DAGmar: Library for DAGs

Christian Bachmaier, Andreas Gleißner, Andreas Hofmeier

Department of Informatics and Mathematics, University of Passau
{bachmaier,gleissner,hofmeier}@fim.uni-passau.de

Technical Report, Number MIP-1202
Department of Informatics and Mathematics
University of Passau, Germany
May 2012

DAGmar: Library for DAGs*

Technical Report MIP-1202
May, 2012

Christian Bachmaier, Andreas Gleißner, and Andreas Hofmeier

University of Passau
94030 Passau, Germany
{bachmaier,gleissner,hofmeier}@fim.uni-passau.de

Abstract. We provide a highly configurable generator for creating random directed acyclic graphs and level graphs. The graphs are drawn uniformly from the set of all possible graphs respecting the provided parameter set. Generating a structured set of benchmark graphs is a built-in feature.

1 Introduction

A basic goal of empirically testing (graph) algorithms is to obtain preferably general statements about properties of the considered algorithm. The graphs of the benchmark set should be as unbiased as possible. Minimizing bias ultimately leads to the demand of drawing the graphs uniformly from the overall set of graphs restricted to the given parameter set like number of vertices or density. However, finding such graphs, which usually also have have further properties as connectedness or levels, is not as straightforward as described by the seminal Erdős/Rényi model [5]. According to the survey [1] among the graph drawing and algorithm engineering community attending the Dagstuhl seminar 11191, connectedness is the most important property for analytical benchmarks.

We generate simple directed acyclic graphs (DAGs) with a specified number of vertices n, a number of edges m, and with or without a leveling of the vertices. *Simple* means that the graphs do not contain self-loops or multi-edges. The generation consists of two steps. First, a leveling is drawn uniformly from all assignments of the vertices to levels (with an optional restriction to a maximum number of levels and/or to a maximum number of vertices per level), which allow for at least m edges. Second, the edges are drawn by sampling uniformly from all m-element subsets, which correspond to a connected graph, of the set containing all potential edges.

The special case $m = n - 1$ is equivalent to sampling a uniform spanning tree, for which there are elaborate algorithms based on the matrix-tree theorem and

* Supported by the Deutsche Forschungsgemeinschaft (DFG), grant Br835/15-2 and motivated by the Dagstuhl Seminar 11191 [1].

Markov chains. Wilson [10] gives a survey of previous approaches and proposes an algorithm which samples a uniform spanning tree in the cover time of a graph. To our best knowledge, up to date there is no special algorithm for sampling uniform connected graphs for $m > n - 1$. The algorithm proposed by Rodionov et al. [8, 9], which draws a spanning tree first and then uniformly samples the remaining $m - n + 1$ edges, guarantees only *attainability*, i. e., each graph is drawn from the set of graphs with the specified parameters with only positive probability rather than uniformly, i. e., with equal probability. Thus, we resort to the commonly used *trial method*, which repeatedly uniformly samples graphs until the first happens to be connected.

We provide an example set of benchmark graphs and a generator, which is given both as a ready to use binary executable, and as well documented JAVA source code on http://www.graphdrawing.org/data.html and within the Graph Archive [2]. All parts are published under a liberal free license, i. e., you can do anything with them as long as their origin is cited.

2 Preliminaries

A k-level graph $G = (V, E, \phi)$ is a DAG $G = (V, E)$ with a surjective level assignment $\phi\colon V \to \{0, 1, \ldots, k-1\}$ for the vertices where $\phi(u) < \phi(v)$ for each edge $(u, v) \in E$. We define $n = |V|$, $m = |E|$, and the width w to be the maximum number of (non-dummy) vertices per level. An edge (u, v) is *short* if $\phi(v) - \phi(u) = 1$ and *long* otherwise. A graph is *proper* if all edges are short.

Sometimes it is useful to make a level graph proper by adding $\phi(v) - \phi(u) - 1$ *dummy vertices* for each edge (u, v), which split (u, v) into $\phi(v) - \phi(u)$ many short edges. For an *embedded* level graph, the vertices on each level are ordered from left to right. As this does not fix the routing for long edges, it can be useful to make the graph proper as described above and let the numbering also contain the dummy vertices.

3 Generation

We consider the problem of sampling uniformly from the set $\mathbb{G}$ of n-vertex m-edge undirected labeled graphs with a specific property. An obvious way of guaranteeing uniformity is to explicitly construct each element of $\mathbb{G}$ and then randomly select one of them. However, this gives rise to astronomically large memory demands. If, for instance, $\mathbb{G}$ was the set of all (labeled) trees, a number of $\binom{\frac{n(n-1)}{2}}{n-1}$ trees would have to be simultaneously held in memory. A common approach to circumvent this is the *trial method* [8, 9]. If there is an effective algorithm for sampling uniformly from some superset $\mathbb{G}' \supset \mathbb{G}$, repeatedly sample an element G of $\mathbb{G}'$ until G happens to be member of $\mathbb{G}$, too, where the outcome of each repetition is stochastically independent. On the downside, the trial method may still be highly impractical if the fraction $\frac{|\mathbb{G}|}{|\mathbb{G}'|}$ is very small, thus requiring a huge number of restarts.

Let k be the maximum number of levels and w be the maximum number of vertices on a level. If we are only interested in uniform DAGs without levelings, we delegate to generating a leveled graph with $k = n$ and $w = 1$ and discard the leveling afterwards.

3.1 Counting Admissible Graphs

The function call $f(n, k, m, 0)$ with f recursively defined as shown by (1) counts the number of admissible level graphs within the given parameters. There, r_n, r_m, and r_l are the numbers of remaining vertices, edges, and levels, respectively. If the result should be proper, then $n_p := n_0$ determines the number of vertices recursively placed on the preceding level. For non-proper graphs $n_p := n - r_n$ is the number of possible edges from a new vertex to the previously placed ones. Then n_0 is ignored.

$$f(r_n, r_l, r_m, n_0) = \begin{cases} 1 & \text{if } r_n = 0 \wedge r_m = 0 \\ 0 & \text{if } r_n = 0 \wedge r_m > 0 \\ 0 & \text{if } r_l = 0 \wedge (r_n > 0 \vee r_m > 0) \\ \binom{r_n \cdot n_p}{r_m} & \text{if } r_l = 1 \wedge r_n \leq w \\ 0 & \text{if } r_l = 1 \wedge r_n > w \\ z & \text{else} \end{cases}$$

$$\text{with } z := \sum_{c=0}^{\min\{r_n, w\}} \binom{r_n}{c} \cdot \sum_{d=0}^{\min\{r_m, c \cdot n_p\}} \binom{r_m}{d} \cdot f(r_n - c, r_l - 1, r_m - d, c) \tag{1}$$

In the first case of (1) all vertices and edges are uniquely placed and the recursion terminates. In both the second and third case no possibility is left to complete the graph and, thus, we are done. In the fourth case there is only one level left, which has to hold all remaining vertices r_n. If possible, then we choose the positions of the remaining r_m edges from the set of all possibilities of an edge to the vertices on the last level from all previously positioned vertices. If there is not enough space on this level there are no admissible graphs, covered by case five. In all other cases we sum up over all possibilities to draw vertices from the remaining vertices times the sum of all possibilities to draw edges from the remaining edges times the previous result.

To evaluate $f(n, k, m, 0)$, we need an overall running time of $\mathcal{O}(n^3 \cdot m^2)$ for arbitrary level graphs and $\mathcal{O}(n^4 \cdot m^2)$ for proper level graphs. For that we use dynamic programming with a $(n^3 \cdot m^2)$- or $(n^4 \cdot m^2)$-dimensional look-up table, respectively, to cache already computed intermediate results.

To generate connected level graphs the only method known so far is the restart method. However, then the above method is too slow since many restarts are necessary for sparse graphs. Thus, we use the more relaxed version (2), which only gives a uniformly drawn leveling from the set of all possible levelings. We then add the edges in an independent second step.

3.2 Algorithm

The outline of our sampling algorithm is as follows. First, a random leveling ϕ is drawn employing the same idea of dynamic programming as above. The admissible candidates are levelings which allow for at least one level graph with the given parameters. This leads to the recursive equation shown in (2). As a consequence, we need only $\mathcal{O}(n^3 \cdot m)$ time for arbitrary levelings graphs and $\mathcal{O}(n^4 \cdot m)$ time for proper levelings.

$$f(r_n, r_l, r_m, n_0) = \begin{cases} 1 & \text{if } r_n = 0 \wedge r_m = 0 \\ 0 & \text{if } r_n = 0 \wedge r_m > 0 \\ 0 & \text{if } r_l = 0 \wedge (r_n > 0 \vee r_m > 0) \\ 1 & \text{if } r_l = 1 \wedge r_n \leq w \wedge r_m \leq r_n \cdot n_p \\ 0 & \text{if } r_l = 1 \wedge (r_n > w \vee r_m > n \cdot n_p) \\ z & \text{else} \end{cases} \tag{2}$$

$$\text{with } z := \sum_{c=0}^{\min\{r_n, w\}} \binom{r_n}{c} \cdot f(r_n - c, r_l - 1, \max\{0, r_m - c \cdot n_p\}, c)$$

The differences to (1) are that we count the number of admissible edges instead of the edges actually drawn. Thus, r_m counts how many potential places for edges have to be provided by the forthcoming levels instead of the remaining edges to be drawn. In the fourth and fifth case we ensure to only count possibilities of edges of the last level to vertices on the second to last level. Further, in the else-case we need not to multiply with the number of edge combinations.

To ensure that there are no empty levels between any two non-empty levels it suffices to let in (2) the sum index c start from 1 instead of 0 if vertices have been placed yet. The same holds for arbitrary level graphs and (1).

Now that we have a fixed leveling, we uniformly draw the edges of the graph. Let $\mathbb{E}(\phi)$ be the set of potential edges which the graph to generate may contain. General leveled graphs forbid intra-level edges, i. e., $\mathbb{E}(\phi) = \{(u, v) \in V \times V : \phi(v) - \phi(u) \geq 1\}$. If the graph has to be proper leveled, then $\mathbb{E}(\phi) = \{(u, v) \in V \times V : \phi(v) - \phi(u) = 1\}$. The algorithm builds up the array representing the set of potential edges $\mathbb{E}(\phi)$ in $\mathcal{O}(n^2)$ by checking each pair of vertices if the edge connecting them is inter level (or proper, if required).

The algorithm (re)draws m-element subsets of $\mathbb{E}(\phi)$ until the corresponding graph is connected. Let s be the number of repetitions required to find such a suitable edge set. The running time for each restart is in $\mathcal{O}(m)$ as the algorithm repeatedly draws an edge from the back part of the array, which is then swapped to the front part. This is a common technique for uniformly sampling m-element subsets.

The complete algorithm is sketched in Algorithm 1. Its overall running time is in $\mathcal{O}(n^3 \cdot m + s \cdot m)$ for arbitrary and $\mathcal{O}(n^4 \cdot m + s \cdot n)$ for proper level graphs. If connectedness is not required, then any m-element edge set is suitable, resulting in an improved runtime with $s = 1$.

Input: Number of vertices n, number of edges m, number of levels k, maximum number of vertices per level w.
Output: Uniformly random simple (connected, proper) leveled graph

$V \leftarrow \{1, \ldots, n\}$
$R_V \leftarrow V$ // $|R_V| = r_n$
$r_m \leftarrow m$
$n_0 \leftarrow 0$
for $l \leftarrow 0$ **to** $k - 1$ **do**
  if *leveling shall be proper* **then** $n_p \leftarrow n_0$ **else** $n_p \leftarrow n - |R_V|$
  for $c \leftarrow 0$ **to** $\min\{|R_V|, w\}$ **do**
    $q_c \leftarrow \binom{|R_V|}{c} \cdot f(|R_V| - c, k - l - 1, \max\{0, r_m - c \cdot n_p\}, c)$
  end
  draw c from $\{0, \ldots, \min\{|R_V|, w\}$ with probability $\frac{q_c}{\sum_{d \in \{0, \ldots, \min\{|R_V|, w\}\}} q_d}$
  draw V_l uniformly from $\binom{R_V}{c}$
  for $v \in V_l$ **do** $\phi(v) \leftarrow l$
  $R_V \leftarrow R_V \setminus V_l$
  $r_m \leftarrow \max\{0, r_m - c \cdot n_p\}$
  $n_0 \leftarrow c$
end
repeat
  draw E uniformly from $\binom{\mathbb{E}(\phi)}{m}$
until (V, E) *is connected or connectedness is not intended*
return (V, E, ϕ)

Algorithm 1: Generation

In practice, we found that the lazily evaluated look-up table for the computation of the leveling is in most cases filled only sparsely. Thus, the running time is dominated by the summand $s \cdot m$ when generating sparse connected graphs, while for dense graphs, the computation of the leveling accounts for the majority of time.

3.3 Checking Parameters

First we have to check the provided (ranges of) parameters, i. e., if the desired graph(s) do exist and thus our algorithm terminates successfully. For connectedness the number of desired edges m must be large enough, i. e., $m \geq n - 1$.

For level graphs, we have to check that there is enough space to contain all vertices, i. e., $n \geq k \cdot w$. Remember that n excludes dummy vertices. Again, for connectedness, $k > 1$ is required. Further, we check that m is at most the maximum number of potential edges in any leveling.

In the proper case the leveling allowing for the highest number of edges $m_{\max}$ has a minimum (but not less than two) number of levels, each occupied by a maximum number of vertices, i. e., only capped by $\lceil \frac{n}{2} \rceil$ and w. This can be proven by a simple inductive argument. Moving a vertex v from the last level to the last but two level does not reduce the number of potential edges, as then v is

potentially adjacent not only to the vertices on the last but one level (as before), but additionally to the vertices placed on the last but three level (if there is one). Thus, if the level width was unrestricted, i. e., $w \geq \frac{n}{2}$, the most dense leveling consists in two nearly equally occupied levels. When n is not divisible by w, it is optimal if the first or last level is the single level with less than $\min\{w, \lceil \frac{n}{2} \rceil\}$ vertices, as its vertices are anyway potentially adjacent to the vertices of only one level. We obtain

$$m_{\max} = \underbrace{(l_> - 1) \cdot w_>^2}_{\substack{\text{between maximally} \\ \text{occupied levels}}} + \underbrace{w_< \cdot w_>}_{\substack{\text{between the last and} \\ \text{the last but one level} \\ \text{if } n \text{ not divisible by } w}}, \tag{3}$$

where $w_> = \min\{w, \lceil \frac{n}{2} \rceil\}$, $w_< = n \bmod w_>$, and $l_> = \lfloor \frac{k}{w_>} \rfloor$.

In the general case with long edges, the leveling allowing for the highest number of edges $m_{\max}$ has k levels, which are occupied by vertices almost equally. This is because the only edges of the complete graph, which are forbidden by a given leveling, are the intra level edges, such that minimizing the number of pairs of vertices on the same level maximizes the number of potential edges. We obtain

$$m_{\max} = \frac{l_< \cdot (l_< - 1)}{2} \cdot w_<^2 + \frac{l_> \cdot (l_> - 1)}{2} \cdot w_>^2 + l_< \cdot l_> \cdot w_< \cdot w_>, \tag{4}$$

where $w_< = \lfloor \frac{n}{k} \rfloor$, $w_> = w_< + 1$, $l_> = n \bmod k$, and $l_< = k - l_>$.

3.4 Uniformity

The primary design goal of our generator was to sample the graph uniformly from the set of all graphs with the requested properties. Note that we are referring to labeled graphs, i. e., all vertices are considered distinct and we do not make any statements about the probability distributions of graphs up to isomorphism. To our best knowledge, up to date there is no special algorithm known that uniformly samples *connected* graphs with prescribed vertex and edge numbers. Therefore, we employ the commonly used *trial method*, which repeatedly uniformly samples graphs until the first happens to be connected.

For generating embedded graphs, we uniformly sample the leveling first and then uniformly sample the order of the vertices on their level under the condition of that fixed leveling.

4 Generator

4.1 Usage

The general calling scheme for the generation of single graphs is `java -jar DAGmar.jar <parameters>`, where `<parameters>` is a valid combination of the options listed in Tab. 1. For example, the call

```
java jar DAGmar.jar -n 10 -d 1.5 -c -l 4,5 -f uniform
```

generates a connected graph with 10 vertices, 15 edges using at most 4 levels with at most 5 vertices per level. The result is written into the file `uniform.graphml`.

Table 1. Parameters for generating single graphs

Option	Description
`-n <n>`	Defines the number of vertices n. Mandatory.
`-e <m>`	Defines the number of edges m. Excludes `-d`, but one of either `-e` or `-d` is mandatory.
`-d <d>`	Defines the density $d = \frac{m}{n}$, i. e., the ratio of edges compared to the vertices. Excludes `-e`, but one of either `-e` or `-d` is mandatory.
`-c`	The graph will be connected. Optional.
`-l <k>[,<w>]`	Generate a level graph. Optional. The mandatory parameter defines the number of levels k. The optional parameter defines the maximum width $w \in \{1, \ldots, n\}$, i. e., the maximum number of (non-dummy) vertices per level (if omitted, $w = n$). Levels may be empty.
`-p`	Generate a proper graph, i. e., all edges connect vertices on consecutive levels. Optional, but only valid in conjunction with `-l` and invalid in conjunction with `-el` or `-ed`.
`-el`	Generate a level embedding for the graph, i. e., positions for the vertices on their levels. Optional, but only valid in conjunction with `-l` and invalid in conjunction with `-p` or `-ed`. Contrary to `-ed`, the routing of long edges is not determined here.
`-ed`	Generate a level embedding for the graph, i. e., positions for the vertices on their levels and crossings of long edges with level lines represented by dummy vertices. Optional, but only valid in conjunction with `-l` and invalid in conjunction with `-p` or `-el`. This may cause the generation of up to $\mathcal{O}(m \cdot k)$ dummy vertices.
`-s <seed>`	Define a seed (long int) for the internal random number generator. Optional. This ensures reproducible results. If omitted, a (pseudo) random seed will be used.
`-f <file>`	The resulting graph will be written into the GraphML file `<file>.graphml`. Optional. If it is omitted, the graph is written to the standard output (e.g., the console).

The general calling scheme for generating suites of graphs is `java -jar DAGmar.jar multi <parameters> <targetdir>`, where `<parameters>` is a valid combination of the options listed in Tab. 2. Here each `<range>` parameter is a comma separated list of ranges of the form `<l> to <h> by <s>`, which defines the set of numbers $\{r \in \mathbb{Q} | l \leq r \leq h, (r - l) \equiv 0 \pmod{s}\}$. Alternatively, the suffix `by <s>` may be omitted, which is a shortcut for `"<l> to <h> by 1"`. Further, a sole number l suffices, which is a shortcut for `"<l> to <l> by 1"`. Each parameter `<form>` can either be the same as a `<range>` or an arithmetic expression whose result will be rounded to the nearest integer. The expression may contain the variables n, m, d, k as defined in Tab. 1, the arithmetic op-

erators `+`, `-`, `*`, `/`, `^` with the usual precedence and unary functions `sqrt()`, `ceil()`, `floor()`, as well as braces `()`. Additionally, the binary operators `&` and `|` represent the minimum and maximum of two values, respectively, and have the least precedence. `-l goldenratio` is a shortcut for `-l "ceil(sqrt(2 * 1.2 * n / (1 + sqrt(5)))), ceil((1 + sqrt(5)) / 2 * k)"`, which corresponds to a bounding rectangle with an aspect ratio of $\frac{w}{k} = \frac{1+\sqrt{5}}{2}$ and an area large enough to contain all vertices (see Sect. 4.5). The resulting graphs are written into the directory `<targetdir>`.

Table 2. Parameters for generating a suite

Option	Description
`-n <range>`	Define the integer range N for the number of vertices $n \in N$.
`-d <range>`	Define the rational range D for the densities $d \in D$.
`-i <range>`	Define the number of instances which will be produced per graph size. Optional. This is an integer range rather than a single number, as this has consequences on the reproducibility. While using the same random generator the ith generated instance is (most likely) different from the $(i+1)$th. For example, the range l to h with step s causes the generation of $h+1$ graphs with the same size, but only $\lfloor \frac{h-l}{s} + 1 \rfloor$ of them will be stored. Omitting this parameter has the same effect as `-i 0`.
`-c`	The graphs will be connected. Optional.
`-l <form>[,<form>]`	Generate level graphs. Optional. The mandatory parameter defines the number of levels k. The optional parameter defines the maximum width $w \in \{1, \ldots, n\}$, i. e., the maximum number of (non-dummy) vertices per level (if omitted $w = n$). Levels may be empty.
`-p`	Generate proper graphs, i. e., all edges of a graph end on consecutive levels. Optional, but only valid in conjunction with `-l`.
`-el`	Generate a level embedding for each graph, i. e., positions for the vertices on their levels. Optional, but only valid in conjunction with `-l` and invalid in conjunction with `-p` or `-ed`. Contrary to `-ed`, the routing of long edges is not determined here.
`-ed`	Generate a level embedding for each graph, i. e., positions for the vertices on their levels and crossings of long edges with level lines represented by dummy vertices. Optional, but only valid in conjunction with `-l` and invalid in conjunction with `-p` or `-el`. This may cause the generation of up to $\mathcal{O}(m \cdot k)$ dummy vertices per graph.
`-s <seed>`	Define an integral seed for the random number generator. Optional. This ensures reproducible results. If omitted, a (pseudo) random seed will be used.
`-flat`	Omit the generation of subdirectories for different densities. Optional. All files will be placed directly under the directory `<targetdir>`.
`-f <file>`	The resulting graphs will be named `<file>_n<n>_e<e>_i<i>.graphml`. Mandatory.

4.2 Example

We created the 1960 provided graphs with the following call.

```
java -jar DAGmar.jar multi -n "20 to 400 by 20" \
  -d  "1.6 to 10.6 by 1" -i "0 to 9" -l goldenratio -c -s 308 \
  -f uniform graphs
```

We obtain for each $n \in \{20, 40, \ldots, 400\}$ and $m \in \{1.6n, 2.6n, \ldots, 10.6n\}$ ten random connected graphs with a leveling respecting the aspect ratio of the *golden rectangle* $\frac{1+\sqrt{5}}{2}$. Thus, the maximum number of vertices per level is about 1.6 times the maximum number of levels. Note that, considering graphs with a number of 20 vertices, there are no graphs with a density of 9.6 or 10.6 as even the complete graph only has 190 edges, and the suite also lacks graphs with a density of 7.6 and 8.6 because the leveling constraints allow for at most 150 edges. This is explained in Sect. 4.5.

The result is a directory `graphs` containing subdirectories for each density d, which in turn contains all graph instances with density d. The graph file names match the `uniform_n<n>_e<e>_i<i>.graphml` pattern. The generation completed after a (sequential) computation time of 198 hours and 37 minutes on an Intel Core 2 CPU at 2.66 GHz.

4.3 Seeds

In order to guarantee reproducible results, the generator uses the pseudorandom number generator of the Java API, which implements an algorithm based on a linear congruential formula proposed by Knuth [7]. The generator can be initialized by a seed. Calling the generator twice with the same parameters and seed yields the same graph.

For generating a suite of graphs, the specified seed is used to in turn generate the seeds of the individual graphs. As this allows to skip the actual generation of individual graphs without influencing the outcome of the others, the generation of different instances can executed in parallel. For an example, the execution of

```
java -jar DAGmar.jar multi -n 400 -d 2.0 -i "0 to 4" -s 1234
```

and

```
java -jar DAGmar.jar multi -n 400 -d 2.0 -i "5 to 9" -s 1234
```

on two different computers yields the same set of graphs as calling

```
java -jar DAGmar.jar multi -n 400 -d 2.0 -i "0 to 9" -s 1234
```

on a single computer.

4.4 Format

The resulting graphs are in the GraphML [6] format with some user defined node (vertex) attributes. The attribute `hierarchy.level` defines the level number from 0 to $k-1$. When using one of the embedding option `-el` or `-ed`, then `hierarchy.pos` defines the order of the vertices on a level by consecutively numbering them from 0 to at most $w-1$. Finally, only when using the option `-ed`, then there is the boolean flag `hierarchy.dummy` indicating whether or not the current vertex is a dummy. Figure 1 shows the general outline.

```
<graphml ...>
  <key id="level" for="node" attr.name="hierarchy.level" attr.type="int" />
  <key id="pos" for="node" attr.name="hierarchy.pos" attr.type="int" />
  <key id="dummy" for="node" attr.name="hierarchy.dummy" attr.type="boolean" />
  <graph edgedefault="directed">
    <node id="n0">
      <data key="level">0</data>
      <data key="pos">0</data>
      <data key="dummy">false</data>
    </node>
    <node id="n1">
      <data key="level">1</data>
      <data key="pos">0</data>
      <data key="dummy">false</data>
    </node>
    ...
    <edge source="n0" target="n1" />
    ...
    </graph>
</graphml>
```

Fig. 1. Example Document

4.5 Golden Ratio

The set of non-proper graphs bundled with this generator were generated with the `-l goldenratio` option, so that the leveling constraints k and w obey the golden ratio, i. e.,

$$w = \frac{1+\sqrt{5}}{2} \cdot k\,. \tag{5}$$

We require the rectangle defined by k and w to be large enough to support for at least $\beta \cdot n$ vertices. $\beta = 1$ would be sufficient but we also want to leave some space for variation on the placement of the vertices. Solving

$$\beta \cdot n \leq w \cdot k = \frac{1+\sqrt{5}}{2} \cdot k^2 \tag{6}$$

for k we obtain

$$k \geq \sqrt{\frac{2 \cdot \beta \cdot n}{1+\sqrt{5}}}\,. \tag{7}$$

We have chosen $\beta = 1.2$ for the generation of the bundled graphs. For graphs with $n = 20$ vertices, k is set to 4 because $k \geq \sqrt{\frac{2 \cdot 1.2 \cdot 20}{1+\sqrt{5}}} \approx 3.85$. Then $m_{\max} = \frac{4 \cdot 3}{2} \cdot 5^2 = 150$, so that in this case, the levelling constraints do not allow for graphs with a density greater than $\frac{m}{n} = \frac{150}{20} = 7.5$. We could have chosen β to be large enough for our suite to contain the graphs with 20 vertices and densities of 7.6 or even 8.6, too. But those graphs would be higher than wide and thus be far from golden ratio.

When the program detects that the suite to generate contains some infeasible combinations of n, m, k, and w, it simply skips those combinations and issues a warning.

5 Conclusion

To our best knowledge, the trial method is the only known method for generating level graphs which are drawn uniformly from the set of all level graphs with the same parameters. This is somehow unsatisfactory, as it causes long running times especially for thin graphs. Then, many results must be discarded and the algorithm restarted. There is no concrete upper bound on the number of passes, although the algorithm terminates almost surely, i. e., with probability one.

A next step will be to randomly generate level graphs with some structure, e. g., the graphs in Rome library [3] are "basically" series-parallel graphs. Thus, specialized algorithms, e. g., crossing reduction algorithms, may have a chance to detect and then exploit such structures. However, an open question is what "basically" means at all. On the other hand, on nearly complete level graphs, each crossing reduction is similarly successful as there is no good embedding with significantly fewer crossings.

A problem with real world graphs, like the North DAGs, which are a subset of the AT&T Graphs [4], is that most of them are very biased, e. g., have irregular sizes, which makes them unsuitable for benchmarks verifying asymptotic running times. In many cases either the characteristics of the graphs are virtually unexplored or the graphs are lacking most of the desired properties, such that special qualities of the algorithms cannot be evaluated.

References

1. C. Bachmaier, F. J. Brandenburg, P. Effinger, C. Gutwenger, J. Katajainen, K. Karsten, M. Spönemann, and M. Wybrow. Graph archive. In C. Demetrescu, M. Kaufmann, and P. Kobourov, Stephen Mutzel, editors, *Dagstuhl Seminar 11191 on Graph Drawing with Algorithm Engineering Methods*, volume 1(5) of *Dagstuhl Reports*, pages 52–53. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 2011.
2. C. Bachmaier, F. J. Brandenburg, P. Effinger, C. Gutwenger, J. Katajainen, K. Klein, M. Spönemann, M. Stegmaier, and M. Wybrow. The open graph archive: A community-driven effort. In M. van Kreveld and B. Speckmann, editors, *Proc. Graph Drawing, GD 2011*, volume 7034 of *LNCS*, pages 435–440. Springer, 2012.

3. G. Di Battista, A. Garg, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7(5–6):303–325, 1997. Graphs available at http://www.graphdrawing.org/.
4. G. Di Battista, A. Garg, R. Tamassia, E. Tassinari, and F. Vargiu. Drawing directed acyclic graphs: An experimental study. *J. Comput. Geom. Apppl.*, 10(6):623–648, 2000. Graphs available at http://www.graphdrawing.org/.
5. P. Erdős and A. Rényi. The evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 5:17–61, 1960.
6. The GraphML file format. http://graphml.graphdrawing.org/.
7. D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 2nd edition, 1998.
8. A. S. Rodionov and H. Choo. On generating random network structures: Trees. In P. M. Sloot, D. Abramson, A. V. Bogdanov, Y. E. Gorbachev, J. J. Dongarra, and A. Y. Zomaya, editors, *International Conference on Computational Science, ICCS 2003*, volume 2658 of *LNCS*, pages 879–887. Springer, 2003.
9. A. S. Rodionov and H. Choo. On generating random network structures: Connected graphs. In H.-K. Kahng and S. Goto, editors, *International Conference on Information Networking, ICOIN 2004*, volume 3090 of *LNCS*, pages 483–491. Springer, 2004.
10. D. B. Wilson. Generating random spanning trees more quickly than the cover time. In G. L. Miller, editor, *ACM Symposium on Theory of Computing, STOC 1996*, pages 296–303, 1996.