

# A Toolset For Visualizing Algorithms On Query Graphs

Master's Thesis

Submitted to the  
Chair of Practical Computer Science III  
Prof. Dr. G. Moerkotte  
University of Mannheim

Submitted by  
Steffen Waldmann  
Matriculation Number 1418219

October 2020



# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Motivation . . . . .	6
1.2	Goal of this Thesis . . . . .	7
<b>2</b>	<b>Basics</b>	<b>8</b>
2.1	Join Ordering . . . . .	8
2.1.1	Relations . . . . .	8
2.1.2	Join . . . . .	8
2.1.3	Selectivity . . . . .	8
2.1.4	Query Graphs . . . . .	9
2.1.5	Operator Trees . . . . .	10
2.1.6	Dynamic Programming . . . . .	11
2.1.7	Bitvector Representation . . . . .	11
2.1.8	Orthogonality . . . . .	11
2.2	Server . . . . .	12
2.2.1	Gin . . . . .	12
2.3	Client . . . . .	12
2.3.1	React.js . . . . .	12
2.3.2	JavaScript XML (JSX) . . . . .	12
2.3.3	Redux . . . . .	13
<b>3</b>	<b>Concepts</b>	<b>14</b>
3.1	Algorithms . . . . .	14
3.1.1	DPccp . . . . .	14
3.2	Server . . . . .	18
3.3	Client . . . . .	18
3.3.1	Query Graphs . . . . .	18
<b>4</b>	<b>Implementation</b>	<b>25</b>
4.1	API Requirements and Design Goals . . . . .	25
4.2	Architecture . . . . .	27
4.3	Choice of Technology . . . . .	29
4.4	Tooling . . . . .	30
4.4.1	Join Problem Generator . . . . .	30
4.5	Data Structures . . . . .	34
4.5.1	Join Ordering . . . . .	34
4.5.2	Visualization . . . . .	35
4.6	Server Implementation . . . . .	38
4.6.1	Utility Functions . . . . .	38

4.6.2	Visualization . . . . .	41
4.6.3	DPccp . . . . .	44
4.6.4	Building . . . . .	50
4.7	JSON Representation . . . . .	50
4.7.1	Join Problem . . . . .	50
4.7.2	Visualization Routines . . . . .	51
4.8	Client Implementation . . . . .	56
4.8.1	Project Structure . . . . .	56
4.8.2	Utility Functions . . . . .	58
4.8.3	Redux Actions . . . . .	59
4.8.4	Building . . . . .	62
4.9	Installation and Development . . . . .	63
4.9.1	Prerequisites . . . . .	63
4.9.2	Dependencies . . . . .	63
4.9.3	Local Development . . . . .	64
4.10	Deployment . . . . .	64
<b>5</b>	<b>Conclusion</b>	<b>66</b>
5.1	Summary . . . . .	66
5.2	Future Work . . . . .	66

## List of Algorithms

1	CreateJoinTree . . . . .	14
2	DPccp . . . . .	16
3	EnumerateCsg . . . . .	17
4	EnumerateCsgRec . . . . .	17
5	EnumerateCmp . . . . .	17
6	Go implementation to calculate symmetrical neighbors . . . . .	33
7	Go implementation of SetMinus . . . . .	39
8	Go implementation of PowerSet . . . . .	39
9	Go implementation of MinUintSlice and MinUintSetBitIndex . . . . .	40
10	Neighbors function for a set of relations . . . . .	40
11	Go implementation of the AddVisualizationStep function to create a new atomic visualization step. . . . .	43
12	Go implementation of EnumerateCsg . . . . .	45
13	Go implementation of EnumerateCsgRec . . . . .	46
14	Go implementation of EnumerateCmp . . . . .	47
15	Go implementation of DPccp . . . . .	49

## List of Figures

1	Query Graphs . . . . .	9
2	Operator trees . . . . .	10
3	Bit vector representation . . . . .	11
4	JSX component sample rendering three paragraphs . . . . .	13
5	Visualizable type conformance requirement . . . . .	26
6	UML sequence diagram of the client-server communication . . . . .	28
7	Sample execution stack for DPccp . . . . .	42
8	Client directory structure . . . . .	57
9	Sample rendered result of DPccp on a “Moerkotte 2018” query graph at step 1 . . . . .	62

## List of Tables

1	Redux Actions . . . . .	60
---	-------------------------	----

# 1 Introduction

## 1.1 Motivation

Computational algorithms introduced in university lectures or scientific papers are not always immediately and intuitively understood by their readers. A significant amount of time has to be allocated to internalizing new concepts from a textual, formal description. While formalization helps with unambiguity it often requires a large degree of domain expertise and can fail to appeal to the readers intuition.

In order to improve accessibility and comprehensibility, researchers and lecturers create visual aids such as sketches, graphs, output tables to keep track of certain variables in the algorithm’s execution, coloring, or informal textual descriptions to make their content more accessible.

However, these visualizations are usually limited by the constraints put on them by the typesetting system they are written in, which produces output to be printed on physical paper: they are static, they cover just one example or a very small and incomplete subset of possible examples, and they are often manually specified instead of generated from an already existing implementation of an algorithm. Moreover, scientific journals even further broaden those constraints by imposing page or word limits on article submissions. While it is possible to create graphical representations of data structures like graphs in a typesetting system such as L<sup>A</sup>T<sub>E</sub>X using libraries like `pgf-tikz`<sup>[1]</sup>, those are often cumbersome and time-consuming to specify. One of the most powerful, but scarcely available tools are algorithm visualizations that are interactive, dynamically created, and can be used on a large subset of all possible inputs.

With today’s consumer electronics, such as laptops or tablet computers, giving the possibility to consume media digitally on flexible graphical user interfaces (GUI) these limitations seem artificial and mostly stem from the “printability” aspect of physical paper. A further explanation for the lack of available visualizations is the fact that researchers and instructors often don’t have the time or resources to produce high-quality, interactive visualizations for their algorithms.

While the 2002 publications from Hundhausen et al. [1] and Naps et al. [2] criticize the efficacy of algorithm visualizations as a pedagogical tool, numerous successful examples have emerged in more recent years. Hohman et al. present and discuss many such examples in their 2020 article “Communicating with Interactive Articles” [3]. The YouTube channel “3blue1brown”<sup>[2]</sup>,

---

<sup>1</sup><https://github.com/pgf-tikz/pgf> (last access: October 27, 2020)

<sup>2</sup><https://www.youtube.com/c/3blue1brown> (last access: October 27, 2020)

who specializes on the visualization of concepts in mathematics and computer science, has gained more than 3 million subscribers and 160 million views in its mere 5 years of existence, and is collaborating with reputable universities such as the Massachusetts Institute of Technology to create supplementary material for their lectures<sup>3</sup>

## 1.2 Goal of this Thesis

This thesis aims to specify and implement a toolset to let us create such an interactive and dynamically created visualization for the algorithm `DPccp` as introduced in [4]. Instead of catering to this specific algorithm we want to keep reusability in mind and try to avoid hardcoded solutions wherever possible.

We also try to minimize the amount of time required to create such a visualization by specifying a declarative toolset that can be used alongside already existing implementations of algorithms—no explicit drawing is needed.

The generic and declarative nature of the toolset makes it not just a useful teaching tool for one particular algorithm, but can even be used among professional computer scientists and software engineers as a quick visual debugging and exploration technique.

---

<sup>3</sup><https://www.patreon.com/posts/mit-lectures-41240316> (last access: October 27, 2020)

## 2 Basics

There are certain terms and concepts the reader should be familiar with in order to understand the aim and purpose of this thesis. We forego a formal mathematical introduction to relational algebra, although a familiarity with its basic concepts is highly recommended. Moreover, we assume that the Landau notation is known. A basic understanding of mathematical set theory, first-order logic, and trigonometry is also strongly recommended. Further, the user should be familiar with fundamentals of distributed applications and their protocols and file formats, such as the Hypertext Transfer Protocol (HTTP), Hypertext Markup language (HTML) and JavaScript Object Notation (JSON). In addition, we assume the reader is proficient in computer programming and scripting, preferably using the Go programming language and JavaScript.

### 2.1 Join Ordering

#### 2.1.1 Relations

Relations are tables in a database. Its attributes are described by a database schema, which specifies the possible attribute types. An entry in the table is described as a *tuple*. Furthermore, we call the number of tuples in a relation the *cardinality* of the relation.

#### 2.1.2 Join

A join links two or more relations by building their cross product and filtering it with the help of a *join predicate*. The result is a new relation which contains all the joined tuples for which the predicate evaluates to true. We denote the join operator using the symbol “ $\bowtie$ ”. As an example, we denote the corresponding join for the two relations  $R_i$  and  $R_j$  and predicate  $p_{i,j}$  with “ $R_i \bowtie_{p_{i,j}} R_j$ ”.

#### 2.1.3 Selectivity

The selectivity of two relations is the ratio between the cardinality of their cross product and the number of elements in the join’s resulting relation. This way we determine the number of entries that is kept after the join.

For two relations  $R_i$  and  $R_j$  regarding a predicate  $p_{i,j}$  the selectivity is thus given by

$$f_{i,j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i \times R_j|} \quad (2.1)$$



**Note 1.** Since we can assume that  $|R_i \bowtie_{p_{i,j}} R_j| \geq 0$ ,  $|R_i \times R_j| > 0$  and  $|R_i \bowtie_{p_{i,j}} R_j| \leq |R_i \times R_j|$  we know that  $f_{i,j} \in [0, 1]$ .

For simplification purposes we only inspect joins with at most one join predicate. Hence, there is just one selectivity between two given relations.

#### 2.1.4 Query Graphs

Relations can be connected in many different ways. These connections can be visualized using a graph. There is a number of different query types, but most importantly we want to have a look at the following six query types as covered in [5], because they represent distinct problem categories in terms of suitability for specific algorithms and general complexity:

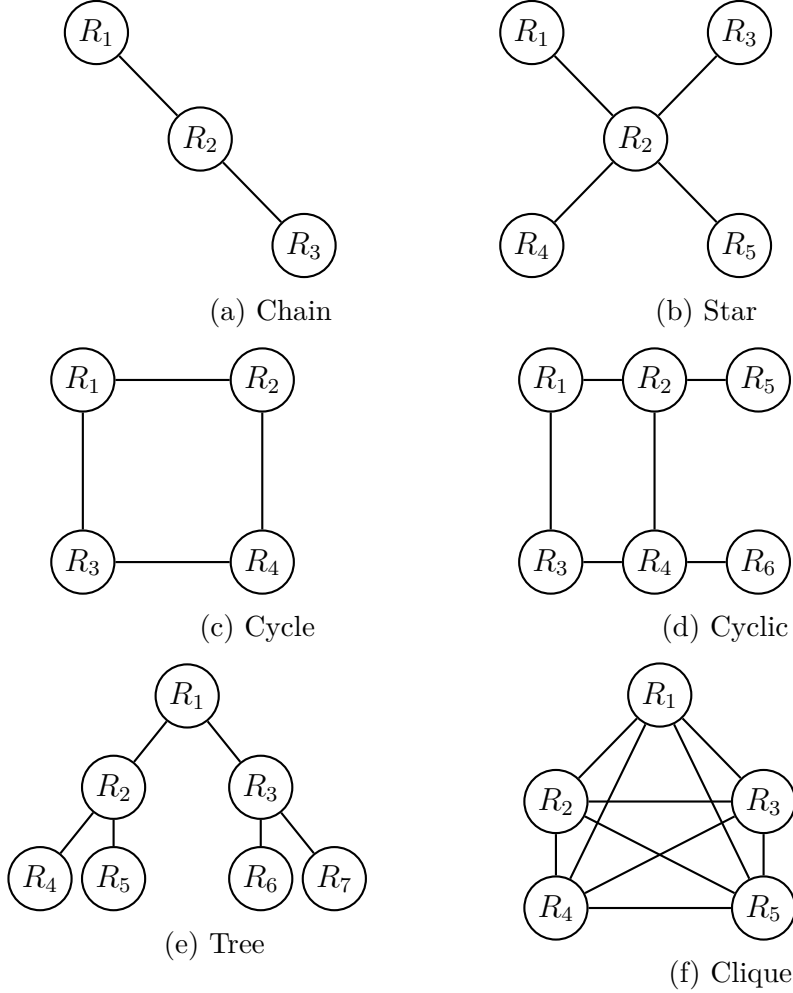


Figure 1: Query Graphs

### 2.1.5 Operator Trees

An operator tree contains operators as inner nodes and relations as leaves. Through this tree the order in which the operations are executed can be graphically visualized. It typically contains any operator of Codd's relational model, however in this thesis we limit those to the only relevant join operator ( $\bowtie$ ), which is derived from a cross product and a subsequent selection. Depending on the structure of the tree it can be classified as either *left-deep*, *right-deep*, *zigzag* or *bushy*.

**Example 1.** The trees for  $(R_1 \bowtie R_2) \bowtie R_3$ ,  $R_1 \bowtie (R_2 \bowtie R_3)$ ,  $(R_1 \bowtie (R_2 \bowtie R_3)) \bowtie R_4$  and  $(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$  can be graphically represented using the following visualizations respectively:

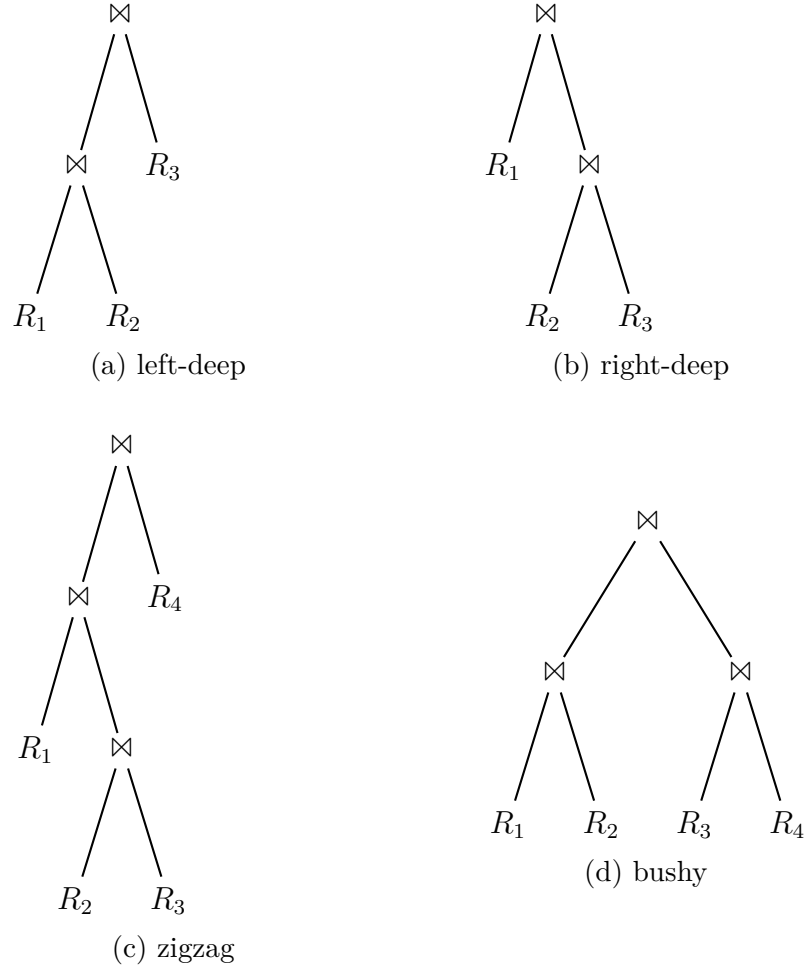


Figure 2: Operator trees

### 2.1.6 Dynamic Programming

Dynamic programming is an optimization technique for dividing complex and computationally expensive problems into sub-problems and reusing their intermediate results in order to prevent redundant calculations.

### 2.1.7 Bitvector Representation

In alignment with [5] we represent sets of relations as bit vectors in the implementation. These are implemented using (unsigned) integer variables in the programming language of choice. When interpreting these integers as base 2, a “1” indicates the presence of a relation  $R_i$ , whereas a “0” is used to indicate absence.

**Example 2.** Hence, the following bit vectors for a 3-bit integer map to the corresponding sets of relations as indicated in figure 3.

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

Figure 3: Bit vector representation

### 2.1.8 Orthogonality

We understand the term *Orthogonality* as defined in *The Art of Unix Programming* by Eric S. Raymond [6]:

**Definition 1.** “Orthogonality is one of the most important properties that can help make even complex designs compact. In a purely orthogonal design, operations do not have side effects; each action (whether it’s an API call, a macro invocation, or a language operation) changes just one thing without affecting others.”

## 2.2 Server

### 2.2.1 Gin

Gin is an open-source HTTP web framework hosted on GitHub<sup>4</sup> and licensed under the MIT license<sup>5</sup>. It provides a router-level middleware that lets us specify API routes for our server-side code.

## 2.3 Client

### 2.3.1 React.js

React.js<sup>6</sup> is a declarative, component-based JavaScript library for building user interfaces. It is an open-source project hosted on GitHub<sup>7</sup>, originally developed and maintained by Facebook. The library was introduced and open-sourced on May 29, 2013<sup>8</sup> and is licensed under the permissive MIT license<sup>9</sup> since September 26, 2017<sup>10</sup>. React components are JavaScript classes extending `React.Component` or implemented as functional components specified using the JSX language (see section 2.3.2). Each component contains two, and only two, instance properties, **props** and **state**, both JavaScript objects. Whereas **props** contains all data passed to the component, **state** manages the component-internal state. Communication takes place in form of a unidirectional, top-down data flow, avoiding two-way data binding. By keeping a library-internal *virtual* Document Object Model (DOM) in-memory, changes in the HTML structure can be efficiently (re-)rendered.

### 2.3.2 JavaScript XML (JSX)

JavaScript XML (JSX) is an XML-style syntax extension to the JavaScript (ECMAScript) programming language. In addition to its HTML-like definition of the website's semantic structure, it allows JavaScript expressions when used in curly braces (`{}`). As such, it cannot be interpreted by web

---

<sup>4</sup><https://github.com/gin-gonic/gin> (last accessed: October 27, 2020)

<sup>5</sup><https://github.com/gin-gonic/gin/blob/master/LICENSE> (last accessed: October 27, 2020)

<sup>6</sup><https://reactjs.org> (last accessed: October 27, 2020)

<sup>7</sup><https://github.com/facebook/react> (last accessed: October 27, 2020)

<sup>8</sup><https://github.com/facebook/react/tags?after=v0.3.3> (last accessed: October 27, 2020)

<sup>9</sup><https://opensource.org/licenses/MIT> (last accessed: October 27, 2020)

<sup>10</sup><https://github.com/facebook/react/commit/b765fb25> (last accessed: October 27, 2020)

browsers, but has to be cross-compiled to valid JavaScript using a transpiler according to its specification<sup>[11]</sup>.

```
class App extends React.Component {
  render() {
    return (
      <div>
        <p>Paragraph 1</p>
        <p>Paragraph {1+1}</p>
        {[3].map(n => <p>Paragraph {n}</p>)}
      </div>
    );
  }
}
```

Figure 4: JSX component sample rendering three paragraphs

### 2.3.3 Redux

Redux<sup>[12]</sup> is an open-source JavaScript library for application state management, commonly used in combination with `React.js`. It is hosted on GitHub<sup>[13]</sup> and licensed under the MIT license<sup>[14]</sup>. The library refers to itself as “A predictable state container for JavaScript apps.”<sup>[15]</sup> It uses elements of functional programming, such as `map`, `reduce` or `filter` to transfer one state to the next. This is not achieved by mutating the current state, but by creating an entirely new state. In order to optimize the expensive copying operations libraries such as Facebook’s `Immutable.JS` <sup>[16]</sup> can be used in combination with `Redux`. All of the application’s state is held in the global `Redux store`, which is changed according to rules defined in *reducers* by invoking defined *actions* holding a payload.

---

<sup>11</sup><https://facebook.github.io/jsx/> (last accessed: October 27, 2020)

<sup>12</sup><https://redux.js.org> (last accessed: October 27, 2020)

<sup>13</sup><https://github.com/reduxjs/redux> (last accessed: October 27, 2020)

<sup>14</sup><https://github.com/reduxjs/redux/blob/master/LICENSE.md> (last accessed: October 27, 2020)

<sup>15</sup>Website title of <https://redux.js.org> (last accessed: October 27, 2020)

<sup>16</sup><https://redux.js.org/recipes/using-immutablejs-with-redux> (last accessed: October 27, 2020)

## 3 Concepts

### 3.1 Algorithms

#### 3.1.1 DPccp

In the following we briefly introduce all algorithms, subroutines and concepts necessary for the later implementation of DPccp. It should be noted that we make the assumption of all nodes in the query graph being indirectly or directly connected to each other. A more extensive introduction can be found in the book “Building Query Compilers” [5] by Prof. Guido Moerkotte, one of the algorithm’s inventors.

**CreateJoinTree** In our implementation of DPccp we’ll make use of the CreateJoinTree algorithm as specified in [5], in order to determine the tree with the minimal cost, depending on an arbitrarily specified cost function.

CreateJoinTree( $T_1, T_2$ )

**Input** : Two (optimal) join trees  $T_1$  and  $T_2$ . For linear trees, we assume that  $T_2$  is a single relation

**Output**: An (optimal) join tree for joining  $T_1$  and  $T_2$ .

BestTree = NULL;

**foreach** implementations impl **do**

**if** !RightDeepOnly **then**

    Tree =  $T_1 \bowtie^{impl} T_2$ ;

**if** BestTree == NULL || cost(BestTree) > cost(Tree) **then**  
      BestTree = Tree;

**end**

**end**

**if** !LeftDeepOnly **then**

    Tree =  $T_2 \bowtie^{impl} T_1$ ;

**if** BestTree == NULL || cost(BestTree) > cost(Tree) **then**  
      BestTree = Tree;

**end**

**end**

**end**

**return** BestTree

**Algorithm 1:** CreateJoinTree

**Csg-cmp-pairs** As DPccp’s name entails, we make use of the notion of *csg-cmp-pairs*. For a set of relation  $S$  we use the term csg-cmp-pair as defined in [4]:

**Definition 2.** Let  $S_1$  and  $S_2$  be subsets of the nodes of a query graph. We say  $(S_1, S_2)$  is a csg-cmp-pair if and only if

1.  $S_1$  induces a connected subgraph of the query graph,
2.  $S_2$  induces a connected subgraph of the query graph,
3.  $S_1$  and  $S_2$  are disjoint, and
4. there exists at least one edge connecting a node in  $S_1$  to a node in  $S_2$ .

**DPccp** The algorithm we focus our visualization efforts on is DPccp, as introduced by Guido Moerkotte and Thomas Neumann in 2006 [4]. This algorithm is a join ordering algorithm that limits the search space of possible joins to a theoretical lower boundary [5]. The pseudo-code is specified in the following algorithm [15].

```

DPccp( $R = \{R_1, \dots, R_n\}$ )
Input : A connected query graph with relations
          $R = \{R_0, \dots, R_{n-1}\}$ 
Output: An optimal bushy join tree

foreach  $R_i \in R$  do
    BestPlan( $\{R_i\}$ ) =  $R_i$ ;
end
foreach csg-cmp-pairs  $(S_1, S_2), S = S_1 \cup S_2$  do
    ++InnerCounter;
    ++OnoLohmanCounter;
     $p_1 = \text{BestPlan}(S_1)$ ;
     $p_2 = \text{BestPlan}(S_2)$ ;
    CurrPlan = CreateJoinTree( $p_1, p_2$ );
    if cost(BestPlan( $S$ )) > cost(CurrPlan) then
        BestPlan( $S$ ) = CurrPlan;
    end
    CurrPlan = CreateJoinTree( $p_2, p_1$ );
    if cost(BestPlan( $S$ )) > cost(CurrPlan) then
        BestPlan( $S$ ) = CurrPlan;
    end
end
CsgCmpPairCounter = 2 * OnoLohmanCounter;
return BestPlan( $\{R_0, \dots, R_{n-1}\}$ )

```

**Algorithm 2:** DPccp

In order to further detail the subroutines, we first define the notation used in the subroutines, which is equivalent to the one specified in [5].

Let  $G = (V, E)$  be an undirected graph. For each of the nodes  $v \in V$  we define the neighborhood  $\mathbb{N}(v)$  of  $v$  as

$$\mathbb{N}(v) := \{v' \mid (v, v') \in E\} \quad (3.1)$$

For a subset  $S \subseteq V$  of  $V$  the neighborhood of  $S$  is defined as

$$\mathbb{N}(S) := \cup_{v \in S} \mathbb{N}(v) \setminus S \quad (3.2)$$

Furthermore, we define  $\mathcal{B}_i$  as

$$\mathcal{B}_i = \{v_j \mid j \leq i\} \quad (3.3)$$



With these definitions we can specify the subroutines `EnumerateCsg`, `EnumerateCsgRec` and `EnumerateCmp` as introduced in [4].

`EnumerateCsg`

**Input** : A connected query graph  $G = (V, E)$

**Precondition:** Nodes in  $V$  are numbered according to a  
breadth-first search

**Output:** Emits all subsets of  $V$  including a connected subgraph of  $G$

**foreach**  $i \in [n - 1, \dots, 0]$  descending **do**

**emit**  $\{v_i\}$ ;

`EnumerateCsgRec`( $G, \{v_i\}, \mathcal{B}_i$ );

**end**

### Algorithm 3: `EnumerateCsg`

`EnumerateCsgRec`

$N = \mathcal{N}(S) \setminus X$ ;

**foreach**  $S' \subseteq N, S' \neq \emptyset$ , enumerate subsets first **do**

**emit**  $(S \cup S')$ ;

**end**

**foreach**  $S' \subseteq N, S' \neq \emptyset$ , enumerate subsets first **do**

`EnumerateCsgRec`( $G, (S \cup S'), (X \cup N)$ );

**end**

### Algorithm 4: `EnumerateCsgRec`

`EnumerateCmp`

**Input** : A connected query graph  $G = (V, E)$ , a connected subset  
 $S_1$

**Precondition:** Nodes in  $V$  are numbered according to a  
breadth-first search

**Output:** Emits all complements  $S_2$  for  $S_1$  such that  $(S_1, S_2)$  is a  
csg-cmp-pair)

$X = \mathcal{B}_{\min(S_1) \cup S_1}$ ;

$N = \mathcal{N}(S_1) \setminus X$ ;

**foreach**  $v_i \in N$  by descending  $i$  **do**

**emit**  $\{v_i\}$ ;

`EnumerateCsgRec`( $G, \{v_i\}, X \cup \mathcal{B}_i(N)$ );

**end**

### Algorithm 5: `EnumerateCmp`

## 3.2 Server

In order for the server not having to memorize client or session information, we want to establish a stateless communications protocol. Thus, we're conforming to the Representational state transfer (REST) architecture as laid out in [7].

## 3.3 Client

### 3.3.1 Query Graphs

Visual representations of query graphs are drawn on a canvas with a top-left origin. Each query graph consists of nodes drawn as a circle depicting the relations, with the relation name centered inside. In addition, we draw edges between two relations signalling a (direct) connection. Furthermore, we want to be able to draw labels for the cardinality of a relation beneath its respective node and show a label for the selectivity of two relations, which is drawn on the edge right between the two corresponding nodes.

**Calculations** In order to calculate the points for each represented node on the canvas we first specify some variables. By  $n$  we denote the number of nodes,  $h$  is the height of the canvas, and  $w$  is the canvas width. As restricted by the user interface, we also set the condition that  $n \geq 3$ .

Additionally, we define  $m$  as the margin to the canvas bounds, and  $r$  as the radius of a node. Taking  $m$  and  $r$  into consideration, the drawable width  $\hat{w}$  and drawable height  $\hat{h}$  of the canvas, used to form a centered layout box inside the canvas with margin  $m$ , is therefore given by subtracting  $m$  and  $r$  from either side of the canvas. Defining these variables here will simplify the specification of some of our coordinate calculation formulae later on.

$$\hat{w} = w - 2(m + r) \tag{3.4}$$

Analogously, the drawable height is given by

$$\hat{h} = h - 2(m + r) \tag{3.5}$$

In order to be able to interchange  $\hat{w}$  and  $\hat{h}$ , e.g. in the radius calculation of a cyclic query, we also set the condition that  $\hat{w} = \hat{h}$ , i.e. our canvas is a square.

One parameter that is not abstracted away however is the query graph type. As of now, this information can not be implicitly derived from the makeup of the join problem itself, using only our toolset. Automatically

classifying the graph into one of the groups as categorized in section 2.1.4 is not within the scope of this thesis.

Besides, our toolset can only calculate chain, star, complete  $k$ -ary tree, and cycle query graphs procedurally, as the only parameters required to unambiguously specify those are the number of relations and the respective query graph type. This information however is sufficient to calculate all node and edge coordinates in a single pass, i.e. with an asymptotic complexity of  $\mathcal{O}(n)$ , by using the formulae described in the following.

We use a similar approach for all query graph types: First, we'll start with the calculation of the node coordinates. Additionally, we make the assumption that the relations are ordered in a breadth-first manner in the input array given to the tree parser algorithm. The relation  $R_i$  corresponds to the  $i$ -th node  $n_i$  in the query graph, and thus the notation used to refer to this node is simply using its index  $i$  of the ordered set/array of nodes. The following paragraphs give an exhaustive walkthrough for the node coordinate calculation for all allowed query types, irrespective of implementation details. More details regarding the specific implementation are discussed in section 4.

As a second step, we calculate all the points used for drawing the graph edges. For this purpose, we need to calculate a start point  $(x_s, y_s) \in \mathbb{R} \times \mathbb{R}$  and an end point  $(x_e, y_e) \in \mathbb{R} \times \mathbb{R}$  and draw a straight line from the start point to the end point. As simple straight lines can be drawn by any rudimentary rendering engine we assume it is sufficient to calculate these two points.

## Chain

**Nodes** For chain query graphs we draw nodes at the vertical center of the canvas and set an equal spacing in between all nodes. Thus—when also considering the canvas margin  $m$  and the node radius  $r$  as defined before—the canvas point  $(x_i, y_i)$  for the node with index  $i$  is given by

$$(x_i, y_i) = (m + r + \frac{i\hat{w}}{n-1}, \frac{h}{2}) \quad (3.6)$$

**Edges** The coordinates for edges in chain query graph trees are easily specified. We take the point for the current node at index  $i$  and its predecessor at  $i - 1$  for all  $i > 0$ .

$$\begin{aligned} (x_s, y_s)_i &= (x_i, y_i) \\ (x_e, y_e)_i &= (x_{i-1}, y_{i-1}) \end{aligned} \quad (3.7)$$

## Star

**Nodes** Star query graphs are drawn by placing the node with index  $i = 0$  at the canvas' center, and all following nodes with  $i > 0$  in a circle around it, while starting (arbitrarily) at the rightmost vertically centered coordinate point ( $\theta_0 = 0^\circ$ ).

First up, we need to specify the radius of circle for nodes with  $i > 0$ . In order to allow for the highest possible number of nodes we are using the entire drawable width  $\hat{w}$ .

$$r_{star} = \frac{\hat{w}}{2} \quad (3.8)$$

For all nodes with  $i > 0$  we also define the angle for the current node by

$$\theta_i = \frac{2\pi i}{n-1} \quad (3.9)$$

This gives us an equally spaced circle of nodes for indices  $\{1, \dots, n-1\}$  since  $|\{1, \dots, n-1\}| = n-1$  and

$$\frac{\frac{2\pi(n-1)}{n-1}}{2\pi} = 1 \quad (3.10)$$

Hence, starting from the canvas center  $(\frac{w}{2}, \frac{h}{2})$  we can then use trigonometric functions to calculate the point of a respective node on this circle. The point  $(x_i, y_i)$  for the node at index  $i$  is therefore given by

$$(x_i, y_i) = \begin{cases} (\frac{w}{2}, \frac{h}{2}), & \text{if } i = 0 \\ (\frac{w}{2} + r_{star} \cos \theta_i, \frac{h}{2} + r_{star} \sin \theta_i), & \text{otherwise} \end{cases} \quad (3.11)$$

**Edges** Edges for star query graphs are drawn for all nodes at index  $i > 0$  from their respective position to the central node at  $(x_0, y_0)$ . Hence, we get

$$\begin{aligned}(x_s, y_s)_i &= (x_i, y_i) \\ (x_e, y_e)_i &= (x_0, y_0)\end{aligned}\tag{3.12}$$

**Complete k-ary Tree** The node position calculation for tree query graphs requires an additional parameter. By  $k$  we denote the degree of the tree. So far only complete  $k$ -ary query graphs can be drawn, since drawing non-complete graphs would require us to specify which tree leaves are filled, e.g. by storing `null` values in the relations array.

**Note 2.** In our toolset implementation however we assume that  $k = 2$ , i.e. representing a complete binary query graph, so that we don't need any additional user inputs for this query type. Irrespective of  $k$  this represents a "tree" type query graph and using any  $k \geq 2$  provides only very little additional insight in conveying how the algorithm works.

First, we calculate the level  $l_i$  for a given node at index  $i$ . This is given by

$$l_i = \lfloor \log_k(i + 1) \rfloor\tag{3.13}$$

Furthermore, we calculate the number of nodes on the respective level, i.e. the width  $\phi_{l_i}$  of the tree at that level, which is given by

$$\phi_{l_i} = k^{l_i}\tag{3.14}$$

In the next step, we calculate the current column  $\rho$  (is there a better name and symbol?) for the current node. In this formula we make use of the width of the tree at the  $i$ -th level as calculated beforehand.

$$\rho_i = i - k^{\phi_{l_i}+1}\tag{3.15}$$

Now we can directly calculate the  $x$ -coordinate for the current node, which is given by

$$x_i = (\rho_i + 0.5) * \frac{\hat{w}}{\phi_{l_i}} + r + m\tag{3.16}$$

Moreover, for calculating the  $y$ -coordinate we first need to calculate the distance  $d_{n,k}$  between the levels in a  $k$ -ary tree with  $n$  relations. This is given by

$$d_{n,k} = \frac{\hat{w}}{\lfloor \log_k n \rfloor}\tag{3.17}$$

Finally, this lets us calculate the  $y$ -coordinate for the node at index  $i$ :

$$y_i = l_i * d_{n,k} + r + m \quad (3.18)$$

Since we calculated  $x_i$  and  $y_i$  now the point  $(x_i, y_i)$  can be defined.

**Edges** The calculation for edge coordinates in tree query graphs requires us to calculate the index  $i_p$  of the parent node first, for all  $i > 0$ . This is given by

$$i_p = \lfloor \frac{i-1}{k} \rfloor \quad (3.19)$$

Now, we can calculate the node position for this parent node as specified in equations [3.13](#) through [3.18](#). Hence, the edge coordinates are given by

$$\begin{aligned} (x_s, y_s)_i &= (x_i, y_i) \\ (x_e, y_e)_i &= (x_{i_p}, y_{i_p}) \end{aligned} \quad (3.20)$$

## Cycle

**Nodes** A cycle query is essentially drawn as a star query with the node at the center missing. Hence, we define  $r_{cycle}$  the same way as we defined  $r_{star}$  in [\(3.8\)](#).

$$r_{cycle} = \frac{\hat{w}}{2} \quad (3.21)$$

Similarly to [\(3.9\)](#), the angle for the current node, denoted  $\theta_i$ , is given by

$$\theta_i = \frac{2\pi(i+1)}{n} \quad (3.22)$$

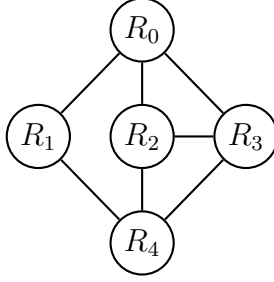
Again, we can use trigonometric functions to calculate the node coordinates. Thus, the point  $(x_i, y_i)$  for a respective node is given by

$$(x_i, y_i) = (r_{cycle} \cos \theta_i + r_{cycle} + r + m, r_{cycle} \sin \theta_i + r_{cycle} + r + m) \quad (3.23)$$

**Edges** In order to calculate edge coordinates for cycle queries we calculate the successor angle  $\theta_{i+1}$  for all  $i$  and with this result the successor coordinate  $(x_{i+1}, y_{i+1})$ . Thus, the edge coordinates are given by

$$\begin{aligned} (x_s, y_s)_i &= (x_i, y_i) \\ (x_e, y_e)_i &= (x_{i+1}, y_{i+1}) \end{aligned} \quad (3.24)$$

**Moerkotte 2018** As this work serves as a supplementary material to the book “Building Query Compilers” by Prof. Guido Moerkotte [5], we also allow to draw the query graph used as an example to explain the EnumerateCsg subroutine of the DPccp algorithm.



**Nodes** This query graph is distinct to the other query graphs calculated beforehand as it can be categorized as a cyclic query graph, but not a cycle one. Since there is a large number of cyclic query graphs even for  $n \leq 10$  and it cannot be uniquely specified by a small number of parameters, the dimensions of this query graph are hardcoded in the implementation. In order to allow a layout box of arbitrary size we however calculate the horizontal center  $c_x$ , vertical center  $c_y$ , and layout offset  $o$  as follows:

$$c_x = \frac{w}{2} \quad (3.25)$$

$$c_y = \frac{h}{2} \quad (3.26)$$

$$o = r + m \quad (3.27)$$

Now, we can calculate the respective points  $(x_i, y_i)$  for all relations  $\{i \in \mathbb{N} | 0 \leq i < 5 = n\}$ :

$$\begin{aligned}
(x_0, y_0) &= (c_x, o) \\
(x_1, y_1) &= (o, c_y) \\
(x_2, y_2) &= (c_x, c_y) \\
(x_3, y_3) &= (w - o, c_y) \\
(x_4, y_4) &= (c_x, h - o)
\end{aligned} \quad (3.28)$$

**Edges** Edges coordinates for this query graph are similarly hardcoded and drawn for the following node coordinate-pairs:

$$\begin{aligned} &\{((x_0, y_0), (x_1, y_1)), ((x_0, y_0), (x_2, y_2)), ((x_0, y_0), (x_3, y_3)), \\ &((x_1, y_1), (x_4, y_4)), \\ &((x_2, y_2), (x_3, y_3)), ((x_2, y_2), (x_4, y_4)), \\ &((x_3, y_3), (x_4, y_4))\} \end{aligned}$$

**Labels** In addition to visualizing the relations as nodes and connections as edges in the query graph we want to draw labels for the cardinality of a relation and the selectivity between two relations.

**Cardinality** The cardinality label is drawn right under a relation's node. We make the assumption that the width of the node's text  $\sigma$  and its height  $\tau$  at the specified font size is either known by being provided by the canvas' layout engine, or can be calculated otherwise. The label is drawn by simply centering it on the corresponding node and moving it down by  $\frac{\tau}{2}$ .

**Selectivity** The selectivity label is drawn at the center point of the straight line edge between two connected nodes. The coordinates for this point  $(l_{cx}, l_{cy})$  are given by

$$l_{cx} = \min(x_s, x_e) + \frac{|x_s - x_e|}{2} \quad (3.29)$$

and analogously

$$l_{cy} = \min(y_s, y_e) + \frac{|y_s - y_e|}{2} \quad (3.30)$$

Since the label has a height and width dimension itself this point however cannot be used as the origin of the label. Most layout engines' implementations for drawing rectangles accept parameters for their origin, width and height, with a top-left origin. Thus, we have to subtract half of its height and width respectively to get the point  $(l_{ox}, l_{oy})$  specifying the label's origin. Analogously to the cardinality frame calculation we also make the assumption that the width of the text  $\sigma$  and its height  $\tau$  are known. Furthermore, we add a small constant  $c$  to those size dimensions specifying the margin of the label's text to its enclosing box's outer bounds, giving an equal margin of  $\frac{c}{2}$  on either side.

$$(l_{ox}, l_{oy}) = (l_{cx} - \frac{\sigma + c}{2}, l_{cy} - \frac{\tau + c}{2}) \quad (3.31)$$



## 4 Implementation

### 4.1 API Requirements and Design Goals

In this section we define and shortly describe some objectives that the toolset has to fulfil.

**Orthogonality** Most importantly, the property of orthogonality as defined by Eric S. Raymond in *The Art of Unix Programming* [6] has to be fulfilled for the visualization code with regards to the algorithm implementation's code, i.e. it should not change its behavior or state.

**Declarativity** One of the main goals of this framework is to minimize the amount of code or configuration that has to be added to existing algorithms' implementations in order to make use of the visualization for its intended purpose in exploration or debugging. The visualization should therefore be specified as declaratively as possible, with the user specifying *only* the relevant parameters—while minimizing the amount of required boilerplate code.

**Separation of Concerns** With the separation of the complex algorithm and user interface/visualization logic in mind it is imperative to keep the amount of code required to create new visualizations as minimal as possible. Thus, we organize their architecture in a client-server relationship, whereby the client specifies the join problem parameters and configuration and the server can generate a client-interpretable output to work with.

The advantages of this approach are multifold:

- The server part and UI part respectively can be updated or even completely exchanged without changing the other
- More complex join ordering algorithms can be performed on a (more powerful) server
- We are able to access the application via a web user interface without sacrificing any performance by limiting ourselves with a slow, interpreting JavaScript engine, but we are able to run compiled and thus more optimized and therefore faster code
- It allows us to formally specify interfaces between client and server

**Flexibility** The visualization tools should be abstract enough that it is not catered towards a specific algorithm, but can be used to visualize a wide range of algorithms, especially ones it has not been explicitly programmed for.

Since there is a large number of data structures an algorithm can use for its implementation we limit ourselves to procedures involving a query graph with information about its number of relations, the query graph type as specified in [2.1.4](#), and its cardinalities and selectivities. This limitation is necessary to enforce a manageable scope for creating a minimum viable product.

More pragmatically, this means that the algorithm has to conform to the Visualizable Go type we define as follows:

```
type Visualizable func(QG QueryGraph, JTC JoinTreeCreator) *Tree
```

Figure 5: Visualizable type conformance requirement

**Example 3.** This way, a Go function `DPccp` that conforms to this type can be instructed to produce the output log necessary for the visualization by simply calling

```
Visualize(DPccp, QG, JTC)
```

instead of the algorithm itself. The `visualize` function then automatically sets the appropriate flags to enable visualization output and can allow the algorithm to run without the visualization code and thus no (significant) runtime or memory overhead.

**High signal-to-noise ratio** In many text editors it is possible to use code folding, enabled through the orthogonality property as mentioned in section [4.1](#). This allows us to completely show or hide the visualization code when required. Thus, we can achieve an unobtrusive debugging tool that can be written alongside an algorithm’s implementation code and doesn’t decrease its signal-to-noise ratio.

**Query Graph Planarity** The visualized query graph should be drawn in a way that no edges are overlapping. This is possible when the query graph is *planar* [\[8\]](#).

## 4.2 Architecture

The following figure [6](#) illustrates a sequence diagram for the client-server architecture used for this project. At first, we retrieve all available algorithms, denoted by  $A$ , from the server. This is used to asynchronously generate the available options in the algorithm selection picker. Retrieving the algorithms from the server has the advantage that new algorithm visualizations can be added server-side without changing, re-installing or re-deploying any client code.

Next up, the user can select all the parameters for the visualization. The first parameter is the algorithm  $a \in A$  to be visualized. Moreover, the query graph type  $qg \in \{\text{“chain”}, \text{“star”}, \text{“cycle”}, \text{“tree”}, \text{“moerkotte”}\}$  can be specified. Contrary to the algorithm, the query type is not fetched from the server as the client implementation explicitly has to know the available query graphs at build time, since they are used for calculating the position of different query graph nodes as detailed in section [3.3.1](#).

Furthermore, the number of relations in the query graph  $n$  has to be specified, whereby  $n \in [3, 10]$ . The upper boundary for  $n$  is set to 10 to prevent calculations that are too complex and might be set even lower for algorithms other than **DPccp**, where the execution time and produced output is even larger.

Depending on the previously selected parameters the algorithm’s steps are fetched upon clicking the “Recalculate” button. This will trigger another HTTP server request that transmits the algorithm (:a), number of relations (:r) and graph type (:g) as path parameters.

As the server receives this request, it calls the corresponding algorithm by using the **Visualize()** function specified in section [4.6.2](#). This produces a JSON response containing the query graph  $QG$  with information about cardinalities, selectivities, and neighbors, as well as the algorithm’s generated visualization steps  $S$ .

Upon receiving this response, the client renders the query graph including labels for cardinalities and selectivities, and edges for connected query graph nodes. Moreover, the algorithm’s steps are rendered in a hierarchical table.

As the rendering completes, the number of total steps  $|S|$  is calculated and the user is presented with two buttons to move to either the next or the previous step. After doing so, the user interface, including the query graph and the variable table, are re-rendered accordingly.

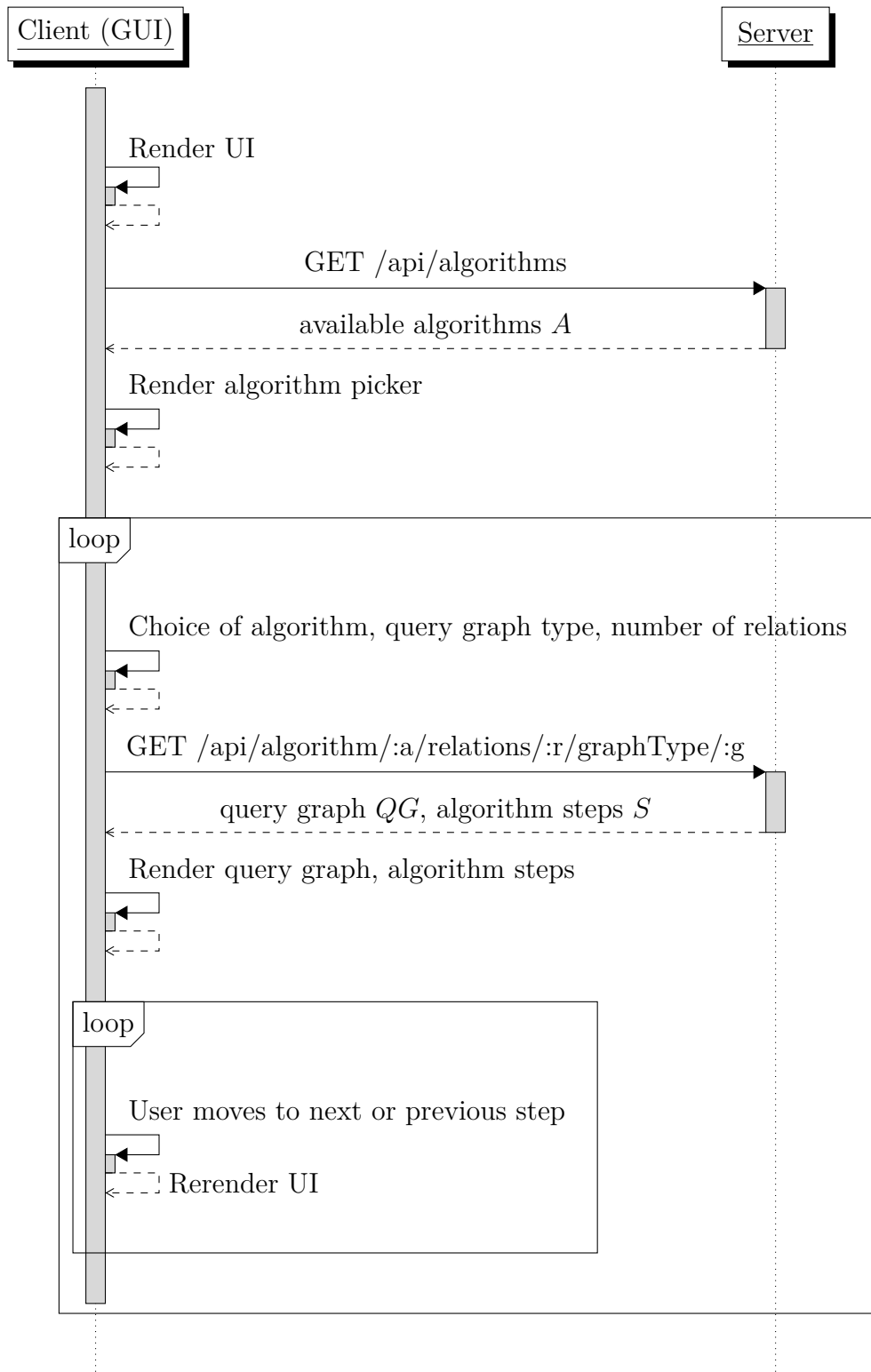


Figure 6: UML sequence diagram of the client-server communication

### 4.3 Choice of Technology

For the technology used in the implementation of both client and server code we want to maximize

- (i) **Accessibility**, a minimal amount of effort required to access the application
- (ii) **Maintainability**, a high degree of technological similarities with related projects
- (iii) **Tried-and-testedness**, a large user base and well-received reference applications

In order to maximize (i) we create an application that runs in a web browser and doesn't need any user-initiated installation or update procedures. With the 2017 released WebAssembly (Wasm)<sup>17</sup> portable byte-code standard still being in its infancy at around 90% browser support<sup>18</sup> and being recommended by the World Wide Web Consortium not until December 5th 2019<sup>19</sup>, this leaves us with JavaScript as the only viable language for the client-side code.

Additionally, in order to modularize the user interface and adding the ability to import existing user interface components, we use `React.js`<sup>20</sup>, as of writing this thesis the most popular JavaScript library for building user interfaces measured by “Stars” on the popular repository hosting website GitHub<sup>21</sup>.

For the server code and algorithm implementations we use the Go programming language and `gin-gonic`<sup>22</sup>, which at the time of writing this thesis is the most-starred HTTP framework on GitHub<sup>23</sup>.

By using two popular libraries and languages we ensure a good community support, a good availability of learning or tooling resources, and continuous development of the libraries. Therefore, contributions to the visualization toolset by developers other than the original author have a lower barrier of entry.

---

<sup>17</sup><https://webassembly.org> (last accessed: October 27, 2020)

<sup>18</sup><https://caniuse.com/wasm> (last accessed: October 27, 2020)

<sup>19</sup><https://www.w3.org/TR/wasm-core-1/> (last accessed: October 27, 2020)

<sup>20</sup><https://reactjs.org> (last accessed: October 27, 2020)

<sup>21</sup><https://github.com/search?l=JavaScript&o=desc&q=language%3AJavaScript&s=stars&type=Repositories> (last accessed: October 27, 2020)

<sup>22</sup><https://github.com/gin-gonic/gin> (last accessed: October 27, 2020)

<sup>23</sup><https://github.com/search?l=Go&o=desc&q=http&s=stars&type=Repositories> (last accessed: October 27, 2020)

## 4.4 Tooling

### 4.4.1 Join Problem Generator

We represent the join problems in a JSON format as defined in section 4.7.1. Some JSON files are taken from the supplementary material to the “IE 630 Query Optimization” course exercises at the University of Mannheim<sup>24</sup>, which contains examples for join problems with a query type  $QT \in \{\text{Chain}, \text{Clique}, \text{Cycle}, \text{Star}\}$  and a number of relations  $n \in [2, 10]$ . All further references to `joinProblemParser.go` or `infra.go` refer to the two (modified) files taken from this supplementary material.

Additionally, we create a join problem generator for complete  $k$ -ary trees with a degree  $k$  and a number of relations  $n$ . These calculations can also be conducted in  $\mathcal{O}(n)$ , as demonstrated in the following.

Since selectivities are symmetrical, they are stored once and once only in the JSON. This means, if an edge  $(R_0, R_1)$  in the query graph exists, we *don't* store  $(R_1, R_0)$  explicitly as well. However, we store the selectivities in a breadth-first manner starting from  $R_0$ , i.e. the edge always points from the relation with the smaller index to the larger one. More formally, we can say that the following implication holds:

$$\forall(R_i, R_j) \in E \implies i < j \quad (4.1)$$

#### Complete k-ary Tree

**Note 3.** Similar to the drawing of  $k$ -ary trees we implement the generator only for  $k = 2$ , despite our formulae giving the calculation steps for an arbitrary degree  $k$ . This is mostly to avoid having to convert the result from Go's `Pow` and `Log` result from `float64` back to `uint64`. For that reason we use a custom implementation of a logarithm returning a `uint`, which so far only works for base 2.

**Selectivities** Since implication 4.1 holds, we only need to calculate selectivities for nodes which have children. Thus, first of all, we calculate the number of vertices with children  $n_c$ , i.e. the number of relation-pairs for which selectivity entries exist in the resulting JSON file. In the case of complete  $k$ -ary trees, when enumerating the relations in a breadth-first manner, this number is given by

$$n_c = \#\{v \in V | \text{hasChildren}(v)\} = \lceil \frac{n}{k} \rceil \quad (4.2)$$

---

<sup>24</sup><https://www.wim.uni-mannheim.de/moerkotte/teaching/courses/query-optimization/> (last accessed: October 27, 2020)

Accordingly, we specify the set of vertices with children  $V_c$ , starting with index 0.

$$V_c := \{v_i\}_{i=0}^{n_c-1} \quad (4.3)$$

**Note 4.** Let  $I$  be the index set for all relations

$$I = \{0, \dots, n-1\} \quad (4.4)$$

Then, since our  $k$ -ary tree is complete and breadth-first, we can infer that if a node at index  $i$  does not have children, any node at a larger index  $j$  doesn't either. Likewise, any node with an index  $i < n_c$  does have children.

$$\forall i, j \in I, i < j: v_i \notin V_c \implies v_j \notin V_c \quad (4.5)$$

$$\forall i < n_c: \exists v_i \in V_c \quad (4.6)$$

This is why it suffices to consider nodes with index  $i < n_c$  when generating selectivity entries.

Furthermore, we need to determine the level  $l(v_i)$  of each node  $v_i$  in the tree. For its respective index  $i$ , this is given by

$$l_i = l(v_i) = \log_k(i+1) \quad (4.7)$$

As a next step, we then define the set of vertices in the levels below  $l_i$ , denoted  $V_{l_i < j}$ , as follows:

$$V_{l_i < j} := \{v_j \in V | l(v_j) < l(v_i)\} \quad (4.8)$$

Now, we need the cardinality of this set, denoted by  $n_{j < i}$ , to get the number of nodes on all level smaller than  $l_i$ . This can be efficiently calculated using the following equation:

$$n_{j < i} := |V_{l_i < j}| = k^{l(v_i)+1} - 1 \quad (4.9)$$

This allows us to calculate the number of children for each level  $l(v_i)$ .

$$\hat{l}_i = \min(k^{l(v_i)+1}, n - n_{j < i}) \quad (4.10)$$

**Note 5.** Note that  $l(v_i)$  is not necessarily equal to the width of the level in a full  $k$ -ary tree, which is given by  $k^{l(v_i)+1}$ , since some leaf nodes could be missing in the bottom level of the tree.

Furthermore, we calculate the “column”  $c_i$  of  $v_i$ , given by

$$c_i = i - k^{l(v_i)} + 1 \quad (4.11)$$

This lets us calculate the number of children  $\hat{n}_i$  for a node at index  $i$ :

$$\hat{n}_i = \min(k, \hat{l} - c_i * k) \quad (4.12)$$

**Note 6.** This  $\hat{n}_i$  is equal to  $k$  for most internal nodes in a sufficiently large tree. However, if  $\log_k(n) \notin \mathbb{N}$  there could be an internal node with  $\hat{n}_i < k$ , i.e. not all possible leaf nodes for the last node with children are filled.

Now we can define the neighbors of  $v_i$  as

$$\mathbb{N}_i = \{x_{i,j}\}_{j=0}^{\hat{n}_i-1} \quad (4.13)$$

We also calculate the lowest index of the node’s neighbors  $j_{\min \mathbb{N}_i}$ .

$$j_{\min \mathbb{N}_i} = ik + 1 \quad (4.14)$$

For each neighbor  $x_{i,j} \in \mathbb{N}_i$  we then calculate the offset for its  $j$  index.

$$\tau_{i,j} = j \mod k \quad (4.15)$$

The current index of the neighbor is thus given by

$$\sigma_{i,j} = j_{\min \mathbb{N}_i} + \tau_{i,j} \quad (4.16)$$

Finally, we can set all  $x_{i,j} \in \mathbb{N}_i$  as  $\sigma_{i,j}$

$$x_{i,j} = \sigma_{i,j} \quad (4.17)$$

Let us now define the set of all neighbors  $\mathbb{N}$  for all indices  $i < n$ :

$$\mathbb{N} = \{\mathbb{N}_i\}_{i=0}^{n-1} \quad (4.18)$$

We can now set a selectivity for each  $(x_i, j) \in \mathbb{N}_i \in \mathbb{N}$ , where the selectivity  $s_{i,j}$  is a random variable following a uniform distribution over the set  $\mathbb{R} \cap [0, 1]$ .

A Go implementation example of the symmetrical neighbor calculation is illustrated in algorithm [6](#). The assignment of selectivities to these entries is trivial and thus not illustrated here, but can be found in the supplementary code.



**Note 7.** The variable names in algorithm 6 correspond to the names used in the previously introduced mathematical definitions. In the supplementary code to this thesis longer, more descriptive variable names are used.

```

symNeighborEntry := func(i uint, n_c uint) string {
    l_i := uint(log2_64(uint64(i + 1)))
    n_j_leq_i := uint(1<<(l_i+1) - 1)
    l_hat_i := min(1<<(l_i+1), n-n_j_leq_i)
    c_i := i - 1<<(l_i) + 1
    n_hat_i := min(k, l_hat_i-c_i*k)

    N := make([]string, n_c)
    j_minN_i := i*k + 1
    for j := range N {
        tau_ij := uint(j) % k
        sigma_ij := j_minN_i + tau_ij
        s := strconv.FormatUint(uint64(sigma_ij), 10)
        N[j] = s
    }
    return strings.Join(N, ",")
}
symNeighbors := func() map[uint]string {
    dict := map[uint]string{}
    n_c := math.Floor(float64(n) / float64(k))
    for i := uint(0); float64(i) < n_c; i++ {
        dict[i] = symNeighborEntry(i, n)
    }
    return dict
}
neighbors := symNeighbors()

```

**Algorithm 6:** Go implementation to calculate symmetrical neighbors.  $k$  is the degree of the tree and  $n$  the number of relations. Both must be set before using them in the above lambda expressions (Go function literals).

**Neighbors** The set of neighbors does not make use of the symmetry of selectivities and thus stores all neighbors for each relation explicitly. Thus, we append the parent node  $\rho_i$  to the set of neighbors  $N$  calculated for the selectivity generation. This assignment is trivial and thus the algorithm is not provided here.

**Relations** Specifying the relations  $R$  is trivial.

$$R := \{R_i\}_{i=0}^{n-1} \quad (4.19)$$

where the cardinality  $c_{r_i}$  of each relation  $r_i$  is a random variable following uniform distribution over the set  $\mathbb{R} \cap [0, 10000]$ .

Likewise, the creation of the Go `relations` slice is trivial and therefore we don't illustrate the algorithm here.

## 4.5 Data Structures

### 4.5.1 Join Ordering

In order to parse and represent a join problem from a JSON file we use the already existing data structures `JSONRelation` and `JSONJoinProblem` from `joinProblemParser.go`. Both are implicitly marshalled and unmarshalled when converted from or to JSON because of the built-in JSON encoding and decoding for custom data structures using the Go standard library's `encoding/json` package<sup>25</sup>. However we add `struct` tags to explicitly set the corresponding JSON keys.

```
type JSONRelation struct {
    Cardinality float64 `json:"cardinality"`
    Name        string  `json:"name"`
    ProblemID    uint    `json:"problemID"`
    RelationID   uint    `json:"relationID"`
}
```

Data Structure 1: JSONRelation type

```
type JSONJoinProblem struct {
    ProblemID      uint    `json:"problemID"`
    Neighbors      map[uint]string `json:"neighbors"`
    NumberOfRelations uint    `json:"numberOfRelations"`
    Relations      []JSONRelation `json:"relations"`
    Selectivities  map[string]float64 `json:"selectivities"`
}
```

Data Structure 2: JSONJoinProblem type

---

<sup>25</sup><https://golang.org/pkg/encoding/json/> (last accessed: October 27, 2020)

We'll take the `QueryGraph` struct from `infra.go` and add a `map` to a `uint` slice of neighbors for each relation, which itself is represented as a `uint` bit vector and used as the map's key.

```
type QueryGraph struct {
    R []uint          `json:"relationCardinalities"`
    S map[uint]float64 `json:"selectivities"`
    N map[uint][]uint  `json:"neighbors"` // Added neighbor map
}
```

### Data Structure 3: QueryGraph type

Also, we introduce a struct for representing csg-cmp-pairs.

```
type CsgCmpPair struct {
    S1 uint
    S2 uint
}
```

### Data Structure 4: CsgCmpPair type

#### 4.5.2 Visualization

For the visualization we define some data structures used to declaratively specify algorithms, their visualization routines and their atomic components, such as visualization steps and the subroutine's configured observed relations.

**Algorithm** First, we specify an `Algorithm` type that can be returned to the client when requesting the list of algorithms which can be visualized. Each algorithm contains a `Label` string describing the algorithm, e.g. "DPccp" as well as a `Value` string giving the algorithm's identifier for server requests. This `Value` string is by convention a camel-cased name of the algorithm starting with a lower-case character, whereas the `Label` can also contain special or whitespace characters. This differentiation is made to allow human-readable algorithm names as well as URL-encodable identifiers for the API's path parameter.

```

type Algorithm struct {
    Label string `json:"label"`
    Value string `json:"value"`
}

```

#### Data Structure 5: Algorithm type

**Variable Table** We want to keep track of a set of user-defined relation variables to aid understanding the algorithm's state's history. For this purpose, we first introduce the `ObservedRelation` type that can be used to specify which sets of relations should be observed. It contains an `Identifier` string that corresponds to the variable's name, as well as a `Color` field, which allows to visually distinguish different variables. This is specified using a `color.RGBA` struct from the `image/color` package, which can be found in Go's standard library.

```

type ObservedRelation struct {
    Identifier string `json:"identifier"`
    Color      color.RGBA `json:"color"`
}

```

#### Data Structure 6: ObservedRelation type

Furthermore, we introduce a `VariableTableEntry` type that can be specified individually for each observed relation, in each atomic visualization step. In our application, this entry is a `uint` slice depicting the current variable's set of relations in bit vector representation.

```

type VariableTableEntry []uint

```

#### Data Structure 7: VariableTableEntry type

These variable table entries are stored in a `VariableTableRow` map. A string identifying the relation that corresponds to the `Identifier` in the `ObservedRelation` as the key of the map.

```
type VariableTableRow map[string]VariableTableEntry
```

#### Data Structure 8: VariableTableRow type

**GraphState** The state of the query graph drawn in the user interface must be represented in order to unambiguously specify its configuration. We want to be able to give each node in the graph a color that corresponds to the `Color` field of the `ObservedRelation`. Hence, we first define a `NodeColor` type storing the `Index` of the relation, as well as the respective `Color`.

```
type NodeColor struct {
    NodeIndex uint `json:"nodeIndex"`
    Color      color.RGBA `json:"color"`
}
```

#### Data Structure 9: NodeColor type

The graph state is thus a slice containing all node colors. Since we explicitly assign a color to each node in every visualization step, the length of the slice is therefore equal to the number of relations  $n$  in the graph.

```
type GraphState struct {
    NodeColors []NodeColor `json:"nodeColors"`
}
```

#### Data Structure 10: GraphState type

**Steps and Routines** Finally, the data structures defined beforehand can be used to construct higher-level constructs such as the *atomic* visualization step, represented as a `VisualizationStep` type. This type contains the `GraphState` for the current step, as well as the variables stored as a `VariableTableRow`. In order to identify a step in equivalence checks, we also assign a UUID, which is generated using Google's own `github.com/google/uuid`<sup>26</sup> package. This UUID is a 128 bit UUID based on RFC4122<sup>27</sup>, which can be

<sup>26</sup><https://github.com/google/uuid> (last accessed: October 27, 2020)

<sup>27</sup><https://tools.ietf.org/html/rfc4122> (last accessed: October 27, 2020)

easily converted to a `string` using the `uuid.New().String()` function. This UUID is necessary as the reference (i.e. memory address) to the exact instance of a struct is lost when marshalling it into a JSON.

```
type VisualizationStep struct {
    GraphState GraphState    `json:"graphState"`
    Variables   VariableTableRow `json:"variables"`
    UUID        string       `json:"uuid"`
}
```

**Data Structure 11:** VisualizationStep type

Finally, we have defined all the types necessary to define a high-level `VisualizationRoutine`. This routine contains a `Name` field to describe its corresponding algorithm, e.g. “EnumerateCsg”, as well as fields to store an `ObservedRelation` slice to specify the sets of relations that should be kept track of for this routine. Furthermore, we store a slice of all the routines’ steps. This `Steps` slice is generically implemented using a pointer to Go’s interface type, as it can be filled with either an atomic `VisualizationStep` or another `VisualizationRoutine`. This allows for subroutines or even recursive calls of other subroutines and creates a hierarchical structure of all steps in the visualization’s JSON representation.

```
type VisualizationRoutine struct {
    Name           string           `json:"name"`
    ObservedRelations []ObservedRelation `json:"observedRelations"`
    Steps          []*interface{ }    `json:"steps"`
}
```

**Data Structure 12:** VisualizationRoutine type

This `VisualizationRoutine` is transferred as a top-level object when making a server request and thus we don’t need to specify any more data structures.

## 4.6 Server Implementation

### 4.6.1 Utility Functions

In order to provide a starting point for the algorithm implementation, we use a modified and extended version of the `infra.go` and `joinproblemparser.go`

file provided for the exercises in the Query Optimization (IE 630) course at the Chair of Practical Computer Science III at the University of Mannheim.

All modifications are briefly discussed in the following.

**Simplification of SetMinus** We simplify the implementation for the `SetMinus` operation by omitting the use of a temporary helper variable for the result calculation. This is equivalent as it produces exactly the same assembly output as the old, commented-out code<sup>28</sup>.

```
func SetMinus(S1 uint, S2 uint, length uint) uint {
    // mask := uint((1 << length) - 1)
    // temp := S1 & S2
    // temp = ^temp
    // temp &= mask
    // return S1 & temp
    return S1 & ^S2
}
```

**Algorithm 7:** Go implementation of `SetMinus`

**Addition of PowerSet** For the implementation of `DPccp` we need an additional function that returns not just the subsets of an (`unsigned int`) bit vector  $S$  as does `Subsets`, but its power set excluding the empty set,  $\mathcal{P}(S) \setminus \{\emptyset\}$ .

```
func PowerSet(S uint) []uint {
    subsets := Subsets(S)
    if len(subsets) == 1 && subsets[0] == S {
        return subsets
    }
    return append(subsets, S)
}
```

**Algorithm 8:** Go implementation of `PowerSet`

Furthermore, we add a function to calculate the minimum element of a `[]uint` slice, as well as a function to get the minimum set bit index for a

---

<sup>28</sup>using amd64 gc 1.15 compiler, compiled on <https://godbolt.org> (last accessed: October 27, 2020)

(sub-)set of relations  $S$ , which makes use of the former.

```

func MinUIntSlice(slice []uint) uint {
    if len(slice) == 0 {
        return uint(0)
    }
    min := slice[0]
    for _, value := range slice {
        if min > value {
            min = value
        }
    }
    return min
}

func MinUIntSetBitIndex(S uint) uint {
    setIndexes := IdxsOfSetBits(S)
    minIndex := MinUIntSlice(setIndexes)
    return minIndex
}

```

**Algorithm 9:** Go implementation of MinUIntSlice and MinUIntSetBitIndex

Moreover, we'll add a method to retrieve the neighborhood for a subset of relations  $S$ .

```

func Neighbors(QG QueryGraph, S uint) uint {
    indexes := IdxsOfSetBits(S)
    result := uint(0)
    for _, index := range indexes {
        for _, neighbor := range QG.N[index] {
            result = result | (1 << neighbor)
        }
    }
    n := uint(len(QG.R))
    return SetMinus(result, S, n)
}

```

**Algorithm 10:** Neighbors function for a set of relations



### 4.6.2 Visualization

For the visualization we create a `visualization` package to handle all operations related to the visualization itself.

Here, we keep track of an exported `VisualizationOn` Boolean variable to indicate whether the visualization code within the algorithm's implementation should be executed or not. Furthermore, we define two package-local variables:

```
var routines = []*VisualizationRoutine{}
var stack = []*VisualizationRoutine{}
```

Both variables are slices of pointers to the `VisualizationRoutine` type defined in figure 4.5.2. They both fulfill different purposes: the former keeps track of consecutively executed visualization routines, whereas the latter stores the execution stack of the current routine, which allow for stacked or recursive routine calls. In order to enforce modularity, those variables cannot be changed directly, but the exported functions to start new routines or add atomic visualization steps have to be used. All available functions will be discussed in the following.

First of all, we create a `visualize` function to change the execution context of a join ordering algorithm.

```
func Visualize(visualization Visualizable,
               QG QueryGraph,
               JTC JoinTreeCreator) []*VisualizationRoutine
```

This `visualize` function can be used to call a function that conforms to the `Visualizable` type as defined in 5, one example being our implementation of `DPccp`. This will automatically set the `VisualizationOn` Boolean to `true`, execute the algorithm, and set it back to its previous value after finishing the execution. Moreover, it will return all the emitted routines as a `*VisualizationRoutine` slice and reset all local variables after execution.

**Example 4.** An implementation of the `DPccp` algorithm defined through the following function

```
func DPccp(QG QueryGraph, JTC JoinTreeCreator) *Tree
```

can be visualized by calling

```
Visualize(DPccp, QG, JTC)
```

instead of directly calling

DPccp(QG, JTC)

**Note 8.** Notice that the function definition of `DPccp` doesn't have to be changed from its original implementation at all here. Thus, potentially already existing calls to `DPccp` don't have to be modified.

Within the function implementation we can now use the exported visualization functions to start new routines and add new visualization steps. Before adding the first step, we have to call the following function at least once:

**func** StartVisualizationRoutine(routine \*VisualizationRoutine)

As the name implies, this function can be used to start a new visualization routine. This will add a new routine to the `routines` slice if the current call stack is empty or push it to the stack if there's already a function being executed. Notice that whether the function is introduced as a consecutive routine or added to the call stack is determined by the `visualization` package implicitly, for the convenience of the package's user. An example how this call stack could look like for `DPccp` at a certain point in its execution is shown in figure 7, with the most recent subroutine pushed onto the top of the stack.

EnumerateCsgRec
EnumerateCsgRec
EnumerateCsgRec
EnumerateCsg
DPccp

Figure 7: Sample execution stack for `DPccp`

After starting a visualization routine we either start a new routine that is added onto the stack or add a visualization step using the following function, which takes a `QueryGraph` and a `VariableTable` as its parameters.

**func** AddVisualizationStep(QG QueryGraph, relations VariableTable)

The implementation of this function will automatically construct a new `GraphState` from information in the passed `VariableTable` by retrieving its respective color configuration. This makes sure the observed relations and their corresponding nodes in the query graph have the same color without the

need to specify it separately or explicitly creating the `GraphState`. Furthermore, a UUID is assigned to the visualization step, which is used for efficient equivalence checks in the client implementation.

```

func AddVisualizationStep(QG QueryGraph, relations VariableTable) {
    n := uint(len(QG.R))
    nodeColors := []NodeColor{}
    currentStackIndex := len(stack) - 1
    observedRelations := stack[currentStackIndex].ObservedRelations
    for i := n - 1; int(i-1) >= -1; i-- {
        for _, relation := range observedRelations {
            relationIndexes := relations[relation.Identifier]
            if contains(relationIndexes, i) {
                color := relation.Color
                nodeColor := NodeColor{NodeIndex: i, Color: color}
                nodeColors = append(nodeColors, nodeColor)
            }
        }
    }
    graphState := GraphState{NodeColors: nodeColors}
    uuid := uuid.New().String()
    step := &VisualizationStep{GraphState: graphState,
                               Variables: relations,
                               UUID: uuid}
    currentRoutine := stack[currentStackIndex]
    var v interface{} = step
    currentRoutine.Steps = append(currentRoutine.Steps, &v)
}

```

**Algorithm 11:** Go implementation of the `AddVisualizationStep` function to create a new atomic visualization step.

**Note 9.** For the time being, we make the assumption that only one node color can be assigned to each node in the visualized query graph. Thus, we must make sure that the sets of all observed relations are disjoint. In order to formalize this requirement we define the set of observed relations  $O \subseteq R$ . Assume there are  $v$  observed variables. For a concise mathematical definition we assume variable keys are integers  $i \in \mathbb{N}$  from 0 to  $v - 1$ . Notice these keys should be interpreted as keys of a dictionary or map data structure and

don't necessarily correspond to the index of the relation.

$$O := \{O_i\}_{i=0}^{v-1} \quad (4.20)$$

Thus, the following equation must hold

$$\bigcap_{i=0}^{v-1} O_i \stackrel{!}{=} \emptyset \quad (4.21)$$

However, the union of these sets  $\bigcup_{i=0}^{v-1} O_i$  does not necessarily have to be equal to the set of all relations  $R$ , but it suffices if  $O \subseteq R$ . If  $O \neq R$  we can just fall back to a default color. This could be handled either in the implementation of `AddVisualizationStep` or, alternatively, in the client-side implementation when drawing the query graph, which is how the supplementary code handles  $O \subset R$ .

**Excursus: Impact on DPccp implementation** In `EnumerateCsgRec` inside `DPccp` we want to visualize the sets of relations  $S$  and  $X$ , among others. Assumption [4.21](#) can then be fulfilled if we slightly change the definition of  $\mathcal{B}_i$ , so that inside `EnumerateCsgRec`  $S \cap X = \emptyset$  holds.

**Conjecture 1.**  $\mathcal{B}_i := \{v_j | j \leq i\}$  can be redefined to  $\mathcal{B}_i := \{v_j | j < i\}$  without changing the emits of the algorithm, i.e. in every call of `EnumerateCsgRec` we can still assume that  $\{v_i\} \notin S', \forall S' \subseteq N$ .

*Proof.* Let us start at the `EnumerateCsgRec` call inside `EnumerateCsg`. The variables  $S := \{v_i\}$  and  $X := \mathcal{B}_i$  are used as parameters to `EnumerateCsgRec` and therefore, inside this call,  $\{v_i\} \in S$ . This will also hold for all recursive calls as the recursive set, denoted  $S_{rec}$ , will be defined as  $S_{rec} := S' \cup S \supseteq S$ . From the definition of  $N(S)$  it follows that  $v \notin N(S) \forall v \in S$  and thus  $v_i \notin N(S)$ . Since  $N := N(S) \setminus X$  it follows that  $N \subseteq N(S)$ , from which it transitively follows that  $\{v_i\} \notin N$ . Thus, since  $S' \in \mathcal{P}(N) \setminus \emptyset$  and therefore  $S' \subseteq N$ , it follows that  $\{v_i\} \notin S'$ . □

### 4.6.3 DPccp

In the following we discuss the Go implementation of `DPccp` and its enumeration routines, namely `EnumerateCsg`, `EnumerateCsgRec`, and `EnumerateCmp`. We begin with the enumeration algorithms so we can later use them in the actual `DPccp` algorithm. For all algorithms, we implement the intersection of two arbitrary sets  $S_1$  and  $S_2$ ,  $S_1 \cap S_2$ , by using a bitwise AND (`&`) and their

union  $S_1 \cup S_2$  using a bitwise OR ( $|$ ).

`EnumerateCsg` will calculate all connected subgraphs in the query graph in a breadth-first manner, beginning with the relation with the highest index. Thus, the slice of relations  $R$  of the query graph  $QG$  is traversed in descending order. As discussed in [2.1.7](#) the relations are stored as a bit vector and can be individually accessed by shifting 1 by the respective relation's index to the left. We make use of arithmetic integer overflow for the `for` loop's exit condition as  $i \geq 0$  cannot be used as the `uint` cannot be smaller than 0. In order to get the set of relations  $\mathcal{B}_i$  we shift 1 by  $(i + 1) - 1$  to the left, leaving us with a 64-bit unsigned integer with all bits set to 1 at index  $j < i$  and  $64 - (i - 1)$  leading zeroes, thus representing relations  $\{R_j \in R | j < i\}$ . We keep track of a subgraphs `[]uint` slice which collects the recursively constructed subgraphs from `EnumerateCsgRec` and return the result of this slice after appending all emitted subgraphs.

```
func EnumerateCsg(QG QueryGraph) []uint {
    n := uint(len(QG.R))
    subgraphs := []uint{}
    for i := n - 1; i > 0; i-- {
        v := uint(1 << i)
        subgraphs = append(subgraphs, v)
        B := uint(1 << (i+1) - 1)
        recursiveSubgraphs := EnumerateCsgRec(QG, v, B)
        subgraphs = append(subgraphs, recursiveSubgraphs...)
    }
    return subgraphs
}
```

**Algorithm 12:** Go implementation of `EnumerateCsg`

Now, we need to implement the `EnumerateCsgRec` subroutine used inside the implementation of `EnumerateCsg`. Similarly, we keep track of all emitted subgraphs by defining a `subgraphs` slice, to which all emitted subgraphs are appended. This approach also works for recursive calls to `EnumerateCsgRec`. In order to calculate the subsets  $S' \subset N$  we use the `PowerSet` function defined in [algorithm 8](#).

**Note 10.** Alternatively, we could also use a pointer to a `[]uint` slice containing all subgraphs that is passed along as a parameter to `EnumerateCsgRec` and thus lets us append to the original slice instance directly. However, we

choose this approach as it requires only function-scoped variables and thus makes the function's state more local and predictable, and thus arguably easier to understand and debug.

```

func EnumerateCsgRec(QG QueryGraph, S uint, X uint) []uint {
    n := uint(len(QG.R))
    Neighbors := Neighbors(QG, S)
    N := SetMinus(Neighbors, X, n)
    subgraphs := []uint{
    for _, SPrime := range PowerSet(N) {
        if SPrime == 0 {
            continue
        }
        SuSPrime := S | SPrime
        subgraphs = append(subgraphs, SuSPrime)
    }
    for _, SPrime := range PowerSet(N) {
        if SPrime == 0 {
            continue
        }
        SuSPrime := S | SPrime
        XuN := X | N
        recursiveSubgraphs := EnumerateCsgRec(QG, SuSPrime, XuN)
        subgraphs = append(subgraphs, recursiveSubgraphs...)
    }
    return subgraphs
}

```

**Algorithm 13:** Go implementation of EnumerateCsgRec

In EnumerateCmp, we find complementary subgraphs to each connected subgraph  $S_1$ . Here, we use the MinUintSetBitIndex function defined in [9] to calculate the index for the relation with the smallest index  $i_{min}$  inside the passed subset  $S_1$ , i.e.

$$i_{min} = \arg \min_i (R_i \in S_1) \quad (4.22)$$

As a result, we want to export all csg-cmp-pairs. In order to distinguish between the two sets we use the data structure [4], CsgCmpPair, defined in

section 4.5, which differentiates between sets  $S_1$  and  $S_2$ . In order to calculate all pairs, we first recursively calculate all complements for the subgraph  $S_1$  for each of its contained relations using `EnumerateCsgRec`. We then go through all of the calculated emits to pair them with  $S_1$  and append it to our `pairs` slice.

```
func EnumerateCmp(QG QueryGraph, S1 uint) []CsgCmpPair {
    minS1 := MinUintSetBitIndex(S1)
    BminS1 := uint(1<<minS1) - 1

    X := BminS1 | S1
    n := uint(len(QG.R))
    neighbors := Neighbors(QG, S1)
    N := SetMinus(neighbors, X, n)

    pairs := []CsgCmpPair{}
    setBits := IdxsOfSetBits(N)
    for i := len(setBits) - 1; i >= 0; i-- { // Descending
        v := setBits[i]
        pair := CsgCmpPair{Subgraph1: S1, Subgraph2: 1 << v}
        pairs = append(pairs, pair)

        Bi := uint(1<<v - 1)
        recursiveComplements := EnumerateCsgRec(QG, 1<<v, X|(Bi&N))
        for _, S2 := range recursiveComplements {
            pair := CsgCmpPair{Subgraph1: S1, Subgraph2: S2}
            pairs = append(pairs, pair)
        }
    }
    return pairs
}
```

**Algorithm 14:** Go implementation of `EnumerateCmp`

Finally, we can specify **DPccp**. First, we allocate a slice to a **\*Tree** pointer, called **bestTree**, where we store the optimal operator tree for the corresponding subgraph  $S$ . As this slice can potentially store the best entries for all combinations of subgraphs in the query graph, this slice reserves memory for  $2^n$  **\*Tree** pointers. Thus, on a 64-bit machine, the corresponding **make** allocates  $64 * 2^n$  bits of memory.

**Example 5.** For the maximum 10 relations, this would be

$$\frac{64 \text{ bit} * 2^n \text{ bit}}{1 \text{ B}} = 8192 \text{ B} = 8 \text{ KiB} \quad (4.23)$$

of memory required to store this slice of pointers.

Since the resulting operator trees for single relations are trivial we set each entry for indices  $2^i$ , representing relations  $R_i \in R$ , with a **&Tree** containing only itself.

Further, we make use of the enumeration functions specified above to create all **csg-cmp-pairs** for all subgraphs generated by **EnumerateCsg** and **EnumerateCmp**. We store the result in the **csgCmpPairs** variable.

Now, we can loop over all **csg-cmp-pairs** with subsets  $S_1$  and  $S_1$  and calculate the best operator tree for the union  $S_1 \cup S_2$ . This optimal tree is generated by taking the currently previously calculated optimal trees for  $S_1$  and  $S_2$ , creating their join trees for  $S_1 \bowtie S_2$  and  $S_2 \bowtie S_1$  respectively, and re-setting the entry in the **bestTree** slice in case the cost of this join tree is lower than the corresponding current entry. Likewise, if the entry does not exist in **bestTree** yet, it is set to the operator tree for  $S_1 \bowtie S_2$  or  $S_2 \bowtie S_1$ , whichever incurs a smaller cost according to the specified cost functions.

Finally, we return **bestTree**'s  $2^n - 1$ th entry, representing all relations  $\{R_0, \dots, R_{n-1}\} = R$ .



```

func DPccp(QG QueryGraph, JTC JoinTreeCreator) *Tree {
    n := uint(len(QG.R))
    bestTree := make([]*Tree, 1<<n)
    for i := uint(0); i < n; i++ {
        card := float64(QG.R[i])
        tree := &Tree{card, 1 << i, nil, nil, 0, nil}
        bestTree[1<<i] = tree
    }
    subgraphs := EnumerateCsg(QG)
    csgCmpPairs := []CsgCmpPair{}
    for _, subgraph := range subgraphs {
        subgraphCsgCmpPairs := EnumerateCmp(QG, subgraph)
        csgCmpPairs = append(csgCmpPairs, subgraphCsgCmpPairs...)
    }
    for _, csgCmpPair := range csgCmpPairs {
        S1 := csgCmpPair.Subgraph1
        S2 := csgCmpPair.Subgraph2
        S := S1 | S2

        p1 := bestTree[S1]
        p2 := bestTree[S2]

        currentTree := JTC.CreateJoinTree(p1, p2, QG)
        if bestTree[S] == nil {
            bestTree[S] = currentTree
        } else if bestTree[S].Cost > currentTree.Cost {
            bestTree[S] = currentTree
        }
        currentTree = JTC.CreateJoinTree(p2, p1, QG)
        if bestTree[S] == nil {
            bestTree[S] = currentTree
        } else if bestTree[S].Cost > currentTree.Cost {
            bestTree[S] = currentTree
        }
    }
    return bestTree[(1<<n)-1]
}

```

**Algorithm 15:** Go implementation of DPccp

#### 4.6.4 Building

The server code is written in Go 1.15.1<sup>29</sup> released on September 1st 2020, and using the `go1.15.1 darwin/amd64` compiler. We do not set any custom build flags.

**Linting** The project uses the Go language linter<sup>30</sup> to ensure a common coding and documentation style with what is used at Google, the creators of Go. All of the supplied files (`infra.go` and `joinproblemparser.go`) have been modified to satisfy all linter requirements by doing some miscellaneous changes to remove warnings generated from this static analysis, e.g. by converting variable and function names from `snake_case` to `camelCase`, or adding descriptive comments to exported functions. In addition, all newly created files adhere to the linter guidelines. As these changes are mostly trivial we omit further details here.

### 4.7 JSON Representation

The JSON representation of join problems and visualizations can be understood as the interface definition between server and client. “JSON” refers to the *JavaScript Object Notation Data Interchange Format* as specified in RFC 8259<sup>31</sup>. It is the output that can be interpreted language-independently and transferred between both entities. Since JSON files do not explicitly define semantics of their content, the server and the client have to agree on the data’s interpretation.

As of today (October 31, 2020), there is no official JSON schema definition file format yet. Therefore, we use the most recent (“2019-09”) draft as specified by the Internet Engineering Task Force (IETF)<sup>32</sup>, released on September 19, 2019<sup>33</sup>. Its specification and further information can be found on its website <http://json-schema.org> (last accessed: October 27, 2020).

#### 4.7.1 Join Problem

The schema definition for join problems can be found in data structure [13](#). All the keys correspond to the explicitly defined JSON tags for the

---

<sup>29</sup><https://github.com/golang/go/releases/tag/go1.15.1> (last accessed: October 27, 2020)

<sup>30</sup><https://github.com/golang/lint> (last accessed: October 27, 2020)

<sup>31</sup><https://tools.ietf.org/html/rfc8259> (last accessed: October 27, 2020)

<sup>32</sup><https://www.ietf.org> (last accessed: October 27, 2020)

<sup>33</sup><https://json-schema.org/draft/2019-09/json-schema-core.html> (last accessed: October 27, 2020)

JSONJoinProblem type in the joinProblemParser.go file.

#### 4.7.2 Visualization Routines

Likewise, there is a fixed JSON definition for an algorithm's visualization routines found in data structure [14](#).

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "title": "JoinProblem",
  "type": "object",
  "required": [
    "problemID",
    "neighbors",
    "numberOfRelations",
    "relations",
    "selectivities"
  ],
  "properties": {
    "problemID": {
      "type": "number"
    },
    "neighbors": {
      "type": "object",
      "patternProperties": {
        "^[0-9]+$": {
          "type": "string"
        }
      }
    },
    "numberOfRelations": {
      "type": "number"
    },
    "relations": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "cardinality": {
            "type": "number"
          }
        }
      }
    }
  }
}
```

```

        "name": {
            "type": "string"
        },
        "problemID": {
            "type": "number"
        },
        "relationID": {
            "type": "number"
        }
    },
    "additionalProperties": false
},
"selectivities": {
    "type": "object",
    "patternProperties": {
        "^[0-9,]+$": {
            "type": "number"
        }
    }
},
"additionalProperties": false
},
"additionalProperties": false
}

```

Data Structure 13: JSON schema of a join problem

```

{
    "$schema": "https://json-schema.org/draft/2019-09/schema",
    "title": "Visualization",
    "type": "object",
    "required": ["queryGraph", "routines"],
    "properties": {
        "queryGraph": {
            "type": "object",
            "required": ["relationCardinalities", "selectivities", "neighbors"],
            "additionalProperties": false,
            "properties": {
                "relationCardinalities": {
                    "type": "array",

```

```

        "items": {
            "type": "number"
        }
    },
    "selectivities": {
        "type": "object",
        "patternProperties": {
            "^[0-9]+$": {
                "type": "number"
            }
        },
        "additionalProperties": false
    },
    "neighbors": {
        "type": "object",
        "patternProperties": {
            "^[0-9]+$": {
                "type": "array",
                "items": {
                    "type": "number"
                }
            }
        },
        "additionalProperties": false
    }
},
"routines": {
    "type": "array",
    "items": {
        "$ref": "#/definitions/subroutine"
    }
},
"definitions": {
    "step": {
        "type": "object",
        "required": ["graphState", "variables", "uuid"],
        "additionalProperties": false,
        "properties": {
            "graphState": {

```

```

        "type": "object"
    },
    "variables": {
        "type": "object"
    },
    "uuid": {
        "type": "string"
    }
},
"subroutine": {
    "type": "object",
    "required": ["name", "observedRelations", "steps"],
    "additionalProperties": false,
    "properties": {
        "name": {
            "type": "string"
        },
        "observedRelations": {
            "type": "array",
            "items": {
                "type": "object",
                "required": ["identifier", "color"],
                "additionalProperties": false,
                "properties": {
                    "identifier": {
                        "type": "string"
                    },
                    "color": {
                        "type": "object",
                        "required": ["R", "G", "B", "A"],
                        "additionalProperties": false,
                        "properties": {
                            "R": {
                                "type": "number"
                            },
                            "G": {
                                "type": "number"
                            },
                            "B": {
                                "type": "number"
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

    },
    "A": {
      "type": "number"
    }
  }
}
},
"steps": {
  "type": ["array", "null"],
  "items": {
    "anyOf": [
      {
        "$ref": "#/definitions/subroutine"
      },
      {
        "$ref": "#/definitions/step"
      }
    ]
  }
}
}
}

```

### Data Structure 14: JSON schema of a visualization

## 4.8 Client Implementation

The client can interpret the JSON files as specified in section 4.7 which are generated by the server. It initiates and manages all communication with the very same and lets the user specify all possible parameters of the visualization. Furthermore, it provides a bitmap GUI to graphically render all steps.

For the client implementation we use the two JavaScript libraries `React.js`<sup>34</sup> and `Redux`<sup>35</sup> in order to be able to declaratively create a performant and maintainable client codebase. A rudimentary introduction is provided in sections 2.3.1 through 2.3.3, and more extensive guides can be found on the libraries' respective websites.

### 4.8.1 Project Structure

In order to bootstrap the project, we use Facebook's `create-react-app`<sup>36</sup> command-line tool, which creates an initial project structure and installs transitive dependencies. From this starting point we create the following directory structure:

---

<sup>34</sup><https://reactjs.org> (last accessed: October 27, 2020)

<sup>35</sup><https://redux.js.org> (last accessed: October 27, 2020)

<sup>36</sup><https://github.com/facebook/create-react-app> (last accessed: October 27, 2020)



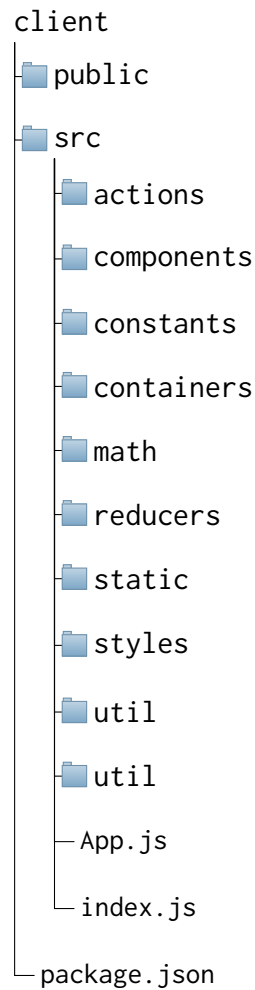


Figure 8: Client directory structure

Supplementary files, such as dot-files configuring the `git` repository are not specified here. Likewise, since listing all files in the directory tree would be too lengthy, we don't explicitly list the sub-files in each directory, but briefly describe each folder's and top-level files' purpose and contents in the following.

- **public** contains the website's `index.html` loaded from the server upon accessing the application's URL and some files specifying the site's metadata.
- **src** is the directory containing all source code making up our application.

- **actions** includes all functions that can manipulate the application’s (Redux) state
  - **components** contains all presentational UI components making up the application’s visual interface
  - **constants** specifies all of the application’s constants
  - **containers** contains all stateful pendants to the files in **components** and is responsible for mapping the required variables from the Redux state to the **props** of the respective component
  - **math** contains all files doing geometric calculations for the query graph visualization
  - In **reducers** all reducers are specified, which specify how the application’s state changes in response to actions sent to the store
  - **static** contains all static resources, such as images
  - **styles** contains the application’s Cascading Style Sheets (CSS) files specifying its layout
  - **util** contains utility functions for different purposes
  - **App.js** is the entry point for specifying the JSX components
  - **index.js** is setting up the app by creating the Redux store, initializing persistence layers and calling the **React render** method to initialize the rendering of the virtual Document Object Model (DOM)
- The **package.json**<sup>37</sup> defines the project’s dependencies, shell scripts for building or deployment and other metadata like version number and linter configuration

**Dependencies** We are using a number of dependencies to make use of existing user interface or middleware components. Those are specified in the **package.json** file. Upon installing these dependencies all recursive dependencies will also be installed.

#### 4.8.2 Utility Functions

**Pairing Function** We use the *elegant pairing function* developed by Matthew Szudzik [9] for two  $k_1, k_2 \in \mathbb{Z}$  in order to create a unique key for repeated elements in a nested loop with depth 2, by combining the two loops’ indices.

---

<sup>37</sup><https://docs.npmjs.com/files/package.json> (last accessed: October 27, 2020)

$$\begin{aligned}
p: \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{N} \\
(k_1, k_2) &\mapsto \begin{cases} k_1 * k_1 + k_1 + k_2, & \text{if } k_1 \geq k_2 \\ k_2 * k_2 + k_1, & \text{otherwise} \end{cases} \quad (4.24)
\end{aligned}$$

This is used as a **key** property for dynamically created `React.js` list items inside two stacked `maps`. This serves as a performance improvement, as per `React`'s official documentation, which states:

“Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity”<sup>38</sup>.

### 4.8.3 Redux Actions

In the following, we list all actions that can mutate the application's Redux state.

---

<sup>38</sup><https://reactjs.org/docs/lists-and-keys.html> (last accessed: October 27, 2020)

updateAlgorithms	Updates available algorithms upon server receiving the server response
changeOptionNumberOfRelations	Sets the number of relations sent to the server when (re-)calculating the visualization steps
changeOptionQueryGraphType	Sets the query graph type sent to the server when (re-)calculating the visualization steps
changeOptionAlgorithm	Sets the algorithm sent to the server when (re-)calculating the visualization steps
changeQueryGraph	Sets the query graph configuration (node coordinates, cardinalities, and selectivities) upon receiving the server response
updateGraphState	Sets the query graph state (color configuration) upon receiving the server response
updateRoutines	Sets the routines including all the steps to be used for setting the query graph state shown in the variable table
increaseStep	Increases the visualization step by a specified amount
decreaseStep	Decreases the visualization step by a specified amount
resetSteps	Resets the visualization step back to 1
updateSteps	Updates steps stored locally
updateCurrentStepUUID	Updates the UUID in order to highlight the current step

Table 1: Redux Actions

**Components** There are five main components making up the user interface. They are described in some detail in the following.

**JoinProblemSettings** JoinProblemSettings contains controls to change the join problem and algorithm settings. It also contains a button to recalculate the specified algorithm with updated settings.

**AlgorithmCanvas** The `AlgorithmCanvas` component contains a HTML5 `<canvas>` element<sup>39</sup> on which the query graph is drawn, and button controls to change the currently visualized step.

**VariableTable** `VariableTable` renders a table of all the routines' steps.

**VariableTableEntry** `VariableTableEntry` is a component used inside a `VariableTable` that displays the observed relation's variable keys in their respective color and the values taken on by these variables as a set of relations.

It can also recursively hold child `VariableTableEntry` components for stacked routines. In order to visualize hierarchy, it can darken its background color depending on the stack's size at the time of the execution of the corresponding subroutine.

Furthermore, if a function is using direct recursion, an arrow or multiple arrow symbols will be prepended to the routine name, depending on the depth of the recursion.

If the `Redux` state's current step UUID is equal to a step's UUID the table entry is highlighted in a different background color.

**AlgorithmVisualization** `AlgorithmVisualization` is a layout container consisting of a `JoinProblemSettings` header, and the `AlgorithmCanvas` and `VariableTable` components in a two-column layout below. An example rendered result is displayed in figure 9.

---

<sup>39</sup><https://html.spec.whatwg.org/multipage/canvas.html> (last accessed: October 27, 2020)

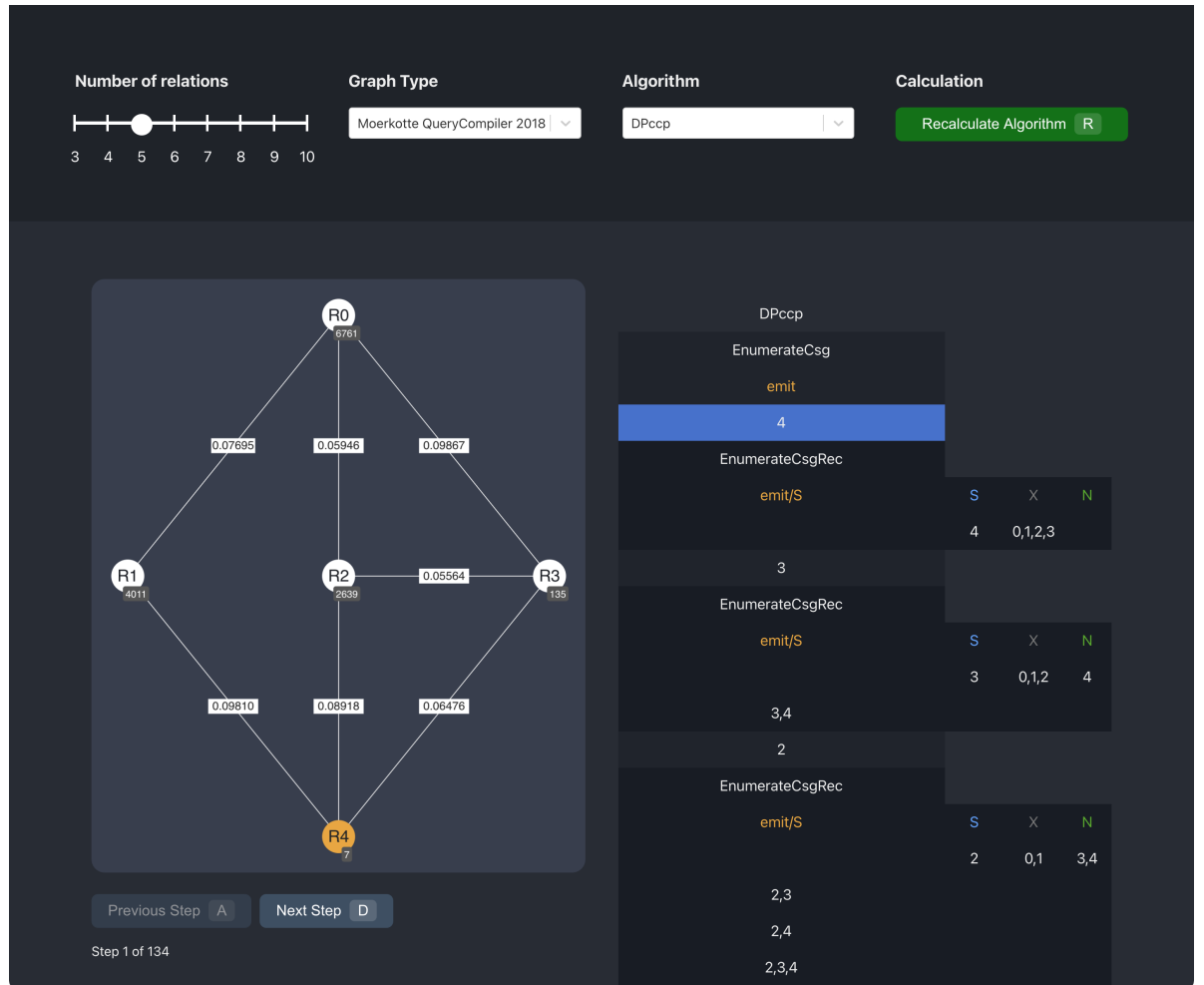


Figure 9: Sample rendered result of DPccp on a “Moerkotte 2018” query graph at step 1. The variable table on the right is scrollable to reveal more steps.

#### 4.8.4 Building

As the project makes heavy use of the modern ECMAScript 2020<sup>40</sup> syntax, it is transpiled using the Babel<sup>41</sup> cross-compiler to a more downwards-compatible JavaScript version when building the project to improve browser support.

<sup>40</sup><https://www.ecma-international.org/publications/standards/Ecma-262.htm> (last accessed: October 27, 2020)

<sup>41</sup><https://babeljs.io> (last accessed: October 27, 2020)

**Linting** For the client-side code we use ESLint<sup>42</sup> configured with the "react-app" template<sup>43</sup>, which itself is an extension of the "eslint:recommended" template as specified on <https://eslint.org/docs/rules/> (last accessed: October 27, 2020).

## 4.9 Installation and Development

### 4.9.1 Prerequisites

In order to run the project locally some prerequisites have to be fulfilled:

- A state-of-the-art web browser has to be installed, such as a recent version of Google Chrome<sup>44</sup>, Mozilla Firefox<sup>45</sup>, Microsoft Edge<sup>46</sup>, or Apple Safari<sup>47</sup>
- The Go programming language has to be installed according to the official installation guidelines<sup>48</sup>
- The Go environment has to be set up, namely correctly specifying the \$GOROOT and \$GOPATH environment variables. Furthermore, the project has to be moved to the \$GOPATH/src directory
- The Node Package Manager (npm) has to be installed along with the Node.js runtime<sup>49</sup>
- While not required, it is strongly recommended to install the Git<sup>50</sup> distributed version control system

### 4.9.2 Dependencies

**Client** Client dependencies are maintained in the `package.json` file of the project's `client` directory. All client repositories have to be installed by navigating to the `client` directory and then using the following command:

```
$ npm install
```

---

<sup>42</sup><https://eslint.org> (last accessed: October 27, 2020)  
<sup>43</sup><https://github.com/facebook/create-react-app/blob/master/.eslintrc.json> (last accessed: October 27, 2020)  
<sup>44</sup><https://www.google.com/chrome/> (last accessed: October 27, 2020)  
<sup>45</sup><https://www.mozilla.org/firefox/> (last accessed: October 27, 2020)  
<sup>46</sup><https://www.microsoft.com/edge> (last accessed: October 27, 2020)  
<sup>47</sup><https://www.apple.com/safari/> (last accessed: October 27, 2020)  
<sup>48</sup><https://golang.org/doc/install> (last accessed: October 27, 2020)  
<sup>49</sup><https://www.npmjs.com/get-npm> (last accessed: October 27, 2020)  
<sup>50</sup><https://git-scm.com> (last accessed: October 27, 2020)

**Server** The server-side code dependencies are defined in the `go.mod` file of the project's `server` directory. They can be installed by navigating to the `server` directory and then using the following command:

```
$ go get ./...
```

### 4.9.3 Local Development

Once dependencies are installed as specified in section [4.9.2](#), the project can be run locally by navigating to the project directory in a Unix shell (macOS and Linux) or PowerShell (Windows) and running

```
$ cd server && go run main.go
```

and, in another shell instance,

```
$ cd client && npm start
```

This will start two locally running web servers that can be accessed via <https://localhost:8080/> (Server) and <https://localhost:3000/> (Client).

**Note 11.** As the client is fetching its data from the server keep in mind that the server has to be started *before* accessing the client URL.

## 4.10 Deployment

One of the major benefits of using a client-server architecture and web technologies is that we can host the server part on an application server or platform-as-a-service offering such as Google's Cloud App Engine<sup>51</sup> and the front-end (client) on a web server. This way, the user of the visualization neither has to install nor update the application himself, as this duty is completely on the host's side. Further, the deployment of new application versions is simple and can be done with a single shell instruction for both client and server.

In the following we demonstrate how to set up the entire application stack, both client and server, in order to get from the cloned repository to a deployed application using the Google Cloud SDK<sup>52</sup> for the server-side application and GitHub Pages<sup>53</sup> for hosting the front-end.

---

<sup>51</sup><https://cloud.google.com/appengine> (last accessed: October 27, 2020)

<sup>52</sup><https://cloud.google.com/sdk> (last accessed: October 27, 2020)

<sup>53</sup><https://pages.github.com> (last accessed: October 27, 2020)



## Prerequisites

**Server** The Google Cloud SDK must be installed using

```
$ curl https://sdk.cloud.google.com | bash
```

Follow the instructions in the terminal session to finish the installation process. Further information is found on <https://cloud.google.com/sdk> (last accessed: October 27, 2020).

**Client** The user needs a GitHub<sup>54</sup> account and host the repository on their platform. GitHub Pages<sup>55</sup> has to be configured for the project according to the instructions on <https://pages.github.com> (last accessed: October 27, 2020).

## Deployment Commands

Now, we can simply deploy the client and server with just one line of shell code. In order to deploy the server one can simply type

```
$ gcloud app deploy
```

and, for the client:

```
$ npm deploy
```

---

<sup>54</sup><https://github.com> (last accessed: October 27, 2020)

<sup>55</sup><https://pages.github.com> (last accessed: October 27, 2020)

## 5 Conclusion

### 5.1 Summary

The toolset presented in this thesis can declaratively visualize algorithms such as `DPccp`, while hiding many details of the visualization itself—such as graph rendering and subroutine stack creation—from its user. Join problems can be fully graphically represented on the client’s browser window, including their cardinalities and selectivities, making it easy to reason about the steps the algorithm under observation will take, in case either or both of them are used in a cost function.

Most importantly, the toolset is designed with reusability and extensibility in mind, providing the opportunity to easily add new visualizations. Naturally, closely related algorithms such as `DPsize`[\[4\]](#), for which we also provide an implementation sample in the supplementary code, are part of this subset of already visualizable routines. To be precise, all algorithms for which we observe disjoint sets of relations can be visualized on a rendered query graph, without any modification to the visualization’s server or client logic, but by merely implementing the algorithm and calling the publicly exposed functions of the library to start subroutines or add new steps. It is the library’s users’ judgement however to evaluate the sensibleness of each application, as there might be a large amount of visualization steps or not all relevant information is conveyed in the portrayed graph or table of relations.

### 5.2 Future Work

There are still some open tasks that could be examined and implemented. As of now, the toolset can only be used on disjoint sets of relations. However, this disjointedness requirement can be eliminated by adding the possibility to render multi-colored nodes in the client’s query graph canvas. Further, as we implicitly interpret a bit vector as a set of relations, the type restriction on tracked variables could only be loosened by adding additional parameters to the variable observers to describe how to interpret them and format their respective outputs, thereby however posing a trade-off between declarativity and genericness.

In addition, the possible query graph types could be extended to cover other types such as non-complete trees, cliques or grids. Moreover, a possibility to user-define arbitrary trees could be provided. Even more generally, the library could be extended to display intermediate results of subroutines or providing the ability to track multiple tables of observed variables.

All of these proposed changes are possibly implemented by using the current state of the project and its architecture as a baseline.

## References

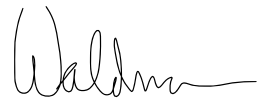
- [1] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, “A meta-study of algorithm visualization effectiveness,” *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.
- [2] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger *et al.*, “Exploring the role of visualization and engagement in computer science education,” in *Working group reports from ITiCSE on Innovation and technology in computer science education*, 2002, pp. 131–152.
- [3] F. Hohman, M. Conlen, J. Heer, and D. H. P. Chau, “Communicating with interactive articles,” *Distill*, vol. 5, no. 9, p. e28, 2020.
- [4] G. Moerkotte and T. Neumann, “Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products,” in *Proceedings of the 32nd international conference on Very large data bases*. Citeseer, 2006, pp. 930–941.
- [5] G. Moerkotte, “Building query compilers (under construction)[expected time to completion: 5 years],” 2009.
- [6] E. S. Raymond, “Compactness and orthogonality,” *The Art of Unix Programming*, 2003.
- [7] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.
- [8] R. J. Trudeau, *Introduction to graph theory*. Courier Corporation, 2013.
- [9] M. Szudzik, “An elegant pairing function.”

## Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, den 31. Oktober 2020

A handwritten signature in black ink, appearing to be 'Waldm' followed by a long horizontal stroke.

Unterschrift