Encoding Presentation Emphasis Algorithms for Graphs

A Poster presented at the DIMACS 1994 Workshop on Graph Drawing

Princeton, New Jersey, October 10–13, 1994

Emanuel G. Noik

CSRI,
University of Toronto,
6 King's College Rd.,
Toronto, ON, Canada M4S 1A1

Tel: (416) 978-8609
email: noik@db.toronto.edu

Abstract

While graphs can effectively visualize one or more relations on a set of elements, drawings of large graphs can be difficult to understand. As such, many presentation emphasis techniques for visualizing graphs such as fisheye views have been proposed. A recent survey paper [8] described an abstract space of techniques and identified common shortcomings. In the companion paper presented at the Workshop we outlined a high-level language, Dye, that addresses several of these limitations. In particular, Dye is used to: 1) select subsets of graph elements; 2) compute a real-valued priority for each element; and, 3) encode presentation strategies that automatically emphasize elements based on subset membership and priority.

1 Dye: A Presentation Emphasis Language

Dye[1] is a special purpose specification language that is based on relational algebra with arithmetic and aggregation [9]. The current version of the language supports the standard relational algebra union and difference operators, and the *closed join* operator, defined as follows:

$$\bowtie(\{R_1, \ldots, R_n\}, F) \quad = \quad \pi_{R_1} \sigma_F (R_1 \times \ldots \times R_n)$$

Notice that for $n = 1$, the closed join reduces to the relational algebra selection operator:

$$\begin{aligned} \bowtie(\{R_1\}, F) &= \pi_{R_1} \sigma_F (R_1) \\ &= \sigma_F (R_1) \end{aligned}$$

A Dye program captures four types of information: subset selections; focal point response functions; priority functions; and, actions that use set membership and DOI values to assign values to various graphical attributes.

In addition, Dye is extensible – algorithms can be added to evaluate queries that cannot be expressed in relational algebra (*e.g.,*transitive closure). The current version of Dye, for example, includes support for computing (the lengths of and elements along) shortest paths [2].

The general syntax of a Dye program is:

```
<program> ::=
  ["sets" [<set_assignment>]+ ]
  ["fpresponse" [<response_function>]+ ]
  ["doi" [<doi_assignment>]+ ]
  ["actions" [<action_block>]+ ]
```

2 An Express Tour

Example 1. Figure 1 shows a graph drawing of a passenger flights database generated by an anchored force-based layout algorithm. The vertices of the graph represent cities; an (undirected) arc between a pair of vertices implies that some airline operates flights between the corresponding

[1] Graphite [1, 6] can be thought of as the raw material (carbon) for producing drawings of graphs; Dye adds *colour* (visual emphasis through the systematic manipulation of various graphical attributes including colour) to Graphite.

cities. Thus in addition to the purely topological information, arcs are labelled (by the name of the airline – arc labels are suppressed by default to reduce visual clutter) and have two additional attributes, namely the price of a one-way ticket (in dollars), and the duration of a one-way trip (in minutes). Similarly, each vertex is labelled by the name of the corresponding city.

Example 2. One of the simplest presentation requirements is to be able to assign colours to the nodes and links in a diagram. It is common to assign link colours based on the associated arc label. Figure 2 shows a diagram in which link colours have been assigned based on the airline name (arc label). As in all subsequent examples, the program that was used to accomplish this is listed below the diagram. To assign colours to links, a single action block iterates over every arc `a` in the set of arcs `A`. Note how the C language macro preprocessor [4] directives permit the input program to be split across multiple files and provide a simple macro facility.

Example 3. Another common common requirement is the ability to control line style and thickness. Figure 3 shows the diagram and emphasis program that sets the width of links based on the ratio of flight price to flying time (the two arc attributes). This has the effect of emphasizing more expensive flights (relative to distance flown); note that it would be just as easy to emphasize the less expensive flights and de-emphasize the expensive ones. The suffix `%n` refers to external numeric attribute n (external string attributes have the form `$n`). Also note that `a.doi` refers to the arc DOI which in this case is simply the arc API (the price to time ratio). Since priority values are normalized, `a.doi` will be in the range $[0, 1]$; `a.line.width` therefore, will be in the range of $[1, 6]$ units.

Example 4. The size (area) of a node is also often used to depict some aspect of the corresponding vertex. Figure 4 shows the diagram and emphasis program that sets the size of each node in proportion to its vertex connectivity. In this domain, this corresponds to emphasizing cities that have busy airports (many flights). The *sum* aggregate operator can be used to compute the sum of some expression (given by its third argument) over a selected subset of elements (set given by its first argument, selection criteria by its second argument). In this example `true` selects all arcs in the union of in-coming and out-going arcs and calculates the sum of the expression `1` over these, which gives the number of incident arcs. The terms `v.s.x` and `v.s.y` are the x and y components of the vertex scaling coefficient `v.s`; during graph layout, the size of each node is scaled by its scaling coefficient.

Example 5. In this variation on the previous example, the *avg* aggregate operator is used to find the average price to time ratio of flights through each airport, and thus to emphasize cities that host *expensive* airports. Figure 5 shows the diagram and emphasis program to accomplish this.

Example 6. Fisheye views have been receiving increasing attention as a means to simultaneously display information at multiple levels of abstraction (as a way to display large highly detailed visualizations). Figure 6 shows a diagram of a simple fisheye view and the corresponding emphasis program. In this case, the vertex API is the same as in the previous example – the average price to time ratio of incident arcs, while the distance between to vertices is simply the two-dimensional Euclidean distance between the corresponding nodes in the original diagram. The focal point response function causes the magnification effect of a focal point to decrease as the cube of the distance to a given node. Note that the terms `v.p.x` and `v.p.y` refer to the x and y coordinates of the vertex position, while `f.p.x` and `f.p.y` refer to the focal point's position. Recall that given a set of focal points $F \subseteq V(G)$, by default $FPI(v, F)$ is the normalized sum of the response of focal points F on vertex v. Finally, note that focal points and their magnifications and response functions are provided to the Dye run-time component during execution (as command line arguments in batch mode or by a front end application in interactive mode).

Example 7. In the flights database as in most domains, paths in graphs usually have a meaningful and significant interpretation. In particular, paths can provide a fundamental measure of distance between a pair vertices in a graph. Since well-known algorithms exist for computing the shortest paths in a graph [2], several primitives were provided as extensions to Dye; these are used to compute the lengths of shortest paths and to collect the arcs (or blobs,[2] if computing paths in the nesting hierarchy) along these paths (vertices can be recovered from the arcs or blobs). Figure 7 shows a diagram and emphasis program which is a small variation on the previous example, in which the distance between vertices u, v is the number of arcs along the shortest uv path. The *minpath* primitive in this example takes four arguments: a set of arcs, a traversal direction (forward, reverse, or undirected – may traverse arc in either direction), an expression to evaluate for each vertex along the path, and an expression to evaluate for each arc along the path.

Example 8. This is a variation on the previous example in which the distance corresponds to the total trip time between two vertices assuming that each vertex (airport stop-over) on the path contributes an additional 30 minutes to the trip. Figure 8 shows the diagram and emphasis program.

Example 9. This is a variation on the previous example in which focal points are assigned an alternative response function – one that tends to emphasize distant elements and deemphasize proximate ones, essentially treating the focal

[2] Given a graph $\langle V, A, B\rangle$, $B \subseteq V \times 2^V$ is a set of blobs [1]; that is, a blob associates a (container) vertex with a set of (contained) vertices.

points as peripheral points. This use of focal point response functions enables us to selectively deemphasize elements. Figure 9 shows the diagram and emphasis program.

Example 10. As promised above, a primitive was also included for collecting the set of arcs along shortest paths. Figure 10 shows the diagram and emphasis program which emphasizes the minimum cost route between chilly Reykjavik and warm Honolulu. Note that variables that stand for vertex sets have the syntax `v_SetName`, arc sets `a_SetName`, and blobs `b_SetName`. In this example, the *minpath* primitive takes six arguments: four as in the previous case, and an additional two that specify a set of sources and a set of sinks; the resulting set of arcs consists of those arcs that lie on all shortest source-sink paths for every source and sink pair for which at least one source-sink path exists. If, as in this example, there is exactly one source and sink, this single shortest source-sink path is selected. The closed join operator `cjoin` takes two arguments: a list of sets (relations) and a selection formula; the notation `Xn.Attr` in a formula refers to attribute *Attr* of the nth set in the list; *X* is one of `V`, `A`, or `B` depending on whether the nth set is a set of vertices, arcs, or blobs. Thus the expressions `V0.id`, `A1.from.id`, and `A1.to.id` refer to the vertex identifier, and the identifiers of the *from* and *to* vertices of an arc. In this example, therefore, the *cjoin* operator is used to select the vertices that lie on the shortest path.

Example 11. As a variation on the previous example and as a service to frigid Icelanders who seek the quickest route to Honolulu at any cost, Figure 11 shows the diagram and emphasis program which emphasizes the minimum time route between Reykjavik and Honolulu.

Example 12. This and the following example illustrate some simple emphasis techniques for nested graphs. A nested version of the airline flights database is used – it contains blobs that organize the cities into countries and the countries into the Western and Eastern hemispheres. Figure 12 shows the diagram and emphasis program that set blob region colours and layout algorithms based on the blob label. The latter capability provides a simple composite layout strategy [3]; the notation `b.layout := string` creates a custom Graphite [6] layout object and assigns it to all blobs on the left hand side of the assignment.

Example 13. This extensions of the previous example shows a simple fisheye view of a nested graph that is similar to the layout independent fisheye view [5, 7]. Vertex API is set inversely to its nesting depth and distance is set to the length of the shortest path to a focal point in the nesting hierarchy. Figure 13 shows the diagram and emphasis program to accomplish this. The notation `V0.parB` refers to the set of blobs that immediately contain the vertex in question; however, since we restrict the nesting relation to be hierarchical, the cardinality of this set can be at most 1.

References

[1] M.P. Consens, F. Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. Architecture and applications of the Hy+ visualization system. *IBM Systems Journal*, 33(3):458–476, Aug. 1994.

[2] R.W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[3] T.R. Henry and S.E. Hudson. Interactive graph layout. In *ACM UIST '91*, pages 55–64. ACM, 1991.

[4] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[5] E.G. Noik. Exploring large hyperdocuments: Fisheye views of nested networks. In *ACM Hypertext '93*, pages 192–205, Seattle, WA, Nov. 1993.

[6] E.G. Noik. Graphite: A suite of hygraph visualization utilities. In A. O. Mendelzon, editor, *Declarative database visualization: recent papers from the Hy+/GraphLog project*, pages 108–126. Tech. rep. CSRI-285, U. of Toronto, Jul. 1993.

[7] E.G. Noik. Layout-independent fisheye views of nested graphs. In *VL '93: IEEE Symposium on Visual Languages*, pages 336–341, Bergen, Norway, Aug. 1993.

[8] E.G. Noik. A space of presentation emphasis techniques for visualizing graphs. In *GI '94: Graphics Interface 1994*, pages 225–234, Banff, AL, Canada, May. 1994.

[9] J.D. Ullman. *Principles of Database and Knowledgebase Systems*, volume 1. Computer Science Press, 1803 Research Boulevard, Rockville, MD, 20850, 1988.

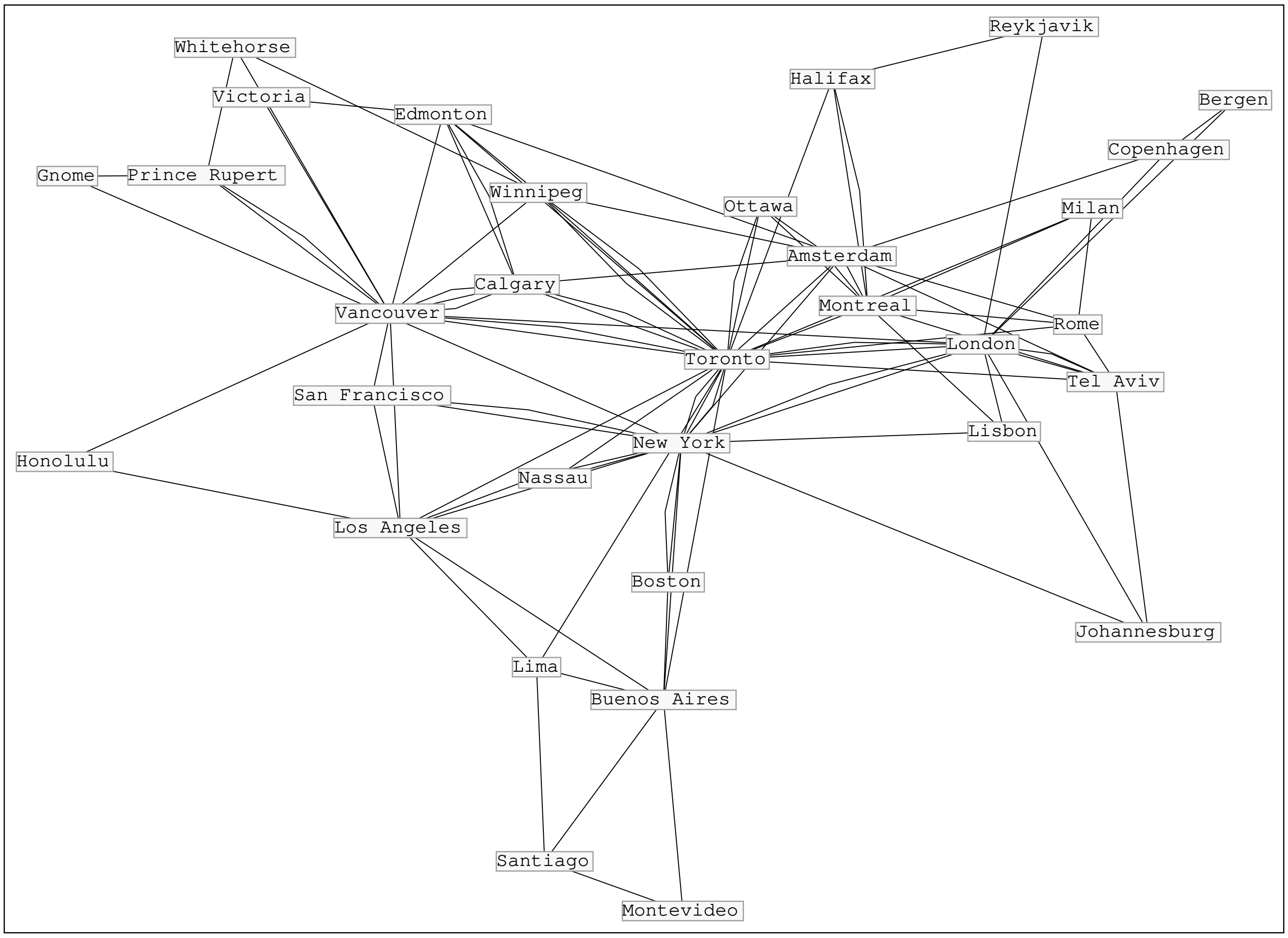
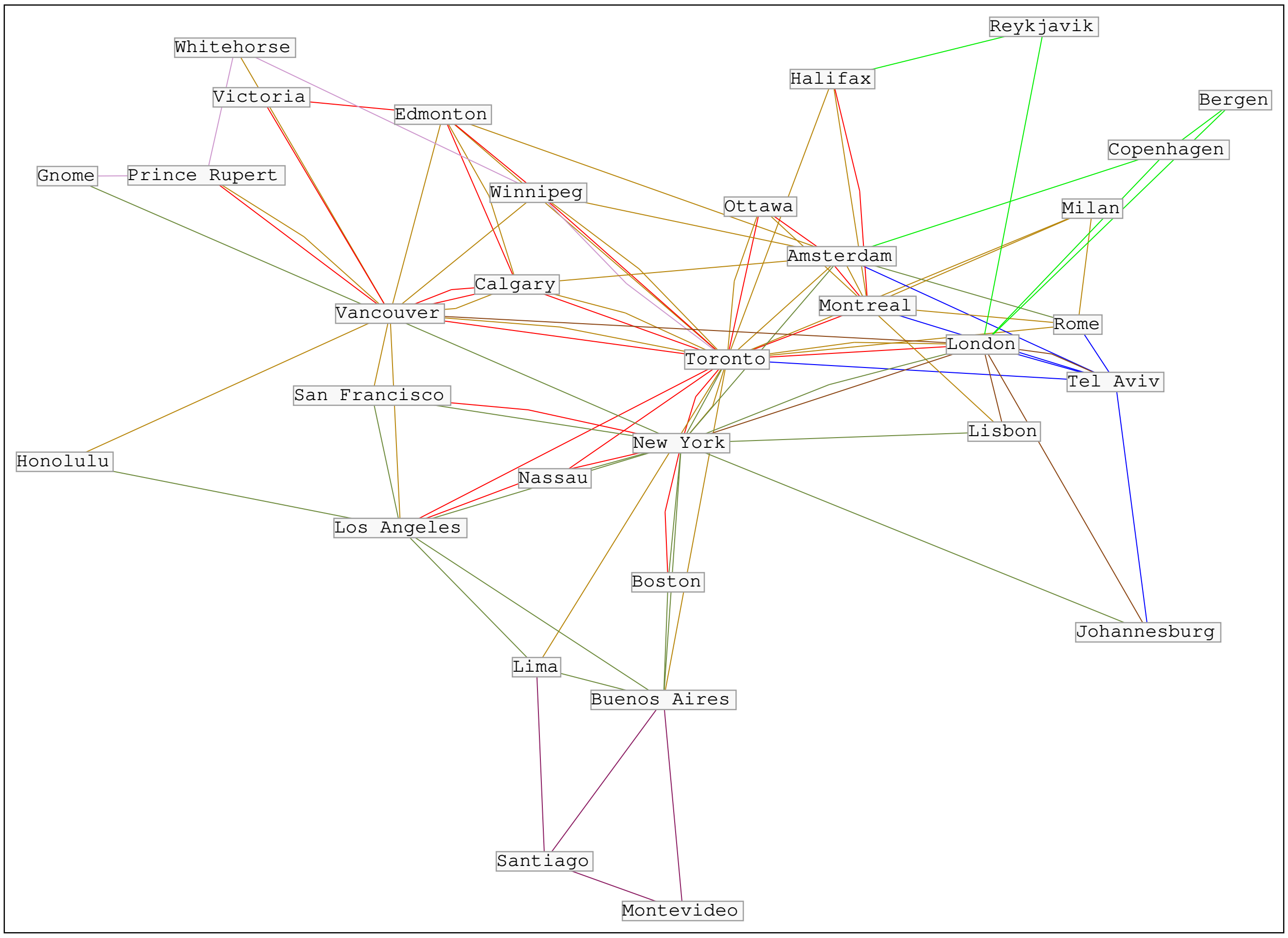
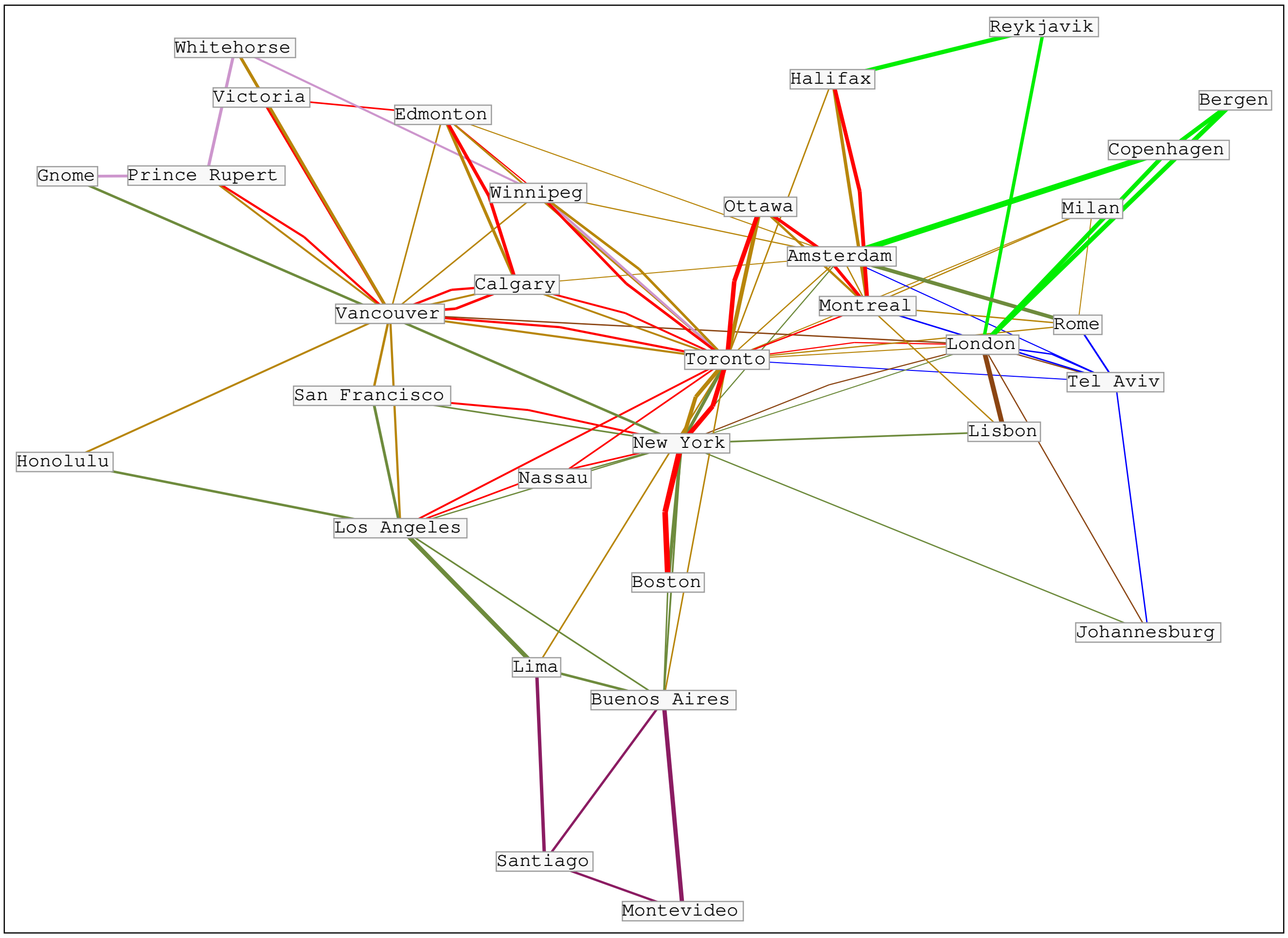


Figure 1: The original drawing of the flights database produced by an anchored force-based layout.



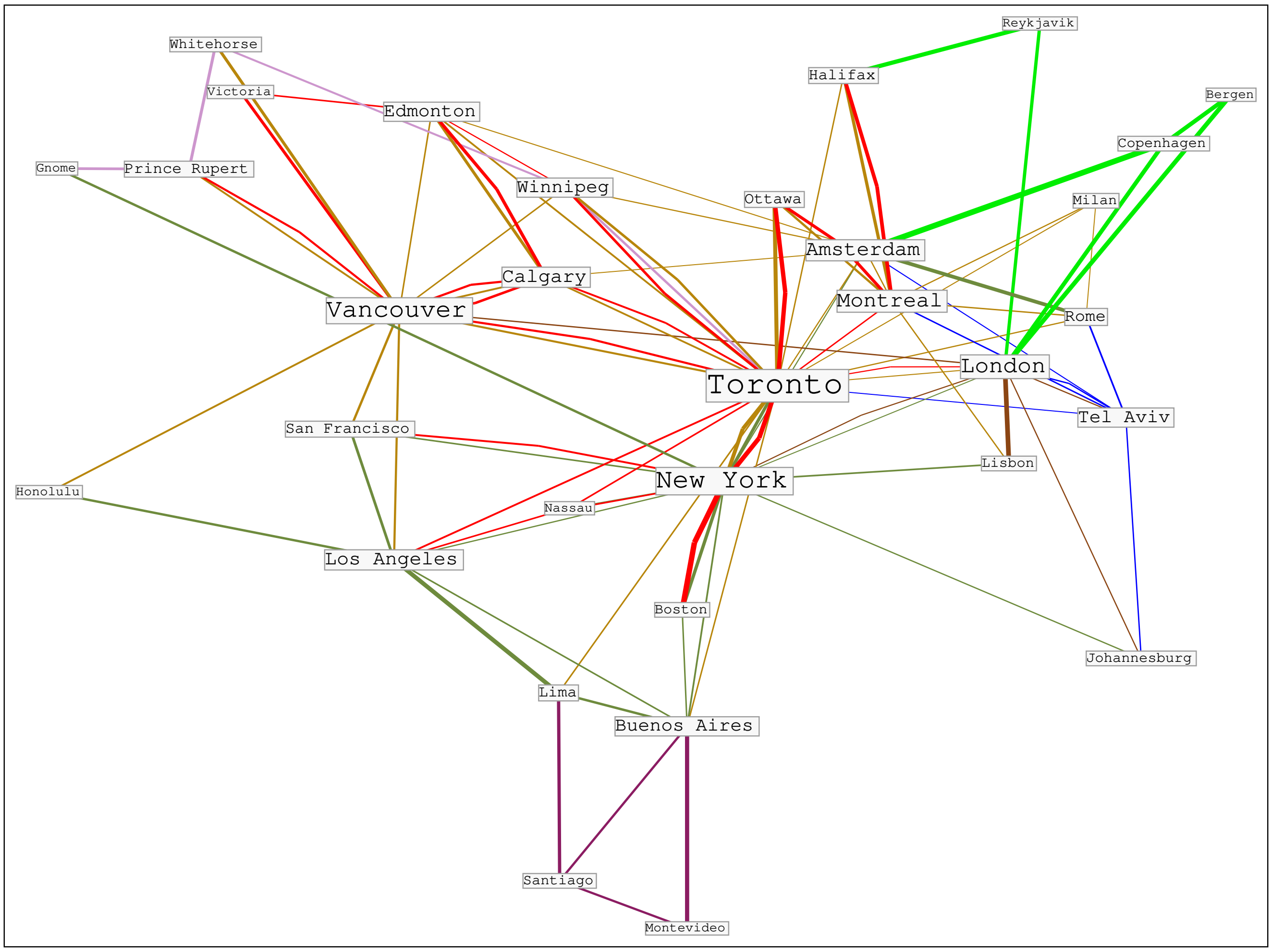

```
#include <dye/colours.h>
#define setColor(Label,Colour) if a.label == Label then a.fg_clr := Colour; endif
actions
  for a in A
    setColor( "aa", DarkOliveGreen4 )
    setColor( "ac", Red )
    setColor( "ba", Chocolate4 )
    setColor( "cp", DarkGoldenrod )
    setColor( "ly", Blue1 )
    setColor( "na", Plum3 )
    setColor( "pl", Maroon4 )
    setColor( "sa", Green2 )
  endfor
```

Figure 2: Colour links based on arc label (airline).



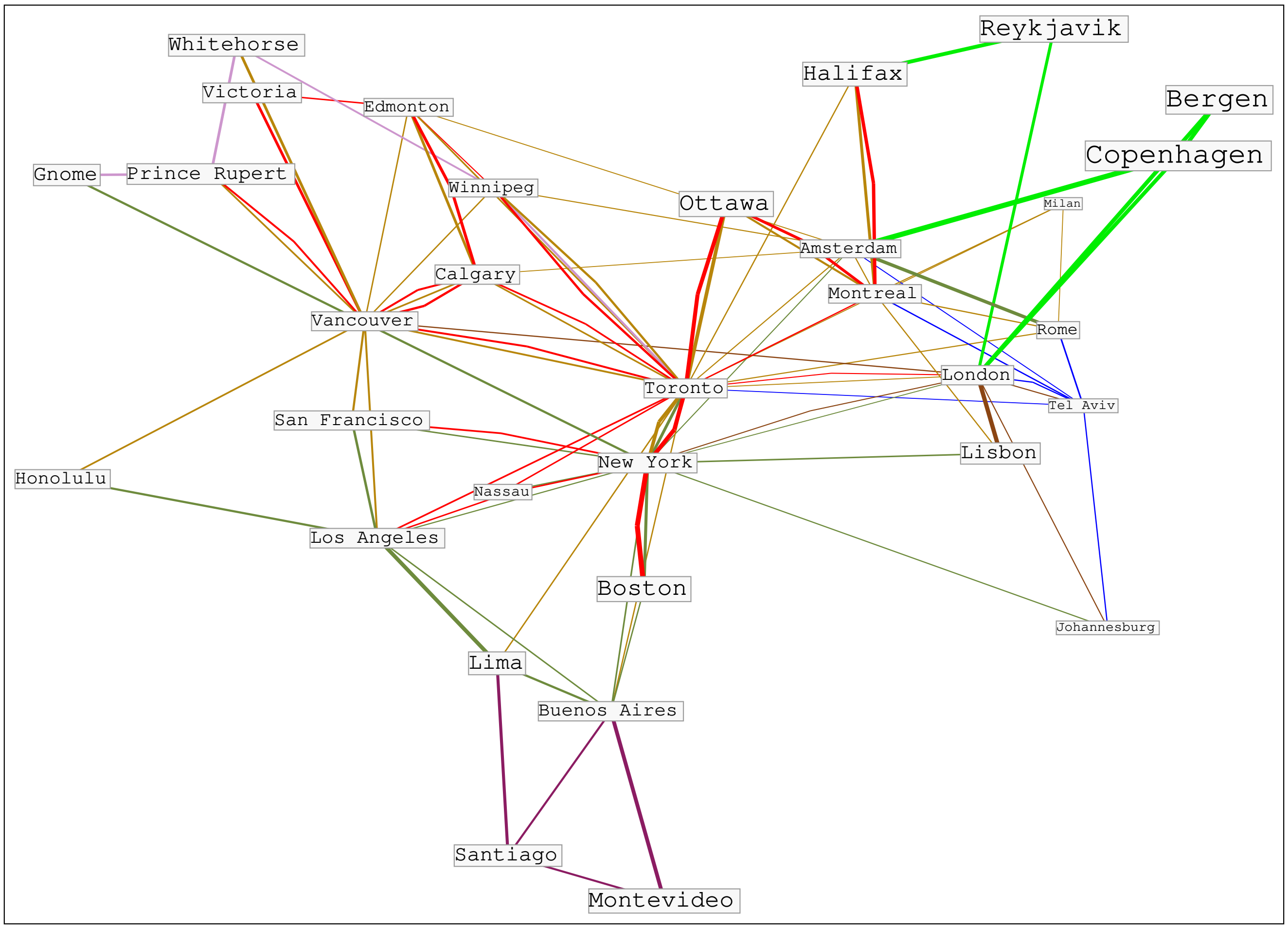

```
...
#define price %0
#define time  %1
doi
  a.api := ( a.price / a.time );
  a.doi := a.api;
actions
  for a in A
    ...set foreground colours...
    a.line.width := (1.0 + (a.doi * 5.0));
  endfor
```

Figure 3: Set link width based on the *price* to *time* ratio.



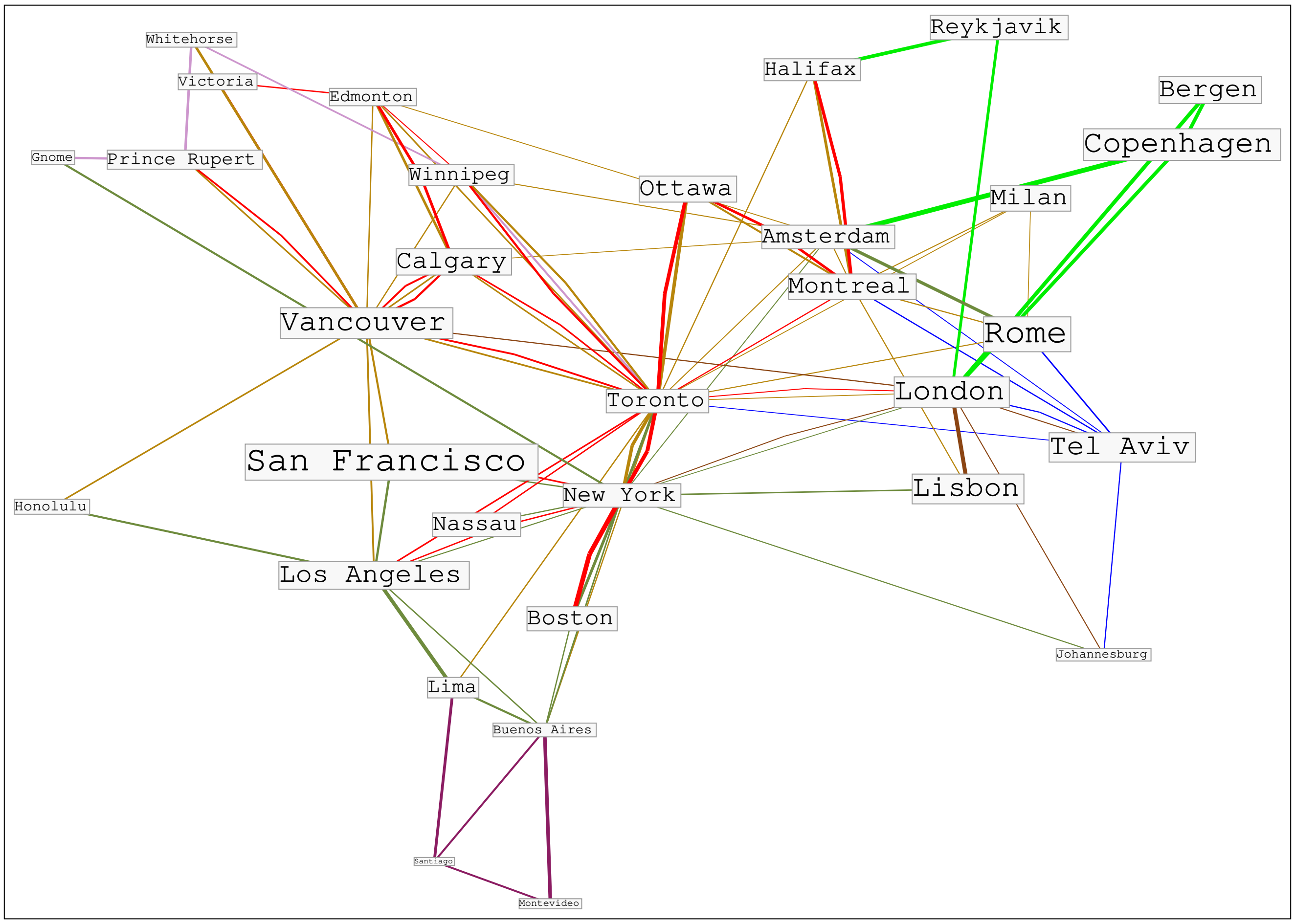

```
...
doi
  v.api := sum( union( v.inA, v.outA ), true, 1 );
  v.doi := v.api;
  ...
actions
  for v in V
    v.s.x = v.s.y := ((0.5 + (v.doi * 2.5)) ** 0.5);
  endfor
  ...
```

Figure 4: Set the size of nodes relative to the number of incident arcs (total number of flights through the corresponding city).



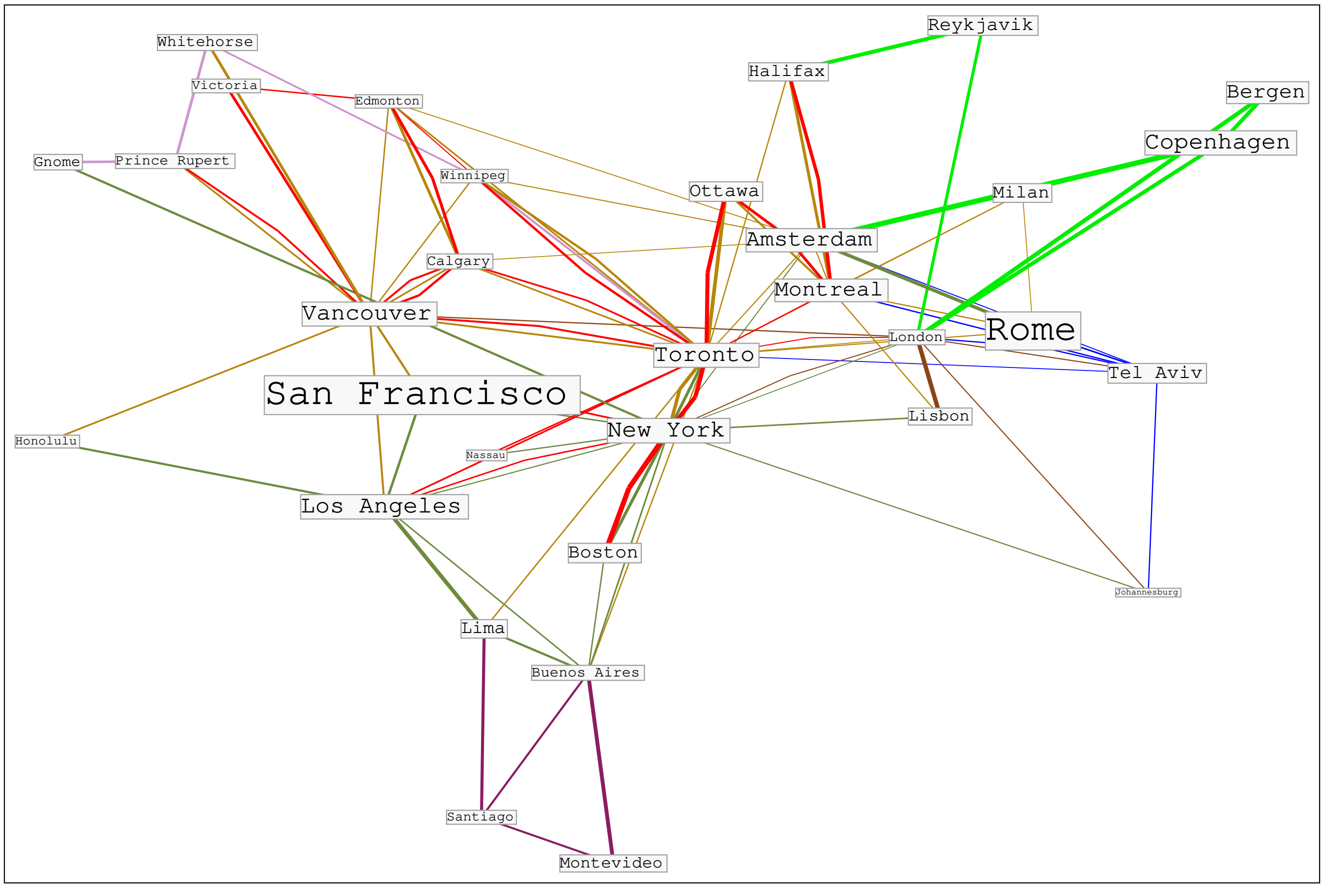

```
...
doi
  v.api := avg( union( v.inA, v.outA ), true, ( a.price / a.time ) );
  ...
actions
  ...
```

Figure 5: Set the size of nodes relative to the average *price* to *time* ratio of incident arcs.



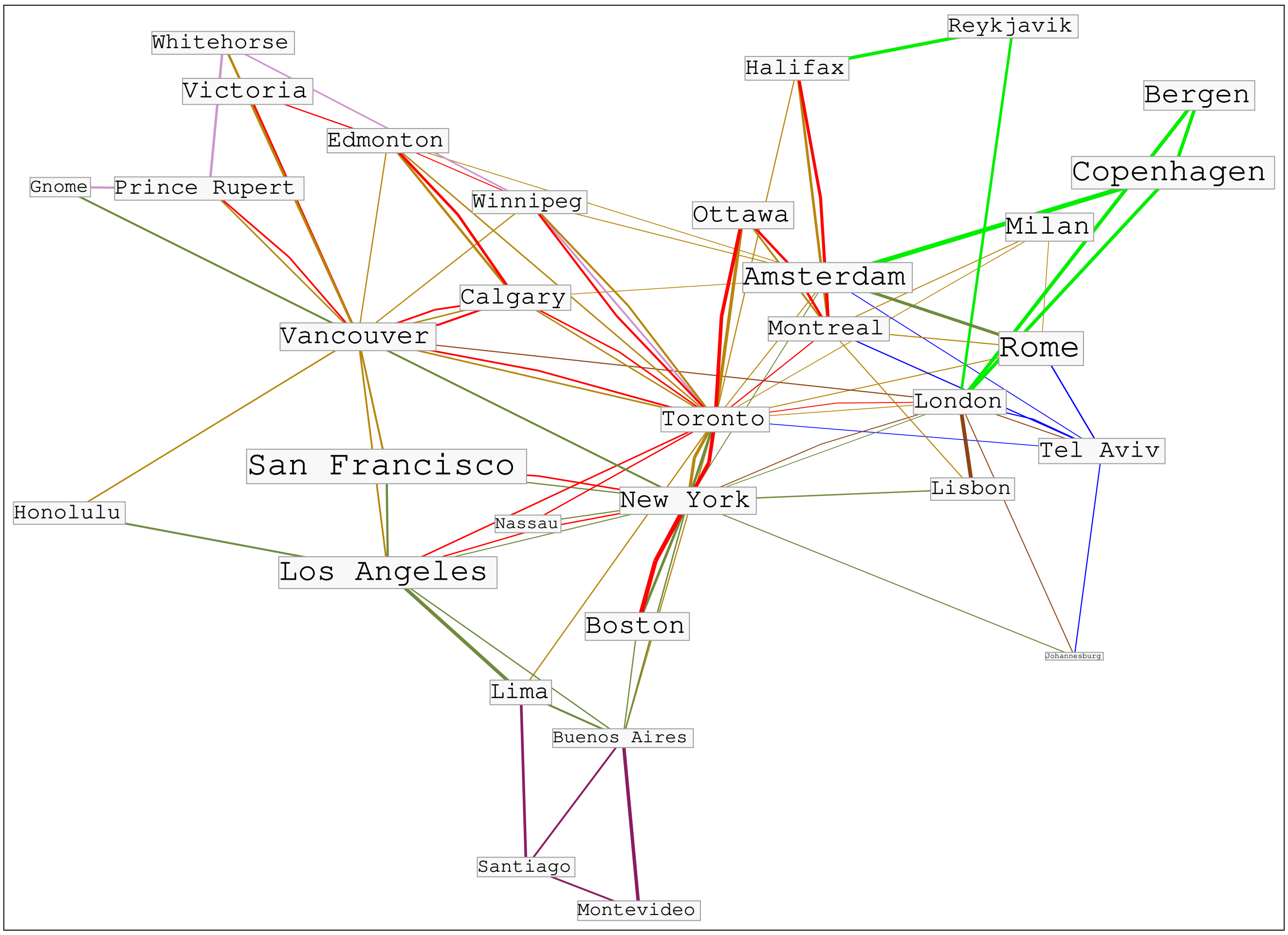

```
...
fpresponse
  fpr := (mag * ((1 - dist) ** 3));
doi
  ...
  v.dist := ((((f.p.x - v.p.x) ** 2) + ((f.p.y - v.p.y) ** 2)) ** 0.5);
  v.doi  := ((v.api * 0.3) + (v.fpi * 0.7));
  ...
actions
  for v in V
    v.s.x = v.s.y := ((0.25 + (v.doi * 4.75)) ** 0.5);
  endfor
  ...
```

Figure 6: A fisheye view with API = average *price* to *time* ratio of incident arcs, and distance = proximity of two nodes in the drawing (geometric distorting fisheye view). Focal points are *San Francisco* and *Rome*.



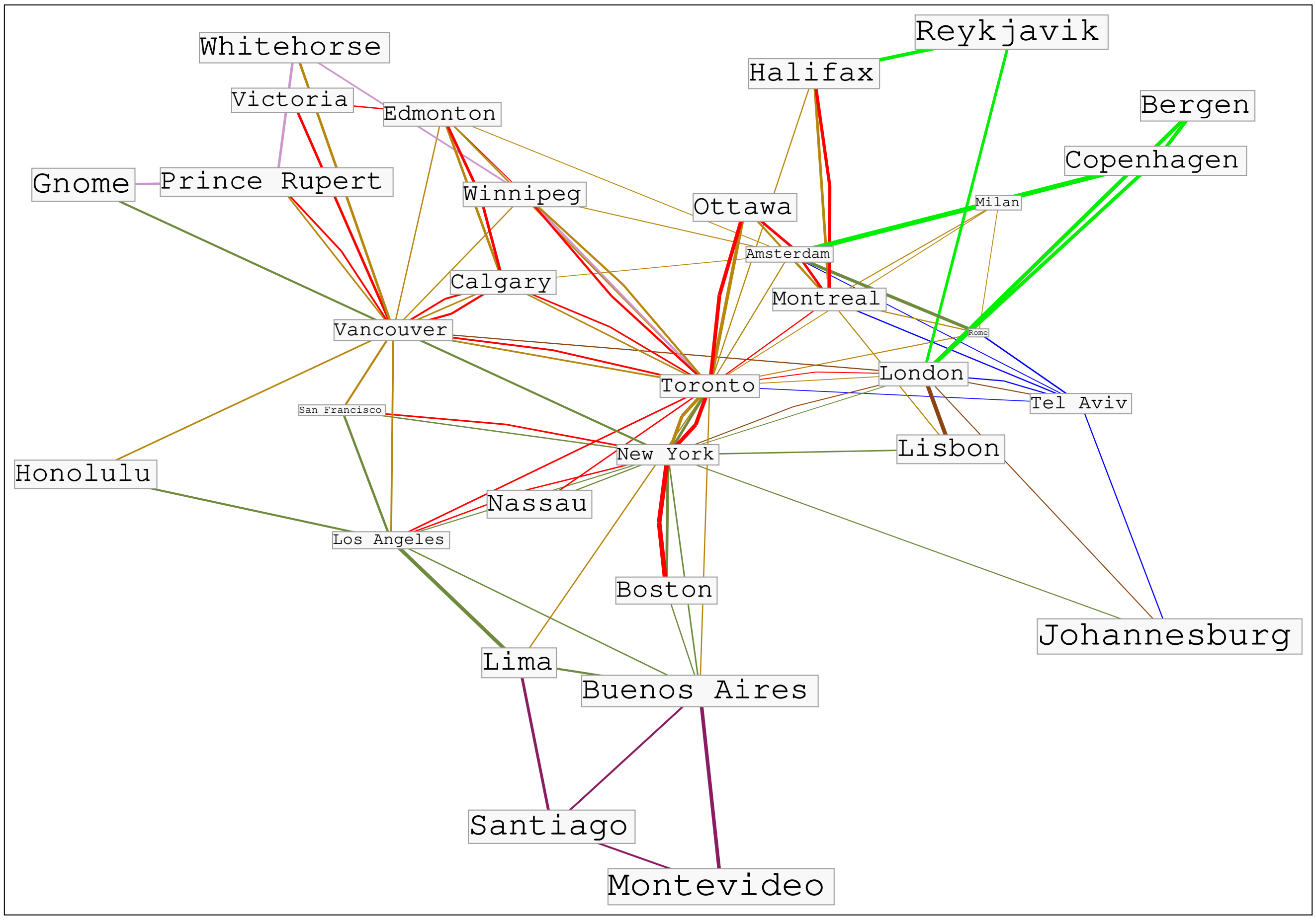

```
...
doi
  ...
  v.dist := minpath( A, undirected, , 1 );
  ...
actions
  ...
```

Figure 7: A fisheye view with API = average *price* to *time* ratio of incident arcs, and distance = minimum number of flights between two cities. Focal points are *San Francisco* and *Rome*.



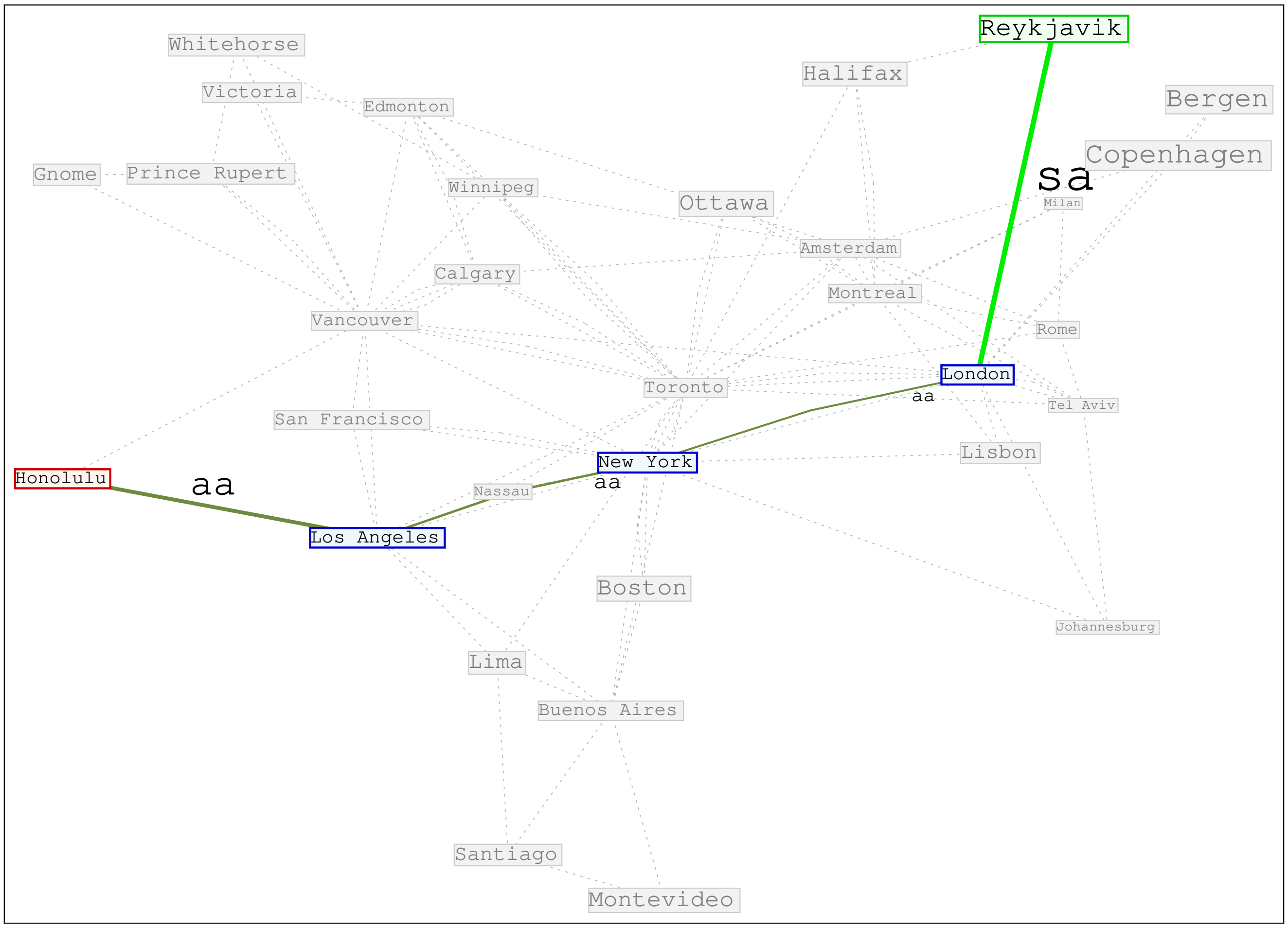

```
...
doi
   ...
   v.dist := minpath( A, undirected, 30, a.time );
   ...
actions
   ...
```

Figure 8: A fisheye view with API = average *price* to *time* ratio of incident arcs, and distance = total amount of flying time between two cities, assuming that every airport visited (including original source and final destination) contributes an additional 30 minutes to the trip. Focal points are *San Francisco* and *Rome*.



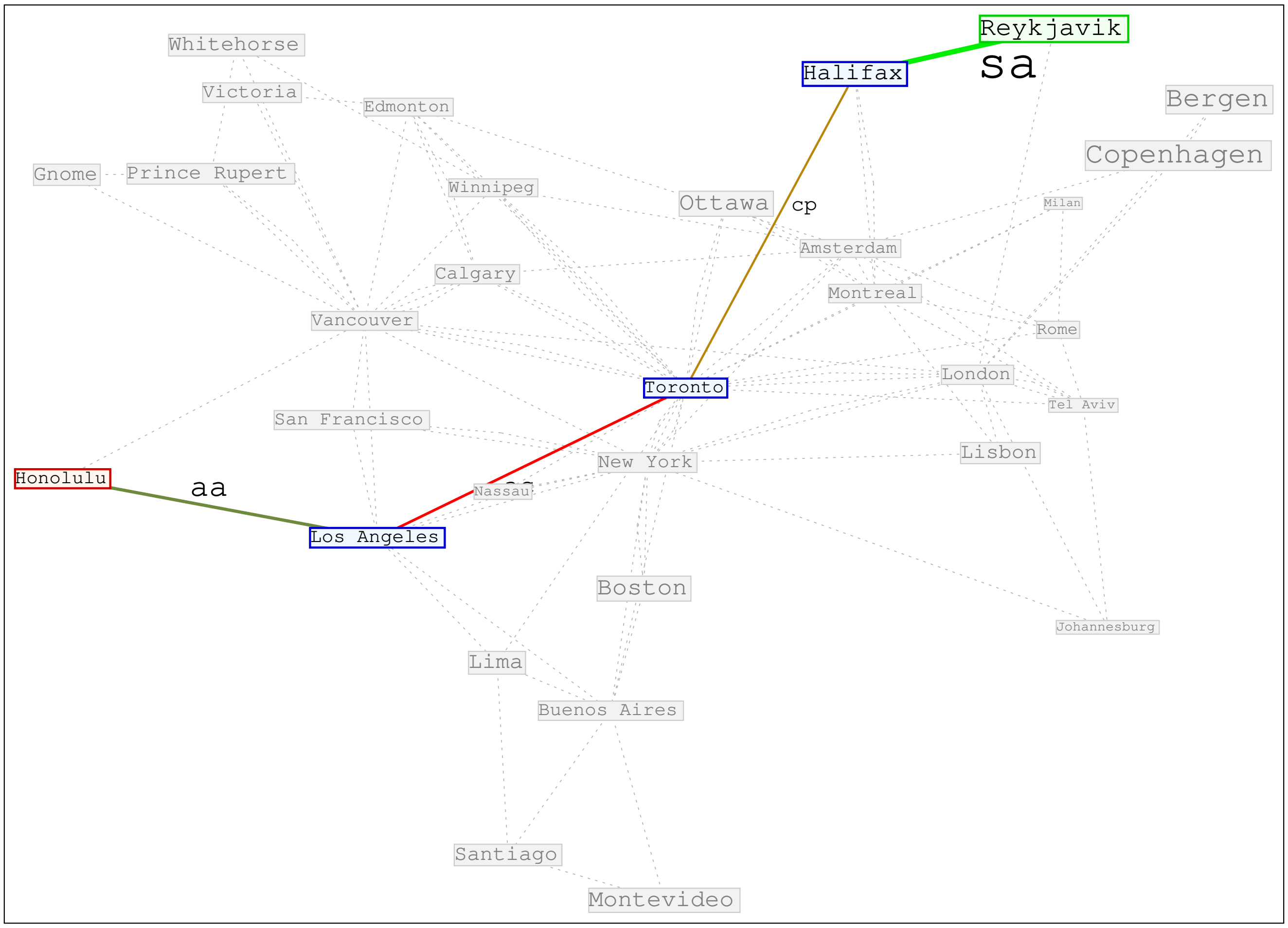

```
...
fpresponse
  fpr[0] := (mag * ((1 - dist) ** 3));
  fpr[1] := (mag * (1 - ((1 - dist) ** 2)));
...
```

Figure 9: A fisheye view with same API, distance, and focal points as in the previous Figure, but now focal points use curve 1 whose response is low at very small distances and high at larger distance (*i.e.*,proximity to focal (peripheral) points lowers relevance).



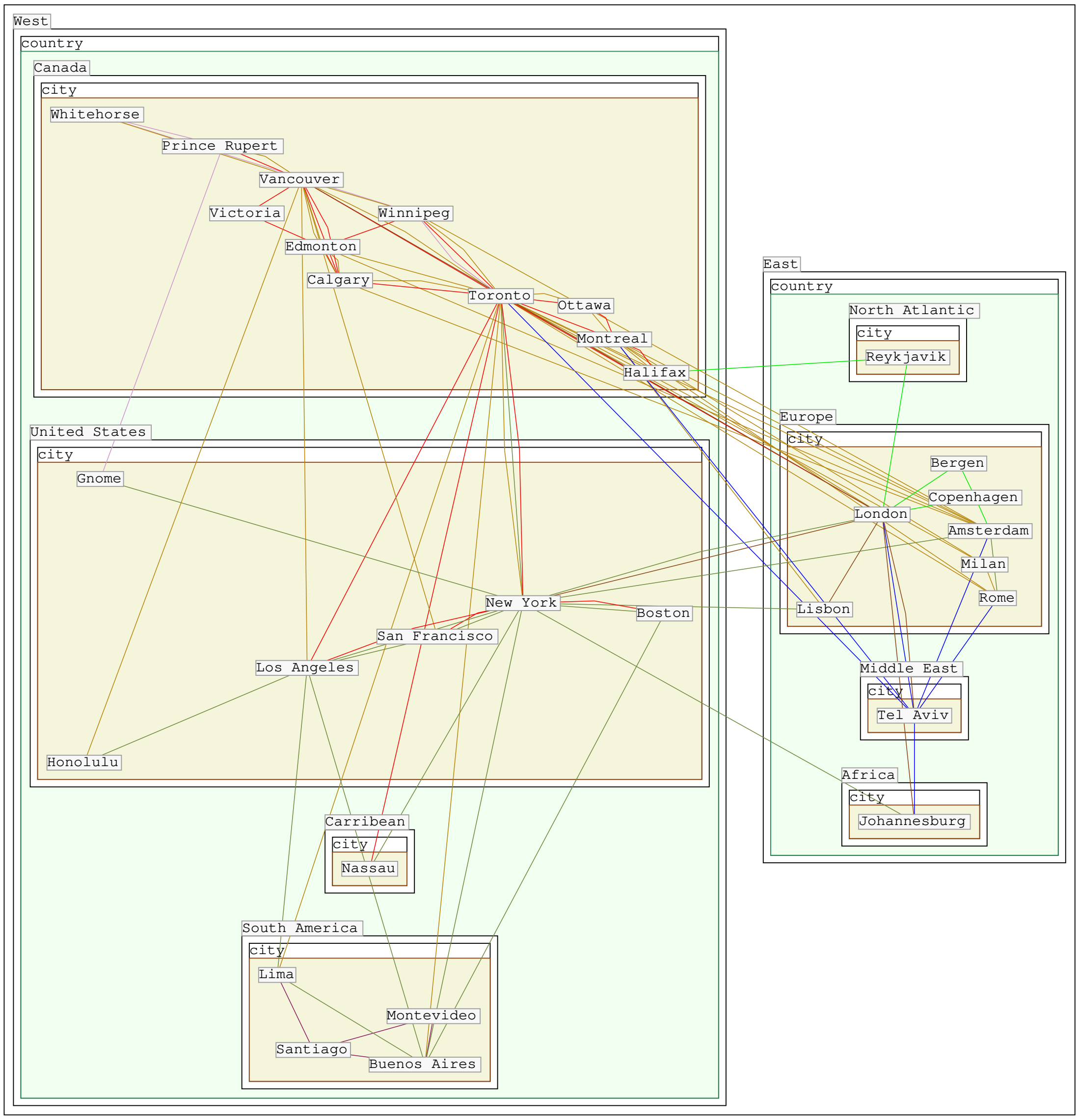

```
...
sets
  v_sources := cjoin( {V}, V0.label == "Reykjavik" );
  v_sinks   := cjoin( {V}, V0.label == "Honolulu" );
  a_minp := minpath( A, v_sources, v_sinks, undirected, , a.price );
  v_minp := cjoin( {V,a_minp}, [ V0.id == A1.from.id | V0.id == A1.to.id ] )
...
actions
  for v in V
    v.s.x = v.s.y := ((0.5 + (v.doi * 2.5)) ** 0.5);
    v.fg_clr := Gray80; v.bg_clr := Gray95; v.txt_clr := Gray50;
  endfor
  for v in v_minp    v.fg_clr  := Blue3;  v.bg_clr := AliceBlue;
                     v.txt_clr := Black;  v.line.width := 2;    endfor
  for v in v_sources v.fg_clr  := Green3; v.bg_clr := Honeydew; endfor
  for v in v_sinks   v.fg_clr  := Red3;   v.bg_clr := Seashell; endfor
  for a in A
    if a in a_minp then a.line.width:=(1+(a.doi*5)); a.label.show:=true; .set fg clrs.
    else a.line.type := dotted; a.fg_clr := Gray70; endif
  endfor
```

Figure 10: Emphasize the minimum cost (cheapest total price) *Reykjavik* to *Honolulu* path.



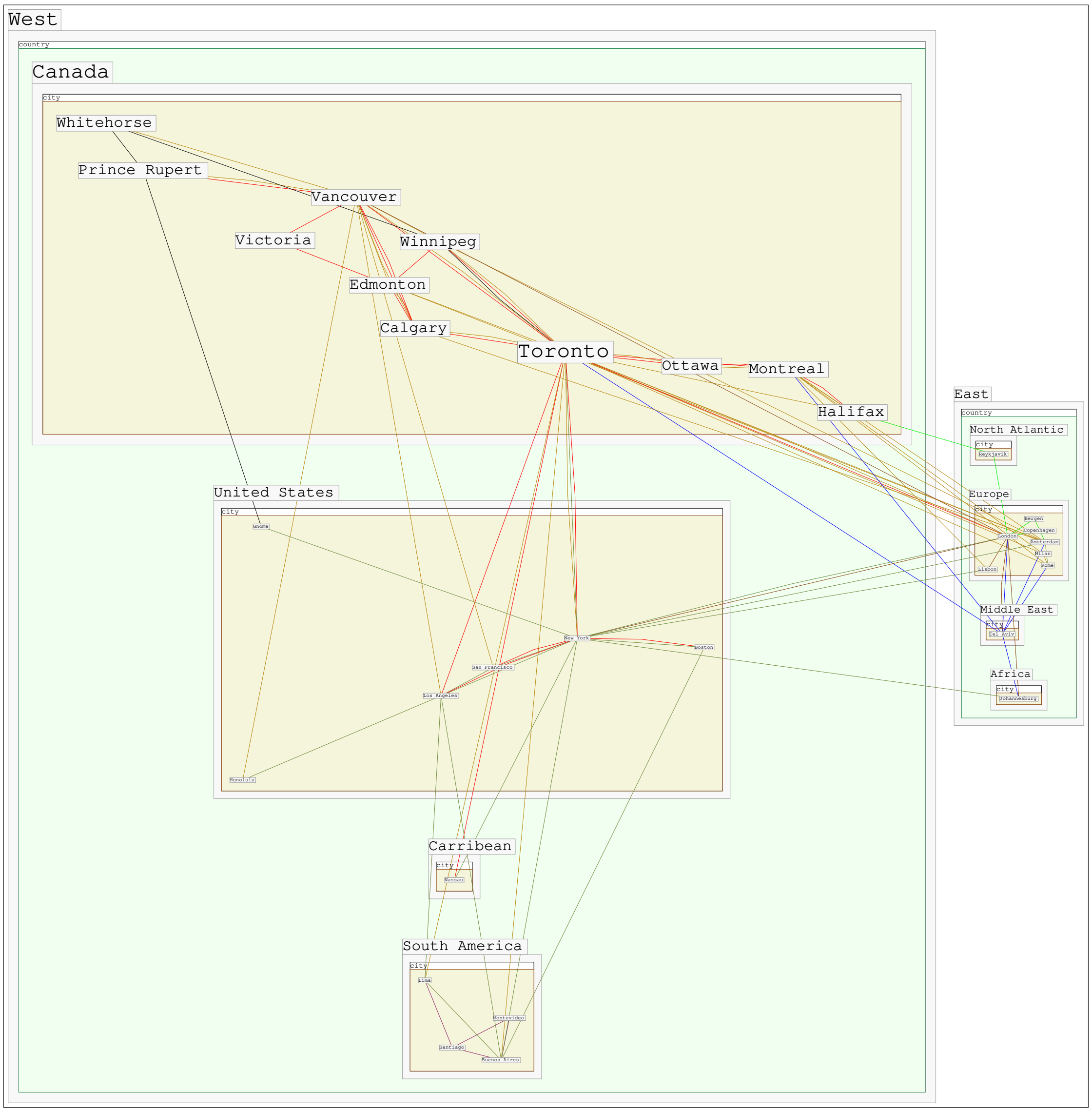

```
...
sets
  ...
  a_minp := minpath( A, v_sources, v_sinks, undirected, 30, a.time );
...
```

Figure 11: Emphasize the minimum time (shortest trip time) *Reykjavik* to *Honolulu* path.




```
...
actions
  G.layout := "1d x minsep 40 40 40";
  for a in A ...set arc colours... endfor
  for b in B
    if b.label == "city" then
      b.fg_clr := SaddleBrown; b.bg_clr := Beige;
      b.layout := "spring e 20 af 2 minsep 20 20 20";
    endif
    if b.label == "country" then
      b.fg_clr := SeaGreen;    b.bg_clr := Honeydew;
      b.layout := "1d y minsep 30 30 30";
    endif
  endfor
```

Figure 12: A nested version of the flights graph: set top-level and blob layout algorithms, and blob colours.




```
...
sets v_nonnested := cjoin( {V}, sum( V0.parB, true, 1 ) == 0 );
fpresponse fpr := (mag * (( 1 - dist ) ** 2));
doi
  v.api  := (-1 * minpath( B, v_nonnested, forward, , 1 ));
  v.dist := minpath( B, undirected, , 1 );
  v.doi  := ((0.3 * v.api) + (0.7 * v.fpi));
  b.doi  := b.from.doi;
actions
  ...
  for v in V v.s.x = v.s.y := ((0.5 + ( 8.5 * v.doi )) ** 0.5);     endfor
  for b in B b.s.x = b.s.y := ((0.5 + ( 8.5 * b.doi )) ** 0.5); ... endfor
```

Figure 13: A fisheye view of a nested graph with focal point *Toronto*. Vertex API varies inversely with nesting depth, and vertex distance is the length of the shortest path to a focal point in the nesting hierarchy. Blob DOI is set to the DOI of its enclosing vertex, and vertex and blob regions are scaled relative to DOI.