Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation:

Computer Science

Spring semester, 2019

Open

Author:
Fabian Legland Boe

……………………………………………………
(signature of author)

Programme coordinator: Morten Mossige

Supervisor(s): Morten Mossige

Title of master's thesis:
Real-time graph visualization in a single-page application

Credits: 30

Keywords:
Graph theory, Graphviz, Angular,
TypeScript, Single-page application,
WebSocket, RPC

Number of pages: 38

+ supplemental material/other: 1

Stavanger, 14/06/2019

Title page for Master's Thesis
Faculty of Science and Technology

Real-time graph visualization in a single-page application

Fabian Legland Boe

June 2019

Acknowledgements

I would like to thanks my supervisor, Morten Mossige, for his swift and detailed guidance. I would also like to thank Tore Fuglestad from ABB for detailed descriptions regarding communication on this project. Their thoughts and feedback have been critical for this project.

Abstract

In this project a single-page application for visualizing factory data was created. The application uses Graphviz to draw graphs. It keeps the graph updated by fetching data from a server using WebSockets and RPC. The graph is easily traversed on mobile and desktop platforms with intuitive controls. The application compiles to regular HTML, CSS and JavaScript for a platform-independent solution.

The paper explains the inner workings of the graph drawing algorithm. It also presents an implementation of the single-page application and a presentation of the tools used. A performance evaluation is also presented.

Contents

Chapter 1

Introduction

The purpose of this project is to replace the current factory visualization software for ABB. The system monitors the status of sensors and alarms, represented as a directed graph. The information is fetched from a server and drawn as a static image. The solution is outdated and has several issues. It contains redundant intermediate steps, making it badly optimized. Since the output is displayed as a still image, the information is quickly outdated. In order to receive updated information, the whole system needs to be refreshed. This creates a lot of overhead and requires manual intervention each time. The solution also has restricted platform support, working solely on Windows systems.

This project intends to solve the aforementioned issues. It is made as an Angular application, which supports all modern platforms by using HTML, CSS and JavaScript. Angular has a modular workflow, making it easier to add and modify parts of solution to accommodate future changes. The project uses WebSocket communication which is fast and reliant, even with high traffic. WebSockets are especially well suited for recurring traffic on the same connection. The graph is drawn once using Graphviz, the same software used in the current solution. Next, the project subscribes to data changes from the Server. As new updated data is received, the relevant parts of the graph is updated.

The resulting product is a fast, modular, real-time and platform-independent solution.

Outline:

Chapter 2 Graph Theory: Provides an in-depth look at how Graphviz draws directed graphs. The chapter is divided into an introduction and the four passes for the algorithm used by Graphviz. The first pass (2.2) assigns the graph nodes into different hierarchical layers. The second pass (2.3) sets the ordering withing those layers. The third pass (2.4) finds the actual Cartesian coordinates of each node. The final pass (2.5) draws the edges between nodes.

Chapter 3 Application framework and communication protocols: Introduces the theory behind the other tools used in the project. This includes a section on the Angular Framework (3.1), explaining the different parts of an Angular single-page application. This chapter also describes the theory behind the communications used in this project. That includes WebSockets (3.2) and RPC (3.3) protocols.

Chapter 4 Methods and Design: Provides an overview of the different phases of the application. The introduction phase (4.2) draws the graph and subscribes to graph data changes. The main phase (4.3) handles changes received from the server, and updates the graph with these values. The rest of the chapter explains the implementation of server data formatting (4.4) and communication (4.5).

Chapter 5 Results and Discussion: Provides a look at the application from a users perspective (5.1), as well as different performance measurements (5.2). The last section discusses the impact of the results and other factors (5.3).

Chapter 6 Conclusion: Concludes the project and suggests further work.

Chapter 2

Graph Theory

2.1 Introduction

This project uses a JavaScript wrapper for the GraphViz software in order to draw graphs. GraphViz uses several tools that are specialized for different graph layouts. *Circo* is used for circular graphs, *neato* is preferred for smaller, unknown networks[2]. For hierarchical directed graphs, GraphViz uses *dot* language. The data for this project fits this description. Chapter 2 explains each step of the *dot* algorithm in depth. This chapter is based on the paper *A Technique for Drawing Directed Graphs* [7] which explains the inner workings of the *dot* algorithm in detail.

The algorithm tries to create graphs that are similar to handmade ones. Handmade graphs usually have an overall direction and convey the information clearly. The *dot* paper defines several aesthetic principles for drawing graphs that achieve this. These are defined as follows:

- *"**A1:** Expose hierarchical structure in the graph. In particular, aim edges in the same general direction if possible. This aids finding directed paths and highlights source and sink nodes.*
- *"**A2:** Avoid visual anomalies that do not convey information about the underlying graph. For example, avoid edge crossings and sharp bends.*
- *"**A3:** Keep edges short. This makes it easier to find related nodes and contributes to A2.*
- *"**A4:** Favor symmetry and balance. This aesthetic has a secondary role in a few places in our algorithm."*

The algorithm tries to follow these principles. Unfortunately it is impossible to consistently uphold all principles. It therefore tries to do best in the most common situations.

The algorithm separates the problem into four separate passes, each handling their own specific problem. The output of one pass is the input for the next pass. The passes are as follows:

1. Optimal rank assignment (section 2.2)
2. Ordering nodes within ranks (section 2.3)

3. Optimal position of nodes (section 2.4)
4. Drawing edges as splines (section 2.5)

First pass sorts all nodes into ranks (see figure 2.1). Ranks are non-negative integers correlating to the hierarchical position of nodes within the graph. In a top-down flowing graph, ranks corresponds to the y direction. This means ranks go from source nodes at the top (rank 0) to sink nodes at the bottom (max rank). This paper assumes a top-down flow of graphs, and will refer to directions this way.

Second pass computes the order of nodes within each rank (see figure 2.1). This helps prevent edge crossings, following principle A2.

Third pass computes the actual position of the nodes. Output from the previous passes are used to calculate this properly. Node positions that results in shorter edges are prioritized which follows principle A3.

Forth pass draws edges between nodes using splines. It tries to follow principle A2 and A4 by avoid edge crossing, sharp bends and keep edges in the same direction parallel.

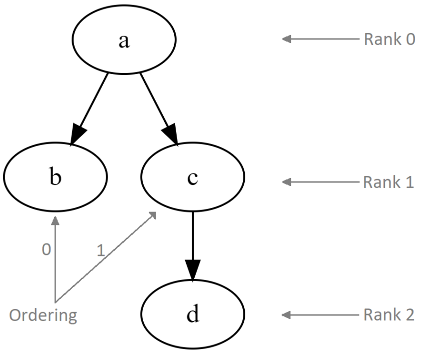


Figure 2.1: Example graph showing rank assignment and ordering within ranks

Graphs contain definitions and attributes as shown below:

$G(V, E)$	The graph. Defined with a set of vertices and a set of edges.
$e(v, w)$	An edge $e \in E$ going from v to w. v and w are called the tail and head node, respectively.
$v(x, y)$	Center point of the bounding box of $v \in V$.
$\delta(e)$	Minimum edge length, usually 1.
$\lambda(v)$	Rank of vector v.
$l(e)$	length of $e(v, w)$. Defined as the $\lvert\lambda(w) - \lambda(v)\rvert$, subject to $l(e) \leq \delta(e)$.
$xsize(v)$, $ysize(v)$	Size of the bounding box defining the borders of node v in x and y direction.
$nodesep(G)$	Minimum horizontal spacing between nodes on the same rank.
$ranksep(G)$	Minimum vertical spacing between nodes on the different ranks.
$\omega(e)$	Weight of the edge e. It defines how important it is to keep edge e short and straight.
S	Contains sets of nodes. $S = S_{min}, S_{max}, S_0, ..., S_k$. Nodes in the same set are kept at the same rank.
Virtual nodes and edges	Temporary entities assisting the calculations in the passes without being present in the final drawing.

The *dot* algorithm receives a string as an input. That string contains $G(V, E)$, $e(v, w)$, $\delta(e)$, $nodesep(G)$, $ranksep(G)$, $w(e)$ and S. The rest of the data is computed in the passes.

2.2 Rank assignment

The first pass computes a rank $\lambda(v)$ for all $v \in V$. Ranks are assigned by breaking cycles in the graph (2.2.1), then using a network simplex algorithm to find the ranks (2.2.2).

Each set of nodes in S are treated as a single node. This makes sure all nodes in a set are given the same rank. Nodes in set S_{min} and S_{max} are assigned rank 0 and max rank, respectively. Sets do not exclusively hold a rank as other nodes may receive that same rank as well. This pass treats all edges with the same head and tail node as a single edge, assigning the merged edge the sum of all the edge weights. This pass also ignores loops[1] as well as leaf nodes not in sets.

---

[1] Edges where the tail and head node are the same. Also called self-edges.

2.2.1 Breaking Cycles

Before assigning ranks, the graph must become acyclic by reversing certain edges. A spanning tree is created using Depth-first search. This returns a list of tree edges and non-tree edges. All non-tree edges are then classified into three types[4]: Forward, back and cross edges.

By looping through all non-trivial strongly connected components we can find the back edge that forms the most cycles. This edge is then reversed. This is repeated until there are no more components. This process reverses all the necessary edges while affecting the end result as little as possible. Note that the edges are only reversed for this pass. Algorithm 1 illustrates how the process works.

Algorithm 1 break_cycles

```
tree_edges, nontree_edges = dfs();
classify_edges(nontree_edges);
while components = find_strongly_connected_components() do
    for c in components do
        e = most_cyclic_edge(c.back_edges);
        e.reverse();
    end for
end while
```

This algorithm does not guarantee that S_{min} and S_{max} are placed at the lowest and highest rank, respectively. To enforce this, all edges going to S_{min} are reversed and all nodes lacking an in-edge gets a virtual one from S_{min}. This makes sure that S_{min} is the oldest ancestor which will give it the lowest rank when ranks are assigned. The opposite is true for S_{max}. All edges going from S_{max} are reversed and all leaf/sink nodes are given a virtual edge going to S_{max}. This makes sure that $\lambda(S_{min}) \leq \lambda(v) \leq \lambda(S_{max})$, for any node v not in S_{min} or S_{max}.

2.2.2 Network Simplex

After reversing edges, the graph is now ready for rank assignment. An optimal rank is defined as having a minimum weighted edge length sum, defined by this equation:

$$\begin{gathered} min \sum_{e \in E} \omega(e) l(e) \\ Subject\ to:\ l(e) \geq \delta(e)\ \forall\ e \in E \end{gathered} \tag{2.1}$$

Equation 2.1 says to minimize the sum of all edge values where the value is defined as the edge length times the edge weight. This follows principle A3 in terms of keeping edges short, while also prioritizes weighted edges. Note that no edge can be shorter than the *minimum edge length* δ.

Equation 2.1, with it's minimization and restriction is an example of a integer program. This can be solved as a min-cost flow problem or solving it as a linear program [8]. However, the *dot* algorithm solves it using the network simplex algorithm.

The network simplex method involves two parts, the first part is finding an initial feasible tree. The second part involves iteratively assigning new ranks until we reach an optimal ranking. A tree is feasible when $l(e) = \delta(e)\ \forall\ e \in$ E. These edges are referred to as tight edges. A tight edge can be more precisely defined as an edge with no slack where slack is defined as $l(e) - \delta(e)$.

To make a feasible tree. first assign each node an initial rank, giving higher rank to nodes with more in-edges. Then create a maximal tree containing only tight edges. Then find the closest non-tree edge, adjust the whole tree until the edge is tight, then create a new maximal tree. Repeat until all nodes are included. The result is a tight spanning tree. The final step involves computing cut values for all edges which is explained after algorithm 2.

Algorithm 2 feasible_tree

```
1: init_rank();
2: while tight_tree() < |V| do
3:     e = a non-tree edge incident on the tree with a minimal amount of slack;
4:     delta = slack(e);
5:     if incident node is e.head then delta = -delta;
6:     for v in Tree do v.rank = v.rank + delta;
7: end while
8: init_cutvalues();
```

Algorithm 2 explained:

init_rank:	Nodes are sorted in ascending order based on the number of in-edges. Every collection of nodes with an equal amount of in-edges are given the next available rank. For example, if the collection of in-edges are as follows: [0, 0, 2, 3, 3, 5] then ranking will be set like so: [0, 0, 1, 2, 2, 3].
tight_tree:	Creates a maximal tree with only tight edges, then returns the number of nodes.
2-7:	Finds an edge not present in the tight tree with minimal amount of slack. All node ranks in the tight tree are then adjusted to make this new edge tight. Since the whole tree is adjusted, no tightness is lost. The node is added to the tight tree in the next iteration. When the tight tree covers all nodes in the graph, the loop exits.
init_cutvalues:	Computes cut values for all edges.

Computing cut values are done individually for every edge in the spanning tree. When calculating the cut value of an edge, the edge is temporarily removed. This creates two strongly connected components A and B. If the cut edge is directed from A to B, we refer to A as the tail component and B as the head component. The original graph has potentially several edges between the components, in either direction. These non-tree edges and the cut edge, are used in computing the cut value. The cut value is the summed edge weight of all these connecting edges, with a negative signed value for edges going to the tail component.

A negative cut value indicates that the weighted edge length sum (equation 2.1) can be reduced by exchanging the cut edge for a non-tree edge. This means the ranking in the feasible tree is not an optimal solution. Replacing an edge that has a negative cut value improves it's cut value and the solution is closer to optimal. When no more negative cut values can be found, the solution is optimal and correct rank values can be calculated.

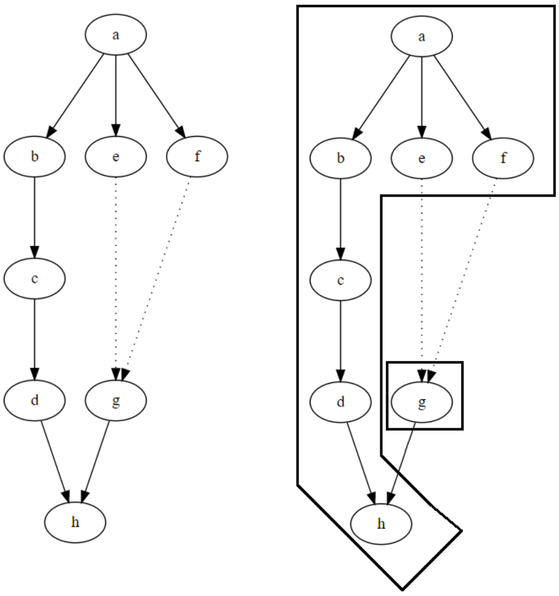


Figure 2.2: Example of a cut value calculation

Figure 2.2 illustrates the cut calculation. The left image shows a spanning tree with only tight edges. Dotted edges are non-tree nodes. Right image shows the strongly connected components if the edge (g, h) is cut. The tail component contains node g and the head component contains all the other nodes. The edges connecting the components are the cut edge, going from tail to head component, as well as edges (e, g) and (f, g), going from head to tail component. The cut value for $(g, h) = \omega(g, h) - \omega(e, g) - \omega(f, g)$. Since (e, g) and (f, g) are going from the head to the tail component, they get a negative value. If all edge weights were 1, the cut value of edge (g, h) would therefore be -1. The edge (g, h) will be exchange for either (e, g) or (f, g).

After creating a feasible tree, we replace negative cut edges until the solution is optimal. This process is shown in algorithm 3.

Algorithm 3 rank

```
feasible_tree();
while (e = leave_edge()) $\neq$ nil do
    f = enter_edge(e);
    exchange(e,f);
end while
normalize();
balance();
```

Algorithm 3 explained:

feasible_tree:	See algorithm 2.
leave_edge:	Finds an edge with negative cut value. If no edge is found, the spanning tree has an optimal structure.
enter_edge:	finds a non-tree edge to replace the cut edge with. The edge from the head to the tail component with the least amount of slack is selected. This function is similar to line 3 in algorithm 2.
exchange:	Exchanges the cut edge with the newly found replacement. The spanning tree ranks are then updated and new cut values are computed.
normalize:	Normalizes the ranks so that zero is the lowest value.
balance:	Some nodes are not very reliant on their given rank. They may have several ranks that are feasible, or their number of in-edges and out-edges are equal. These nodes can be moved freely while still keeping an optimal solution. The nodes are moved to other less crowded ranks if available (with a greedy approach). This helps avoid clustering of nodes and keep a good aspect ratio which follows principle A4. The result is an optimal ranking for the input graph.

2.3 Ordering nodes within ranks

This pass defines the order of nodes within the same rank. Proper ordering is important to avoid edge crossing, which follows principle A2. Ordering is done by first creating an initial order within each rank, then adjusting the ordering of each rank. The ranks are iterated in ascending order, giving each node a weight based on the position of it's incident[2] nodes in the previous rank. The nodes are then given a new position based on the sorting of the weights.

Before ordering nodes, some preparatory steps are taken. Self-edges are ignored and multi-edges are merged into one edge, similar to the last pass. When ranks were assigned, some nodes were moved while balancing. This potentially created non-tight edges. These edges are replaced with chains of virtual nodes and edges. These virtual edges are one rank long so that the graph is now tight.

---

[2] Nodes that are connected directly to the given node by an edge.

The nodes are given weights based on the median function[5]. The function computes the weight of a node v, using the list of incident nodes P. The list contains the index position of incident nodes on the previous rank. Note that since iteration is done in ascending order, nodes in P are always the parent nodes of v. The weighted value is computed as the median value of the indices in P. If the list contains an even number of values, their will be two median values in the list. In that case, either the left or right value is chosen consistently throughout the calculations.

This pass uses an extended version of the median function. When two median values are present, the algorithm is biased towards the side where incident nodes are more densely placed. To further improve positions, a separate check for crossings is computed after the median function has computed all node positions.

Algorithm 4 ordering

```
1: order = init_order();
2: best = order;
3: for i = 0 to Max_iterations do
4:     wmedian(order,i);
5:     transpose(order);
6:     if crossing(order) < crossing(best) then best = order;
7: end for
8: return best;
```

Algorithm 4 explained:

init_order:	traverses the tree using DFS or BFS. As nodes are traversed, they are added to the next position of their rank from left to right. The result contains no crossings.
Max_Iterations:	Integer value, default is 24.
3-8:	A new solution is calculated using the weighted median function. transpose further transposes the position of nodes if possible. if the solution has fewer crossings than the current best solution, it replaces that solution. After looping, the best solution is returned.

Algorithm 5 wmedian

Input: order, iter

```
if iter mod 2 == 0 then
    for r = 1 to Max_rank do
        for v in order[r] do median[v] = median_value(v,r-1);
        sort(order[r],median);
    end for
end if
```

Algorithm 6 median_value

Input: v, adj_rank

```
P = adj_position(v,adj_rank);
m = |P|/2;
if |P| = 0 then
    return -1.0;
elseif |P| mod 2 == 1 then
    return P[m];
elseif |P| = 2 then
    return (P[0] + P[1])/2;
else
    left = P[m-1] - P[0];
    right = P[|P|-1] - P[m];
    return (P[m-1]*right + P[m]*left)/(left+right);
end if
```

Algorithm 5 and 6 explained:

wmedian:	Ascends iteratively trough the ranks. At each rank, every node is assigned a new weight based on the incident nodes in the previous rank. The nodes at each rank is then sorted based on their weights. Nodes without incident nodes in the previous rank stays at their old position.
median_value:	Returns the median value if the list of incident node position has an odd length. If not, it interpolates the two median values, favouring the side more dense with nodes. If no incident nodes are found, -1 is returned as the median value. The sort function in wmedian handles these values as mentioned in it's description.

```
Algorithm 7 transpose
Input: rank
 1: improved = true;
 2: while improved do
 3:     improved = false;
 4:     for r = 0 to Max_rank do
 5:         for i = 0 to |rank[r]|-2 do
 6:             v = rank[r][i];
 7:             w = rank[r][i+1];
 8:             if crossing(v,w) > crossing(w,v) then
 9:                 improved = true;
10:                 exchange(rank[r][i],rank[r][i+1]);
11:             end if
12:         end for
13:     end for
14: end while
```

Algorithm 7 explained:

2-14:	The while loop runs as long as there are improvements. If a iteration does not result in an improvement, the function terminates. The for loops makes sure the function runs through every possible node. When at a node, the node and it's next neighbour is swapped if this results in less crossings.

Edges between nodes at the same rank, referred to as flat edges, should point perpendicular to the direction of ranks in regards to principle A1. If there are flat edges in the graph, the initial order, sort and transpose algorithms try to keep nodes next to each other if they are incident to a flat edge.

2.4 Positioning nodes

This pass finds the coordinates of each node, including virtual nodes from the previous step. The X and Y coordinates are computed separately. X coordinates are calculated based on order within ranks. Y coordinates are calculated based on rank and $ranksep(G)$. Nodes on the same rank are given the same Y coordinate. This process is so simple, the Y coordinate is not explained in this pass.

An optimal node placement upholds the following equation:

$$\begin{aligned} &min \sum_{e(v,w)\,\in\,E} \Omega(e)\omega(e)|x_w - x_v| \\ &Subject\ to:\ x_b - x_a \geq \rho(a,b) \end{aligned} \tag{2.2}$$

x_a and x_b are the X coordinates of two nodes on the same rank. The nodes are next to each other with a on the left side of b. $\rho(a, b)$ defines the minimum space between the center of two nodes based on the horizontal size of the nodes as well as $nodesep(G)$.

Edges between nodes on the same rank are trivial to keep horizontal. the edge is usually a simple straight line. Therefore virtual nodes between ranks are more important to keep straight and short. Edges are therefore separated into three types: between real nodes, from real node to virtual node, between virtual nodes. $\Omega(e)$ returns different values depending on the three types. Values defaults to 1, 2 and 8, respectively. This makes sure virtual edges are prioritized when minimizing.

The main algorithm for this pass is similar to the previous pass in that it tries to find an initial solution, then iteratively improves it:

Algorithm 8 xcoordinate

```
xcoord = init_xcoord();
xbest = xcoord;
for i = 0 to Max_iterations do
    medianpos(i,xcoord);
    minedge(i,xcoord);
    minnode(i,xcoord);
    minpath(i,xcoord);
    packcut(i,xcoord);
    if xlength(xcoord) < xlength(xbest) then xbest = xcoord;
end for
return xbest;
```

Algorithm 8 explained:

init_xcoord:	Initializes all node positions as tightly packed to the left as possible while still maintaining $x_b - x_a \geq \rho(a, b)$.
medianpos:	Assigns node coordinates based on the median position of it's child nodes. This works by queueing all nodes on a given rank. The queue order is decided by prioritizing nodes. Nodes with a higher weighted sum of out-edges get higher priority in the queue. Nodes are then dequeued in prioritized order. Dequeued nodes are placed on the median of it's child node positions. If there are two median positions, the average is taken. This creates symmetry and follows principle A4. if the median position is unavailable due to $nodesep(G)$ or higher priority nodes already placed, the position is adjusted but stays as close to the median as possible.
minedge:	Functions the same way as medianpos, but only for edges between two non-virtual nodes [6].
minnode:	Moves nodes to the median of all it's parent and children.
minpath:	Straightens virtual node chains, making them more vertically aligned.
packcut:	Tries to compact ranks by removing unnecessary spacing. The precision is improved by using the network simplex algorithm in chapter 2.2.2 with X coordinates instead of ranks. This method uses an auxiliary graph which allows for offsetting the endpoint of edges in the x direction.
xlength:	Uses the minimize function (equation 2.2) to check whether the new solution is an improvement.

2.5 Drawing splines

This step draws the edges between nodes. The edges are drawn using splines from piecewise bezier curves. Splines create smooth curves with follows principle A2. The algorithm is split into two parts. The first part finds the greatest polynomial area where the spline can be drawn without colliding with other nodes and splines. The second part draws the spline. Virtual nodes are then resized to fit the splines bounding boxes.

The area is defined using a set of connected boxes $B_0, ..., B_i$. The boxes are defined by two lines in the X direction and two lines in the Y direction, meaning the box edges line up with the coordinate axes. All boxes have at least one edge overlapping with another box so that there are no gaps between the boxes. The first box has a point q and the last box has a point r. These points define the start and end positions of the spline, respectively. Optionally, one can define θ_q and θ_r as the spline slope at point q and r.

Control points define the spline's path. The positioning of these points are critical for making the spline move from q to r without leaving the area defined by the boxes. Boxes $BB_0, ..., BB_i$ are also created, defining the smallest boxes still containing the splines path. These boxes are referred to as the splines bounding boxes. The spline bounding boxes are used when calculating the polynomial area of new splines to avoid collisions, if possible.

2.5.1 Finding the polynomial area

Edges are classified into three types: Edges between ranks, flat edges and self-edges.

The most frequent type is the edges between ranks. When defining the area for these splines, virtual nodes are ignored to give the spline as much space as possible. If the area finding algorithm encounters a long vertical section, it stops and draws a straight line for that section. This especially looks better when several edges are almost vertical as this keeps them parallel by simply offsetting one of the splines. Multi-edges are given areas that keeps them identical but slightly offset in the X direction.

Flat edges are treated similar to edges between ranks. Multi-edges are treated the same, except the offset is in the Y direction since these edges are mostly horizontal.

Lastly there's the self-edges. If an input or output position is specified, a polynomial area is generated and a spline is placed clockwise or counter-clockwise depending on the positions. If no position is specified, a simple loop from two splines is used. Mutli-edges are drawn with a slight radial offset.

2.5.2 Drawing splines

To draw a spline, a piecewise linear path is created. This is simply a collection of straight lines connected to each other, defining a path. The points on this path is used to make a piecewise bezier curve. A bezier curve is a curved path made from linearly interpolating between several points, called control points. Finally, the BB_i boxes are calculated for the spline. This makes sure the next spline tries to avoid colliding with this one. To reduce collisions, shorter splines are drawn first. The algorithm for drawing splines are as follows:

Algorithm 9 compute_splines

Input: B_array, q, theta_q, use_theta_q, r, theta_r, use_theta_r

```
compute_L_array (B_array);
compute_p_array (B_array, L_array, q, r);
if use_theta_q then vector_q = anglevector(theta_q)
else vector_q = zero_vector;
if use_theta_r then vector_r = anglevector(theta_r)
else vector_r = zero_vector;
compute_s_array (B_array, L_array, p_array, vector_q, vector_r);
compute_bboxes ();
```

Algorithm 9 explained:

compute_L_array:	All boxes in the B_array have a side shared with another box. This function finds those lines.
compute_p_array:	Creates an array of the points for the piecewise linear path (see algorithm 10). Uses the divide-and-conquer approach.
3-6:	If either use_theta_q or use_theta_r is true, angle the vector from the respective point by theta_q or theta_r.
compute_s_array:	Generates an array of spline control points (see algorithm 11). Uses the divide-and-conquer approach.
compute_bboxes:	Generates the BB_i array defining boundary boxes of the space used by the spline.

Algorithm 10 compute_p_array

Input: B_array, L_array, q, r

```
1: if line_fits (B_array, L_array, q, r) then return;
2: p = compute_linesplit (B_array, L_array);
3: addto_p_array (p);
4: compute_p_array (B_array1, L_array1, q, p);
5: compute_p_array (B_array2, L_array2, p, r);
```

Algorithm 10 explained:

line_fits:	Checks if a line from q to r stays within the polynomial area and only intersects with the L segments of the boxes. Detection is done by sampling along the line.
compute_linesplit:	Finds the L segment furthest away from the line and splits the L array and B array at that segment into two arrays each. The function returns a point p. Point p is the end point of the segmented line closest to the q-r line.
3-5:	Point p is then added to the list of points and a recursive call for each split part is called.

Algorithm 11 compute_s_array

Input: B_array, L_array, p_array, vector_q, vector_r

```
spline = generate_spline (p_array, vector_q, vector_r);
if size (p_array) == 2 then
    while spline_fits (spline, B_array, L_array) == false do
        straighten_spline (spline);
elseif spline_fits (spline, B_array, L_array) == false then
    count = 0;
    ospline = spline;
    while (fits == false) and (count $\leq$ max_iterations);
        spline = refine_spline (p_array, ospline, mode (count, max_iterations));
        fits = spline_fits (spline, B_array, L_array);
        count = count + 1;
    endwhile
    if fits == false then
        p = compute_splinesplit (spline, p_array);
        compute_s_array (B_array1, L_array1, p_array1, vector_q, vector_p);
        compute_s_array (B_array2, L_array2, p_array2, reverse (vector_p), vector_r);
        return;
    endif;
endif;
addto_s_array (spline);
```

Algorithm 11 explained:

generate_spline:	generates a spline going from q to r using the p array.
spline_fits:	Checks if the spline is within the polynomial area. Similar to line_fits.
straighten_spline:	reduces spline curvines by adjusting the control points.
refine_spline:	Similar to straighten_spline but will also try to increase curvature. Mode decides whether to increase or decrease curvature.
compute_splinesplit:	similar to compute_linesplit.
compute_s_array:	recursively calls the procedure again with the two split parts from compute_splinesplit.

The final pass is now complete and a graph is now displayed. For an overview on what the input data for the algorithm looks like, see section 4.4.

Chapter 3

Application framework and communication protocols

3.1 Angular web framework

Angular is a TypeScript-based web application framework managed by Google. All Angular applications are Single Page Applications (SPA). This means any implied redirecting between pages is in reality just dynamic switching of content. What can be perceived as a redirect is referred to as changing views in Angular. Since no page reloading takes place, load times are faster and usage is more reminiscent of desktop applications. Each view is controlled by an Angular component. Components are the middleman between views and the application logic. Angular services are responsible for fetching data and are usually narrow, purpose specific scripts used by components.

This section explains these terms in more detail. The statements in this section is based on the Angular documentation [1].

3.1.1 Components

A component functions as a controller for a view. A view is defined as a screen element that can be modified by angular.

A component consists of a HTML, CSS and typescript file. The files define the visibility and function of the component. Components also includes a spec.ts file. This file is used for running unit tests on the typescript file.

The HTML template (as well as the CSS file) defines the look of the component, referred to as a view. In addition to normal markup functionality, the template can contain angular specific code. This code allows for if conditions, loops and variables. The code supports two-way communication between the template and the typescript document using data-binding. This allows both the view and the controller to share and manipulate shared data.

The typescript document defines a class containing the data and logic used by the component. This file replaces most of of the functionality of a regular JavaScript document. It contains decorators which provides metadata on how it should be used.

Components are initialized the same way as HTML elements. For instance, a component named `test` would be initialized by using a `<app-test></app-test>` tag in the root component template. This functionality allows components to be modular, reused and easy to implement.

3.1.2 Services

Services contain functionality that isn't associated with a specific view. Services share data between components. All services are singletons and are therefore not initializable. This allows for states to be shared between components. They are often used for fetching data outside of the application, like external servers and APIs. Services also contains certain special objects like Subjects. Angular Subjects are objects that functions as queues with triggers. Adding data to the queue allows a component who is listening to the Subject to react when new objects are added to the queue.

A service can be injected into a component. Injecting a service is how a component has access to the functionality of the service.

3.2 WebSockets

This section introduces WebSockets and how it compares to the alternatives. To understand WebSockets it helps to compare it to well a known protocol like HTTP.

HTTP is a stateless protocol. That means that even though several requests are sent over the same connection, the server still treats them as separate requests. This is done to reduce overhead which would otherwise be used to reestablish connections. Each HTTP request is therefore meant to be intrinsically fulfilling.

HTTP requests and responses contain headers. They are used to provide metadata like authentication, data type and length. Headers are always included in every request and response. In HTTP, the client always specifies which actions the server should take. These actions (e.g. GET) are specified in the header. Each request is given exactly one response by the server and no response are sent without a request.

HTTP has certain advantages over WebSockets. These include automatic caching and load balancing [9].

WebSockets have certain properties that separates it from HTTP. The WebSocket protocol is a full-duplex system. This means that both the client and server can communicate simultaneously. WebSockets are an upgrade from HTTP. A WebSocket connection is established by first sending a HTTP GET request where the client indicates a desire to upgrade the connection. All future requests on this connection is sent using the WebSocket protocol and without the need of HTTP headers. The protocol is standardised and easily implemented [9].

If a connection requires full-duplex functionality, WebSockets far outperform HTTP [9]. An example of such a service is real-time messaging (e.g. group chats). Another example is video games that requires fast and frequent data transfer (e.g. online shooters).

There are several alternatives to WebSocket functionality. Polling works by sending an AJAX request every x seconds to the server. The delay between requests means that polling does not have the same real-time benefits as WebSockets. It also polls even though the server has no updates.

Another form of Polling is called Long Polling. Here every AJAX request asks the server to keep the connection open until new data is returned. As soon as the client receives a response, it sends a new request. This makes sure data is returned in real-time.

The mentioned alternatives to WebSockets have the benefit of being compatible with all REST APIs and HTTP-only servers.

3.3 RPC

Remote Procedure Call (RPC) is a communication protocol which can be described as an API [3]. It allows a client to directly call functions on the server as if it was a local machine. This essentially means the client accesses the server functions directly from a remote location. Functions are called with a method name and parameters. The method name is the name of the function on the remote server and the parameters are the inputs to that function.

An issue is that documentation usually only describes what functions do. It can be challenging to know what function to call and where to start. The tight coupling offers some critical advantages however. For starters, there's usually a less overhead since functions are called directly.

RPC benefits systems that needs a high message rate and low overhead. It requires users to have intricate knowledge of the underlying system (e.g. Administrator level clients) since they can directly call the system functions.

A popular alternative to RPC is the REST architecture. Instead of using functions as the fundamental unit REST uses resources. REST models resources, relations between resources and collections of resources.

The focus on resources means a stronger level of abstraction. This means a greater decoupling between the client and the server. This abstraction also means changes to the underlying system doesn't affect the API to the same degree [3].

Unfortunately there's no single standard specifying how a REST API is defined. RPC inherently follow the standard of the underlying system while REST does not. This can cause confusion and false sense of understanding in REST. Additionally, REST usually have big payloads and more ”chatty” networks than RPC [3].

REST benefits systems with a focus on entities and resources and also works best for systems with a great diversity of clients and a high need for easily understandable functions [3].

Chapter 4

Method(s) and Design

4.1 Introduction

The application displays a graph representing a factory layout of sensors and alarms. The nodes have a name and a status, and each edge have labels identifying the relationship between nodes. The graph keeps an up to date version of each node's status. The data is fetched from a server and formatted by the application before being drawn.

The application consists of four separate processes:

Visualizer:	The starting point of the application. It calls the Data Service and is responsible for drawing the graph. It also manipulates the graph in order to keep it up to date.
Data service:	The center of the application. All other processes interact only with the Data Service, and do so directly. It is called by the visualizer, fetches data from the Server, and calls the Formatter.
Formatter:	Parses graph data into *dot* formatted strings. This allows it to be drawn in the right format by the Visualizer.
Server:	The endpoint storing all graph data. Data includes graph structure and node status.

The application has two phases. In the initialization phase, the application fetches the graph structure and subscribes to status changes. In the main phase, the application responds to changes by updating the graph.

4.2 Initialization Phase

This phase runs once when the application starts. The code runs once, and prepares the application for the main phase. The phase consists of three steps:

1. Fetching graph structure
2. Building the graph
3. Subscribing to signal changes

The Visualizer requests the graph structure from the Data Service. The graph structure is represented as a parsed string in the *dot* format. This gives the Visualizer all the data it needs to draw the graph correctly. With the parsed string, the Visualizer can simply draw the graph by handing the string to the Viz.js wrapper for Graphviz which will then draw the graph.

After drawing the graph, the Visualizer triggers the Data Service to subscribe to signal updates from the Server. The Data Service communicates the request using the RPC protocol over a Websocket connection.

Below is the Visualizer algorithm for this phase:

Algorithm 12 Fetching parsed graph structure string

```
Procedure Visualizer.init()
    dot_formatted_string = Data_Service.get_graph_string();
    draw_graph(dot_formatted_string);
    Data_Service.subscribe_all_signals();
end
```

Algorithm 12 explained:

get_graph_string:	Triggers the data service to get the parsed string. The data service formats the graph structure using the Formatter.
draw_graph:	Draws the graph using GraphViz.
subscribe_all_signals:	The Data Service sends subscribe requests to the server. A request is sent per node in the graph.

The algorithm below shows the pseudocode for the Data Service when calling get_graph_string. Note that the Formatter and WebSocket communication is explained more in-depth in section 4.4 and 4.5.

Algorithm 13 Fetching Graph structure

```
Procedure Data_Service.get_graph_string()
    nodes = get_graph_structure(root_node_name);
    return Formatter.make_graph_string(nodes);
end

Procedure Data_Service.get_graph_structure(node_name)
    node = new Node(node_name);
    child_edges = websocket_get_child_edges(node_name)
    for child_edge in child_edges do
        child_node = get_graph_structure(child_edge.target_name);
        node.out_edges.add(child_edge)
    endfor
    return node;
end
```

Algorithm 13 explained:

get_graph_structure:	This function recursively finds nodes using the Depth first search algorithm. An edge is created for each child and added to the parent. When complete, the root node is returned. Note that this function expects an acyclic graph. Checking for previously visited nodes would probably be necessary in order to support cyclic graphs.
Formatter.make_graph_string:	The Formatter receives an array of nodes and uses it to create a *dot* formatted string (see section 4.4).
Node:	The graph nodes are represented by a Node object. The object has a name for labeling itself, a type which defines it's design and represents it's function. It also has a status which is it's current signal value and an array of out-edges. These edges point to it's child nodes, and each edge has a label.
websocket_get_child_edges:	The Data Service sends a WebSocket request for child edges. The edges have a label and a target node name. (see section 4.5 for WebSocket communication).

After drawing the graph, the Visualizer now activates subscriptions on the Data Service. The algorithm below shows the pseudocode for the Data Service when calling subscribe_all_signals:

Algorithm 14 Subscribing to signal changes

```
Procedure Data_Service.subscribe_all_signals()
    on(websocket_open) do
        signals = fetch_all_signals();
        handle_subscriptions();
        enable_subscriptions();
        for signal in signals do
            signal_subscribe(signal);
        endfor
    end
end
```

Algorithm 14 explained:

on websocket_open:	This code is triggered as soon as the WebSocket connection to the Server is open (see section 4.5). This makes sure the code does not run prematurely and fails.
fetch_all_signals:	Requests a list of signals from the Server and filters it for relevant signals. The list need to be filtered since many signals represent other values like Server status and node values not used in this project. The list contains one signal for each node, representing the nodes status.
enable_subscriptions:	Sends a request defining how frequent updates should be sent from the Server. The default is currently every 100ms. The function also sends a request to the Server that returns an error. This is necessary since the request also triggers the WebSocket package to handle new values received from the Server (function websocket_new_value_received in algorithm 15). The error does not affect anything else and is purely for client side activation. Neither requests trigger the Server to start sending updates.
signal_subscribe:	This triggers the Server to send new updates. The Data Service sends a request for each signal.

4.3 Main Phase

When the Data Service receives a new value, it queues the new value in an Angular Subject. This triggers the Visualizer to handle the new value. The Visualizer then searches the graph for the correct node and updates it's status, and optionally, it's visuals.

The algorithm below shows the pseudocode for the Data Service when a new signal update is received from the Server and also how the Visualizer is triggered by the new updated value:

Algorithm 15 Received signal update

```
 1: Procedure Data_Service.handle_subscriptions()
 2:     on(websocket_new_values_received) do
 3:         update global list of signal values with the new values;
 4:         notify subscribers about the new values;
 5:     end
 6: end
 7: Procedure Visualizer.updated_signal(signal)
 8:     node = get_element_by_id(signal.node_name);
 9:     if is_sensor(node) do
10:         update_sensor(node);
11:         if animate_sensors_on do animate_sensor(node);
12:     else if is_alarm(node) do
13:         update_alarm(node);
14:         if animate_alarms_on do set_alarm_color(signal.value);
15:     end
16: end
```

Algorithm 15 explained:

2-5:	This code gets triggered each time a new signal update is received from the server. Since each update is received at intervals, the Server returns a list of all changes since the last update. The function iterates over the signals and each signal update is pushed to a queue on an Angular Subject. This triggers Visualizer.updated_signal once for each updated value.
updated_signal:	The function finds the current HTML element representing the node referred to in the signal data. It then updates the node value and triggers a visual change if configured to do so.
update_sensor:	Updates the status text on the node element with the new value.
animate_sensor:	The node changes color and fades slowly back to normal to indicate that a new value was received. This gives a better overview of general signal traffic. Animations can be turned on or customized with different colors, strength and timing.
update_alarm:	Switches the alarm status between True and False depending on the signal value. If the new signal value is 0, the alarm is set to False, else it is set to True. The value indicating a False alarm defaults to 0 but can be customized.
set_alarm_color:	Sets the alarm color based on it's status. Can be deactivated and colors can be customized.

4.4 Formatting graph data

To draw the graph, the graph data needs to be converted to a string. The *dot* formatted string enables GraphViz to draw the graph as intended.

The input given to the Formatter is a root node with name, type and outbound edges pointing to other nodes. Below is an example of the output from the Formatter. The output creates the draws the graph in figure 4.1.

```
digraph {
    /* ===== Options: ===== */
    ratio="1"
    size="10"
    margin="2"
    center="true"

    /* ===== Declarations: ===== */
    node [shape=diamond,height=2,width=2] e;
    node [shape=circle] a; b; c; d;

    /* ===== Labels: ===== */
    a [label = <<b>a</b><br/>signalValue>, id = "a"];
    b [label = <<b>b</b><br/>signalValue>, id = "b"];
    c [label = <<b>c</b><br/>signalValue>, id = "c"];
    d [label = <<b>d</b><br/>signalValue>, id = "d"];
    e [label = <<b>e</b><br/>signalValue>, id = "e"];

    /* ===== Connections: ===== */
    a->b[label=ab];
    a->d[label=ad];
    a->e[label=ae];
    b->c[label=bc];
}
```

Output above explained:

Options:	Global properties for the graph that improve readability. Property *ratio* is set based on screen size ratio to dynamically fit any client. The property *size* scales the graph and text, *margin* adds node spacing, and *center* centers the whole graph. The properties are simply chosen based on subjective appearance.
Declarations:	Design definitions for alarms and sensors, respectively. The keyword ”node” is used to define node styling. The brackets contain the styling and after that the affected nodes are written. Sensors are given strict dimensions since they tend to be horizontally stretched without it. A declaration is created by filtering nodes on the current type and adding the node names to the end of the string.
Labels:	The text on each node. Includes HTML markup for readability in the graph. If HTML markup is used, Graphviz requires the label string to contain less than (<) and greater than (>) signs on each end of the string, respectively. Labels are generated by looping through all nodes and putting each nodes name into a template.
Connections:	Draws edges between nodes. The label property gives each node an edge label. The list is generated by looping through all nodes and generating a connection string based on a template, the current node and all it’s out-edges.

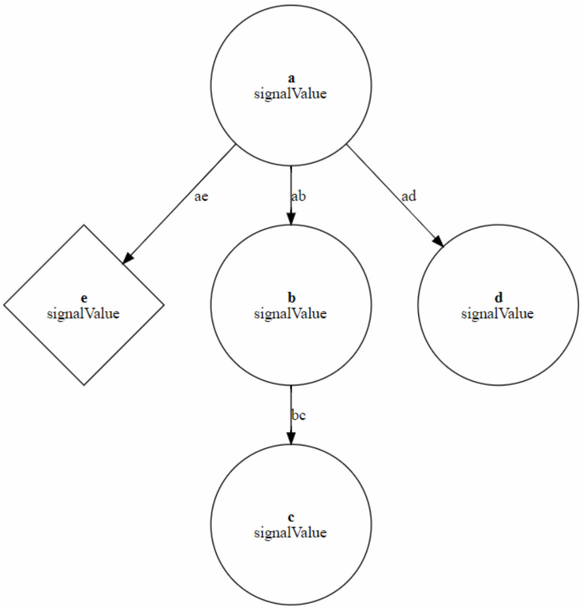


Figure 4.1: Example output

The drawing contains no colors. Colors are added after the graph is drawn in the Visualizer. Since color is changed when animations are active, initial color is set using the same methods. This is more consistent since there are irregularities between the styling conventions in *dot* and HTML.

4.5 Server communication

Communication between the Data Service and the Server is done using RPC over WebSocket. A connection is established which triggers further transmission from the Data Service. Below are the different methods used for communicating using the Angular websocket-rpc package:

Incoming communication using WebSocket.on(method name, handler function):
This is triggered when the Server responds to certain specific function calls. The method name specifies what function the Server is responding to. The handler function is triggered by the response and sometimes the response contains data the handler function can use. The following methods are used in this project:

open: Received when the WebSocket connection is opened. Used in line 2 in algorithm 14 *"Subscribing to signal changes"* to trigger activation of subscriptions.

Signal.OnUpdate: Received in a response to the server subscribing to signal updates. It returns a list of signals. Each signal contains a new value as well as a integer called handle for identifying nodes. Nodes with a handle value matching any of the signals are updated with the new value. Used in line 2 in algorithm 15 *"Received signal update"*. Note that **Signal.OnUpdate** is only triggered because **WebSocket.Subscribe('Signal.OnUpdate')** is called. The subscribe function is not recognized by the Server, so an error is returned. The subscribe function is essential in making **Signal.OnUpdate** work so it is still necessary to call it.

Outgoing communication using WebSocket.call(method name, parameters):
This function sends a request to the Server. The method name corresponds to the method that gets triggered on the server. The parameters allows the Data Service to further specify the request. The following methods are used in this project:

IPS.Device.Connections: Requests a list of outgoing edges for a given node. The node is specified using the format {`Device: node_name`} where `node_name` is replaced with the actual name of the node. Each edge has the properties `Target`, for child node name, and `Label`, for the edge label text. This method is used in line 8 in algorithm 13 *"Fetching Graph structure"*.

Signal.List: Requests a list of signals. The list contains all signals and therefore no input parameters are necessary. Each signal is in the form {`path, type, unit`}. `path` is in the form `node_name:signal_name`. `type` refers to the value type (e.g. integer, boolean, etc.). `unit` is the unit of measure (e.g. ml/min, bar, etc.). Used in part of fetch_all_signals in algorithm 14 *"Subscribing to signal changes"*.

Signal.Subscription: This function takes a parameter, `rate`. The parameter defines how frequent signals should be sent from the Server in (milliseconds). Used in enable_subscriptions in algorithm 14 ”Subscribing to signal changes”.

Signal.Subscribe: Subscribes to a given signal. The method takes one parameter, `path`, which is the same as mentioned in **Signal.List**.

4.6 Running the project

The application requires Node.js 10.9.0 or higher. To run the app, simply run these commands in the project folder:

1. `npm i`
2. `ng serve --open`

To make the application accessable on the local network, `--open` can be replaced with `--host=xxx.xxx.xxx.xxx` with the computers IPv4 address. If this causes an ”Invalid host header”-error, an aditional argument can be appended: `--disable-host-check`.

Controls, visuals and connectivity can be customized. These values are configured in the environmental variables. The variables contained in the file **root\src\environments\environment.ts** are listed below:

root_node	Name of the root node in the graph.
websocket_url	The server endpoint
false_alarm_value	The value that indicates if an alarm is off.
subscription_interval	How often the Server should return new signal values (in ms).
size	Size of node graph.
margin	extra spacing between nodes.
animate_sensors	whether to animate sensor signal updates.
animate_alarms	whether to update colors for alarm signal updates
minFill	opacity when update detected
maxFill	opacity when update animation is done
fps	how many frames of animation for the update animation.
cooldown	How long the animation should last in seconds.
node_style_settings	Control color, shape, size and font settings for alarms and sensors.
zoom_scale_timeout:	retry interval for activating zoom and scale functionality in milliseconds.
zoom_scale_settings	Control zoom sensitivity and thresholds.

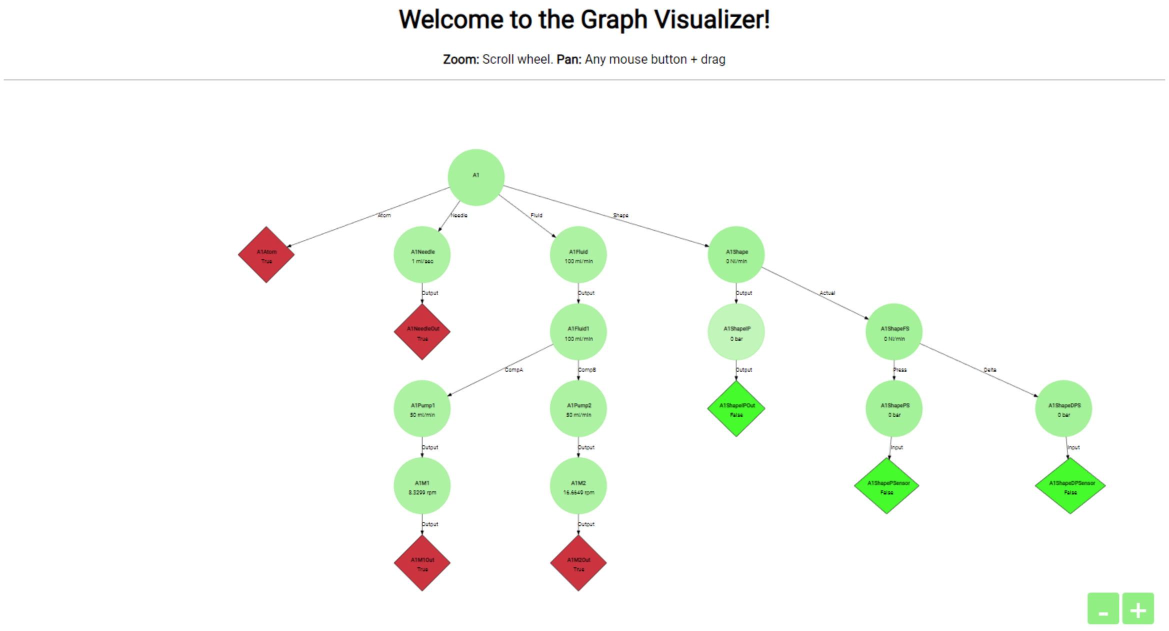
Chapter 5

Results

This chapter shows the result of running the application. This includes the user interface and interaction with the application. This chapter also looks at different metrics and how they can be interpreted.

5.1 User interface

When the application runs, the initialization phase starts and the graph is drawn. The user is greeted with the image in figure 5.1.



Welcome to the Graph Visualizer!

Zoom: Scroll wheel. **Pan:** Any mouse button + drag

Figure 5.1: The user interface.

To interact with the graph, the user can zoom and pan. The user zooms by scrolling the mouse wheel. To support mobile devices, zoom controls are present in the lower right corner. To pan around, the user can drag with any mouse button or finger. These controls allows the user to navigate any sized graph and focus on areas of importance.

5.2 Performance metrics

This section looks at performance with varying signal update frequency and animation settings. Figure 5.2 shows an performance review of the initialization phase. The total time elapsed is 1109ms on average. Each category is shown in percentage of time elapsed. The figure illustrates that almost 74% of time elapsed is due to scripting [1].

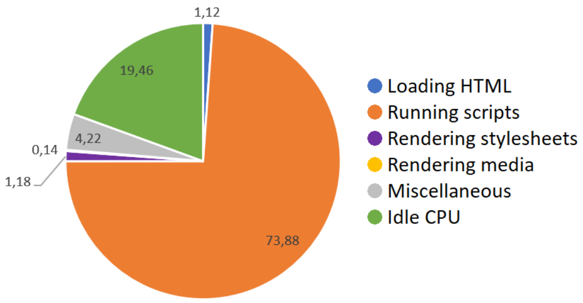


Figure 5.2: Phase 1 performance.

The main phase runs indefinitely. To give a good estimation of time elapsed, all further metrics runs for 60 seconds each. Figure 5.3 shows the performance of the main phase with different signal update intervals. 5.3a has a 100ms interval, while 5.3b has a 1ms interval. 5.3c has a 5 seconds update interval while the 5.3d has a 100ms interval but no animation.

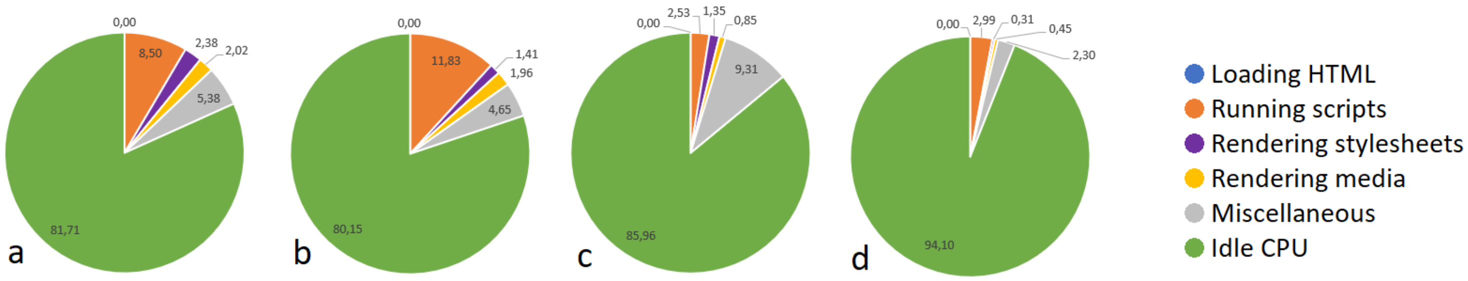


Figure 5.3: Phase 2 performance with different signal update intervals.

5.3a and b are very similar. 5.3a is configured to receive updates 100 times less frequent than 5.3b but the difference in performance is very low. This is probably due to the server having a minimal interval time, meaning 5.3b is not actually running at a 1ms interval. The difference between 5.3a and 5.3c is more noticeable. 5.3c receives updates 50 times less frequent than a. The scripting time is about 3.3 times lower while the overall work on the CPU is about 30% lower.

5.3a and d have the same update interval. 5.3d however has no animations active. Instead of changing opacity and colors, nodes are simply updated with new text for their values. The performance is highly improved with a 3.1 times performance improvement in overall CPU load.

---

[1] Benchmark hardware specification for tests: Windows 10, Intel Core i7-4770K, 16GB 1600MHz RAM, GTX 1080

5.3 Discussion

Looking at figure 5.2 and 5.3 There's a clear majority of idle CPU time in the main phase compared to the initialization phase. This is possibly because the initialization phase has to load everything as quickly as possible. The main phase mostly waits for signal updates. Looking at the figures we can see how increasing the signal update frequency to reduce latency has little effect on performance. Disabling animations however, impacts performance more significantly. Improving animation efficiency could possibly improve performance in the main phase by a huge amount.

Overall the application runs fast and updates the graph quickly compared to the current solution. Replacing the static image output with a HTML document allows for more flexibility in manipulation. Using HTML also allows for far greater platform support. Using Angular allows for a modular workflow that makes it easier to update existing features and add new ones.

Chapter 6

Conclusion and further work

This thesis presented a replacement for the current system used by ABB. The replacement was developed with the Angular framework and used RPC WebSocket communication. The results show that the solution is efficient even with low latency hight traffic loads. The application runs fast, is flexible and has wide platform support.

Several improvements can be done. Adding support for additional signal information by adding a contextual menu to nodes. Allowing for live modifications and grouping in the graph. Delivering notifications for certain alarms or threshold signal values. Users might also want additional services that correlate with the information in the graph, like live video feeds of the factory floor or signal/alarm logs.

Chapter 7

References

[1] Architecture overview. `https://angular.io/guide/architecture`. Accessed: 2018-04-28.

[2] What is graphviz? `https://www.graphviz.org/`. Accessed: 2018-12-05.

[3] N. Barbettini. Api throwdown: Rpc vs rest vs graphql, iterate 2018. `https://www.youtube.com/watch?v=IvsANO0qZEg`. Accessed: 2018-05-01.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2009.

[5] P. Eades. The median heuristic for drawing 2-layerde networks. *Tech. Report, 69, Dept. of Computer Science, Univ. of Queensland*, pages 1–13, 1896.

[6] H. Eichelberger. Aesthetics and automatic layout of uml class diagrams. 2005.

[7] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[8] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[9] D. Misic. A beginner's guide to websockets. `https://2018.pycon-au.org/talks/45211-a-beginners-guide-to-websockets/`. Accessed: 2018-04-30.