Evaluation of Graph Drawings as Embeddings for Routing in Distributed Systems

Bachelor-Thesis von Nico Haase
März 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
P2P

Evaluation of Graph Drawings as Embeddings for Routing in Distributed Systems

Vorgelegte Bachelor-Thesis von Nico Haase

Prüfer: Prof. Dr. Thorsten Strufe
Verantwortlicher Mitarbeiter: Benjamin Schiller

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 5. März 2012

(Nico Haase)

Abstract

Routing in distributed networks, where the network structure with nodes and links between them is not to be modified, requires an identifier space such that each vertex has a coordinate, in contrast to other networks where routing tables help to choose the next hop to forward a message to. Graph drawing algorithms can handle the regular computation and assignment of coordinates. Although primary designed after visual aesthetics, we will simulate and analyze their field of application for routing purposes on graph models and existing real world networks.

Zusammenfassung

Für das Routing in verteilten Netzwerken, bei denen die Netzwerkstruktur aus Knoten und Verbindungen nicht verändert werden darf, wird ein identifer space benötigt, mit dem jedem Knoten eine Koordinate zugeordnet wird. In anderen Netzwerken kann mit Hilfe von Routingtabellen der nächste Hop bestimmt werden, zu dem eine Nachricht auf dem Weg zu ihrem Ziel weitergeleitet werden muss. Graphzeichnungsalgorithmen können diese Koordinaten systematisch berechnen und den Knoten zuweisen. Obwohl diese Algorithmen nach ästhethischen Grundsätzen entworfen wurden, simulieren und analysieren wir deren Einsatzmöglichkeiten im Routing in nach Modellen erzeugten Graphen und real existierenden Netzwerken.

Contents

1 Introduction

Routing is an essential part of computer networks. Packets of data need to be transported from one computer to another, and this should be done fast. In most cases, a packet travels over multiple hops. At each hop, the next hop to forward a message to must be chosen. In the Internet, this happens through routing tables which need to be computed, either statically or adaptively to the ongoing traffic, and the hops need additional storage for this table.

A different approach appears in DHTs like CAN[24] or Chord[27] where n-dimensional identifier spaces for the nodes are used. When a node joins such a network, a random partition of the identifier space is assigned to the node. It does not yet have any links to other nodes, and new links are created based on the common structure within that DHT. This ensures perfect greedy routing performance such that a simple routing algorithm will always succeed in any routing attempt with a small number of hops.

In other networks like wireless sensor networks or Darknets described in [8], the network structure is fixed. No new links should be formed, thus nodes cannot be placed at random positions where later added links maintain routing capabilities. Embeddings can be used to assign coordinates to each vertex, and finding a greedy embedding in which routing always succeeds would be good. Papadimitriou and Ratajczak postulated in [22] that for any planar 3-connected graph such an embedding can be found. Even if their conjecture was since proofed, it restricts the graph characteristics tightly.

We want to analyze the usage of graph drawing algorithms as embeddings of graphs. They can care about the assignment of coordinates as they were designed to visualize graphs. In our work, the coordinates that are computed by these algorithms are not used to place them onto a visible sketch, but to place them in the identifier space that will be used later for routing purposes.

This thesis is structured as followed: Section 2 introduces common notations. In Section 3, common characteristics of graph drawing algorithms and a classification is given. Afterwards, the drawing algorithms we evaluate are presented. Section 4 gives a brief introduction to routing and the three routing algorithms we use to evaluate the routing performance. Section 5 handles the implementation and Section 6 the evaluation of our simulations. In Section 7, a summary is given and the conclusions are drawn. Open questions for further work are given in Section 8.

2 Preliminaries

A graph $G = (V, E)$ is a pair of vertices V and edges E. Each edge $e \in E$ has a source vertex u and a destination vertex v and is denoted as the ordered pair (u, v). $E^-(v)$ denotes the set of incoming edges for v, such that $E^-(w) = \{(u, v) \in E : v = w\}$. $E^+(v)$ denotes the set of outgoing edges for v, such that $E^+(w) = \{(u, v) \in E : u = w\}$. $A^+(v)$ denotes the set of vertices that are connected through outgoing vertices of v, such that $A^+(v) = \{w \in V : (v, w) \in E\}$. $d^+(v)$ and $d^-(v)$ denote the out- and indegree of v, the number of outgoing or incoming edges.

An *embedding* is the mapping of one mathematical structure onto another. In our work, spatial coordinates of an identifier space are assigned to the vertices of a graph through such an embedding.

3 Graph drawing algorithms

3.1 Purpose of graph drawing algorithms

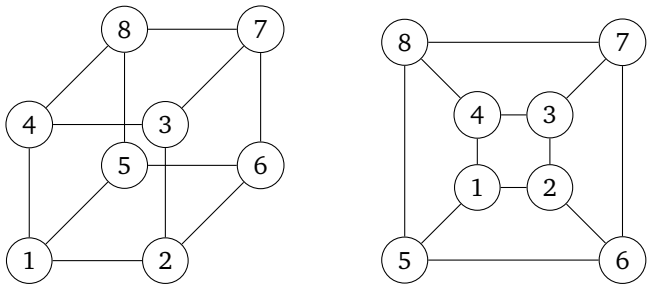
Graphs represent a wide variety of data in a structured form denoted in a pure mathematical way. Vertices and edges are substitutes for their counterparts in an interpretation. For example a social graph represents people (by vertices) and friendships between them (by edges). Another graph may represent the program flow of an algorithm.

The mathematical notation is usable for further processing of the data. But often this graph data should also be visualized to give a human readable interpretation of the data. Graph drawing algorithms have been developed for this task. They transform a graph from its pure notation to a graphical representation by adding an embedding.

Different algorithms can accentuate certain characteristics. For example, a drawing of a social network should accentuate groups of people, like colleagues or private friends, by clustering them. Other algorithms produce VLSI layouts for the positioning of the tiny parts of a microchip and care more about the lengths of circuits than a good looking drawing (Even et al. give an example for the connection between VLSI layouting and graph drawing in [11]). Depending on the requirements of the different fields of application, different metrics and balances between these metrics can be found to characterise a “good drawing”.

In [23], Purchase et al. study three aesthetics and their influences onto a drawing: the symmetry of a graph, the number of edge crossings, and the number of bends. They performed an empirical study with second year computer science students. Even if they cannot give a clear statement about the symmetry, they postulate that an increasing number of crossings or bends decreases the perceptual understandability of a graph.

As an introduction to his PhD thesis[30], Tunkelang defines three principles for a drawing. All edges should have a uniform length, edges that are not adjacent should have a large distance, and the number of edge crossings should be minimal. He gives an example for different weighted metrics which can be found in Figure 3.1.



(a) Drawing with a uniform edge length

(b) Drawing with no crossings

Figure 3.1.: Comparison of two drawings with different weighted metrics (copied from [30])

Another set of rules for a drawing is given in [29] by Tamassia et al. They differentiate between aesthetics, which define a graphical view onto the drawing, and constraints, which take the semantic of the graph into account. Examples for aesthetics were already given above. One example for a constraint is the need to put specific vertices into the center of the drawing, like the root vertex of a tree. A second constraint is given through clustering: there could be the need to place groups of vertices closer together. Afterwards they give a list of drawing algorithms and the aesthetics and constraints they follow.

In [5], Bridgeman and Tamassia analyze how a good drawing in interactive environments can be found. If a human moves one vertex within a drawing and an algorithm is to position this vertex according to the input and the system’s rules (for example aligning the edges to a rectangle grid structure), the output should resemble the input. They further define six categories of difference metrics. With these metrics, the difference between two drawings of the same graph can be measured.

Fulfilling these purposes might be a different task. The most prominent example is the minimization of edge crossings. With an increasing number of vertices, the number of edges also grows, and this also increases the number of edge crossings. In [13], Garey and Johnson show that the minimization of crossing edges is a NP-complete problem. A lot of algorithms try to minimize the crossings through different techniques, which differ in both time complexity and efficiency of removed crossings.

3.2 Classification

Three classes of graph drawing algorithms can be identified which can hold all of the considered algorithms. They focus onto different aspects of an aesthetical drawing and produce very different layouts. Inspirations to clarify the characteristics were given by Tamassia et al.[28].

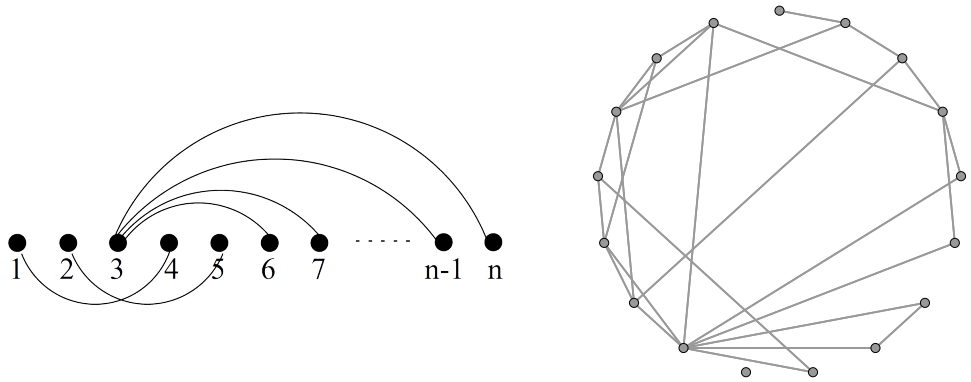
In the first two classes, the algorithms work in a very geometric way to produce a drawing of a chosen layout. Drawing different graphs with the same algorithm gives similar drawings. Drawings from the third class achieve aesthetical layouts as in most cases the vertices are well distributed (i.e. the space needed for the drawing is good utilized), but these drawings are less similar as the layouting is more flexible.

Fixed-vertex drawing algorithms

In fixed-vertex drawing algorithms, a set of coordinates that will be used for the vertices is calculated initially based on the number of vertices, but without deeper knowledge about the structure of the graph. In the next step, the algorithm will determine a mapping of the vertices to the coordinates through ordering the vertices. A good metric to determine the order is the number of crossing edges. Minimizing crossings, through reordering the vertices, increases the readability.

The basic layout is chosen initially: all vertices may be arranged on a single line. Edges between the vertices are either drawn over or under that line. In other algorithms the vertices are arranged such that they can be seen as a circle, and edges are drawn with straight lines or arches.

Two graphs with little differences (for example one graph has one more edge or vertex than the other) will have very similar drawings as the base layout (a straight line or a circle) will not be changed. Figure 3.2 gives examples for both types of fixed-vertex drawings.



(a) Straight-line fixed-vertex drawing (taken from [6]) **(b)** Circular fixed-vertex drawing

Figure 3.2.: Example of fixed-vertex drawings

Hierarchical drawing algorithms

Hierarchical drawing algorithms were especially designed to draw hierarchical graphs like rooted trees. These algorithms arrange the vertices in respect to a layout that is chosen initially. In contrast to the first class, the abstract layout is chosen (for example, a tree as in Figure 3.3). But a hierarchical algorithm can not calculate coordinates initially by simply counting vertices. These algorithms do more positioning of vertices, and some also contain iterative algorithms for a later movement of already positioned vertices to gain free space for other vertices.

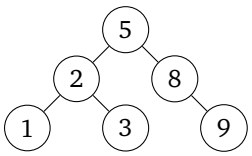
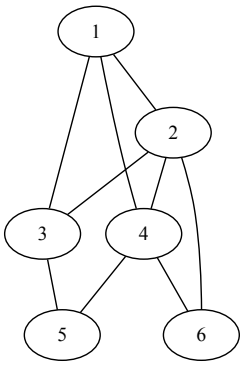


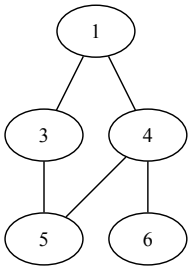
Figure 3.3.: Example tree drawing

Two very similar graphs will still have very similar drawings as the common structure of the drawing is set. In Figure 3.4a, the whole graph is drawn by Graphviz's `dot`[1] which draws a hierarchical algorithm. The drawing in Figure 3.4b shows the graph after vertex 2 has been removed. A common layout is obvious here.

But the difference between the drawing of two similar graphs in this class will be more obvious than in the class of fixed-vertex drawing algorithms. Even while a common structure for positioning the vertices is set, an algorithm could produce a completely changed positioning of the vertices such that the visual alignment is changed.



(a) Initial graph



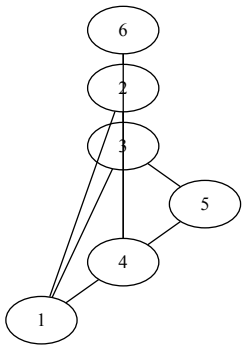
(b) Drawing of the graph where vertex 2 has been removed

Figure 3.4.: Example drawing using GraphViz's `dot` for hierarchical drawings

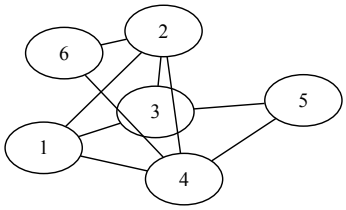
Force-driven drawing algorithms

Peter Eades[9] introduces an heuristic which forms the basis for all later so-called force-driven algorithms. These algorithms work iteratively and enhance the drawing in each step. Figuratively, all vertices are replaced by steel rings and all edges by springs. In each step of the calculation, the spring forces are determined and the vertices are moved in respect of the power of these forces. Vertices will be well distributed in these drawings, as these forces will move adjacent vertices close to each other and increase the distance to the other vertices.

These algorithms might take a longer time to layout. Each movement of a vertex, caused by the force of one edge, will change the forces of other edges that are connected to the same vertex. Eades states that almost all graphs he analyzed got into a stable state (a state that does not differ from the previous state) after 100 iterations. Figure 3.5 shows an example for this refinement: an initial layout is randomly determined, and with each iteration, the drawing is improved. Later work onto algorithms in this class is not only done to produce better drawings, but also to achieve a stable state faster.



(a) Drawing after 10 iterations

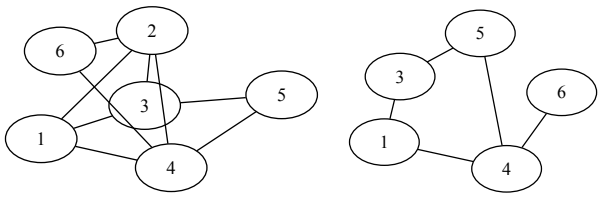


(b) Drawing after 10,000 iterations

Figure 3.5.: Example drawing using GraphViz's `fdp` for force-driven drawings

As each vertex has an influence onto its neighborhood, adding or removing vertices might produce very different drawings. Figure 3.6a shows a whole graph drawn by Graphviz's `fdp` which uses a force-driven algorithm. The drawing in Figure 3.6b shows the graph after vertex 2 has been removed.

[1] `http://www.graphviz.org/`



(a) Initial graph

(b) Drawing of the graph where vertex 2 has been removed

Figure 3.6.: Example drawing using GraphViz's `fdp`

In contrast to the first two classes, there are no deterministic positions for the vertices. Generally, drawings will neither resemble commonly known structures like trees or circles. Algorithms from this class are well suited to draw even very large, dense graphs without using overproportional amounts of space.

3.3 Fixed-vertex drawing algorithms

3.3.1 Canonical circular algorithm (CCC)

In a circular drawing, the vertices should be distributed uniformly on the circumference. The positioning of vertices is based on the minimization of edge crossings. A simple layouting algorithm is presented in Algorithm 1. The vertices are ordered randomly at the beginnning. Afterwards, neighboring vertices will be swapped as long as this decreases the number of crossings.

Algorithm 1 Canonical circular algorithm

```
    function LAYOUT(G)
        todoList := V
        while |todoList| > 0 do
            v := GETANDREMOVEFIRST(todoList)
 5:         w := GETPREDECESSOR(v)  { see algorithm 10 on page 44 }
            currentCrossings := GETCROSSINGSFOREDGES(v, w)
            SWAPPOSITIONS(v, w)  { see algorithm 10 on page 44 }
            if GETCROSSINGSFOREDGES(v, w) < currentCrossings then
                todoList := todoList ∪ {v,w}
10:         else
                SWAPPOSITIONS(v, w)
            end if
        end while
    end function
```

3.3.2 Six/Tollis (ST)

In [26], Six and Tollis present a very sophisticated algorithm for circular drawings. It works in two phases: the first will order the vertices such that the number of edges appearing on the circumference is maximized. After this phase, the number of edges crossing within the circle is already much smaller than within a random order. The second phase further reduces the number of crossings by swapping vertex positions.

Six and Tollis use some special terms for the first phase. A *pair edge* connects two vertices which share a common third vertex, and the pair edge is established by this third vertex. A *triangulation edge* is a new created pair edge that will be added to the graph during the algorithm. Vertices can be sorted into two subgroups. *Wave front vertices* are adjacent to the last processed vertex, and *wave center vertices* are adjacent to any already processed vertex.

The first phase iterates over the vertices similar to a breadth-first walk. A vertex to process is chosen in a wave-like style from the list of wave front vertices, or (if there are no wave front vertices that have not yet been processed) from the list of wave center vertices, or (if both former lists are empty) randomly from the list of all vertices. In each case the lists are ordered by the degrees such that one of the lowest degree vertices is chosen. For the chosen vertex, the pair edges are searched. New triangulation edges, will be inserted, if the number of pair edges is less than the degree of the vertex minus one[2]. Before processing the next vertex, the current one is removed from the graph. The triangulation edges help to keep the graph connected.

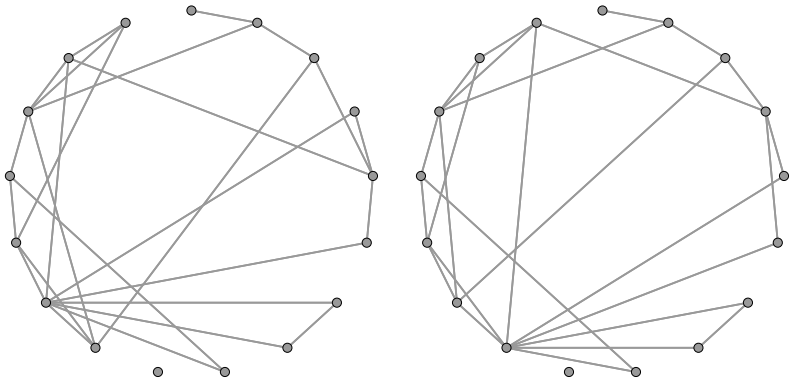
After all but the last four vertices are removed from the graph, the original graph is restored (which means that all triangulation edges that were introduced are removed and all vertices are restored). The pair edges that were found are now removed from the graph. Using a depth-first walk, the longest path in the graph is searched. It will be placed on the circumference. Each remaining node will be placed between its two neighbors, next to one of its neighbors or at a random position.

The second phase in our implementation is identical to Algorithm 1 shown in Section 3.3.1. Another approach is used in the original algorithm. To further reduce crossings, it is checked whether the movement of a vertex to one of its neighbors reduces the number of crossings caused by that vertex. This might be more effective than our approach, but is also more complex and thus takes even more time.

[2] Baur did simulations regarding the triangulation edges in his diploma thesis[3, page 49ff]. He compared layout results without triangulation edges, with the given algorithm and with two other ways of inserted edges, and found the given way to be the most effective one.

Algorithm 2 Fixed-vertex algorithm by Six and Tollis

```
    function LAYOUT(G)
        removedVertices := ∅
        waveCenterVertices := ∅
        vertexList := SORT(V) { Sort vertices in descending degree order }
 5:     additionalEdges := array(0,...,|V|) of edges
        removalList := ∅

        for i = 1,...,|V| − 3 do
            currentVertex := CHOOSEVERTEX
10:         pairEdges := GETPAIREDGES(currentVertex) { see algorithm 11 on page 44 }
            removalList := removalList ∪ pairEdges
            CREATETRIANGULATIONEDGES(currentVertex) { see algorithm 11 on page 44 }
            waveFrontVertices := A(currentVertex)
            waveCenterVertices := waveCenterVertices ∪ A(currentVertex)
15:     end for

        longestPath := LONGESTPATH { see algorithm 11 on page 44 }
        PLACEREMAININGVERTICES(longestPath) { see algorithm 11 on page 44 }
        for i := 0,...,|longestPath| do
20:         longestPath[i].position := i
        end for
    end function

    function CHOOSEVERTEX
25:     { All three lists need to be sorted in ascending degree order, such that always a node with lowest degree from these
    lists is returned }
        for all u ∈ waveFrontVertices do
            if u ∉ removedVertices then return u
            end if
        end for
30:     for all u ∈ waveCenterVertices do
            if u ∉ removedVertices then return u
            end if
        end for
        for all u ∈ vertexList do
35:         if u ∉ removedVertices then return u
            end if
        end for
    end function
```



(a) Graph after first phase (36 crossings) **(b)** Final graph (18 crossings)

Figure 3.7.: Example drawing of Six/Tollis

3.4 Hierarchical drawing algorithms

3.4.1 Knuth

Donald E. Knuth proposes an algorithm to produce trees in [19]. It is designed to find and draw a binary search tree for a set of words. As the original algorithm is only working on binary trees with at most two child vertices per vertex, we present a slightly modified version in Algorithm 3 that can cope with an unlimited amount of child vertices per vertex.

The algorithm uses a single depth-first walk to compute the coordinates. The vertical coordinate is taken from the depth of the vertex in the tree. In horizontal direction, a counter is used such the first processed vertex is put at position 0, the second at position 1 and so on. Thus, each horizontal position is only used once. An example drawing can be found in Figure 3.8a which shows a downside of this simple algorithm: A drawing never looks aesthetically pleasing as child vertices always get placed shifted compared to their parent.

Algorithm 3 Hierarchical algorithm by Knuth

```
    function LAYOUT(G)  { G needs to be a spanning tree for this algorithm }
        rootVertex := G.root
        nextPosition := 0
        WALK(rootVertex, 0)
 5: end function

    function WALK(v, height)
        if d+(v) = 0 then
            v.X := nextPosition
10:         v.Y := height
            nextPosition := nextPosition + 1
        else
            positionOfRoot := |children| / 2
            for all w ∈ A+(v) do
15:             if positionOfRoot = 0 then
                    v.X := nextPosition
                    v.Y := height
                    nextPosition := nextPosition + 1
                end if
20:             positionOfRoot := positionOfRoot - 1
                WALK(w,height + 1)
            end for
        end if
    end function
```

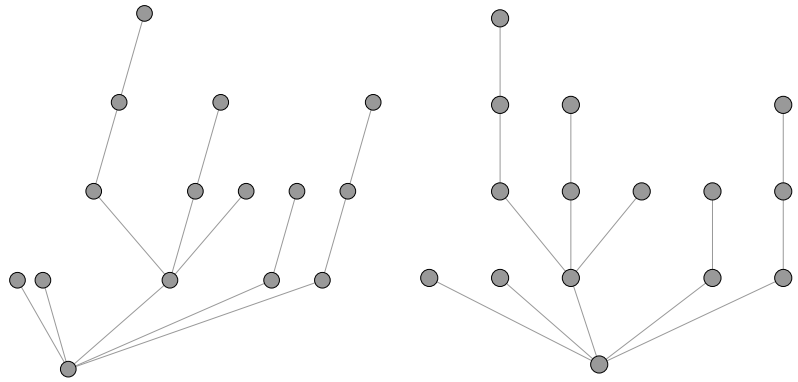
3.4.2 Wetherell/Shannon (WS)

In [32], Wetherell and Shannon propose an algorithm to draw rooted trees in 1979. Initially they discuss problems and aesthetics they want to cope with in their algorithm. The output should respect physical limits as the output is limited in at least one dimension (e.g. the width of paper used by a printer with continuous paper), if not in both as for displaying on a computer monitor. Vertices at the same height (i.e. that have the same number of vertices between them and the root and thus lay on the same layer) should lie along a straight line, and these lines should be parallel. Finally a "parent vertex" should be centered over its "children".

The authors do not only present the final algorithm with annotations, they also show the way they took to reach it. But once more, one important point is left out that is necessary for our purpose: The given algorithm is only working on binary trees. It works in an iterative way where a pointer always points to the currently processed vertex, and holds a ternary status flag for each vertex to remember its current state. In Algorithm 4, we present a modified algorithm that uses a recursive approach and better clarifies the functioning of the algorithm than the original algorithm. It lacks the status flag, but works in the same way.

In a first post-order walk, a provisional positioning of all vertices, relative to their parent vertex, is done. For each level of the tree, two arrays store positioning information; $nextPos$ holds the next free position, starting at 0. After a provisional value for the horizontal position of the currently processed vertex is found, this value is set to the next position right of this vertex. The second array $heightModifier$ holds an additional horizontal shift. As each vertex should be centered above its children (see lines 19 to 26), subtrees might need to be drawn to the right. The $nextPos$ array can only store the next free position in each level, but it will not know anything about the upper level. If the calculated horizontal position is too far left, whichs might be the case if a node that is already positioned far right in the graph has a child that is the first in its level, the $heightModifier$ will be increased to handle this shift in the second walk. Otherwise, a parent node would be drawn too far to the left side by its children. Each vertex will hold the value of $heightModifier$ that was accumulated at processing the vertex.

The second pre-order walk will then compute the final absolute coordinates based on the information gathered in the first walk. The modifiers of the vertices along a way from the current vertex to the root vertex are summed up and added to the horizontal position of the vertex. The vertical position is solely based on the height in the tree.



(a) Knuth **(b)** Wetherell/Shannon

Figure 3.8.: Example drawings of Knuth and Wetherell/Shannon

Algorithm 4 Hierarchical algorithm by Wetherell and Shannon

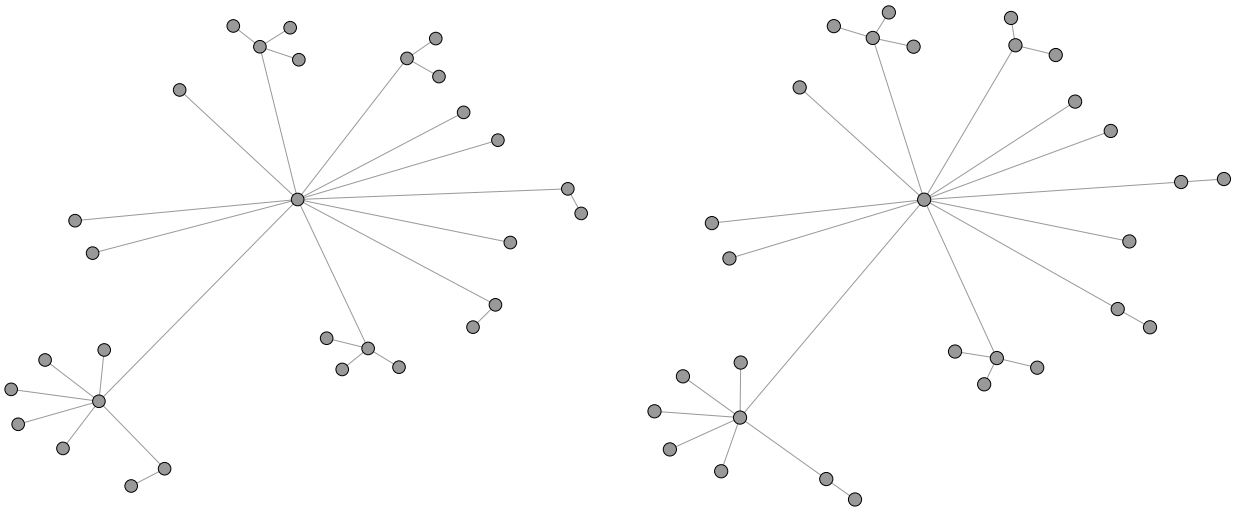
```
    function LAYOUT(G)  { G needs to be a spanning tree for this algorithm }
        rootVertex := G.root
        heightModifiers := array[0..maxHeight] of integer
        nextPosition := array[0..maxHeight] of integer
 5:     for i = 0,...,maxHeight do
            heightModifiers[i] := 0
            nextPosition[i] := 0
        end for
        FIRSTWALK(rootVertex, 0)
10:     modifierSum := 0
        SECONDWALK(rootVertex, 0)
    end function

    function FIRSTWALK(v, height)
15:     for all w ∈ A+(v) do
            FIRSTWALK(w, height + 1)
        end for
        place := 0
        if d+(v) > 0 then
20:         for all w ∈ A+(v) do
                place := place + w.X
            end for
            place := place / d+(v)
        else
25:         place := nextPosition[height]
        end if
        heightModifiers[height] := max ( heightModifiers[height], nextPosition[height] − place )
        if d+(v) > 0 then
            v.X := place + heightModifiers[height]
30:     else
            v.X := place
        end if
        nextPosition[height] := v.X + 2
        v.Modifier := heightModifiers[height]
35: end function

    function SECONDWALK(v, height)
        v.X := v.X + modifierSum
        modifierSum := modifierSum + v.Modifier
40:     v.Y := 2 * height + 1
        for all w ∈ A+(v) do
            SECONDWALK(w, height + 1)
        end for
        modifierSum := modifierSum - v.Modifier
45: end function
```

3.4.3 Melançon/Herman (MH)

Melançon and Herman describe their algorithm for a circular tree drawing in [20]. The root vertex is placed in the center and child vertices are placed on a circle around their parent vertex recursively. As given in Algorithm 5, it works in two phases. The first computes angular sectors, relative to its parent vertex, and radii for each vertex. Afterwards, the second phase computes absolute positions based on the angular sectors.

The first phase of the algorithm uses a depth-first walk to compute angular sectors in half angles, as the vertex should be positioned in the middle of the angle, that each vertex may use for itself and its child vertices, and a radius for their parent vertex. Two additional scaling factors are computed by ADJUSTCHILDREN: If the sum of angular sectors is larger than π, which means that the whole space around a vertex is used for its children, the angles will be scaled down in the second walk. Otherwise, the remaining angular sector will be assigned equally to all children's angles. The second walk computes the absolute positions for all vertices. An example drawing can be found in Figure 3.9 on page 15.

Looking at the drawings by an exact implementation of this algorithm, we found a minor error. In line 27, the remaining angular sector $v.f$ is divided by the number of child vertices of v. The original algorithm adds 1 to the denominator without giving an explanation for this. We assume this takes an edge to the parent vertex into account, but this has a negative impact on the drawings. Figure 3.9 compares the exact implementation with the minimally changed one which omits the additional 1. Differences can be seen in the rotation of subtrees, especially those which have only one child vertex.



(a) Drawing by our implementation of the original algorithm

(b) Drawing by the minimally changed implementation

Figure 3.9.: Example drawings of the original Melançon/Herman and a minimally changed algorithm

Algorithm 5 Hierarchical algorithm by Melançon and Herman

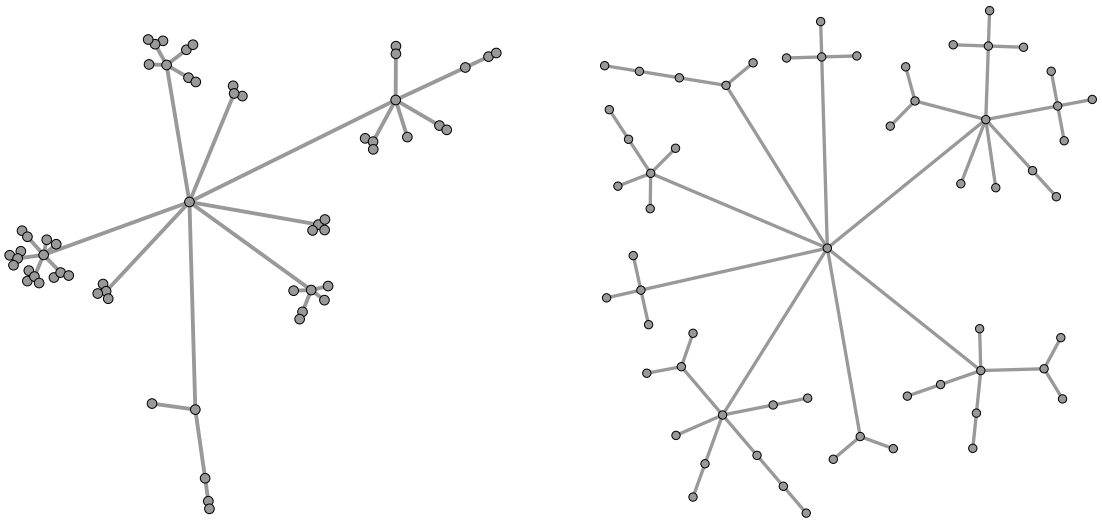
```
    function LAYOUT(G)  { G needs to be a spanning tree for this algorithm }
        rootVertex := G.root
        FIRSTWALK(rootVertex)
        SECONDWALK(rootVertex, 0, 0, 1, 0)
 5: end function

    function FIRSTWALK(v)
        v.d := 0
        s := 0
10:     for all w ∈ A⁺(v) do
            FIRSTWALK(w)
            v.d := MAX(v.d, w.r)
        end for
        for all w ∈ A⁺(v) do
15:         w.α := ARCTAN(w.r / (v.d + w.r))
            s := s + w.α
        end for
        ADJUSTCHILDREN(v, s)  { see algorithm 12 on page 46 }
        v.r := MAX(v.d, leafRadius) + 2 * v.d  { leafRadius denotes a minimal radius for a leaf vertex }
20: end function

    function SECONDWALK(v, x, y, λ, θ)
        v.X := X
        v.Y := Y
25:     d' := λ * v.d
        φ := θ + π
        freespace := v.f / |A⁺(v)|
        previous := 0
        for all w ∈ A⁺(v) do
30:         α' := v.c * w.α
            r' := v.d * TAN(α') / (1 − TAN(α'))
            φ := φ + previous + w.α + freespace
            w.X' := (λ * r' + d') * COS(φ)
            w.Y' := (λ * r' + d') * SIN(φ)
35:         previous := w.α
            SECONDWALK(w, w.X' + v.X, w.Y' + v.Y, λ * r' / w.r, φ)
        end for
    end function
```

3.4.4 Bubbletree (BT)

A huge disadvantage of the algorithm of Melançon and Herman is the scaling. As the length of edges decrease exponentially with growing depth, their algorithm is not suitable for drawing larger trees. On the other hand, this is a simple way to avoid overlapping of subtrees. An example can be found in Figure 3.10.

In [15], Grivet et al. present Bubbletree, a different algorithm for circular drawings. It computes smallest enclosing circles for each subtree, and chooses their radii just that small that all subtrees find place. The rest of the algorithm works similar to Melançon/Herman. There are two more differences that need to be mentioned: the method CALCULATEANGULARSECTOR assigns the angular sectors to child vertices based on the sum of their radii. The largest subtree may use at most half the circle; otherwise, space would be wasted.

The second difference is the rotation of subtrees. The first walk computes coordinates of subtrees relative to their common parent vertex starting with an angle of 0° pointing horizontally to the right side. In COORDASSIGN, the final coordinates are assigned in a second walk. This starts with the calculation of a rotation angle, such that the zero angle of the subtree equals the angle of the edge from the center to the parent vertex.



(a) Melançon/Herman **(b)** Bubbletree

Figure 3.10.: Example drawings of Melançon/Herman and Bubbletree

Algorithm 6 Hierarchical algorithm Bubbletree by Grivet et al.

```
    function LAYOUT(G)  { G needs to be a spanning tree for this algorithm }
        WALK(G.root)
        G.root.x := 0; G.root.y := 0
        COORDASSIGN(G.root)
 5: end function

    function WALK(v)
        for all w ∈ A+(v) do
            WALK(w)
10:     end for
        if d+(v) = 0 then
            v.r := leafRadius  { leafRadius denotes a minimal radius for a leaf vertex }
        else
            CALCULATEANGULARSECTOR(v)
15:         Θ := 0
            for all w ∈ A+(v) do
                δ := MAX(leafRadius + w.r, w.r / sin(w.θ/2))
                Θ := Θ + w.θ
                w.x' := δ*COS(Θ − w.θ/2)
20:             w.y' := δ*SIN(Θ − w.θ/2)
            end for
            v.freeAngle := 2π − Θ
            v.r := 1.3*SMALLESTENCLOSINGCIRCLE(v)  { see algorithm 13 on page 47 }
        end if
25: end function

    function CALCULATEANGULARSECTOR(v)
        sumRadii := leafRadius  { leafRadius denotes a minimal radius for a leaf vertex }
        for all w ∈ A+(v) do
30:         sumRadii := sumRadii + w.r
        end for
        largestSon := null
        for all w ∈ A+(v) do
            if w.r > sumRadii/2 then
35:             largestSon := w
                w.θ := π
                sumRadii := sumRadii − w.r
            end if
        end for
40:     for all w ∈ (A+(v) \ largestSon) do
            w.θ := w.r / sumRadii * 2π
        end for
    end function

45: function COORDASSIGN(v)
        if HASPARENT(v) then
            rotation := ∢(v, parent(v)) + v.freeAngle/2
        else
            rotation := 0
50:     end if
        for all w ∈ A+(v) do
            w.x := w.x' * cos(rotation) − w.y' * sin(rotation) + v.x
            w.y := w.x' * sin(rotation) + w.y' * cos(rotation) + v.y
            COORDASSIGN(w)
55:     end for
    end function
```

3.5 Force-driven drawing algorithms

3.5.1 Fruchterman/Reingold (FR)

In his work [9] from 1984, Peter Eades was the first to use a heuristic, force-driven approach in graph drawing. Metaphorically, the vertices of a graph are replaced by steel rings and the vertices by springs such that the graph forms a mechanical system. On releasing the steel rings, the springs move the rings, either together or apart, until the springs reach a position of low energy. Adjacent vertices appeal each other through the connecting spring and nonadjacent vertices are to repel each other. Eades' corresponding algorithm calculates the forces and the following movement according to the force iteratively. Hence, the algorithm needs some iterations until a stable state is reached such that no more (or at most very little) movement is done in following iterations. The vertices may be positioned randomly at the beginning without any negative impact onto the algorithm.

Eades leaves much place for interpretation. He gives formulas for the two forces containing constants, but does neither explain how different values for these constants affect the drawing nor give a hint about how to weigh the two forces to calculate one resulting force value for each vertex.

Fruchterman and Reingold give a more precise algorithm in [12], based on Eades work. They postulate two principles which they base their algorithm on: Vertices connected by an edge should be drawn near each other and vertices should not be drawn *too* close to each other.

The resulting algorithm (see Algorithm 7) consists of three steps for each iteration. In the first step, repulsive forces are calculated. Afterwards, attractive forces are calculated based on the set of edges. Finally, the new position of each vertex is calculated. In the worst case a very simple force-driven algorithms could run endless, as in each step some movement is done. Oscillating vertices that swap positions repeatedly could prevent the termination. Using a "temperature" (as here in line 34) that is decreased in each step can overcome this, as the movement of a vertex is limited to the temperature. In the beginning, a relatively large temperature is chosen such that the movement is done in large steps. As the temperature cools down in later steps, the movement slows down.

The value k (line 2) depicts the optimal distance between vertices. It is based on the n-dimensional euclidean space in which the vertices should be placed, and is calculated such that the vertices are uniformly distributed over the available space. Fruchterman and Reingold experimented with several formulas for the repulsive force f_r and the attractive force f_a (for example without squaring the numerator or with logarithmic attractive forces like Eades proposed it), and chose the two formulas as they are more simple to compute than logarithmic formulas and show good characteristics in overcoming local minima in forces.

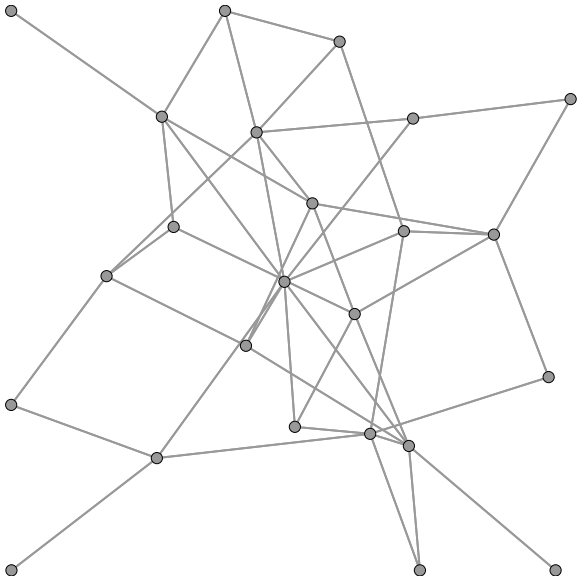


Figure 3.11.: Example drawing of Fruchterman/Reingold

Algorithm 7 Force-driven algorithm by Fruchterman and Reingold

$space := \prod_{i=1}^{\#dimensions} moduli$
$k := \sqrt[\#dim]{\frac{space}{|V|}}$ { *optimal distance* }

function $f_a(x)$ { *Attractive force* }
    return x^2/k
end function

function $f_r(x)$ { *Repulsive force* }
    return k^2/x
end function

function LAYOUT(G)
    for $i = 1, \ldots, iterations$ **do**
        { *Calculate repulsive forces between all vertices* }
        for all $v \in V$ **do**
            $displacement[v] := 0$
            for all $u \in V$ **do**
                if u != v **then**
                    $\Delta := v.position$ - $u.position$
                    $displacement[v] := displacement[v] + \frac{\Delta}{|\Delta|} * f_r(|\Delta|)$
                end if
            end for
        end for

        { *Calculate attractive forces for all edges* }
        for all $e \in E$ **do**
            $\Delta := e.source.position$ - $e.destination.position$
            $displacement[e.source] := displacement[e.source]$ - $\frac{\Delta}{|\Delta|} * f_a(|\Delta|)$
            $displacement[e.destination] := displacement[e.destination] + \frac{\Delta}{|\Delta|} * f_a(|\Delta|)$
        end for

        { *Calculate new coordinates for each vertex* }
        for all $v \in V$ **do**
            $v.position := v.position + \frac{displacement[v]}{|displacement[v]|} * \min(|displacement[v]|, t)$
        end for

        $t := cool(t)$
    end for
end function

4 Distributed routing algorithms

The purpose of the embeddings from Section 3 is the assignment of coordinates to a graph's vertices to enable routing (finding a path from one vertex to another with which messages can be transported) on these graphs. We use three routing algorithms to evaluate the routing performance. As a distance metric between two vertices, the algorithms use euclidean distances.

In contrast to large precomputed routing tables where each vertex knows about the best next hop to reach a given destination, we use distributed routing algorithms which have only local knowledge, such that each vertex knows its neighbors and their coordinates, but do not know all vertices' coordinates.

Finding a greedy embedding as described by Papadimitriou and Ratajczak in [22] would be optimal. In such an embedding, greedy routing always succeeds. In [24], CAN is presented as a network where connections between vertices are drawn such that a greedy embedding is formed. The embeddings the analyzed graph drawing algorithms calculate are no greedy embeddings, so we also simulate routing using more sophisticated algorithms.

4.1 Greedy routing

At each routing step of greedy routing, the distances from all neighbor vertices of the current vertex to the routing destination are calculated. The message is forwarded to the vertex in closest distance to the destination. The routing attempt might stop if no next hop can be found. Errors in the hop choice cannot be undone. As greedy routing is the simplest routing algorithm to implement and the fastest to run even in large networks, reaching high success rates here is an indicator for a good embedding.

4.2 Greedy routing with backtracking

The second routing algorithm we used is an alternative greedy routing algorithm with backtracking (GB). If at any vertex a dead end is reached, because all outgoing vertices enlarge the distance to the routing destination, the search for a route continues at the preceding vertex that routed to this dead end vertex. The final routing path contains also these indirect paths. To avoid endless long paths, a *time to live* value (TTL) is used. Routes may not be longer than this value. With an infinite large TTL value, all routes would be successful as this resembles a random walk.

4.3 Lookahead routing

Using a neighbor lookahead list (LS[1]), a route along a non-greedy path with less hops than using other algorithms can be found. A vertex does not only know the coordinates of adjacent vertices, but also of the vertices that are adjacent to them. With this knowledge a packet is not forwarded to the vertex that minimizes the distance to the destination, but to the vertex from which such a minimizing vertex can be reached. In our evaluation, we use an approach where lookahead lists are generated during the routing and not beforehand. This leads to higher runtimes, but enables us to simulate more routing attempts in parallel as the memory consumption is much lower.

4.4 Route lengths and runtimes

Table 4.1 shows some characteristic data to compare these routing algorithms. They are based on a random embedding into a plane twodimensional identifier space. Each data set is based on 60 simulations on each graph configuration and each simulation runs five attempts per vertex to a random destination vertex. Runtimes are given in seconds as the average runtime for one simulation run (not for the simulation of a single route).

[1] abbr. for `LookaheadSimple`

Graph type	Size	Greedy routing (ttl: 100 hops)				
		max	avg	med	success	runtime
ER	1,000	4.44	1.84	1.8	1.16%	0.08
	5,000	4.68	1.86	1.84	0.23%	0.1
	10,000	4.72	1.86	1.82	0.12%	0.13
BA	1,000	4.08	1.98	2	36.68%	0.14
	5,000	4.06	2	2	35.03%	1.39
	10,000	4	2	2	34.48%	6.15
CAIDA	20,057	5.24	2.24	2	1.23%	1.23
sPI	11,407	5.66	2.01	2	10.31%	4.11
WOT	25,487	5.62	2.26	2	0.28%	0.72

Graph type	Size	Greedy backtracking (ttl: 100)				
		max	avg	med	success	runtime
ER	1,000	99.825	32.27	24.98	7.76%	0.13
	5,000	99.83	34.95	27.75	1.67%	0.44
	10,000	99.88	34.82	27.68	0.83%	0.86
BA	1,000	100	19.76	2.18	69.48%	1.29
	5,000	100	14.37	2	47.16%	41.19
	10,000	100	10.22	2	41.69%	175.6
CAIDA	20,057	100	27.57	14.9	4.17%	32.18
sPI	11,407	100	15.38	2.05	21.2%	94.29
WOT	25,487	100	39.92	34.85	2.93%	28.89

Graph type	Size	Lookahead list (ttl: 50 hops)				
		max	avg	med	success	runtime
ER	1,000	10.28	3.6	3.15	10.41%	0.1
	5,000	10.9	3.71	3.28	2.24%	0.31
	10,000	10.98	3.74	3.45	1.13%	0.55
BA	1,000	3.18	1.98	2	100%	2.49
	5,000	3.5	2	2	100%	34.45
	10,000	3.84	2	2	100%	137.05
CAIDA	20,057	8.2	3.53	3	60.28%	162.28
sPI	11,407	11.88	2.5	2	62.57%	209.44
WOT	25,487	11.23	3.95	4	14.89%	162.82

Table 4.1.: Routing characteristics, using a random plane identifier space

5 Implementation

As a basis for this work, the framework *GTNA*[25], written in Java, is used to generate graphs, embed them using the graph drawing algorithms and simulate routing. Algorithms for the generation of random graphs and the simulation of routing were already present.

Matching identifier spaces are implemented for each of the graph drawing classes into which the drawing algorithms can be seperated. The `RingIdentifierSpace` implements an identifier space over one-dimensional identifiers in the range $[0, n)$, where the distance is measured as the difference between two identifiers. A two-dimensional identifier space is implemented in the class `PlaneIdentifierSpaceSimple` and contains identifiers in the range $[0, x) * [0, y)$. An `MDIdentifierSpaceSimple` can be used for a n-dimensional identifier space where coordinates range within $[0, n_i)^i$. Distances are measured as the euclidean distance between two identifiers. All identifier spaces consume moduli to define the upper limits for identifiers. On this basis, random identifiers can be created. Their distribution within the identifier space is not restricted in any other way.

The drawing algorithms are implemented as transformations of `Graph` objects, which adds properties like an identifier space for routing to the graph. `GraphDrawingAbstract` is the common super class of all graph drawing algorithms. As abstract methods, it defines the methods that need to be implemented. For each of the three classes of drawing algorithms that were identified in 3.2, an abstract class contains methods that are useful for all algorithms within that class of algorithms, like the method `initIDSpace` for the initialization of a random identifier space in force-driven drawing algorithms. After the actual graph drawing algorithm computed the coordinates, the common method `writeIDSpace` adds an according property to the graph containing the resulting identifier space.

`CircularAbstract` is the abstract class for the circular algorithms of the fixed-vertex class. It holds the edge crossing algorithm CCC which is used as a single algorithm or as the second phase for Six/Tollis. `HierarchicalAbstract` is the abstract class for the hierarchical algorithms and contains the transformation from a simple coordinate array used in the drawing algorithms to coordinates within a plane identifier space. The abstract class `ForceDrivenAbstract` for force-driven drawing algorithms holds a method to convert coordinates. The components of the coordinates might be negative, which causes problems in routing and plotting of such embeddings. Thus, the algorithms work internally on a identifier space within the range of $[-\frac{n_i}{2}, \frac{n_i}{2})$ which is shifted linearly to $[0, n_i)^i$.

The actual drawing algorithms are implemented as subclasses in which the method `transform` contains the actual algorithm.

To visualize embedding results, we implemented a connector to *Gephi*[2], an open-source graph visualization tool written in Java. It allows us to plot embeddings as PDF or SVG vector graphics. Additional decorators for this connector can colorize or remove vertices using property filters.

6 Evaluation

The following subsections contain the evaluation of our simulations. In subsection 6.1, the setup is explained. Characteristic of the graph types and runtimes for the graph drawing algorithms are given here. Subsection 6.2 covers an overview over the routing performance within each of the three classes of graph drawing algorithms. Afterwards, peculiarities are highlighted. Subsection 6.3 handles the choice of a root vertex for hierarchical drawing algorithms. In subsection 6.4 the long runtime of the circular algorithm by Six and Tollis in scale-free networks is analyzed. In Subsection 6.5, the routing performance in random embeddings is analyzed. Finally, Subsection 6.6 covers the routing performance on Barabási-Albert graphs.

6.1 Analyzed graphs

To evaluate the routing perfomance of the embeddings, we use three different kinds of networks: one random graph model, one scale-free graph model and three existing real world networks which are good examples for later applications of our work.

- Using the Erdös-Rényi model (ER) described in [10], a homogeneous network can be generated. A given set of vertices is connected randomly such that a given average degree of all vertices is reached. Using these construction model, the generated graph must not be connected. We use multiple runs containing 1,000, 5,000, and 10,000 vertices with an average degree of 10.
- Using the Barabási-Albert model (BA) presented in [1], one can generate a random scale-free network where the degree distribution follows a power law such that a small fraction of vertices has a high degree, and most vertices have a small degree. This distribution can be observed in existing networks like the Internet or social networks. We use multiple runs containing 1,000, 5,000, and 10,000 vertices with a minimal degree of 10.
- The Cooperative Association for Internet Data Analysis (CAIDA)[7] monitors the connectivity of autonomous systems. An autonomous system is a collection of subnets that form the network of an Internet service provider or a large organization. CAIDA monitors 30,000 systems and generates daily snapshots of the topology of the Internet. The graph we use contains 20,000 vertices and 41,000 undirected edges, dated 06 december 2011.
- A social graph has been extracted from the Studentenportal Ilmenau (sPI)[1] which is a social network for the students of TU Ilmenau similar to Facebook. We use a graph containing 11,500 vertices and 80,000 undirected edges, dated august 2010.
- The OpenPGP Web of Trust (WOT) consists of people who want to secure their digital communication. Each participant owns a certificate containing an encryption key and signatures of other participants. Through an assurance process, one can sign other certificates and thus express the trust in the certificate's owner. A connected, undirected graph of these trust connections has been extracted, and we use a graph containing 25,000 vertices and 300,000 edges, dated 25 february 2005.

Some of these graphs are undirected, for example the Web of Trust where one party signs another's certificate, but these other party does not necessary sign the first certificate. As the graph drawing algorithms we took into account work onto all graphs as if they were undirected, we replace all edges by undirected edges to ease their handling.

Furthermore, the graph might be disconnected. This is a problem for most graph drawing algorithms as only one partition of the graph will be embedded. As disconnected vertices also lead to problems in the following routing, we use only the largest connected component of each graph. This reduces the number of vertices by less then 1% for graphs generated by the Erdös-Rényi model, 0.13% for the CAIDA graph and 0.46% for sPI. Graphs generated by the Barabási-Albert model are always connected, so they all have the desired size. Table 6.1 shows characteristic numbers for all used graphs. The shortest path length gives a lower bound for optimal routing results. The local clustering coefficient for a vertex v as described in [31] is described as the fraction of existing edges between the neighbors of v and the maximum number of edges that can be formed between these neighbors, and the given global clustering coefficient is the average value over all local clustering coefficients within one graph. The degree distribution is further visualized in Figure 6.1.

[1] http://spi.tu-ilmenau.de/

Graph type	Size	Degree distribution				Shortest path length			Clustering coefficient
		min	max	avg	med	max	avg	med	
ER	1,000	1	13.53	5.03	5	9.02	4.46	5	0.0049
	5,000	1	15.06	5.03	5	10.92	5.47	6	0.00094
	10,000	1	15.65	5.04	5	11.73	5.9	6	0.00046
BA	1,000	10	501.89	19.72	10	3	1.98	2	0.3796
	5,000	10	2,383.92	19.94	10	3	2	2	0.3803
	10,000	10	4,643.75	19.97	10	3	2	2	0.3774
CAIDA	20,057	1	2,135	3.96	1	10	3.83	4	0.2245
sPI	11,407	1	7,603	9.95	5	11	3.15	3	0.35
WOT	25,487	1	1,169	11.9	4	15	5.07	5	0.4665

Table 6.1.: Degree distribution, shortest path lengths, and clustering coefficient (for Erdös-Rényi and Barabási-Albert: averages over 100 graphs)

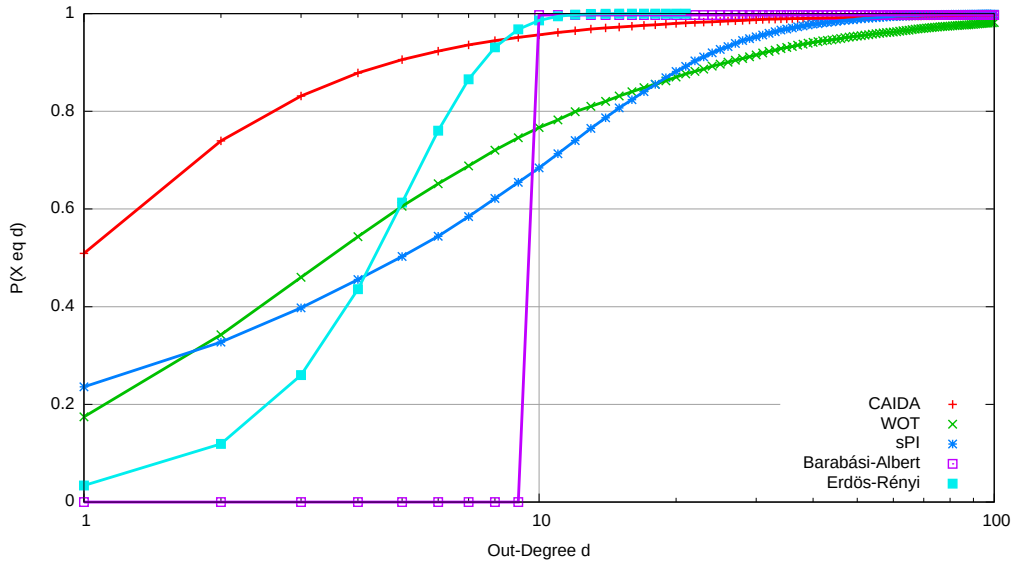


Figure 6.1.: Degree distribution in the used graphs

We generated multiple runs of 60 graph drawings with the seven graph drawing algorithms described in chapter 3, independently from the routing simulation covered in the next subsection. Table 6.2 shows the average runtimes of the drawing algorithms. As the first phase of Six/Tollis takes a lot of time to compute the initial positioning (see Section 6.4 for an analysis), we could only generate 13 embeddings for CAIDA and for WOT during the limited time of evaluation. As this gives no stable fundament for statistically significant routing simulations, these embeddings are left aside in the later evaluation.

Graph type	Size	CCC	ST	Knuth	WS	MH	BT	FR
ER	1,000	16.05s	5s	0.45ms	0.5ms	3.58ms	8.13ms	35.95s
	5,000	1.57min	2.63min	1.1ms	1.37ms	4.67ms	15.23ms	16.06min
	10,000	3.33min	18.5min	4.73ms	3.88ms	14.22ms	31.1ms	119.72min
BA	1,000	12.75min	11.12min	0.3ms	0.22ms	1.45ms	2.02ms	36.93s
	5,000	112.29min	277.5min	1.08ms	1.45ms	4.13ms	8.45ms	17.1min
	10,000	239.12min	827.66min	3.37ms	2.57ms	7.5ms	14.95ms	117.5min
CAIDA	20,057	39.87s	392.22min	4.67ms	5.22ms	18.08ms	34.52ms	387.52min
sPI	11,407	12min	164.04min	1.87ms	2.97ms	8.62ms	18.48ms	122.48min
WOT	25,487	22.2min	3164.26min	5.4ms	11.92ms	24.57ms	50.82ms	768.24min

Table 6.2.: Average runtimes for the graph drawing algorithms

6.2 Result overview

This section gives a brief overview over the routing performance within the different classes of graph drawing algorithms. We primarily present the results for greedy routing onto embeddings of Erdös-Rényi graphs with 10000 vertices. If not otherwise stated, the results within each class are similar to embeddings of other graphs. The complete results can be found in Appendix B (page 36 ff).

For each combination of the graph types with the drawing algorithms, we simulate routing with the three routing algorithms that are described in section 4. In a single routing simulation, five routing attempts from each vertex to a random destination vertex are simulated. To minimize errors, we use 60 routing simulations for each combination of graph type, drawing algorithm, and routing algorithm. The TTL parameter is set to 100 hops for greedy routing and greedy routing with backtracking, and to 50 hops for lookahead routing.

6.2.1 Fixed-vertex drawing algorithms

The characteristic routing performance using fixed-vertex drawing algorithms as an embedding is shown in Figure 6.2. Embedding through a very simple ordering approach, CCC, does not give a better starting point than a random embedding. Even if the success rate of Six/Tollis (16%) is not overwhelming, routes can be very long. None of the other algorithms are able to compute embeddings where greedy routing is able to use more than fourty hops. This could be explained with the first phase of Six/Tollis which places as much edges as possible on the perimeter such that long routes with small steps (from one vertex to its direct neighbor on the ring) are possible.

Using a backtracking approach, the routing success can exceed 50% after 90 hops and grow further, and a lookahead approach reaches a 50% success rate after 20 hops and grows up to 75%.

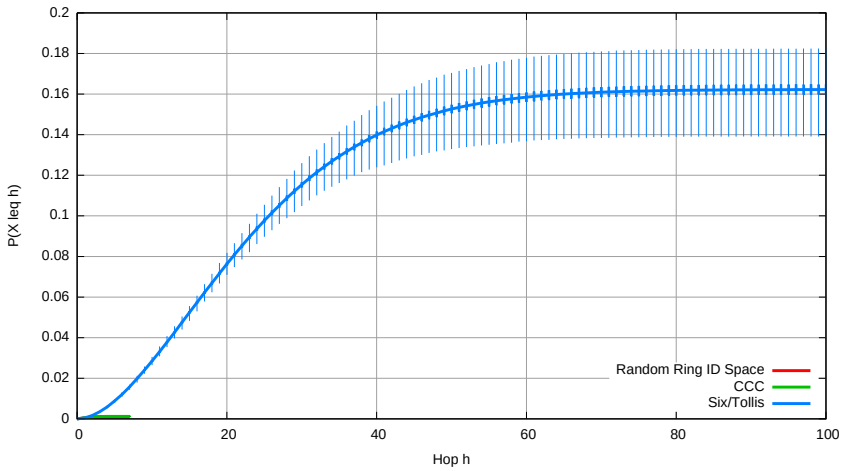


Figure 6.2.: Hop plot (CDF) of routing in an embedding through fixed-vertex drawing algorithms

6.2.2 Hierarchical drawing algorithms

The characteristic routing performance using hierarchical drawing algorithms as an embedding is shown in Figure 6.3. We can identify three classes: Routing on an embedding by Melançon/Herman or Bubbletree (circular drawing algorithms) gives the best results, Knuth and Wetherell/Shannon (tree-like drawing algorithms) form a second class, and a random embedding the third. It seems advantageous to have the spanning tree's root vertex in the center, like in Melançon/Herman and Bubbletree, and not at the border of the given identifier space, like in Knuth and Wetherell/Shannon, which makes routing attempts more robust.

Within the class of circular algorithms, Melançon/Herman performs distinctively better than Bubbletree. This is a contrast to the visual advantage of Bubbletree shown in figure 3.10. Here, it is better that the given space is less good used and the edge length scale exponential with increasing depth in the spanning tree. The performance of tree-like drawing algorithms does not rise above 2% except for embeddings of Barabási-Albert graphs, where all drawing algorithms succeed better than in other graphs (see section 6.6), and sPI.

The order of routing success is the same using the backtracking and the lookahead approach. Knuth and Wetherell/Shannon do still not reach 10% success rate, while Melançon/Herman reaches a success rate of 50% after

70 hops using backtracking and 11 hops using backtracking, and Bubbletree reaches the same success rate after 85 (12) hops. About 75% of all routing attempts succeed. Surprisingly, the backtracking approach performs worse when routing on an embedding of CAIDA, while pure greedy routing has a higher success rate here. Using the lookahead approach on a CAIDA embedding nearly all routing attempts are successful within the first 10 hops.

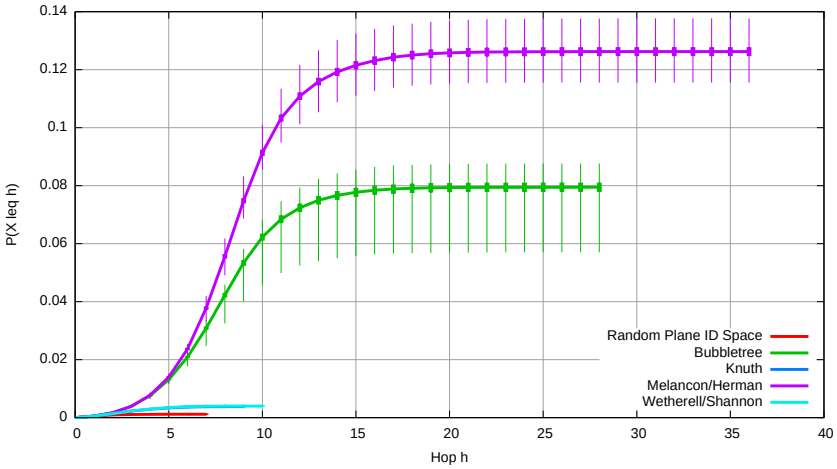


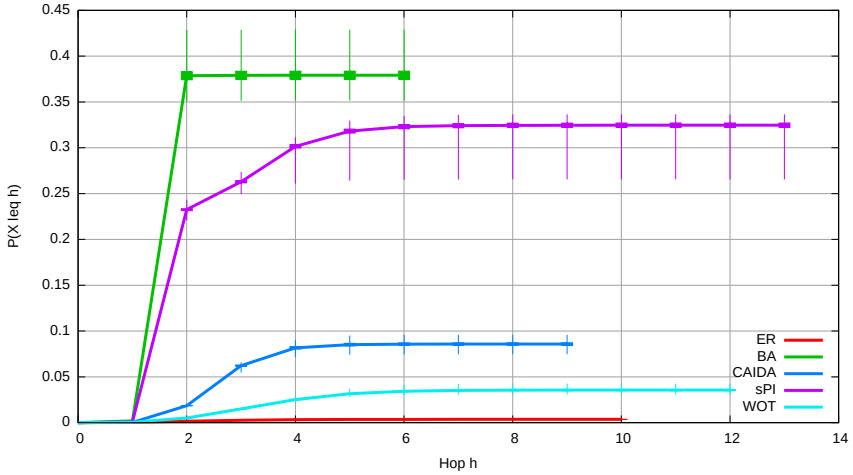
Figure 6.3.: Hop plot (CDF) of routing in an embedding through hierarchical drawing algorithms

6.2.3 Force-driven drawing algorithms

Fruchterman/Reingold is the only algorithm that has configurable options: the number of dimensions and iterations can be changed. In our work, we use twodimensional drawings and 100 iterations.

The routing results for embeddings using Fruchterman/Reingold, a force-driven drawing algorithm, rely strongly on the embedded graph. Figure 6.4 shows the routing performance for embeddings of the different graph types. The clustering coefficient gives a first indicator about the routing success. The higher the coefficient is (which means: the more clusters exists), the better appears a Fruchterman/Reingold embedding to be suited for routing. On the other hand, all BA graphs show a huge increase in the routing success at the second hop (see section 6.6); and within embeddings of BA graphs, Fruchterman/Reingold performs worse than Bubbletree or Melançon/Herman and even not remarkably better than a random embedding. sPI is the only graph on which Fruchterman/Reingold is the best embedding.

Even using lookahead routing, a Fruchterman/Reingold embedding is useless for ER graphs with a success rate below 2%. In BA graphs, all routing attempts succeed after two hops which is no improvement over a random assignment. In the other graphs, between 18% and 57% routing success can be reached using backtracking and between 28% and 78% using lookahead routing. Regarding the high runtimes for Fruchterman/Reingold, this is a very unattractive embedding for the given graphs.



Graph	CC	Routing success
BA	0.3774	38.47%
sPI	0.35	32.54%
CAIDA	0.2245	8.56%
WOT	0.4665	3.55%
ER	0.00046	0.36%

Figure 6.4.: Greedy routing of Fruchterman/Reingold on all graphs

6.2.4 Preliminary results

The best performing embedding algorithms from each class are compared in Table 6.3. The poor performance of Fruchterman/Reingold embeddings was already discussed in the previous section, and the comparison shows that this embedding has clear disadvantages in four of the five used graph types. Using greedy routing, the fast embedding algorithm Melançon/Herman allows a nearly as good routing performance as the long-running Six/Tollis. With more sophisticated routing algorithms there is still no obvious "better" drawing algorithm. For later use, one should either choose a fast recomputable approach with a Melançon/Herman embedding or, if the network structure is to change less often, a Six/Tollis embedding. The clustering coefficient might help once more here: Six/Tollis performs better in ER networks with a low clustering coefficient, and Melançon/Herman better in CAIDA with a higher one.

Graph type	Greedy routing			Backtracking			Lookahead routing		
	ST	MH	FR	ST	MH	FR	ST	MH	FR
ER (10,000 vertices)	16.22%	12.62%	0.36%	53.82%	59.72%	4.09%	76.69%	65.17%	1.95%
BA (5,000 vertices)	49.65%	48%	38.47%	63.12%	60.19%	41.57%	100%	100%	100%
CAIDA	13.55%	19.37%	8.58%	40.44%	41.40%s	20.36%	85.85%	95.22%	70.75%
sPI	21.12%	19.29%	32.54%	54.80%	42.49%	56.73%	94.79%	92.72%	78.07%
WOT	missing	10.1%	3.55%	missing	44.53%	18.06%	missing	84.91%	28.41%

Table 6.3.: Comparison of routing results for Six/Tollis, Melançon/Herman, and Fruchterman/Reingold embeddings of different graphs

6.3 Influence of the root vertex choice in hierarchical drawing algorithms

Hierarchical drawing algorithms need a spanning tree as input. We use a breadth-first search to extract a spanning tree from all kinds of graphs. Two different approaches to choose the root vertex of that tree are compared here.

The algorithms of Knuth, Wetherell/Shannon, Melançon/Herman, and Bubbletree (see Section 3.4) are designed to draw trees. As the graphs we use are no trees, we extract spanning trees before drawing them, with two strategies for the choice of a root vertex: the first strategy chooses a random vertex from the graph, without further limitations on the properties of the vertex (denoted *R*). The second strategy sorts the list of vertices in ascending degree order and chooses a random vertex from the last 10% vertices, which is a vertex with relatively high degree (denoted *HD*). Afterwards, the spanning tree is extracted using a breadth-first search.

Table 6.4 shows the average success rates for greedy routing on 60 graphs of each combination of graph type and drawing algorithm. As there are no results that either choice is obviously better than the other, we furthermore only use routing results from hierarchical drawings with a highest degree root vertex choice.

Graph type	Size	Knuth		WS		MH		BT	
		R	HD	R	HD	R	HD	R	HD
Erdös-Rényi	1,000	2.95%	2.79%	3.05%	2.98%	28.46%	28.48%	22.8%	23.74%
	5,000	0.74%	0.7%	0.73%	0.73%	16.49%	16.47%	10.78%	11.33%
	10,000	0.39%	0.38%	0.4%	0.4%	12.64%	12.57%	7.47%	7.94%
Barabási-Albert	1,000	30.6%	31.36%	31%	31.61%	49.76%	48.57%	49.35%	49.29%
	5,000	29.34%	29.42%	29.55%	29.73%	48.05%	47.95%	48.89%	49.37%
	10,000	29.3%	29.34%	29.61%	29.83%	47.96%	47.53%	49.09%	48.71%
CAIDA	20,057	2.03%	1.85%	2.31%	2.08%	18.23%	19.37%	21.25%	21.07%
sPI	11,407	16.02%	14.95%	16.72%	16.28%	18.64%	19.33%	22.23%	22.15%
WOT	25,487	1.75%	1.69%	1.72%	1.73%	9.71%	10.1%	8.9%	8.78%

Table 6.4.: Success rates for greedy routing in hierarchical drawing embeddings

6.4 Runtime of Six/Tollis in scale-free networks

While much drawings can be done in manageable time, we observe that the circular algorithm by Six and Tollis (see Section 3.3.2) takes very long time in larger graphs: it takes more than four hours to create an embedding of a Barabási-Albert network with 5,000 vertices, more than six hours for the CAIDA network and more than 50 hours for the WOT network. This section gives an explanation for this behaviour.

The first phase of Six/Tollis places all vertices of a graph on the perimeter of a circle. This is done through an algorithm inspired by a outerplanarity test described in [21], as each outerplanar graph can be drawn such that all vertices lie on a circle and no edges cross. If the graphs we use were outerplanar, this algorithm would terminate fastly. In [17], Jao and West present upper bounds for the number of vertices from an outerplanar graph of n vertices that have a degree of at least k. For $k \geq 6$ and $n \geq k+2$ this bound is $\lfloor \frac{n-6}{k-4} \rfloor$. Thus, a graph with 1,000 vertices must not have more than $\lfloor \frac{1000-6}{10-4} \rfloor = \lfloor \frac{994}{6} \rfloor = 165$ vertices (16.5%) with a degree of 10 or higher to be outerplanar. As in a graph created using the Barabási-Albert model all vertices have a (arbitrarily chosen) minimal degree, in our evaluation of 10, a test for outerplanarity must always fail.

But usually, this is no harm to this algorithm as it tries to place as much vertices as possible on the perimeter first. Through the computation of the longest path within a depth-first search tree of the reduced graph, as much vertices as possible should get placed in an order such that there is only a small amount of edge crossings. Afterwards the remaining vertices get merged into this order. The crossings introduced with these placements are tried to be decreased in the second phase of the algorithm.

Measuring the number of vertices within the longest path gives an indicator for the success of the first phase. Within graphs of 1,000 vertices, this path is noticable smaller than 100 vertices in Barabási-Albert graphs and contains mostly high degree vertices (average degree of the vertices in the longest path: >150; average degree of all vertices: <20). In Erdös-Rényi graphs of the same size, the path is around 700 vertices long and the average degree of the covered vertices is only slightly higher than the average degree of a usual Erdös-Rényi graph of that size. For graph sizes of 5,000 vertices, the paths do not grow in Barabási-Albert graphs, but are about 3,500 vertices long in Erdös-Rényi graphs.

The more vertices are placed in the first placement phase, the less work needs to be done to decrease crossings. In Erdös-Rényi graphs, the crossing reduction phase needs about 7,000 iterations for 5,000 vertices connected by 25,000 edges, while for the same number of vertices of a Barabási-Albert graph, connected by 100,000 edges, more than 500,000 iterations need to be executed. Even if the number of edges is similar, the second phase runs longer in Barabási-Albert graphs: about 48,000 iterations in Barabási-Albert graphs with 1,000 vertices and 20,000 edges, compared to 7,000 iterations in Erdös-Rényi graphs with 25,000 edges. Embedding the WOT graph takes even more time for the first phase (about 60h), but as the longest path contains nearly 7,000 vertices (of a total of 25,487 vertices), the crossing reduction phase terminates after less than 50,000 iterations.

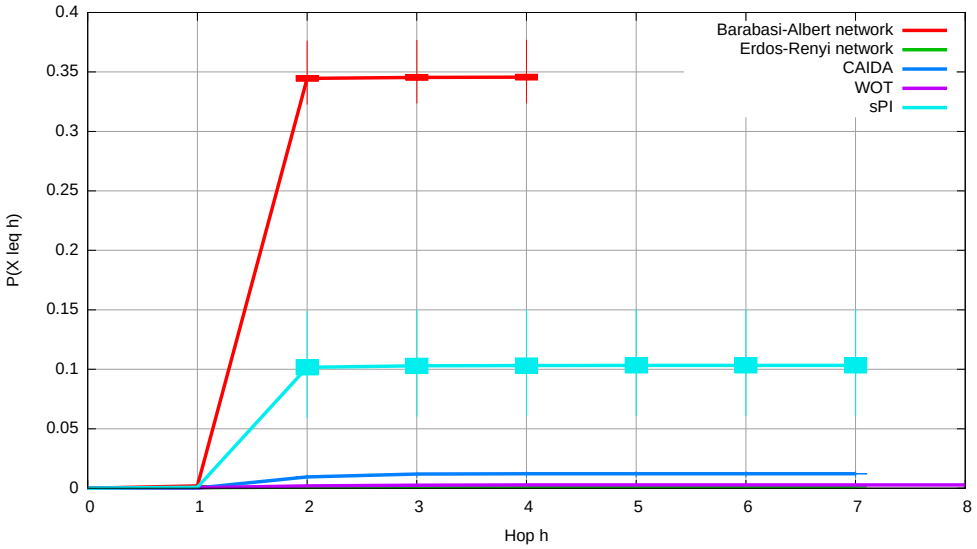
It looks very strange that the second phase of Six/Tollis works that much slower than a run of CCC, even if the approach (minimizing the crossings within vertices randomly placed on a circle) looks similar. We assume that the first phase delivers an order in which the crossing reduction phase can work better than in a completely random order. Further analysis needs to be postponed, as counting edge crossings is a very time consuming metric to run even on small graphs.

6.5 Performance of routing on random identifier spaces

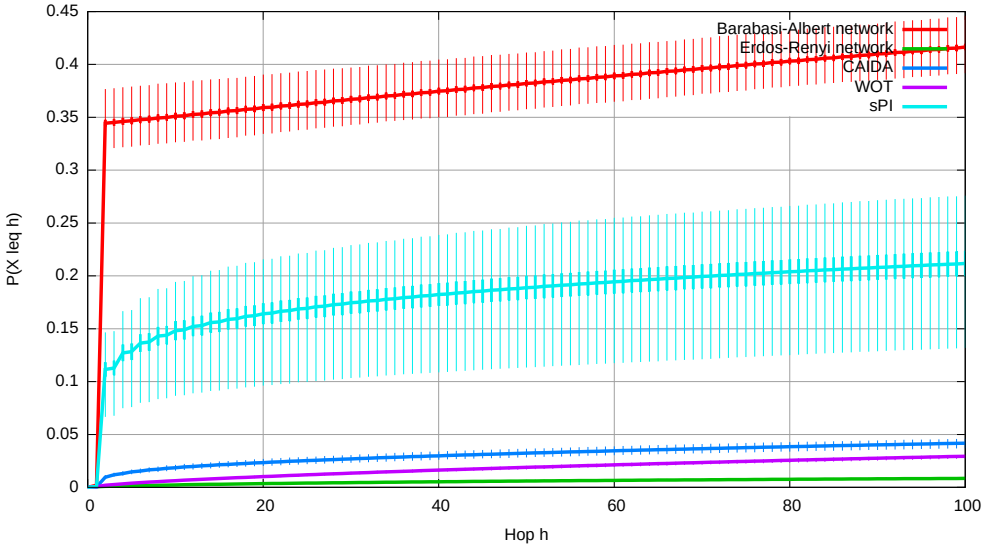
The main purpose of graph drawing algorithms is to produce an aesthetical visualization of a graph. Placing the vertices at random positions does not fulfill this purpose. The same fact should occur within routing: assigning random identifiers to each vertex and trying to route on this should result in poor routing performance.

Routing results for the three analyzed routing algorithms are shown in Figure 6.5, based on a randomly assigned two-dimensional identifier space. The high success in a Barabási-Albert network is explained in Section 6.6 and is a consequence of the structure of such a network. With about 10% routing success in sPI and much less than 5% in other networks, this is no good embedding choice. Even a backtracking approach does not perform much better. Regarding that the routing success for all networks except BA only doubled, while the TTL of 100 hops is completely exhausted, this might be a result of a random walk.

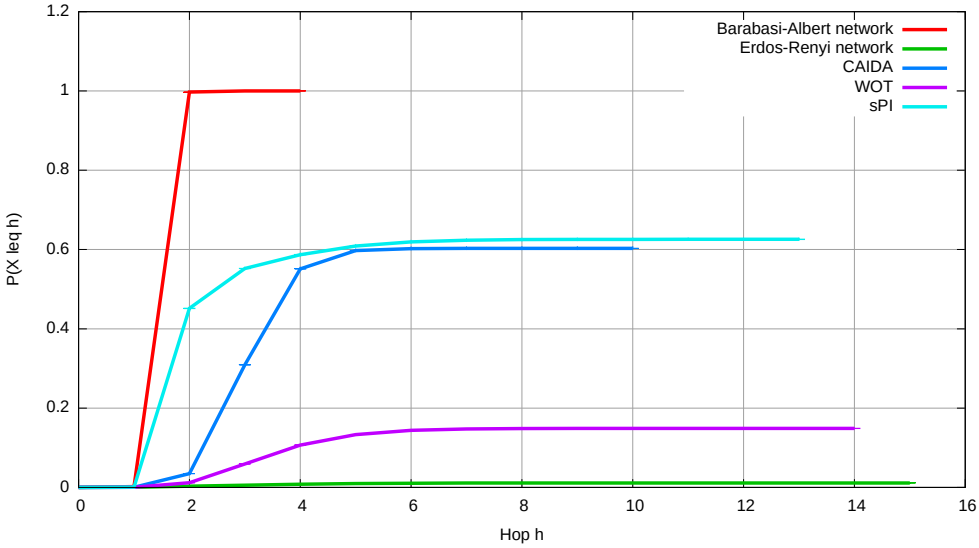
Using lookahead routing, the routing success increases strongly in the graphs of CAIDA and sPI. Once more, this is a characteristic of this routing algorithm which is more robust against dead ends and apparently ineffective hop choices. As shortest paths can grow up to 11 hops in these two graphs, the success rate is surprisingly high.



(a) Greedy routing



(b) Greedy backtracking routing



(c) Lookahead routing

Figure 6.5.: Hop distribution of routing on a randomly assigned identifier space

6.6 Routing in Barabási-Albert graphs

In contrast to all other networks, routing results in Barabási-Albert networks show a huge increase for the routing success from the first to the second hop and only little increase afterwards. A characteristic plot comparing the routing performance of all graph types embedded in a random identifier space is shown in Figure 6.6; the performance looks similar in other embeddings. This is surprising as the three networks CAIDA, sPI, and WOT are real world instances of the Barabási-Albert model. In this section we will have a look at this, comparing the behaviour in homogeneous networks and scale-free networks.

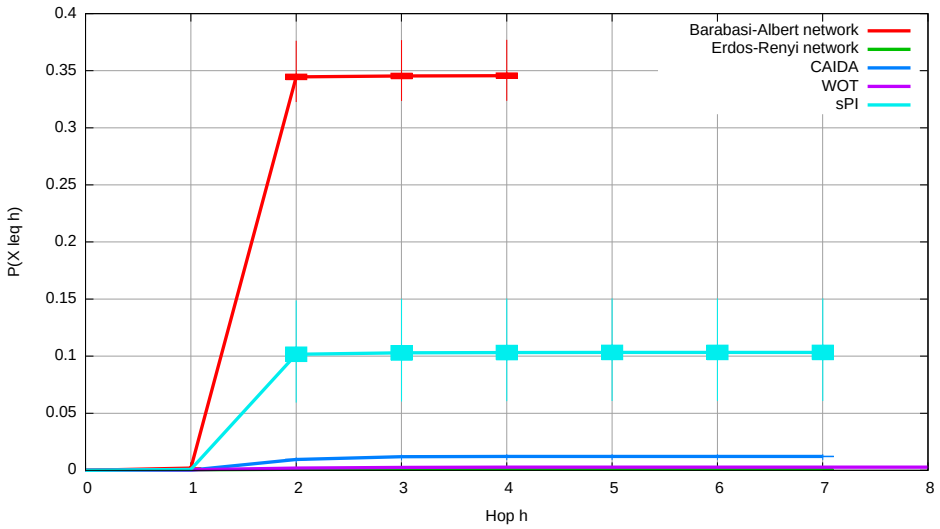
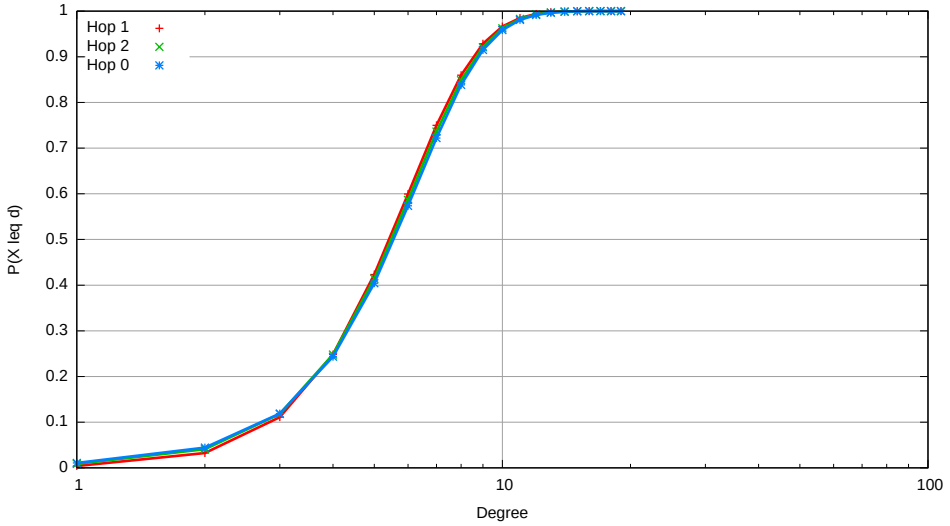


Figure 6.6.: Hop distribution of greedy routing on a randomly assigned identifier space

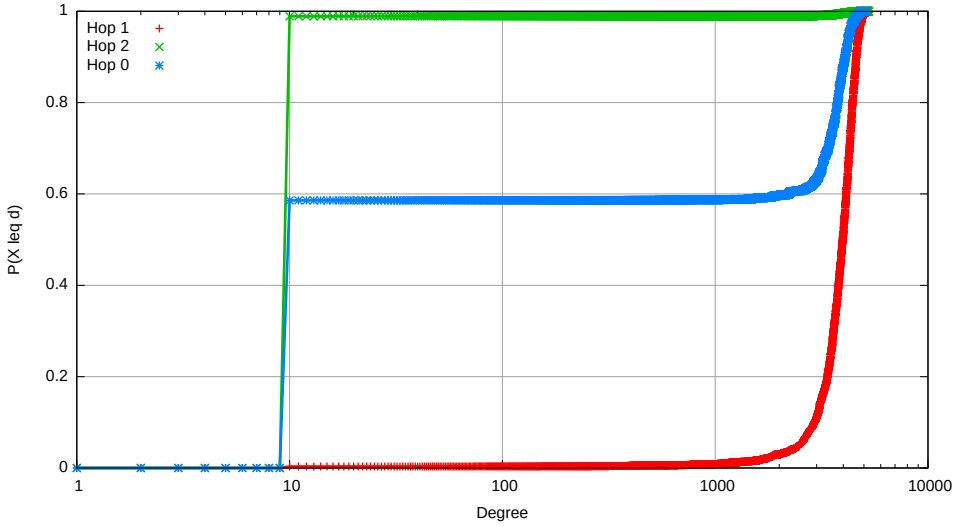
The first step is to analyze the special structure of these networks. The degree distribution of all networks can be found in Figure 6.1. In Barabási-Albert networks, the average diameter is two hops long, the maximum diameter is three hops long; both independent of the sizes of generated networks. Thus, the probability of finding a short routing path is very high: the average routing path is two hops long, the longest routing path takes five hops. In Erdös-Rényi networks, the diameter depends more on the graph size. At 1,000 vertices, the average diameter is about 4.5 hops long; at 5,000 vertices it is about 5.5 hops long, and 5.9 hops at a size of 10,000 vertices. Maximum diameters can take up to 11.7 hops. This makes the routing success more dependent on the choice of the next intermediate hop.

Figure 6.7 gives us an insight in the behaviour of greedy routing in these networks, based on Melançon/Herman embeddings. It shows the degree distribution of the first, second and third routing hop. Where there is no obvious difference in the Erdös-Rényi networks, the distribution changes seriously in Barabási-Albert networks. In most cases the first hop reaches a high degree vertex (the highest degree is 4,667, the median degree nearly 4,000; for a graph size of 10,000 vertices). Thus, it is not unlikely that the routing destination is reached after the second hop. On the other hand, in the second hop nearly always a low degree vertex is reached. In 37% of routing attempts, the destination is reached. In Figure 6.6, one can also see that in a very small number of attempts more hops are done until routing succeeds. This means that the high degree vertex has no connection to the routing destination, thus it tries to find another vertex closer to the destination. As its neighbors are mostly small degree vertices, the route soon comes to a dead end: the probability that just this third hop has a connection to the destination, but the second hop with plenty of neighbors has not, is very small.

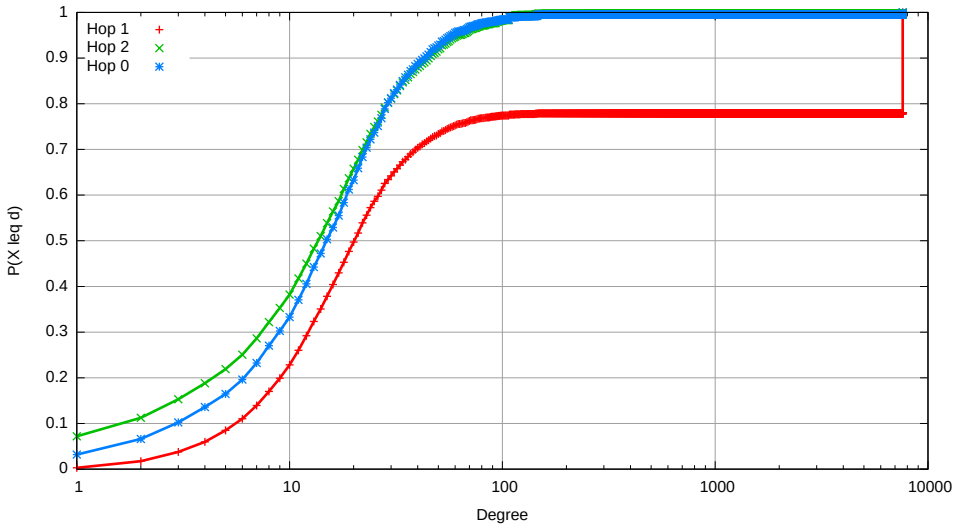
This behaviour can not only be seen in greedy routing, but also in greedy backtracking routing; and not only when embedding the graph using Melançon/Herman, but also using other graph drawing algorithms.



(a) Degree distribution (CDF) of routing hops in Erdös-Rényi networks



(b) Degree distribution (CDF) of routing hops in Barabási-Albert networks



(c) Degree distribution (CDF) of routing hops in the sPI graph

Figure 6.7.: Degree distribution of routing hops using a Melançon/Herman embedding

7 Summary & Conclusion

We evaluated the usability of graph drawing algorithms as an embedding, measured primarily by their routing success, to enable greedy routing in distributed systems. An important preliminary in this systems is the fixed network structure, where no additional links may be formed.

Seperated into three classes, six existing drawing algorithms and one simple vertex ordering algorithm were used. After the first phase of our work, the implementation of these drawing algorithms and the generation of numerous embeddings, we took a second indicator into account that might be important for later use: the runtime of each embedding. The force-driven approach by Fruchterman and Reingold was one of the longest running algorithms, but routing on an embedding by this algorithm is not more successful than on a faster computed embedding. On the other hand, longer runtimes for a higher success might be suited comparing the fast running algorithm by Melançon and Herman to the slowest algorithm we took into account, the one by Six and Tollis. In some cases, the slower algorithm computed an embedding on which routing performs better, in other cases the faster algorithm arranges an embedding with better routing performance - and in even other cases, we were unable to compute a Six/Tollis embedding as it took too long time to reach a statistically broad fundament of sixty embeddings. Thus, we cannot clearly state which graph drawing algorithm is best suited for all possible applications, but we can recommend to narrow the field of view to the algorithms by Six/Tollis and Melançon/Herman.

Following the general routing results, we analyzed peculiarities that might help other to analyze similar results within other drawings. For example, we analyzed differences between networks generated using the Barabási-Albert model and real world instances of such scale-free networks. Working on self-generated networks might be more controlable, but our results show that there are huge differences between an idealized model and realistic data sets.

The final application of graph drawing algorithms in an existing environment is still to be evaluated. We evaluated algorithms that do not yet work in a distributed manner such that each vertex could locally compute an identifier, but where the identifiers are assigned using global knowledge over the whole topology. This is a major issue that needs to be solved for a realistic usage.

8 Future work

The influence of the following points onto the routing performance is still to be evaluated. Some of these were already present at the beginning of our work, others arose during the evaluation.

Edge crossings Especially the circular drawing algorithms, from which only the one by Six/Tollis is analyzed here, considers the minimization of edge crossings for a better perception. As Garey and Johnson proof in [13], counting crossings is a NP-complete problem. A naïve (and thus very time consuming) algorithm is implemented, but due to time constraints, we are not able to analyze this hypothesis further.
Secondly, the puzzling behaviour of Six/Tollis on large Barabási-Albert graphs needs to be analyzed in conjunction with edge crossings.

Other spanning tree choices The hierarchical drawing algorithms requires the generation of a spanning tree to layout cyclic graphs. Currently, a root vertex is chosen and the tree is built using a breadth-first walk. This leads to flat, but broad trees. One could also build a degree-constrained spanning tree where the maximum number of children vertices is limited. As Garey and Johnson proof in [14], this is once more a NP-complete problem. Using such a tree, also other hierarchical algorithms like the one that Kar et al. present in [18] that are not built to handle too broad trees could be evaluated.

Scaling factors In hierarchical drawing algorithms, some parameters are chosen arbitrarily. The `leafRadius` in Bubbletree is an example for this. Changing it scales the distances between vertices. On the other hand, tree approaches like the algorithms by Knuth or Wetherell/Shannon could be enhanced by a horizontal scaling factor depending on the tree's depth, such that vertices lying in deeper levels are moved closer together as if they are layouted close to the root vertex. A comparison between the algorithm of Melançon/Herman and Bubbletree (in Erdös-Rényi networks with 10,000 vertices, the performance of greedy routing in Melançon/Herman embeddings is about 50% better than in Bubbletree embeddings) indicates that a relation might exist.

Termination conditions In his initial work onto force-driven algorithms [9], Eades states that a stable state is mostly reached after 100 iterations. Other algorithms of this class introduce better approaches to determine whether a stable state is reached earlier to avoid lots of iterations without further improvement. Thus, using such an algorithm is less cheap in runtime which increases the acceptance rate. A circular algorithm could also use other termination conditions. It might for example terminate if the number of edge crossings is minimized by a given fraction of the initial edge crossing number.

High-dimensional embeddings The algorithm by Fruchterman/Reingold supports a dynamic number of dimensions. A connection between the number of dimensions and the routing success onto such an embedding needs to be analyzed. Secondly, Harel and Koren present a high-dimensional algorithm in [16] that is supposed to run much faster. Brandenburg et al. give a comparison of other force-driven algorithms in [4], where two more complex algorithms are prefered to the rather simple by Fruchterman/Reingold. It might also be possible to modify other algorithms to use more than two dimensions, for example could Bubbletree put the vertices on the surface of a sphere instead of a circle.

Distributed embeddings In our work, we use drawing algorithms with global knowledge. It would be more suited to use drawing algorithms that can create embeddings distributed. Like in Darknets, it might be impossible to receive knowledge about the whole network with all vertices and edges.

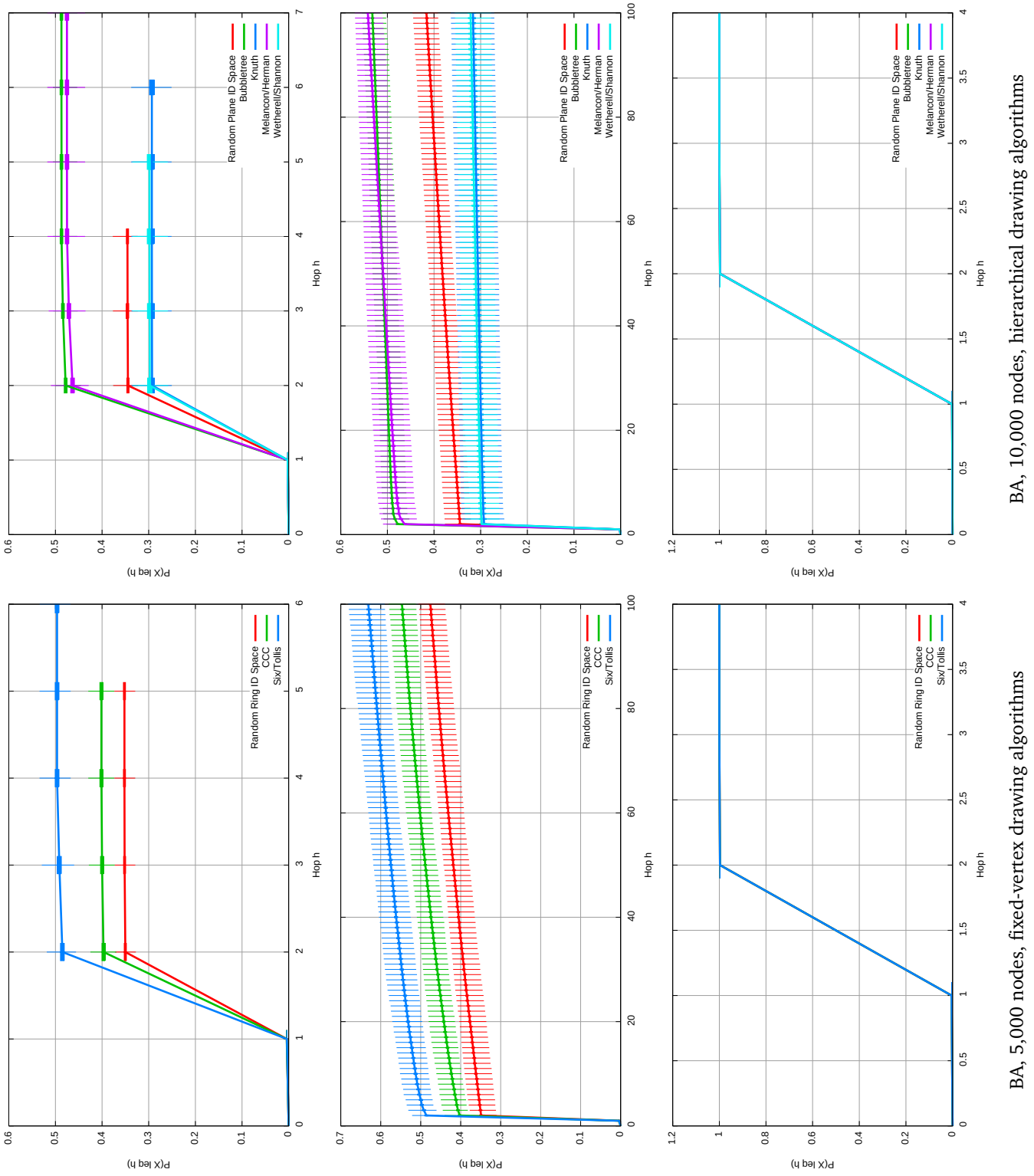
A List of Illustrations

List of Figures

List of Tables

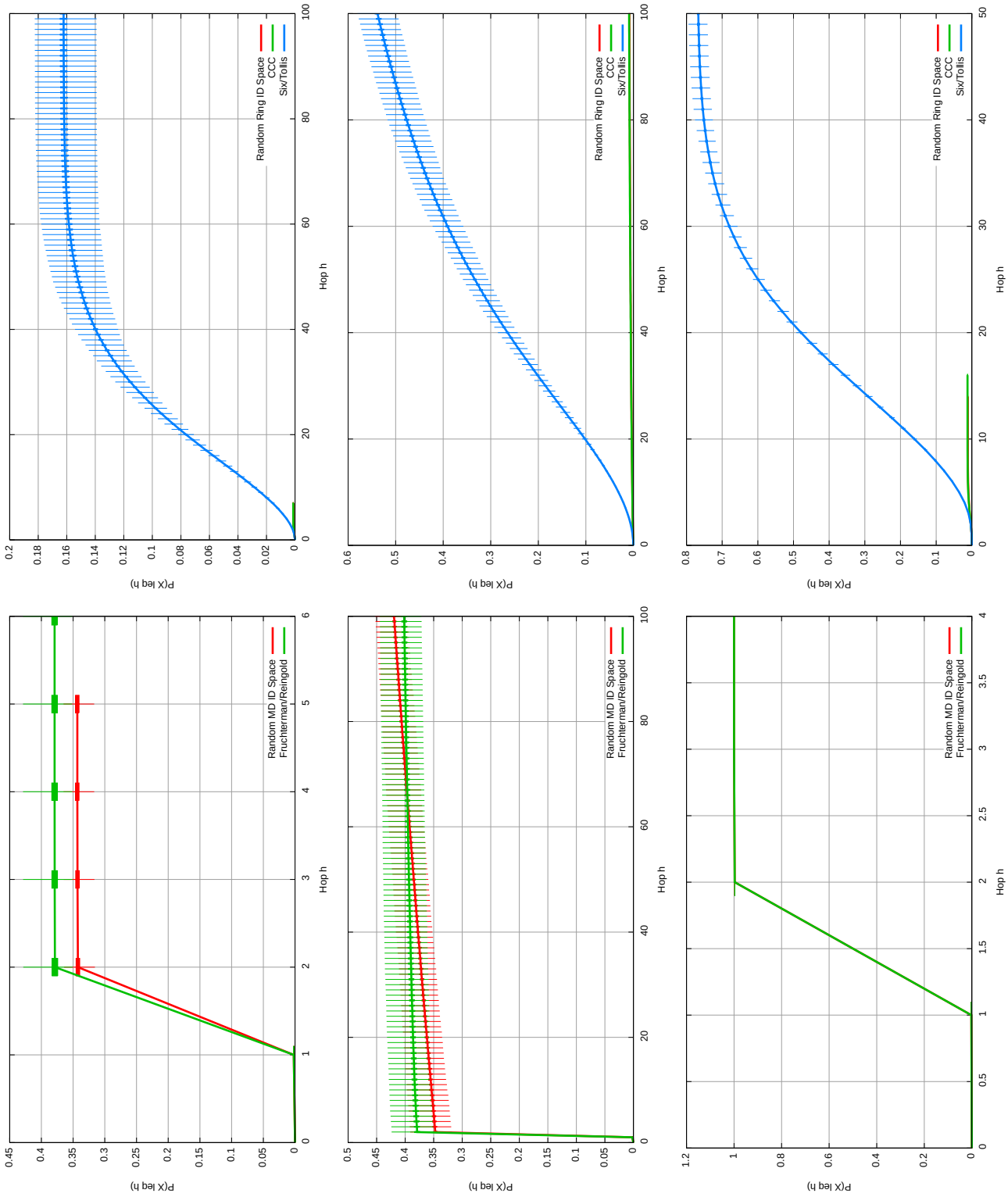
List of Algorithms

B Complete routing results



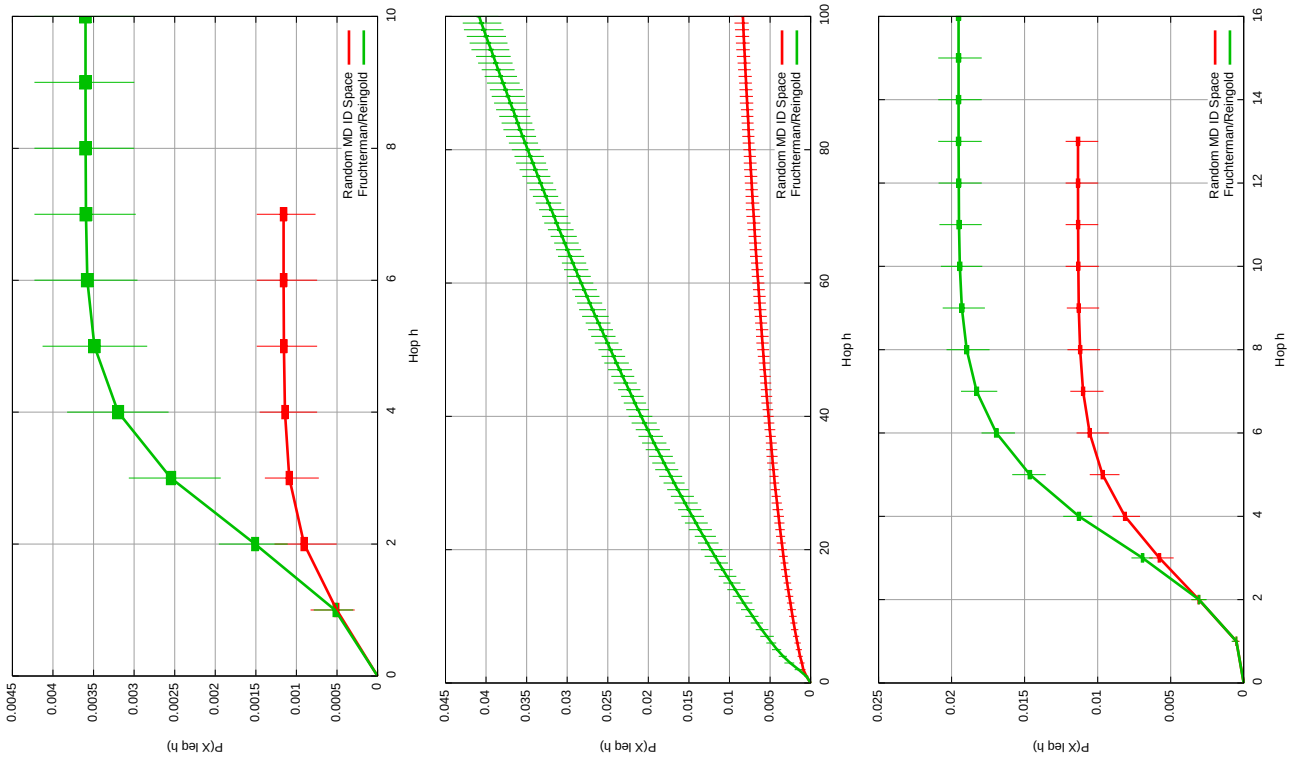
BA, 10,000 nodes, hierarchical drawing algorithms

BA, 5,000 nodes, fixed-vertex drawing algorithms

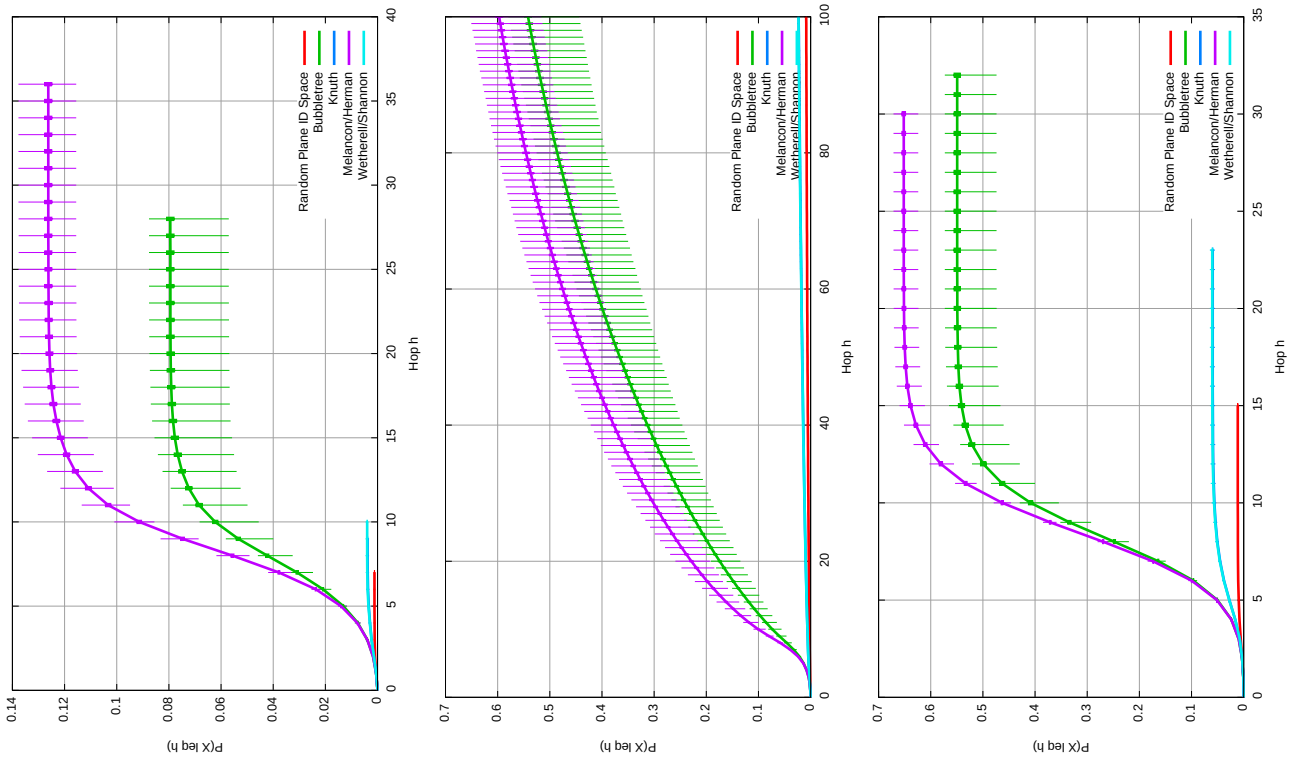


ER, 10,000 nodes, fixed-vertex drawing algorithms

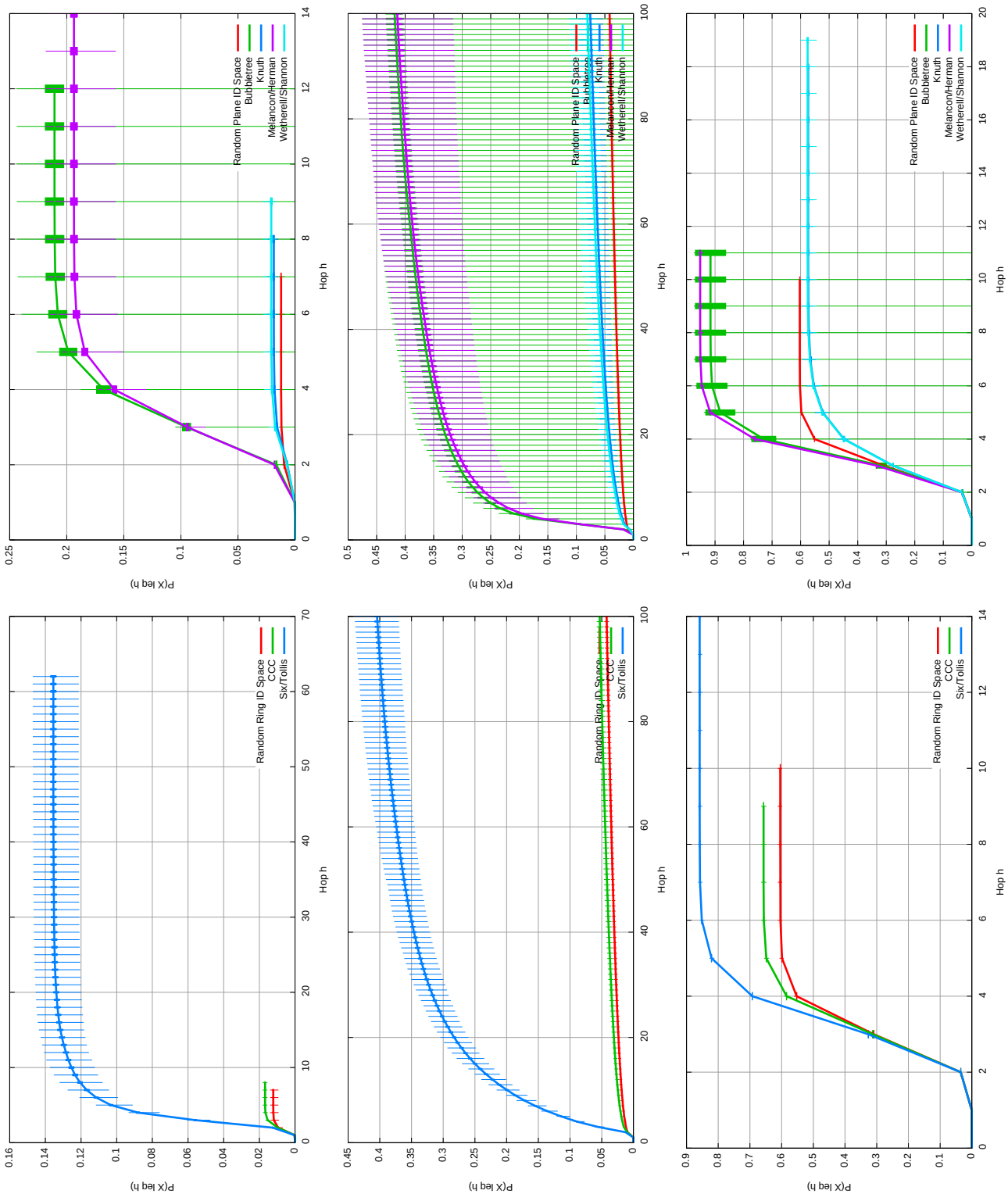
BA, 10,000 nodes, force-driven drawing algorithms



ER, 10,000 nodes, force-driven drawing algorithms

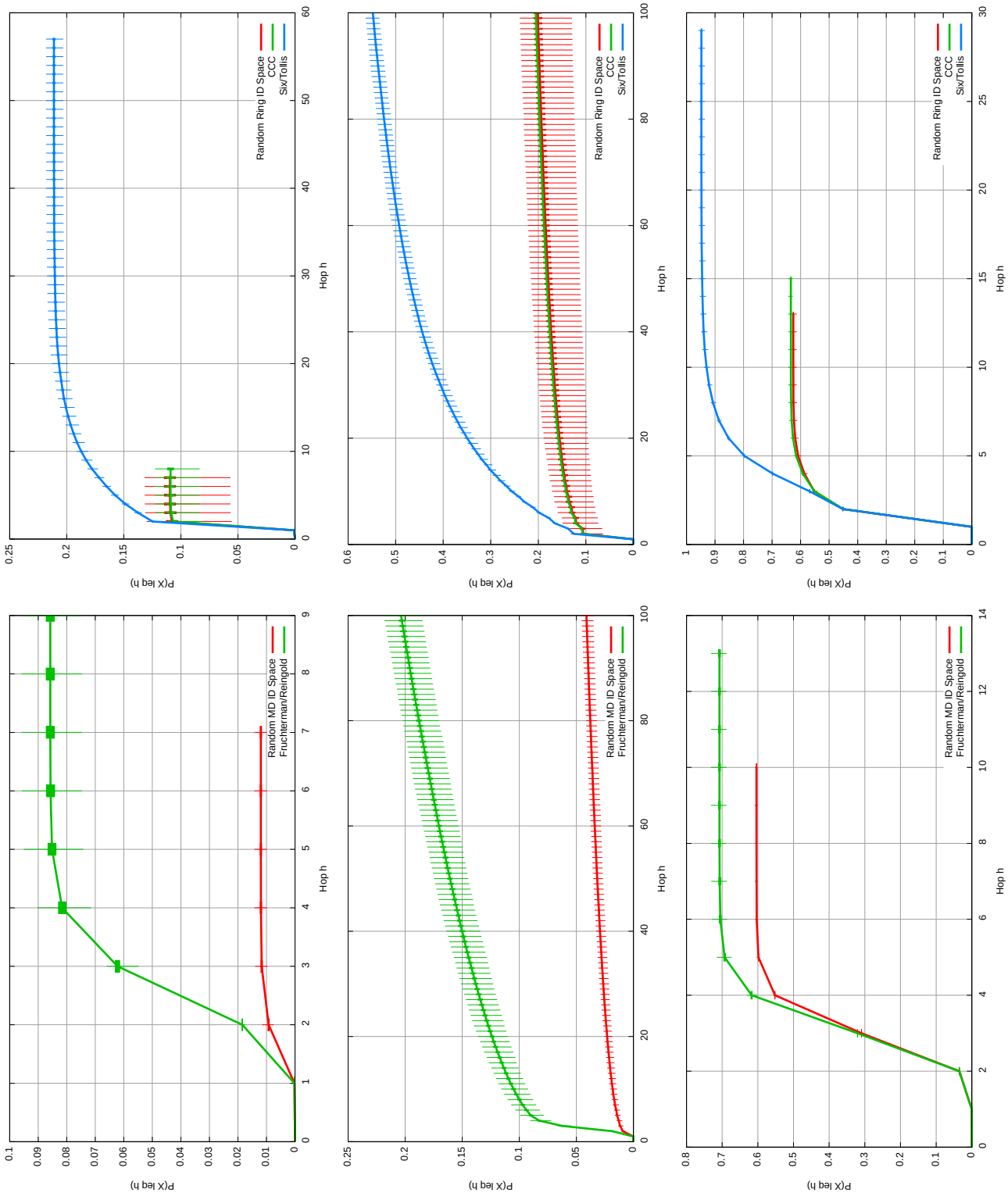


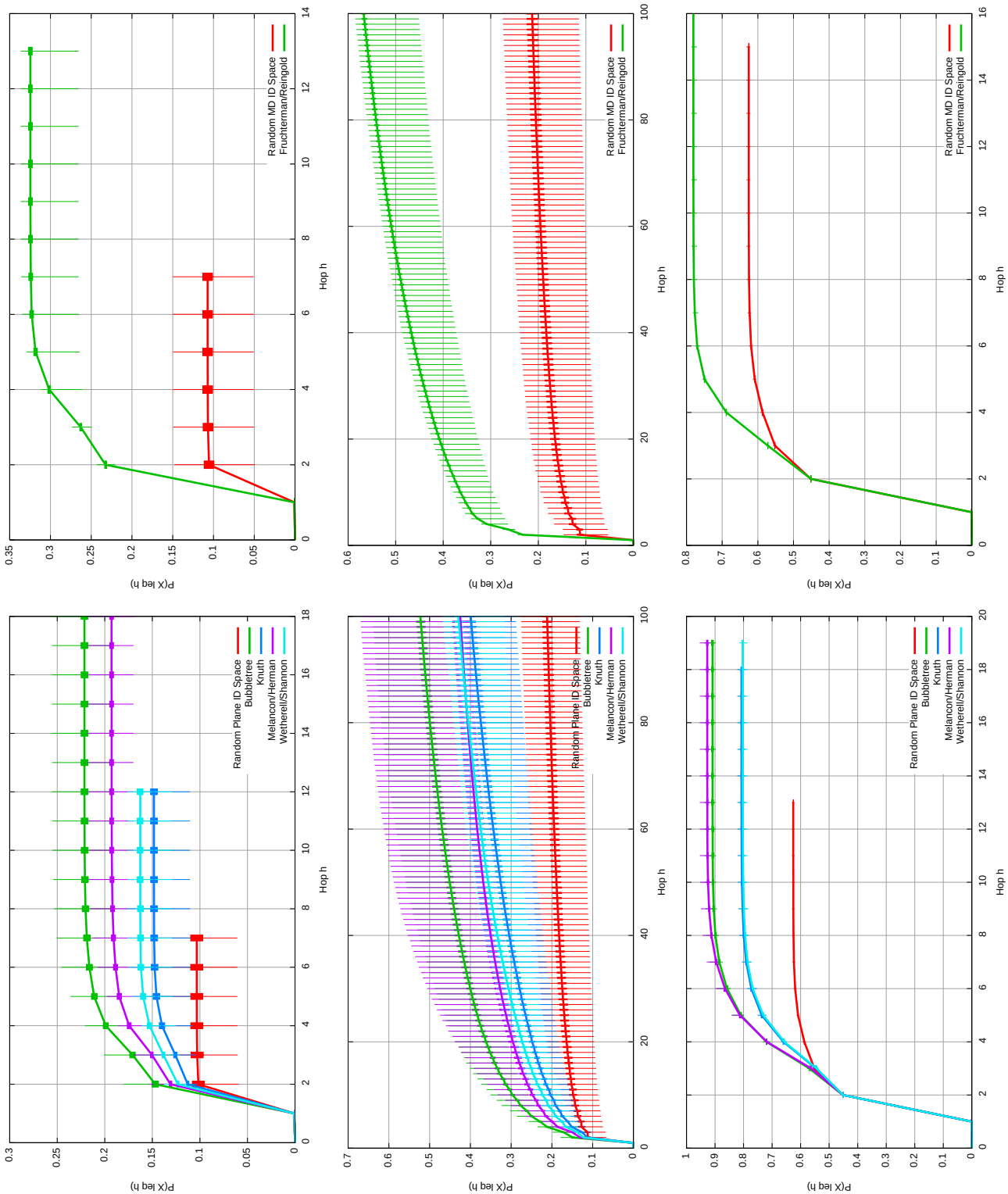
ER, 10,000 nodes, hierarchical drawing algorithms



CAIDA, hierarchical drawing algorithms

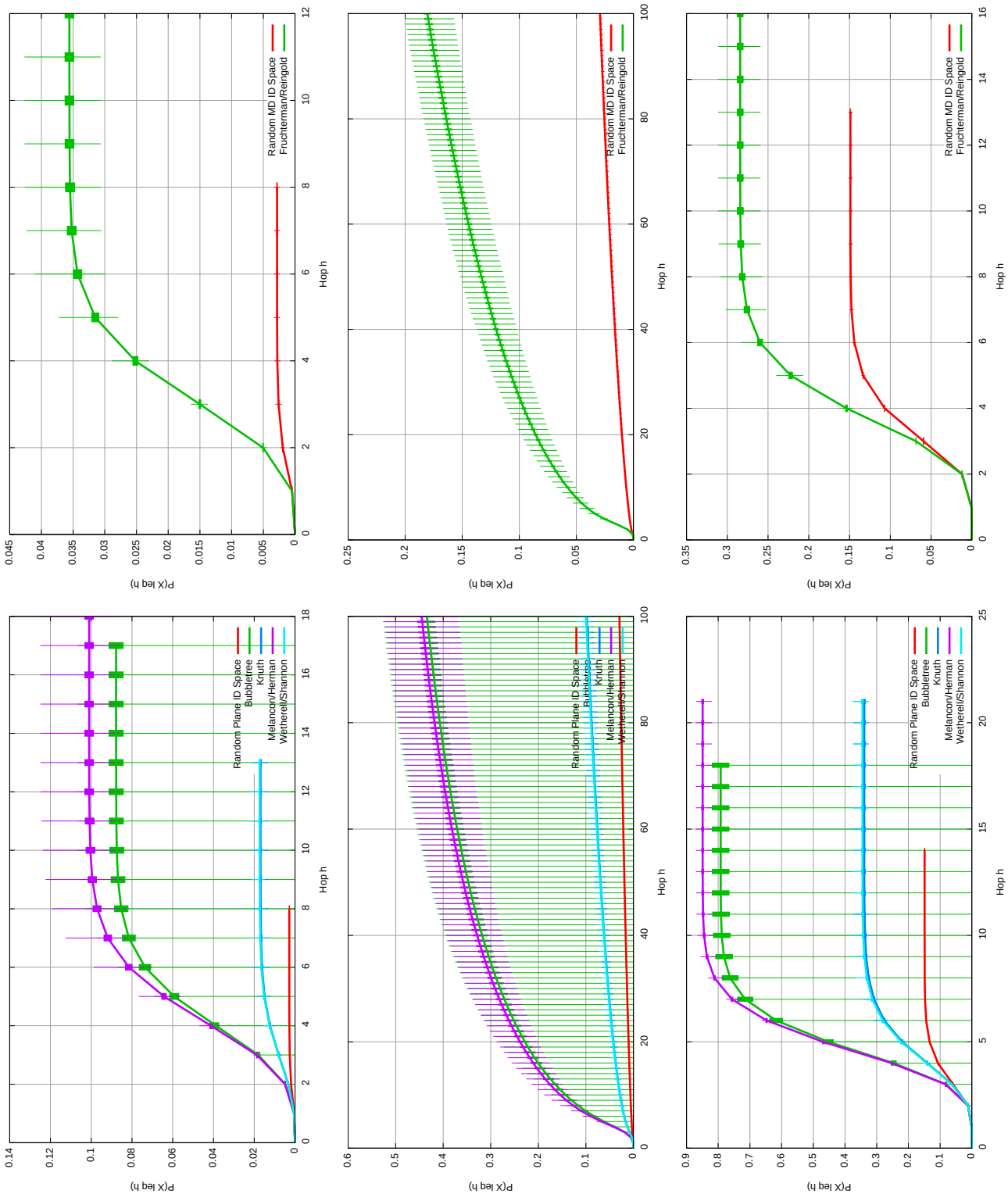
CAIDA, fixed-vertex drawing algorithms


sPI, fixed-vertex drawing algorithms
CAIDA, force-driven drawing algorithms
Random Ring ID Space
CCC
Six/Tollis
Random MD ID Space
Fruchterman/Reingold
Hop h
P(X |eq h)



sPI, force-driven drawing algorithms

sPI, hierarchical drawing algorithms



WOT, force-driven drawing algorithms

WOT, hierarchical drawing algorithms

C Additional algorithms

C.1 Calculation of edge crossings

Some graph drawing algorithms, like Six/Tollis (see Section 3.3.2), require the computation of edge crossings. We implemented two ways to count crossings: the first is a general and canonical approach which can be used on any twodimensional layout; the second is only applicable to circular layouts.

C.1.1 Canonical approach

The canonical approach to count edge crossings is simple, but inefficient, and checks for each pair (n, m) of edges whether they do cross. Line equations are formed for both edges and an intersection point is calculated. If there is none, the edges do not cross. If there is one that also lies on both lines, there is a crossing in this point.

Algorithm 8 Count edge crossings in any layout

```
    function countCrossings(G)
        numberOfCrossings := 0
        handledEdgePairs := ∅
        for all n ∈ E do
5:          for all m ∈ E do
                if (n,m) ∉ handledEdgePairs & hasCrossing(n,m) then
                    numberOfCrossings := numberOfCrossings + 1
                end if
                handledEdgePairs := handledEdgePairs ∪ (n,m)
10:         end for
        end for
        return numberOfCrossings
    end function
```

C.1.2 Crossings in a circular layout

Six and Tollis use a special algorithm to count crossing edges in a circular layout in [26]. Based on Figure C.1.2, the term *open* edge needs to be introduced. The edges e_i and e_j can only cross if one of the endpoints of e_j lies between the endpoints of e_i. In this case, v (one endpoint of e_j) lies between u and v, and e_j is an *open* edge.

The algorithm holds a list of open edges and traverses the circle clockwise. For each vertex v, the edges ending in v are removed from this list as they are no longer open. For all edges that end in v, crossings with the remaining open edges are calculated: the edges $e_i \in E^-(v)$ and $e_j \in openEdges$ have a crossing when the source vertex of e_j has a larger identifier than the source vertex of e_i.

The time complexity for the canonical approach lies in $\mathcal{O}(|E|^2)$, the algorithm of Six and Tollis counts the number of crossings in $\mathcal{O}(|V| + |E| + \chi(G))$, where $\chi(G)$ denotes the number of crossings in the graph G[3, page 36]. This version of the algorithm is taken form [3] where it it described more detailed and intuitive than in [26].

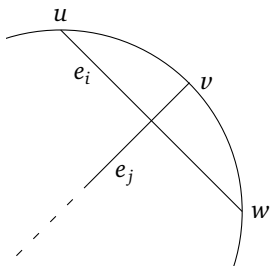


Figure C.1.: Illustration of an open edge (u, w) (after [26])

Algorithm 9 Count edge crossings in a circular layout

```
function countCircularCrossings(graph)
    numberOfCrossings := 0
    list of edge openEdges := ∅

    for i = 0, ..., n − 1 do
        openEdges := openEdges \ E⁻(i)
        for all e ∈ E⁻(i) do
            for all f ∈ openEdges do
                if source(f) > source(e) then
                    numberOfCrossings := numberOfCrossings + 1
                end if
            end for
        end for
        openEdges := openEdges ∪ E⁺(i)
    end for

    return numberOfCrossings
end function
```

C.2 Additional functions to some graph drawing algorithms

Algorithm 10 Canonical circular algorithm - additional functions

```
function getPredecessor(u)
    return v ∈ V : v.position < u.position and ∄w ∈ V : v.position < w.position < u.position
end function

function swapPositions(u, v)
    tempPos := u.position
    u.position := v.position
    v.position := tempPos
end function
```

Algorithm 11 Fixed-vertex algorithm by Six and Tollis - additional functions

```
function getPairEdges(u)
    pairEdges := ∅
    for all v ∈ getAllEdges(u) do
        if v.source = u then
            vEnd := v.destination
        else
            vEnd := v.source
        end if
        for all w ∈ getAllEdges(vEnd) do
            if w.source = w then
                wEnd := w.destination
            else
                wEnd := w.source
            end if
            if wEnd ≠ u and (u, wEnd) ∈ getAllEdges(u) then
                pairEdges := pairEdges ∪ {w}
            end if
        end for
    end for
    return pairEdges
end function
```

```
    function getAllEdges(v)
        edgeList := A+(v) ∪ additionalEdges[v]
        for all u ∈ edgeList do
25:         if u.source ∈ removedVertices or u.destination ∈ removedVertices then
                edgeList := edgeList \ {u}
            end if
        end for
        return edgeList
30: end function

    function createTriangulationEdges(u)
        i := 0
        j := 1
35:     currentConnections := getAllEdges(u)
        pairEdges := getPairEdges(u)
        triangulationEdgesCount := (|currentConnections| − 1) − |pairEdges|
        while triangulationEdgesCount > 0 do
            v_i := currentConnections[i]
40:         v_j := currentConnections[j]
            if v_i ∉ removedVertices and v_j ∉ removedVertices and v_i ∉ A(v_j) then
                additionalEdges[v_i] := additionalEdges[v_i] ∪ {(v_i, v_j)}
                additionalEdges[v_j] := additionalEdges[v_j] ∪ {(v_i, v_j)}
                triangulationEdgesCount := triangulationEdgesCount - 1
45:         end if
        end while
        j := j + 1
        if j = |currentConnections| then
            i := i + 1
50:         j := i + 1
        end if
    end function

    function longestPath
55:     startVertex := x ∈ V : D(x) = max({D(y)|y ∈ V})
        startVertex.depth := 0
        deepestVertex := startVertex  { deepestVertex holds the vertex with the maximum depth value }
        dfs(startVertex)
        return findLongestPath(deepestVertex, deepestVertex)
60: end function

    function dfs(v)
        for all w ∈ getAllEdges(v) do
            if !w.visited then
65:             w.visited := true
                w.parent := v
                w.parent.children := w.parent.children ∪ {w}
                w.depth := v.depth + 1
                if w.depth > deepestVertex.depth then
70:                 deepestVertex := w
                end if
                if w.source = v then
                    wEnd := w.destination
                else
75:                 wEnd := w.source
                end if
                dfs(wEnd)
            end if
        end for
80: end function
```

```
    function findLongestPath(v, comingFrom)
        connections := v.children ∪ {v.parent} \ {comingFrom}
        if |connections| = 0 then { v is a leaf vertex }
            return {v}
        end if
        longestPath := ∅
        for all w ∈ connections do
            if |findLongestPath(w, v)| > |longestPath| then
                longestPath := findLongestPath(w, v)
            end if
        end for
        longestPath := longestPath ∪ {v}
        return longestPath
    end function

    function placeRemainingVertices(longestPath)
        unplacedVertices := nodeList \ longestPath
        for all u ∈ unplacedVertices do
            for all v ∈ A+(u) do
                if v ∈ longestPath then
                    x := position of v in longestPath
                    break
                end if
            end for
            for i := |longestPath|, ..., x do
                longestPath[i + 1] := longestPath[1]
            end for
            longestPath[x] := u
        end for
    end function
```

Algorithm 12 Hierarchical algorithm by Melançon and Herman - additional functions

```
    function adjustChildren(v, s)
        if s > π then { As we use half sectors, s is compared to π and not 2π }
            v.c := π/s
            v.f := 0
        else
            v.c := 1
            v.f := π − s
        end if
    end function
```

Algorithm 13 Hierarchical algorithm Bubbletree - additional functions

We found this algorithm to compute the smallest enclosing circle for a set of points at `http://stackoverflow.com/a/2086118/1116230`.

```
    { A circle is noted as the set (center, r) with its center point (x, y) and the radius r. }

    function SMALLESTENCLOSINGCIRCLE(v)
        circles := ∅
 5:     for all w ∈ A⁺(v) do
            circles := circles ∪ {((v.x, v.y), v.r)}
        end for
        c1 := GETANDREMOVEFIRST(circles)
        while |circles| > 0 do
10:         c2 := GETANDREMOVEFIRST(circles)
            c1 := ENCLOSINGCIRCLE(c1, c2)
        end while
        return c1.radius
    end function
15:
    function ENCLOSINGCIRCLE(c1, c2)
        if CIRCLEINENCLOSINGCIRCLE(c1, c2) then
            return c2
        else if CIRCLEINENCLOSINGCIRCLE(c2, c1) then
20:         return c1
        else if c1.radius > c2.radius then
            return ENCLOSINGCIRCLE(c2, c1)
        else
            lineBetweenCenters := |c1.center, c2.center|
25:         radius := c1.radius + c2.radius + (lineBetweenCenters)/2
            θ := 0.5 + (c2.radius − c1.radius)/(2 ∗ lineBetweenCenters)
            center.x := (1 − θ) ∗ c1.center.x + θ ∗ c2.center.x
            center.y := (1 − θ) ∗ c1.center.y + θ ∗ c2.center.y
            return (center, radius)
30:     end if
    end function

    function CIRCLEINENCLOSINGCIRCLE(inner, outer)
        lineBetweenCenters := |inner.center, outer.center|
35:     return (lineBetweenCenters + inner.radius) < outer.radius
    end function
```

D Bibliography

[1] Albert-László Barabási. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, October 1999.

[2] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. *American Journal of Sociology*, pages 361–362, 2009.

[3] Michael Baur. *Kantenkreuzungen in Kreislayouts*. PhD thesis, 2003.

[4] F. Brandenburg, Michael Himsolt, and Christoph Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. *Graph Drawing*, pages 76–87, 1996.

[5] Stina Bridgeman and Roberto Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In *Graph Drawing*, volume 4, pages 57–71. Springer, 1998.

[6] Robert Cimikowski. An analysis of some linear graph layout heuristics. *Journal of Heuristics*, 12(3):143–153, May 2006.

[7] Kimberly C Claffy. CAIDA: Visualizing the Internet, 2001.

[8] Ian Clarke, Oskar Sandberg, Matthew Toseland, and Vilhelm Verendel. Private Communication Through a Network of Trusted Connections: The Dark Freenet. 2010.

[9] Peter Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.

[10] Paul Erdös and Alfréd Rényi. On random graphs I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[11] Guy Even, Sudipto Guha, and Baruch Schieber. Improved approximations of crossings in graph drawings and VLSI layout areas. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 269–305, Portland, Oregon, United States, 2000. ACM.

[12] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, November 1991.

[13] M R Garey and D S Johnson. Crossing number is NP-complete. *Siam Journal On Algebraic And Discrete Methods*, 4(3):312–316, 1983.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, volume 44 of *Books in the Mathematical Sciences*. W. H. Freeman, 1979.

[15] Sébastien Grivet, David Auber, Jean Philippe Domenger, and Guy Melançon. Bubble tree drawing algorithm. *Computer Vision and Graphics*, pages 633–641, 2006.

[16] David Harel and Yehuda Koren. Graph Drawing by High-Dimensional Embedding. *Journal of Graph Algorithms and Applications*, 8(2):195–214, 2004.

[17] Kyle F. Jao and Douglas B. West. Vertex Degrees in Outerplanar Graphs. Technical report, Mathematics Department, University of Illinois, Urbana, 2010.

[18] Gautam Kar, Brendan Madden, and Richard S. Gilbert. Heuristic layout algorithms for network management presentation services. *Network, IEEE*, 2(6):29–36, 1988.

[19] Donald E. Knuth. Optimum Binary Search Trees. *Acta Informatica*, 1(1):14–25, 1971.

[20] Guy Melançon and Ivan Herman. Circular drawings of rooted trees. In *IN REPORTS OF THE CENTRE FOR MATHEMATICS AND COMPUTER SCIENCES*, number December. Citeseer, 1998.

[21] Sandra L Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, 1979.

[22] Christos Papadimitriou and D Ratajczak. On a conjecture related to geometric routing. *Theoretical Computer Science*, 344(1):3–14, November 2005.

[23] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In *Graph Drawing*, pages 435–446. Springer, 1996.

[24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *ACM SIGCOMM Computer Communication Review*, 31(4):161–172, October 2001.

[25] Benjamin Schiller, Dirk Bradler, Immanuel Schweizer, Max Mühlhäuser, and Thorsten Strufe. GTNA - A Framework for the Graph-Theoretic Network Analysis. *Networks*, Di(8):111, 2010.

[26] Janet M. Six and Ioannis G. Tollis. Circular drawings of biconnected graphs. *Algorithm Engineering and Experimentation*, (70):662–662, 1999.

[27] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[28] Roberto Tamassia, editor. *Handbook of Graph Drawing and Visualization*. CRC Press, 2012.

[29] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, 1988.

[30] Daniel Tunkelang. *A practical approach to drawing undirected graphs*. PhD thesis, Carnegie Mellon, December 1994.

[31] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–2, June 1998.

[32] Charles Wetherell and Alfred Shannon. Tidy Drawings of Trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, September 1979.