A Simple Language for Novel Visualizations of Information

Wendy Lucas[1] and Stuart M. Shieber[2]

[1] Computer Information Systems Department, Bentley College, Waltham, MA, USA
wlucas@bentley.edu
[2] Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA shieber@deas.harvard.edu

Abstract. While information visualization tools support the representation of abstract data, their ability to enhance one's understanding of complex relationships can be hindered by a limited set of predefined charts. To enable novel visualization over multiple variables, we propose a declarative language for specifying informational graphics from first principles. The language maps properties of generic objects to graphical representations based on scaled interpretations of data values. An iterative approach to constraint solving that involves user advice enables the optimization of graphic layouts. The flexibility and expressiveness of a powerful but relatively easy to use grammar supports the expression of visualizations ranging from the simple to the complex.

1 Introduction

Information visualization tools support creativity by enabling discoveries about data that would otherwise be difficult to perceive [8]. Oftentimes, however, the standard set of visualizations offered by commercial charting packages and business intelligence tools is not sufficient for exploring and representing complex relationships between multiple variables. Even specialized visual analysis tools may not offer displays that are relevant to the user's particular needs. As noted in [9], the creation of effective visual representations is a labor-intensive process, and new methods are needed for simplifying this process and creating applications that are better targeted to the data and tasks. To that end, we introduce a language for specifying informational graphics from first principles. One can view the goal of the present research as doing for information visualization what spreadsheet software did for business applications. Prior to Bricklin's VisiCalc, business applications were built separately from scratch. By identifying the first principles on which many of these applications were built — namely, arithmetic calculations over geographically defined values — the spreadsheet program made it possible for end users to generate their own novel business applications. This flexibility was obtained at the cost of requiring a more sophisticated user, but the additional layer of complexity can be hidden from naive users through prepackaged spreadsheets.

Similarly, we propose to allow direct access to the first principles on which (a subset of) informational graphics are built through an appropriate specification language. The advantages are again flexibility and expressiveness, with the same cost in terms of user sophistication and mitigation of this cost through prepackaging.

For expository reasons, the functionality we provide with this language is presented by way of example, from simple scatter plots to versions of two quite famous visualizations: Minard's depiction of troop strength during Napoleon's march on Moscow and a map of the early ARPAnet from the ancient history of the Internet.[3] We hope that the reader can easily extrapolate from the provided examples to see the power of the language. We then describe how constraints are specified and how the constraint satisfaction process is coupled with user advice to reduce local minima. This is followed by descriptions of the primary constructs in the language and the output generation process for our implementation, which was used to generate the graphics shown in this paper.

2 The Structure of Informational Graphics

In order to define a language for specifying informational graphics from first principles, those principles must be identified. For the subset of informational graphics that we consider here, the underlying principles are relatively simple:

- Graphics are constructed based on the rendering of generic graphical objects taken from a small fixed set (points, lines, polygons, text, etc.).
- The graphical properties of these generic graphical objects are instantiated by being tied to values taken from the underlying data (perhaps by way of computation).
- The relationship between a data value and a graphical value is mediated by a function called a *scale*.
- Scales can be depicted via generic graphical objects referred to as *legends*. (A special case is an *axis*, which is the legend for a location scale.)
- The tying of values is done by simple constraints, typically of equality, but occasionally of other types.

For example, consider a generic graphical object, the *point*, which has graphical properties like horizontal and vertical position, color, size, and shape. The standard scatter plot is a graphic where a single generic object, a point, is instantiated in the following way. Its horizontal and vertical position are directly tied by an equality constraint to values from particular data fields of the underlying table. The other graphical properties may be given fixed (default) values or tied to other data fields. In addition, it is typical to render the scales that govern the mapping from data values to graphical locations using axes.

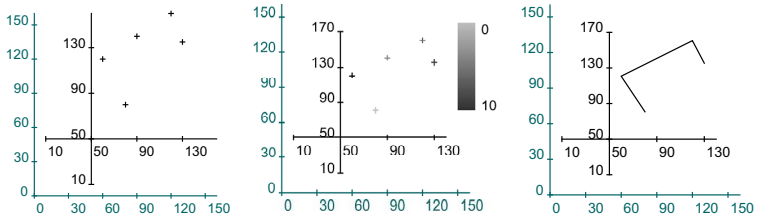
Suppose we have a table `Table` with fields `f`, `g`, and `h` as in Table 1. We can specify a scatter plot of the first two fields in just the way that was informally described above:

[3] See Figures 4 and 7 for our versions of these visualizations.

```
{make p:point with
   p.center = Canvas(record.f, record.g)
| record in SQL("select f, g from Table1")};
make a:axis with
  a.aorigin = (50,50),
  a.ll = (10,10),
  a.ur = (160,160),
  a.tick = (40,40);
```

f	g	h
80	80	0
60	120	10
90	140	3
120	60	5
140	135	7

Table 1



(a) (b) (c)

Fig. 1. Data table and simple scatter plots defined from first principles.

The `make` keyword instantiates a generic graphical object (`point`) and sets its attributes. The set comprehension construct ($\{\cdot|\cdot\}$) constructs a set with elements specified in its first part generated with values specified in its second part. Finally, we avail ourselves of a built-in scale, `Canvas`, which maps numeric values onto the final rendering canvas. One can think of this as the assignment of numbers to actual pixel values on the canvas. A depiction of the resulting graphic is given in Figure 1(a). (For reference, we show in light gray an extra axis depicting the canvas itself.)

Other graphical properties of chart objects can be tied to other fields. In Figure 1(b), we use a `colorscale`, a primitive for building linearly interpolated color scales. A legend for the color scale is positioned on the canvas as well. Some scaling of the data can also be useful. We define a 2-D Cartesian frame to provide this scaling, using it instead of the canvas for placing the points.

```
let frame:twodcart with
  frame.map(a,b) = Canvas(a, (3*b)/4+ 10)
in
  let color:colorscale with
    color.min = ColorMap("red"),
    color.max = ColorMap("black"),
    color.minval = 0,
    color.maxval = 10
  in
   {make p:point with
      p.center = frame.map(rec.f, rec.g),
      p.color = color.scale(rec.h)
   |rec in SQL("select f, g, h from Table1")},
   make a:axis with
```

```
    a.scale = frame.map,
    a.aorigin = (50,50),
    a.ll = (0,0),
    a.ur = (140,170),
    a.tick = (50,50),
  make c:legend with
    c.scale = color,
    c.location = frame.map(150, 180);
```

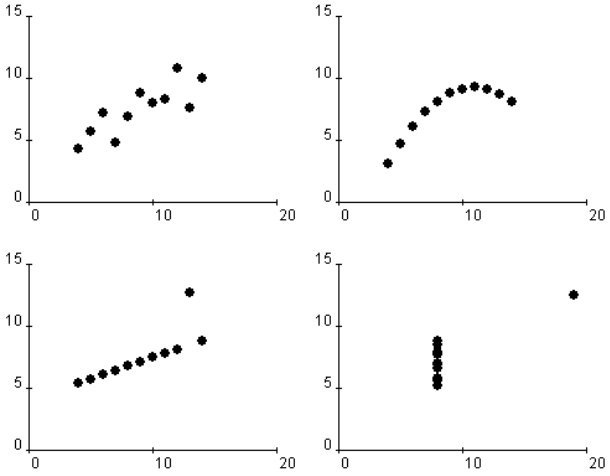
A line chart can be generated using a line object instead of a point object. Suppose we take the records in Table 1 to be ordered, so that the lines should connect the points from the table in that order. Then a simple self-join provides the start points (x_1, y_1) and end points (x_2, y_2) for the lines. By specifying the appropriate query in SQL, we can build a line plot (see Figure 1(c)).

```
let frame:twodcart with
  frame.map(a,b) = Canvas(a,(3*b)/4+ 10)
in
  {make l:line with
     l.start = frame.map(record.x1, record.y1),
     l.end = frame.map(record.x2, record.y2)
   | record in SQL("select tab1.f as x1, tab1.g as y1,
                    tab2.f as x2, tab2.g as y2
                   from Table1 as tab1, Table1 as tab2
                   where tab2.recno = tab1.recno+1")},
   make a:axis with
        a.scale = frame.map,
        a.aorigin = (50,50),
        a.ll = (10, 10),
        a.ur = (140,170),
        a.tick = (40,40);
```

The specification language also allows definitions of more abstract notions such as complex objects or groupings. We can use this facility to define a chart as an object that can be instantiated with specified data. This allows generated visualizations themselves, such as scatter plots, to be manipulated as graphical objects. For instance, it is possible to form a scatter plot of scatter plots. Figure 2 depicts a visualization of the data sets for Anscombe's quartet [1], generated by the following specification:

```
define s:splot with
    let frame:twodcart with
          frame.map = s.map
    in
        { make o:oval with
           o.center ~ frame.map(rec.x, rec.y),
           o.width = 8,
           o.height = 8,
           o.fill = true
           | rec in s.recs },
         make a:axis with
              a.scale = frame.map,
              a.aorigin = (0,0),
```

```
            a.ll = (0,0),
            a.ur = (20,15),
            a.tick = (10,5)
in
   let outer:twodcart with
      outer.map(x,y) = Canvas(10*x, 10*y)
   in
      let FrameRecs = SQL("select distinct a, b from anscombe")
      in
        {make sp:splot with
            sp.map(x,y) = outer.map(x + 25*frec.a, y + 20*frec.b),
            sp.recs = SQL("select x, y from anscombe where a=frec.a
                           and b=frec.b")
         | frec in FrameRecs};
```



a=0 x	b=1 y	a=1 x	b=1 y	a=0 x	b=0 y	a=1 x	b=0 y
10	8.04	10	9.14	10	7.46	8	6.58
8	6.95	8	8.14	8	6.77	8	5.76
13	7.58	13	8.74	13	12.74	8	7.71
9	8.81	9	8.77	9	7.11	8	8.84
11	8.33	11	9.26	11	7.81	8	8.47
14	9.96	14	8.1	14	8.84	8	7.04
6	7.24	6	6.13	6	6.08	8	5.25
4	4.26	4	3.1	4	5.39	19	12.5
12	10.84	12	9.13	12	8.15	8	5.56
7	4.82	7	7.26	7	6.42	8	7.91
5	5.68	5	4.74	5	5.73	8	6.89

Fig. 2. Graphic depicting Anscombe's quartet.

3 Specifying Constraints

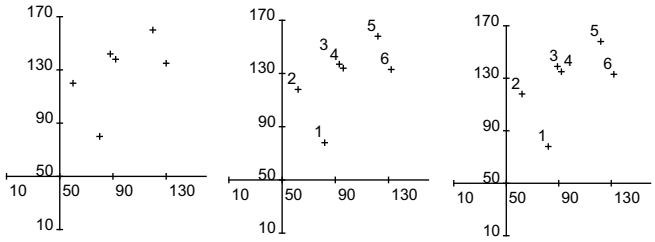
All of the examples in the prior section made use of constraints of type equality only. The ability to add other types of constraints dramatically increases the flexibility of the language. For instance, stating that two values are approximately equal (`~`), instead of strictly equal (`=`), allows for approximate satisfaction of equality constraints. Further, constraints of non-overlapping (`NO`) force values apart. Together, these constraints allow dither to be added to graphs.

Suppose we have another table `Table` with fields `id`, `f`, and `g`, as in Table 2. We again specify a scatter plot of fields `f` and `g` but with two differences from our earlier examples: the center of each `point` is approximately equal to a data value from the table, and none of the `point` objects can overlap. The resulting graphic is shown in Figure 3(a).

```
NO({make p:point with
         p.center ~ Canvas(record.f, record.g)
    | record in SQL("select f, g from Table1")});
make a:axis with
     a.aorigin = (50,50),
     a.ll = (10,10),
     a.ur = (160,170),
     a.tick = (40,40),
     a.color = RGB(0, 0, 0);
```

id	f	g
1	80	80
2	60	120
3	90	140
4	90	140
5	120	60
6	140	135

Table 2



(a) (b) (c)

Fig. 3. Data table and scatter plots with positional constraints.

We would like to add labels at each data point and specify three conditions: (1) no `point` can overlap with another point, (2) no `label` can overlap with another label, and (3) no `point` can overlap with a `label`. We need the ability to associate a variable name with an object or a set of objects for later reference. This is accomplished with the `let` statement, which provides access to that variable within the body of the statement. The following code contains the required specifications, with its output shown in Figure 3(b).

```
let points = NO({make p:point with
   p.center ~ Canvas(rec.f, rec.g)
  | rec in SQL("select f, g from Table2")})
  in
    let labels = NO({make l:label with
       l.center ~ Canvas(rec.f, rec.g),
       l.label = rec.id
      | rec in SQL("select f, g, id from Table2")})
      in
        NO(points, labels);
make a:axis with
     a.aorigin = (50,50),
     a.ll = (10,10),
     a.ur = (160,170),
     a.tick = (40,40),
     a.color = RGB(0, 0, 0);
```

The non-overlap constraints between `point` objects, between `label` objects, and between `point` and `label` objects have moved the label on point 3 farther away from the actual data point than may be desirable. This is a result of the

force of a non-overlap constraint being stronger than that of an approximate equality constraint. The user can make adjustments to the layout by dragging any object whose location has been specified as approximately equal to a value, but these adjustment are subject to any constraints placed upon that object. Thus, the user may drag the label `3` to a different location, but the '`~`' constraint will keep it near `(90, 140)`, while the `'NO'` constraint will prohibit it from overlapping with any other point or label. The results of user manipulation to move label `3` to a more desirable location are shown in Figure 3(c).

4 A Detailed Example: The Minard Graphic

As evidence of the flexibility of this language, we describe its use for specifying Minard's well known depiction of troop strength during Napoleon's march on Moscow. This graphic uses approximate geography to show the route of troop movements, with line segments for the legs of the journey. Width of the lines is used for troop strength and color depicts direction. The locations of the labels on cities are of approximate equality, as they are not allowed to overlap, and in some cases have been adjusted by the user for better clarity. A parallel graphic shows temperature during the inbound portion of the route, again as a line chart. Our version of the graph is provided in Figure 4.

To generate this graphic, we require appropriate data tables: `marchNap` includes latitude and longitude at each way point, along with location name, direction, and troop strength; `marchTemp` includes latitude, longitude, and temperature for a subset of the inbound journey points, and `marchCity` provides latitude, longitude, and name for the traversed cities.

The main portion of the graphic is essentially a set of line plots, one for each branch of the march, where a branch is defined as a route taken in a single direction by a single division. Additional graphical properties (width and color) are tied to appropriate data fields. Textual labels for the cities are added using a text graphic object. A longitudinally aligned graph presents temperature on the main return branch.

The specification for this graphic (sans the temperature portion) is provided in Figure 5. After specifying some constants (lines 1–2), we define the depiction of a single branch of the march (6–16): a mapping (`m.map`) specifies a Cartesian frame (7–8) in which a line plot composed of a set of line segments (10–16) is placed, with one segment for each set of records (16). These records provide start and end points, along with widths at start and end (to be interpolated between), and color (11–15).

Thus, to depict a march leg, all that must be provided is the mapping and the set of records. These are constructed in lines 19–35. For each distinct direction and division (19–20), a separate march leg depiction is constructed (23–35). The mapping is a scaling of the `Canvas` frame (24), with records for the appropriate division and direction extracted from the `marchNap` database (25–35). Finally, the cities are labeled using the information in the `marchCity` database by setting the coordinates of the text labels to be approximately equal to latitude and

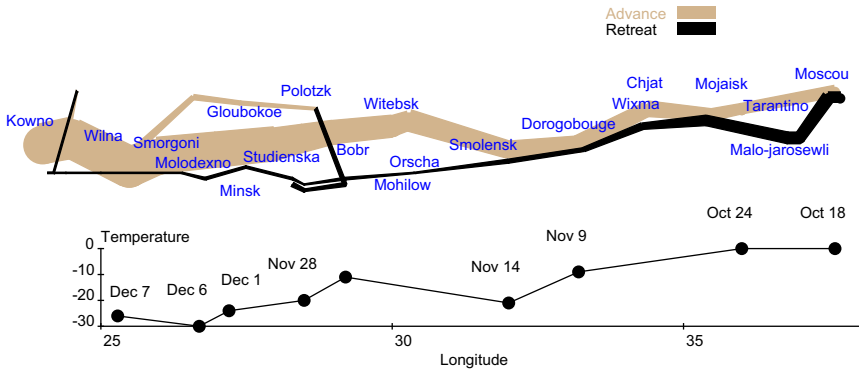


Fig. 4. The graphic generated by a specification extended from that of Figure 5, depicting the troop strength during Napoleon's march on Moscow.

longitude, setting a fixed color, and specifying that labels cannot overlap (38–42).

5 The Language

The examples from this paper were all generated from an implementation of our language. The implementation techniques are outlined in Section 6. The underlying ideas could, however, be implemented in other ways, for instance as a library of functions in a suitable functional language such as Haskell or ML.

The language is built around the specification of objects. The main construct is *objspec*, with a program defined as one or more objspecs. *Objspec*s are used for instantiating a graphical object of a predefined type, specifying relationships between the set of actual data values and their graphical representations, and defining new graphic types.

The `make` statement is used for instantiating a new instance of an existing object type (either built-in, such as `point` or `line`, or user-defined, such as `march`) with one or more conditions. There are two types of predefined objects: *generic* and *scale*. Generic objects provide visual representations of data values and include the primitive types of point, line, oval, rectangle, polygon, polar segment, and labels. Scales are used for mapping data values to their visual representations and are graphically represented by axes and legends. A scale is associated with a coordinate system object and defines a transformation from the default layout canvas to a frame of type `twodcart` or `twodpolar` (at this time).

In addition to a unique identifier, each object type has predefined attributes, with conditions expressed as constraints on these attributes. Constraints can be of type equality ('=') or approximate equality ('~'). Constraints enforcing visual organization features [3] such as non-overlap, alignment, or symmetry, can be applied to a set of graphical objects. These types of constraints are particularly useful in specifying network diagrams [7]. While such constraints can be specified

```
let scalefactor = 45 in
let weight = .000002 in

% Define the depiction of one (multi-leg) branch of the march
% identified by a distinct direction and division
define m:march with
 let frame:twodcart with
     frame.map = m.map
 in
  {make l:line with
        l.start = frame.map(rec.cx1, rec.cy1),
        l.end = frame.map(rec.cx2, rec.cy2),
        l.startWidth = scalefactor * weight * rec.r1,
        l.endWidth = scalefactor * weight * rec.r2,
        l.color = ColorMap(rec.color)
     | rec in m.recs}
in
 % Extract the set of branches that make up the full march
 let FrameRecs = SQL("select distinct direct,
                                     division from marchNap")
 in
  % For each branch, depict it
  {make mp:march with
    mp.map(x,y) = Canvas(scalefactor*x - 1075, scalefactor*y - 2200),
    mp.recs = SQL("select marchNap1.lonp as cx1, marchNap1.latp as cy1,
                     marchNap2.lonp as cx2, marchNap2.latp as cy2,
                     marchNap1.surviv as r1, marchNap2.surviv as r2,
                     marchNap1.color as color
                   from marchNap as marchNap1, marchNap as marchNap2
                   where marchNap1.direct = framerec.direct
                     and marchNap2.direct = marchNap1.direct
                     and marchNap1.division = framerec.division
                     and marchNap1.division = marchNap2.division
                     and marchNap2.recno = marchNap1.recno+1")
     | framerec in FrameRecs};

% Label the cities along the march and do not allow overlap
NO({make e d2:label with
  d2.center ~ frameTemp.map(recc.lonc, recc.latc),
  d2.label = recc.city,
  d2.color = ColorMap("blue")
| recc in SQL("select lonc, latc, city from marchCity")});
```

Fig. 5. Partial specification of the Minard graphic depicted in Figure 4. This specification does not include the temperature plot.

in the language, our current implementation supports constraints in the forms of equality, approximate equality, and non-overlap for a subset of object types.

All of the generic objects have a `color` attribute and at least one `location` attribute, represented by a coordinate. For 2-D objects, a Boolean `fill` attribute defaults to `true`, indicating that the interior of that object be filled in with the specified color. This attribute also applies to line objects, which have start and end `widths`. When widths are specified, lines are rendered as four-sided polygons with rounded ends.

Each type of scale object has its own set of attributes. A coordinate system object's `parent` attribute can reference another coordinate system object or the canvas itself. The mapping from data values to graphical values can be specified by conditions on its `origin` and `unit` attributes. Alternatively, a condition can be applied to its `map` attribute in the form of a user-defined mapping function that denotes both the parent object and the scale. Thus, a `twodcart` object whose `parent` is Canvas, `origin` is (5, 10), and `unit` is (30, 40) can be defined by the mapping function: `Canvas.map(30x + 5, 40y + 10)`.

Axes are defined in terms of their `origin`, `ll` (lower left) and `ur` (upper right) coordinates, and `tick` marks. Legends can be used to graphically associate a color gradient with a range of data values by assigning a color scale object to the `scale` attribute and a coordinate to the `location` attribute. Discrete colors can also be associated with data values, as in the Minard Graph example, where tan represents "Advance" and black represents "Retreat."

Attributes can also be defined by the user within a `make` statement. For example, it is often helpful to declare temporary variables for storing data retrieved from a database. These user-defined attributes are ignored when the specified visualization is rendered.

Another construct for an *objspec* is the `type` statement. This is used for defining a new object *type* that satisfies a set of conditions. These conditions are either constraints on attribute properties or other objspecs. An example of the former would be a condition requiring that the color attribute for a new chart type be "blue." An example of the latter is to require that a chart include a 2-D Cartesian coordinate frame for use in rendering its display and that this frame contain a set of lines and points corresponding to data retrieved from a database.

A third type of *objspec* is a set of objects, a *setspec*. This takes the form of either a query string or a set of objspecs to be instantiated for each record retrieved by a query string. These two constructs are often used in conjunction with one another to associate data with generic graph objects. For example, a setspec may define a query that retrieves two values, `x` and `y`, from a database table. A second setspec can then specify a `make` command for rendering a set of points located at the `x` and `y` values retrieved by the query. Alternatively, the query can be contained within the `make` statement itself, as shown in the code for the scatter plots in Section 2.

Lastly, the `let` statement associates a variable name with an *objspec* so that it can be referenced later within the body of the statement. Because this

construct is commonly used in conjunction with a `make` or `type` statement, two shorthand expressions have been provided. The first of these is an abbreviated form of the `let` statement in which the `make` clause is not explicitly stated, with: `let` *var*`:`*type* `with` *conditions* `in` *body* corresponding to `let` *var*`=make` *var*`:`*type* `with` *conditions* `in` *body*. Similarly, the `define` statement associates a new object definition with a variable name, where: `define` *var*`:`*type* `with` *conditions* `in` *body* expands to `let` *var*`=type` *var*`:`*type* `with` *conditions* `in` *body*.

Figure 6 demonstrates the usage of the above *objspec* constructs for defining a new type of object called a `netplot`. The code specifies the creation of a set of ovals, referred to as nodes, with width and height of 10, color of "black," and center locations corresponding to `x` and `y` values queried from a database and mapped to a frame. (Data values were estimated from a depiction of the ARPAnet circa 1971 available at `http://www.cybergeography.org/atlas/historical.html`.) A set of blue lines are specified, with the endpoint locations of the lines equal to the centers of the circles. Labels are also specified and are subject to constraints, with the center of each label approximately equal to the center of a corresponding node and no overlap allowed between labels or between labels and nodes.

The output generated from this specification is shown in Figure 7, with the user having made similar adjustments as the one shown in Figure 3(c) to the placement of some labels. This chart and the Minard graphic from the prior section demonstrate the flexibility and control for specifying visualizations that result from working with the right set of first principles. Further, the intricacies of the code required to generate these charts are hidden behind a relatively simple to use but powerful grammar.

6 Implementation

The output generation process that renders visualizations from code written in our language, such as those shown in the prior sections, involves four stages.

1. The code is parsed into an abstract syntax tree representation.
2. The tree is traversed to set variable bindings and to evaluate *objspec*s to the objects and sets of objects that they specify. These objects are collected, and the constraints that are imposed on them are accumulated as well.
3. The constraints are solved as in related work on the Glide system [7] by reduction to mass-spring systems and iterative relaxation. Equality constraints between objects are strictly enforced, in that neither the user nor the system can change the position of those objects. Non-overlap constraints between objects are enforced by the placement of a spring between the centers of the objects (at the current time, the system does not solve for non-overlap constraints involving lines). The spring's rest length is the required minimum distance between those objects. Approximate equality, or near constraints, are enforced by the placement of springs with rest lengths of zero between specified points on two objects. The latter set of springs have a smaller spring

```
define n:netplot with
  let frame:twodcart with
        frame.map = n.map
  in
    let nodes = {make o:oval with
         o.center = frame.map(rec.x, rec.y),
         o.width = n.radius * 2,
         o.height = n.radius * 2,
         o.color = n.ccolor,
         o.fill = true
       | rec in n.netRecs}
    in
      let lines =  {make l:line with
           l.start = nodes[rec.v1].center,
           l.end = nodes[rec.v2].center,
           l.color = n.lcolor
         | rec in n.edgeRecs }
      in
        let labels =  NO({ make d:label with
     d.center ~ frame.map(rec.x, rec.y),
     d.label = rec.node,
     d.color = n.ccolor
          | rec in n.netRecs })
        in NO(labels, nodes)
in
  make net:netplot with
      net.map(x,y) = Canvas(7*x, 7*y),
      net.netRecs = SQL("select node, x, y from ArpaNodes"),
      net.radius = 5,
      net.edgeRecs = SQL("select v1, v2 from ArpaLinks"),
      net.lcolor = ColorMap("blue"),
      net.ccolor = ColorMap("black");
```

Fig. 6. Specification of the ARPAnet graphic depicted in Figure 7.

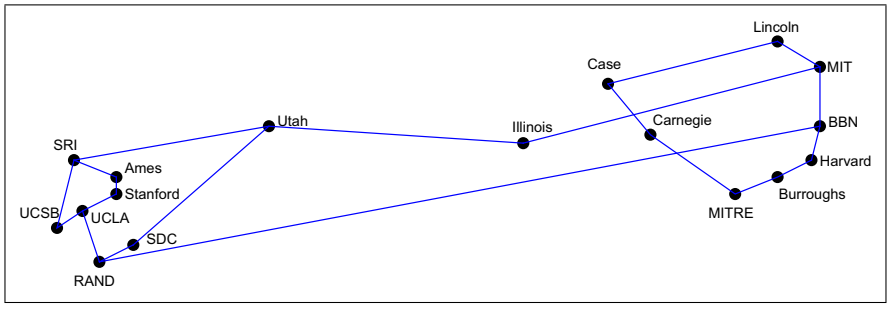


Fig. 7. A network diagram depicting the ARPAnet circa 1971.

constant than those used for non-overlap, thereby exerting less force. Solving one constraint may invalidate another, so an iterative process is followed until the total kinetic energy on all nodes falls below an arbitrarily set value.

4. The objects are rendered. Any primitive graphic object contained in the collection is drawn to the canvas using the graphical values determined as a result of the constraint satisfaction process. The user can make positional adjustments to any objects not subject to equality constraints. At the same time, the system continues to reevaluate its solution and update the position of the objects based on the solving of the constraints.

7 Related Work

Standard charting software packages, such as Microsoft Chart or DeltaGraph, enable the generation of predefined graphic layouts selected from a "gallery" of options. As argued above, by providing pre-specified graph types, they provide simplicity at the cost of the expressivity that motivates our work. More flexibility can be achieved by embedding such gallery-based systems inside of programming languages to enable program-specified data manipulations and complex graphical composites. Many systems have this capability: Mathematica, Matlab, Igor, and so forth. Another method for expanding expressiveness is to embed the graph generation inside of a full object-drawing package. Then, arbitrary additions and modifications can be made to the generated graphics by manual direct manipulation. Neither of these methods extend expressivity by deconstructing the graphics into their abstract information-bearing parts, that is, by allowing specification from first principles.

Towards this latter goal, discovery of the first principles of informational graphics was pioneered by Bertin, whose work on the semiology of graphics [2] provides a deep analysis of the principles. It is not, however, formalized in such a way that computer implementation is possible.

Computer systems for automated generation of graphics necessarily require those graphics to be built from a set of components whose function can be formally reasoned about. The seminal work in this area was by Mackinlay [4], whose system could generate a variety of standard informational graphics. The range of generable graphics was extended by Roth and Mattis [6] in his SAGE system. These systems were designed for automated generation of appropriate informational graphics from raw data, rather than for user-specified visualization of the data. The emphasis is thus on the functional appropriateness of the generated graphic rather than the expressiveness of the range of generable graphics.

The SAGE system serves as the basis for a user-manipulable set of tools for generating informational graphics, SageTools [5]. This system shares with the present one the tying of graphical properties of objects to data values. Unlike SageTools, the present system relies solely on this idea, which is made possible by the embedding of this primitive principle in a specification language and the broadening of the set of object types to which the principle can be applied.

The effort most similar to the one described here is Wilkinson's work on a specification language for informational graphics from first principles, a "grammar" of graphics [10]. Wilkinson's system differs from the one proposed here in three ways. First, the level of the language primitives are considerably higher; notions such as Voronoi tesselation or vane glyphs serve as primitives in the system. Second, the goal of his language is to explicate the semantics of graphics, not to serve as a command language for generating the graphics. Thus, many of the details of rendering can be glossed over in the system. Lastly, and most importantly, the ability to embed constraints beyond those of equality provides us the capacity to generate a range of informational graphics that use positional information in a much looser and more nuanced way.

8 Conclusions

We have presented a specification language for describing informational graphics from first principles, founded on the simple idea of instantiating the graphical properties of generic graphical objects from constraints over the scaled interpretation of data values. This idea serves as a foundation for a wide variety of graphics, well beyond the typical sorts found in systems based on fixed galleries of charts or graphs. By making graphical first principles available to users, our approach provides flexibility and expressiveness for specifying innovative visualizations.

References

1. Anscombe, F. J.: Graphs in statistical analysis. *American Statistician*, 27(February 1973):17–21 (1973)
2. Bertin, J.: *Semiology of Graphics*. University of Wisconsin Press (1983)
3. Kosak, C., Marks, J., and Shieber, S.: Automating the layout of network diagrams with specified visual organization. *Transactions on Systems, Man and Cybernetics*, 24(3):440–454 (1994)
4. Mackinlay, J.: Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141 (1986)
5. Roth, S. F., Kolojejchick, J., Mattis, J., and Chuah, M. C.: Sagetools: An intelligent environment for sketching, browsing, and customizing data-graphics. In *CHI '95: Conference companion on Human factors in computing systems*, pages 409–410, New York, NY, USA. ACM Press (1995)
6. Roth, S. F. and Mattis, J.: Data characterization for graphic presentation. *In Proceedings of the Computer-Human Interaction Conference* (CHI '90)(1990)
7. Ryall, K., Marks, J., and Shieber, S. M.: An interactive constraint-based system for drawing graphs. In *Proceedings of the 10th Annual Symposium on User Interface Software and Technology* (UIST)(1997)
8. Shneiderman, B.: Creativity support tools: accelerating discovery and innovation. *Communications of the ACM*, 50(12):20–32(2007)
9. Thomas, J. J. and Cook, K. A.: A visual analytics agenda. *IEEE Computer Graphics and Applications*, 26(1):10–13 (2006)
10. Wilkinson, L.: *The Grammar of Graphics*. Springer-Verlag, New York, NY (1999)