Supporting layout routines in MetaPost

Wentao Zheng

Abstract

METAPOST is known as a powerful graphics drawing language. However, METAPOST dose not provide any mechanisms to automatically lay out graphic objects. In this article, we present two approaches to help METAPOST users to automatically or semi-automatically lay out objects that they are drawing.

1 Introduction

METAPOST is widely adopted by LaTeXers to generate high quality graphics in their documents. It is well known for its precisely controlled geometric restriction, textual label integration with TeX, and extendable macros. A variety of packages/macros, like MetaUML, m3D, have been created that allow LaTeX users to draw high quality graphics easily and professionally. Although these packages provide us with the functionality to draw objects and links, it is often noticed that we still need to spend a lot of effort laying out the objects we are drawing.

Let's take a look at an example diagram (Figure 1) from John Hobby's METAPOST manual [1] on page 63 (it may appear on different pages in different versions of the METAPOST manual). It is a simple finite state diagram that has five states, ten arrow links and corresponding labels. There are several routines in the source code that have something to do with the diagram's layout, i.e., state (node) positioning, arrow (link) direction tuning and label positioning. And these routines take at least half of the total source code. The problem is clear now: can we develop a METAPOST package that provides users with automatic or semi-automatic layout routines?

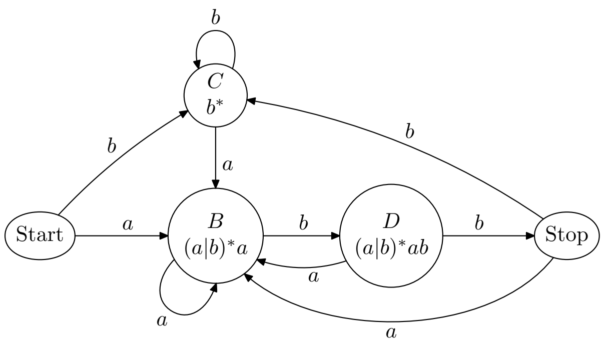

Figure 1: An example diagram from the METAPOST manual

The problem of automatically laying out general graphs is not new. A number of academic activities, such as the International Symposium on Graph Drawing, have been existed for decades on the research of algorithms and methods for graph visualization. Aesthetics and computational complexity are the major concerns in the research. However, algorithms/methods for general graph drawing problem with both efficiency and aesthetics have not been presented. Most current research is focused on special graph drawing problems or approximated solutions.

Therefore, the question that whether we can develop a (semi-)automatic layout package for METAPOST cannot be answered simply by a 'Yes' or 'No'. In this article, we first address some issues and challenges in developing a layout package for METAPOST (Section 2), then propose several possible approaches (Section 3). Some future work is also discussed at the end (Section 4).

2 Challenges

Although we will not develop a general method to lay out graphs, even (semi-)automatic layout is difficult.

First of all, METAPOST is not an object-oriented programming language. Although it has facilities to simulate some aspects of OO programming, there is no "base object" in METAPOST. Therefore it is not easy to write a layout routine that can be applied on different graphic objects. For example, METAPOST's `boxes` macro introduces a kind of object (box and circle) with properties

`c n e s w ne nw se sw`

so we can manage an object's positions by manipulating geometric relations on those properties. But what if another user wants to use the routine to lay out objects without the properties mentioned above? A practical, though not friendly, solution is to specify rules (properties and methods) to objects that need to use the layout routine.

Secondly, developing a purely automatic layout routine is difficult, even impossible. In the research of graph drawing, practical algorithms exist only for special graphs, such as trees, DAGs (direct acyclic graphs). There exists no general layout method for an arbitrary graph with satisfactory aesthetics and acceptable running time.

The first solution to this problem is designing on demand. That is to say, to develop layout routines for specific graphs. Graphviz is a graph drawing program that takes this approach, providing several practical routines to draw graphs.

Another solution is using the KISS (keep it simple, stupid) principle. That is to keep layout routines small, easy to understand and practical to use. However, by taking this approach, another problem arises: how to design those routines? That is, what to provide, and what to omit? It has been seen in some diagramming tools that small layout routines are very useful and easy to use. For example, "horizontal or vertical alignment", "equal height or width" are good layout routines. But these examples are not enough; we need to design more routines and expect some combinations of them will generate very useful and sophisticated results.

3 Possible approaches

In this section, we will present several approaches to developing layout routines in METAPOST.

3.1 Reusing Graphviz

Actually, Graphviz is a set of programs for automatically specifying graph layout:

dot makes hierarchical or layered drawings of directed graphs.

neato and **fdp** make spring model layout.

twopi makes radial layout.

crico makes circular layout.

For more information, please take a look at their web site: `http://www.graphviz.org`.

There is a LaTeX package called `dot2tex` that makes use of Graphviz to generate PSTricks and PGF/TikZ commands in LaTeX documents. For detailed information, please take a look at their web site: `http://www.fauskes.net/code/dot2tex`. It is obvious that we can take a similar approach to adopt Graphviz in METAPOST.

In the rest of this section, we are going to use **dot** as an example to show how to use Graphviz to generate automatic layout routines for METAPOST.

Let's first take a look at how to represent the simple graph in Figure 2 in the **dot** language:

```
digraph G {
    A -> B [label = "x"];
    A -> C [label = "y"];
    C -> B [label = "z"];
}
```

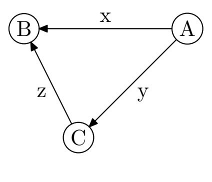

Figure 2: A simple graph

By using the **dot** program, a diagram with automatic layout is generated, as shown in Figure 3. We can see that nodes are separated with proper distances, links are placed with appropriate angular resolutions, and labels are displayed at the right

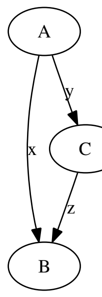

Figure 3: Diagram generated by **dot**

places. Although the diagram is not as "pretty" as the one in Figure 2, the layout is at least readable.

In **dot**, some mechanisms are provided to tune the graph layout with manual control of nodes, links, and labels. We won't introduce them here because we want to keep our focus on the adoption of **dot** in METAPOST.

dot supports several kinds of output format, such as plain text, PostScript, SVG, and binary images. Among those, plain text is the easiest to reuse in METAPOST. The following text is the compiled output of the aforementioned **dot** source code.

```
digraph G {
  node [label="\N"];
  graph [bb="0,0,85,212"];
  A [pos="27,194",
     width="0.75",
     height="0.50"];
  B [pos="27,18",
     width="0.75",
     height="0.50"];
  C [pos="58,106",
     width="0.75",
     height="0.50"];
  A -> B [label=x, pos="e,23,36
                        23,176
                        ...
                        21,46",
                 lp="18,106"];
  A -> C [label=y, pos="...",
                 lp="46,150"];
  C -> B [label=z, pos="...",
                 lp="47,62"];
}
```

We can see that layout information can be extracted from the output. Graph nodes, such as `A`, are indicated by `pos` (position), `width`, and `height`, while links, such as `A -> B`, are indicated by `label`, `pos` (path points), and `lp` (label position). It is easy to automatically extract the layout information from the output and then use it in METAPOST.

In order to use **dot** in METAPOST, we should firstly write METAPOST/**dot** hybrid code (named as an **MPdot** file) as follows:

```
input boxes;
beginfig(1);
  circleit.a(btex A etex);
  circleit.b(btex B etex);
  circleit.c(btex C etex);

  begindot % begin of dot code
    digraph G {
      a -> b [label = "x"];
      a -> c [label = "y"];
      c -> b [label = "z"];
    }
  enddot % end of dot code
endfig;
```

It is noticed that the content between `begindot` and `enddot` is written in **dot** language. We are using METAPOST suffixes, such as `a`, instead of their labels, such as `"A"`, to represent nodes in **dot**. This is because we want to connect the **dot** with the METAPOST code. We show the importance and desirability of doing this below.

In the next step, we use a program to parse **dot** code from the **MPdot** file and rewrite it into another intermediate **dot** file (named an **IMdot** file) for compilation. For those nodes represented by METAPOST suffixes, such as `a`, their respective definitions, like `circleit.a(...)`, will be used to determine their dimensions (width and height). The following code shows what the generated **IMdot** file looks like.

```
digraph G {
  a [ height = 0.19595, width = 0.19595,
      label = "" ];
  b [ height = 0.19174, width = 0.19174,
      label = "" ];
  c [ height = 0.19313, width = 0.19313,
      label = "" ];
  a -> b [ label = "x" ];
  a -> c [ label = "y" ];
  c -> b [ label = "z" ];
}
```

We can see that nodes are defined with `height`, `width`, and `label` properties. The height and width of a node are calculated based on the corresponding suffix defined in METAPOST code. This is the reason why we use METAPOST suffixes to represent **dot** nodes in the **MPdot** file.

The **IMdot** is then sent to the **dot** program for compilation, and layout information is generated. We can extract the layout information from the output and generate METAPOST code to replace the **dot** code in **MPdot** file, resulting in the final METAPOST file. The following is the final METAPOST file, in which the **dot** code is replaced by generated METAPOST code. After compilation, it outputs a graph shown in Figure 4.

```
input boxes;
beginfig(1);
  circleit.a(btex A etex);
  circleit.b(btex B etex);
  circleit.c(btex C etex);

  % the following code is auto generated
  a.c = (7pt,141pt);
  b.c = (7pt,7pt);
  c.c = (24pt,74pt);
  drawunboxed(a,b,c);
  draw fullcircle scaled 0.19in
       shifted a.c;
  draw fullcircle scaled 0.19in
       shifted b.c;
  draw fullcircle scaled 0.19in
       shifted c.c;
  label(btex $x$ etex, (4pt,74pt));
  label(btex $y$ etex, (19pt,108pt));
  label(btex $z$ etex, (19pt,40pt));
  drawarrow
  (6pt,134pt)..(0,62pt)..(6pt,14pt);
  drawarrow
  (9pt,134pt)..(16pt,105pt)..(22pt,81pt);
  drawarrow
  (22pt,67pt)..(15pt,38pt)..(9pt,14pt);
endfig;
```

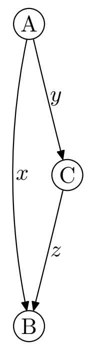

Figure 4: Graph generated by METAPOST with **dot** layout information

With TeX labels integrated and METAPOST's curve path tuning, the graph shown in Figure 4 looks better than that in Figure 3.

The approach of reusing Graphviz that we just explained can be summarized in Figure 5. At first, a user writes a **MPdot** file, in which the **dot** code is translated into a **IMdot** file. The **IMdot** file is then sent to **dot** for compilation, and layout information is returned. The information is extracted and translated into METAPOST statements to replace the **dot** code in **MPdot** file, resulting a pure METAPOST file, based on which the final graphics is generated.

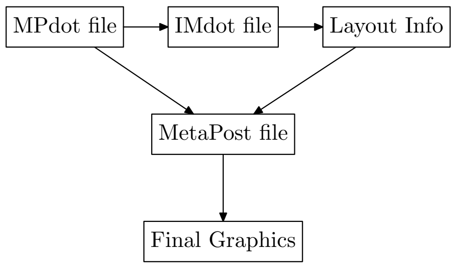

Figure 5: Approach of reusing Graphviz (**dot**)

3.2 Small, stupid routines

As we can see in the previous section, Graphviz is not a perfect layout tool. Users with strong sense of aesthetics may not be satisfied with Graphviz's result. This is why we propose another approach: designing small and stupid routines.

Trivial layout routines have long existed in various diagramming software and user interface designing tools. For example, you can select a number of graphic objects, make them align horizontal, from left to right, and have same width and height. These routines, including alignment, order, and dimension specification are very useful when we are drawing diagrams. So we are going to extend them and make them available in METAPOST.

Generally, there are three types of graphic objects in diagrams, i.e., shapes, links, and labels. A shape is an object with a surrounding path (usually closed), such as a rectangle, ellipse, etc. A link is a path connecting two shapes, usually parameterized with a start shape and end shape, such as arrow link, line link, etc. A label is a textual container containing formatted text, and a transparent surrounding path. Figure 6 shows two shapes (rectangles) connected by an arrow link labeled by "Label".

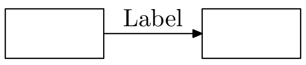

Figure 6: Three types of graphic objects

For a shape, its internal properties should be set by end users or calculated based on its inner label. For a link, the objects it connects to should be set by end users. For a label, only the textual content should be set by end users. Therefore, layout routines should care about where a shape is located, what path points a link should go through, and where a label is placed.

As we mentioned in Section 2, METAPOST is not an object-oriented language. So it is difficult to design routines for different graphic objects. For simplicity, let us focus on laying out graphic objects defined by the `boxes` package, i.e., `box` and `circle`. The common attributes they share are (as shown in Figure 7):

`c`	center point of a shape
`n`	north point of a shape
`s`	south point of a shape
`e`	east point of a shape
`w`	west point of a shape

Another very important attribute is `bpath`, which is the surrounding path of the shape. We can use this path to determine a shape's bounding area, and ensure that a link's end points are tightly connected on the path.

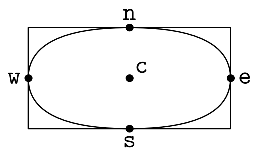

Figure 7: Attributes of a `box` or `circle`

Let's focus on shape layout routines first. The simplest and most frequently used is linear alignment. That is to say, align a number of objects through a line. Consider the following macro

```
line_align <dir>,<gap>,<objects>
```

It uses `dir` (the direction of the line), `gap` (distance between consecutive objects), and `objects` (objects to be aligned) as parameters.

Sometimes, line alignment is not sufficient, so we present another way to align objects: general path alignment. It looks like

```
path_align <path>, <objects>
```

The parameter `path` specifies a path (line or curve) along which the `<objects>` are placed and separated evenly.

Being different from shapes, there is no need to specify the location of a link, because it is used to connect two shapes (in most cases). After the laying out of shapes, the question of where links start and end is quite easy to answer. So link layout should be focused on how we link two shapes: on a straight line, curve or orthogonal polyline. The following macros are used to specify how to layout links:

```
line_link <start_shape>, <end_shape>
curve_link <start_shape>, <start_dir>,
```

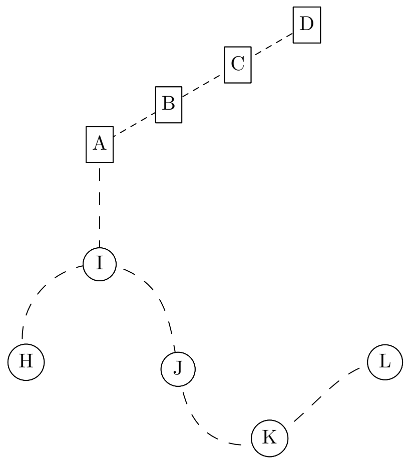

Figure 8: Alignment of shapes

```
            <end_shape>, <end_dir>
orth_link <start_shape>, <start_side>,
          <end_shape>, <end_side>
```

The `line_link` is used to connect `start_shape` and `end_shape`. The `curve_link` takes two other parameters, i.e., `start_dir` (the direction of link path at the start point) and `end_dir` (the direction of link path at the end point). Similarly, the `orth_link` takes parameters `start_side` (north, east, south, or west) and `end_side`. For the first and second link macros, it's easy to implement. But for the last one, more effort is required, and we are not going to solve it in this article. Figure 9 shows three types of link layout (the orthogonal one is drawn manually, just to show what it looks like).

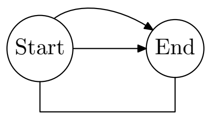

Figure 9: Layout of links

A label's layout is a little complicated. First of all, labels can be treated as a special kind of shape without a surrounding path. It is natural that we let shape layout routines, such as linear alignment, be applicable for labels. Besides, labels have other means for layout. An example is creating a label for a link or a shape. We name this kind of label an association label.

Let's start with association labels for links. Because a label is usually placed somewhere along the path of a link, we can use the following macro to lay out the label.

```
link_label <label> <link> <portion>
```

`label` defines what textual content to be displayed, `link` is a link object suffix, and `portion` is a number between 0 and 1 that denotes where the label is placed along the link path.

Association labels for shapes are easier to handle. In most cases, a shape's properties like `n` and `c` is sufficient for manipulating the positions of labels. Figure 10 shows a number of labels for links and shapes.

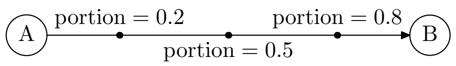

Figure 10: Layout of association labels

After introducing some small and stupid layout routines, we suggest users use them in the following order:

1. Declaring shapes with macros like `boxit` and `circleit`
2. Laying out shapes by using the aforementioned routines
3. Declaring and laying out links
4. Declaring and laying out labels

The reason for this order is that label positions rely on links and shapes, and link paths relies on shapes. So it is necessary to first lay out shapes, then links, and do labels last.

4 Future work

In this article, we propose two approaches of supporting layout routines in METAPOST, to make the drawing of diagrams convenient and aesthetic. We introduce them separately with detailed explanation and some examples. However, the methods presented in this article are at a very early stage; refinement and extension must be done to make them more practical. This is planned for the near future.

References

[1] John Hobby, "METAPOST: A User's Manual", `ctan:graphics/metapost/base/manual/mpman.pdf`.

◇ Wentao Zheng
IBM China Research Laboratory
`zhengwt (at) cn dot ibm dot com`