

POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

**A Tidy Tree Interactive  
Visualization Library to Query  
Distributed Databases**



**Supervisor**

prof. Alessandro Fiori

**Candidate**

Emil DARI

December 2019

This work is subject to the Creative Commons Licence.

# Contents

<b>List of Figures</b>	v
<b>1 Introduction</b>	1
<b>2 Laboratory Assistant Suite</b>	3
2.1 LAS Query Module . . . . .	4
2.1.1 Current Interface . . . . .	5
2.1.2 Query Execution . . . . .	6
2.1.3 Considerations . . . . .	7
<b>3 Tidy Tree Algorithms</b>	9
3.1 Introduction . . . . .	9
3.1.1 Definitions . . . . .	10
3.2 Tidier Drawings of Trees . . . . .	12
3.2.1 WS algorithm . . . . .	12
3.2.2 RT algorithm . . . . .	17
3.3 Algorithm for General Trees . . . . .	24
3.3.1 W algorithm . . . . .	24
3.3.2 W Enhanced Algorithm . . . . .	27
3.4 Non-Layered Tidy Trees . . . . .	32
3.4.1 Complexity Proof . . . . .	34
3.4.2 Improving Layout Techniques . . . . .	37
3.5 Considerations . . . . .	39
<b>4 D3 Framework Overview</b>	41
4.1 Data-driven DOM Manipulation . . . . .	43
4.2 Considerations . . . . .	48
<b>5 GGen Library Implementation</b>	49
5.1 Requirements . . . . .	49
5.2 Methods and Techniques . . . . .	50

5.2.1	Module Loader . . . . .	50
5.2.2	Versioning . . . . .	52
5.2.3	SVG . . . . .	52
5.3	Implementation . . . . .	55
5.3.1	Configurability . . . . .	56
5.3.2	Init and Update . . . . .	56
5.3.3	Nodes and Arcs . . . . .	57
5.3.4	Customizable Functions . . . . .	58
5.3.5	Multiple-input Nodes . . . . .	59
5.3.6	Load and Store JSON . . . . .	61
5.3.7	Alternative Algorithms . . . . .	62
5.4	Considerations . . . . .	62
<b>6</b>	<b>GGEN - Use Cases</b>	<b>65</b>
6.1	Background . . . . .	65
6.2	Create and Submit a Query . . . . .	66
6.3	Save a Template . . . . .	72
6.4	Save a Translator . . . . .	73
6.5	Reload last-submitted Query . . . . .	74
<b>7</b>	<b>Conclusion</b>	<b>77</b>
7.1	Future Works . . . . .	78
<b>A</b>	<b>VDP Algorithm Implementation</b>	<b>81</b>
<b>B</b>	<b>Acronyms and Definitions</b>	<b>87</b>

# List of Figures

2.1	MDMM architecture. . . . .	4
2.2	Query Generator Interface. . . . .	5
3.1	Final positioning example of tree as drawn by WS Algorithm. . . . .	13
3.2	Mirror image of a tree positioned by WS Algorithm. . . . .	16
3.3	Example of Contours and Threads. . . . .	18
3.4	Example tree as drawn by RT Algorithm. . . . .	19
3.5	Example tree as drawn by Extended RT Algorithm. . . . .	26
3.6	Example tree as drawn by W Algorithm. . . . .	27
3.7	Example of $T_3$ tree. . . . .	29
3.8	Example of $T^3$ tree. . . . .	29
3.9	Example of updating ancestor pointers when adding new subtrees. . . . .	31
3.10	A layered and a non-layered representation of a tidy tree. . . . .	33
3.11	Example of Layered Tree. . . . .	34
3.12	Example of Contour Pairs in a non-layered tree. . . . .	36
3.13	Example of $T'_k$ tree. . . . .	38
3.14	Example of Sibling Lookup Linked List. . . . .	39
4.1	D3 Logo. . . . .	41
4.2	Mozilla Firefox Scenegraph Inspector. . . . .	42
4.3	D3 Data Joins. . . . .	46
4.4	Enter, Update and Exit subselections. . . . .	47
5.1	General Code Flow. . . . .	55
5.2	Block functions triggers. . . . .	59
5.3	Operator node example. . . . .	60
5.4	Exceptional 4-input node example. . . . .	61
5.5	Example of VDP algorithm drawings with curve edges and rectangular nodes . . . . .	62
5.6	Example of RT algorithm drawing with linear edges and circular nodes. . . . .	63
6.1	Query Module Starting Situation. . . . .	66
6.2	Select Entity/Operator Modal Menu (Biobank tab). . . . .	67
6.3	Select Entity/Operator Modal Menu (Operator tab). . . . .	67

6.4	Query Graph Example. . . . .	68
6.5	Entity Configuration Example. . . . .	69
6.6	Entity Configuration Example. . . . .	69
6.7	Query Graph Example. . . . .	70
6.8	Select Entity/Operator Modal Menu (Operator tab). . . . .	71
6.9	Query Graph Example. . . . .	72
6.10	Table of Result Example. . . . .	73
6.11	Table of Result Example (with Translator). . . . .	73
6.12	Template Definition Dialog. . . . .	74
6.13	Insert Title and Description Dialog. . . . .	75



# Chapter 1

## Introduction

Nowadays, Interactive Visualization has acquired growing importance, particularly in web-oriented environments. Modern web application struggles to provide as intuitive as possible interfaces, to guaranty fast and accessible data visualization.

In this context, a web platform for cancer genomic data management as the LAS (Laboratory Assistant Suite), which supports the research activities at Cancer Institute IRCCS Candiolo, has to deal with user interaction and Interactive Data Visualization.

Currently, this platform provides a query module allowing users to query different distributed databases from a single easy-to-use interface. The module, in fact, gives the possibility to build a graphical representation of the query, also to users without any knowledge of underlying databases' structures and technology (such as query language).

The current query builder allows users to manually interconnect nodes to create a query; this approach requires multiple consistency checks on the defined graph and may lead to a confuse, untidy representation. For this reasons, a new approach is needed, where the graph construction is guided by the application, allowing only correct predefined paths and defining a ordered tree giving a comprehensible, hierarchical visualization of the query.

The deprecation of some libraries used, in addition to the needs of a new visual and behavioural approach, has brought the necessity of restructuring the LAS query module.

Thus, the aim of this thesis is to renovate the query module interface, in order to provide a responsive tool, compliant to modern standards, that reduces the effort required to the users by implementing a supervised procedure for the query graph creation. To accomplish this goal, it has been developed a JavaScript library for drawing interactive tidy trees, that suites the LAS query module needs.



In the next Chapter (2) it will be briefly described the structure of the Laboratory Assistant Suite, focusing in particular on the LAS query module. In Chapter 3, it can be found the analysis of the current state of art in tidy-tree visualization algorithm in order to choose the one most suited by our library needs. In Chapter 4, instead, it will be presented *D3.js*, the JavaScript Framework on top of which we developed our library. Chapter 5 will be devoted to the high level description of the structure and the main methods concerning the implemented library. Finally, in Chapter 6 we will present some use-case scenarios to illustrates how the query module interface, interacting with the library, provides the desired results. The last Chapter (7) conveys some final considerations about the library and its possible future development.

# Chapter 2

## Laboratory Assistant Suite

The introduction of high-throughput technologies and automation techniques in laboratory environments has raised multiple issues related to the amount and heterogeneity of the data produced. This made necessary the adoption of computer-based systems able to assist researchers in their daily laboratory practice.

Laboratory Information Management Systems (LIMS) are commonly used for this purpose, allowing to ensure quality control among the laboratory activities, to efficiently handle large amount of data and to keep track of information related to biological experiments.

Although there are many commercial solutions providing a large set of features, often they tend not to fit highly specialized laboratory activities, such as the monitoring of xenopatients life cycle (i.e. biological models based on the transplantation of human tumors in mice), fundamental in cancer research.

For this reason, the Candiolo Cancer Institute in 2011 has invested in a LIMS project named LAS [2; 7; 8; 9] (Laboratory Assistant Suite) based on web-oriented software technologies.

The LAS attempts at covering a wide range of different laboratory procedures and, thanks to its modular and general-purpose structure, it can be extended to support new functionalities with a limited effort.

Among all supported features, it provides some graphical tools to facilitate decision-making tasks and all kind of tasks involving the analyses on integrated data.

In particular, in order to provide a way to retrieve information from multiple databases exploiting different technologies, it was developed the Multi-Dimensional Data Manager (MDDM) [9]. This module allows researchers to execute complex queries without any knowledge of the technologies involved in the databases interrogation. By means of intuitive interfaces and the MDDM, the LAS platform has improved data insertion and retrieval tasks, giving a wider perspective on the data interconnections.

## 2.1 LAS Query Module

Two aspects are fundamental in order to support researchers to discover new knowledge related to tumors: first, it is necessary to track experimental procedures and record data related to biological entities; then, it is needed an easy and structured way for retrieving the information tracked by the platform.

In order to lighten the development and maintenance processes, each LAS module operates on a separate database instance, partially replicating some information.

Thus, for extracting useful knowledge, the Multi-Dimensional Data Manger module has to address the issues of retrieving and integrating heterogeneous information, merging data from the different LAS databases. Furthermore, it has to provide methods for giving access to this actionable knowledge.

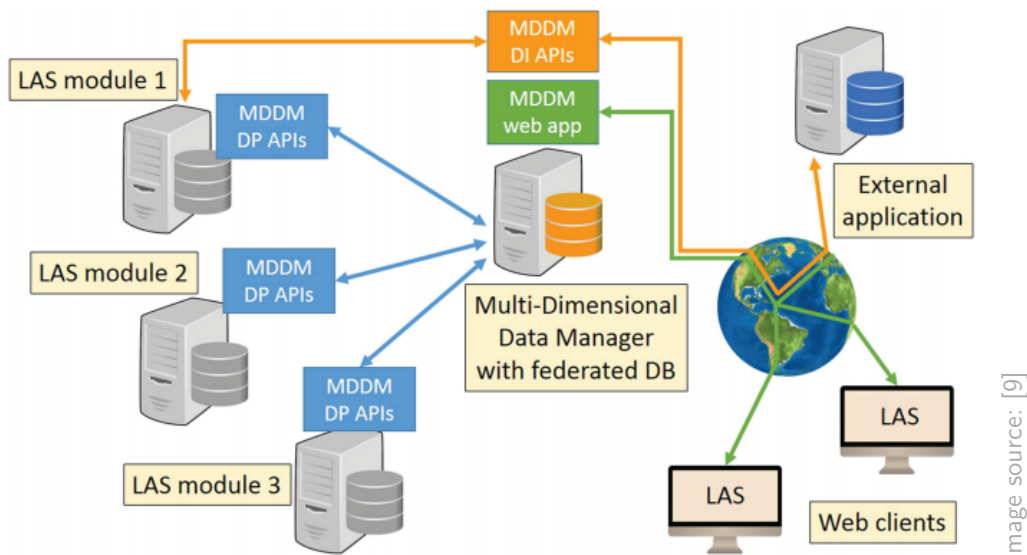


Figure 2.1. MDDM architecture.

Specifically, it provides a set of web APIs exploitable by other LAS modules, to retrieve data of interest. In addition, users are provided with a graphical interface for building customized queries, which gives them, in an intuitive fashion, a unified view on the entire collection of databases.

The main purposed addressed by our thesis, is the restructuring of this graphical tool, by means of the design and development of a interactive graph visualization library to enhance and update the current interface.

In Figure 2.1, it is possible to see how the MDDM has a distributed architecture. It is composed by a Data Integrator module (DI), providing a logical view

on databases belonging to other modules, and a Data Provider module (DP), providing an interface for collecting the schema information (e.g. database entities, entity attributes and relationships among entities) from each LAS database, for running SQL queries on the LAS modules' local databases and returning data to the DI module.

### 2.1.1 Current Interface

The MDDM query generator interface is the one of interest for the purpose of this thesis.

The tool exploits an intuitive graphical representation based on cascaded nodes, each representing one *entity* or an operator. In addition, it defines the kind of object returned as the nodes's output. *Entities* are defined as classes of objects or concepts that are relevant in the given context, similarly to entities in the Entity-Relationship database model. In this way, even an inexperienced user should be able to easily perform complex queries.

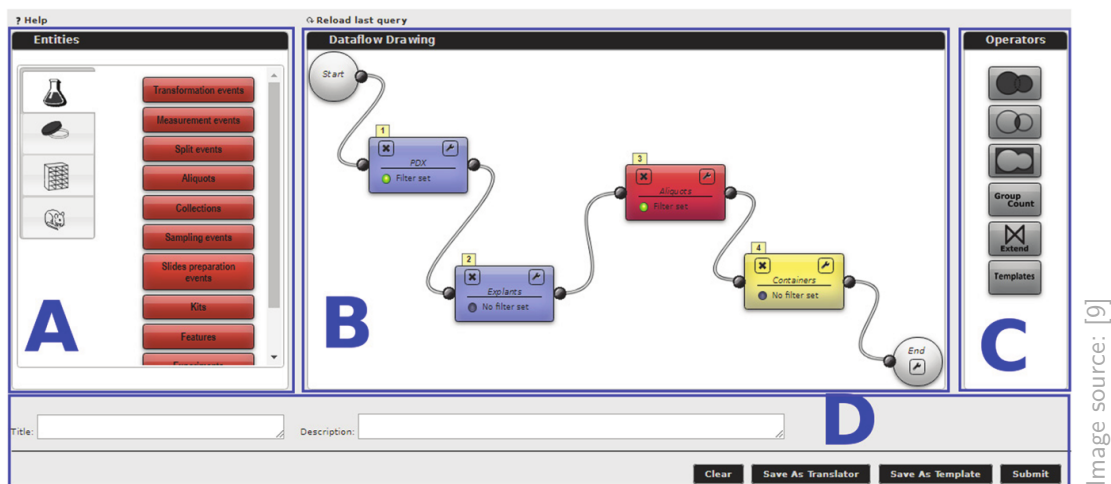


Figure 2.2. Query Generator Interface.

On the left section (identified by A, in Figure 2.2) of the editor, they can be found all available blocks, categorized accordingly to the Data Provider Module to which they belong (e.g. the mouse icon represents xenopatient data).

The blocks can be dragged and dropped into the workflow editor (identified by B, in Figure 2.2). In the example shown, that is taken from article [9], the blocks selected are PDX models (patient-derived xenograft, also known as xenopatients) and Explants, belonging to the Xenopatient DP, Aliquots from the Biobank DP

and Containers from the Storage DP. The goal of the query in the Figure is to retrieve all the containers (e.g. tubes) carrying aliquots explanted from PDX models.

For each block it is possible to specify some filtering conditions clicking the wrench-shaped icon. Instead, on the right section (identified by C, in figure 2.2) of the editor, they can be taken set operators (Union, Intersection and Difference) and special operators (Group-count, Extend and Template blocks).

To build a query workflow, blocks must be connected with arcs, that can be drawn between a block's output terminal and another block's input terminal, again by means of mouse press and drag.

The MDDM administrators are in charge of instructing the system on the possible ways to match entities. In order to allow the submission only of consistent and meaningful query, they defines *query paths*. This abstraction generalize the relational notion of a foreign key, stating how two different Entities A and B should be related to each other. In the graphical representation the interface has to check if a *query path* exists, before allowing an user to connect two blocks.

More specifically, *query paths* are defined as the set of DP tables that must be joined, through their foreign keys, in order to link A's base table to B's base table. Taken an instance of A, the maximum number of instances of B that correspond to it can be equal to one (if the foreign key chain includes only many-to-one or one-to-one relationships) or larger than one (if it includes at least one one-to-many relationship).

At this point, the user may give a title and a description to the query defined (section D, in Figure 2.2) and submit it; or, it may save it as a template, in order to run it later.

In addition, there is the option of saving the query as a Translator, which is a special type of template that may be optionally run for every row appearing in the result set of a query, to enrich it with additional, related information.

Eventually, the query generator interface allows an user to build any kind of query, not having any knowledge of the databases' structures, of the query language and on the low-level procedures that connects data among different databases.

### 2.1.2 Query Execution

An example of query completely defined is provided in figure 2.2, in section B; it is possible to notice how the query flow is visualized as a sequence of interconnected blocks, starting from a start block and ending to an end block.

On the other end, internally the query is represented as a completely different graph: a tree with the end terminal as root and the start terminals of each initial block as leaves. Hence, queries are executed performing a postorder traversal of the internal query tree.

This tree is then translated into a relational query structure. For each query block visited during the traversal, an access to the DI database is done to identify the underlying DP tables. At this point, the required DP tables are instantiated, the appropriate join conditions are set, the query block parameters are applied as filtering conditions and, eventually, aggregation operations (if present) are added.

At the same time, the *query path* connecting the current block to the next one is loaded, while any other required DP tables are instantiated. In the end, the query is sent to the DP by means of the APIs.

At this point, the DP checks if both the current block and its successor belongs to the same DP. If this is the case, the DP APIs, instead of issuing a real query to the underlying DBMS, will just create a logical view that wraps it. Differently, if the successor block resides in a different DP, it will issue a query, thus resulting rows are returned to the MDDM. When, during the traversal, is taken into consideration the next block, the results are sent to the DP together with the new query and inserted in an indexed temporary table that is joined with the rest of the query schema, in order to improve the performance of cross-DP queries.

Template queries, instead, are managed by storing the tree structure of the corresponding query in the MDDM database. Hence, whenever a Template is executed the tree can be reloaded and its parameters populated. Then, the tree is traversed as previously described.

### 2.1.3 Considerations

The query generator interface provided by the LAS query module is indeed easy-to-use, but it is possible to imagine ways to lighten even more the effort requested to the users.

In the current situation, an erratic connection between two nodes is notified by an alert, but other kind of errors does not generate any notification until the "submit", "save as Template" or "save as Translator" buttons are pressed (see D section, in Figure 2.2). The main consistency checks on the resulting graph are, in fact, invoked by these triggers. For example, to procedure which checks that all blocks in the workflow are connected to an input and an output or the one which checks that the graph is not cyclic.

In order to reduce the number of notifications that a inexperienced user may receive and to avoid the time loss due to a misconfiguration alerted only when the user is ready to submit or save the query, the graph construction should be made more constrained.

How is it possible to add more constraint on the user choices, reducing at the same time the alerts? A way, that allows also to build a nicer and clearer graphs, is provided in Chapter 6. The left and right section in figure 2.2 will be lost, in favour of a menu dynamically instantiated whenever a user wants to add a new block in

the workspace. In this way, the application guaranties a-priory the consistency of query representation with the underlying system. Basically, it automatically supervises the user action improving the ease of use and handiness of the entire interface.

# Chapter 3

## Tidy Tree Algorithms

### 3.1 Introduction

This chapter presents an overview of different published algorithms for drawing tidy rooted trees and compares their performance, in order to justify the choice made in implementing the *ggen* library.

Firstly, it is useful to explain what is intended with *drawing a tree*. Basically the process of drawing a tree consists in two stages: the first task is the assignment of positional,  $x$  and  $y$ , coordinates to each node of the tree; the second is the rendering of a graphical representation of nodes and edges, generating the actual drawing based on the previously defined coordinates. It is easy to notice that the main operation to be performed is the coordinates assignment.

The next sections present the state of the art works in determining the nodes' positions, while the algorithms focused on the graphical representation will be discussed in the next chapters.

At this point, it should be clear the meaning of *drawing a tree*, but is still to be defined what a *tidy tree* is. We know is a type of tree but, which does tidy means exactly? This question cannot be answered objectively: it is just possible to give a general definition and, then, rely on the slightly different definitions provided by each author.

Basically, they are considered *tidy tree* all drawings of a tree that are aesthetically pleasing and use minimum drawing space. As many definitions involving the human perception, it is not easy to give a universally accepted definition.

In order to provide a more rigorous description of *tree pleasantness*, in 1979, Wetherell and Shannon [14] introduced three aesthetic constraints, that may be called, for simplicity, Aesthetics.

In addition, they stated that a pleasant drawing should aim at maximizing the tree compactness, to achieve better readability. Thus, presenting their linear time



algorithm for drawing binary tidy trees, they provided the following rules to define them:

**Definition 3.1.1.** Aesthetic 1. Nodes belonging to the same tree level should lie along a straight line and the straight lines which defines the levels should be parallel. This is necessary to require that the relative order of nodes across any level be the same as in the level order traversal of the tree. This can be shown to guarantee that edges in the tree do not intersect except at nodes.

**Definition 3.1.2.** Aesthetic 2. A left son should be positioned to the left of its father and a right son to the right.

**Definition 3.1.3.** Aesthetic 3. A father should be centered over its sons.

In 1981, Reingold and Tilford [13] added a fourth constraint in order to overcome some deficiencies of the Wetherell and Shannon algorithm:

**Definition 3.1.4.** Aesthetic 4. A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree.

The Reingold and Tilford algorithm generates symmetrical drawing in linear time. The width of the drawings is not always minimized, but in general it remains sufficiently close to the minimum.

The Reingold and Tilford algorithm is then extended by Walker [10] in 1990 to draw rooted ordered tree of unbounded degree. The algorithm is no more limited in drawing binary trees but does not work in linear time. Thus, in 2002 Buchheim, Jünger and Leipert [5] present a linear improvement of Walker’s algorithm.

A limitation of all the aforementioned algorithms is that they are able to draw only layered trees and are designed for supporting nodes of the same size. This is adequate for many applications, but a more general solution is preferable. In a paper from 2014, Van Der Ploeg [11] enhanced the algorithm to allow variable-sized nodes, while keeping its linear time nature.

In the next sections the algorithms will be presented in details. For sake of brevity, each algorithm will be referred to as an acronym composed by the first letter in the names of its authors. For example the Wetherell and Shannon algorithm will be referred as WS algorithm, the Reingold and Tilford as RT etc...

### 3.1.1 Definitions

As a preliminary step some definitions are provided. The definitions follow the ones proposed by Buchheim’s paper [5].

**Definition 3.1.5.** A rooted tree is a direct acyclic graph with a single source, called root, such that there is a unique direct path from the root to any other node.

**Definition 3.1.6.** The level, or depth, of a node is the length of its path.

**Definition 3.1.7.** For each edge  $(v, w)$ ,  $v$  is called the parent of  $w$ , and  $w$  the child of  $v$ .

**Definition 3.1.8.** If  $w_1$  and  $w_2$  are two different children of  $v$ ,  $w_1$  and  $w_2$  are siblings.

**Definition 3.1.9.** Each node  $w$  on the path from the root to a node  $v$  is called an ancestor of  $v$ , while  $v$  is called a descendant of  $w$ .

**Definition 3.1.10.** A leaf of the tree is a sink of the graph: a node without children.

**Definition 3.1.11.** Each node  $v$  of a rooted tree  $T$  induces a unique subtree of  $T$  with root  $v$ .

**Definition 3.1.12.** In a binary tree each node has at most two children.

**Definition 3.1.13.** In an ordered tree, a certain order of the children of each node is fixed. The first child according to this order is called the leftmost child, the last is called the rightmost child.

**Definition 3.1.14.** The left sibling of a node  $v$  is its predecessor, the right sibling is its successor in the list of children of the parent of  $v$ .

**Definition 3.1.15.** The leftmost descendant of  $v$  on level  $l$  is the leftmost node on level  $l$  belonging to the subtree induced by  $v$ ; the rightmost descendant of  $v$  on level  $l$  is the rightmost node of that subtree.

**Definition 3.1.16.** Given that  $v_1$  is the left sibling of  $v_2$ ,  $w_1$  the rightmost descendant of  $v_1$  on some level  $l$ , and  $w_2$  is the leftmost descendant of  $v_2$  on the same level  $l$ , we can call  $w_1$  the left neighbor of  $w_2$  and  $w_2$  the right neighbor of  $w_1$ .

**Definition 3.1.17.** The reflection of a tree is the tree with reversed order of children for each parent node.

**Definition 3.1.18.** The left contour of a subtree is defined as the sequence of leftmost nodes in each level, traversed from the root to the deepest level. The right contour is the the sequence of rightmost nodes.

## 3.2 Tidier Drawings of Trees

Reingold and Tilford present, in the paper entitled "*Tidier Drawings of Trees*" [13], a definition of tidy binary trees and proposes an algorithm for drawing tidy trees compliant with that definition.

The authors based their article on the analysis and improvement of the Wetherell and Shannon algorithm [14]. The WS is able to draw tidy trees respecting the first three aesthetic rules, but Reingold and Tilford show how this algorithm may lead to aesthetically unpleasing or wider than necessary drawings.

### 3.2.1 WS algorithm

As a preliminary step, the WS algorithm requires that each node keeps the information relative to its level within the tree; the level of a node within the tree is defined as the number of hops needed to reach it, starting from the root. Moreover it is required to know the maximum depth of the tree, in other word, the number of its levels.

The complete algorithm, which is provided in the paper [14] written in an extended version of Pascal, can be found in a JavaScript version in the repository github [6], together with the other source code of this thesis. Here, in algorithm 1, we propose an high level description.

The algorithm is based on two different loops. The decision of implementing the algorithm in a iterative way instead of using a recursive one, was made by Wetherell and Shannon to make the algorithm translatabe also to languages not supporting recursion, but, as we will see, the most recent, and also much more readable, algorithms make use of recursion.

During the first loop, it is performed a postorder walk that assigns a preliminary  $x$  coordinate to each node. The preliminary  $x$  is given following this rule: a leaf node will take the next available  $x$  position considering its level; in case it has only a left son, it will be positioned one unit to the right of it; if it has only a right son, it will be positioned one unit to the left of it; while, if the node has two sons, it will take the average of their positions.

In addition, for each level is kept a *next\_pos* and a modifier value. The *next\_pos* keeps track of the next available position in that level. If the preliminary position is lower than the next available one on the considered level, the node takes the next available position, and in the modifier variable is recorded the shift that has to be applied to its subtrees, to move them accordingly.

During the second loop, instead, it is performed a preorder walk, in which to each node is given a final  $x$  coordinate, obtained by summing its preliminary  $x$  coordinate and the modifier of all the ancestors of the node. The modifier's are

cumulated, during this walk, in a *modifier\_sum* variable and applied to every node.

Unfortunately, the algorithm does not work well in every possible situation, as can be seen in Figure 3.1, where the drawing is not sufficiently pleasing, neither compact. The third and fourth node of the fifth level are too far apart.

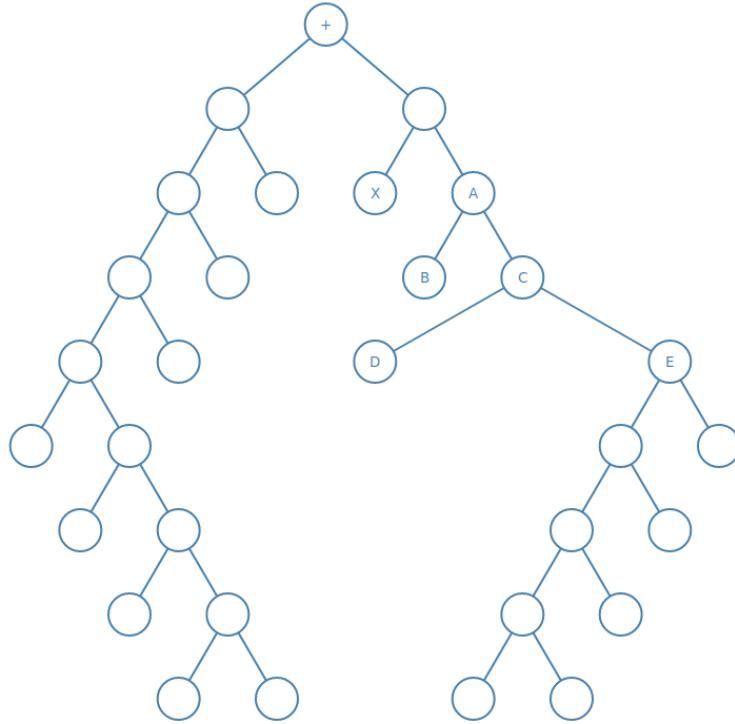


Figure 3.1. Final positioning example of tree as drawn by WS Algorithm.

To overcome this issues, Wetherell and Shannon presented a modified algorithm that guarantees minimum width drawings but that does not consider anymore Aesthetic 3 (Definition 3.1.3). Reingold and Tilford, on the other hand, proposed a new definition of tidy tree, obtained by adding a new aesthetic constraint, and an algorithm able to draw tidy ordered binary trees without getting rid of any Aesthetics.

They noticed that the main issues in the WS algorithm are caused by the influence that nodes outside a subtree may have on the shape of that subtree. As a consequence, trees that should be symmetric may be drawn asymmetrically; hence, a tree and its reflection will not always produce mirror image drawings; moreover, the same subtree may appear differently depending on its location inside the bigger tree. The tree in Figure 3.1, for example, should be symmetric while

**Algorithm 1:** Wetherell and Shannon Algorithm 3 - A tidy Tree Drawer

**Data:** Root node of the tree. Each node is assumed to have its height assigned.

**Result:** A tree positioned to satisfy Aesthetics 1, 2 and 3.

**Variables:**

modifier: array of Integer  $[0 - max\_height]$  initialized to 0

next\_pos: array of Integer  $[0 - max\_height]$  initialized to 1

root: root node of the tree

max\_height: number of levels, Integer

modifier\_sum: Integer initialized to 0

**begin**

```

    current = root
    current.status = FIRST_VISIT
                                                                    /* Postorder Walk */
while current != null do
    switch current.status do
        case FIRST_VISIT do
            if current has not a left son:
                current.status = LEFT_VISIT
            otherwise: current.status = FIRST_VISIT and
                current = left_son
        case LEFT_VISIT do
            if current has not a right son:
                current.status = RIGHT_VISIT
            otherwise: current.status = FIRST_VISIT and
                current = right_son
        case RIGHT_VISIT do
            if current has no children (leaf):
                place = next_pos[current.height]
            if has only right child: place = current.right_son.x - 1
            if has only left child: place = current.left_son.x + 1
            if has both:
                place = (current.left_son.x + current.right_son.x)/2
                modifier[h] = max(modifier[h], next_pos[h] - place)
                                                                    /* where h is current.height */
            if is leaf: current.x = place
            otherwise: current.x = place + modifier[current.height]
                next_pos[current.height] = current.x + 2
                current.modifier = modifier[current.height]
                current = current.parent

```

---

---

```
begin
    /* Preorder Walk */
    current = root
    current.status = FIRST_VISIT
    while current != null do
        switch current.status do
            case FIRST_VISIT do
                current.x = current.x + modifier_sum
                modifier_sum = modifier_sum + current.modifier
                current.y = 2 * current.height + 1
                if has left son: current = current.left_son and
                    current.status = FIRST_VISIT
                otherwise: current.status = LEFT_VISIT
            case LEFT_VISIT do
                if has a right son: current = current.right_son and
                    current.status = FIRST_VISIT
                otherwise: current.status = RIGHT_VISIT
            case RIGHT_VISIT do
                modifier_sum = modifier_sum - current.modifier
                current = current.parent
```

---

it is drawn asymmetrically; an even more clear example is provided in Figure 3.2, where it is easy to see how two specular trees drawn by WS algorithm are not mirror images.

Let's analyse Figure 3.1. The WS algorithm works in a left-to-right fashion, so, when the left branch is completely formed it considers the right branch and everything is normal until the fifth level. Here it comes the issue: to the leaf node D, is assigned the next available position, accordingly to *next\_pos* array (so  $Y.x = 6$ ); then it is considered the right branch, rooted at E (with  $E.x = 12$ ); so, it is assigned to the root of D and E the average of their positions ( $C.x = 9$ ) and the same to A, the root of B and C ( $A.x = 8$ ). At this point, A preliminary  $x$  is 8 but, next available position at its level is 10, because the position 8 is occupied by node X, so the A and its subtree is shifted to the right. The result is much wider than necessary, specifically the empty space in the middle of the tree causes D to be placed far to the left, when it should have been placed at the minimum distance from E.

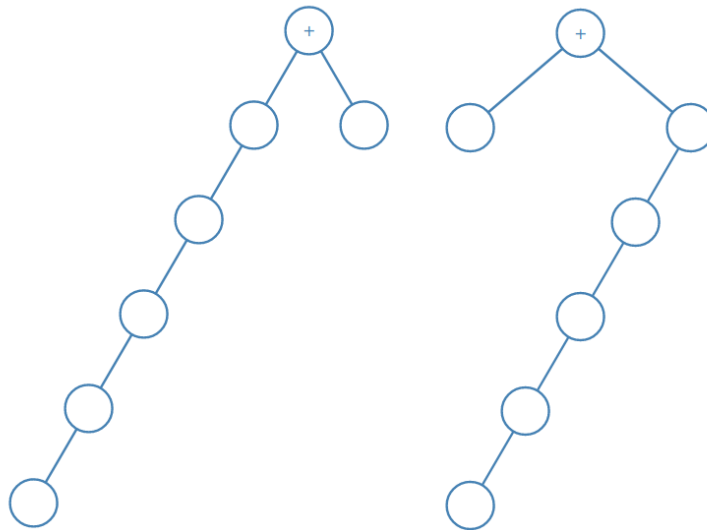


Figure 3.2. Mirror image of a tree positioned by WS Algorithm.

In order to prevent this issue a new constraint is defined: Aesthetic 4 (Definition 3.1.4). The introduction of this new Aesthetic has a cost in terms of tree width, but Reingold and Tilford consider the cost worth to be paid, because at the time none of the published tree printing algorithm produces a minimum width placements and, in their opinion, the newly introduced Aesthetic is more important than reaching a minimum width, for the pleasantness of the drawing.

Reingold and Tilford says that:

"satisfying Aesthetic 4 requires an algorithm in which nodes outside a subtree do not interfere with the relative positioning of nodes in the subtree, so that the inherent asymmetry of the postorder traversal will not be manifested in the drawing."

In order to achieve that, they proposed algorithm TR, based on the assumption that two subtrees of a node should be formed independently, and then placed as close together as possible, thus requiring that the subtrees be rigid at the time they are put together. Basically, in this way the algorithm positions subtrees (as static groups of nodes) rather than nodes themselves.

### 3.2.2 RT algorithm

As said, the RT algorithm aims at satisfying all the four Aesthetics. For achieving this objective it needs to be able somehow to form the left and right subtree of each node independently, in a way that nodes outside a subtree does not influence the relative positioning of nodes inside it.

The idea is basically to imagine that, initially, the two subtrees we want to rigidly move apart are superimpose: exactly as if they are drawn on two different pieces of paper put one over the other. In this trivial example the action of moving them apart is easy: it is sufficient, in fact, to take the piece of paper above and move it to the right until it does not cover up the other.

Coming back to the algorithm, in order to place as close together as possible two rigid subtrees, the algorithm should compare, at each level, the positions of the node inside the right contour of the left subtree with the position of node belonging to the left contour of the right subtree. To see what is intended with contour see the Definition 3.1.18, in the introduction of this chapter, and take a look at Figure 3.3: the node highlighted with diagonal stripes represent the right contour of the left subtree with respect to the root, while, the dotted nodes represent the left contour belonging to the right subtree.

The algorithm was implemented using, as starting point, the WS algorithm. In fact, it has a similar structure, although it uses recursive procedures instead of iterative ones.

The two procedures are the following:

- The first, called SETUP, is a recursive postorder traversal, which compare the two subtrees of a node (using the method of the contours) and moves them apart (algorithm 5).
- The second, called PETRIFY, is a preorder traversal which simply converts the relative preliminary positions into absolute coordinates (algorithm 2).



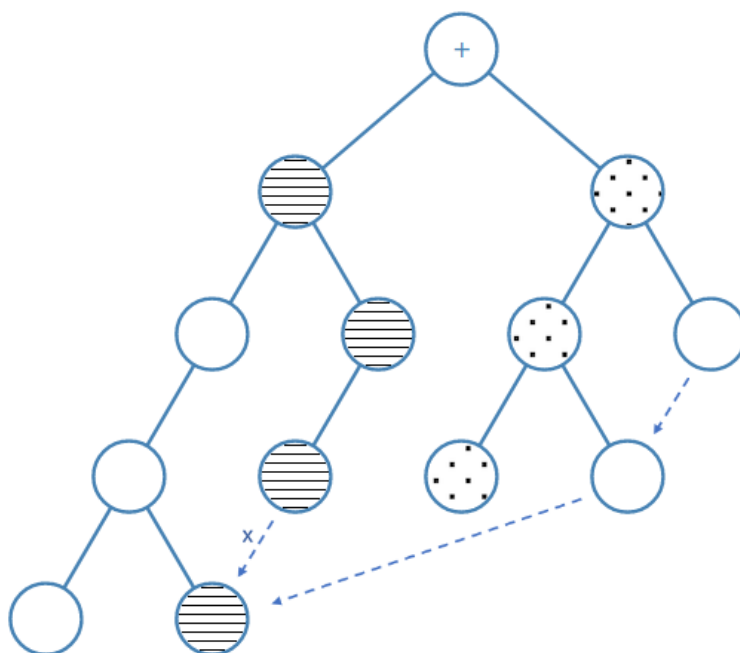


Figure 3.3. Example of Contours and Threads.

As can be seen in the pseudocode provided at the end of this section (algorithm 5), the SETUP procedure performs at each node three tasks.

Firstly, in the while loop, it determines how close together the two subtrees can be placed, by scanning down the right contour belonging to the left subtree and the left contour belonging to the right subtree. This is the most tricky part of the algorithm: comparing the contours can take a considerable amount of time, due to the traversal of almost all nodes of the tree. Luckily, Reingold and Tilford showed a clever technique to save up time, avoiding to traverse nodes not belonging to a contour. To accomplish it, without traversing also nodes not belonging to the contours, they adopted this rule: if the node is not a leaf, the next element of the left contour is its child placed the most to the left, and the next element of the right contour is its child placed the most to the right; if a node is a leaf, instead, the two variables dedicated to point to its children, that are empty, will be exploited, if needed, to keep the next element of the left and right contour. When they are not pointing actual children, but are used for storing next nodes in the contours, they call them threads. In order to state their difference from normal pointers, a boolean field is introduced in each node to explicitly track if they are to be considered threaded or not.

As second task, the algorithm keeps track of the leftmost and rightmost nodes

on the closest-to-the-root level of the subtree, because those are nodes that may need to be threaded later; specifically a new thread must be added, whenever two non-empty subtrees with different heights have to be combined.

Finally, the third task is indeed inserting a thread when needed. In Figure 3.3 it is shown an example: the three threads, represented by the dashed arrows, are created at different points in time, when the left subtree of a node is taller with respect to the right one: thus a thread must be inserted in the lowest node of the shorter subtree, pointing to the lowest node of the taller subtree. Specifically, the arrow labeled with a  $x$  allows to scan the entire striped contour.

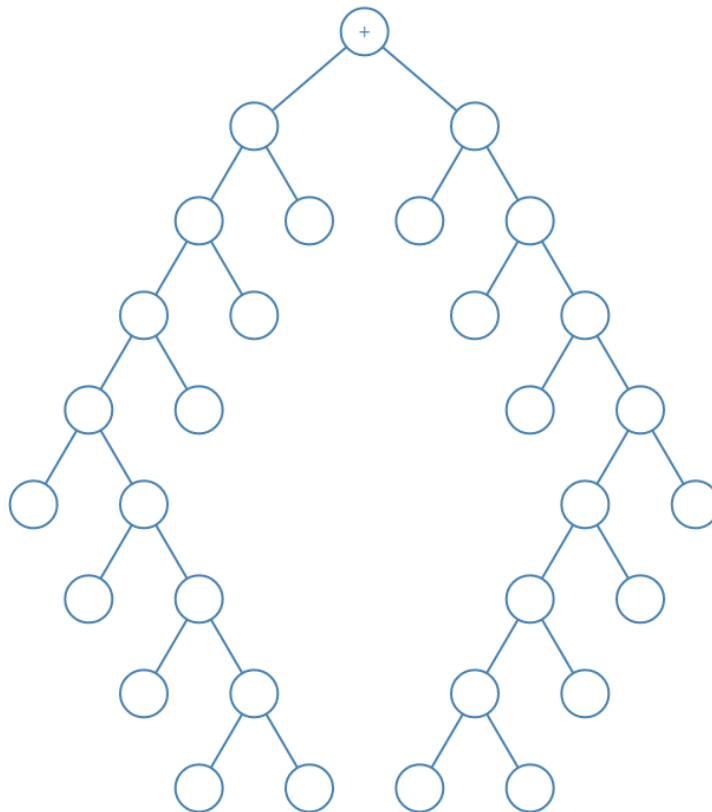


Figure 3.4. Example tree as drawn by RT Algorithm.

This idea on which RT is based allows to overcome the bad positioning of nodes produced by WS, for example node D and E in the situation depicted in Figure 3.1; it can be seen in Figure 3.4, how RT generates much nicer results: here the same tree appears symmetric and clear.

It is interesting to analyse more in details the time complexity of RT algorithm.

Firstly, it is possible to notice that the time required is completely determined by the while loop, because the SETUP procedure is executed precisely once per node of the tree. Moreover, it can be noticed that the while loop is executed only as long as both L and R are not-null, hence only until the depth of the shorter tree is reached. Another consideration important to underline is that the authors consider the height of a tree as the number of levels composing that tree, differently from Wetherell and Shannon. In their definition, e.g, the tree in figure 3.2 has height 6.

Keeping in mind those premises, to proceed demonstrating the linearity of the algorithm, we have to compute the number of times the loop body is executed:  $F(T)$  for a tree rooted at T.

The loop is executed a number of times equal to the sum of the number of executions done in each subtree, plus one iteration at each level in the shorter of its subtrees, hence:

$$F(T) = F(T_1) + F(T_2) + \min(\text{height}(T_1), \text{height}(T_2))$$

The authors demonstrate by induction on the number of nodes composing the tree, that the loop body is run  $F(T) = N(T) - \text{height}(T)$ ; where  $N(T)$  is the number of nodes belonging to the tree. It is easy to see how this claim is true in the degenerate cases in which  $N(T) = 0$  (empty tree) and  $N(T) = 1$  (the root is a leaf itself, thus the loop is not executed).

At this point, the inductive hypothesis is to suppose the claim is true for tree with less than N nodes. Thus, for  $k < N$ :

$$\begin{aligned} F(T) &= [k - \text{height}(T_1)] + [N - k - 1 - \text{height}(T_2)] + \min(\text{height}(T_1), \text{height}(T_2)) \\ &= N - 1 - \text{height}(T_1) - \text{height}(T_2) + \min(\text{height}(T_1), \text{height}(T_2)) \\ &= N - [\max(\text{height}(T_1), \text{height}(T_2)) + 1] \end{aligned}$$

The expression between square brackets is exactly  $\text{height}(T)$ , hence this demonstrates that the claim is valid for trees with any number of nodes N.

In the worst case, for a complete binary tree, the loop is executed about  $N(T) - \log(N(T))$  times: consequently the algorithm is linear.

---

**Algorithm 2:** Reingold and Tilford Algorithm - second walk

---

/\* PETRIFY - Preorder Walk \*/

**Data:**

T: current node

xpos: x coordinate

converts the relative offsets to absolute coordinates

**begin**  **if**  $T \neq \text{null}$  **then**     $T.x = xpos$     **if**  $T.data.thread$  **then**       $T.data.thread = \text{false}$        $T.children = []$      $PETRIFY(T.childrenleft, xpos - T.data.offset)$      $PETRIFY(T.childrenright, xpos + T.data.offset)$ 

---

**Algorithm 3:** Reingold and Tilford Algorithm - first walk

---

/\* SETUP - Postorder Walk - 1/3 \*/

**Data:**

T: root node of the tree

LEVEL: current level

RMOST, LMOST extremes nodes

An extreme node is an object containing 3 fields:

.addr: points to the corresponding extreme node

.off: the offset from the root of the subtree

.lev: tree level of the extreme node

MINSEP: parameter giving minimum separation between two nodes on the same level

**Result:** A tree positioned to satisfy Aesthetics 1, 2, 3 and 4.**Variables:**

L, R: left child and right child of node T

LR, LL: left rightmost and left leftmost node

RR, RL: right rightmost and right leftmost node

CURSEP: separation at current level

ROOTSEP: accumulated separation for the children of current node

LOFFSUM, ROFFSUM: accumulated offset of the current L and R

---



---

```
/* SETUP - Postorder Walk - 2/3 */
```

```
begin
  if  $T \neq \text{null}$  then
     $T.y = \text{level}$ ; Assign to L T's left child and to R the T's right child
    SETUP(L, level+1, lr, ll)
    SETUP(R, level+1, rr, rl)
    if  $L == \text{null}$  and  $R == \text{null}$  then
      We are in a leaf: this node is both rightmost and leftmost
       $RMOST.addr = LMOST.addr = T$ 
       $RMOST.lev = LMOST.lev = \text{level}$ 
       $RMOST.off = LMOST.off = 0$ ;  $T.data.offset = 0$ ;
    else
      T is not a leaf: set up for subtree pushing.
       $CURSEP = ROOTSEP = MINSEP$ 
       $LOFFSUM = ROFFSUM = 0$ 
      Now consider each level in turn until one subtree is exhausted,
      pushing the subtrees apart when necessary
      while  $L \neq \text{null}$  and  $R \neq \text{null}$  do
        if  $CURSEP < MINSEP$  then
           $ROOTSEP = ROOTSEP + (MINSEP - CURSEP)$ 
           $CURSEP = MINSEP$ 
        if L has a right child then
           $LOFFSUM = LOFFSUM + L.data.offset$ 
           $CURSEP = CURSEP - L.data.offset$ 
           $L = L.children[1]$ 
        else
           $LOFFSUM = LOFFSUM - L.data.offset$ 
           $CURSEP = CURSEP + L.data.offset$ 
           $L = L.children[0]$ 
        if R has a left child then
           $ROFFSUM = ROFFSUM - R.data.offset$ 
           $CURSEP = CURSEP - R.data.offset$ 
           $R = R.children[0]$ 
        else
           $ROFFSUM = ROFFSUM + R.data.offset$ 
           $CURSEP = CURSEP + R.data.offset$ 
           $R = R.children[1]$ 
      Continue next page...
```

---



---

```
/* SETUP - Postorder Walk - 3/3 */
```

```

...continue from previous page.
Update extreme descendant Information RMOST, LMOST
if rl.lev > ll.lev or T has no left child then
  | LMOST = rl
  | LMOST.off = LMOST.off + T.data.offset
else
  | LMOST = ll
  | LMOST.off = LMOST.off - T.data.offset
if lr.lev > rr.lev or T has no right child then
  | RMOST = lr
  | RMOST.off = RMOST.off - T.data.offset
else
  | RMOST = rr
  | RMOST.off = RMOST.off + T.data.offset
If subtree are of uneven height, check if threading is necessary
if L is not left child of T then
  | rr.addr.data.thread = true
  | rr.addr.data.offset =
    | Math.abs((rr.off - T.data.offset) - loffsum)
  | if loffsum - T.data.offset <= rr.off then
    | rr.addr.children[0] = L
  | else
    | rr.addr.children[1] = L
else if R is not right child of T then
  | ll.addr.data.thread = true
  | ll.addr.data.offset =
    | Math.abs((ll.off - T.data.offset) - roffsum)
  | if roffsum + T.data.offset >= ll.off then
    | ll.addr.children[1] = R
  | else
    | ll.addr.children[0] = R

```

---

### 3.3 Algorithm for General Trees

In a very popular paper[10] published in 1990, John Q. Walker II proposes an algorithm to determine the position of nodes for an arbitrary general tree; but what is intended as *general tree*? As described in the paper, a *general tree* is defined as a rooted (Definition 3.1.5), directed tree and of unbounded degree. The degree of a tree is just the number of children that each node can have.

Whereas no node may have more than one parent, in a general tree there is no limit on the number of offspring per node; Differently from binary and ternary trees, for example, which are trees with a limit of 2 and 3 children per node.

The aforementioned algorithms are only able to compute the nodes' positions for binary trees; for this reason, Walker's paper represent an important improvement in the search for a tree-drawing algorithm which aims at being as general as possible.

Unfortunately, the W algorithm can be proved to be quadratic. This is, in fact, what the second paper[5] authors focus on; Buchheim, Junger and Leipert not only point out the non-linearities of W algorithm, but also, present some improvements that guaranty to obtain the same results in linear time.

In the next sections will be described the W algorithm and its enhancement.

#### 3.3.1 W algorithm

While Reingold and Tilford with their work have expanded the Aesthetics proposed by Wetherell and Shannon, Walker reduces them collapsing Aesthetic 1 and 2 together (Definition 3.1.1 and 3.1.2). In particular, considering the fact that left son and right son distinction does not apply any more in a general tree, if a node has a single child, it should be placed directly below its parent. Thus, the Aesthetic 3 (Definition 3.1.3) is removed and Aesthetic 2 is replaced with:

**Definition 3.3.1.** Aesthetic 2. A parent should be centered over its children.

The RT algorithm itself can be easily extended for drawing general trees: all the children of a node can be traversed from left to right to assign the  $x$  coordinate and apply the shift to each corresponding subtrees, one after another. However, this RT extension breaks the symmetry rule stated by the fourth Aesthetic, as show in Figure 3.5.

The W algorithm, instead, respects all the remaining Aesthetics (1, 2 and 4, Definition 3.1.1, 3.3.1, 3.1.4), minimizing the width of the tree, with only two different walks, as the previously seen algorithms. It also has the advantage of being able to handle alternate orientations of the tree and variable node sizes. The main problem of this algorithm is that, differently from what it is claimed in

the author's article [10], the time complexity is higher than  $O(N)$ , where  $N$  is the number of nodes.

This algorithm is strongly based on ideas derived from the aforementioned positioning algorithms. In particular, the following three concepts:

1. Subtrees are built as rigid units. When a node is moved, all of its descendants, if it has any, are also moved accordingly.
2. A general tree is positioned by building it up recursively from its leaves toward its root. Also in this case the first walk is a postorder traversal.
3. Two fields are used for the positioning of each node. These two fields are: a preliminary  $x$  coordinate, and a modifier field (which in RT was called OFFSET).

A first postorder traversal assigns the preliminary  $x$  coordinate and the modifier value to each node. It works, as in RT, positioning the smallest subtrees first, starting from the leaves, and recursively proceeding from left to right, to define the position of subtrees, that are larger at any step. Adjacent nodes are kept separated one another by at least a predefined minimal distance, the one called MINSEP in RT, just called separation in W; at the same time, adjacent subtrees are separated by at least the same predefined separation.

However, in the first walk procedure, they can be highlighted some differences with respect to the RT first walk. Subtrees of a node are built up independently and they are placed as close as possible keeping in consideration a minimum separation value, exactly as in RT, but when it is to be moved a large subtree to the right, the distance it is moved is also distributed to smaller, interior subtrees. More specifically, the moving of these subtrees is accomplished, in a procedure called Apportion, by adding the proportional values to the preliminary  $x$  coordinate and modifier fields of the roots of the small interior subtrees.

The algorithm proceeds as follows.

Firstly, if the current node is a leaf there may happen two different situation: in case it has no left sibling its preliminary  $x$  is set to 0; in case it has a left sibling, instead, its preliminary  $x$  is set following this rule:

$$node.x = leftsibling.x + separation + mean(leftsibling.size, node.size)$$

Otherwise, if the current node is not a leaf, the first walk procedure is called recursively for each child, similarly to RT. Then, if the current node has no left sibling, its preliminary  $x$  is given by:

$$node.x = node.x - (leftmost.x + rightmost.x)/2$$



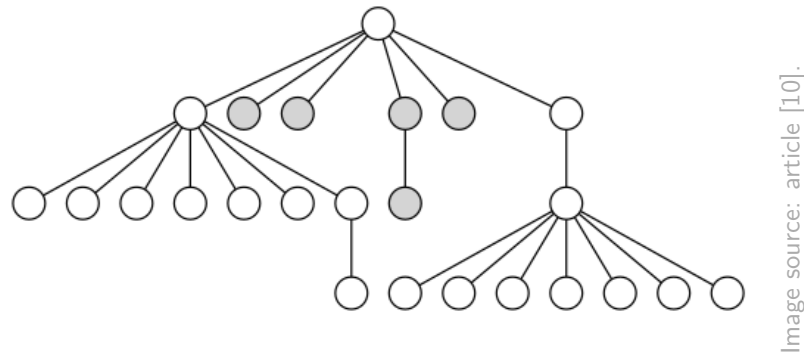


Image source: article [10].

Figure 3.5. Example tree as drawn by Extended RT Algorithm.

If it has left sibling its preliminary  $x$  position is set following the rule:

$$node.x = leftsibling.x + separation + mean(leftsibling.size, node.size)$$

And its modifier is set to:

$$node.modifier = node.x - (leftmost.x + rightmost.x)/2$$

Then, the APPORTION function is applied on the node. This function is the main idea behind Walker’s algorithm. He noticed, in fact, that pushing a new large subtree farther to the right, a gap may open between the new subtree and smaller subtrees that had been previously positioned correctly: after the push, in fact, they may tend to appear to be bunched on the left, leaving an empty area to their right. It is possible to see this issue in Figure 3.5.

What this function do is adjusting the positioning of small adjacent subtrees, in order to fix the issue stated above: when moving the new, large subtree to the right, the distance it is moved is also apportioned to smaller, interior subtrees, which means that the distance is distributed proportionally among them.

For example, if three small subtrees are bunched to the left because a new large subtree has been positioned to their right, the first small subtree will be shifted right by  $1/4$  of the gap, the second small subtree is shifted right by  $1/2$  of the gap, and the third small subtree is shifted right by  $3/4$  of the gap.

In the end, a preorder traversal is devoted to compute the final  $x$  coordinate of each node by summing the node’s preliminary horizontal coordinate with the modifier fields of all of its ancestors, starting from the root. In addition, it also adds a value that guarantees centering of the drawings with respect to the position of the root node.

The result obtained with this technique is a pleasing aesthetic placement of nodes, look at the example in Figure 3.6 with respect to Figure 3.5: the problem

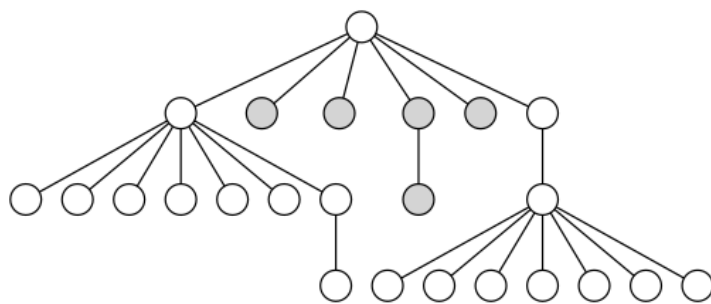


Image source: article [10].

Figure 3.6. Example tree as drawn by W Algorithm.

of small subtrees clustered on the left is solved and the internal subtrees are evenly spaced.

The pseudocode provided by Walker in its paper [10] is not included in this thesis, in the next section an improved version from the article [5] written by Buchheim, Junger and Leipert will be analysed. In their article, in fact, they provided an enhanced implementation of W algorithm that is linear and satisfy Aesthetics 1, 2 and 4 (Definitions 3.1.1, 3.3.1, 3.1.4).

### 3.3.2 W Enhanced Algorithm

As it has been said in the previous section, the W algorithm is a nice solution, which generates pleasant-to-see tidy trees, but has an inconvenience: it is  $\Omega(n^2)$ . The purpose of this section is to present the paper [5] by Buchheim, Junger and Leipert, which improves W algorithm by making it run in linear time.

First of all, three part of W algorithm present linearity issues:

- The function used to traverse the right contour (GETLEFTMOST).
- The function used to find the greatest uncommon ancestors.
- The function counting and shifting smaller subtrees (APPORTION).

GETLEFTMOST is a recursive function, used to find the leftmost descendant of a given node  $v$  at a given level  $l$ . It is a postorder traversal of the subtree rooted at node  $v$ . If the level of the current node (that corresponds to the leftmost descendant found) is equal to  $l$  that node is returned; otherwise the function is applied recursively to all children of the current node, from left to right.

To prove that the GETLEFTMOST run time is not linear in general, Buchheim, Junger and Leipert build a series of trees  $T_k$ , defined as follows:

"Beginning at the root, there is a chain of  $2k$  nodes, each of the last  $2k - 1$  being the right or only child of its predecessor. For  $i = 1, \dots, k$ , the  $i$ -th node in this chain has another child to the left; this child is the first node of a chain of  $2(k - i) + 1$  nodes."

In in Figure 3.7 it can be seen an example for  $k = 3$ .

The number of nodes in  $T_k$  is:

$$nodes(k) = 2k + \sum_{i=1}^k [2(k - i) + 1] = 2k + k(k - 1) + k$$

Thus the number of nodes  $nodes(k)$  is  $\Theta(k^2)$ .

At this point, for each  $i = 0, \dots, k - 1$ , when visiting the node on the right contour of  $T_k$  on level  $i$ , two subtrees have to be combined. By construction of  $T_k$  the highest level which is common to the subtrees is  $2k - i - 1$  and GETLEFTMOST is always to be applied to every node of the right subtree up to its level. The number of these nodes is:

$$k - i + \sum_{j=0}^{k-i-1} (2j) = (k - i) + (k - i)(k - i - 1) = (k - i)^2$$

Thus, for all combination, the total number of calls to GETLEFTMOST is given by:

$$calls(k) = \sum_{i=0}^{k-1} (k - i)^2 = \sum_{i=1}^k i^2 = k(k + 1)(2k + 1)/6$$

Thus,  $calls(k)$  is  $\Theta(k^3)$ .

Now, given that  $nodes(k)$  is  $\Theta(k^3)$ , so  $k$  is  $\Theta(n^{1/2})$ , the total runtime of GETLEFTMOST is proven to be non linear: GETLEFTMOST is executed  $calls(k)$  times, that is  $\Omega(k^3) = \Omega(n^{3/2})$ .

In regard to the second non linearity present in W algorithm, the authors highlighted as the function to find the uncommon ancestor is clearly quadratic. In fact, the greatest uncommon ancestor of the possibly conflicting neighbors are detected for each level by traversing the graph up to the current root; since the distance of the levels grows linearly, the total number of steps is  $\Omega(n^2)$ .

Finally, the last but not least non-linearity of W algorithm is related to the APPORTION function, which, when moving a large subtree to the right because in conflict with a subtree to the left, immediately, it shifts also all smaller subtree in between; moreover, to do that it has to compute the number of subtrees in between counting them one by one.

The authors show with an example that APPORTION total runtime is  $\Omega(n^{3/2})$ . Let the  $T^k$  tree (which can be seen in Figure 3.8) be constructed as follow:

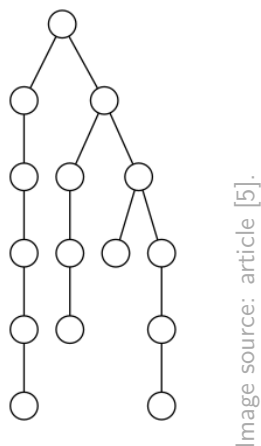


Figure 3.7. Example of  $T_3$  tree.

"Add  $k$  children to the root. The  $i$ -th child, counted  $i = 1, \dots, k$  from left to right, is root of a chain of  $i$  nodes. Between each pair of these children, add  $k$  children being leaves. The leftmost child of the root has  $2k + 5$  children, and up to level  $k - 1$ , every rightmost child of the  $2k + 5$  children has again  $2k + 5$  children."

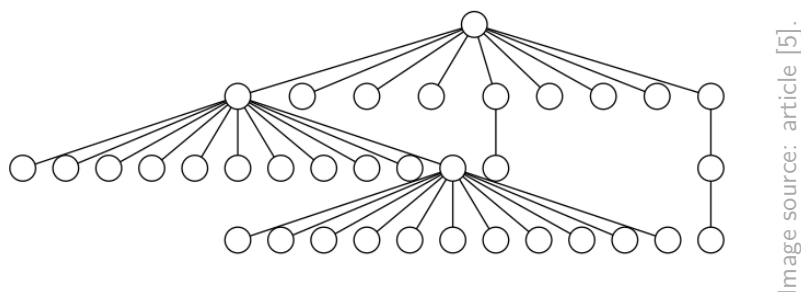


Figure 3.8. Example of  $T^3$  tree.

Hence, the number of nodes of  $T^k$  is:

$$nodes(k) = 1 + \sum_{i=1}^k i + (k - 1)k + (k - 1)(2k + 5)$$

Which means that  $nodes(k)$  is  $\Theta(k^2)$ .

Finally, by construction of the left subtree, adding the  $i$ -th subtree chain for  $i = 2, \dots, k$  results in a conflict on level  $i$ . Thus, all  $(i - 1)(k + 1) - 1$  smaller subtrees in between needs to be counted and shifted; the total number of counting and shifting is:

$$\text{count\_and\_shift}(k) = \sum_{i=2}^k [(i - 1)(k + 1) - 1] = (k + 1)k(k - 1)/2 - k + 1$$

As in the first non-linearity considered, we can derive that, since  $\text{nodes}(k)$  is  $\Theta(k^2)$ , so  $k$  is  $\Theta(n^{1/2})$ , and the number of times  $\text{count\_and\_shift}(k)$  is executed is  $\Theta(k^3)$ , therefore  $\text{count\_and\_shift}$  needs  $\Omega(n^{3/2})$  in total.

At this point, it is time to describe the enhancement apported to W algorithm.

Firstly, it can be noticed how the contour traversal (that in W is done by using the non-linear GETLEFTMOST function) can be performed in the same way as it is performed in RT algorithm, by using threads. The fact that subtrees are not binary trees does not create any additional difficulty.

The problem of finding the greatest uncommon ancestors, instead, is more tricky. First of all, it is useful to notice a few things.

The first thing to pay attention to is the moment in which we need to compute the ancestors: we need to compute it when we are in the situation in which we are placing a new subtree on the right and we need to know, for each node in the right contour of the subtree already placed, which is its greatest ancestor that is not ancestor of the new subtree.

Furthermore, it is possible to observe that, at that moment, the right uncommon ancestor is known: it is the root of the subtree being added.

Another thing to take into consideration is that the left greatest uncommon ancestor depends only on the nodes in the right contour of the already placed subtree, not on the nodes in the newly added subtree.

In order to keep track of the greatest uncommon ancestor, for each node it is kept a pointer called *ancestor* and, in a general pointer, it is kept one called *defaultAncestor*.

Now, it is better to explain the method with a practical example: suppose we are placing the subtrees rooted at  $r$ , and that we want to keep up to date the *ancestor* and *defaultAncestor* pointers; as in Figure 3.9, we call  $w_-$  the left greatest ancestor,  $w_+$  the right one; at the same time,  $v_-$  are called the nodes belonging to the right contour (circled in red) of the left subtree,  $v_+$  the nodes belonging to the left contour (in green) of the subtree we are placing.

In order to have the pointers always correctly up to date, we want that the following property holds:

"For all nodes  $v_-$  on the right contour of the left subforest after each subtree addition: If  $\text{ancestor}(v_-)$  is up to date, e.g., is a child of the

root  $r$ , then it points to the correct ancestor  $w_-$  of  $v_-$ ; otherwise the correct ancestor is *defaultAncestor*".

Initially, the first subtree, rooted at  $w$ , is placed; it does not need any ancestor computation, and *defaultAncestor* is set to  $w$ ; we can see how the property holds since all *ancestor*( $v_-$ ) points either to  $w$  or to higher level nodes. See again Figure 3.9, where the *ancestor* pointers are represented with a solid arrow if up to date, with a dashed arrow if expired: in this latter case the black node represent the one pointed by *defaultAncestor*.

After that, we place the subtree rooted at  $w'$ , another child of root  $r$ . When the new subtree is shorter than the left subtree, as in this case, depicted in the second drawing in Figure 3.9, we can update *ancestor* pointer of nodes belonging to its right contour setting them to point to  $w'$ . When the subtree added is taller, instead, to avoid runtime overhead, we update only *defaultAncestor*; it is the case of the third drawing, where *defaultAncestor* is set to point to  $w''$ . Again, since all pointer of the subtree induced by  $w''$  either points to  $w''$  or to a node of a higher level, the property holds.

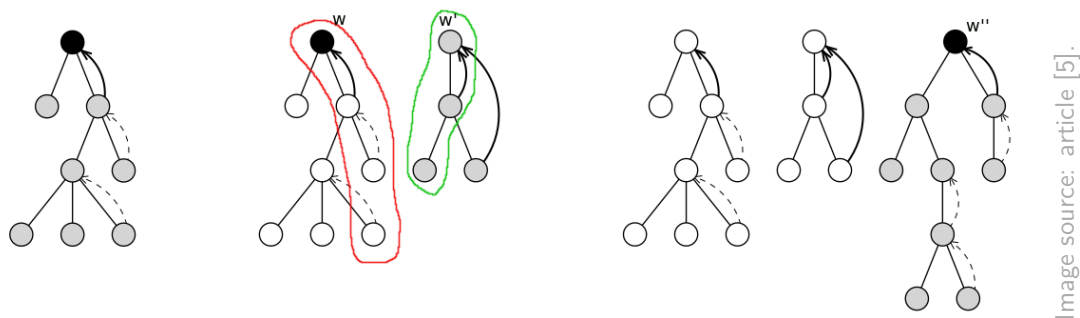


Image source: article [5].

Figure 3.9. Example of updating ancestor pointers when adding new subtrees.

In order to solve the last issue, Buchheim, Junger and Leiport address the problem in two ways.

The first way is related to computing the number of smaller subtree to be shifted. They linearize the process by performing a preprocessing step in which each child of a node is numbered consecutively; then, computing the number of smaller subtrees between two greatest uncommon ancestor  $w_-$  and  $w_+$  is reduced to a simple difference:  $number(w_+) - number(w_-) - 1$ .

The second way is related to the actual shifting of smaller subtrees: to obtain a linear runtime all subtrees, except the currently added, are shifted at most once: this is done in constant time by updating  $x$  preliminary coordinates and modifier of the root of the subtree.

To memorize the shifting in the moment they arise, in order to perform them in one only traversal, for each node we record a *shift* and *change* value, both initialized to zero. Now, let's see a practical example: suppose that the algorithm is currently placing the subtree rooted at  $w_+$  and that a conflict with the subtree rooted at  $w_-$  arises: the algorithm is forced to push the current subtree to the right by an amount of *shift*; assume that  $n\_subtrees$  is the number of subtrees among them.

According to Walker's idea, the  $i$ -th of these subtrees has to be moved by  $i * shift/n\_subtrees$ . In order to do that in one traversal, we increase  $w_+.shift$  by *shift*, decrease  $w_+.change$  by  $shift/n\_subtrees$  and increase  $w_-.change$  by  $shift/n\_subtrees$ . That can be interpreted as follows: nodes to the left of  $w_+$  are shifted by an amount initialized to *shift*, but this amount start decreasing by  $shift/n\_subtrees$  per subtree at node  $w_+$  and ends decreasing at  $w_-$ , where this division is zero. Finally, we can execute all the shift traversing the children, starting from the right and proceeding to the left: when visiting child  $v$ , we move it to the right by *shift* (increasing both its preliminary  $x$  and modifier values by *shift*), then we increase *change* by  $v.change$  and *shift* by  $v.shift$  and *change*; then we proceed going on the left sibling  $v$ .

The algorithm presented in this Section produces a proportional spacing distributed among subtrees, preserving the linear runtime. This algorithm is sufficiently general and can be used in many different application, it can be also improved to support node of different shapes. Although, in the next section will be described an algorithm even more general, which supports by design both nodes of different shapes and non-layered trees of unbounded degrees.

## 3.4 Non-Layered Tidy Trees

In 2014, A. J. Van Der Ploeg proposes an algorithm [11] that extends RT to make it work also for non-layered trees.

The main advantage of a non-layered tidy tree drawing algorithm is its flexibility.

Firstly, considering tree containing node of different sizes, it allows to generate more pleasant drawings: when node have varying height, in fact, layered drawings may use more vertical space than necessary; differently, non-layered drawings place children at a fixed distance from the parents, thus the result is indeed more vertically compact; see Figure 3.10 for a comparison.

Moreover, non-layered drawings can be used to draw trees where it is possible to decide a priori the vertical position of each node: this can be obtained for example adding dummy hidden nodes.

Then, both of the advantages stated above can be used in case we want to

show some information inside each node of the tree, or even within the dimensions and/or sizes of each node; for examples in software engineering class diagram or in formal languages parse trees.

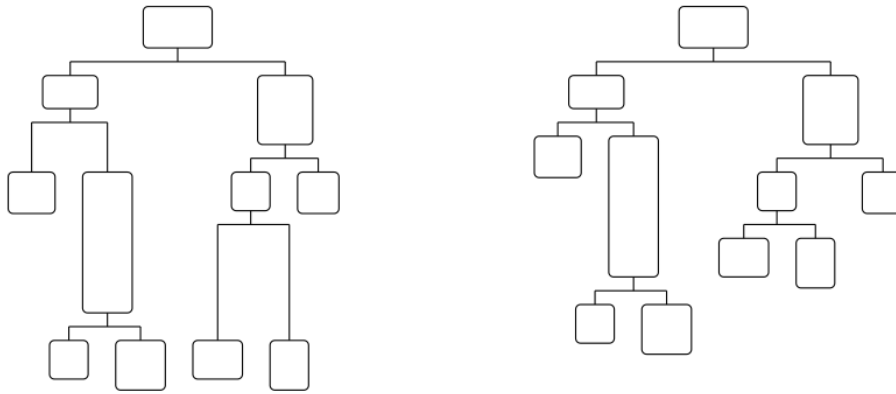


Image source: article [11].

Figure 3.10. A layered and a non-layered representation of a tidy tree.

With the aim of extending the techniques seen in the previously Sections, the author reformulate the entire tidy tree drawing problem, in order to include non-layered drawings. In the new formulation it is not considered anymore the spacing between nodes, the spacing is then included within the nodes themselves, by adding a gap to their widths and heights, a sort of invisible margin. This abstraction helps to simplify the process.

The author identifies two possible situations:

- A layered one, in which all the nodes at the same level can be considered to be at the same vertical coordinate. In this case the resulting tree is a rooted, ordered tree, whose nodes have a specific width, and whose levels have a specific height.
- A non-layered, more general, one. In this case the tree produced is a rooted, ordered tree with a width and height linked to each node. Actually, the vertical top position of a node is the bottom coordinate of its parent, which in turn is its top coordinate plus its height. Obviously, the case in which all the nodes at the same level are assigned with the same height, a non-layered drawing does not differ from a layered one.

The Aesthetics, to be adequate to this new kind of problem, can be reformulated as follows:

**Definition 3.4.1.** Nodes do not overlap.



**Definition 3.4.2.** Children are positioned horizontally in the order given in the tree.

**Definition 3.4.3.** Parents are centered above their children.

**Definition 3.4.4.** The drawing of a subtree does not depend on its position in the tree, which means that identical subtrees are drawn identically.

**Definition 3.4.5.** The drawing of the reflection of a tree, is the mirror image of the drawing of the original tree.

### 3.4.1 Complexity Proof

Since the VDP algorithm is based on the RT algorithm extended for the non-layered case, it is correct to introduce it by describing the proof of its linearity, provided by Van Der Ploeg [11]. He considered necessary to provide an alternative complexity proof, because the original one was based on an assumption that does not hold anymore. Specifically, the original proof depend on the total number of contour pairs considered to move all subtrees: which is equal to the number of times the while body is executed. We have seen in section 3.2.2 as Reingold and Tilford exploits techniques as *threads* and *extreme nodes*, to make the algorithm work in linear time. However, the assumption on which the analysis was done is that the number of nodes in the left and right contour is equal to the depth of the tree: the length of the longest path from root to leaves. In the non-layered case this is not always true.

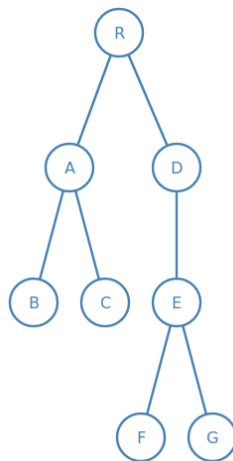


Figure 3.11. Example of Layered Tree.

Firstly, we consider the layered case. The new complexity proof is based on a different, kind of assumption: after that a node in the left contour of a subtree has been processed to push right that subtree, than it cannot be part of the left contour of another subtree. Consider Figure 3.11 as an example: the left contour of the right child consists of node D, E, F, while the left contour considered when moving the right child are D and E which in fact cannot reoccur in another left contour. That is because the contour is composed by all the nodes exposed from the left; hence, in a situation in which the current subtree has been already moved, all the left contour nodes that were processed would have a left adjacent node; in the case in Figure 3.11 the left adjacent nodes are A and C. In practice, the nodes belonging to left contour of the current subtree, that has been already processed, cannot be part of the merged contour: they are hidden from the left by adjacent nodes. Of course, an analogous assumption holds for the right contour.

Let consider that the input tree consists of "n" nodes  $v_1, \dots, v_n$  and suppose that  $f_l(v_i)$  is the set of nodes belonging to the left contour processed to push right the subtree rooted at  $v_i$ . We know, thanks to the assumption, that if  $v_i$  belongs to the set  $f_l(v_j)$ , it cannot belong to any other set  $f_l(v_z)$ , with  $z \neq j$ . Thus, the total number of nodes belonging to the left contour processed to build the the entire tree layout is certainly less than or equal to  $n$ .

$$\sum_{i=1}^n |f_l(v_i)| \leq n$$

Where  $|f(\cdot)|$  represent the number of elements in the set f. Obviously the same consideration can be done for the set of right contour nodes that were processed to move the subtree rooted at  $v_i$ :  $f_r(v_i)$ .

Now, the proof for the layered case is trivial. Consider that  $f(v_i)$  is the total set of contours pairs processed to move the subtree rooted at  $v_i$ ; since in this case node are aligned vertically:

$$|f_l(v_i)| = |f_r(v_i)| = |f(v_i)| \leq n$$

In the non-layered case, nodes are not necessarily vertically aligned. Thus, to consider the worst case we take the one in which nodes belonging to right and left contours are never aligned. The amount of nodes pairs to be confronted in order to move a child is no longer the same as the number of left (or right) contour nodes processed. In this case, after processing a pair, for considering the next we can advance either along the left or the right contour or along both. Thus, we can define an upper bound on the number of contour pairs processed:

$$|f(v_i)| \leq |f_l(v_i)| + |f_r(v_i)|$$

In this case a node in the left contour processed to move the subtree can be included in another left contour; the same for right contour nodes. Consider Figure 3.12 as an example: node 4 is in the right contour when moving the subtree rooted at 7, but it is also in the right contour of the merged tree. However, this can only happen to a node that is the last right contour node; because the top part of the last node considered is hidden by nodes to the right while other nodes that were considered must be totally occluded by nodes to the right. In the merged contour node 4 is, in fact, partially hidden by two nodes (7 and 8), while the other nodes considered, belonging to the right contour (3 and 6), are totally occluded by nodes to their right (again 7 and 8). The same holds for the last left contour node.

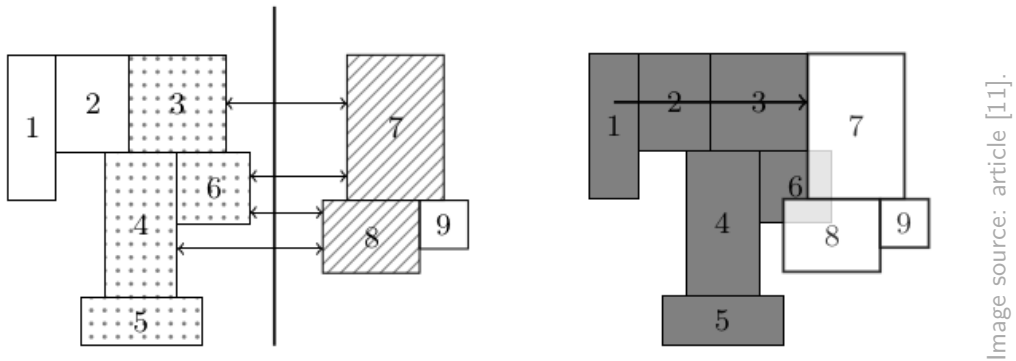


Figure 3.12. Example of Contour Pairs in a non-layered tree.

Thus, assume  $f_l^p(v_i) = f_l(v_i) - l_i(v_i)$  is the set of nodes belonging to the left contour processed to move the subtree rooted at  $v_i$ , with the exception the last left contour node that was processed, that is represented by  $l_i(v_i)$ .

Since only last elements of a contour set can appear again in another one:

$$\sum_{i=1}^n |f_l^p(v_i)| \leq n$$

Since we know that there are  $n$  subtrees and that, whenever moving one of them, there remains only a last node belonging to the left contour to be considered:

$$\sum_{i=1}^n |f_l(v_i)| = \sum_{i=1}^n [|f_l^p(v_i)| + |l_i(v_i)|] = \sum_{i=1}^n |f_l^p(v_i)| + n \leq 2n$$

Due to the fact that the same can be stated for right contour, we have that:

$$\sum_{i=1}^n |f(v_i)| \leq \sum_{i=1}^n |f_l(v_i)| + \sum_{i=1}^n |f_r(v_i)| \leq 4n = O(n)$$

This proves, generally, the linearity of RT algorithm extended for non-layered trees.

### 3.4.2 Improving Layout Techniques

The main problem in the RT algorithm extended to non-layered tree is that, while it satisfies the former four Aesthetics provided in this section, it is not able to satisfy Aesthetic 5 (Definition 3.4.5): the reflection of a tree is not represented as the mirror image of the original tree.

Also Walker [10] noticed this issue, that is caused by the fact that when subtrees are enclosed in between larger siblings they will be piled to the left; which also is not very aesthetically pleasing, to be honest.

Walker propose a method to distribute the extra space by moving proportionally the small subtrees in between the larger ones; then, Buchheim, Junger and Leipert [5] proposed an alternative method to employ Walker's idea in a linear way, by making use of *shift* and *change* field. This technique is applicable without difficulties in the non-layered case.

The method that, instead, is not applicable to non-layered tree is the one related to the function used to find the greatest uncommon ancestors. Basically, the function, given a node in the right contour, gets the index of the sibling subtree containing that node.

The BJJ solution requires to update all the nodes *ancestor* pointer in the right contour of a subtree, after moving that subtree, but only if the subtree is less tall than its left sibling, because in this case all its left contour nodes are considered to move that subtree. Thanks to the fact that, in the layered case, the left and right contours which are compared must have exactly the same number of elements, the number of right contour nodes of a subtree that is shorter than its left sibling subtree is equal to the number of contours pair considered to move that tree. Therefore, updating the right contour of a subtree shorter than its left sibling does not modify the linear runtime.

Differently, in the non-layered case this technique causes an  $O(n)$  runtime, because left and right contours are not constrained to have the same number of elements. To prove this point Van Der Ploeg [11] provides an example of a tree  $T'_k$  constructed in the following way:

"The root node has width and height  $2^k$  and has three children: A child consisting of a single node of width  $\frac{1}{4}2^k$  and height  $\frac{5}{4}2^k$ . A child subtree constructed in the same way with  $k = k - 1$ ". A child consisting of a single node of width  $\frac{1}{4}2^k$  and height  $\frac{1}{4}2^k$ ."

When  $k = 1$ , the tree is formed with the same method, with the exception of the middle child that is a leaf with width 1 and height 1. Notice that, for

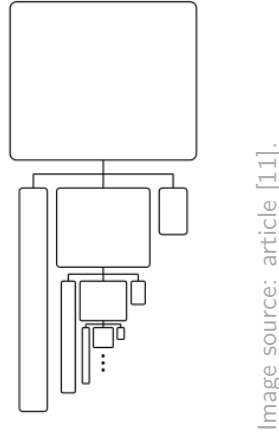


Figure 3.13. Example of  $T'_k$  tree.

every value of  $k$ , the middle child is shorter than its left sibling, as can be seen in Figure 3.13. Hence, the nodes in the right contour of the middle child must always be updated whenever it is pushed to the right. Considering that the particular tree built in this example has  $3k + 1$  nodes, the middle child right contour consists of all nodes in its subtree:  $3(k - 1) + 1$ . Hence the number of updates that must be done is:

$$\sum_{i=1}^{k-1} (3i + 1) = 3 \sum_{i=1}^{k-1} i + k - 1 = \frac{3}{2}k(k + 1) + k - 1 = O(k^2)$$

The result is obtained applying the Gauss formula for computing the first  $k$  natural number:  $\sum_{i=1}^k i = k(k + 1)/2$ ; since there is a linear relation between  $n$  and  $k$ , the algorithm runs in  $O(n^2)$ .

Thus, in order to find the lowest uncommon ancestor, VDP algorithm adopted a much simpler technique. It maintains a linked list of the siblings that currently have a node in the right contour; this list contains both the index of the corresponding sibling and its lowest vertical coordinates; moreover, it is sorted in descending order per index. Figure 3.14 provides an example. Whenever pushing a child subtree to the right, if the current node belonging to the right contour has a vertical coordinate lower with respect to the head of the list, the list is advanced. This operation only costs  $O(1)$  for each contour pair.

In order to update the list, after each call of the first walk function, the elements at the head of the list having a highest lowest coordinate than the new pair are removed. These removed nodes are still in the right contour, but they are occluded

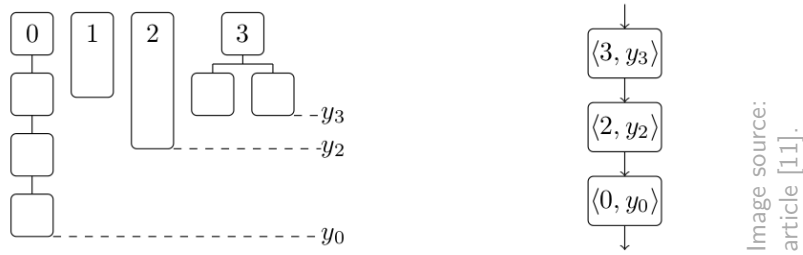


Figure 3.14. Example of Sibling Lookup Linked List.

by the current subtree. Then, we prepend the new pair to the list, so that the list always corresponds to the siblings that currently have a node in the right contour.

The number of operation needed for the list update is at most  $2m(v_i)$ , where  $m(v_i)$  is the number of children of node  $v_i$ . The reason is that the list is updated once after moving each child; when it is updated a certain number of elements can be removed and, of course, at most can be removed all elements, but we added  $m(v_i)$  elements to the list, hence in the worst case we have  $m(v_i)$  removals.  $m(v_i)$  removals multiplied for  $m(v_i)$  children gives the total number of operations, which, summed for the whole tree, is:

$$\sum_{i=1}^n 2m(v_i) = 2 \sum_{i=1}^n m(v_i) = 2(n - 1) = O(n)$$

Because every node is child of exactly one other node, except for the root which has no parent, therefore the sum of the number of children of all nodes  $\sum_{i=1}^n m(v_i)$  is equal to  $n - 1$ . Since the list advancement costs  $O(1)$  per contour pair, this will add an extra  $O(n)$ . The runtime is, thus, linear with respect to the number of nodes in the tree.

## 3.5 Considerations

The history of tidy trees algorithm is quite long, since Wetherell and Shannon [14] advanced the first  $O(n)$  algorithm that produces tree drawings satisfying some aesthetics rules. The algorithms discussed in the previous sections represent the state of art in literature and provides a sufficiently wide overlook of the problems involved in the tidy tree drawing task and the techniques used to overcome them.

The following table summarizes the analysed algorithms main characteristics.

For the purpose of our work, the development of a JavaScript library for interactive hierarchy visualization, the VDP algorithm was chosen. It combines all the positive aspects of older algorithm, representing the most flexible one.

Table 3.1. Algorithms Comparison

Algorithm	Year	Only Binary	≠Node Size	Only Layered	Linear
WS	1979	yes	no	yes	yes
RT	1981	yes	no	yes	yes
W	1990	no	yes	yes	no
BJL	2002	no	yes	yes	yes
VDP	2014	no	yes	no	yes

Another point in favour of the choice of VDP is that *D3.js*, the library on which the *ggen* JavaScript library relies on, implements already the enhanced Walker’s algorithm (BJL), but how it will be explained in Chapter 5, *D3.js* implementation alone is not enough for reaching the expected results.

For these reasons, it has been chosen to translate the VDP Java implementation, which Van Der Ploeg provided in [11], and integrate it in the *ggen* library.

# Chapter 4

## D3 Framework Overview

The purpose of this chapter is to present an overview of *D3*, a popular JavaScript visualization framework, on top of which we designed the graph-generator library (*ggen*) described in Chapter 5.

When building visualisation for the web, designer usually employ different tool simultaneously, in order to combine the various technologies involved in the creation of a web page. The following is a list of technologies that will be considered in this chapter: HTML, CSS, SVG, DOM, W3C DOM, JavaScript, jQuery. Some of them are strictly related to the *D3.js* framework. In Appendix B, it can be found the complete list of acronyms and definitions.



Figure 4.1. D3 Logo.

Among of the technologies used in web development, the DOM is probably one of the most important. It enables a common, standard representation of the content of a web page, which it describes in its hierarchical structure. It allows reference and manipulation of elements within the page.

In addition, through the DOM, modern browsers provide efficient ways for displaying the element tree, the inherited style values and allowing interactive debug.

As discussed in [4], many visualization toolkit and low-level graphical libraries, such as Processing and Raphaël, exist for helping web designers. But, while they provide a substantial gain in efficiency, reducing developers effort in specifying a visualization, they have some disadvantages.



Firstly, they produce a loss in interoperability, due to the encapsulation of the DOM in more specialized form: basically each toolkit provides its intermediate representation between the DOM and the user. Then, they reduce expressiveness, the possibility of representing different visualizations, and introduce a runtime overhead.

Furthermore, many graphics libraries do not provide a scenegraph inspector, which is a useful tool for debugging, often provided natively by browsers (example in Figure 4.2). Nevertheless, even when provided, toolkit-specific scenegraph abstractions may reduce compatibility and expressiveness: for example when elements cannot be styled using external stylesheets, or when some graphical effects are not available, even if they are provided natively.

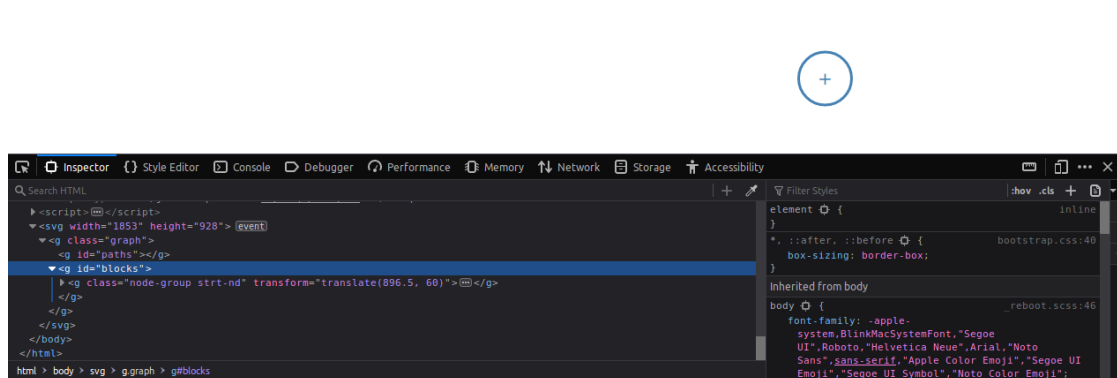


Figure 4.2. Mozilla Firefox Scenegraph Inspector.

Moreover, toolkit-specific graphical abstractions may vary among different toolkits and standards adding a hurdle to new users. In [4], it is brought as an example the case of drawing a wheel: in processing it has to be used an operator *ellipse* taking 4 arguments; in Raphaël a *circle* operator that takes three arguments. Both slightly different from the standard SVG *circle*.

For those many reasons, M. Bostock, V. Ogievetsky and J. Heer, in an article [4] published in 2011, provide the description of a new approach to visualization for the web. On this representation-transparent, standard-focused kind of approach the *D3.js* framework is based.

In this chapter we will provide a brief explanation of how D3 is designed, we will describe its base structure and provide some example of usage that may be helpful to understand the *ggen* implementation.

## 4.1 Data-driven DOM Manipulation

*D3.js* (D3 stays for Data-Driven Documents) is a JavaScript library used to create dynamic and interactive visualization, starting from structured data.

It has been designed as a collection of modules that can be used independently. In particular, for the development of *ggen* library, they were used *Selections*, *Paths* and *Transitions* modules. See the API [3] for the complete reference to *D3.js*.

The main characteristic of D3 is that, being based on the most used web standards, as HTML5, SVG, and CSS, it is able to exploit the full potential of modern browsers, without getting rid of expressiveness and compatibility.

Moreover, it provides a declarative web-specific language for visual design aimed at mapping data to visual elements. While it does not strictly impose a toolkit-specific lexicon, it allows to overcome the verbosity of browser built-in API for manipulating DOM, by directly mapping data attributes to elements in the Document Object Model.

Some other interesting D3 features that will be detailed in this sections are:

- query-driven selection.
- data binding to scenegraph elements.
- document transformation as an atomic operation.
- immediate property evaluation semantics.

With regard to selection, there are many JavaScript libraries designed to overcome the verbosity and imperative approach to the DOM provided by JavaScript. This approach is not convenient and often requires manual iteration throughout array of objects, as can be seen in the code example below, where it is applied a document transformation that colors all paragraphs white.

```
1 /*Coloring the paragraph text white with pure JavaScript.*/
2 var paragraphs = document.getElementsByTagName("p");
3 for (var i = 0; i < paragraphs.length; i++){
4     var paragraph = paragraphs.item(i);
5     paragraph.style.setProperty("color", "white", null);
6 }
```

The same task can be accomplished in a more straightforward way using a CSS stylesheet.

```
1 /*Coloring the paragraph text white with CSS.*/
2 p {
3     color:white;
```

```
4 };
```

There exist many popular JavaScript libraries that enable more convenient DOM manipulation. Of these, jQuery is probably the most successful one. Here it is an example of the paragraph coloring task performed with jQuery.

```
1 /*Coloring the paragraph text white with jQuery.*/
2 $("p").css("color","white");
```

D3 share with jQuery the concept of selection: which means identifying a set of elements using simple predicates, similar to CSS selectors. Then, the operations, as the transformation that changes the color, can be done on the selected elements: in the example above *style* changes the css aesthetic rule relative to the html element identified by tag *p*.

In any case, all the methods above are not suitable for dynamic data visualisation, because for this kind of task the document transformations must handle the creation and deletion of elements, not just the styling of existing ones. This is, of course, impossible with CSS, tedious with pure JavaScript and, also, not so easy with jQuery, as it lacks a mechanism for adding and removing elements to match a dataset; with jQuery data can be bound to node individually in case of need.

Let's take a closer look to D3 *selection*. D3 adopts the W3C Selector API, which is a standard mini-language used to identify elements for selection. This language is able to filter elements composing a page by html tag (e.g "p"), class (e.g ".class"), unique identifier (e.g. "#id"), attribute (e.g. "[name=value]"), containment (e.g. "parent~child"), adjacency (e.g. "before after") and various other predicates. In addition, predicates can be intersected (e.g. ".classA.classB") or unioned (e.g ".classA, .classB"). Thanks to all the combination of the seen rules, this simple language guaranties an adequate number of selection possibilities.

In D3 the keyword *d3* is used to access all the methods and objects exposed in D3 namespace. The two methods used for selection are *select*, for selecting the first element matching the predicate, and *selectAll*, returning all matching elements in document traversal order. The following code shows how the paragraph coloring task can be done with D3.

```
1 /*Coloring the paragraph text white with D3.*/
2 d3.selectAll("p").style("color","white");
```

Selection methods can be chained to generate subselection or elements grouping. For example, in the following code, the first line return the first bold element

(identified by tag `b`) in every paragraph; the second line return all the bold elements within paragraphs, grouped by paragraph.

```
1 d3.selectAll("p").select("b");
2 d3.selectAll("p").selectAll("b");
```

Then, on any element selection there may be applied a wide range of operators, compliant with the W3C DOM API, such as: *attr*, *style*, *property*, *html*, *text* and many others. The elements in a selection can be also looped over with the *each* operator, or accessed directly as in an array(e.g [0]). Again, method chaining is allowed, to apply multiple operators.

Another winnig feature of D3 is that, differently from other DOM-manipulation frameworks as jQuery, often values passed to operators can be specified as functions of data, not just as simple constants. These functions can be surprisingly useful. The following is a trivial example coloring of different shade of gray odd and even paragraphs: data is passed to functional operators as the first argument *d*, while the second argument *i* is the index of the datum inside the selection.

```
1 d3.selectAll("p").style("color", function(d, i) {
2   return i % 2 ? "#fff" : "#eee";
3 });
```

Now, it has been explained that selection is an atomic operand returning a filtered set of elements queried from the current document, and that operators can be applied on selections to modify the content, also in a dynamic way, by specifying parameters as function of the data. What remains to know is how this data can be linked to DOM elements.

In order to bind some generic input data to elements D3 introduces an abstraction, mutuated from relational algebra, called data joins.

First, the *data* operator is used to bind the data, expressed as an array of arbitrary values (e.g. numbers, strings or objects) to a selection, that may also be an empty one. Once the data has been bound to the document, it is possible to omit the *data* operator: D3 will retrieve autonomously the previously-bound data.

By default, data is joined to elements by index but, in certain cases, it can be useful to bind each datum to a specific data element. For example, this can be particularly useful in transition when is needed for the user to be able to follow a node and its corresponding datum into an animation: we want that at the end of the animation the datum belong to the same element as at the beginning.

In order to achieve that, it is sufficient to pass a key function as second parameter to *data* operator: this function takes a data point as input and returns

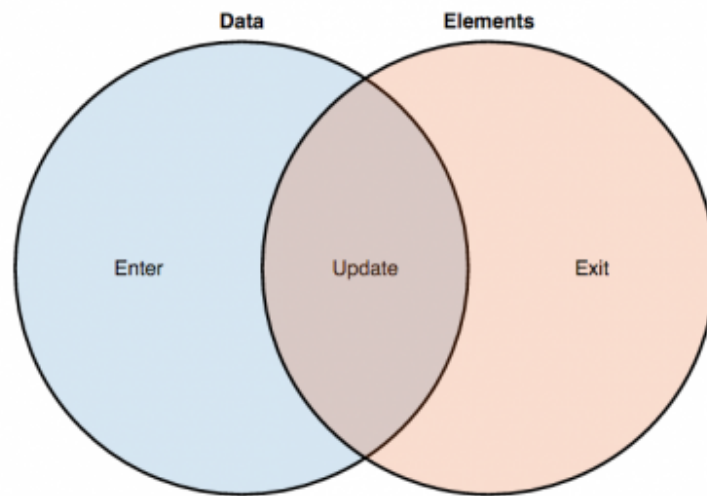


Image source: <https://bost.ocks.org/mike/join/>

Figure 4.3. D3 Data Joins.

a corresponding unique identifier, a key, which guarantees the matching datum-element and preserves the object constancy.

After a data binding, each datum and each element belong to one of three possible states. Data points joined to existing elements belong to the update selection. Leftover unbound data belong to the enter selection, representing missing elements. Likewise, any remaining unbound elements produce the exit selection, which represents elements to be removed.

These three states are also called data joins, because, as shown in Figure 4.3, they resemble relational algebra joins: the update selection corresponds to the inner join between dataset and element set, the enter to the left join and the exit to the right join.

In this way, exiting nodes, having no corresponding data, can be easily removed and a specific animation can be associated to them. Specific operators can be applied to nodes belonging to the enter and update selection as well, producing a dynamic visualisation driven by data. For example, properties that should be constant for the life of an element can be set once on enter, while dynamic properties can be recomputed on update.

In Figure 4.4 it is shown an example of join selections: in this picture data are represented as array of letter; when new data, in blue, are joined with old nodes, in orange, they are shown the three resulting subselections.

The *append* and *insert* operators add a new element for each element in the current selection returning the added node; thus, they can be used either for

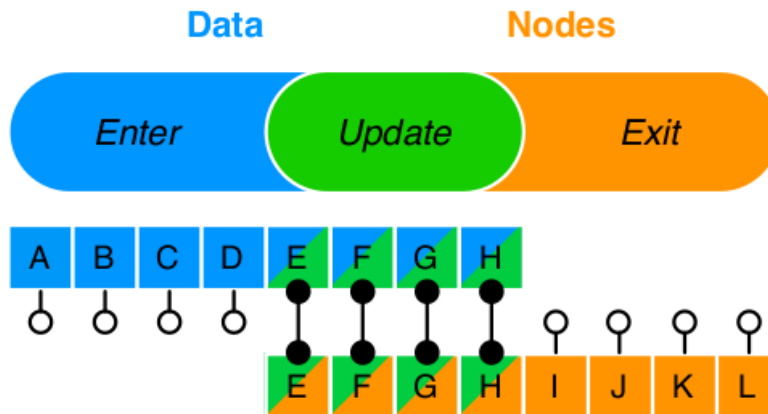


Image source: article [4]

Figure 4.4. Enter, Update and Exit subselections.

creating nested structure, or for instantiate new node based on entering data.

Other two convenient operators are *sort*, for reordering nodes, and *filter*, which returns a subselection based on a filter function.

Moreover, D3 *on* operator exposes the support to event listener, which receive user input targetted at a specific DOM element. Consistently with other operators, *on* callback receives the data and the index associated with the target element (the element itself can be retrieved using the keyword *this*, while the current event using *d3.event*) allowing data-driven user interaction.

Finally, through the *transition* operator, it is possible to create animated transition derived by selection. After a transition has been applied on a selection, operators applied next, as *style* and *attr*, are going to behave differently: basically they will interpolate from the current state to the newly specified state gradually over time. Both delay and duration of a transition can be specified as function of data.

D3 automatically manages transition scheduling guaranteeing efficient and consistent timing through a unified timer queue; this mechanism scale easily to thousands of concurrent timers.

To sum up, transition enables dynamic visualizations through explicit control over which elements are mutated, added or removed, and how. In addition, they helps keeping separate the manipulation from the generation of the DOM, by being able to manipulate already existing document. Hence, there is the possibility to build the initial state of the visualization on server-side, and then apply dynamic behaviours on client-side.

One last thing has to be said to complete this rapid overview on D3 scope and functionalities. Whether some application act by deferring evaluation of property

functions to the final rendering phase, to allow implicit reevaluation of properties, D3 applied operators immediately: this reduce control flow internal to the library moving it up to the user code. While the former approach can be convenient in term of runtime, the one used by D3 simplifies the internal structure of the library and gives more control to the users.

## 4.2 Considerations

As described in the previous section, D3 represents a flexible and standard-transparent solution to dynamic data visualization for the web.

Its transformation module makes dynamic visualization easier to implement on top of element selection, which eliminates redundant computation: transformation involved only the elements and attributes that need updating or that have been selected, rather than the entire scenegraph.

By adopting immediate evaluation of operators and the browser's native representation, D3 improves compatibility and debugging. The native representation also provides CSS support for sharing simple properties definition and has many advantages, as previously discussed, including interoperability, the presence of a large documentation and expressiveness.

In addition, combining transformation and immediate evaluation D3 is able to reduce overhead: the DOM is modified directly, avoiding the indirection of an intermediate scenegraph. This design choices improve performance with respect to higher-level existing framework, as discussed in Bostock's article[4].

D3 has become highly popular among web developers and, as said by its creators [4], "keeps pace with the evolving technological ecosystem of the web improving *expressiveness* and *accessibility*".

Whether D3 is highly customizable, it also provide a vast selection of optional modules, based on strong reliable and reusable solutions to common problem. This kind of approach can be efficiently summed up by the Tufte's principle [4]: "Don't get it original, get it right".

Which means that is better not to loose time on problems already solved by others. It is actually, following this principle that it has been decided to use the VDP algorithm to empower the *ggen* library, and to make it rely on *D3.js* well-proved methods for animation and data visualization.

# Chapter 5

## GGen Library Implementation

In this Chapter we will provide a description of the methods and techniques used for implementing the graph-generator (*ggen*) JavaScript library. Although the library name derives from its specific purpose in the LAS environment, which is supporting the creation and drawing of an acyclic direct graph representing the data flow of a query, the library is agnostic to this scope: it can potentially be used in other contexts, for example to present and interact with hierarchical data.

How it will be shown in the next Sections, the library is based on *D3.js*, the well-known JavaScript framework for data visualization described in Chapter 4. D3 will be exploited by *ggen*, in particular as far as concerns *selections*, *paths* and *transitions* modules, but also for the library structure and the programming pattern concerning data joins.

The following is a list of some technologies and techniques that may be useful to understand the rest of the Chapter: UMD, CommonJS, AMD, ES6, IIFE, Bézier Curve. Their definition can be found in Appendix B.

### 5.1 Requirements

In order to achieve the goal of this thesis, the new LAS query module interface should respect the requirements listed in this Section. Firstly, the new features that need to be implemented are the following:

**Requirement 5.1.1.** Responsivity. The application should be accessible from different devices and has to scale with the size of the viewport.

**Requirement 5.1.2.** Predefined Path. User should be led through a predefined path in the building of the query tree. This is required to reduce user errors, reduce user effort and increase usability. In addition this will reduce the number of client-side integrity checks to be performed.



**Requirement 5.1.3.** Hierarchy. The data flow visualization needs to be ordered and hierarchically structured. This increase usability and reduce overhead, because the application has no more to keep track of the  $x$  and  $y$  coordinates of each node, the coordinates should be automatically defined by the position of the node in the hierarchy.

**Requirement 5.1.4.** Draggability. In order to increase user experience, the user should be able to move rigidly the query tree by drag and drop. Optionally the tree could be made zoomable.

The old requirements that have to be supported also in the enhanced implementation, instead, are:

**Requirement 5.1.5.** Start node is unique, cannot have any entering arc, but can have any number of children.

**Requirement 5.1.6.** End node is unique, can have just one arc entering and cannot have children.

**Requirement 5.1.7.** Nodes different from start and end are divided in two types: entity node, with one input, and operator node, with two inputs. There is also an exceptional entity node with 4 inputs.

**Requirement 5.1.8.** Nodes different from start and end must have at most one output.

**Requirement 5.1.9.** Nodes different from start and end must be named, must be configurable through a menu and must be removable.

**Requirement 5.1.10.** When a node is deleted, also the arcs entering in it must be deleted. The entire subtree induced by the node must be deleted as well.

## 5.2 Methods and Techniques

### 5.2.1 Module Loader

For designing the library, we adopted the Universal Module Definition, a standard way to guaranty the library to be supported by different module loaders, such as CommonJS or Asynchronous Module Definition. We could have been chosen the ES6 standard module export, but UMD has a better compatibility, also, with older browsers. Thus, coherently with *D3*, it has been chosen.

As shown in the following code snippet, UMD is just a series of if-then-else statements to identify the module-loading style that the current environment supports.

```

1 (function(global, factory){
2
3     if (typeof define === 'function' && define.amd) {
4         define(['exports', 'd3'], factory);
5     } else if (typeof exports === 'object') {
6         factory(exports, require('d3'));
7     } else {
8         (factory((global.ggen = global.ggen || {}), global.d3));
9     }
10
11 }(this, (function(exports, d3){ 'use strict'; })));

```

UMD pattern is composed in two parts. First, an Immediately Invoked Function Expression that checks which the module loader is being implemented by the user. This will take two arguments: *global* is a reference to the global scope (that we pass to the IIFE through *this*); *factory* is the function where we declare our module. Then, the anonymous function in which we define our module. This is passed as the second argument to the IIFE.

In the example above the code checks first if the environment uses AMD, then if it uses CommonJS. If neither of those loaders are in use we make the module and its dependencies available globally.

In case of AMD, it is defined an array of dependencies (in our case containing just the our exposed variables and methods, *exports* and *d3*) and a callback function, *factory*, which is only executed when the dependencies are available. In case of CommonJS, the *require* function is called to check the dependency (*d3*).

Thus, the use of this JavaScript Module Pattern, allows to enforce some fundamental properties needed by a library, such as "code reusability" (the library must work on its own independently by the environment) and "dependency resolution" (the user should not be in charge of taking care of the dependencies order).

When a module loader is used, it also allows to avoid the so called "pollution of global namespace": avoiding to have all the functions and variables reside in global scope. When a module loader is not used, we can still reduce global scope pollution; in fact, thanks to module object and IIFE, we expose to global scope only one single object, containing all the methods and values we need (in our case it is called *ggen*).

Basically, this is based on the fact that in JavaScript, every function, when invoked, creates a new execution context. Because variables and functions defined within a function may only be accessed inside that context. It is possible to say that all of the code that runs inside the function lives in a closure, thus providing privacy.

The round brackets around the anonymous function are required for the parser

to be able to understand that that is a function expression and not a function declaration.

Finally, the ‘use strict’ enables the strict mode, which does not allow to delete functions objects and variables, does not allow to reference not defined objects or variables and it does not allows to use keyword as variable names. Thus, it helps producing a much more secure and clean JavaScript code.

## 5.2.2 Versioning

Also in defining the versioning method for our graph-generator library, we chose to emulate *D3*. Thus, the current version is exposed in *ggen.version*, exactly as in *D3* is in *d3.version*.

We decided to use semantic versioning [12]. That, basically, means that the version is a number, composed by three part: *MAJOR.MINOR.PATCH*.

- The MAJOR number is incremented when an incompatible API changes is made.
- The MINOR number when a new functionality is added, in a way that is backwards compatible.
- The PATCH number is incremented when a simple backwards compatible bug fixes is made.

## 5.2.3 SVG

As said in Chapter 4, *D3* uses the Scalable Vector Graphic standard. As regards our library, we use four standard SVG abstraction: groups, paths, shapes and texts.

The first, SVG Group, is the most important because it enables to apply transformations and transition on multiple different elements at the same time. It is defined by a tag  $\langle g \rangle \langle /g \rangle$ . Every SVG element inside that tag are considered part of the group; any transformation applied to the SVG Group is applied to all of the child elements contained inside.

There are six types of transformations available, that may be concatenated in the *transform* attribute of tag  $\langle g \rangle$ :

- $\text{matrix}(\langle a \rangle \langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle \langle f \rangle)$ . This transform specifies a transformation in the form of a transformation matrix of six values.
- $\text{matrix}(a,b,c,d,e,f)$ . It is equivalent to applying the transformation matrix  $\begin{bmatrix} a & b & c & d & e & f \end{bmatrix}$

- `translate(<x> [<y>])`. This transform specifies a translation by  $x$  and  $y$ . If  $y$  is not provided, it is assumed to be zero.
- `scale(<x> [<y>])` This transform specifies a scale operation by  $x$  and  $y$ . If  $y$  is not provided, it is assumed to be equal to  $x$ .
- `skewX(<a>)` This transform definition specifies a skew transformation along the X axis by  $a$  degrees.
- `skewY(<a>)` This transform definition specifies a skew transformation along the Y axis by  $a$  degrees.

In our graph-generator library we used only the translate transition for positioning the nodes.

Regarding SVG Paths, they represent the outline of a shape that can be stroked, filled or used as a connection link. Theoretically, one can use an SVG Path to make any type of SVG shape. The shape of an SVG Path element is defined by one attribute:  $d$ , which contains a series of commands and parameters in the SVG Path mini-language. These commands and parameters are a sequential and case-sensitive set of instructions.

- *moveto*. It sets a new current point.
- *lineto*. It draw a straight line.
- *curveto*. it draws a curve using a cubic Bézier.
- *arc*. It draws an elliptical or circular arc.
- *closepath*. It closes the current shape by drawing a line to the last moveto.

Drawing complex graphs with just these instructions can be uncomfortable, hence *D3* includes a set of helper classes for generating them automatically: functions that convert our data into the SVG Path mini-language.

Among these helpers, to draw the arc connecting nodes, we used some path generators for cubic béziers, such as: `linkHorizontal`, `linkVertical` and `linkRadial`.

```
1 arcs.attr('d', d3.linkVertical()  
2   .source(function (a) {return [a.src.x, a.src.y]})  
3   .target(function (a) {return [a.dst.x, a.dst.y]});
```

In the example above, considering that *arcs* is a *D3* selection, *d3.linkVertical* will draw a cubic Bézier from the  $(x, y)$  coordinates returned in the source callback function, to the  $(x, y)$  coordinates returned by the target callback function, for each arc in the selection. It is possible to notice how the values of  $x$  and  $y$  returned are taken directly from the data associated to the arcs, specifically the source coordinates from the object *src*, the destination coordinates from *dst*.

SVG Texts and Shapes are similarly positioned using *D3*. In the example below it is possible to see how one SVG Text element, corresponding to tag `<text>`, is inserted for each element in *groups\_selection*.

```

1 groups_selection
2   .append('text')
3   .attr("class", "node-title")
4   .attr("x", function(d){return (d.size.width/2);})
5   .attr("y", function(d){return (d.size.height/2);})
6   .attr("font-family", "sans-serif")
7   .attr("font-size", "0px")
8   .attr("fill", "steelblue")
9   .attr("text-anchor", "middle")
10  .attr("dominant-baseline", "central");

```

To each text element it is given the same color and font style, while the positional coordinates are given based on data. In this example, the data corresponding to a group in *groups\_selection* contains a *size* object defining *width* and *height*; the callbacks return respectively  $x$  and  $y$  in order to center the text over the group.

Notice the *text-anchor* attribute and the *dominant-baseline* attribute. The former define which point in the text has to be considered for anchoring it to its  $x$  and  $y$  coordinates: if *start*  $(x, y)$  will correspond to the left edge of the text, if *middle* to the center, if *end* to the right edge. The latter define the baseline, which is the line where text naturally sits.

In order to insert in a group a shape the procedure adopted is quite similar: see the example below, in which a circle is added to the selection.

```

1 groups_selection
2   .append("circle")
3   .attr("r", 1e-6)
4   .attr("cx", function(d){return (d.size.width/2);})
5   .attr("cy", function(d){return (d.size.height/2);});

```

*D3* also provides helpers for drawing tidy trees in *hierarchies* module but, unfortunately, they are not suited for our purposes, because they do not support nodes of different shapes and size, neither they support nodes with more parents (Requirement 5.1.7).

## 5.3 Implementation

In this section we will provide an high level description of the structure of *ggen* and its main methods. The library, which has been designed to draw layered tree of unbounded degree, can be used in the LAS Query Module interface as well as in the context of other web applications.

In this section, following the names given to them in the library, we refer to the tree nodes as *blocks*, with the exception of starting and ending node (that are called *start* and *end* respectively), and to arcs connecting the node as *edges*.

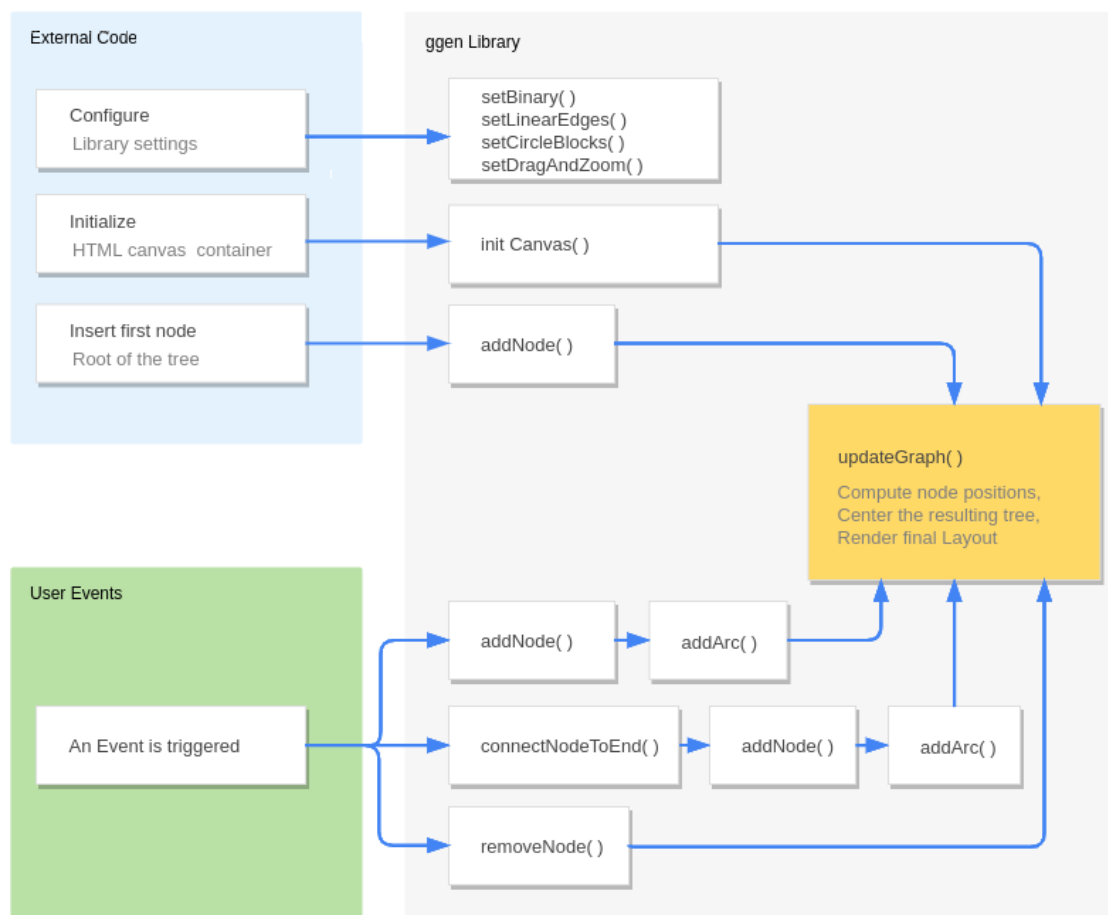


Figure 5.1. General Code Flow.

### 5.3.1 Configurability

The *ggen* library has been designed to be as configurable as possible. The configuration parameters are kept in three objects: *state* keeping the current state of the graph (e.g. current selected node and counter of node id), *settings*, containing properties that depends on the environment, as starting location of the nodes, and *constants*, containing all the other configurable properties that must be constant during the graph construction. The user can for example decide if the graph should be draggable or both draggable and zoomable and the library keeps track of this decision in *constants*. The modification of constants during the construction of the graph is not strictly forbidden, but can lead to undetermined results.

The following functions are the one available for changing the configuration.

- *setBinary(binary)*. If *binary* is true, it forces the tree to be binary.
- *setLinearEdges(linear)*. If *linear* is true, it sets linear edges, otherwise the graph has by default curve edges (enforced by using SVG path, as seen in section 5.2.3).
- *setCircleBlocks(circle)*. Set the block shape; by default are rectangles.
- *setDragAndZoom(draggable, zoomable)*. If *draggable* is set to true the graph will be movable at mouse drag: when the mouse is pressed on the canvas, the graph will rigidly follow the position of the cursor. If *zoomable* is true it will be possible to enlarge and scale down the graph using the mouse wheel.

### 5.3.2 Init and Update

The entire library is build on two main functions: *initCanvas*, *updateGraph*.

The first has to be invoked, after having eventually called the configuration function, and it is necessary to set up the SVG canvas and the settings. It is possible to pass to it a W3C selector (see section 4.1) defining the HTML element that is going to host the graph, it is recommended to use a `<div>`, a `<section>` or an `<article>` with a fixed width and height, to avoid unpleasant results.

The second is the fundamental function exploiting the *D3* update pattern; it is never used by the user, it is called by the library whenever a change is made on the graph in order to visualize the change inside the canvas. This function firstly compute the position of each node by using the VDP algorithm (Chapter 3.4 and Appendix A for reference) invoking *VDPtreeLayout*; then it performs the *D3* data join, as illustrated in section 4.1 and in the following code snippet.

```
1 // node update selection: existing nodes
2 var el_up = blocks.selectAll("g."+constants.nodeClass)
3                               .data(nodes, function(d){ return d.id; });
```

```

4 // node enter selection: new nodes
5 var el_en = el_up.enter().append("g")
6     .classed(constants.nodeClass, true);
7
8 //Transition update selection: update nodes to their new position.
9 el_up.transition()
10     .duration(constants.duration)
11     .attr("transform", function(d) {
12         //translate to current position x, y
13         return "translate(" + d.x + "," + d.y + ")";
14     });

```

Basically, for each selection returned by the *D3* data join, enter, update and exit, it perform different actions. The data to be associated with *blocks* and *edges* are contained in two array of objects, called respectively *nodes* and *paths*.

At line 2 of this example, the function selects all blocks element and computes the data join using a key function to bind permanently the SVG group composing the node to the corresponding data point (as key it is used an unique id). Then, it computes the enter selection, corresponding to new nodes, not yet in the canvas, and for each of them it inserts a new group (corresponding to a block), giving to it the class *constants.nodeClass*. Notice that it is preferable to use *.classed(..., true)* instead of *.attr("class",...)*, because the *.attr* method overwrites the entire class list. Finally, at line 8, it is provided an example of transition moving the nodes to the new *x* and *y* coordinates, applied on the update selection. Obviously, the same pattern is applied also on edges. At the end of the function exiting nodes and edges are removed, as follows.

```

1 // remove eliminated/old paths
2 path_up.exit().remove()
3
4 // remove old/eliminated nodes
5 el_up.exit().remove();

```

### 5.3.3 Nodes and Arcs

Other two fundamental functions belonging to the library are the two devoted to add nodes and arcs:

- *addNode(type, title, parent, nodeclass, numInputs, x0, y0)*
- *addArc(source, destination)*



*addNode* accepts many parameters; the most important are type, title and parent. Type is a string denoting the type of node, currently the library supports only four values: "start", "end", "block" and "operator", which is a special block, as explained in section 5.3.5. Title is a string containing the name of the block that will be displayed in the drawing. Parent, instead, must contain the data object associated with the parent node; it must be an object currently contained in *nodes* array, otherwise the library throws an error. The other parameters are optional: *nodeclass* is used to give a specific CSS class to the node; *numInputs* to specify the number of parent the node may have (currently this features is not fully supported); *x0* and *y0* are used to specify the starting position.

*addNode*, after having created the new data objects, pushes it into *nodes* array and invokes *addArc*, passing to it the parent as source and the new node as destination. *addArc*, after a some integrity checks and a precomputation phase, executed only in case of "operator" nodes (it will be explained later in section 5.3.5), creates a new arc object and pushes it into *edges* array; finally it calls the *updateGraph* functions to propagate the changes to the canvas.

Two other useful functions involving arcs and nodes are:

- *removeNode(t)*
- *connectNodeToEnd(t)*

The first is needed for removing nodes (Requirement 5.1.10). When a node is removed, also its references within the other nodes must be removed. Each node maintains a *parent* and a *children* queue, containing respectively the references to parents and children of that node. In order to do so we implemented a recursive solution: the function visits recursively all children of the node to be deleted setting their *id* to  $-1$ , and then filtering them out from the arrays referencing them. At the end of this recursion, the function filters out also from *nodes* the nodes with *id* ==  $-1$  and from *edges* the arcs with as source or destination a node with *id* ==  $-1$ .

The second function need to be used to connect a node to the end node; it has been implemented to consider the case in which more nodes have to be connected to a common end: this is not the case of LAS query module, but we tried to made the library as general as possible. In addition, this function returns false in case a node is already connected to end, allowing the application to handle this situation properly.

### 5.3.4 Customizable Functions

The feature of customizable functions is born with the aim of being compliant to Requirement 5.1.9 and to maintain the configurability property. The idea is to

allow the user to set the functions triggered by all the buttons belonging to the nodes; specifically the start and end block can trigger one function each, while block and operator node can trigger up to three functions: one by default deletes the node, another it is used to configure the node, the last one trigger the function to add a child node (see Figure 5.2).

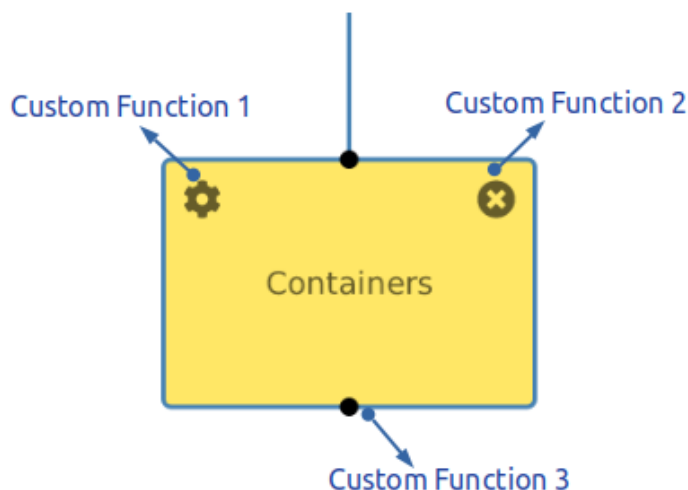


Figure 5.2. Block functions triggers.

- $setCustomFunction(f, type, i)$

The user can define its own functions and pass it, through parameter  $f$ , to the  $setCustomFunction$ ; in order to identify which node type and which button has to trigger the function parameters  $type$  and  $i$  are provided;  $i$  for end and start node is 1 in any case, while for the other nodes it can be 1, 2 or 3.

### 5.3.5 Multiple-input Nodes

In the LAS query module, a part for the start node, which may have any number of children, any other nodes should have just one child. This Requirement (5.1.8), that is a relaxation of VDP constraints, is easily absolved by the *ggen* library.

Although VDP algorithm has been designed to support tree with any number of children nodes, it does not support graphs with any number of parents. This arises an issue, because another LAS query module Requirement (5.1.7) asks that

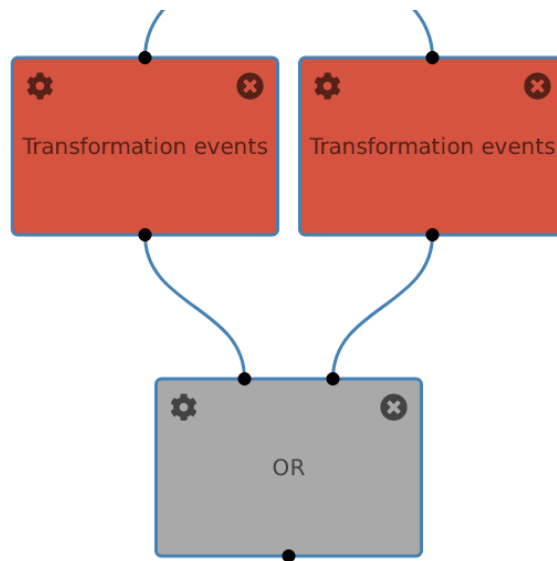


Figure 5.3. Operator node example.

some nodes can have two inputs. This eventuality is not considered by VDP and requires an algorithm extension specifically designed for that purpose. Moreover, there is an exception in which a LAS node may have four inputs. In this section we will explain how these two cases have been handled.

The function *addArc*, whenever the destination node is of type "operator" and is already connected to an input, does some operations. Firstly, it checks if the operator is positioned under the rightmost parent: if it is it swaps the parents, so it will be under the leftmost. Secondly, it searches the lowest node with more than one child and moves all nodes in between the children that are ancestors of the "operator"; in this way the two elements attached to "operator" are closer to each other. This is an heuristic way of producing a pleasant drawing, avoiding overlapping arcs. In LAS query module, in which only the start node can have many children, the goal is reached in the vast majority of cases; in other kind of applications this mechanism could have been improved.

The exception case in which a node of type "block" has four inputs is such rare that the library accepts arcs overlapping.

In order to position correctly in a nice way nodes with more than one inputs, it had to be introduced a correction to VDP algorithm. Thus, in *updateGraph* after VDP has already computed nodes' positions, it is invoked another recursive function that adjusts "operator" (and "block" with four inputs) positions and propagates the changes to their subtrees.

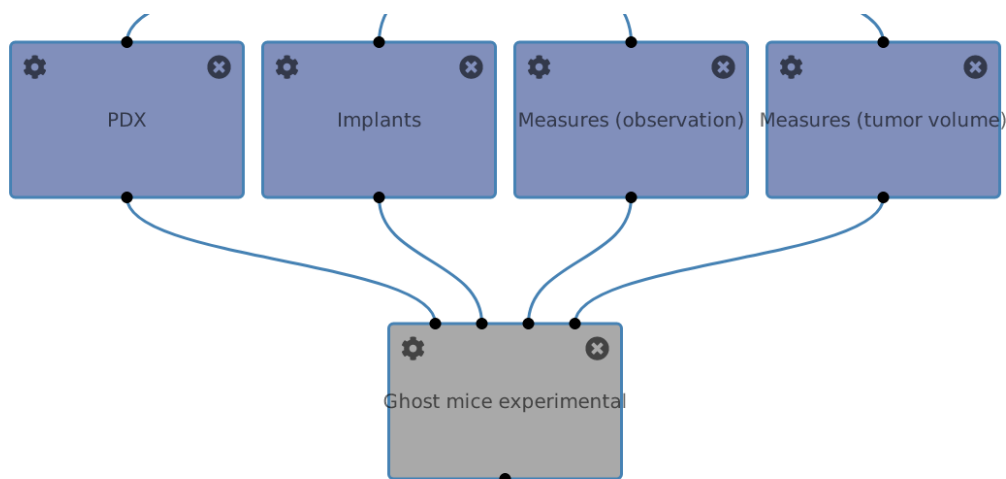


Figure 5.4. Exceptional 4-input node example.

- *moveNodesRecursive(node, offset)*

This function performs a traversal of the tree, changing the  $x$  coordinate of nodes with more than one input connected, placing them centered of two parents; as regards the  $y$  coordinates, it positions these node one layer below their deepest parent.

### 5.3.6 Load and Store JSON

Although not used in the LAS environment, these two utility functions can be useful whenever one wants to store a graph that has been build and later wants to be able to reload it. As noted in section 5.4 they may be used in some future improvement of the library.

- *storeGraphJSON()*
- *loadGraphJSON()*

The first one, decycles *nodes* and *edges* arrays of objects in order to stringify them in two JSON string. Then, it downloads a file named "gen-graph.json" containing them.

In order to reload the graph is sufficient to call *loadGraphJSON* and select a json file with a compatible structure. The function parses it and rebuilds the *nodes* and *edges* arrays, then it calls *updateGraph* to draw the graph inside the canvas.

### 5.3.7 Alternative Algorithms

One feature that has been implemented for testing some of the algorithms described in Chapter 3 is the possibility to select a different algorithms to compute nodes' coordinates, with respect to the default one (VDP).

- *setAlternativeAlgorithm(alg)*
- *triggerAlternativeAlgorithm()*

Calling the *setAlternativeAlgorithm* function we can decide which algorithm we want to use; at the moment the two supported are algorithm WS and RT. For both, the function configure the library in order to draw only binary trees with nodes of circular shape and linear arcs

The *triggerAlternativeAlgorithm* introduce some attributes in the nodes' data object needed by the alternative algorithm chosen and invokes *updateGraph*.

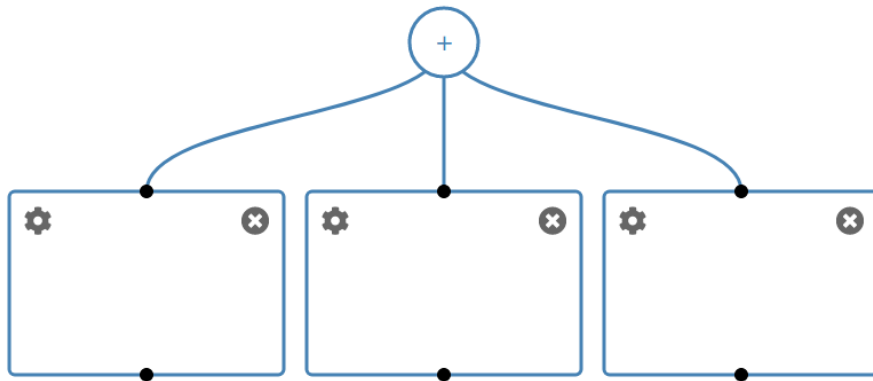


Figure 5.5. Example of VDP algorithm drawings with curve edges and rectangular nodes

## 5.4 Considerations

In this Chapter we gave an overview on how *ggen* is implemented. As it has been described, whether this library has been developed specifically for the LAS query module purposes, following the Requirements listed in section 5.1, it has been developed in a way that keeps open also other possibilities.

We think that the library may be improved to be used in other fields of application. Drawing trees is, in fact, a useful task in many fields of Software Engineering,

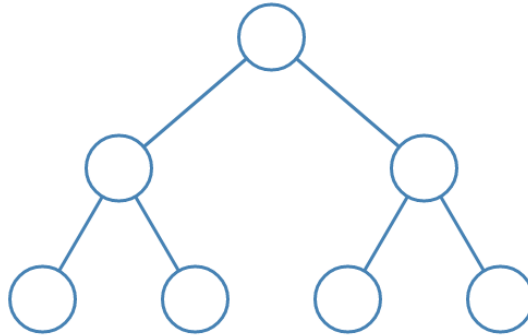


Figure 5.6. Example of RT algorithm drawing with linear edges and circular nodes.

such as Formal Languages (e.g parse trees), Relational Databases (e.g. query optimization trees) or Operative Systems (e.g. process tree diagrams).

In the next Chapter (6), we will provide some example of usage of the library in the LAS context, showing also how the query module interface has been developed in order to employ it.



# Chapter 6

## GGEN - Use Cases

The purpose of this Chapter is to present the modification applied on the LAS query module interface and how the new implemented library (*ggen*) has been integrated into the module. The main features introduced will be illustrated through some real use-case scenarios. Specifically, the following examples are provided:

- The procedure to draw and submit a query.
- The procedure to save a Template.
- The procedure to save a Translator.
- The procedure to reload of the last query submitted.

The application uses Bootstrap toolkit (in particular for Modal dialogs, Tab and Navbar menus) to be as responsive as possible (Requirement 5.1.1). Make reference to Appendix B for acronyms and definitions used in the rest of the Chapter (e.g. Bootstrap or JSON).

### 6.1 Background

The LAS query module interface relies on local data to make consistency checks on the graph drawn by the user. The query defined by the graph is, thus, sent to the module backend only when it has been assured to be consistent with the Data Integrator (DI) database (see Chapter 2).

The data used for performing all this kind of checks it is also used for building the menus, containing all the block that can be inserted into the workspace (section A and C in Figure 2.2). This data is contained in three JSON objects loaded together with the page: *qent*, the list of queryable entities, *ops*, the list of operators, and *templates*, the list of templates.



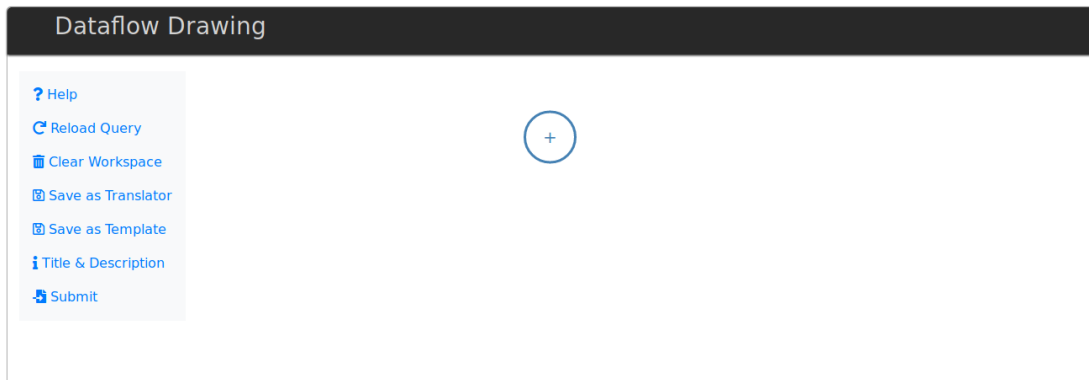


Figure 6.1. Query Module Starting Situation.

These objects are managed and maintained updated by the DI: they store information about each available Entity, Operator or Template and their available filters and configurations. The new interface heavily relies on them to provide the predefined path property (Requirement 5.1.2).

In Figure 6.1, it is shown the interface as it appears to an user, when he or she opens the Query Module. It is possible to notice how all the user possible actions has been grouped in a menu on the left-side of the page.

For example, here it is possible to open the help dialog (whose button was originally positioned in the top-left portion of the interface), clear the workspace or setting a query title and/or a description (that once were inside the D section, as shown in Figure 2.2). The other four buttons belonging to this menu triggers the use-case scenarios described in the following sections.

## 6.2 Create and Submit a Query

The creation and submission of a query is the most important task performed by the LAS Query Module, and it is also the one that is changed the most in the new interface.

The major change is in the way a user can insert a node in the workspace. As it can be seen in Figure 6.1, in fact, there are no more lists of entities and operators among which to choose the one to insert. Whenever an user wants to add a new node, he or she should always decide first to which node it would be connected; at the beginning the only node available is the *start* node.

In Figure 6.2 it is shown the dialog menu (Bootstrap Modal), opened by clicking on the *start* node. The same menu is opened by triggering function 3, clicking on the black output connector of a block (see Figure 5.2). Inside the menu each

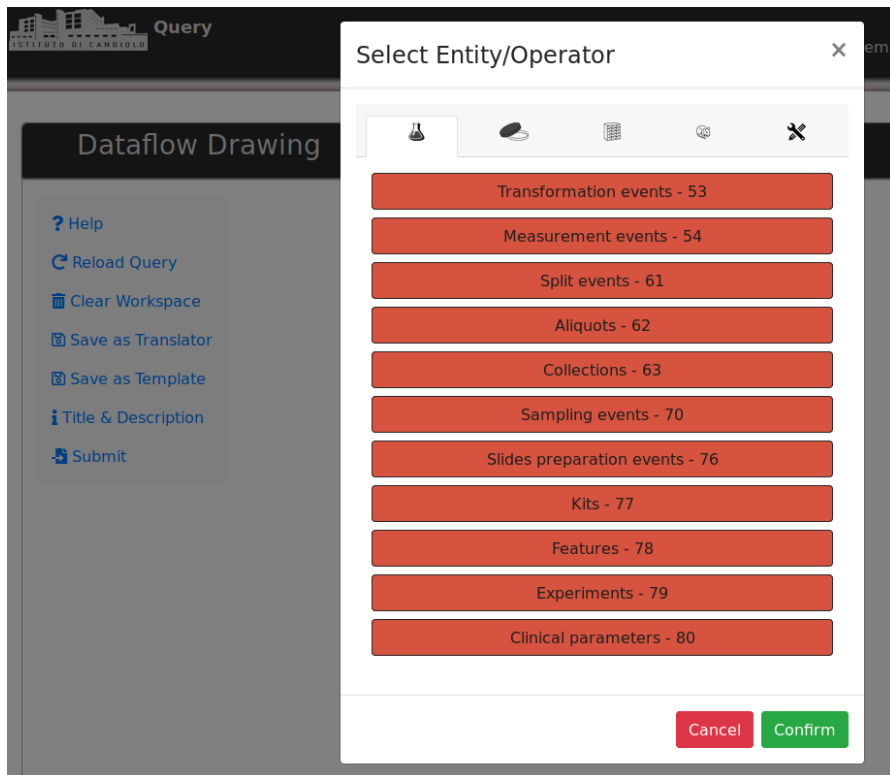


Figure 6.2. Select Entity/Operator Modal Menu (Biobank tab).

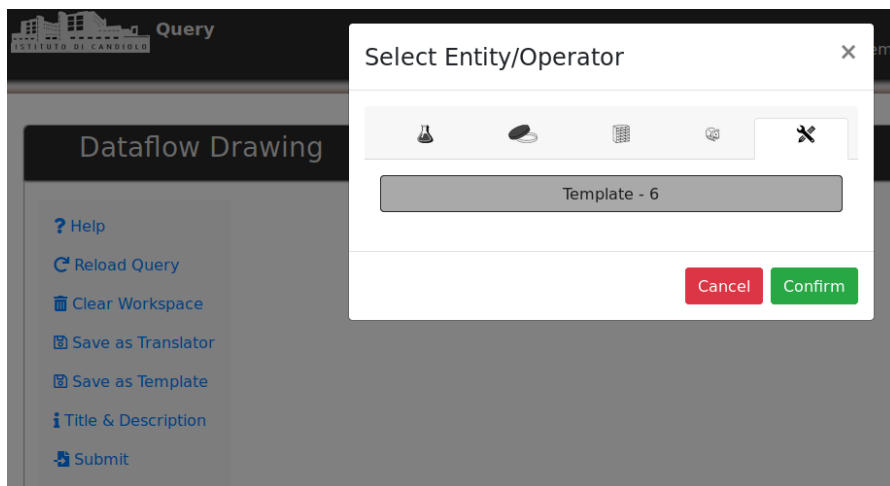


Figure 6.3. Select Entity/Operator Modal Menu (Operator tab).

tab (Bootstrap Tab) represent a list of entity belonging to a different LAS Module database; the last tab contains the Operators, the Template and the button to connect the current node to the *end* node.

In the example in Figure 6.2 the tab relative to a module called Biobank is active, while in Figure 6.2 the Operator tab is active. It is interesting to notice that in this particular case (remember that we opened the menu clicking on the *start* node) the only possible selection is Template. This is because the only node belonging to this category (containing, as said, Operators, Template and the connect-to-end option) that can be added to *start* is that one. Whenever opening the same dialog from a different node the menu content will be dynamically inserted, based on the available options that can be chosen from the current node we are considering.

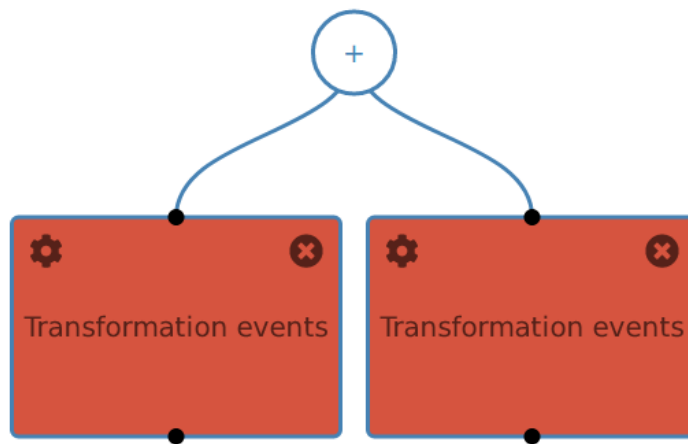
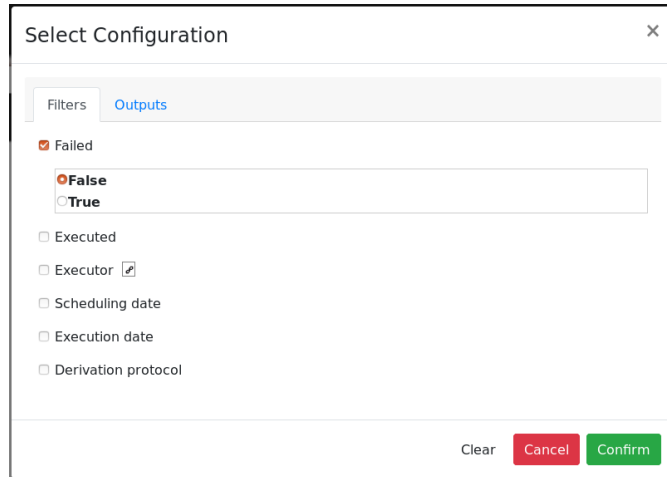


Figure 6.4. Query Graph Example.

The function that dynamically populates the menu with the current options works as follows. For the entities modules (Biobank, Cellline, Storage and Xenografts) tabs, if the current node is *start*, it inserts all entities listed in *gent*; if the current node is an Entity, it inserts only the queryable entities belonging to its *query path* (that is defined by the Administrator, as described in Section 2.1.1); instead, if the current node is an Operator (not a Template), it inserts only entities in the *query path* of its first parent node. In the case of last tab (containing Operators, Template and connect-to-node options) for the *start* only the Template option is available; in all the other cases the Operators contained in *ops* are inserted; moreover, the application checks if there are available operator missing an input, if so and if their parent is compatible with the current node, the options to connect

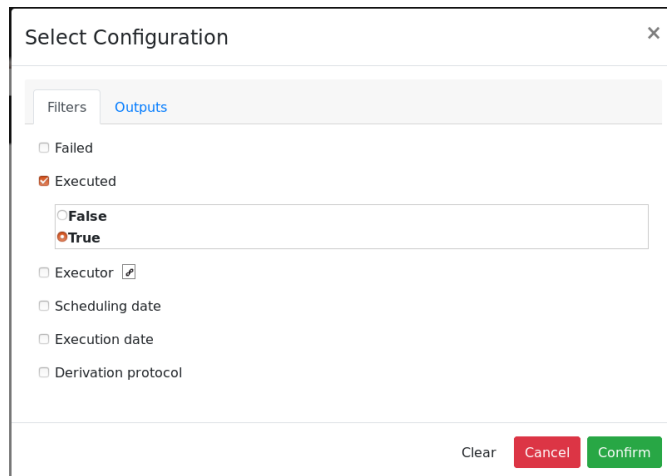
current node to them are inserted; the connect-to-end is inserted only for nodes with the *canBeLast* flag set to true.

At this point, suppose we want to add two "Transformation events" entities: we select the first option in Figure 6.2 and click "Confirm". Then, we press again on the *start* and repeat the actions done so far. The result is shown in Figure 6.4.



The screenshot shows a "Select Configuration" dialog box with a close button (X) in the top right corner. It has two tabs: "Filters" (selected) and "Outputs". Under the "Filters" tab, there are several checkboxes: "Failed" (checked), "Executed" (unchecked), "Executor" (checked), "Scheduling date" (unchecked), "Execution date" (unchecked), and "Derivation protocol" (unchecked). Below these is a configuration section with two radio buttons: "False" (selected) and "True" (unchecked). At the bottom right, there are three buttons: "Clear", "Cancel", and "Confirm".

Figure 6.5. Entity Configuration Example.



The screenshot shows a "Select Configuration" dialog box with a close button (X) in the top right corner. It has two tabs: "Filters" (selected) and "Outputs". Under the "Filters" tab, there are several checkboxes: "Failed" (unchecked), "Executed" (checked), "Executor" (checked), "Scheduling date" (unchecked), "Execution date" (unchecked), and "Derivation protocol" (unchecked). Below these is a configuration section with two radio buttons: "False" (unchecked) and "True" (checked). At the bottom right, there are three buttons: "Clear", "Cancel", and "Confirm".

Figure 6.6. Entity Configuration Example.

Now, we may want to set two different filtering for one node and for the other, in order to apply a "AND" operator next, to see which records resulted from the two differently-filtered queries are in common. It is obviously just a trivial example:

the same results can be obtained by applying the two filters on the same entity node; of course, they can be done much more complex queries.

The click on the top-left icon within a block will trigger the opening of another dialog, dynamically populated based on the current node, where it is possible to choose some filtering options. In this example, in the first node we select only the "Transformations events" terminated with success (see Figure 6.5), while in the second only the ones actually executed (see Figure 6.6).

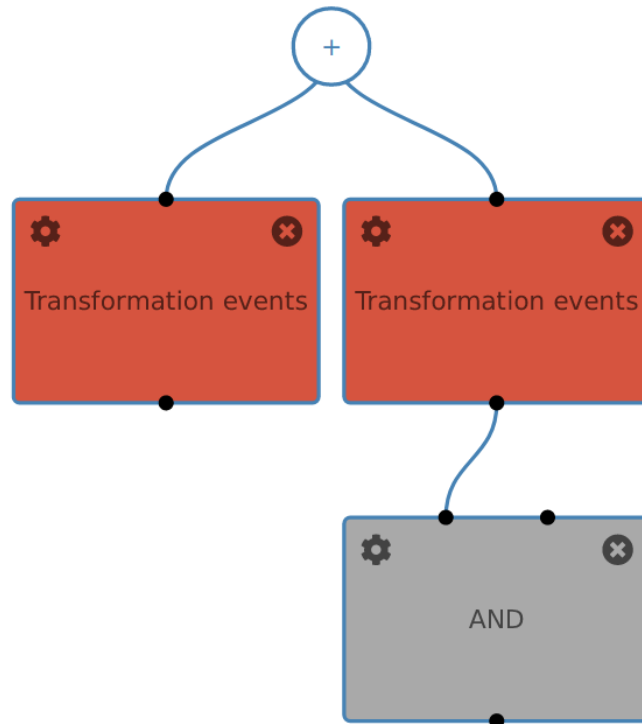


Figure 6.7. Query Graph Example.

Next, to add the operator we open the "Select Entity/Operator" dialog from the second node by clicking on the black terminal at the bottom of the node (the one devoted to trigger Function 3, see Figure 5.2), go to the last tab, select the "AND" and press "Confirm". The current situation is shown in Figure 6.7. At this point, we want to connect the other node to the operator; although we cannot manually draw an arc as in the previous interface, the operation is as simple as that: we open the "Select Entity/Operator" dialog and go to the last tab. What we will see it is in Figure 6.8: what happened is that, when populating the menu options, the application notice the presence of an Operator missing an input and, due to the fact that it is connectable to the current node, it shows the option.

Pressing "Confirm", what the application does is adding an arc between the current Entity node and the already present Operator node. The *ggen* library, then, manages the positions of the nodes, as described in Section 5.3.5, in a way that avoids interleaving arcs and centers the Operator with respect to its parents. In order to obtain the final query graph (in Figure 6.9) is sufficient to add the *end* node (selecting the "Connect Node to END" option that is possible to see in Figure 6.8).

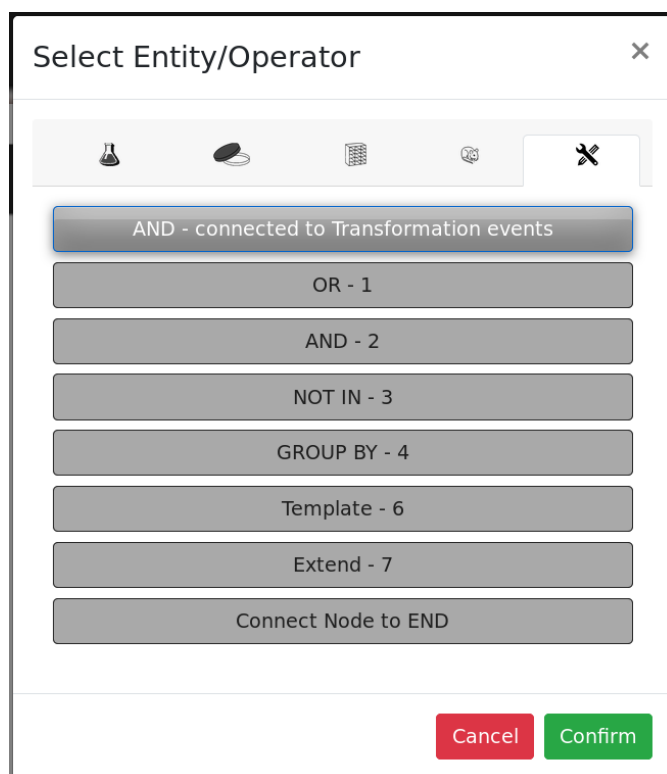


Figure 6.8. Select Entity/Operator Modal Menu (Operator tab).

In the end, we may decide to submit immediately the query by clicking the "Submit" in the left-side menu, or we may decide to save it as a Template or as a Translator. Submitting the query will produce the results in Figure 6.10.

Before submitting it, it is also possible to choose an already saved translator for the current query; to do that, it is sufficient to click on the *end* node: a menu dialog will be displayed with all the available options. In this particular case there is only one option entitled "Mother Aliquots" which will provide the result in Figure 6.11: as showed, each record can be expanded to display the information added by means of the Translator.

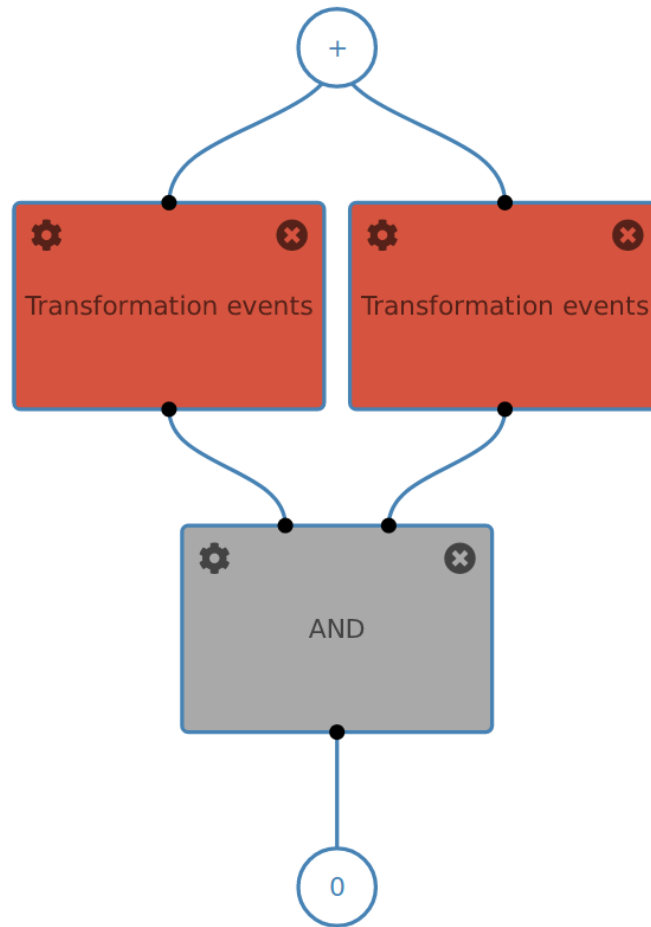


Figure 6.9. Query Graph Example.

### 6.3 Save a Template

Templates are predefined queries whose structures are stored in the MDDM database; a user may want to store the query he or she has built, aiming at running it later or making it available to other users.

The query must respect some constraints, such as: all nodes outputs has to be connected to another node, all paths have to converge to the *end* node. Those checks are performed when "Save as Template" button is clicked. Immediately after, the dialog in Figure 6.12 shows up. For each node the user can define some parameters, a name and a description.

When the user is done, he/she clicks "Confirm". Given that a title is required to identify the template, if the user has not previously defined the "Insert Title and Description" dialog (Figure 6.13) appears; once added a title it is enough to

## 6.4 – Save a Translator

Select	Delete timestamp	Executed	Executor	Exhausted	Failed	Initial date	Load quantity	Measurement executed	Volume outcome (ul)	Delete operator	Derivation protocol	Assigner	Kit	Execution date
<input type="checkbox"/>	2018-03-06 12:00:00+00:00	True	██████████	False	False	2018-12-17	25.0	True	100.0	None	DNA extraction (ReliaPrep)	██████████	XY0090099	2018-12-18
<input type="checkbox"/>	2019-03-06 12:00:00+00:00	True	██████████	False	False	2019-01-14	25.0	True	100.0	None	DNA extraction (ReliaPrep)	██████████	XY0090099	2019-01-16
<input type="checkbox"/>	2019-03-06 12:00:00+00:00	True	██████████	False	False	2019-01-25	25.0	True	100.0	None	DNA extraction (ReliaPrep)	██████████	XY0090099	2019-01-25
<input type="checkbox"/>	2018-03-06 12:00:00+00:00	True	██████████	False	False	2019-03-05	25.0	True	55.0	None	DNA extraction (Maxwell RSC)	██████████	xy00900390	2019-03-06
<input type="checkbox"/>	None	True	██████████	True	False	2012-05-29	35.0	True	49.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-05-29
<input type="checkbox"/>	None	True	██████████	False	False	2012-05-29	50.0	True	50.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-05-29
<input type="checkbox"/>	None	True	██████████	True	False	2012-06-01	51.0	True	50.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-06-01
<input type="checkbox"/>	None	True	██████████	True	False	2012-06-01	33.0	True	50.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-06-01
<input type="checkbox"/>	None	True	██████████	False	False	2012-06-01	50.0	True	50.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-06-01
<input type="checkbox"/>	None	True	██████████	True	False	2012-06-01	31.0	True	50.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-06-01
<input type="checkbox"/>	None	True	██████████	True	False	2012-05-29	25.0	True	40.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-05-29

Figure 6.10. Table of Result Example.

Select	Delete timestamp	Executed	Executor	Exhausted	Failed	Initial date	Load quantity	Measurement executed	Volume outcome (ul)	Delete operator	Derivation protocol	Assigner	Kit	Execution date
<input type="checkbox"/>	None	True	██████████	True	False	2012-05-29	35.0	True	49.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-05-29
<input type="checkbox"/>	None	True	██████████	True	False	2012-06-01	29.0	True	50.0	None	DNA extraction (Qiagen Mid)	██████████	0013343000000139315112	2012-06-01
<b>T:Mother aliquot</b>														
		Genealogy ID	Availability	Sampling date	Time used	Archive date	Aliquot type	ID						
		CRCM36PRH000000000RL0100	False	2012-04-11	1	2012-04-11	RNAlater	145						
<input type="checkbox"/>	None	True	██████████	True	False	2012-06-28	20.0	True	100.0	None	DNA extraction (Qiagen Mid)	██████████	0069504000000142327936	2012-07-02
<input type="checkbox"/>	None	True	██████████	False	False	2012-12-07	44.0	True	29.0	None	RNA extraction (RNeasy)	██████████	xy00900054	2012-12-07
<input type="checkbox"/>	None	True	██████████	True	False	2014-11-18	25.0	True	100.0	None	DNA extraction (ReliaPrep)	██████████	xy00900422	2014-11-27
<input type="checkbox"/>	None	True	██████████	True	False	2014-09-29	25.0	True	100.0	None	DNA extraction (ReliaPrep)	██████████	XY00900412	2014-10-03
<input type="checkbox"/>	None	True	██████████	True	False	2014-10-27	25.0	True	100.0	None	DNA extraction (ReliaPrep)	██████████	xy00900414	2014-10-28

Figure 6.11. Table of Result Example (with Translator).

press again the "Confirm" button to send the query to the Query Module backend.

Finally, the MDDM backend is in charge of storing the query tree in its database and changing the *templates* object accordingly. In this way all users, after loading the JSON object, are able to see the new Template and eventually add it to their queries.

## 6.4 Save a Translator

The Translator is a particular type of Template that can be optionally run for each row in a query result, to enrich it with additional information. In order to save the current query as a Translator is sufficient to click on the "Save As Translator" button". Also in this case a title is required; hence, if not present, the dialog in



Template Definition

Please define the template parameters

Block **8**: **Transformation events**

Input	Description
1	Name: <input type="text"/>
	Description: <input type="text"/>

Filter name	Option
Failed	Do not use ▼
Executed	Do not use ▼
Executor	Do not use ▼
Scheduling date	Do not use ▼
Execution date	Do not use ▼

Derivation

< Previous    Next >

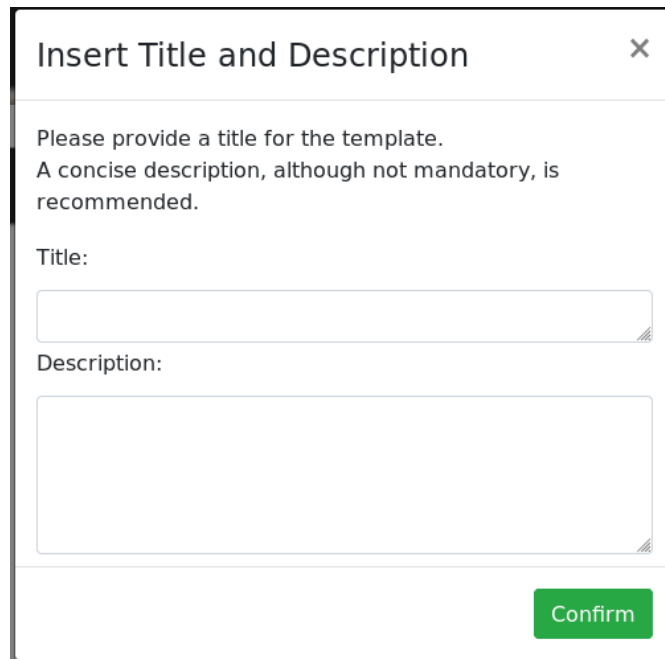
Cancel    Confirm

Figure 6.12. Template Definition Dialog.

Figure 6.13 it is shown and, after the title insertion, the "Save As Translator" has to be clicked again. Finally, as in the previous case, the query is sent to the Query Module backend, which stores it and propagates the changes to the *templates* object.

## 6.5 Reload last-submitted Query

One interesting feature provided by the LAS Query Module interface is the possibility to reload the last query submitted. From the user perspective it is sufficient to click on the "Reload Query" in the left-side menu; a pop up it is shown in case the current query has more than two nodes to let the user know that, if he/she confirms, the current query will be lost.



Insert Title and Description

Please provide a title for the template.  
A concise description, although not mandatory, is recommended.

Title:

Description:

Confirm

Figure 6.13. Insert Title and Description Dialog.

Next, basically, what the application does is reloading the page, inserting in the URL parameter of the query string an identifier of the last submitted query. The list of nodes of the query is retrieved from the backend and the graph is reconstructed from the root to the leaves.

To rebuild the graph, the application calls a recursive function for each child of the *start* node. The function works as follows: if the current node is *end* it calls *connectNodeToEnd()* (see Section 5.3.3), if it is an operator with one input or an entity, it calls *addNode()* (Section 5.3.3); if, instead, is an operator with more inputs or the special block with four inputs, it checks if that operator is already been inserted into the canvas, in this case it connect an arc to it with *addArc()* (Section 5.3.3), otherwise it calls *addNode()* as well. The function then calls itself over the children of the current node, until the end is reached (if child has already been processed by the function, of course, it will not be considered any more).



# Chapter 7

## Conclusion

As it had already been pointed out during this dissertation, the goal of this thesis was the design and development of a graph visualization library, able to build hierarchical tidy trees in web environments. In particular, the library needed to be suitable to be used in the LAS Query Module interface.

The LAS (Laboratory Assistant Suite) is a web platform for cancer genomic data management developed in-house at Cancer Institute IRCCS Candiolo. Its Query Module interface provides a easy-to-use tool for structuring, in a graphical form, queries to be submitted to the LAS distributed databases.

This thesis addressed some issues arisen during the every-day use of the module. Specifically, the interface needed some improvements in term of usability and responsiveness, in addition to an update related to the libraries used. In order to accomplish these goals, the *ggen* library had been implemented; in the LAS environment, it is able to support the creation and drawing of an acyclic direct graph representing the data flow of a query, but it can potentially be used in any other web context as a graph visualizer. The development of this thesis can be, ideally, divided into three phases.

The first preliminary phase was devoted to the choice of the best algorithm for computing the nodes' positions, producing a tree that should be compact and pleasant-to-see: a so called *tidy tree*. Thus, they have been analysed some algorithms for drawing tidy trees, in order to compare them and make an informed choice. In particular, they were analysed two algorithms for drawing binary trees, Whetherell and Shannon's [14] and Reingold and Tilford's [13], and two algorithms for drawing general trees of unbounded degree, Walker's [10] (with its enhancement [5]) and Van Der Ploeg's [11]. At the end of this preliminary phase, the VDP algorithm had been chosen, as it is the most recent and most general algorithm suitable for our needs.

During the second phase, the *ggen* library had been designed and implemented, with an as general and standard as possible approach. It had been built on top

of *D3.js* framework, using the VDP algorithm. In order to respect the query representation requirements, the algorithm had been extended for being able to draw ordered hierarchical trees containing also nodes with more than one input. The code related to the library and to the algorithm analysis can be found in a github repository [6].

The last phase consisted in the restructuration of the LAS Query Module interface, supported by the integration of the *ggen* library. The final results are a library that dynamically recomputes node positions, producing a nice and compact tree-shaped representation of a query, and a new supervised interface that provides a wizard to the users guiding them, step by step, in the creation of a consistent query tree.

## 7.1 Future Works

As it had been said, the library was built to be agnostic with respect to the LAS Query Module interface.

Although the library already provides some methods allowing applications to set some configuration parameters, such as node shapes and vertical and horizontal offsets between nodes, in term of configuration it can be furthermore improved. Specifically, the currently available shapes are only circle and rectangle, while the arcs can be only linear or curve; surely can be interesting to add new shapes, such as rombs or parallelograms, and new type of edges, as dashed arrows or dotted lines.

Moreover, another characteristic that can be interesting in other fields of application, could be the possibility to create cyclicity in the representation, for example given the possibility to connect a node to a previous one with a sort of backward arc.

Actually, in the purposes of LAS Query Module interface, the VDP algorithm had not been exploited at full potential; for example, it may eventually be used to build both unlayered and non-vertical trees.

From a graphical point of view the range of possible improvements is huge, and, of course, it should be made a choice based on the needs of the target application.

Therefore, potentially the *ggen* library can be improved to be suitable for many other kind of applications. One, again related to the LAS, is the representation of a operational flow, such as visualizing the dependencies among processes managed by a scheduler.

Nowadays, dynamic Data Visualization is a task of growing importance in many fields. In particular, in web-oriented contexts, where users expect to interact with fast and intuitive interfaces allowing them to extract useful knowledge in structured, organized way. We think that *ggen* may represent a nice and interesting

tool for supporting this kind of tasks, respecting modern standards.



# Appendix A

## VDP Algorithm Implementation

```
1
2 /* --- js implementation of A. J. van der Ploeg algorithm --- */
3 function VDPtreeLayout(t){
4     VDPfirstWalk(t);
5     VDPsecondWalk(t,0);
6 }
7
8 function VDPfirstWalk(t){
9     t.subtree.mod=0;
10    t.subtree.prelim=0;
11    if(t.children==null || t.children.length==0 ){ //leaf
12        VDPsetExtremes(t);
13        return;
14    }
15    VDPfirstWalk(t.children[0]);
16    //create sibling in contour minimal vertical coordinate and
17    //index list
18    var ih = VDPupdateIYL(VDPbottom(t.children[0].subtree.el), 0,
19    null);
20    for(var i=1; i< t.children.length; i++){
21        VDPfirstWalk(t.children[i]);
22        //stores lowest vertical coordinate while extreme nodes
23        //still point in current subtree
24        var minY = VDPbottom(t.children[i].subtree.er);
25        VDPseparate(t, i, ih);
26        ih = VDPupdateIYL(minY, i, ih);
27    }
28    VDPpositionRoot(t);
29    VDPsetExtremes(t);
30 }
31
32 function VDPsetExtremes(t){
33     if(t.children==null || t.children.length == 0){
```



```

31     t.subtree.el = t;
32     t.subtree.er = t;
33     t.subtree.msel = t.subtree.mser = 0;
34 }else{
35     t.subtree.el = t.children[0].subtree.el;
36     t.subtree.msel = t.children[0].subtree.msel;
37     t.subtree.er = t.children[t.children.length-1].subtree.er;
38     t.subtree.mser = t.children[t.children.length-1].subtree.
mser;
39     }
40 }
41
42 function VDPseparate(t, i, ih){
43     //right contour node of left sibling and its sum of modifier.
44     var sr = t.children[i-1];
45     var mssr = sr.subtree.mod;
46     //left contour node of current subtree and its modifier.
47     var cl = t.children[i];
48     var mscl = cl.subtree.mod;
49     while(sr!=null && cl!= null){
50         if(VDPbottom(sr) > ih.lowY )
51             ih = ih.nxt;
52         //how far to the left of the right side of r is the left
side of cl?
53         var dist = (mssr + sr.subtree.prelim + sr.subtree.w)-(mscl
+c.l.subtree.prelim);
54         if(dist > 0){
55             mscl += dist;
56             if(ih==null)
57                 console.log("VDP ERR");
58             else
59                 VDPmoveSubtree(t, i, ih.index, dist);
60         }
61         var sy = VDPbottom(sr), cy = VDPbottom(cl);
62         //advance highest nodes and sums of modifiers (coord syst
increases downward)
63         if(sy <= cy){
64             sr = VDPnextRightContour(sr);
65             if(sr!=null) mssr += sr.subtree.mod;
66         }
67         if(sy >= cy){
68             cl = VDPnextLeftContour(cl);
69             if(cl!=null) mscl += cl.subtree.mod;
70         }
71     }
72     //set threads and update extreme nodes
73     //in the first case the current subtree must be taller than
the left siblings
74     if(sr==null && cl!=null) VDPsetLeftThread(t,i,cl,mscl);

```

```

75     //in the second the left siblings must be taller than the
76     current subtree.
77     else if(sr!=null && cl==null) VDPsetRightThread(t,i,sr,mssr);
78 }
79 function VDPmoveSubtree(t, i, si, dist){
80     //Move subtree by changing mod.
81     t.children[i].subtree.mod += dist;
82     t.children[i].subtree.msel += dist;
83     t.children[i].subtree.mser += dist;
84     //console.log("movesubtree mod " + dist);
85     VDPdistributeExtra(t,i,si,dist);
86 }
87
88 function VDPnextLeftContour(t){
89     return t.children.length == 0 ? t.subtree.tl : t.children[0];
90 }
91
92 function VDPnextRightContour(t){
93     return t.children.length == 0 ? t.subtree.tr : t.children[t.
94     children.length-1];
95 }
96
97 function VDPbottom(t){
98     //console.log("vdp bottom "+(t.y+t.size.height));
99     return t.y + t.size.height;
100 }
101
102 function VDPsetLeftThread(t, i, cl, modsumcl){
103     var li = t.children[0].subtree.el;
104     li.subtree.tl = cl;
105     //console.log("thread left");
106     //console.log(li.subtree.tl);
107     //change mod so that the sum of modifier after following
108     thread is correct
109     var diff = (modsumcl - cl.subtree.mod)-t.children[0].subtree.
110     msel;
111     li.subtree.mod += diff;
112     //change preliminary x coordinate so that the node does not
113     move
114     li.subtree.prelim -= diff;
115     //update extreme node and its sum of modifiers
116     t.children[0].subtree.el = t.children[i].subtree.el;
117     t.children[0].subtree.msel = t.children[i].subtree.msel;
118 }
119
120 function VDPsetRightThread(t, i, sr, modsumsr){
121     var ri = t.children[i].subtree.er;
122     ri.subtree.tr = sr;

```

```

119     var diff = (modsumsr - sr.subtree.mod)-t.children[i].subtree.
120     mser;
121     ri.subtree.mod += diff;
122     ri.subtree.prelim -= diff;
123     t.children[i].subtree.er = t.children[i-1].subtree.er;
124     t.children[i].subtree.mser = t.children[i-1].subtree.mser;
125 }
126 function VDPpositionRoot(t){
127     //position root between children, taking into account their
128     //mod
129     t.subtree.prelim = (t.children[0].subtree.prelim +
130     t.children[0].subtree.mod +
131     t.children[t.children.length-1].subtree.
132     mod +
133     t.children[t.children.length-1].subtree.
134     prelim +
135     t.children[t.children.length-1].subtree.w
136     )/2 - t.subtree.w/2;
137     //console.log("positionROOT: "+t.subtree.prelim);
138     //console.log("width "+t.subtree.w);
139 }
140 function VDPsecondWalk(t, modsum){
141     modsum += t.subtree.mod; //+centermod
142     // set absolute (non-relative) horizontal coordinate
143     t.x = t.subtree.prelim + modsum + t.size.marginl;
144     VDPaddChildSpacing(t);
145     for(var i=0; i<t.children.length; i++){
146         VDPsecondWalk(t.children[i], modsum);
147     }
148 }
149 function VDPdistributeExtra(t, i, si, dist){
150     //are there intermediate children?
151     // distribute distances to children to be good looking
152     if( si!= i-1 ){
153         var nr = i - si;
154         t.children[si+1].subtree.shift+=dist/nr;
155         t.children[i].subtree.shift-=dist/nr;
156         t.children[i].subtree.change-=dist - dist/nr;
157     }
158 }
159 function VDPaddChildSpacing(t){
160     //process change and shift to add intermediate spacing to mod
161     var d = 0, modsumdelta = 0;
162     for(var i=0; i<t.children.length; i++){
163         d += t.children[i].subtree.shift;

```

```
164     modsumdelta += d + t.children[i].subtree.change;
165     t.children[i].subtree.mod += modsumdelta;
166 }
167
168 }
169
170 class IYL{
171     constructor(lowY, index, nxt){
172         this.lowY = lowY;
173         this.index = index;
174         this.nxt = nxt;
175     }
176 }
177
178 function VDPupdateIYL(minY, i, ih){
179     //remove siblings that are hidden by the new subtree
180     while(ih != null && minY >= ih.lowY)
181         ih = ih.nxt;
182     //prepend the new subtree
183     return new IYL(minY, i, ih);
184 }
```



# Appendix B

## Acronyms and Definitions

**Definition B.0.1.** HTML (HyperText Markup Language) is the standard markup language used to describe the content of Web pages.

**Definition B.0.2.** CSS (Cascading Style Sheets) is the most used language for describing the style of an HTML document.

**Definition B.0.3.** SVG (Scalable Vector Graphics) is a technology able to visualize vector graphical objects and to manage scalable images; it is a language derived from the XML.

**Definition B.0.4.** DOM (Document Object Model) is an object-oriented hierarchical representation of a web page, which can be modified with a scripting language such as JavaScript.

**Definition B.0.5.** W3C DOM is a standard document object model implemented in most modern browsers. It is maintained by the World Wide Web Consortium.

**Definition B.0.6.** JavaScript is a high-level, interpreted scripting language widely-supported.

**Definition B.0.7.** jQuery is a JavaScript library designed to simplify the creation of interactive web pages; it simplifies DOM hierarchy traversal and manipulation, event handling, CSS animation etc...

**Definition B.0.8.** UMD (Universal Module Definition) is a pattern used to allow a JavaScript module to be imported by a number of different module loaders, as AMD or CommonJS.

**Definition B.0.9.** CommonJS is a specification for JavaScript module loaders aimed at defining conventions on module ecosystem for JavaScript outside of the web browser (e.g. when the language is use on serverside).

**Definition B.0.10.** AMD (Asynchronous Module Definition) is a specification for JavaScript module loaders able to load different module asynchronously. RequireJS is its most popular implementation.

**Definition B.0.11.** ES6 (ECMAScript 2015) is the sixth edition of the standard specification for JavaScript language.

**Definition B.0.12.** IIFE (Immediately Invoked Function Expression) is a JavaScript abstraction, well-described in Ben Alman’s article [1]; it defines a function expression immediately executed.

**Definition B.0.13.** Bézier Curve is a parametric curve, based on Bernstein polynomial, used in computer graphics and related fields.

**Definition B.0.14.** JSON (JavaScript Object Notation) is a human-readable data-interchange format designed to be language-independent and easy for machines to parse and generate.

**Definition B.0.15.** Bootstrap is an open source toolkit for developing with HTML, CSS, and JS. It provides some useful set of primitives and abstractions, such as a responsive grid system, extensive prebuilt components and plugins.

# Bibliography

- [1] B. Alman. Immediately invoked function expression. URL <http://benalman.com/news/2010/11/immediately-invoked-function-expression/>.
- [2] E. Baralis, A. Bertotti, A. Fiori, and A. Grand. LAS: a software platform to support oncological data management. *Journal of medical systems*, 36(1): 81–90, 2012.
- [3] M. Bostock. D3 - Data-Driven Documents API reference. URL <https://github.com/d3/d3/blob/master/API.md>.
- [4] M. Bostock, V. Ogievetsky, and J. Heer. Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.
- [5] M. Jünger C. Buchheim and S. Leipert. "improving walker's algorithm to run in linear time". *Lecture Notes in Computer Science*, pages 344–353, 2002.
- [6] E. Dari. Graph generator library. URL <https://github.com/emil-d/ggen>.
- [7] A. Fiori, A. Grand, P. Alberto, E. Geda, F. G. Brundu, D. Schioppa, and A. Bertotti. A case study of a laboratory information system developed at the institute for cancer research at candiolo. *Laboratory Management Information Systems: Current Requirements and Future Perspectives*, pages 252–279, 2014.
- [8] A. Fiori, A. Grand, E. Geda, D. Schioppa, F. G. Brundu, A. Mignone, and A. Bertotti. LAS: A bio-clinical integrated laboratory information system for translational data management. *Emerging Developments and Practices in Oncology*, pages 56–93, 2018.
- [9] A. Grand, E. Geda, A. Mignone, A. Bertotti, and A. Fiori. One tool to find them all: a case of data integration and querying in a distributed lims platform. *Database*, page 1–11, 2019.
- [10] J. Q. Walker II. "a node-positioning algorithm for general trees". *Software - Practice and Experience*, 20(7):685–705, 1990.



## BIBLIOGRAPHY

---

- [11] A. Van Der Ploeg. "drawing non-layered tidy trees in linear time". *Software - Practice and Experience*, 44(12):1467–1484, 2014.
- [12] T. Preston-Werner. Semantic versioning. URL <https://semver.org/>.
- [13] E. M. Reingold and J. S. Tilford. "tidier drawings of trees". *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.
- [14] C. Wetherell and A. Shannon. "tidy drawings of trees". *IEEE Transactions on Software Engineering*, SE-5:514–520, 1979.