Layout in Visual Modelling

Gitte Bluekens

University of Antwerp, Belgium
supervisor: S. Van Mierlo

Abstract

As part of a Model Driven Engineering course I decided to tackle visual modeling in AToMPM. A lot of us like working with modeling languages. We want them to be visually understandable, but we don't want to spend time making our model 'pretty'. This paper will talk you through different layout algorithms and how they can be implemented, specifically in a transformation rule in AToMPM. Simply running one of these algorithms on your model will modify the vertices to the visually best place on the canvas.

Keywords: Visual modeling, AToMPM, Spring-embedder, Force-transfer, Tree-like layout, Circle layout

1. Introduction

The usefulness of visual modeling is dependent on how elements of a model are visually arranged. Hence, any tool supporting visual modeling should provide some mechanisms to reduce the burden of drawing models with good layouts. In this paper, I will discuss some automatic layer techniques implemented in the tool AToMPM.

AToMPM. ATOMPM is an acronym for "A Tool for Multi-Paradigm Modeling". It is used for modeling, meta-modeling, and transforming models with graph grammars. AToMPM allows language developers to create visual domain-specific languages, and domain experts to use these languages. A language is defined by its abstract syntax in a metamodel, its concrete syntax(es), which

define(s) how each abstract syntax element is visualized, and its semantics definition(s), either operational (a simulator) or translational (by mapping onto a known semantic domain). [1]

Modelling a Model Transformation in AToMPM. A model transformation consists of a set of rules that matches and rewrites parts of the model, and a schedule that governs the order in which rules are executed. [1] For every layout algorithm that I implemented in AToMPM, I created a new transformation schedule which has the name of the algorithm. These schedules all look alike. They only consist of one rule, a starting point and a success and failure point. The layout magic will happen in that one rule.

When creating a rule, it is initialized with the three basic components of every rule: a Negative Application Condition (NAC), a Left-Hand-Side (LHS), and a Right-Hand-Side (RHS). Visually, all the rules I wrote will look the same. They don't need the NAC, so I deleted this box in every rule. The Left-Hand-Side and Right-Hand-Side only contain one place (the same for both Left- and Right-Hand-Side). This makes sure that every valid PetriNet will be matched to this rule. The actual layout algorithm is implemented in the action part of the Right-Hand-Side of the rule.

2. PetriNets in AToMPM

I decided to implement visual layout algorithms for the PetriNet language. This language consists of two vertices, namely Place and Transition, and two edges, namely PlaceToTransition and TransitionToPlace. The PetriNet language is available in AToMPM in the folder Formalisms/PN.

To be able to use this language, I first had to make some small adjustments. A Place or Transition in AToMPM, hereafter called vertex, is derived from the Positionable class. This means that the vertex can be placed anywhere on the canvas, and that the position of the vertex will change as the vertex changes position on the canvas. However the Right-Hand-Side of a rule can only change attributes of a vertex. Therefore I had to give the vertices an attribute 'position'

in the PetriNet model. This attribute is mapped on the position of Positionable, by using the mapper and parser functions of the PetriNet metamodel.

3. Spring-embedder algorithm

3.1. General algorithm

Algorithm 1 Spring-Embedder

```
1: Input: A Graph G = V, E
2: Output: An embedding of G
3: for all v in V do
4:     v.forceVector = [0,0]
5:     v.charge = chargeStrength * v.diagonalLength
6: end for
7: for i in range(0, 101) do
8:     Repulsion(V)
9:     Attraction(E)
10:    Gravity(V)
11:    for all v in V do
12:        v.pos = v.pos + v.forceVector
13:        v.forceVector = 0
14:    end for
15:    convergence check
16: end for
```

In the spring-embedder algorithm, edges are simulated as springs and vertices as rings to which the springs are attached to. It is fairly simple to implement. To improve the convergence speed and quality of the final drawing, a pre-processing step of circle layout or a random layout algorithm is recommended.

The initialization step consists of setting 2D force vectors to zero for every vertex, and setting repulsion charges to prevent vertices from overlapping. They

are set to the diagonal length of each vertex, multiplied by the chargeStrength. This variable can be adjusted by the user, I've chosen a default value of 2. After the initialization, the forces acting on the vertices are repeatedly calculated. This number of iterations is fixed to a default of 100 iterations. This means that the algorithm stops after 100 iterations, or if a convergence threshold is triggered. The default convergence threshold is 10, since that number was experimentally found to work well. [2] I also implemented two forgiveness rounds in the convergence check. This means that the maximum force must be less than the threshold for three consecutive rounds to break out of the loop.

3.2. Repulsion algorithm

The repulsion algorithm is responsible for avoiding the vertex overlaps. This is done by generating large repulsive forces whenever two vertices overlap. Initially, this algorithm calculates the Manhattan and Euclidean distances between the pair of vertices. We will weight off the impact of the force to a given threshold. If the distance between the vertices is large enough, the force will be ignored to make the algorithm more efficient. In my implementation, I defined the threshold as 100.

If the impact of the force is significant, a scalar force is calculated proportional to the charges of the vertices and inversely proportional to the square of the distance separating the vertices. This force is used to alter the force vector of the first vertex. If the Euclidean distance is less than 0.1, then the previously calculated charge will simply be added to the force vector of the first vertex.

3.3. Attraction algorithm

The attractive algorithm first tries to find the Manhattan and Euclidean distances between the pair of vertices connected to a chosen edge. The distance cannot be smaller than a chosen minimum distance to avoid precision and divide by zero issues. In my implementation, I chose the value of minDistance to be 0.1. Next, the spring force is calculated using the physical equation for springs, since the algorithm treats edges as physical springs. The springConstant is set

Algorithm 2 Repulsion

Input: A set of vertices V
Output: Update force vectors for V
for all v_i in V **do**
    for all v_j in V **do**
        if $v_i \neq v_j$ **then**
            Calculate the Euclidean distance between v_i and v_j
            Calculate the Manhattan distance vector using v_i and v_j
            if abs(Euclidean distance) > threshold **then**
                charge = v_i.charge + v_j.charge
                if abs(Euclidean distance) > 0.1 **then**
                    force = charge / (Euclidean distance)^2
                    v_i.forceVector = v_i.forceVector + (Manhattan distance vector) * force
                else
                    v_i.forceVector = v_i.forceVector + charge
                end if
            end if
        end if
    end for
end for

Algorithm 3 Attraction

Input: A set of edges E
Output: Update force vectors for vertices linked to E
for all e in E **do**
    v_s = e.getSource()
    v_t = e.getTarget()
    if $v_s \neq v_t$ **then**
        Calculate the Euclidean distance between v_s and v_t
        Calculate the Manhattan distance vector using v_s and v_t
        if abs(Euclidean distance) < minDistance **then**
            Euclidean distance = minDistance * sign(Euclidean distance)
            Manhattan distance = minDistance
        end if
        force = springConstant * ((Euclidean distance) - idealSpringLength) / (Euclidean distance)
        v_s.forceVector = v_s.forceVector + (Manhattan distance vector) * force
        v_t.forceVector = v_t.forceVector - (Manhattan distance vector) * force
    end if
end for

to a value of 0.1 in the implementation, since it is proven that this value works well across a wide range of graphs. The idealSpringLength is set to 100. If this lenght is chosen to be a smaller value, the chances are bigger that there will be overlapping vertices. [2] Finally, the computed spring force is multiplied by the 2D Manhattan distance vector and added to the force vector of one vertex, and subtracted from the other.

3.4. Gravity algorithm

Algorithm 4 Gravity

1: **Input:** A set of vertices V
2: **Output:** Update force vectors for V
3: barycenter = ($\sum_{v \in V}$ v.pos) / | V |
4: **for all** v in V **do**
5: Calculate unit vector between v.pos and barycenter
6: v.forceVector = v.forceVector + unit vector * gravityStrength
7: **end for**

The gravity algorithm imparts upon each vertex a velocity towards the gravitational field source. This is determined to be the barycenter of all the vertices. This algorithm will make sure that the area is used efficiently, and that the vertices are not spread over the entire canvas. In fact, the algorithm will yield a circular drawing because of the two dimensional character of gravity. The force vector imparted on each vertex is calculated as the unit vector between the vertex and the barycenter. This vector is then multiplied by the strength of the gravity field. A value of 10 for this strength is proven to work well for small sparse graphs. [2]

3.5. Analysis

The spring-embedder algorithm runs with a constant iteration amount. The repulsion algorithm dominates the time complexity for each simulation iteration, requiring O(|V|^2) time. Since the attractive algorithm only uses O(|E|) time and

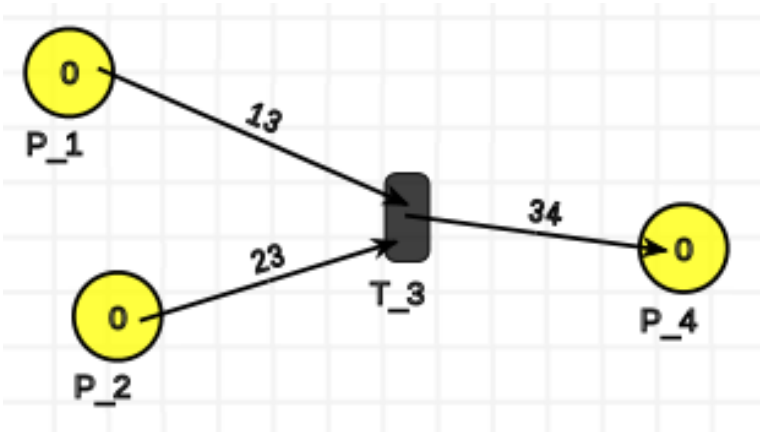

Figure 1: Before Spring-Embedder algorithm

the gravity algorithm only uses O(|V|) time, the overall time complexity for the spring-embedder algorithm is O($|V|^2$).

3.6. Reflection

There were some difficulties I encountered implementing this algorithm. The most prominent one was the use of the Manhattan distance vector. I had never heard of this before and I couldn't find any information on the internet. After discussing this with my supervisor, I decided to take the Manhattan distance and split it up in a vector. Since the Manhattan distance is calculated as $d(p,q) =\mid p_1 - q_1 \mid + \mid p_2 - q_2 \mid$, I decided to interpret the Manhattan distance vector as $[\mid p1 - q1 \mid, \mid p2 - q2 \mid]$.

I also had to adjust the pseudocode that I based my implementation on. [2] In his paper, Dubé talks about the sign of the Manhattan vector. This didn't make much sense to me as in my opinion a vector doesn't have a sign. Since the Manhattan distance also only uses absolute values, I decided to always interpret this sign as positive. Furthermore, I made some small adjustments to the pseudocode to make it more understandable.

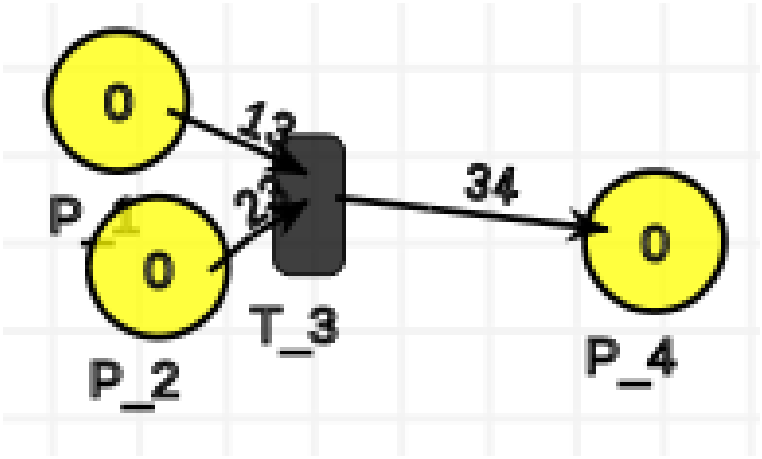

Figure 2: After Spring-Embedder algorithm

4. Force-transfer layout algorithm

The force-transfer drawing technique consists of a an initialization phase and a simulation phase. The initialization phase sets the forces acting on each vertex to zero. In the simulation phase, each vertex exerts forces on overlapping neighboring vertices. These forces are calculated in the calculateForce function. We start by calculating the unit vector and Euclidean distance between the pair of vertices. Then, a scalar force magnitude is computed. In this computation, we speak of minSeparation, namely the minimum separation that we expect between two vertices. I chose this value to be 50 in my implementation. There is also the variable separationForce, which I chose to be 1 in the implementation. The direction of the force is determined by the greatest separating distance between the vertices, so the vertices are moved as little as possible. This means that the vertices will only move vertically or horizontally. The simulation terminates once the forces have pushed all vertices apart such that no overlap

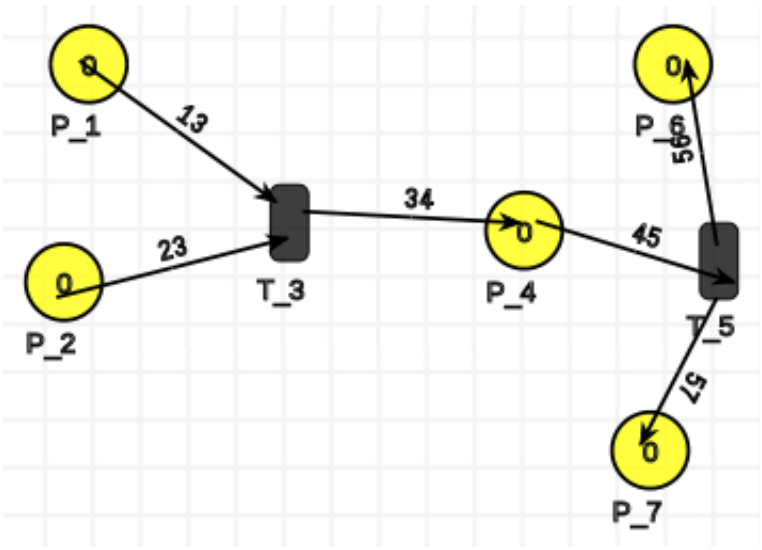

Figure 3: Before Spring-Embedder algorithm

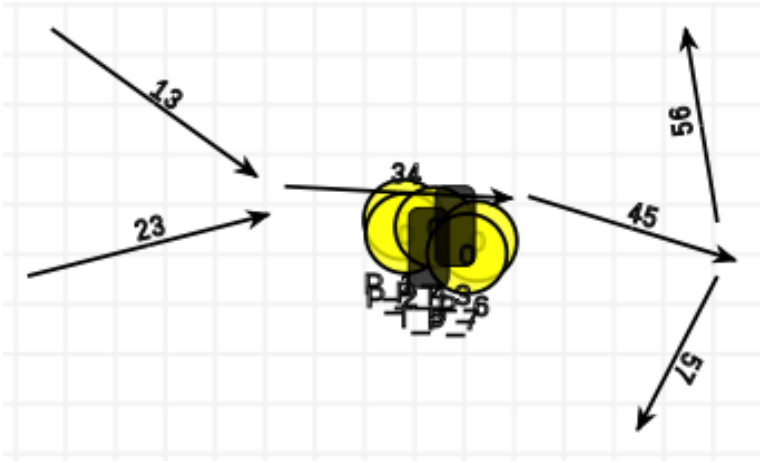

Figure 4: After Spring-Embedder algorithm: Force-transfer layout is needed

Algorithm 5 Force Transfer

```
1: Input: A graph G = (V, E)
2: Output: An embedding of G
3: for all v in V do
4:     v.forceVector = 0
5: end for
6: for all i in range(0, 51) do
7:     isMoving = False
8:     i = 0
9:     j = 0
10:    while i <| V | do
11:        while j <| V | do
12:            if i ≠ j then
13:                isMoving = calculateForce(v_i, v_j)
14:            end if
15:            j = j + 1
16:        end while
17:        i = i + 1
18:        j = i
19:    end while
20:    if not isMoving then
21:        break
22:    end if
23:    for all v in V do
24:        v.pos = v.pos + v.forceVector
25:        v.forceVector = 0
26:    end for
27: end for
```

Algorithm 6 CalculateForce

Input: A pair of vertices, v_i, v_j
Output: Update force vectors for v_i and v_j
$[u_x, u_y]$ = the unit vector between v_i and v_j
d_x = u_x-1 * ((((v_i.width + v_j.width) / 2) + minSeparation)
d_y = u_y-1 * ((((v_i.height + v_j.height) / 2) + minSeparation)
forceMagnitude = separationForce * (Euclidean distance - min(abs(d_x), abs(d_y)))
if forceMagnitude < -1 **then**
    if abs(u_x) > abs(u_y) **then**
        v_i.forceVector.x = v_i.forceVector.x + (u_x * forceMagnitude)
        v_j.forceVector.x = v_j.forceVector.x - (u_x * forceMagnitude)
    else
        v_i.forceVector.y = v_i.forceVector.y + (u_y * forceMagnitude)
        v_j.forceVector.y = v_j.forceVector.y - (u_y * forceMagnitude)
    end if
    return True
end if
return False

remains. This is checked by using the isMoving variable. The simulation can also be ended by a fixed number of iterations.

4.1. Analysis

The first loop of the force-transfer algorithm is bounded by O(|V|), the inner loop is bounded by O(50*|V|^2) and the last loop is bounded by O(50*|V|). This means that the overall time complexity of the force-transfer algorithm is O(|V|^2).

4.2. Reflection

The Force-Transfer layout algorithm is an algorithm that will make sure that there are no overlapping vertices in the canvas. This makes it particularly useful to run this algorithm after running a first algorithm that does not care about overlapping. It also means that the algorithm itself is not very hard to understand. It will simply increase the distance between vertices whenever necessary. When testing this algorithm, I found that it only works well with small PetriNets. I don't think that it has anything to do with the algorithm itself, but it's more an issue in AToMPM. It takes very long to run the algorithm for a bigger PetriNet and this will eventually lead to AToMPM not responding anymore. I wasn't able to solve this problem, but the effects of the algorithm can be seen in smaller PetriNets.

5. Cirle layout algorithm

The circle layout algorithm is best used on subgraphs or small graphs. It makes an excellent preprocessing step for a force directed method such as the spring-embedder algorithm.

In the circle layout algorithm, all vertices are first sorted topologically. I did this intuitively by writing an algorithm that checks for every vertex if it is already marked sorted. If it is not, it will be the next topologically sorted vertex followed

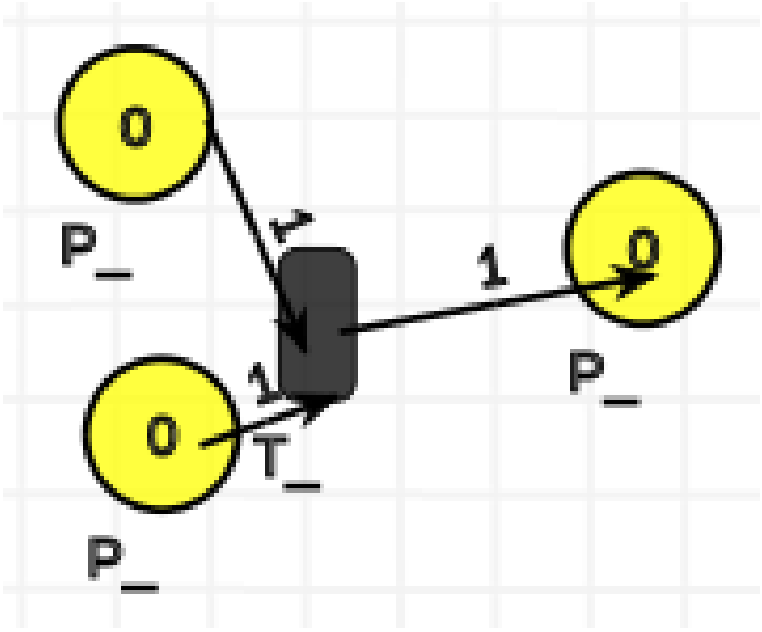

Figure 5: Before Force-Transfer algorithm

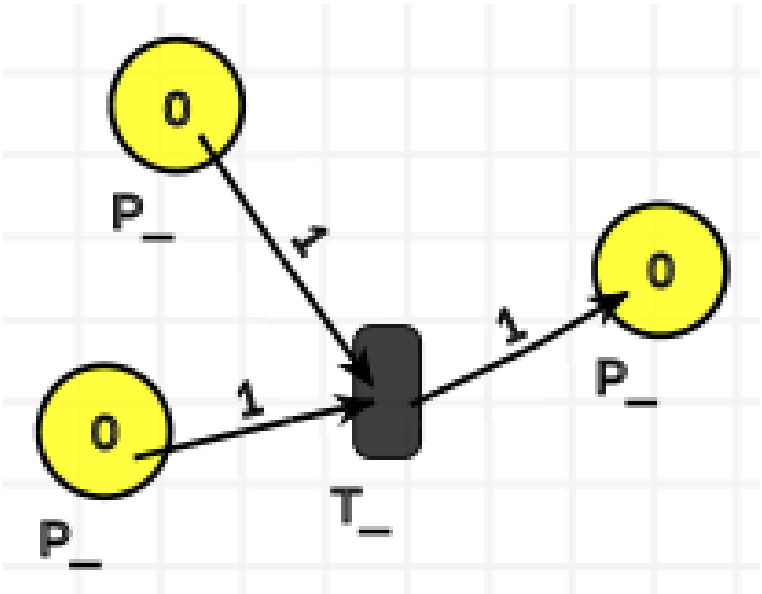

Figure 6: After Force-Transfer algorithm

Algorithm 7 Circle

Input: A graph G = (V,E)
Output: An embedding of G
Obtain a topological sort of V
perimeter = 0
for all v in V **do**
    v.boundingCircleDiameter = sqrt(v.width^2 + v.height^2) + offset
    perimeter = perimeter + v.boundingCircleDiameter
end for
diameter = perimeter / π
interval = $v_{|V|-1}$.boundingCircleDiameter / (2 * perimeter)
for all i in range (0, len(V)) **do**
    x = diameter * (1 - sin(interval * 2π))
    y = diameter * (1 - cos(interval * 2π))
    v_i.pos = [x, y]
    if j $\neq$ len(V)-1 **then**
        interval = interval + ((v_i.boundingCircleDiameter + v_{i+1}.boundingCircleDiameter) / (2 * perimeter))
    end if
end for
end for

by its children until no children are left. Only then the next vertex is checked. The next step is then to calculate the perimeter of the circle. This calculation uses an offset value to make sure that there is room left for edges. The larger you choose this value to be, the further the vertices will be positioned from each other.

Finally, an interval fraction is calculated between 0 and 1. This interval will become the radian angle used to calculate the vertex positions on the circle.

5.1. Analysis

Since all steps of the circle algorithm are done in linear time, the algorithm has a linear overall run-time.

5.2. Reflection

Personally, I think this algorithm gives a very nice result. There is no overlap between the vertices and all of them are evenly spaced over the canvas. By first sorting the vertices topologically, it is made sure that there is as little as possible edge overlapping.

There were some slight changes that I had to make to the pseudocode of Dubé to get a working algorithm. [2] The most prominent change is situated in the second for-loop. Where Dubé chooses to loop from 1 to len(V), I chose to loop from 0 to len(V). Starting from 1 results in the first vertex not changing place. This is not desirable. I also made use of an extra if statement. Dubé didn't make use of this statement, which resulted in a failure to get v_{i+1}.boundingCircleDiameter since v_{i+1} does not exist.

6. Tree-like algorithm

The tree-like layout algorithm gives good results on graph structures that are really trees.

The first step of the algorithm is to find all the root vertices in the graph. This

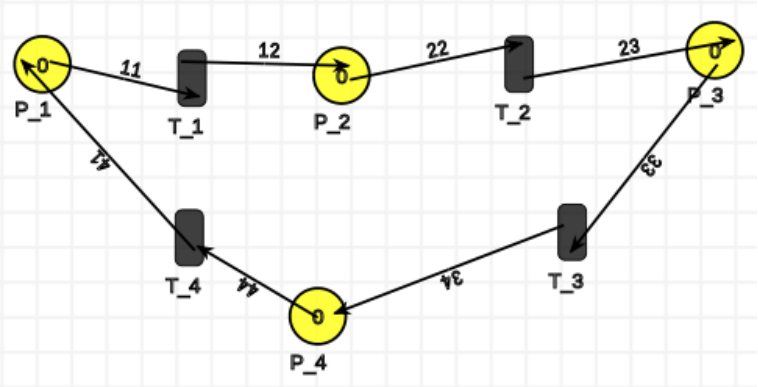

Figure 7: Before Circle-like layout algorithm

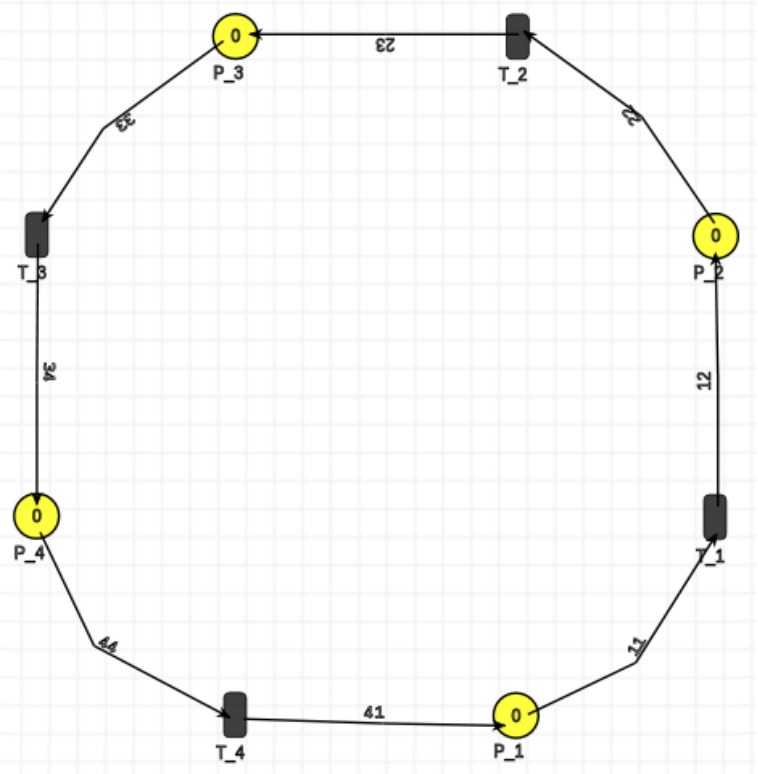

Figure 8: After Circle-like layout algorithm

Algorithm 8 Tree-like

```
1: Input: A graph G = (V,E)
2: Output: An embedding of G
3: R = findRootVertices(V)
4: maxHeight = maximum height of all root vertices
5: x_pos = 0
6: y_pos = 0
7: for all r in R do
8:     w = 0
9:     for all v in r.getChildren() do
10:        w = layoutNode(v, x_pos + w, y_pos + y_offset + maxHeight) + x_offset
11:    end for
12:    r.pos.x = x_pos + (w / 2) - (r.width / 2)
13:    r.pos.y = y_pos
14: end for
```

is done by the findRootVertices function. This function simply checks if a vertex has an incoming edge. If not, this means that the vertex is a root, and all children of this vertex are considered not to be roots. If after this loop there are still vertices left unmarked, this means there is a cycle in the vertices. We then pick a random vertex as root and mark all children until all vertices are marked.

A recursive process is then started to assign the children of each root vertex coordinates before the root itself. This is done in the layoutnode function. The algorithm is designed to assign vertices with no children coordinates immediately, and vertices with children make the recursive call.

6.1. Analysis

Since all steps of the tree-like algorithm are done in linear time, the algorithms has a linear overall run-time.

Algorithm 9 FindRootVertices

```
1: Input: A set of vertices V
2: Output: A set of root vertices R
3: R = []
4: for all v in V do
5:     if v has no incoming edges then
6:         R.append(v)
7:         mark all children of R as not root
8:     end if
9: end for
10: for all v in V do
11:     if not v.marked then
12:         R.append(v)
13:         mark all children of R as not root
14:     end if
15: end for
```

Algorithm 10 LayoutNode

1: **Input:** A vertex v, x_{pos} coordinate, y_{pos} coordinate
2: **if** not v.hasChildren() **then**
3: v.pos.x = x_{pos} + ((v.width + x_{offset}) / 2) - (v.width / 2)
4: v.pos.y = y_{pos}
5: return v.width
6: **else**
7: w = 0
8: h = v.height + y_{offset}
9: **for all** v_{child} in v.getChildren() **do**
10: w = layoutNode(v_{child}, x_{pos} + w, y_{pos} + h) + x_{offset}
11: **end for**
12: v.pos.x = x_{pos} + (w / 2) - (v.width / 2)
13: v.pos.y = y_{pos}
14: return w - x_{offset}
15: **end if**

6.2. Reflection

This algorithm is the first one I tried to implement in AToMPM. I thought it would be the easiest one, but it actually turned out to be the hardest one. The algorithm uses recursion to assign coordinates to the children of a vertex before assigning coordinates to the vertex itself. It is however undesirable to use recursion in the AToMPM rules. I tried rewriting the algorithm to work with loops instead of recursion, but I wasn't able to complete this algorithm. Since I did complete the findRootVertices function and a good part of the rest of the algorithm, I decided to explain the algorithm here anyway.

7. Conclusion

I really liked working on this project. Since everything you do gives a visual result, it was very satisfying to see the algorithms come to life. The thing left to do is fixing the edges. AToMPM does not support altering the begin or end position of an edge. Changing the position of the edge itself doesn't give a proper result either. Off course, it is possible to change the position of the edges after running the algorithm, but this is not desirable since it is often not clear anymore which edge belongs to which vertices. It is left to the reader to find a good way to modify the edges inside the algorithm.

References

[1] R. Mannadiar, S. V. Mierlo, H. Ergin, C. Hansen, E. Syriani, J. Corley, Atompm documentation, `https://msdl.uantwerpen.be/documentation/AToMPM/` (2016).

[2] D. Dubé, Graph layout for domain-specific modeling (2006).