implementation of GraphQL contains a file called a *GraphQL schema*, or schema for short. A schema contains a representation of how data objects that can be accessed via the GraphQL API are related to each other. These relations are built upon the properties of the objects that refer to the other objects, when objects are related it is said that they have a relationship. The schema also contains information on how request of the server that implements a schema resolves the request. A schema can contain several objects that have *relationships*.

```
interface Character {
  id: ID!
  name: String!
  age: Int!
}

type Human implements Character {
  id: ID!
  name: String!
  age: Int!
  height(unit: METER): Float
  ownsCar: [Car]
  work: Occupations
}

type Pet implements Character {
  id: ID!
  name: String!
  age: Int!
  breed: String!
}

type Car {
  id: ID!
  colour: String!
  yearModel: Int!
}

union Friends = Human | Pet

type Query {
  getCharacter(id: ID!): Character
  getHuman(id: ID!): Character
}

enum Occupations {
  Doctor
  Programmer
  Pensioner
}
```

Figure 2.1: An example of some of the more important GraphQL schema types.

#### 2.1.2.1 Types in schema

GraphQL schemas contain several different structures called *types*. Types are the main building blocks of a schema and they define how the data is structured. All types can have several attributes called *fields*. Fields can have standard values such as integers, strings or be a unique ID and can either be a single value or a list of values. They can also be references to types in the schema which is the most important aspect of the schema, in this thesis. The most important types in GraphQL schemas are:

- *Object Types*

  *Object types* are used for the GraphQL implementation to organise what fields belong together, and therefore allow the clients to request more complex data structures than only requesting individual fields. They can contain fields that take arguments; the passing of arguments allows the client to *query* the length of the data in a field and specify if the return value should be in either feet or meters [6]. What queries are will be explained further below. The data in a field could represent the distance between cities. A practical example of an object type is the "Car" type shown in Figure 2.1. The type "Car" has the fields "id", "colour" and "yearModel".

- *Interface Types*

  An *interface* is a type that functions as an umbrella-term for multiple object types that share most attributes and could in most circumstances be considered as the same type. An example of this would be the interface "Character" in Figure 2.1 that is implemented by both the type "Pet" and "Human". A field could therefore reference both "Pet" and "Human" as "Character" since there might be no need to differentiate between them as they share some fields. If only attributes from one of the types are needed it is possible to specify that particular type as any other type.

  All attributes in the interface need to be included in all types that implement the interface. In Figure 2.1 this would include "id", "name" and "age". Using an interface instead of a type does not limit another type's, or another interface's ability to reference neither the types that implement the interface nor the interface itself. It is however, not possible for an interface to implement another interface as of the time of writing.

- *Union Types*

  A *union* allows fields to reference more than one type simultaneously, much like how interfaces work. However, a union does not contain fields of its own and cannot be implemented by object types. Unions are more commonly used to allow a field to reference multiple object types. An example of this is the union "Friends" in Figure 2.1 which combines the two types "Human" and "Pet".

- *Queries and Mutations*

  GraphQL *queries* are sent in a typescript likening JSON (JavaScript Object Notation), but returned in actual JSON. A query is structured in layers with each layer having a root object that encapsulates each layer. The root object in the example query, see Figure 2.2, is "getHuman(id: "1")" that contains all other information requested. In the same example the "ownsCar" field is an example of a *sub-selection* mentioned in Section 2.1.1. A *mutation* operates in much the same way as a queries but with the additional feature of modifying the data stored in a database. This makes it possible to modify the data stored in the database as well as returning the data modified indicating what data was changed [6].

- *Custom defined Types*

  These include types such as a *scalar*. There already are examples of this type such as Integer, Float and Boolean. It is possible to define custom additional types as well to

```
                                  {
{                                   "data" : {
  getHuman(id: "1") {                 "getHuman" : {
    name                                "name" : "John Smith"
    ownsCar {                           "ownsCar" : {
      colour                              "colour" : "Blue"
      yearModel                           "yearModel" : 2018
    }                                   }
  }                                   }
}                                   }
                                  }
```

Figure 2.2: An example of a GraphQL query and the expected result.

be used when the pre-defined types are unsuitable. A special kind of scalars are *enums*, they restrict the possible values of the field to one of a finite number of pre-defined values. A practical use for enums is for example "Occupations" defined in Figure 2.1 and referenced in type "Human". "Occupations" is used to specify which out of the, in this example, three Occupations a "Human" occupies be it "Doctor", "Programmer" or "Pensioner".

## 2.2 Graph Theory

In order to find out if a GraphQL schema contains cyclic relationships and to enumerate all such relationships, one must first create a mathematical structure that can represent the schema and the relationships described in said schema. In graph theory there exists a type of graph that is called a *directed graph*. A directed graph is composed of *vertices* (singular *vertex*). If an edge leads to another vertex, then the edge and vertex are said to be related and if the edge leads from a vertex back to itself, then the edge is called a *loop*.

A vertex contains a loop if an edge originates and terminates in that same vertex, see Figure 2.3 vertex H. Inside a directed graph, all edges are unidirectional, the relationships only go one way. If a directed graph contain vertices that have multiple edges that all lead to the same vertices, the graph is called a directed multigraph according to Astratian et al. [1].

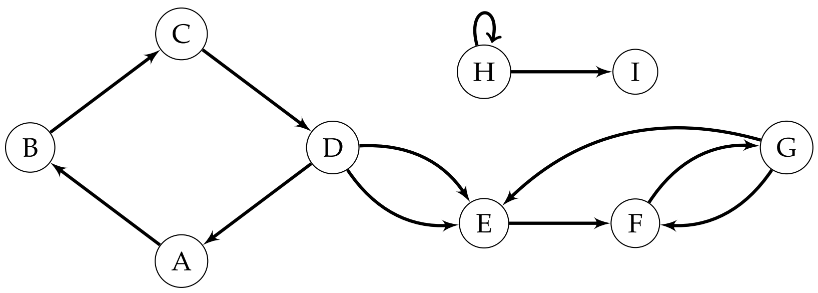


Figure 2.3: The graph illustrated is an example of a directed multigraph. The edges are illustrated as arrows and are unidirectional, leading from a vertex to another.

### 2.2.1 Cycles

Before explaining what *cycles* are, one would first need to define the term *path*. A path in a directed graph can be defined as a sequence of vertices via edges connected to vertices that are distinct from one another. Contained in a directed graph, a cycle is a path that starts from the same vertex in which it ends. Formally, a path $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$ with $n \geqslant 1$ is a cycle if $v_1 = v_n$. as explained by Asratian et al. [1]. A *simple cycle* is defined as a cycle where no vertex except the start and end vertex may be present more than once. The simple cycles in Figure 2.3 are:

- $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$
- $F \rightarrow G \rightarrow F$
- $E \rightarrow F \rightarrow G \rightarrow E$
- $H \rightarrow H$

Note that there are also cycles that do not satisfy the requirement of simple cycles. Such a cycle could be formed by: $E \rightarrow F \rightarrow G \rightarrow F \rightarrow G \rightarrow E$. But since the requirement of simple cycles is that no vertex except the start/end vertex exists more than once in the cycle, they will not be considered when studying simple cycle. Every non-simple cycle may be broken down into simple cycles.

### 2.2.2 Strongly connected components

A Strongly Connected Component *SCC* can be defined as a *sub-graph* of a graph *G* where every vertex *V* in *G* can reach all other vertices $V_n$ in the same SCC, for all vertices in the SCC. The directed multigraph illustrated in Figure 2.4 can be divided into these SCC's:

- { A, B, C }
- { D, E }
- { F }

Even if D is reachable from C, there is no path that leads back to C going from vertex D. Therefore, C and D are, by definition, not part of the same SCC. Furthermore, since F does not have any edges leading to other vertices, it is considered its own SCC according to Tarjan [10]. The reason for studying SCC's when searching for simple cycles is that the definition of SCC indicates the existence of at least one simple cycle. Since the definition of SCC demands that all vertices are reachable from an arbitrary vertex inside a SCC, there cannot exist cycles that span from a SCC to another SCC. Resources spent searching for such cycles are therefore spent in vain.

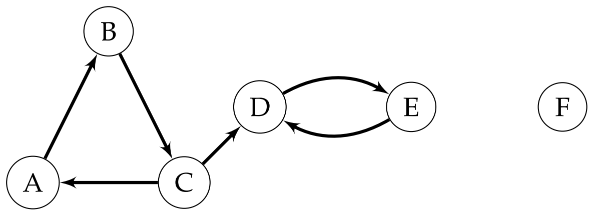


Figure 2.4: An example of a directed multigraph that contains multiple *SCC's*.

## 2.3 Algorithms

Finding and enumerating all cycles requires adequate algorithms for that purpose. When choosing what algorithms to employ in the implementation of the cycle detection tool, the strengths and weaknesses of a number of candidate algorithms needed to be evaluated. These candidates included Tiernan's [11], Tarjan's [10], Johnson's [2] and Wienblatt's [12] algorithms that could be used to answer the first and second research questions.

### 2.3.1 Tiernan's algorithm

Tiernans's algorithm [11], published in 1970, was intended to find all "Elementary Circuits", or simple cycles as called in this paper, in a directed graph.

Tiernan's algorithm has a time complexity of $O(V(C)^V)$ and a space bound of $V + E$, where V is the number of vertices, E is the number of edges and C is a constant according to Mateti and Deo [9]. The algorithm uses three arrays: first, a one-dimensional array $P$ (1 x C, where C is the number of cycles found) that is filled with all *paths* that are found to be cycles; second, a two-dimensional array $H$ where *blocked* vertices are stored when finding cycles; third, a two-dimensional array $G$ that represents the graph, where each row is a vertex and each column in that row represents an edge to a neighbouring vertex. These vertices are unavailable to use during the process of finding cycles because those vertices have already been explored. The pseudo-code implementation of Tiernan's algorithm can be found in Algorithm 1. Tiernan's algorithm is initialized by choosing a start-vertex that is subsequently *indexed* as 1 and inserted into stack representing the current path $P$ (line 1 until 5 in Algorithm 1). The algorithm proceeds by extending $P$ using a depth first search down the vertices that are reachable with a outgoing edge from the start-vertex. This step is what Tiernan called a *path extension* [11], and can be seen at line 7 to 14 in Algorithm 1. An extension of $P$ can only be performed if the candidate vertex for extension meets these conditions:

1. It is not currently included in $P$.
2. The vertex's *index* must be higher than the starting-vertex's in $P$.
3. It can not be *blocked*, i.e. included in the array H.

The algorithm will extend outward from the start-vertex to its neighbours and then their neighbours and so on as long as the conditions outlined above are met. When no more extensions are possible then a *circuit confirmation* will take place as described by Tiernan [11] and can be seen at line 17 to 22. However, circuit confirmation is called *cycle confirmation* instead.

A cycle confirmation consists of confirming if the start-vertex is the same as the last vertex in $P$, and can be seen at line 17. If the first and last vertex are the same, then a cycle has been found. The algorithm then enters the *vertex closure* step, as seen at line 24. After the initial cycle is found and to minimize unnecessary searching the algorithm tries to find cycles that mostly consist of the same path as the currently found cycle. This is done by backtracking from the last vertex, blocking the vertex in the process and then trying to extend the path via another vertex (line 28 to 30). If no outgoing edge is found then the backtracking continues until one is found or if the algorithm have returned to the start-vertex. The algorithm operates in accordance with these conditions:

1. *Block* the last vertex *(n)* for the next-to-the-last vertex *(n - 1)* in $P$, i.e. add *(n)* in $P$ to the array $H$ for the vertex *(n - 1)* in $P$.
2. Remove the last vertex in $P$ from $H$ as well as any vertices blocked by the last vertex.
3. Backtrack to the next-to-the-last vertex *(n - 1)* by removing the last vertex *(n)* from $P$.

Once the vertex closure step is completed the algorithm moves on to the *initial vertex advancement* step (line 34); it ensures that for each iteration of the vertex closure steps one less vertex is considered. In addition to this the *blocked* array $H$ is completely cleared, this is done in order to achieve a clean slate for the next iteration of the *path extension* step, and the following steps. Disregarding the vertices below the current index in the consecutive iteration is made possible by incrementing the index of the start-vertex by one each iteration (line 37). After many iterations the algorithm arrive at the point where only one vertex remains; at this point the algorithm has served its purpose and subsequently terminates.

**Algorithm 1** Tiernan's algorithm [11]

```
[Initialize]
$P \leftarrow 0$                                   ▷ The array P is emptied.
$H \leftarrow 0$                                   ▷ The array H is emptied.
$k \leftarrow 1$                                   ▷ k is the size of the path stack.
$P[1] \leftarrow 1$                                ▷ The start-vertex is indexed as 1.

[Path Extension]
for each edge E in P[k] do     ▷ P[k] is the top of the path stack, E -> indicates the vertex
that E "leads" to.
    (1) E -> index > P[1]       ▷ following three conditions needs to be true to continue.
    (2) E -> index not in P
    (3) E -> index not in H
    if (1),(2) and (3) are true then
        Push E -> index onto P
        $k \leftarrow k + 1$
        [Go to Path Extension]
    end if
    [Circuit Confirmation]
    if E -> index == P[1] then
        Go to [Vertex Closure]
    else
        Print and/or save P.
    end if
end for
[Vertex Closure]
if size of P == 1 then
    Go to [Advance Initial Vertex]
else
    $H[P[k]] \leftarrow 0$        ▷ Clear the row representing the current vertex in the blocked set.
    $H[P[k-1]] \leftarrow$Pop top of P
    $k \leftarrow k - 1$
    Go to [Path Extension]
end if

[Advance Initial Vertex]
if P[1] == The last vertex then Go to [Terminate]
else
    $P[1] \leftarrow P[1] + 1$    ▷ Place the vertex with index P[1] + 1 on the bottom of the stack,
remove P[1].
    $k \leftarrow 1$
    $H \leftarrow 0$                               ▷ Blocked-list H is cleared.
    Go to [Path Extension]                         ▷ Start again from new start-vertex
end if
[Terminate]
```

A running example of Tiernan's algorithm can be given by studying Figure 2.5 and Algorithm 1. The algorithm will start by taking the vertex with the lowest index $V_1$ and push it on $P$. The algorithm expands and places $V_2$ and then $V_3$ on the stack $P$. When the algorithm tries to expand from $V_3$ and finds out that $V_1$ is already on the stack it will stop expanding and will start with cycle confirmation behaviour. It will then find the cycle $\langle V_1 \rightarrow V_2 \rightarrow V_3 \rangle$. The algorithm will then backtrack to $V_3$ and $V_1$ will be added to the blocked map, there it will not find any other edges to expand to and will therefore backtrack to $V_2$ and $V_3$ will be blocked. From $V_2$ there is a edge that has not been explored, P is therefore extended to include $V_4$. From there the algorithm extends to $V_5$ and then back to $V_2$. Since $V_2$ is not the vertex at the bottom of the path stack $P$, a cycle will not be found. The algorithm backtracks, and eventually end up at the vertex with the lowest index. The lowest index gets incremented and the algorithm starts from the new lowest index vertex.

### 2.3.2 Johnson's algorithm

Johnson's algorithm [2] was published in 1975 with the purpose to enumerate all simple cycles in a directed graph. It has a time complexity of $O(V + E) * C)$ and a space bound of $Vertices + Edges$, according to Mateti and Deo [9]. Johnson's algorithm requires that the graph has been divided into *SCC's*, and the vertices composing the SCC's have been assigned unique indexes. In Johnson's paper [2], Tarjan's algorithm is put forward as a contender for that role. How Tarjan's algorithm creates SCC's is explained in Section 2.3.3.

Johnson's algorithm uses a boolean array called *blocked*, a set called *Blocked set*, and a stack *Path*, as well as a integer called *Start Vertex*. Johnson's algorithm takes a graph divided into SCC's as input. The reason for divisions into SCC's is that cycles cannot exist across the borders of two SCC's. Therefore, searching for such cycles is resources spent in vain.

The algorithm works by deploying a depth first search *DFS* that goes out from the vertex with the lowest index in the current SCC. The index will be saved as the *Start Vertex*. Johnson's algorithm recursively traverses the vertices to find and enumerate all cycles.

Each vertex that the deapth first search traverses will be pushed to the *path* stack, as seen at line 24 in Algorithm 2. Its index will get blocked in the *blocked* array (line 25). blocking vertices is done to avoid vertices appearing multiple times in the *path* stack; since the criteria of a simple cycle is that only the start/end vertex appears more than once.

A cycle is found when the DFS reaches the current start-vertex again (line 27 Algorithm 2). If a cycle is found, then all vertices forming the cycle are copied from the *Path* stack and then saved to a result list. Each result list will contain one cycle each (line 28). If not then the DFS cannot continue, it starts to backtrack. During backtracking it removes vertices from the top of the stack until it reaches a non explored edge. When a vertex V gets popped from the stack, the index of V is pushed to the *Blocked-set* at the index of the vertex currently on top of the stack, as seen at line 37 and 38.

The purpose of *blocked* and *blocked-set* is to hinder the DFS to traverse vertices already evaluated (from the current *start-vertex*).

If the depth first search is concluded, the integer *start vertex* is incremented and the DFS will start anew from the vertex with the index *start vertex*. This is done at the top of each recursive transition (line 5 until 10). The general workflow is as follows:

1. Push the vertex $V$ onto the stack.
2. Add $V$ to the blocked array.
3. For each edge of $V$ check if edge returns to the *start vertex*.
    - If true, then a cycle has been found and all vertices in the stack are added to a result array, and saved together with all other result arrays.
    - If false, expand to the neighbouring vertex, and continue from step 1.

**Algorithm 2** Johnson's algorithm [2]

```
input: A SCC
output: Lists each containing a cycle
[Initialize]

$StartVertex \leftarrow 0$
for each vertex V in SCC do
    $Blocked(V) \leftarrow False$
    Clear BlockedSet(V)
    Cycle(V)                                  ▷ Function Cycle defined below
end for

Function Unblock(U)                           ▷ U is index of vertex to unblock.
$Blocked(U) \leftarrow False$
for W in BlockedSet(U) do
    delete W From BlockedSet
    if blocked(W) then
        Unblock(W)
    end if
end for
Function End

Function Cycle(V)                             ▷ V is an index representing a vertex.
$F \leftarrow False$
Stack.push(V)
$Blocked(V) \leftarrow True$
for E in Graph(V).edges do
    if E Leads to StartVertex then
        [Output cycle composed of stack followed by StartVertex]
        $F \leftarrow True$
    else if Not Blocked(W) then               ▷ W is the index of the vertex that edge E leads to.
        $F \leftarrow Cycle(W)$
    end if
    if F then
        Unblock(v)
    else
        for edges in Graph(V).edges do
            if V is not in BlockedSet(W) then
                BlockedSet(W).push(V)
            end if
            Stack.pop()
            Return F                          ▷ Returns from Cycle() if a cycle has been found or not.
        end for
    end if
end for
Function End
[Terminate]
```

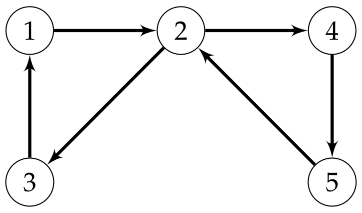


Figure 2.5: An example SCC that contains 2 cycles ($1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$).

4. Depending on if a cycle was found after returning from the recursive transition or not two different results would arise.
    - If a cycle was found, then the vertex $v$ is removed from the *blocked* array. When this happens all vertices that $v$ blocked (as seen in the *blocked-set*) are also removed from the list.
    - If no cycle was found, then the vertex $v$ is added to the blocked list on the place of vertex backtracked to $w$, so that $v$ will be blocked until $w$ gets unblocked.
5. The last step is to pop the current vertex index from the stack and return to step 1. If no more vertices are left in the *path stack*, then the algorithm advances the start vertex and starts anew from step 1.

An example of how Johnson's algorithm works can be given by studying Figure 2.5. Starting out the algorithm will start from $V_1$. $V_1$ is put on the *path* stack, and in the *blocked* array. The algorithm will continue passing by $V_2$, blocking it and putting it on the path stack. The algorithm could at this point expand to either $V_3$ or $V_4$. In this example $V_4$ is randomly chosen to be expanded to. $V_4$ and $V_5$ both gets put on the path stack and blocked in the blocked array. When the algorithm tries out to expand from $V_5$ to $V_2$ it will realise that $V_2$ is blocked. Since there are no way to further expand the DFS the algorithm backtracks. $V_5$ is added to the *blocked set* on the same index as $V_2$, so when $V_2$ gets unblocked, $V_5$ unblocks as well. The algorithm continues to backtrack, $V_4$ is added to *blocked set* at the index corresponding to $V_5$. The algorithm now finds a unexplored edge leading to $V_3$. $V_3$ gets put on the stack and is also blocked. From $V_3$ a edge is found that leads back to the *Start Vertex*, the vertex at the bottom of the stack i.e $V_1$. The vertices in the found cycle $\langle V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_1 \rangle$ are stored in a result array, and the algorithm backtracks. Since a cycle was found, it unblocks V(3) and v(2). When V(2) gets unblocked, V(4) and v(5) unblock as well. The algorithm then reaches V(1) again, and since no more edges can be explored, it starts over with a new *start-vertex*. This time the *start-vertex* is V(2). The algorithm continues with the above described behaviour until all vertices have been considered as *start-vertex*. It is worth noting that the algorithm does not extend the DFS to a vertex that has an index lower than the index of the start-vertex since all potential cycles trough that vertex has already been evaluated.

### 2.3.3 Tarjan's algorithm

Tarjan's algorithm [10] was developed in 1971 to find SCCs and was created by Robert Tarjan. The algorithm has a time complexity of $O(V + E)$ and a space bound of $V + E$, where V are vertices, E are edges and according to Mateti and Deo [9]. The time complexity is so low since all vertices $V$ are only visited once. Tarjan's algorithm works by using a stack $P$ where the algorithm stores vertices and employs deapth first search to traverse the graph.

The algorithm works by taking a random vertex $V$ in the graph. From $V$ the algorithm starts to do a depth first search saving each vertex passed by on top of the stack. Each vertex passed by also gets assigned a unique incrementing *id* and a lowlink cost, as seen at line 14 to 16 in Algorithm 3. The lowlink is initially set to the same value as the *id*, in such a way

**Algorithm 3** Tarjans's algorithm [10]

```
 1: input: graph G = (V,E)
 2: output: set of SCC
 3: [Initialize]
 4:
 5: $index$ ←0
 6: $P$ ←Empty Stack
 7: for V in G do                                              ▷ V = vertex, G = Graph.
 8:     if V.index is undefined then
 9:         StronglyConnect(V)
10:     end if
11: end for
12:
13: Function StronglyConnect(V)
14: $V.index, V.lowlink$ ←index
15: $index$ ←index + 1
16: P.push(V)
17: $V.onstack$ ←True
18: for each edge E in V do                                    ▷ E = edges.
19:     if if E leads to vertex(W) with unassigned index then
20:         StronglyConnect(W)
21:         $V.lowlink$ ←min(V.lowlink, W.lowlink)
22:     else if W.onStack then
23:         $V.lowlink$ ←min(V.lowlink, W.index)
24:     end if
25: end for
26: if V.lowlink == V.index then
27:     [Start new strongly connected component]
28:     do
29:         $W$ ←P.pop()
30:         $W.onStack$ ←false
31:         [add W to strongly connected component]
32:     while $W \neq$V
33:     [add V to strongly connected component]
34:     [Output the current strongly connected component]
35: end if
36: End function
37: [Terminate]
```

that the start vertex $V$ gets *id* and lowlink value of 0. The next vertex explored gets *id* 1 and lowlink value of 1, and vertex $V_n$ gets *id* and lowlink equal to $n$. The algorithm continues to traverse the graph and put vertices on the stack until one of two cases occurs.

1. A vertex already on the stack is encountered.
2. There are no vertices that the DFS can expand to.

If a vertex already on the stack is encountered, the algorithm compares the lowlink value of the vertex at the top of the stack to the lowlink value of the vertex that the algorithm tried to expand to (line 22 and 23 in Algorithm 3). the vertex on the top of the stacks lowlink value will then be assigned the lowest of the compared lowlink values. After which the algorithm continues to traverse the graph until case there are no edges that the DFS can expand through.

If no vertices can be expanded to, the algorithm backtracks from the last vertex $Vn$ and pops it from the stack. The next to last vertex, $V_{n-1}$, becomes the new last vertex on the top of the stack (line 21). $V_{n-1}$ will also change its lowlink value to be the lowest lowlink value between $V_{n-1}$ and $V_n$. The backtracking will continue until the algorithm either:

1. Encounters a non-explored edge as indicated by the neighbouring vertex not having assigned index and lowlink values (line 19 is true).
2. Finds a vertex that has the same lowlink and index value (line 26 is true).

If the algorithm finds an unexplored edge, it continues to explore down that path, and eventually backtrack again. During backtracking, if the algorithm finds a vertex that has the same index and lowlink value, it is said to be the start vertex of the current SCC. If the start vertex is found, the algorithm pops vertices from the stack $P$ until it has removed the start vertex of the current SCC. The algorithm stores them together in an output structure, such as an array or list, one for each SCC. Tarjan's algorithm continues to do this until the start vertex is stored in the output list.

the algorithm continues the above described behaviour until there are no vertices left on the stack. Tarjans algorithm will then take a new unvisited node in the graph and start anew from there. If there are no unvisited nodes left, the algorithm stops since all vertices have been visited and stored in SCCs.

An example of how the algorithm would operate can be seen in Figure 2.4. Tarjan's algorithm would start with an arbitrarily chosen vertex, for example choosing $A$. $A$ will get assigned a $lowlink$ and $index \rightarrow 0$, the algorithm would then expand to $B$. $B$ will get $lowlink$ and $index \rightarrow 1$. Expand again and give $C$ $lowlink$ and $index \rightarrow 2$. The algorithm could go two ways here, in this example the algorithm continues to $D$ that gets assigned 3. The algorithm continues down E that gets assigned 4. When the algorithm encounters D again it will compare the lowlink of E with the index of D. Since *Index(D) < lowlink(E)*, $lowlink(D) \rightarrow index(E)$. No more edges can be explored so the algorithm starts backtracking, when Tarjan's algorithm reaches D it realises that lowlink(D) equals index(D), and therefore, D starts a new SCC, the SCC will be saved and the algorithm continues. The edge from C to A rediscovers A and compares *lowlink(C) > Index(A)* therefore, $lowlink(C) \rightarrow Index(A)$. Backtracking from C to B will do the comparison *lowlink(C) < lowlink(B)* therefore, $lowlink(B) \rightarrow lowlink(C)$, and then backtrack again. Since lowlink(A) equals index(A) A starts a new SCC and is saved together with all vertices with the same lowlink value. Since there is no edges left to explore the algorithm takes a new vertex to start from $F$. Since $F$ cannot expand anywhere it is considered to be a SCC of its own, since it can always reach itself.

### 2.3.4 Weinblatt's algorithm

The algorithm proposed by Weinblatt [12] in 1972 has a time complexity of O($V(C)^V$) and a space bound of $V * C$, where V is the number of vertices and C is a constant, according to Mateti and Deo [9]. Weinblatt had a different initial approach to the other algorithms discussed in this chapter. This difference primarily consists of removing: first, any vertices on which no edges terminate as well as any edges originating from them; second, any vertices that do not have any edges originating from it as well as all edges to those vertices that terminate on that vertex; i.e, any vertices that cannot be a part of any cycles. The step of removing these vertices is repeated until no vertices or edges remain that cannot be a part of a cycle. The next step is to choose a vertex as a *start-vertex*. The vertex is chosen arbitrarily, i.e., it is not important which vertex is chosen as the start-vertex. Once a start-vertex is chosen, its neighbouring vertices are explored. The process of exploring consists of adding each vertex found to a list structure termed the *TT* (Trial Thread) when encountered while following each edge.

If a vertex is found twice and the vertex is still on the TT, then a cycle has been found. This cycle would consists of the vertices from the last time the vertex was found to the current vertex. However, if a vertex is found twice and the vertex is not on the TT anymore, then

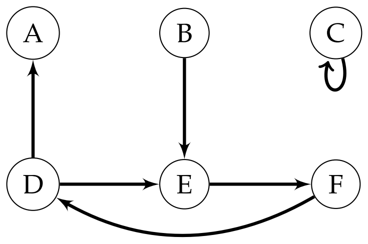


Figure 2.6: An example illustration the operation of the Weinblatt algorithm.

the algorithm employs a recursive procedure with the aim of finding cycles that include the once-again-found vertex and a vertex still on the TT. As it turns out, if this cycle is found, it is always an already found cycle or a "terminal subpath" as described by Weinblatt [12]. A "terminal subpath" is a cycle that is a decomposed other cycle e.g. a cycle could consist of: $A = \langle a \rightarrow b \rightarrow a \rightarrow c \rightarrow d \rightarrow e \rightarrow a \rangle$. In this example the "subpaths" of $A$ would be $\langle a \rightarrow b \rightarrow a \rangle$, and $\langle a \rightarrow c \rightarrow d \rightarrow e \rightarrow a \rangle$.

After a cycle is found, the algorithm advances to the next step, namely tracking the TT back to the vertex called a "branch point" or, a *branching* vertex. A branching vertex is a vertex that includes more than one edge to be explored and could therefore lead to unexplored vertices. When backing up to the branching vertex the algorithm operates by extending from a chosen start vertex and backtracking when a dead-end is found. Once every vertex that can be extended to from the start vertex has been explored, then a new start vertex, if possible, is chosen and the process is started anew. The algorithm terminates when no extensions are possible, and no vertices are available to act as a start vertex. The pseudo-code of this algorithm can be found in Algorithm 4.

An example of how Weinblatt's algorithm works in practise can be seen in Figure 2.6. The algorithm starts off with removing any vertices on which no edges terminates, in this case B. Then also removing the edges originating from that vertex, and in our example that would be the edge from B to E. In the next step the algorithm removes any vertices from which no edges originates, in this case A, and then the edge from A to D. Then after the [Initialize] step the starting vertex is chosen (via the [Select] step). In this example C is choosen. C is added to the TT and the state of C is set to 1. The only edge from C is subsequently followed back to C ([Extend] step). The edge has not previously been explored and is therefore 0. That edge is now set to 2 to mark that it has been explored. Then the algorithm continues to the [Examine] step. In this step it is concluded that C is already found and therefore the TT is saved to the list of cyclic paths. Then the algorithm jumps to [Backup] step where the last edge is removed from the TT (set the edge to 2). The next step again involves the [Extend] step, where the algorithm conclude that there are no more edges to follow and therefore go back to [Backup] after removing C from the TT (set C to 2) and then conclude that the TT is empty. This will cause us to go to the [Select] step again where the whole process starts anew. In this fashion the algorithm will find, extend and then examine the TT to find the cycle $\langle D \rightarrow E \rightarrow F \rangle$ as well as the already found loop in C.

**Algorithm 4** Weinblatt's algorithm [12]

```
[Eliminate]
(1) If there are any vertices on which no edges terminate, eliminate all such vertices as well as any edges originating on them, and repeat Step (1); if none, then continue.
(2) If there are any vertices on which no edges originate, eliminate all such vertices as well as any edges terminating on the, and repeat Step (2); if none, then continue to [Initialize]
[Initialize]
$S(v) \leftarrow 0$    ▷ v = 1, 2, ..., V, where V is the last vertex in set of states S.
$S(e) \leftarrow 0$    ▷ e = 1, 2, ..., E, where E is the last edge in set of states S.
$TT \leftarrow void$
[Select] (a starting vertex)
Find a vertex v such that S(v) = 0; if none then stop.
$TT \leftarrow v$
$S(v) \leftarrow 1$
[Extend] (the TT)
Find an edge, e, which originates on End(TT) and such that S(e) = 0; if none then set $S(End(TT)) \leftarrow 2$, remove End(TT) from the TT, and go to [Backup].
$TT \leftarrow TT\_e$    ▷ Where TT_e is the TT's path with the edge e appended.
$S(e) \leftarrow 2$
[Examine] (the terminal vertex of e)
$v \leftarrow T(e)$
**if** S(v) = 0 **then**
    $TT \leftarrow TT\_v$    ▷ Where TT_v is the TT's path with the vertex v appended.
    $S(v) \leftarrow 1$
    Go to [Extend]
**else if** S(v) = 1 **then**
    Add v_Tail(TT,v)_v to CP (list of cycles)    ▷ The path from the vertex v appended to the subpath returned by the operation Tail(TT,v), then append v again.
    Go to [Backup]
**else if** S(v) = 2 **then**
    $Recur \leftarrow 0$ and Call subroutine Concat(v) and go to [Backup]
**end if**
[Backup]
Remove the last edge from the TT
**If** TT = void **then** Go to [Select] **else** Go to [Extend].
Subroutine Concat(P)
Establish an empty local list of examined cycle-tails.
$v \leftarrow End(P)$
Go through the steps below to [Continue] for each CP which is on the list of cycles and which contains v.
If Tail(CP,v) = void or in CT then go to [Continue]    ▷ CT is the list of examined cycle-tails
Add the cycle-tail to the list of examined cycle-tails.
If the cycle-tail has any vertices on P, then go to [Continue].
**if** S(End(CP)) = 2 **then**
    $Recur \leftarrow 1$ and call Concat(P_Tail(CP,v)) and then go to [Continue]
**end if**
**if** S(End(CP)) = 1 **then**
    $C \leftarrow End(CP)\_Tail(TT, End(CP))\_P\_Tail(CP, End(P))$    ▷ Where the end-vertex of CP has the TT's path beyond the end of CP as well as the current vertex being extended to, P, and the path beyond the extending vertex P appended to it.
**end if**
If Recur = 0, then add C to the list of examined cyclic paths, and go to [Continue].
If C is not among those cyclic paths which have been added to the list of cyclic paths generated since the last external call to Concat (i.e., the last call from [Examine]), then add C to the list of cyclic paths.
[Continue]    ▷ Return to Concat was called from.
```

# 3 Method

In this chapter the pros and cons of the algorithms introduced in Chapter 2 are discussed, it is also determined which algorithm or algorithms that is going to be implemented. Later in the chapter it is explained what steps that is taken in order to implement and test the cycle detection tool as well the methods used to collect the data discussed in Chapter 4. The approach that is taken in this chapter is outlined in Section 3.1, to answer the research questions and satisfy the main goals of the thesis.

## 3.1 Issues to be addressed

1. Evaluate and implement algorithms that can check for the existence of at least one cycle in a GraphQL schema.
2. Evaluate and implement algorithms that can enumerate all cycles in a GraphQL schema.
3. Create a web application where users can upload their schemas for live evaluation.
4. Write a script to test the implementation, as well as allowing the collection of empirical data from a collection of GraphQL schemas.

Since the thesis does not have a primary focus on web design, the presentation of the website will not be a primary focus, but it must be functional to use and to serve as a practical implementation of the thesis.

## 3.2 Comparing the algorithms

When choosing the most fitting algorithm to perform the tasks that the thesis requires, we had two criteria for choosing an algorithm over another. First, the time-, and to some extent the space-complexity of the algorithms are of interest. Second, the ease of implementation is important, i.e. how much documentation and how well the authors of the different algorithms had explained the algorithms in their respective papers.

The major differences between the algorithms are between those that find SCCs, and those that find all simple cycles. Among the algorithms presented in the theory chapter: Tiernan's, Johnson's and Weinblatt's algorithms fall into the category of algorithms that enumerate all simple cycles in directed graphs. Tarjan's algorithm on the other hand instead finds all SCCs.

Tiernan's algorithm [11] was the first developed among the studied algorithms. However, documentation from previous implementations of this algorithm was not easily found.

Weinblatt's algorithm [12] has a different approach than the others in that it starts off by removing any edges that cannot be a part of any cycles. By removing these edges Weinblatt claims that the processor- and time-resources used is lower when compared to using Tiernan's algorithm. The basis for this claim lies in that Tiernan's algorithm only considers each vertex once. By removing any vertices that cannot be a part of a cycle there are less vertices to consider. There could therefore exist cases where Weinblatt's algorithm has a shorter runtime than Tiernan's. However, this difference is only noticeable if there is an extreme number of vertices to consider, and since the expected maximum number of vertices that our implementation will consider lies approximately around 1000 to 10,000 vertices, this approach will most likely not give any noticeable improvements. Nonetheless, a benefit of removing any vertices that cannot be a part of a cycle is that it ensures that the TT can never be filled with such vertices.

Tarjan's algorithm [10] is an algorithm that only finds SCCs, and since SCCs consists of at least one cycle it could be used to find a single cycle quickly without having to run another algorithm as well. The reason for studying Tarjan's algorithm was that the algorithm could also potentially speed up the execution time for the accompanying algorithm by dividing the graph into SCCs. This is the case since cycles, by definition, cannot exist between two SCCs and therefore the number of paths to consider are considerably fewer. In addition implementing Tarjan's algorithm allows us to study the SCCs found.

Johnson's algorithm [2] had the purpose of enumerating all simple cycles in SCCs. This meant that it had to be accompanied with Tarjan's algorithm, or any other SCC creating algorithm as discussed in Johnson's paper [2]. The main benefit with using two algorithms is that they can focus on different aspects of the detection of cycles. Johnson's algorithm does not need to take into account if a vertex is not part of a SCC since that is already taken care of by Tarjan's algorithm. Another beneficial aspect of the algorithm is the use of a set of blocked vertices which ensures that each vertex is not evaluated more than necessary.

Summarised, the candidates are, in order of time complexity from best to worst: Tarjan's and Johnson's, Weinblatt's and Tiernan's. We therefore, after taking the aspects outlined in the introductory part of this section into account, came to the conclusion that Johnson's and Tarjan's algorithm taken together satisfied the requirements of this thesis to the largest extent.

## 3.3 Creating the cycle detecting tool

Once the algorithms were chosen, the next step was to implement the main part of the tool. The tool was written in JavaScript, or JS, since a requirement of the thesis was to implement the tool in a web application. This resulted in that the a viable language to implement the algorithms in, that had to be able to run the code on a website, was JS. The tool's work flow can be summarised into four points:

1. Converting the given GraphQL schema to a JavaScript object.
2. Converting the object to a graph that the algorithms can operate on.
3. Running Tarjan's algorithm to create the SCCs.
4. Running Johnson's algorithm to enumerate all simple cycles.

### 3.3.1 GraphQL schema to JavaScript Object

While searching for a solution to the first point of the tool's work flow, a tool was found that converts GraphQL schemas to JSON [7]. To be able to use this tool and make the appropriate

```
type A {                    type D {
  id : ID!                    id : ID!
  refB : B                    refE : E
}                                  }

type B {                    type E {
  id : ID!                    id : ID!
  refC : C                    refD : D
}                                  }

type C {                    type F {
  id : ID!                    id : ID!
  refA : A                  }
  refD : D
}
```

Figure 3.1: An example of a GraphQL schema. This schema is the schema for the graph shown in Figure 2.4

changes to it we had to fork the github repository. The necessary changes such as adding support for using interfaces and unions as well as multiple line comments was added. Two branches were made: "master", for the locally run version; "web", for the version that was implemented to be used in the web application as illustrated in Figure A.1. This tool reads the schema from a file and uses a lexer to collect all characters, or tokens, it needs to then be able to parse the tokens and create JS objects. After these steps are completed, it calls the stringify() function native to JS with the object as a parameter to convert it to JSON. This last step however was not needed for our purposes since we needed the JS object to create the graph and was therefore discarded in the implementation. To get a better understanding of what we want to accomplish with this step of the conversion from schema to finally enumerate the simple cycles, see the example schema in Figure 3.1.

### 3.3.2 JavaScript Object to Graph

The conversion starts with the process of adding all the vertices that represent the interfaces, unions and object types in the graph. The interfaces are added to the graph first, followed by unions and types, the function is called once for each of the three types. This can be seen at line 15 in Algorithm 5. The reason for this order is because interfaces need to exist in the graph before the object types or unions are inserted. This is necessary since when adding the object types that implement an interface, the interface should be assigned a reference to the object type that implements said interface. The process of creating references from a given interface to a object type can be seen at line 20 to 25 in the same Algorithm.

The next step involves connecting all the vertices together based on the references that exist between them. This is done by adding references to the vertices to the *reference List*, in each vertex. The reference list consists of pairs of values, where the first value is the name of the field. The second value is a reference to the vertex the field references, as can be seen in Figure 3.2. These second value were, to begin with only names of the vertices that they were a reference to, saved as strings, as seen at line 23, 31 and 40 in Algorithm 5. After adding the corresponding vertices to the reference list, it will be a direct reference to a vertex, which is done at line 52.

The reference list can now be said to be the list of that vertex's edges saved with a label and the corresponding object. Figure 3.2 is a representation of how each vertex is stored in the graph as well as what the vertices contain such as the labels and references. This is not a complete list of the information stored in the vertices. In the same figure the "label" is the name of the field and the "reference" is the actual type, interface or union it is a reference to. "ref" is the index of each entry in the reference list.

```
[graph]
  [vertex]
    [vertexID]
    . . .
    [referenceList]
      [ref]
        [label]
        [reference]
```

Figure 3.2: An illustration of how the data is structured when converted to a graph.

### 3.3.3 Detection and Enumeration of Cycles

To be able to answer the research questions one and two it was decided to divide the process into two parts. The first part consisted of the detection of a single cycle, and the second consisted of the enumeration of all cycles, as described in Section 3.1, points 1 and 2. When trying to find only a single cycle, there are two occasions when it is possible to detect if one exists in the graph. Both occasions arise in the first algorithm, namely Tarjan's Algorithm. The first occurs when a SCC is found since a SCC contains at least one cycle by definition. The other occurs when producing the SCCs a vertex previously encountered is found twice, i.e. when a vertex is found that already exists on the stack. If either one of these cases is found to be true, then one can determine that at least one cycle exists in the graph, discussed more in Section 2.3.3. If the user's aim is to find all cycles and loops in their schema, then Johnson's algorithm takes the produced SCCs and enumerates all cycles and loops, as described in Section 2.3.2.

Between the step of running Tarjan's and Johnson's algorithms on the graph there was a need to prune some of the edges from each vertex. This process consisted of removing the references to the JavaScript objects, i.e. the edges of those vertices that are not a part of a SCC that is larger than one vertex. This was done to minimise the unnecessary exploration those edges would cause when running Johnson's algorithm.

## 3.4 Web application

The JavaScript, or JS, portion of the implementation allowing for the web application's functionality was developed in parallel with the main implementation. This was done because the main implementation and the GraphQL parsing tool discussed in section 3.3.1 was both written in Node.js and was intended to be run directly on the user's machine. The problem is caused by the way inclusion of external files are handled in Node.js versus the way it is most commonly done in web applications. Therefore, some minor changes had to be made in order to implement the JS solution into the web application.

A benefit of moving the tool online was that it would have access to libraries such as Cytoscape.js, allowing for rendering of a visual representation of the graph for the user in real-time. However, the main goal of the web application was that it should be simple to use as well as not requiring any plug-ins to be downloaded to run. The script would be run client-side, drastically easing the load on the server hosting the web application.

**Algorithm 5** Graph conversion module

```
$Graph \leftarrow EmptylistofVertices$                                    ▷ Graph starts empty.
$ExcludeList \leftarrow "Subscrption", "Query", "Mutation"$
for scalar S and enum E in JSON-Object do
    [Add S and E to exclude list]  ▷ Scalar values cannot form cycles and will as such not
be considered
end for

addToGraph(interfaces)
addToGraph(unions)
addToGraph(types)
connectGraph()
[Return Graph]

Function addToGraph(target)
for target T in JSON-Object do
    [Create vertex]
    $Vertex.id \leftarrow T$       ▷ Vertex gets assigned its id from the type name in the schema.
    $Vertex.referenceList \leftarrow Emptylist$              ▷ edges leading out from Vertex
    for  each Interface I that T implements  do
        [Create ref]
        $tmpRef.label \leftarrow "\#implements"$
        $tmpRef.reference \leftarrow T$                                ▷ name to current vertex
        Graph[I].referenceList.push(tmpRef)
    end for
    if target == Union then
        for Union-member U in T do
            if U.type is not in Exclude-List then              ▷ if field is a reference to type
                [Create ref]
                $ref.label \leftarrow "\#unionRef"$
                $ref.reference \leftarrow U.type$
                referenceList.push(ref)
            end if
        end for
    else
        for field F in T do
            if F.type is not in Exclude-List then              ▷ if field is a reference to type
                [Create ref]
                $ref.label \leftarrow F.name$
                $ref.reference \leftarrow F.type$
                referenceList.push(ref)
            end if
        end for
    end if
    $Graph[T] \leftarrow Vertex$
end for
End Function addToGraph

Function ConnectVertices()
for  Vertex V in Graph  do
    for  Ref R in V.ReferenceList  do
        $R.reference \leftarrow [Reference\ to\ vertex\ in\ graph\ indexed\ at\ Graph[R.reference]]$
    end for
end for
End Function ConnectVertices
```

## 3.5 Testing the implementation

Testing of the implementation was done in two parts: a part for parsing schemas and another part for enumerating cycles in a given directed multigraph. During the development of the tool, example schemas were created that, when converted into a graph, would contain all permutations of edge cases that could cause trouble for the implementation. The test cases mainly focus on SCCs that include *bottlenecks* (one or more vertices connecting two otherwise separated SCCs) to test if Johnson's algorithm would correctly block and unblock each vertex. Another aspect tested was making sure that Tarjan's algorithm would find and save all SCCs to allow for Johnson's algorithm to be able to enumerate all cycles in said SCCs.

The tests were created in such a way that each vertex would have multiple edges leading to a single vertex, to make sure that searching through directed multigraphs would not be a problem. The graphs were created with a high degree of interconnectivity in order to further increase the likelihood of finding abnormal behaviour. Loops were also tested. The schemas would then be scrambled to ensure that the order of the types would not dictate if it was possible for the graph to be created and searched through correctly. The cycles the implementation found were then compared with the correct result to ensure the implementation could handle all cases.

The collection of real world GraphQL schemas provided served as the main test for the application, especially for the parsing and graph creation part of the implementation. If a graph was parsed incorrectly, there would be problems with the creation of the graph causing the cycle detection step to not be able to operate correctly. Where this happened, it was made clear what needed to be added or modified in the parser or lexer.

## 3.6 Empirical evaluation

The empirical evaluation was performed on a collection of schemas provided by Yun Wan Kim in his master thesis project [8]. The collection of schemas amounted to 2094 schemas. The data most of interest aside from enumeration of simple cycles, were the execution time for parsing, graph creating and the time to enumerate all simple cycles. Data was also collected regarding SCCs, vertices and edges as well as the number of interfaces, unions and types. With this data it was possible to record the largest SCCs, as well as the median and average of the cycle length but also the distribution of cycles versus the edge-vertex ratio.

The data was collected when executing the tool and appended during runtime in a *CSV* (comma separated values) file. The data was saved with a comma separating each value and a newline separating the schemas. The impact of saving the mentioned data during runtime was thought to be insignificant since the writing to file would be done after a schema had been examined. However, space bound was expected to increase by a couple of hundred bytes of data holding variables at most. Storing additional data for the purpose of collecting data does not increase the time complexity of the implementation.

# 4 Results

This chapter aims to present the resulting data that was collected from running our cycle detection tool on the collection of GraphQL schemas provided by Kim [8] as well as the runtime of the tool for each step, as mentioned in Section 3.6. Among the 2094 schemas tested, 44 schemas were found to contain several orders of magnitude more simple cycles than the other schemas. These 44 schemas will be referred to as *edge case schemas* and will be presented separately due to the fact that the edge case schemas skew the statistics if they were to be included together. They were identified as any schema where complete enumeration of simple cycles would take so long that it would not be feasible to achieve, due to the schema containing a staggering amount of simple cycles.

## 4.1 Adjustments of the schema converter

Some minor adjustments had to be done to the converting component of the implementation in order to correctly parse the given schemas and to avoid faulty parsing of future schemas. The changes were small syntactical changes mostly in the lexer belonging to the schema parser component. The lexer did not differentiate between "[field!]" and "[field]!" both of which are valid to use. There also existed some problems with correctly lexing lists inside of lists , e.g., [[field]], and therefore lists "[]" as well as "!" was removed, since they are not needed for the rest of the implementation. Lists are not needed because a field referencing another type will reference that type in the same way regardless if it is a list or not, and "!" indicates that the field is mandatory, which is not relevant for the graph representation of the schemas. Another change was the added functionality to parse union types as well as correctly discarding comments that spanned multiple lines.

## 4.2 Runtime

The runtime of the cycle detection tool was measured with respect to the three steps that the implementation needed to enumerate all cycles: parsing the schema, creating the graph, and enumerating all cycles as seen in Figure 4.1. The difference between the sum of the measured time of each step and the total measured time is represented as *overhead*. This overhead includes the time of reading and writing to disk as well as rounding errors between

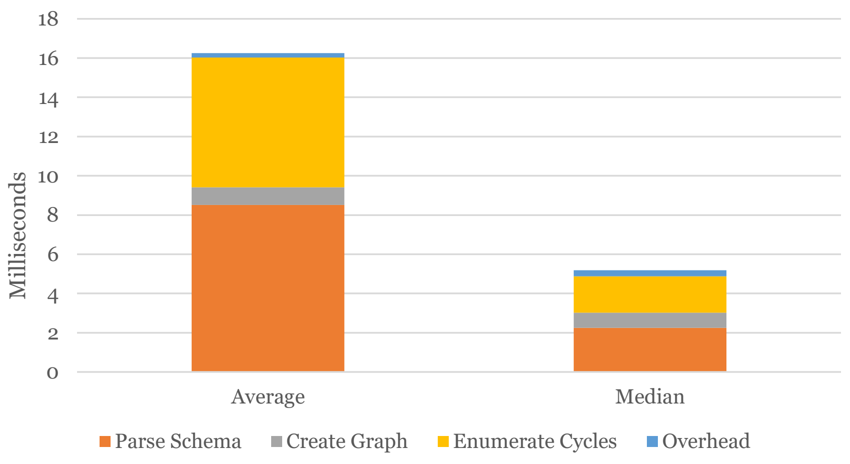


Figure 4.1: Average and median runtime of the cycle detection tool for the collection of schemas. Note that none of the edge case schemas are represented in this graph.

the sum of average/median and the average/median total measured time. The majority of schemas, 1657 schemas of the total 2094, had a runtime below 10 milliseconds, as can be seen in Figure 4.2. These 1657 schemas have an average of 23.72 vertices and 36.87 edges each. The average largest SCC size was measured to be 3.20. The rest of the schemas needed between 11 and 100 milliseconds with an average of 40.65 vertices and 36.87 edges. The schemas with a runtime of above 100 milliseconds have an average of 1238.82 vertices and 1896.29 edges. The schema with the longest runtime was measured to be 10.02 seconds with 162 vertices and 505 edges and 256,348 simple cycles.

## 4.3 Distribution of Types

When gathering the data from the collection of schemas, we were interested in the distribution of types of the vertices as mentioned in Section 3.6. As shown in Figure 4.3, the overall majority of vertices in all schemas are of the object type. For the total of 57,399 vertices they accounted for 54,429, or 94.83 %, of all types. Unions and interfaces accounted for 704 and 2266 or 1.23% and 3.95 %, respectively.

## 4.4 Cycles and SCCs in schemas

Due to the schemas that are edge cases, as mentioned in the introduction of this chapter, we divide this section into two subsections. One section presents the result of the schemas that are regular cases and another for those that are edge cases. Among the 2094 tested schemas, 832, or 39.73%, contained at least one cycle see Figure 4.4. Among these, 44, or 2.10 % of the total number of schemas, are considered edge case schemas.

### 4.4.1 Regular cases

Since there are more schemas without cycles than with, we decided not to include the schemas that did not contain any cycles in the following Figures to increase readability. Another reason was to find the most influential factors causing increased cycle density to occur.

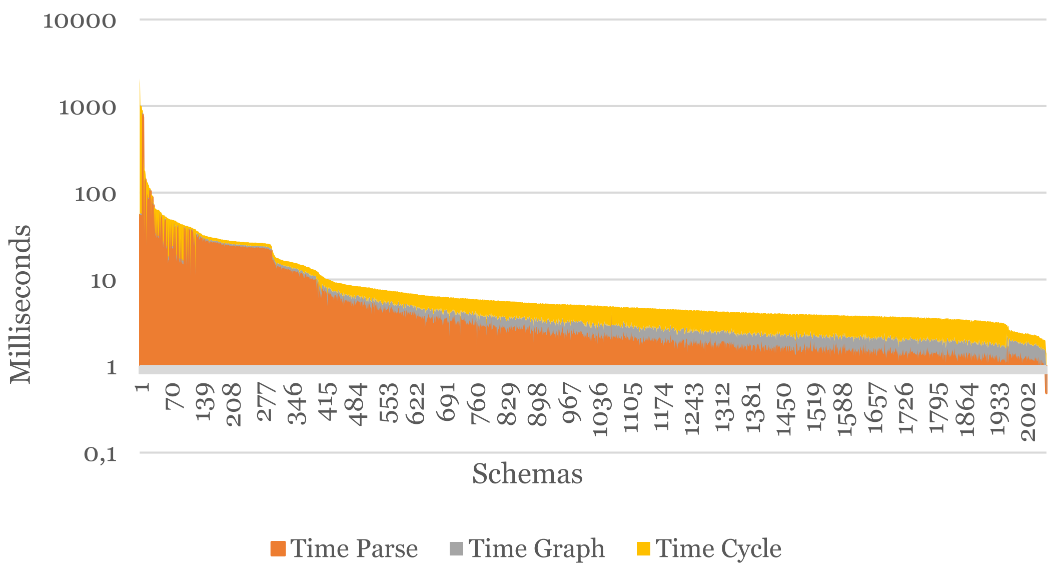


Figure 4.2: Runtime for each schema (without the edge case schemas), where the time is divided into the three phases of the cycle detection tool.

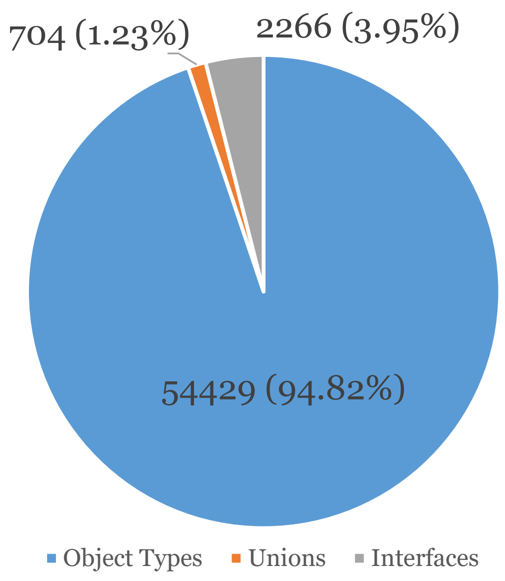


Figure 4.3: The average distribution of vertex types in graphs.

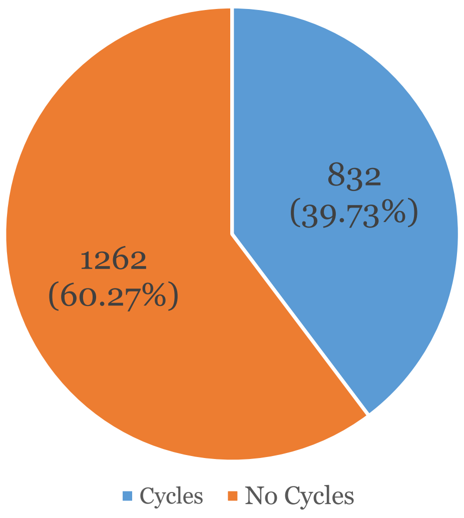


Figure 4.4: Proportion of schemas that contained cycles versus those that did not.

Among the 788 non-edge case schemas, a total of 46567 cycles were found, resulting in an average of 969 in each schema. The median number of cycles in all non edge case schemas containing cycles is four. The huge difference in median and average number of cycles in schemas comes from the fact that most schemas have few cycles, while a small number of schemas have many cycles. One could therefore conclude that most cycles are to be found in only a few schemas as can be seen in Table 4.1 and in Figure 4.5.

The first span of schemas fall into the case where there only exists one cycle in each. This span be seen as clusters on the bottom of the graph in both Figure 4.6a and 4.6b as well as in the ratio between the edges and vertices in Figure 4.7. These clusters lie between 5 and 100 on the x-axis in the first two Figures (edges in the first and vertices in the second) and between 0.5 and 2 on the x-axis (Edge/Vertex ratio) in the last Figure. It is also possible to see these cycles at the far right in Figure 4.5, which illustrates the distribution of cycles over the schemas.

As shown in Table 4.1, a majority of schemas with cycles contain 2 to 10 cycles. These schemas are easily spotted in the figures mentioned above as they appear within the lower bottom quarter of each figure. The following three spans are layered within one quarter, each starting with 11 - 100, followed by 101 - 1000 and so on. It is worth noting that the scales are logarithmic, so the charts' content would be more spread out if they were presented linearly.

We notice that the dot diagrams in Figure 4.6a and 4.6b are very much alike but the dots in Figure 4.6b are more compact, whereas 4.6a seems more spread out and shifted to the right. The shift might indicate that there exists a slightly stronger correlation between a graph's number of edges and a higher density of cycles, compared to a graph's number of vertices. As shown in Figure 4.7, the growth of vertex connectivity could correlate to an exponential growth of cycle density in a given schema.

The result of comparing the SCC size to the amount of cycles in a given schema is presented in Figure 4.8. There seems to exist an exponential connection between the size of the SCCs and the amount of cycles in the studied graph, since if the SCC size grows linearly the amount of cycles grows exponentially. In the tests none of the regular case graphs contained an SCC larger than 100 vertices.

| Cycles | Avg. Vertices | Avg. Edges | Avg. Edge/vert | Largest SCC size | Schemas |
|---|---|---|---|---|---|
| 0 | 20.79 | 23.36 | 0.50 | 1 | 1262 |
| 1 | 11.78 | 13.26 | 0.98 | 2.09 | 245 |
| 2 - 10 | 19.79 | 28.57 | 1.37 | 3.85 | 395 |
| 11 - 100 | 48.76 | 106.44 | 2.20 | 11.45 | 75 |
| 101 - 1000 | 77.22 | 212.13 | 2.82 | 34.44 | 32 |
| +1001 | 134.69 | 408.12 | 3.74 | 33.73 | 41 |
| edge cases | 232 | 764.36 | 4.05 | 162.73 | 44 |

Table 4.1: Average vertices, edges, average connectivity and the average size of the largest SCC in evaluated schemas, clustered by the number of cycles contained in the schemas.

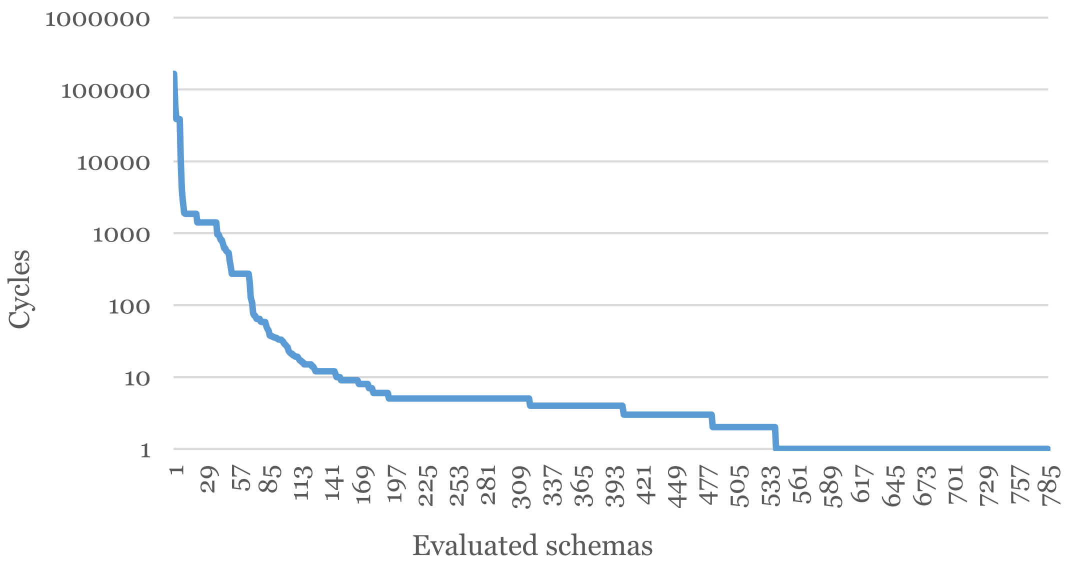


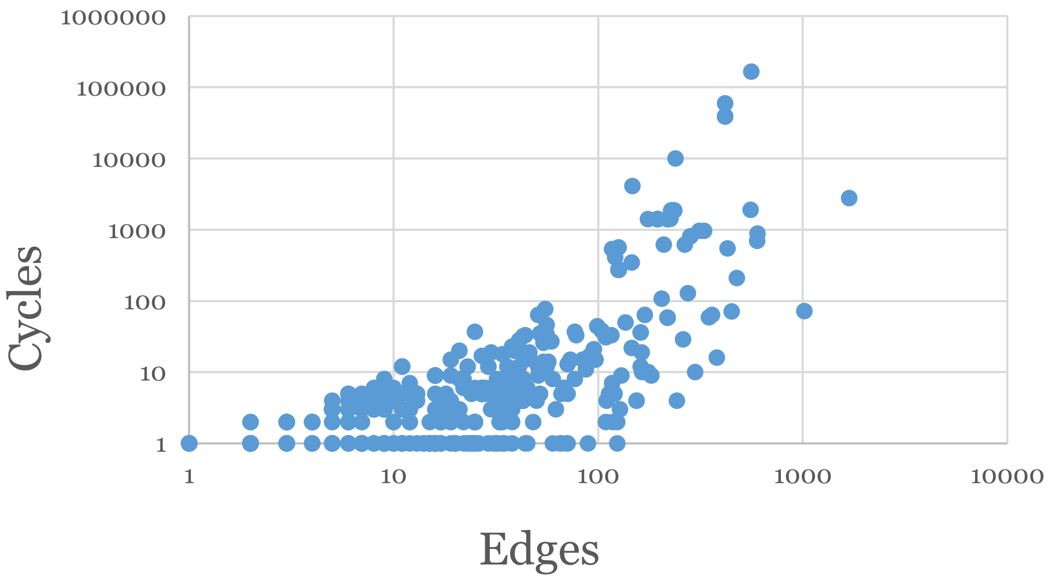
Figure 4.5: The distribution of cycles in non-edge case schemas. Only the schemas that contain cycles are included.

### 4.4.2 Edge cases

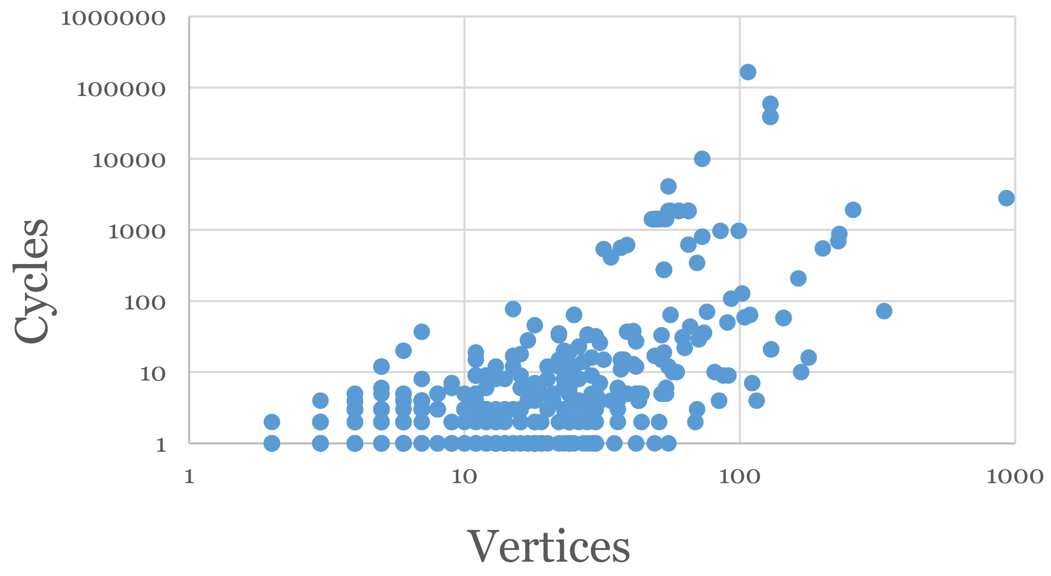
In this subsection the schemas that are considered edge cases are discussed. It is worth noting that the evaluation of these schemas was halted after 40 seconds of execution since the program would be terminated shortly thereafter for using too much RAM. The reason for the high RAM usage is that the program store cycles and metadata about the cycles in RAM during execution.

Even letting the tool use up to 8GB of RAM forces the OS to terminate the application. The reason for storing the cycles during runtime is because we want to be able to present all cycles to the user after the enumeration of all cycles has been completed.

In order to circumvent the problem with memory usage we added functionality to only count and not save the cycles. In addition the ability to run the application and save the cycles to a file, or direct write to the terminal were added as well. However both of the latter methods added some overhead, as writing to disk or to terminal is much slower than saving the data in the main memory.



(a)



(b)

Figure 4.6: a) The distribution of cycles compared to the number of edges in schemas. b) The distribution of cycles compared to the number of vertices in schemas.

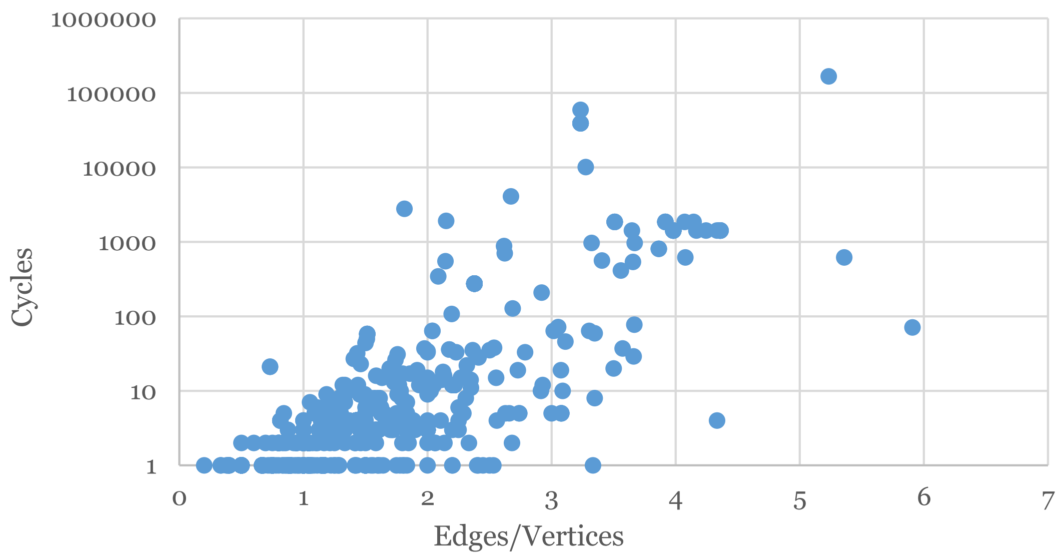


Figure 4.7: The distribution of cycles compared to the ratio between edges and vertices in schemas.

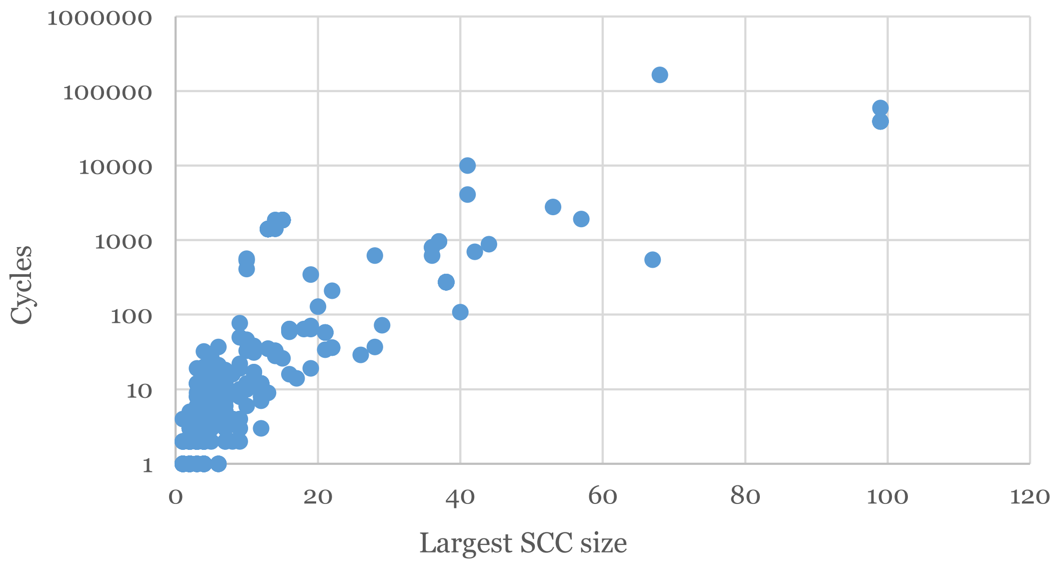
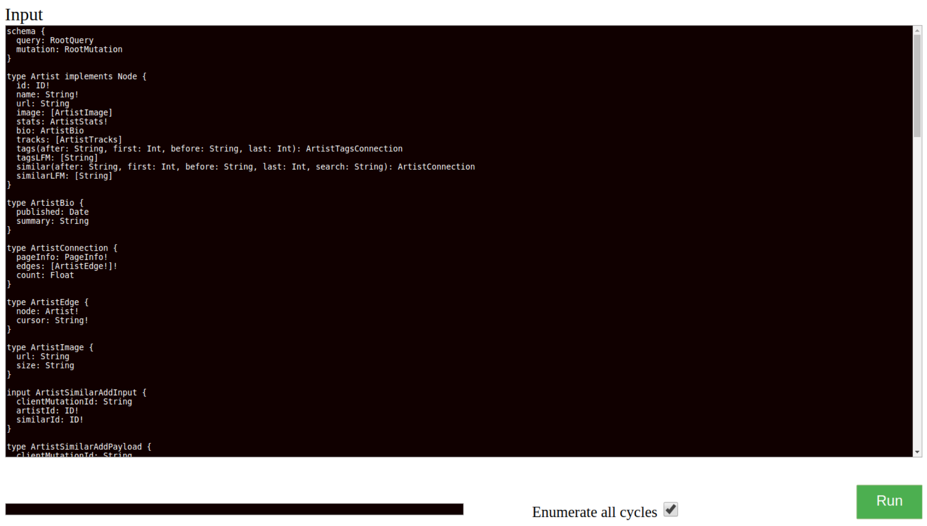


Figure 4.8: The amount of cycles in schemas compared to the size of the largest SCC in each schema.



```
schema {
  query: RootQuery
  mutation: RootMutation
}

type Artist implements Node {
  id: ID!
  name: String!
  url: String
  image: [ArtistImage]
  stats: ArtistStats!
  bio: ArtistBio
  tracks: [ArtistTracks]
  tags(after: String, first: Int, before: String, last: Int): ArtistTagsConnection
  tagsLFM: [String]
  similar(after: String, first: Int, before: String, last: Int, search: String): ArtistConnection
  similarLFM: [String]
}

type ArtistBio {
  published: Date
  summary: String
}

type ArtistConnection {
  pageInfo: PageInfo!
  edges: [ArtistEdge!]!
  count: Float
}

type ArtistEdge {
  node: Artist!
  cursor: String!
}

type ArtistImage {
  url: String
  size: String
}

input ArtistSimilarAddInput {
  clientMutationId: String
  artistId: ID!
  similarId: ID!
}

type ArtistSimilarAddPayload {
  clientMutationId: String
```

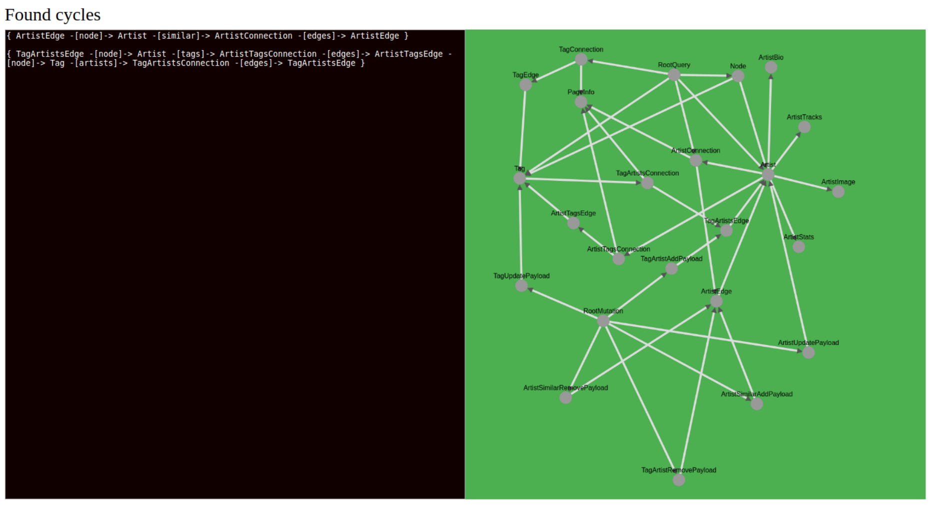
Figure 4.9: Screenshot of the web application before the user have entered a GraphQL schema to be evaluated.

After letting the implementation evaluate an edge case schema for roughly 8 hours, only counting the cycles it would still not have completed the enumeration of all cycles. It did however find close to 200,000,000 simple cycles. In addition, data was recorded in order to decide on the most common factors that would allow a graph to contain such an extreme amount of cycles that would cause problems for the application. The data collected for the edge cases can be seen in Table 4.1.

The data collected seems to indicate that the largest SCC size and the interconnectivity in the graph would be the two factors most responsible for creating the edge case schemas.

### 4.4.3 Web application

Since the implementation of the cycle detection tool was done in JavaScript, conversion from a desktop run implementation to a web application was made easy. However some changes had to be implemented since NodeJS and client side JavaScript applications includes files in different ways. The performance of the web tool was also not as good as the NodeJS implementation since it has to be run trough a web browser, however the web application was able to enumerate all cycles in all non edge case schemas. Therefore it was concluded that the web application was working as intended. The layout of the web application can be seen in Figure 4.9. When a user enters a GraphQL schema in the given input text box and clicks on submit the schema will be evaluated and the page will extend to show a text-box containing all simple cycles detected, as well as a graphic representation of the schema, as can be seen in Figure 4.10


Found cycles

```
{ ArtistEdge -[node]-> Artist -[similar]-> ArtistConnection -[edges]-> ArtistEdge }

{ TagArtistsEdge -[node]-> Artist -[tags]-> ArtistTagsConnection -[edges]-> ArtistTagsEdge -
[node]-> Tag -[artists]-> TagArtistsConnection -[edges]-> TagArtistsEdge }
```

Figure 4.10: Screenshot of the web application after the user entered a schema. The application evaluated the schema and produced a directed graph and all simple cycles are presented to the user in the leftmost text-area.

# 5 Discussion

In this Chapter we discuss our findings as well as offer some criticism of some of the methods used to derive the results.

## 5.1 Implementation

The implementation of the cycle enumerating tool is deemed to be successful, as can be corroborated by the data gathered from the implementation from the previous chapter. The tool managed to parse all 2094 schemas and to enumerate all simple cycles in 2060 of the schemas.

A note of caution is due here since even if all possible schema permutations were covered by our test cases, there might still exist some cases where the implementation could possibly provide a faulty result. Therefore, the results should be evaluated cautiously, and the subject of enumerating cycles should be further studied.

Studying the results, it was found that the average runtime for most schemas was quite low. In the non-edge cases, the runtime was at worst slightly above 10 seconds. However, as we found in the tests, only six of the non-edge case schemas had a runtime exceeding one second, and only two took more than two seconds.

However, since the implementation would not need to be run continuously, we concluded that the runtime was sufficiently low to be able to answer research question 1 ("Can a GraphQL schema efficiently be tested for cyclicity?"). According to our tests it is viable to use Tarjan's algorithm on a graph in order to find a SCC of sizes greater than one, or finding a loop in the graph, and therefore be able to prove the existence of cycles in said graph. Showing the existence of cycles in a schema can be done effectively as well, since the runtime of Tarjan's algorithm is in $O(Vertices + Edges)$ only.

With respect to the second research question ("How can all cycles in a GraphQL schema be identified efficiently?") we found that we can use Tarjan's and Johnson's algorithm to create a tool that can enumerate all simple cycles in a given GraphQL schema. We did not expect that cycle density could be so high as to not be able to enumerate the cycles in the edge case schemas. In practice, there should be a maximum number of cycles enumerated, after which the tool simply states that there are at least as many cycles as the maximum value and then simply quit. However, when not taking those edge cases into consideration, then the tool managed to enumerate all cycles in the collection of schemas.

## 5.2 Cycles in schemas

In order to answer research question 3 ("What are the characteristics of existing GraphQL schemas in terms of cyclicity and cycles?") we need to examine the results to decide the prevalence of cycles in GraphQL schemas as well as the circumstances in graphs that allow high cycle densities to form.

As the study shows, the existence of cycles was found in 39,73 % of the considered schemas. The assumption was made that GraphQL was especially suited for social media applications and would therefore include a lot of "users" that relate to other "users", or equivalent scenarios, allowing each schema to contain a few cycles each, but not much more.

A possible explanation that the number of schemas with cycles was lower than originally assumed might be that efforts are already being taken to avoid cyclic relationships when developing GraphQL schemas. This idea is further supported by the existence of very large schemas (more than 5000 vertices) that do not contain cycles at all.

By studying the data in Table 4.1, and the charts in the results chapter, we get results that seem to indicate that no one factor alone is responsible for creating high cycle densities. However, the results indicate that the edges-to-vertex ratio, as well as the size of the largest SCC, seem to have the strongest correlation with the amount of cycles found (as seen in Figure 4.7 and Figure 4.8). This is further supported by the fact that almost all edge case graphs include at least one SCC containing more than 150 vertices, and the average edge/vertex ratio is four. There are a few schemas that were successfully enumerated that have at least one very large SCC, but none of these SCCs is larger than 100 vertices as can be seen in Figure 4.8. It can be seen in Figure 4.6b that there exist non-edge case schemas, that contain more vertices than the average among the edge cases, and as such the amount of vertices can be assumed not to be directly linked to the amount of cycles in a graph.

The absolute worst-case scenario would be if the vertices and edges form a *clique*. A clique is when every vertex inside a SCC can reach all other vertices directly. Such a high interconnectivity in huge graphs would result in a staggering amount of simple cycles. In such a case it might be impossible to parse trough by the tool no matter how much time is given for the task. Johnson's algorithm has a time complexity of $O(Cycles(Vertices + Edges))$[9]. Therefore, the runtime for Johnson's algorithms would grow really huge when the amount of cycles in the graph grows.

However, this is not only related to Johnson's algorithm. All studied algorithm's runtime would grow huge with the studied edge case graphs. Except for Tarjan's algorithm since it grows linearly independent on the amount of simple cycles existing in the graph. Instead taking examples from the collection of edge case schemas, we could try to estimate the amount of simple cycles in an edge case schema. Since the study shows that the average edges/vertex ratio is slightly above 4 as can be seen in Table 4.1, making the assumption that all vertices have 4 outgoing edges each. Giving the previous assumptions, if one were to study two vertices that connect to each other with four outgoing edges each, there would exist 16 simple cycles in a worst-case scenario. The best case would be if 6 of the edges were loops, whereby it would exist 7 simple cycles. If another vertex were added with 4 outgoing edges, and then repeat the thought experiment there would be a potentially exponential increase in cycles; where the worst-case of cycle existence would be $4^3 = 64$, and the best case would be $4 - 1 * 3 + 1 = 10$. The cycle growth is further increased with a fourth and a fifth vertex, and so on. Repeating this thought experiment 160 times (the average largest SCC length in the edge case schemas), provides a worst-case scenario of $4^{160} = 2.135987e + 96$, and a best case scenario of $(4 - 1) * 160 + 1 = 481$. It can therefore be assumed that the amount of cycles in the edge case graphs can be well over what would be feasible to calculate with our tool, since even if the tool manages to enumerate 1,000,000,000 cycles/second, it would take close to: 2.135987e+87 seconds or 6.7731703e+79 years to completely enumerate the graph given a worst-case scenario.

## 5.3 Evaluation of the Method

The main critique we would direct towards ourselves is in how we tested the implementation. There could have been written more automated tests that could have created a graph and compared outputs in order to ensure that there are no cases where the cycles detection tool would fail. This would maybe have been a project in and of itself since the creation of such test cases would require the cycles in the test graph to already been known. Therefore, we opted for handmade test cases where we tried to fit all permutations that we thought could be problematic in a couple of test schemas.

The cycle detection tool could also have been written in some other language like PHP or even C++ in order to further increase performance. However, implementation was done in JavaScript since the converter used was already implemented in JavaScript, and it would be easy to convert the cycles enumeration tool to be implemented in a web application.

If the time constraints were different and we could have spent more time on the project, it would probably be a good idea to write the implementation in C++ or Java[3].

It could be argued that an algorithm with better space bound than Tarjan's and Johnson's algorithms could have been used since the edge cases would not be able to run to completion since the machine tested on run out of available working memory. But as mentioned in the result chapter, decreased space bound would not really have been of much help since it is mostly the enumeration of cycles that is the memory limiting factor. Just counting the cycles or saving them to disk circumvented this, but would had a negative impact on the run time of the application.

# 6 Conclusion

In conclusion, we found that it was possible to answer all the research questions. In doing so we tried to shed some light on how prevalent cycles are in GraphQL schemas. In this thesis we managed to pinpoint some of the reasons for why cycles appear and describe the characteristics of these schemas that have cycles.

## 6.1 The work in a wider context

The tool and data presented in this thesis will hopefully be of help when further developing the GraphQL API as well as for researchers and developers to contribute to their work with developing and understanding the features and limitations of GraphQL. By helping to make GraphQL more effective to use, the work could also in a small way help with lowering electric costs and by extension ease the pressure somewhat on the environment.

An ethical aspect of the report is that it could by extension help large corporations or other actors store more information about citizens/consumers/users in a more efficient manner, which could already be somewhat done but could become easier more cost effective to do.

## 6.2 Future work

Even after more than 2000 tests against real GraphQL schemas, we cannot fully guarantee that there exist no cases where the tool would produce a faulty result. Additional testing would be required, and that could potentially be done in the future if the tool were to be provided to developers writing their own schemas.

# Bibliography

[1] B.O Turesson A.Astratian A.Björn. *Diskret Mattematik*.

[2] Donald B. Johnson. "Finding All the Elementary Circuits of a Directed Graph". In: 4 (Mar. 1975), pp. 77–84.

[3] G. Miller F. D. Hsu X. Lan and D. Baird, eds. *A Comparative Study of Algorithm for Computing Strongly Connected Components*. Vol. 15. IEEE, 2017.

[4] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. 2000.

[5] Olaf Hartig and Jorge Pérez. "Semantics and Complexity of GraphQL". In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1155–1164. ISBN: 978-1-4503-5639-8. DOI: 10.1145/3178876.3186014. URL: https://doi.org/10.1145/3178876.3186014.

[6] Facebook Inc. *Introduction to GraphQL*. 2018. URL: https://graphql.org/learn/ (visited on 07/23/2018).

[7] Jellesma J. URL: https://github.com/jarnojellesma/graphql-to-json-converter.

[8] Yun Wan Kim. "Exploratory Analysis of GraphQL API Repositories and Schema". MEng Project Report. MA thesis. University of Toronto, 2018.

[9] P. Mateti and N. Deo. "On Algorithms for Enumerating All Circuits of a Graph". In: *SIAM Journal on Computing* 5.1 (1976), pp. 90–99. DOI: 10.1137/0205007. eprint: https://doi.org/10.1137/0205007. URL: https://doi.org/10.1137/0205007.

[10] Robert Tarjan. "Depth first search and linear graph algorithms". In: *SIAM JOURNAL ON COMPUTING* 1.2 (1972).

[11] James C. Tiernan. "An Efficient Search Algorithm to Find the Elementary Circuits of a Graph". In: *Commun. ACM* 13.12 (Dec. 1970), pp. 722–726. ISSN: 0001-0782. DOI: 10.1145/362814.362819. URL: http://doi.acm.org/10.1145/362814.362819.

[12] Herbert Weinblatt. "A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph". In: *J. ACM* 19.1 (Jan. 1972), pp. 43–56. ISSN: 0004-5411. DOI: 10.1145/321679.321684. URL: http://doi.acm.org/10.1145/321679.321684.

# A Appendix

## A.1 Git repositories

To be able to work on the implementation independently, as well as to make it possible for others to access our work, we used Git to create multiple repositories for the different components of the tool. To be able to address the first point of the list outlined above we made use of an external repository. It was therefore necessary to make a *fork* of the external repository. A fork is an extension of a repository. In this extension we can make changes that do not affect the original repository. If the original author approves of the changes made to the code, he or she can merge the changes back into the original repository.

During the development of the application we decided that two versions of the application were appropriate: a version that runs locally, used for collecting data and testing the implementation thoroughly; another version for the web application, to be able to run the code online. Since the two versions used parts of the same code from the repositories, we had to split two of the repositories into two *branches*. A branch is a separate saved version of the repository where any changes made can be discarded or merged back into the master branch. With branches it is possible to work on a project in parallel without fear that a commit might change the original code. Why we had to divide the project in this way is explained further in Section 3.3.1.

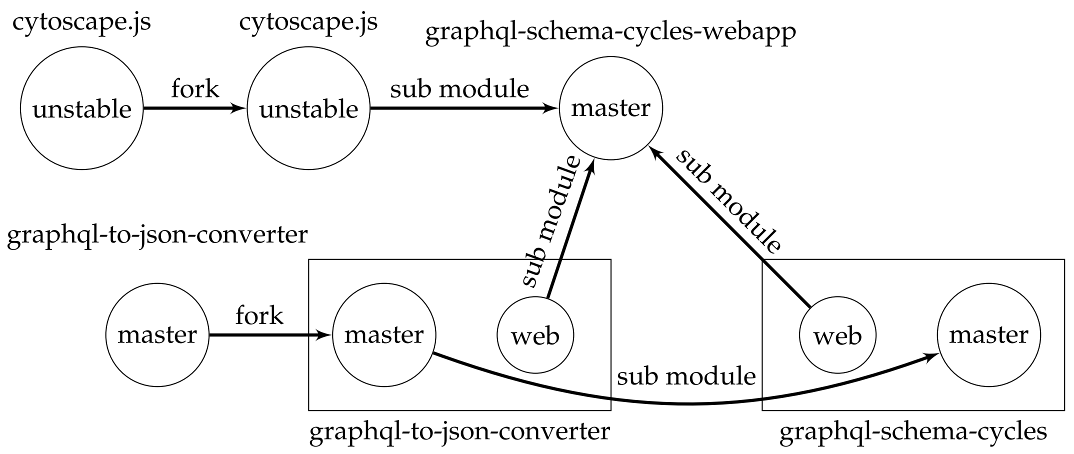


Figure A.1: An illustration of the general layout of the git repositories, branches, forks and how they are connected for the web application as well as for the locally run tool.