The pacedrawing package

Max Bannach

December 20, 2019 v1.1

Abstract

This small package allows to draw graphs and tree decompositions specified in different PACE file formats (see Section 4) using LuaLaTeX and the graph drawing engine of TikZ. The package is open source and the latest version can be obtained from `https://github.com/maxbannach/pacedrawing`.

1 Drawing Graphs and Tree Decompositions

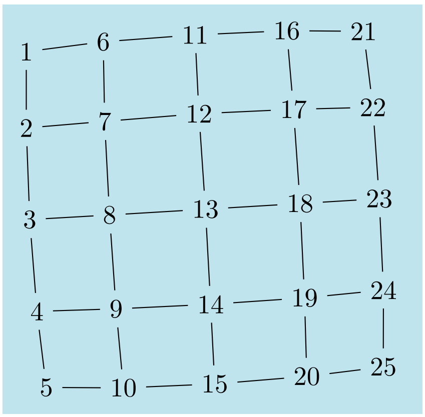
To get started load the package in the usual way:

```
\usepackage{pacedrawing}
```

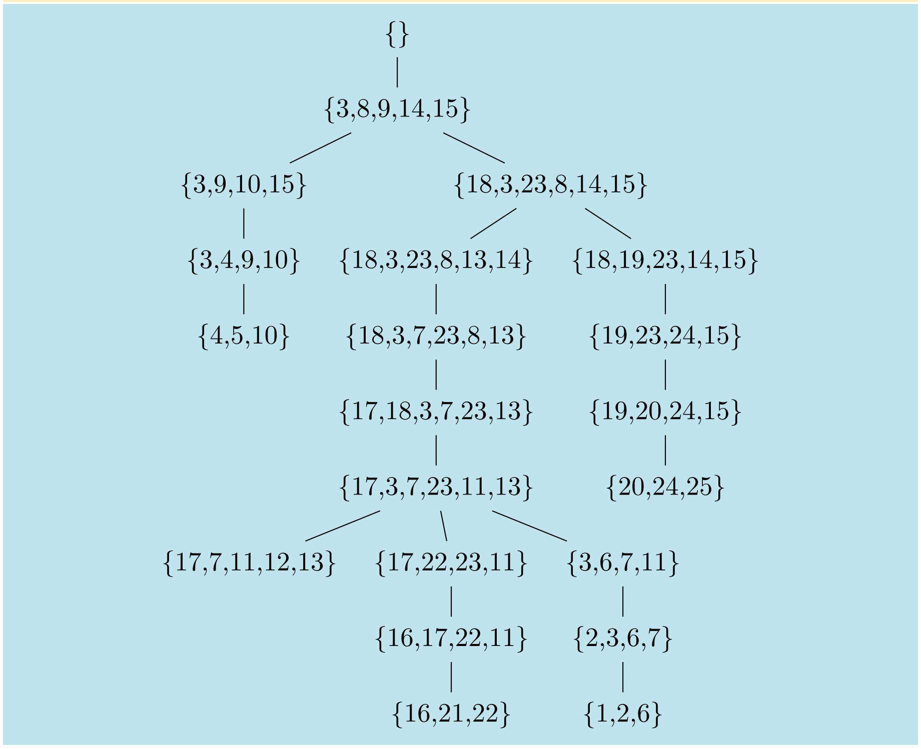
This will ensure that the packages luacode and tikz are loaded, together with the tikz libraries graphs and graphdrawing, and the graph drawing libraries force and trees.

Loading the pacedrawing package will grant access to the macro `\pace{<file>}` that can be used within a TikZ environment. The argument is the path to either a file storing a graph or a tree decomposition (see Section 4 for details). The macro will automatically detect the file format, choose a suitable layout, and generate the corresponding TikZ picture.

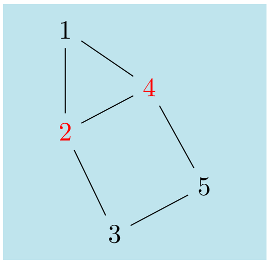
```
% draw a PACE 2016/17 graph
\tikz\pace{graphs/grid.gr};
```



```
% draw a PACE 2016/17 tree decomposition
\tikz\pace{graphs/grid.td};
```



```
% draw a PACE 2018 steiner tree graph
\tikz\pace{graphs/example.gr};
```



In more detail, the `\pace` macro will synthesize the TikZ `\graph` command with the loaded graph inserted. The ids of the vertices of this graph will be as in the encoded file, i. e., in most cases $\{1,\ldots,n\}$. If the input was a tree decomposition (`.td`) the vertices are created with text, where the text of each vertex is the content of the corresponding bag in set representation (e. g., if bag 1 contains the vertices $\{3,5,6\}$ there will be a vertex with id "1" and content "{3,5,6}").

2 Configure the Layout

As in the graphs library of TikZ, the appearance of `\graph` environments generated by pacedrawing can be modified by the key-value system of TikZ. If the reader is not familiar with this system, the TikZ manual is a good starting point before proceeding in this section. As the `\pace` macro synthesizes the `\graph` command on the fly, we can not add keys directly to it. Instead, the pacedrawing package provides the following three ways to configure the layout: (a) globally change the layout of graphs of a certain file format, (b) locally change the layout of the next drawn graph, (c) directly work with the generated TikZ string "by hand."

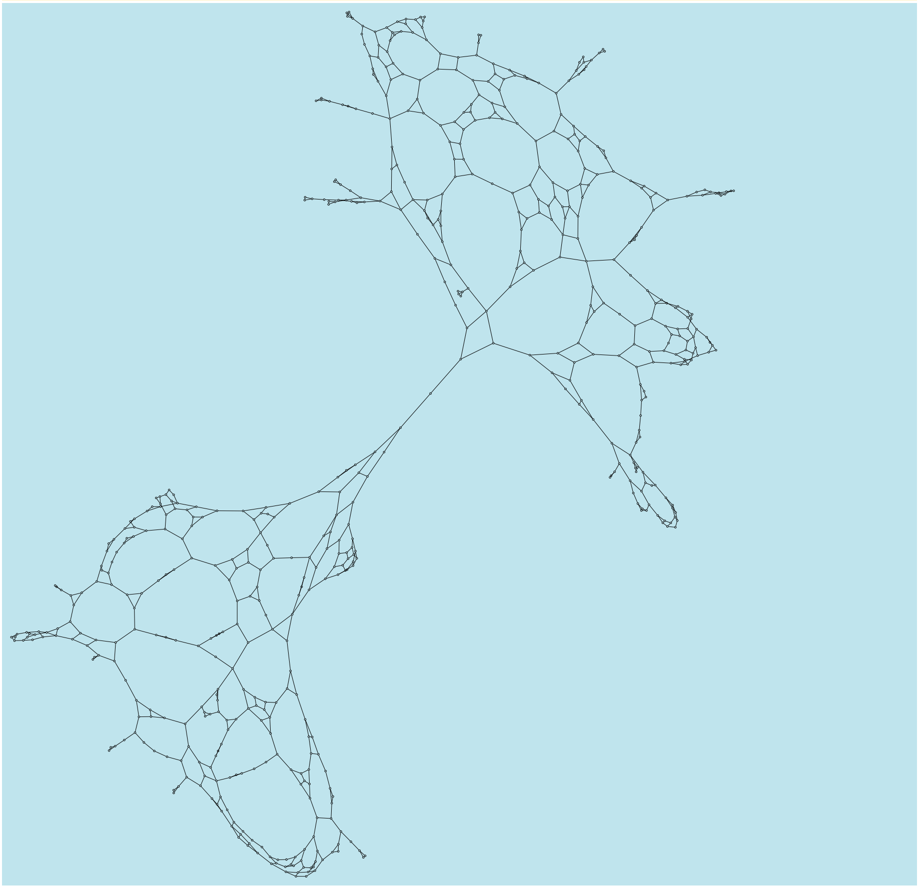
2.1 Modify the Layout Globally

Each graph drawn by the `\pace` macro obtains a predefined style, which depends on the file format (see Section 4 for an overview). By overriding this style, you may change the appearance of all drawn graphs. The keys and their default implementations are the following:

```
graphs/pace/dimacs/.style = {
  spring electrical layout
},
graphs/pace/treedecomposition/.style = {
  tree layout
},
graphs/pace/stp/.style = {
  spring electrical layout
},
graphs/pace/edgelist/.style = {
  spring electrical layout
},
graphs/pace/embedding/.style = {
  use existing nodes,
  edges = {thin, color = lightgray}
},
graphs/terminal/.style = {
  color = red
}
```

The `pace/dimacs` style is applied to PACE 2016/17 graphs (and to graphs stored in the similar dimacs file format), the `pace/treedecomposition` is used to draw tree decompostions, the `pace/edgelist` style is used for PACE 2016/17 Track B graphs (which are stored in a simple edge list), and `pace/stp` is used to draw PACE 2018 steiner tree instances (where terminals are marked with the style `pace/terminal`).

```
\tikzset{
  graphs/pace/dimacs/.style = {
    spring electrical layout,
    empty nodes,
    node distance = 1.5cm,
    nodes = {draw, circle, inner sep=0.3mm, semithick},
    edges = {semithick}
  },
}
\tikz\pace{graphs/ex045.gr};
```



Turn On Edge Weights. Some graph formats store *weighted* graphs, i. e., a number is assigned to each edge. We do not draw these numbers by default, as too many edge weights make the picture overwhelmingly confusing. However, you may turn edge weights on or off with the following commands:

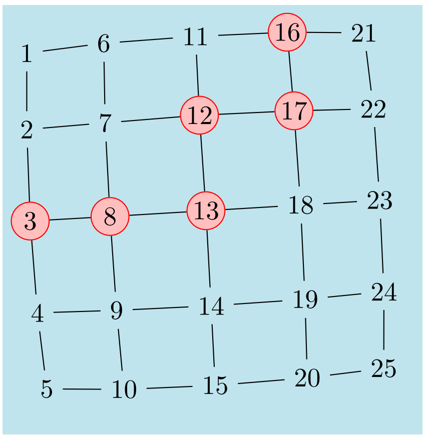
```
\paceShowEdgeWeights
\paceHideEdgeWeights
```

3 Modify the Layout Locally

In the previous section we have seen how we can change the layout of all graphs of a certain type. We may change these settings between each call to `\pace` to obtain different layouts, but all in all it is a quite global approach. If we wish to "fine-tune" the layout of a certain graph, the pacedrawing package provides us with macros to apply certain styles to individual nodes, edges, or to the whole graph. These macros will only effect the *next call* of `\pace` and no further layouts.

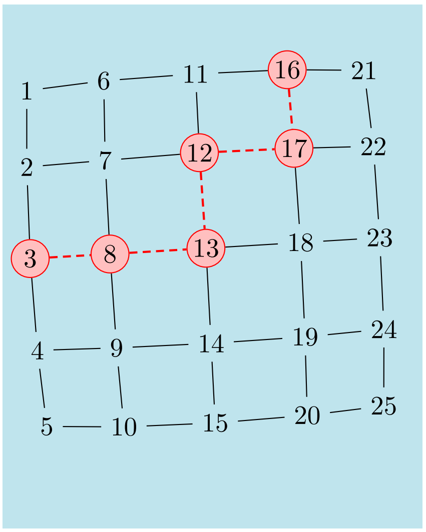
In detail, the package provides the three macros `\paceApplyVertexStyle` (which adds a specific style to a set of vertices), `\paceApplyEdgeStyle` (which adds a specific style to a set of edges), and `\paceApplyGraphStyle` (which adds a style to the whole graph). Let us go through these macros this with a running example, in which we modify the grid graph from the first section. We may start by highlighting a separator in the graph with a custom style.

```
\tikzset{
  separator/.style = {
    draw = red,
    fill = red!25,
    inner sep = 0pt,
    minimum width = 0.5cm,
    circle
  }
}
\paceApplyVertexStyle{3, 8,
  12, 13, 16, 17}{separator}
\tikz\pace{graphs/grid.gr};
```



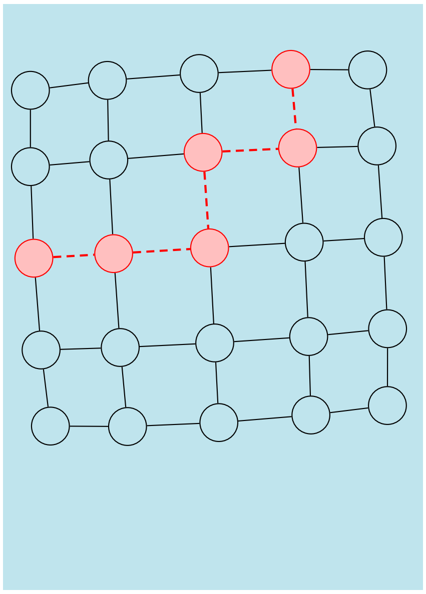
Note that the `\paceApplyVertexStyle` gets a *set* of vertices and a single style, which then is applied to all these vertices. Let us proceed by also highlighting the edges between these vertices as well. This is possible with the `\paceApplyEdgeStyle` macro, which works in exactly the same way as the `\paceApplyVertexStyle` macro, with the only difference that it obtains a set of *pairs* of vertices as argument.

```
\tikzset{
  rededge/.style = {
    thick,
    densely dashed,
    color = red
  }
}
\paceApplyVertexStyle{3, 8,
  12, 13, 16, 17}{separator}
\paceApplyEdgeStyle{ {3,8},
  {8,13}, {12,13}, {12,17},
  {16,17} }{rededge}
\tikz\pace{graphs/grid.gr};
```



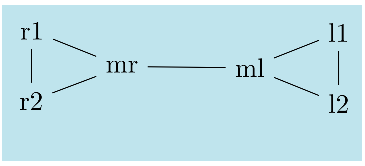
It is important to note that the graph representation internally is directed – in the same way as in the input file format. For instance, if the input file specified the edge $(3, 8)$, we may only apply a style to edge $(3, 8)$ and not to $(8, 3)$. The reason is that we may wish to add the backward edge additionally, and we therefore wish to distinguish between both cases. More about this topic will follow later in this section. For now, let us finish the example by applying a custom style to the graph as well.

```
\tikzset{
  graphs/mygraph/.style = {
    empty nodes,
    nodes = {draw,
      circle,
      inner sep = 0pt,
      minimum width=0.5cm
    }
   }
}
\paceApplyVertexStyle{3, 8,
  12, 13, 16, 17}{separator}
\paceApplyEdgeStyle{ {3,8},
  {8,13}, {12,13}, {12,17},
  {16,17} }{rededge}
\paceApplyGraphStyle{mygraph}
\tikz\pace{graphs/grid.gr};
```

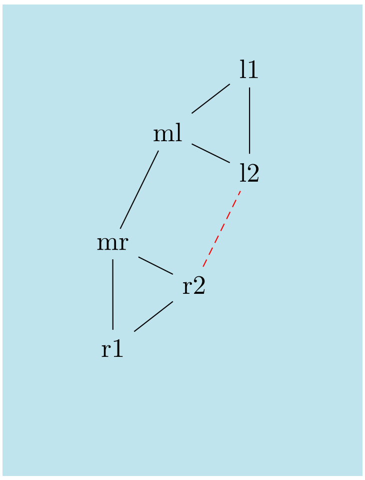


As a final note to local layout modifications let us mention that it is also possible to apply a custom style to vertices or edges that are not present in the graph. This will *add* these vertices and edges to the graph. Besides the obvious advantage that we can modify the graph, big advantage is the possibility to add *invisible* edges – another convenient way to influence the layout produced by the graph drawer.

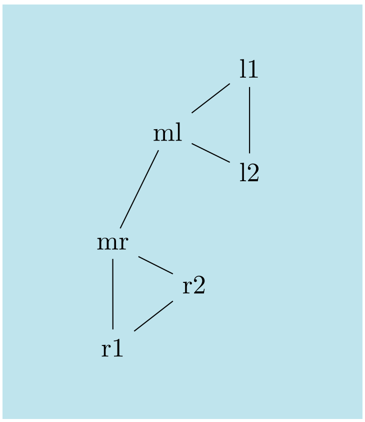
```
\tikz\pace{graphs/example.graph};
```



```
\tikzset{
  newedge/.style = {
    draw,
    densely dashed,
    color = red
  }
}
\paceApplyEdgeStyle{
  {"r2", "l2"}
}{newedge}
\tikz\pace{graphs/example.graph};
```



```
\tikzset{
  invisible/.style = {
    opacity=0
  }
}
\paceApplyEdgeStyle{
  {"r2", "l2"}
}{invisible}
\tikz\pace{graphs/example.graph};
```



Observe that in the example above we have used a graph stored as simple edge list, in which the ids of the vertices are *strings* (in previous examples the vertices were always integers). To encounter this circumstance, we have to address the node names in quotes (as seen in the `\paceApplyEdgeStyle` command).

Manually Clear Local Styles. Normally the situation is quite clear: we first apply some local styles, then we call `\pace` to draw the graph, and the styles get cleared for the next run. However, this approach may run into trouble if we call `\pace` *conditionally*, for instance by the use of tikzexternalize. In those cases we should ensure that all the local modifications are also conditionally, as they otherwise may be applied to the wrong graph. If, however, this is not possible for some reason, you may use the `\paceClear` command to manually clear all styles (which is exactly the function that `\pace` would call after it finishes).

Directly Apply Options to the Pace-Command. Instead of adding additional styles using `\paceApplyGraphStyle`, we may also use `\pace` with an *optional* argument. For instance, the situation in the aforementioned example could be mimicked with `\pace[mygraph]{graphs/grid.gr}`. However, the difference is that by using the optional argument we will *not* add the style as additional style (as `\paceApplyGraphStyle` would do), but will *override* the default style. Therefore, to obtain the same result we have to write `\pace[pace/dimacs,mygraph]{graphs/grid.gr}` – but of course, we can apply any other layout and style as well.

3.1 Working Directly With the Graph or the TikZ String

As mentioned in the introduction, the pacedrawing packages internally creates a TikZ string representing the parsed graph. Before this string is created, it creates an internal representation of the graph in form of a Lua table. If the global or local layout modification is not enough for your needs, you may modify the internal graph representation or work directly with the TikZ string. In order to do so, the `\pace` command will call two Lua functions, which you may override.

```
function pacedrawing.willGenerateString(g)
end

function pacedrawing.didGenerateString(g, tikz)
end
```

The first function is called *after* the graph is parsed and *before* the TikZ string is created. The argument `g` is the graph representation, which is a Lua table with the following fields:

```
  g = {
    options = {},
    V = {},
    E = {}
  }
```

Here `options` is an array of strings with the options that are applied to the whole graph. The table `V` contains the vertices, and we have `V[v] = {}` if `v` is a vertex of the graph, i. e., we store a table at the vertex id, if the vertex is part of the graph. This table may contain an array of strings `V[v].options` and a key `V[v].label` storing an string. The options array contains all options (e. g., styles) that are applied to the given vertex and the label is the text shown for the vertex (the id is used if this key is `nil`). The table `E` works similar and stores *directed* edges, i. e., we have `E[u][v] = {}` if `(u,v)` is an edge in the graph. This table may contain options applied to the edge (but, other than for vertices, these are not stored in an nested table but rather directly). Edges also do not have a label key (as this is an option anyway).

You may modify the table `g` to change the graph. For instance, you may add vertices or edges, delete them, or add new styles to them. *After* the function `pacedrawing.willGenerateString` was called and your changes may have taken place, the `\pace` command will generate the corresponding TikZ string and pipe it back to TEX. Afterwards, the second function from above (`pacedrawing.didGenerateString`) is called, which obtains the same table `g` as above, and a table `tikz` as arguments. Here `tikz` essentialy is an array of substrings of the final TikZ string, i. e., `table.concat(tikz,"\n")` creates the desired string. An intended usecase of this function is, for instance, to store the created TikZ string in an external file, modify it "by hand", and later include it manually to the TEX-document. Note that this is the last point where you may have access to `g` or the corresponding string, as the internal data structures are cleared afterwards.

4 The File Format

The pacedrawing package supports file formats of multiple PACE challenges, as well as file formats that are "similar" to those. In fact, the package is implemented in a general fashion such that any file format "close" to the described ones will be parsed as far as possible. There is no need to specify the file format, the `\pace` macro will automatically detect and parse any supported file by itself. In particular, the following formats are supported.

4.1 The Dimacs Format

The PACE 2016 and 2017 Track A challenge used a file format similar to the dimacs format. The PACE format is stored in `.gr` files and a complete description can be found here:

`https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`.

The similar format used by some Dimacs challenges, stored in `.dfg` files, can be parsed, too.

For example, a file may look like this:

```
c This file describes a path with five vertices.
p tw 5 4
1 2
2 3
c we are half−way done with the instance definition.
3 4
4 5
```

4.2 The Tree Decomposition Format

The output format of the PACE 2016 and 2017 Track A challenge describes a tree decomposition. The used file extension is `.td` and a detailed description can be found at:

`https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`.

For example, a file may look like this:

```
c This file describes a tree decomposition with 4 bags
s td 4 3 5
b 1 1 2 3
b 2 2 3 4
b 3 3 4 5
b 4
1 2
2 3
2 4
```

4.3 The Simple Edgelist Format

The PACE 2016 and 2017 Track B challenge used a simple edge list format, which only contains the edges as pairs of vertices; and comment lines with a #. These graphs are stored with the ending `.graph`. A detailed description can be found at:

`https://pacechallenge.wordpress.com/pace-2016/track-b-feedback-vertex-set/`.

For example, a file may look like this:

```
ml mr
l1 ml
l2 ml
l1 l2
r1 mr
r2 mr
r1 r2
```

4.4 The STP File Format

The PACE 2018 challenge about the steiner tree problem uses a file format similar to the STP format. These files store not only the graph, but also the corresponding set of terminals. An exact specification of this format, which is usually stored in `.stp` files, can be found at:

`https://pacechallenge.wordpress.com/pace-2018/`.

For example, a file may look like this:

```
SECTION Graph
Nodes 5
Edges 6
E 1 2 1
E 1 4 3
E 3 2 3
E 2 4 4
E 3 5 10
E 4 5 1
END

SECTION Terminals
Terminals 2
T 2
T 4
END

[SECTION Tree Decomposition  \\only in Track 2
...
END]

EOF
```

4.5 The Tree File Format

The goal of PACE 2020 is to compute treedepth decompositions of undirected graphs. The input format is unchanged, but for the output a simple `.tree` format is defined that specifies such decompositions. The file contains $n + 1$ lines, where the first line describes the depth of the tree, while each line i defines the parent of vertex $i - 1$ (0 means that this vertex is the root). The exact specification can be found at:

`https://pacechallenge.org/2020/td`.

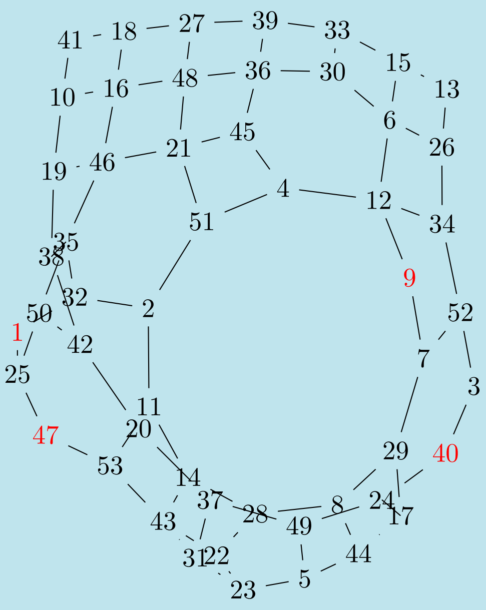
For example, a file may look like this:

```
3
2
3
0
3
4
```

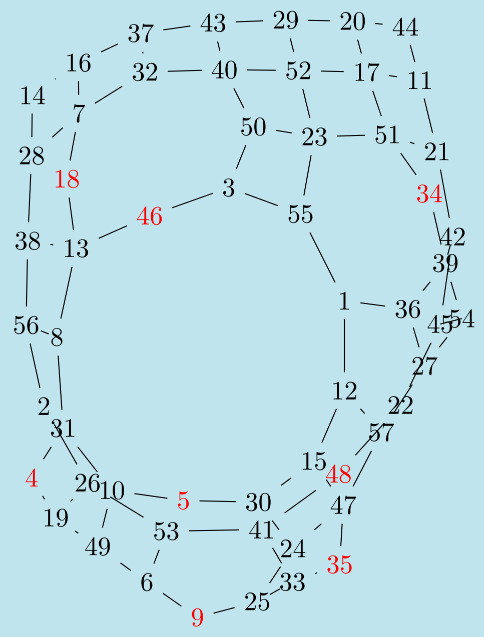
5 Some Examples

The following shows some of the (smaller) public graphs of various PACE challenges with different layout options.

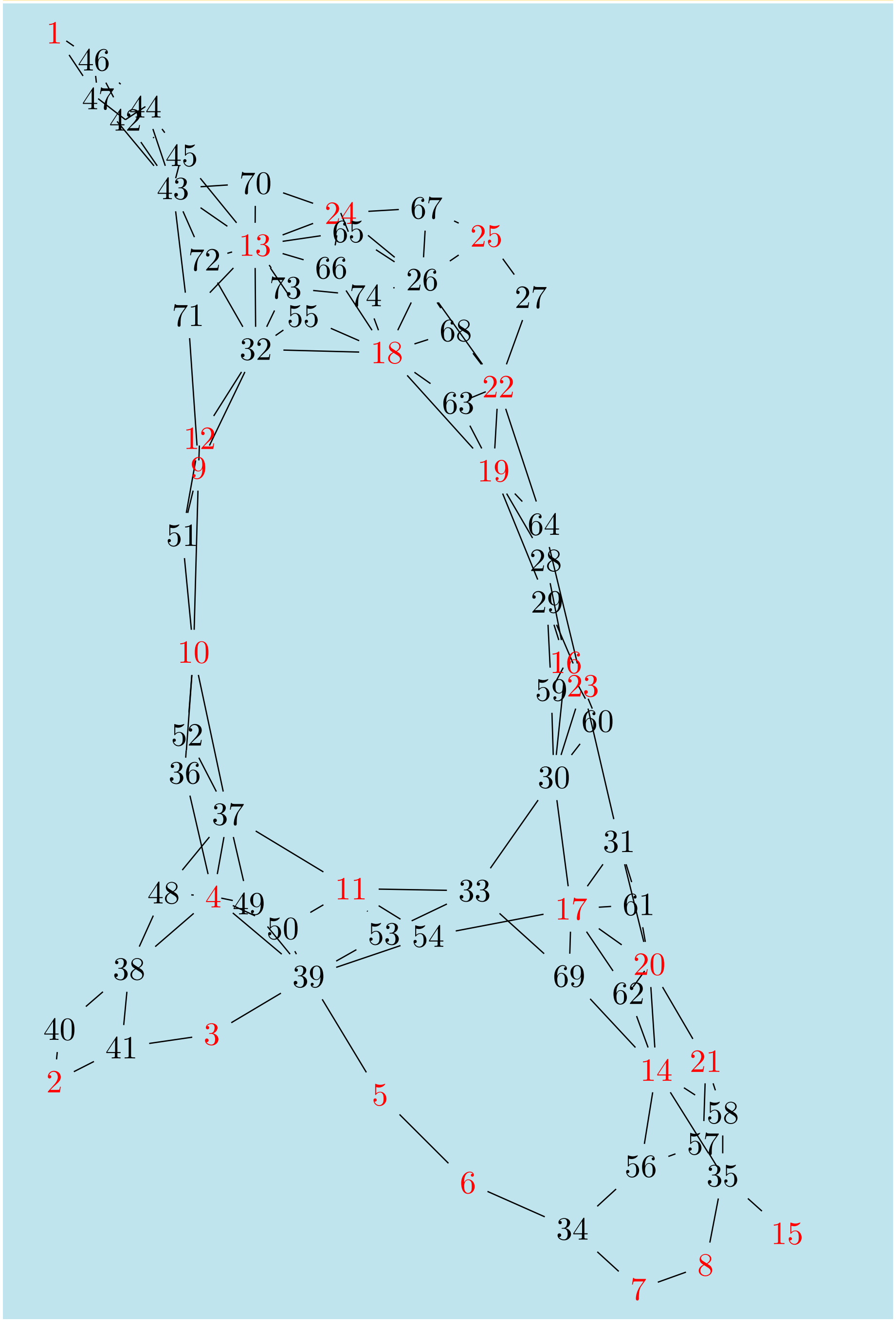
```
\tikz\pace{graphs/2018-A/instance001.gr};
```



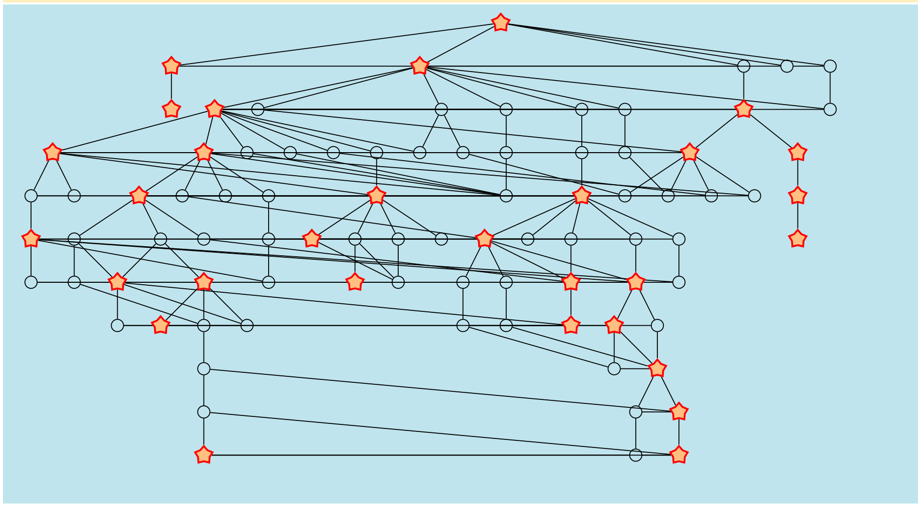
```
\tikz\pace{graphs/2018−A/instance009.gr};
```



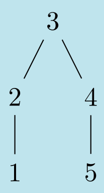
```
\tikz\pace{graphs/2018-B/instance001.gr};
```



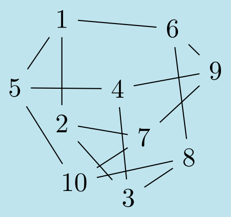
```
\tikzset{
  graphs/pace/stp/.style = {
    tree layout,
    empty nodes,
    nodes = {draw, circle, inner sep=1mm, semithick},
    edges = {semithick}
  },
  graphs/terminal/.style = {
    rectangle,
    very thick,
    color = red,
    fill = orange!50
  }
}
\tikz\pace{graphs/2018–B/instance003.gr};
```



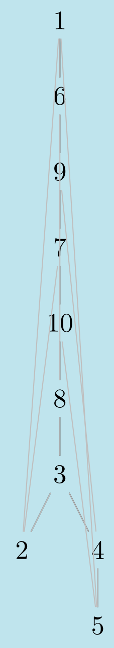
```
\tikz{
  \pace{example.tree};
}
```



```
\tikz{
  \pace{2020-001.gr};
}
```



```
\tikz{
  \pace{graphs/2020-001.tree};
  \pace[pace/embedding]{graphs/2020-001.gr};
}
```


1
6
9
7
10
8
3
2
4
5