SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **00**(S1), 1–5 (1999) *Prepared using* speauth.cls *[Version: 1999/06/11 v1.1a]*

An open graph visualization system and its applications to software engineering

EMDEN R. GANSNER[1] and STEPHEN C. NORTH[1]
[1]*AT&T Labs - Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA*

SUMMARY

We describe a package of practical tools and libraries for manipulating graphs and their drawings. Our design, which aimed at facilitating the combination of the package components with other tools, includes stream and event interfaces for graph operations, high-quality static and dynamic layout algorithms, and the ability to handle sizable graphs. We conclude with a description of the applications of this package to a variety of software engineering tools. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: Graph visualization; software engineering; open systems

INTRODUCTION

Graphs are appropriate models for many problems that arise in computer science and its applications. Graph drawings are a useful way of depicting these models, and so graph visualization has found many applications in the design and analysis of communication networks, linked documents, and the static and dynamic structure of programs. Thus, there is a need for tools to display and manipulate graphs.

Much work in graph manipulation and visualization has focused either on high-level interactive editors or on low-level graph libraries, and the usefulness of both is well established. A middle approach is offered by filters, which read and process an input stream, and produce an output stream. Filters have proven useful in many areas, such as in text processing, program compilation, and digital signal processing. They serve as a computational model for many scripting and functional languages, and are well-suited to tasks that focus on symbolic, language-based computation, and for the automation of repetitive tasks. Manual interactive editors fall short in this area.

This paper describes a toolkit of graph filtering and rendering tools. In the toolkit, filters are on an equal footing with interactive tools. We have not assumed that all programs have a graphical user interface (GUI). This is important because in many situations, language-based programmable tools are an invaluable alternative to interactive tools that must be controlled manually [1]. In principle, this should not be not a fundamental issue in library design because modularity favors layering GUI features over basic data structures. But in practice, systems have not always been designed this way. For example, base graph objects may refer to callback functions whose types are defined in a large, complex GUI client library. Such an occurrence is probably more a matter of implementation convenience than an intractable design flaw, but details can make a big difference in practice, when tools are created and used.

CCC 0038–0644/99/S10001–05$17.50 *Received 13 April 1999*
Copyright © 1999 John Wiley & Sons, Ltd. *Revised 26 May 1999*

Table I. Toolkit components

Libraries		
	Libgraph	attributed graph data structures and I/O
	Dynagraph	incremental layout library
Layout Tools		
	Dot	hierarchical layout
	Neato	"symmetric" layout
Graphical Tools		
	Dotty	programmable interactive graph editor
	Tcldot, Webdot	related interactive front ends
	Grappa	compatible graph package in Java
	Montage	generic ActiveX diagram container
	Tcldg/Dged	Tcl/Tk graph editor for incremental layout
Graph Filters		
	Gpr	generic graph filter
	Sccmap	decomposes graphs into strongly-connected components
	Colorize	computes node colors from initial seeds
	Unflatten	adjusts edges to improve aspect ratio of hierarchical layouts

Creating a common set of graph filters requires sharing graph data. There have been proposals for a standard graph file language [2, 3], but none is widely accepted. The challenge is to find a common graph model that fits a broad range of algorithms and programs. Some of the key decisions include: the typing and encoding of attributes; whether to support multi-graphs, hybrid graphs (having both directed and undirected edges), subgraphs and other higher-order objects (hyperedges and compound edges), edge ports, and syntactic or semantic constraints. Some systems need dynamic graph editing in addition to static graph descriptions. Moreover, programmers seek not only generality, but also efficiency and ease of implementation. So it has been difficult to achieve a consensus on a graph model to underlie a common file language.

With these ideas in mind, we created a toolkit of libraries and programs for creating, filtering, displaying and interacting with graphs. The toolkit contains base libraries for handling attributed graphs; a collection of graph layout algorithms; platform-dependent front-ends; and a complement of file-stream graph processors. The main components are given in Table I.

After discussing related work, this paper describes the various components of our system in detail, providing rationale for the design of the toolkit. It then discusses a wide range of software engineering tools that have been based on the toolkit.

Related Work

Rowe et al.'s graph editor, Grab[4], was the first interactive system described in the literature. It exemplified first-generation systems with hard-wired layout and display algorithms and very limited programming interfaces. As such, it lacked the degree of extensibility and reusability that were developed in later systems, but proved that a generic graph editor can be an important software engineering tool.

Newbery and Tichy's Edge system[2] was notable in being the first object-oriented graph editor toolkit described in the literature, and the first to employ a flexible graph data language for inter-tool communication. The editor could be customized at the C++ object level. This characterizes second-generation systems. Although it pioneered key design concepts in graph visualization tools, it was not refined enough as a practical re-usable software component to fully demonstrate the significance of these ideas.

GraphEd[5] and the Tom Sawyer Software Graph Editor Toolkit[6] were also introduced as well-engineered second-generation systems with APIs to enable addition of new layout algorithms and customization to create task-specific tools. Both led to more flexible third-generation systems integrated with generic graphics and interprocess communication toolkits (such as Tcl/Tk in the case of GraphEd's successor Graphlet[3], or Win32/OLE and Java for Tom Sawyer Toolkit). Da Vinci [7] has a user-interface component with a hard-wired hierarchical layout algorithm including some advanced features such as collapsing of subgraphs into nodes and incremental layout. It also uses Tcl/Tk as an extension language, where an external program defines task or application-specific semantics. The focus in these tools seems to be on defining a powerful, extendible graph editor, and somewhat less on creating a general framework for construction of stream-oriented tools.

Also of note are libraries for prototyping or implementing graph algorithms. Mehlhorn and Näher's LEDA[8] is a C++ data structure library that has a graph class as one of its main components. Knuth's GraphBase project[9] was designed for experiments in programming combinatorial algorithms. Neither LEDA or GraphBase deal with issues of inter-tool communication. LINK[10], written by Berry, Dean, Goldberg, Shannon and Skienna, is an interactive graph environment in Scheme/Tk. The intended application is teaching and exploring discrete mathematics. Its rich set of programming primitives and libraries includes support for hypergraphs. Although LINK does not have any built-in layout algorithms, it is easy to see how external layout servers could be tied in. However, without automatic layout, its application to software visualization is limited.

Libraries

Our toolkit contains two libraries, Libgraph and Dynagraph, for low-level tool construction. Libgraph supports reading, writing and manipulating graph abstractions, allowing fine-tuning of performance-critical code. Dynagraph is layered on top of Libgraph and realizes a framework for displaying incrementally changing graphs. Both share a common graph specification language.

File Language

Libgraph embodies a common attributed graph data language for graph manipulation tools. Embedding tool-specific data and command syntax in graph descriptions makes it difficult to write compatible graph filters. By delegating graph file I/O to Libgraph, graph tools are syntactically compatible by default. (The Libgraph language is conventionally known as the Dot format, after its best-known application.)

We made several engineering decisions based on our earlier experience with the Dag layout program. Although Dag defines a custom layout specification language, we observed that most commands simply define graph objects and their attributes; the few exceptions describe actions on sets of objects. Thus, the Dot language provides syntax for defining (named and anonymous) graphs, nodes and edges, plus the ability to attach string-valued name-attribute pairs to graph components. Sets of objects are modeled as subgraphs.

Sample graph files are exhibited in Figure 1.

```
digraph G {
  a -> b;
  a -> x -> y -> z;
  node28 -> {node29,node30,node31};
}

graph ER {
    node [shape=box]; course; institute; student;
    node [shape=ellipse]; {node [label="name"] name0; name1; name2;}
        code; grade; number;
    node [shape=diamond,style=filled,color=lightgrey]; "C-I"; "S-C"; "S-I";

    name0 -- course;
    code -- course;
    course -- "C-I" [label="n",len=1.00];
    "C-I" -- institute [label="1",len=1.00];
    institute -- name1;
    institute -- "S-I" [label="1",len=1.00];
    "S-I" -- student [label="n",len=1.00];
    student -- grade;
    student -- name2;
    student -- number;
    student -- "S-C" [label="m",len=1.00];
    "S-C" -- course [label="n",len=1.00];

    fontsize=20;
    label = "\n\nEntity Relation Diagram\ndrawn by NEATO";
}
```

Figure 1. Sample files in Dot format

A user can specify tool arguments and options either by setting certain attributes in graph files or by command-line options. The default behavior for tools is to simply pass through any attributes not explicitly operated upon. Thus cooperating tools can communicate through oblivious intermediate filters.

For example, to indicate a subgraph whose nodes `a,b,c,d` are constrained to the same rank (level) in a leveled graph, we write

```
subgraph H {rank = same; a b c d}
```

Tools that deal with rank assignments need to search for and process such subgraphs, while other tools will ignore this subgraph.

Reliance on attributes to specify command options also requires a bit of cooperation between tools to avoid attribute name conflicts. In practice we have not seen conflicts because we have only a few central tools, their functions are usually orthogonal, and they were written by a small group of programmers in close cooperation.

Another decision in the language design was to favor convenience over safety. The graph language has an intuitive, forgiving syntax so users can edit graphs with text editors and write generator scripts easily. All attributes are strings; there are no other data types. (Early in this project, we experimented with automatically converting an external attribute to native representation at runtime, but this presented complications in C because of its reliance on pointers*. So we made it the responsibility of programmers to perform data conversions.)

Programming Model

Since the library already incorporates an external graph model, it seemed reasonable to extend this into an application programming library for graphs. To be effective, the library needed to be expressive yet simple; provide regularity in function names and types; maintain orthogonality among the operations; and give the programmer the ability to write highly efficient code when necessary.

The resulting library provides an in-memory representation of graphs as described in Dot, with graphs, nodes, edges and string-valued attributes. Graph components can be created explicitly, or read from a file. The library provides a basic set of operations on graphs, such as asking how many nodes a graph has, or traversing all of the out edges of a node. When possible, a single function (e.g., `agraphof`) is used to provide properties applicable to all component types, whether graphs, edges or nodes. In order for the programmer to tune an algorithm, the library provides efficient access to attribute values and fine-grained control over memory and sets of objects. This is described in more detail below.

Table II lists Libgraph's primitives.

Libgraph contains no application-specific data. In external files this information is contained in string-valued attributes. To implement algorithms efficiently, it is helpful if these attributes can be stored in memory in native format (including not only character strings, but also booleans, integer and floating point numbers, arrays and records). Libgraph provides a flexible, efficient, although unsafe, mechanism that allows applications to attach any number of records to a graph object. These records can have programmer-assigned names, usually corresponding to a data type. There is a way of finding named records, and also a way of locking a (unique) pointer within each object onto a given record. This design allows layering of C libraries, and also allows programs to operate on attributes in native format directly, and without function calls, in procedures where performance is critical.

Libgraph's implementation involves set maintenance: graphs have node and subgraph sets, and nodes have in- and out-edge sets (per containing graph or subgraph). Libgraph relies on Vo's Libcdt[12] for set operations. Libcdt supports both hashed and ordered sets, but Libgraph uses ordered sets only. (Hashing is appropriate for working with large graphs and can be enabled specifically.) Sets are indexed both by external identifier (usually node name), and by an object ID assigned sequentially in order of creation. The latter feature is important for creating tools that process graphs without scrambling their contents.

Libgraph employs interface *disciplines* as exemplified in Libcdt and SFIO [13] to allow an application to tailor its functions at runtime. The disciplines allow the programmer to control:

- memory management
- the file channel interface
- the object ID allocator
- object event callbacks

*The SWIG interface generator [11] demonstrates an approach to a similar problem.

Table II. Libgraph primitives

Types	
`Agraph_t`	graph or subgraph
`Agnode_t;`	node
`Agedge_t;`	edge
`Agdesc_t;`	kind, *e.g.* strict, directed
`Agdisc_t;`	resource discipline
`Agsym_t;`	attribute symbol table entry
Objects	
`Agraph_t *agraphof(void*);`	get container graph
`char *agnameof(void*);`	get name string
`ulong AGID(void *obj);`	get internal ID
`int AGTYPE(void *obj);`	get object type
Graphs and subgraphs	
`Agraph_t *agopen(char *name, Agdesc_t kind, Agdisc_t *disc);`	create
`int agclose(Agraph_t *g);`	destroy
`Agraph_t *agread(void *file, Agdisc_t *);`	file access
`Agraph_t *agconcat(Agraph_t *g, void *chan, Agdisc_t *disc)`	
`int agwrite(Agraph_t *g, void *file);`	
`int agnnodes(Agraph_t *g),agnedges(Agraph_t *g);`	find size
`Agraph_t *agsubg(Agraph_t *g, char *name, int createflag);`	bind subgraph
`Agraph_t *agfstsubg(Agraph_t *g), agnxtsubg(Agraph_t *);`	search subgraphs
`Agraph_t *agparent(Agraph_t *g),*agroot(Agraph_t *g);`	move in subgraph tree
Nodes	
`Agnode_t *agnode(Agraph_t *g, char *name, int createflag);`	create node
`Agnode_t *agidnode(Agraph_t *g, ulong id, int createflag);`	
`Agnode_t *agsubnode(Agraph_t *g, Agnode_t *n, int createflag);`	bind in subgraph
`Agnode_t *agfstnode(Agraph_t *g), *agnxtnode(Agnode_t *n);`	search nodes
`int agdelnode(Agraph_t *g, Agnode_t *n);`	destroy
`int agrename(Agraph_t *g, Agnode_t *n, char *newname);`	
`int agdegree(Agnode_t *n, int use_inedges, int use_outedges);`	
Edges	
`Agedge_t *agedge(Agnode_t *t, Agnode_t *h, char *name, int createflag);`	bind edge
`Agedge_t *agsubedge(Agraph_t *g, Agedge_t *e, int createflag);`	bind in subgraph
`int agdeledge(Agraph_t *g, Agedge_t *e);`	destroy
`Agnode_t *aghead(Agedge_t *e), *agtail(Agedge_t *e);`	get endpoints
`Agedge_t *agfstedge(Agnode_t *n), *agnxtedge(Agedge_t *e, Agnode_t *n);`	search edges
`Agedge_t *agfstin(Agnode_t *n), *agnxtin(Agedge_t *e);`	
`Agedge_t *agfstout(Agnode_t *n), *agnxtout(Agedge_t *e);`	
`int agflatten(Agraph_t *graph, int flag);`	linearize edge sets
String Attributes	
`Agsym_t *agattr(Agraph_t *g, int kind, char *name, char *value);`	bind attribute
`Agsym_t *agattrnxt(Agraph_t *g, int kind, Agsym_t *attr);`	search attributes
`char *agget(void *obj, char *name);`	
`int agset(void *obj, char *name, char *value);`	
Records	
`void *agnewrec(Agraph_t *g, void *obj, char *name, unsigned int size);`	create record
`Agrec_t *aggetrec(void *obj, char *name, int move_to_front);`	find
`int agdelrec(Agraph_t *g, void *obj, char *name);`	remove
Callbacks	
`Agcbdisc_t *agpopdisc(Agraph_t *g);`	install callback functions
`void agpushdisc(Agraph_t *g, Agcbdisc_t *disc);`	remove
`void agmethod(Agraph_t *g, void *obj, Agcbdisc_t *disc, int initflag);`	invoke super-method

Event callbacks are a powerful mechanism that allows independently-designed modules to communicate about graph events. For example, a display module can rely on callbacks to update layouts, while other modules update graphs via Libgraph primitives. This style of programming is employed in Tcldot, described in a following section.

Dynamic Layout Managers

Dynamic layout is needed when a program wishes to handle graph changes incrementally. This happens in interactive graph editing, or when visualizing dynamic graph data, or when browsing subsets of large graphs by incremental navigation.

To support dynamic layout managers with Libgraph, we defined a new interface called Dynagraph. Architecturally, the Dynagraph interface is layered over Libgraph. A Dynagraph layout manager provides on-line layout service for a specific class of diagrams. For example, the DynaDAG program maintains hierarchical layouts incrementally, and other libraries have been created for orthogonal and force-directed layouts.

A Dynagraph layout manager communicates with an external agent (typically a graphical interface) by means of client function calls and resulting layout event callbacks. The available function calls (and symmetrically, callback events) are:

open,close	a dynamic layout
insert, modify, delete	an object
control	event delivery

The event delivery control permits batching of events through event queues. Any sequence of input events may be queued before requesting callback events to update the graph diagram display. This capability is essential for loading or importing external diagrams atomically, and allows interactive control over redrawing.

Figure 2 shows an animation sequence from the DynaDAG layout manager, reflecting a sequence of node and edge insertions.

Retrospect

Libgraph supports primarily stream-based graph tools and efficient in-memory operations. Some of its features clearly represent attempts to engineer around language features that C lacks but are found in more modern languages. Some examples are inheritance, polymorphism, and object initialization and finalization. Ongoing work is aimed at capturing Libgraph's model in an object-oriented, compiled language. Grappa (described in a later section) is a Java implementation, and the C++ graph template library of Lee, Lumsdaine and Siek[14] partly addresses the problem of cleanly managing the typed attributes necessitated by the use of multiple libraries.

Areas where we have contemplated improvements include:

- **typed file attributes**. For example, intrinsic support for arrays and records would remove a burden from application code.
- **higher-order objects**. Libgraph's only higher-order objects are subgraphs. (These could be considered directed hyperedges, as the order of object insertion into subgraphs is stored.) A more natural notation for hyperedges and compound edges would be reasonable. But C programming seems to require trading off type-safety against the

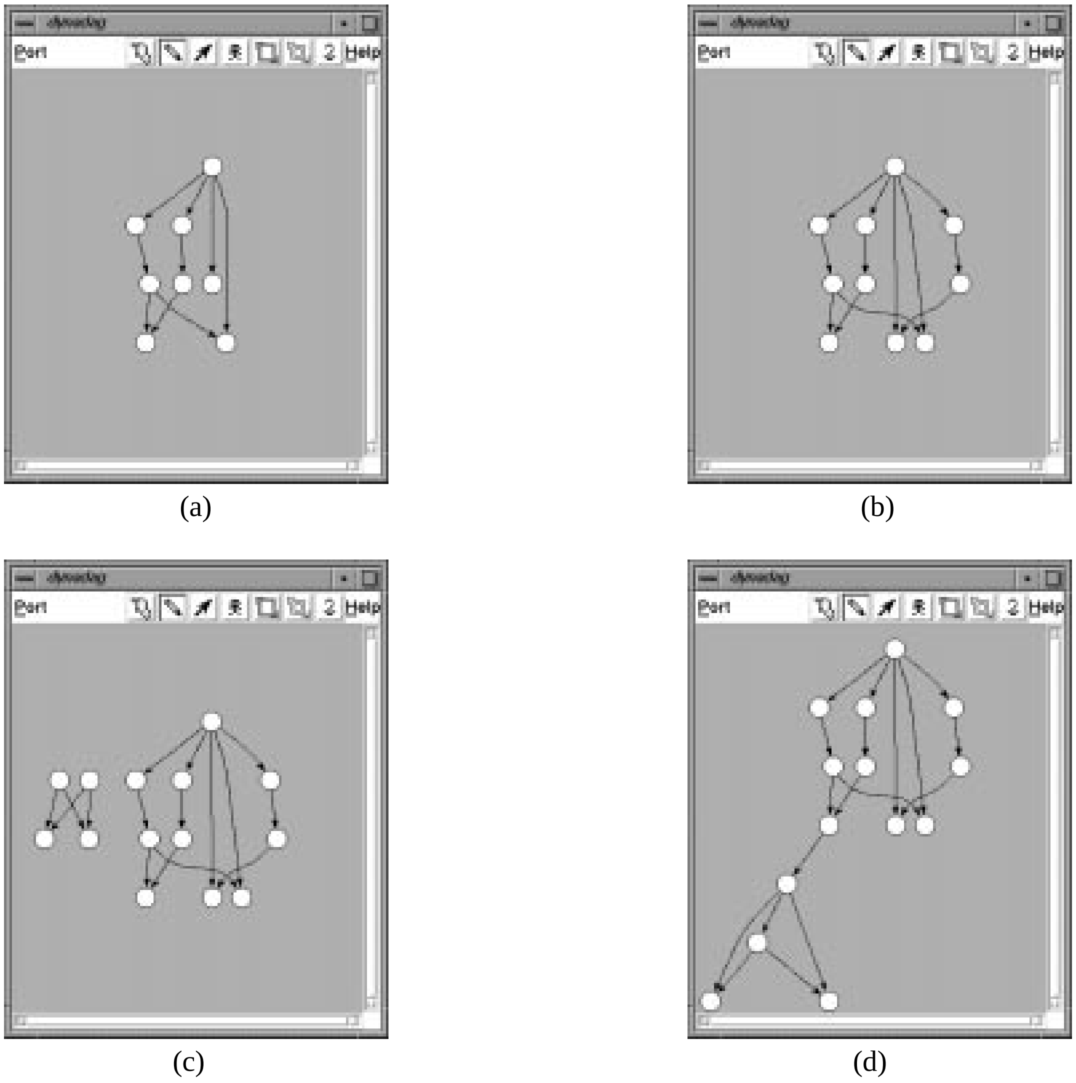


Figure 2. Example DynaDAG animation sequence

number of interface functions. For example, type-safe functions to create, search, modify, and delete binary edges might require individual functions for every pair of endpoint types.

- **on-line operations**. Graph files are static. In recent experiments with dynamic layouts, we modified Libgraph's parser to work line-by-line, and added new commands for synchronization of communication streams between clients and servers.
- **semantic constraints**. These could range over some simple syntactic constraints ("all nodes in this subgraph must have `color=red` or `degree` $\leq$ `4`"), to sophisticated tests ("this subgraph must be planar"). The consequent problem of how to incorporate executable code in graph descriptions seems to be at least partially solved by new

language platforms for mobile networked code that allow downloading constraint-checkers at runtime.

Layout Tools

Graph layout tools are the cornerstone of our system. The main layout tools are Dot (for directed graphs) and Neato (for undirected graphs). These are stream-oriented programs that read graphs, compute layouts, and write the graphs either as layouts in a graphics language (PostScript, GIF, etc.) or as attributed graphs whose objects have associated layout coordinates. The layout programs can operate either as stream tools, or as layout servers to interactive interfaces.

The primary goal of our layout tools has been to provide good diagrams of reasonable size graphs, and scale well to large graphs. Our criterion for "good" is that drawings should be as acceptable as ones a human might make with a manual drawing editor. "Reasonable" size means no larger than what fits on a single screen or printed page with readable labels; this means perhaps 50 or 100 nodes. When graphs are much larger, additional interaction or layout techniques are often needed to cope with visual complexity.

Dot[15] makes hierarchical layouts of directed graphs. It is a successor to Dag[16] and incorporates the general approach of Warfield, Carpano, and Sugiyama et al. [17, 18, 19] though most of its sub-heuristics are new.

Dot makes good layouts and provides an assortment of shapes, styles, and colors appropriate to software engineering diagrams. For example, for drawing data structure graphs, Dot can format nodes as nested box lists, with ports for connecting pointers to boxes that represent record fields. Dot also incorporates an algorithm for drawing graphs with *clusters* or recursive node set partitions [20]. Clusters at the same nesting level are drawn in non-overlapping rectangles. The intended applications are in diagrams of hierarchical structures, such as nested source code modules. Figure 3b shows a Dot layout for the graph in Figure 3a. Figure 4 is a layout of a finite state machine, and Figure 5 shows a distributed program drawn as a clustered graph.

For some tasks involving undirected graphs (such as the entity-relationship database schema graph of Figure 6), layouts that emphasize path distance and symmetry seem better than hierarchical layouts. Force-directed placement has been applied successfully to this problem. Neato is an undirected graph embedder that uses the virtual physical model of Kamada and Kawai [21] (closely akin to layout by multidimensional scaling [22]). Neato is compatible with Dot to the extent of accepting the same input files and command line options. The user interface tool Dotty (described in the next section) can switch between either of the two by changing the path name of the layout server.

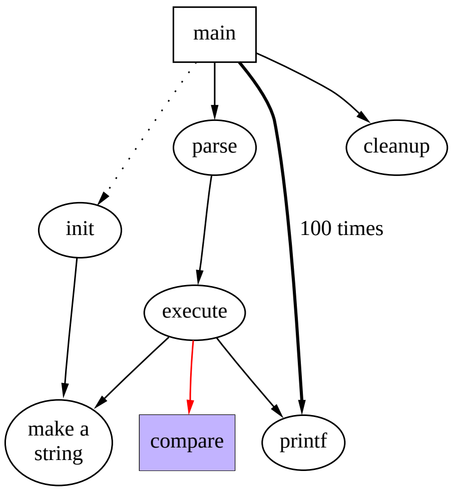
Interface Components

The primary interactive tool is Dotty, a browser that can display graph layouts and incorporate them in user interfaces for external programs. Dotty can be controlled either through a WYSIWYG interface, or through a text (procedural) interface. As a stand-alone tool, Dotty is similar in operation to other systems based on treating pictures of graphs as structured objects. GRAB, EDGE, GraphEd, Graphlet and da Vinci are some well-known examples. Like these tools, Dotty provides menu and pointer-driven graph operations.

The procedural interface is convenient for algorithmic operations. A simple example would be to merge two graphs. The procedural interface allows for re-programming the graphical

```
digraph G {
  size ="4,4";
  main [shape=box]; /* comment */
  main -> parse [weight=8];
  parse -> execute;
  main -> init [style=dotted];
  main -> cleanup;
  execute -> { make_string; printf}
  init -> make_string;
  edge [color=red];
  main -> printf [
    style=bold,label="100 times"
  ];
  make_string [label="make a\nstring"];
  node [
    shape=box
    style=filled
    color="blue"
  ];
  execute -> compare;
}
```

(a)



(b)

Figure 3. A sample graph description and its layout

interface at the script-language level. This turned out to be a good decision. For example, the left mouse button can be bound to a function that highlights all edges attached to the node under the mouse pointer. The underlying programming language has primitives to start external processes and to establish interprocess communication channels. This enables Dotty

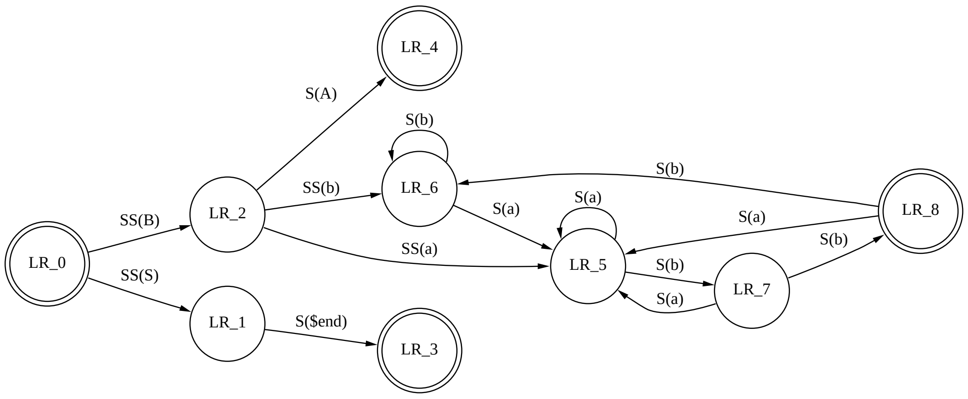


Figure 4. Dot drawing of a finite state machine

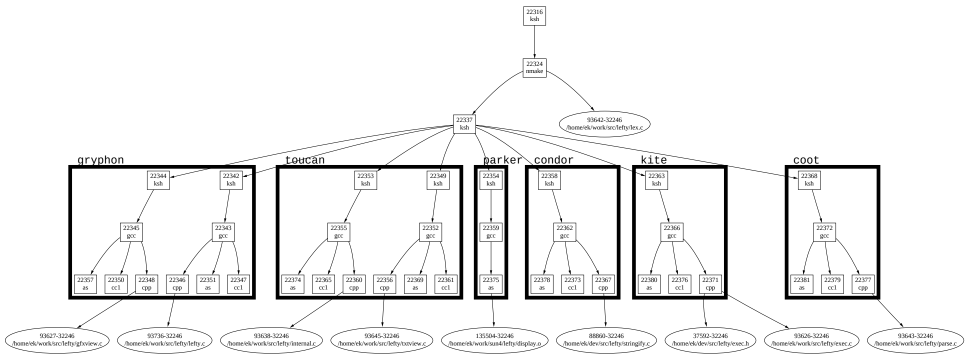

22316 ksh
22324 nmake
22337 ksh
93642-32246 /home/ek/work/src/lefty/lex.c
gryphon
toucan
parker
condor
kite
coot
22344 ksh
22342 ksh
22353 ksh
22349 ksh
22354 ksh
22358 ksh
22363 ksh
22368 ksh
22345 gcc
22343 gcc
22355 gcc
22352 gcc
22359 gcc
22362 gcc
22366 gcc
22372 gcc
22357 as
22350 cc1
22348 cpp
22346 cpp
22351 as
22347 cc1
22374 as
22365 cc1
22360 cpp
22356 cpp
22369 as
22361 cc1
22375 as
22378 as
22373 cc1
22367 cpp
22380 as
22376 cc1
22371 cpp
22381 as
22379 cc1
22377 cpp
93627-32246 /home/ek/work/src/lefty/gfxview.c
93736-32246 /home/ek/work/src/lefty/lefty.c
93638-32246 /home/ek/work/src/lefty/internal.c
93645-32246 /home/ek/work/src/lefty/txtview.c
135504-32246 /home/ek/work/sun4/lefty/display.o
88860-32246 /home/ek/dev/src/lefty/stringify.c
37592-32246 /home/ek/dev/src/lefty/exec.h
93626-32246 /home/ek/work/src/lefty/exec.c
93643-32246 /home/ek/work/src/lefty/parse.c

Figure 5. Dot drawing of communication in a distributed program

to operate as a front-end for other processes. In this context, graphs can represent state information maintained by a back-end process, and user actions can be bound to functions that translate graph operations to corresponding state change requests sent to the back-end.

Dotty is an application written in the Lefty[23] graphical editor. Lefty programs are written in a scripting language (also called Lefty) whose semantic level is about the same as conventional scripting languages (Visual Basic, Unix shell, Tcl/Tk). Lefty has string variables with automatic runtime conversion for arithmetic, associative arrays, hierarchical namespaces for organizing code and data, and functions with arguments and local variables. Its standard environment includes function libraries for working with files and processes, and a collection of common graphical widgets such as canvases with scroll bars, dialog boxes and file selection widgets. Like most similar programming systems, Lefty runs on a variety of popular window systems. Lefty has a small library of C code specifically for supporting graph editors.

Dotty itself is constructed as two co-operating processes, Dot and Lefty. Lefty runs Dot to make graph layouts. These programs communicate via pipes, as shown in Figure 7, with the process hierarchy indicated by the graph of rectangles, files drawn as ellipses, and pipes and sockets shown as circles.

A disadvantage of our design is that sometimes the same function must be provided by two different applications. For example, both Dot and Dotty need to render nodes, but Dotty has a different language, running in a different environment, and so does not use Dot's drivers. This opens a hole for incompatibility between Dot's renderer and Dotty's.

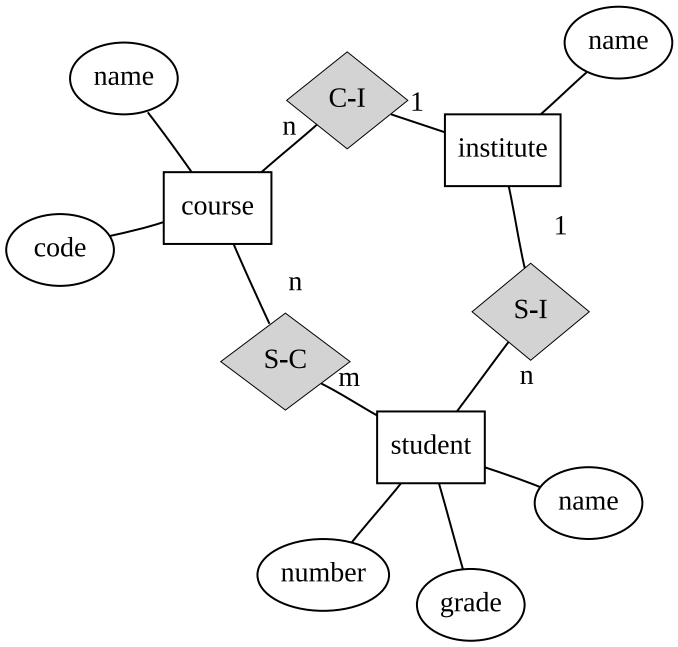


Figure 6. A layout from Neato

On the other hand, Dotty shares many of the virtues of the best alternative scripting languages. For example, like Tcl/Tk, Python and Perl, it is very portable, and hides low-level details such as fonts and color maps.

Tcldot and Tcldg

After we wrote Dotty, other scripting languages with graphics toolkits became widespread. One of these, Tcl/Tk[24], is portable and is supported by an active community that has created many extension packages for graphics and networking. Tcldot combines our graph tools with Tcl.[†] Specifically, it binds Libgraph functions to Tcl commands, and has internal functions for rendering graphs in Tk canvases. Tcldot is essentially Dotty in Tcl, where the base graph editor is customized by loading scripts and extensions. Although Dot is linked in as a library, communication is still handled through graph string attributes.

Tcldot introduced graphics drivers for GIF and HTML image maps to our tool set. Webdot is an interesting application of this facility. It is a CGI script, written in several hundred lines of Tcldot code, to display graphs in remote web pages. Image maps provide a way of employing a graph's nodes as a user interface for web navigation.

To support dynamic layouts, a new user interface toolkit, Tcldg, was written in Tcl/Tk. The renderer is not specifically bound to Tk, so other devices such as Pad++[25] can be used as well. The Tcl script writer has control over all interactions and events that relate base graphs, dynamic graph layouts, and canvas objects. This provides considerable flexibility to script authors. Further details can be found in [26]. Tcldg was created specifically to prototype distributed network management systems for the MONET project [27], and incorporates an

[†]Tcldot and Tcldg were written by John Ellson, Lucent Corp.

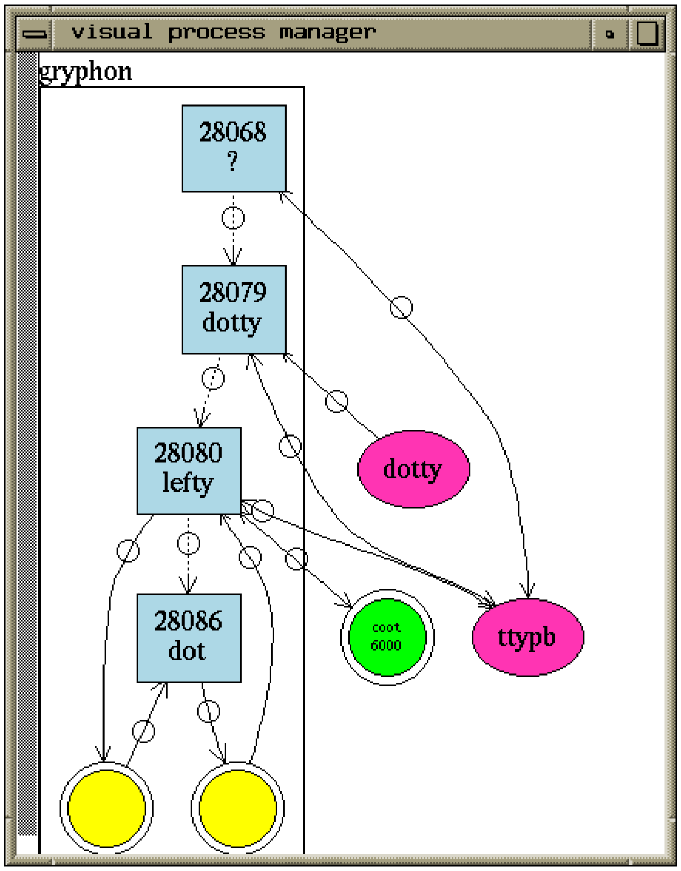


Figure 7. A Dotty session

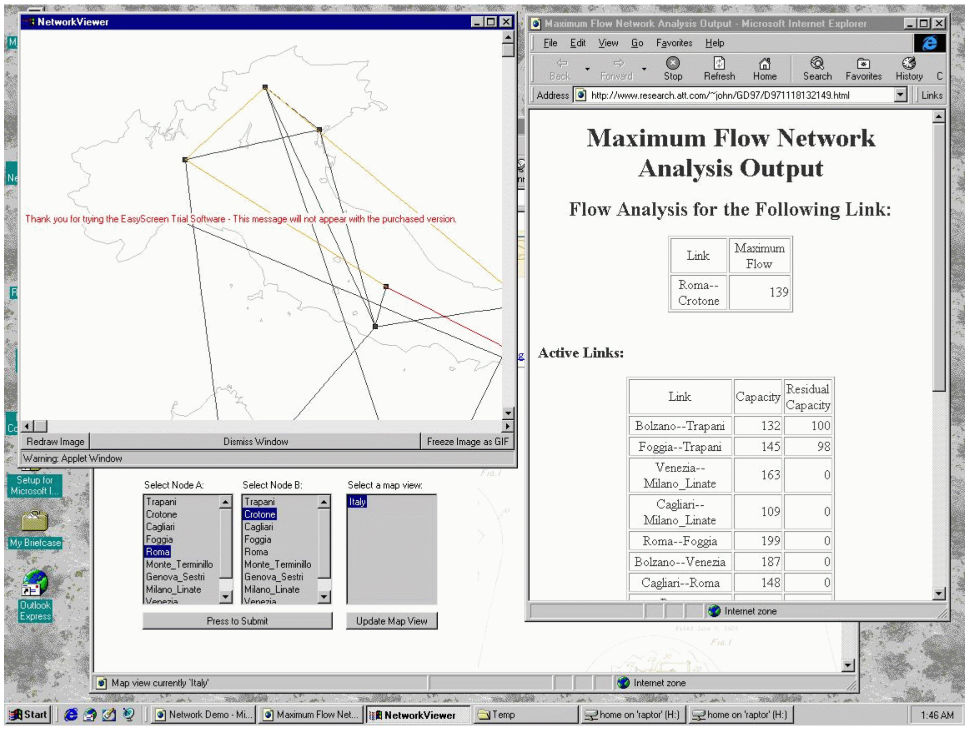
interesting notion of distributed namespaces for graph objects shared among graph editors running on separate host computers.

Grappa

Grappa[28] is a graph package written in Java. It was created to provide graphical interaction with graphs through web browsers, integrated with other Java packages. Figure 8 shows an example of a Grappa display.

Grappa has classes for graph representation, presentation, and communication with layout services (via a Dot language parser). Grappa is extendible in the sense that application-dependent classes can be defined naturally as sub-classes of Grappa object classes. Although Grappa does not share any code with the rest of our system, it adheres to Libgraph's model and file language, including Dot's conventions for graphical symbols and attributes, and so fits in well with the other graph processing tools.

Grappa's user and API interfaces are well-engineered, but Java applet performance is a serious problem on current platforms when downloading graphs that have hundreds of objects. For example, on a typical high-end personal computer or workstation, downloading and initializing the display of a telecommunications network that has about 1200 nodes and


NetworkViewer
Thank you for trying the EasyScreen Trial Software - This message will not appear with the purchased version.
Redraw Image
Dismiss Window
Freeze Image as GIF
Warning: Applet Window
Select Node A:
Select Node B:
Select a map view:
Trapani
Crotone
Cagliari
Foggia
Roma
Monte_Terminillo
Genova_Sestri
Milano_Linate
Italy
Press to Submit
Update Map View
Map view currently 'Italy'
Internet zone
Maximum Flow Network Analysis Output - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History
Address http://www.research.att.com/~john/GD97/D971118132149.html
Start
Network Demo - Mi...
Maximum Flow Net...
NetworkViewer
Temp
home on 'raptor' (H:)
1:46 AM

Maximum Flow Network Analysis Output

Flow Analysis for the Following Link:

Link	Maximum Flow
Roma--Crotone	139

Active Links:

Link	Capacity	Residual Capacity
Bolzano--Trapani	132	100
Foggia--Trapani	145	98
Venezia--Milano_Linate	163	0
Cagliari--Milano_Linate	109	0
Roma--Foggia	199	0
Bolzano--Venezia	187	0
Cagliari--Roma	148	0

Figure 8. A Grappa session

1800 edges takes about 75 seconds of wall-clock time. The overhead of initializing tens of thousands of Java objects is thought to be the bottleneck.

Montage

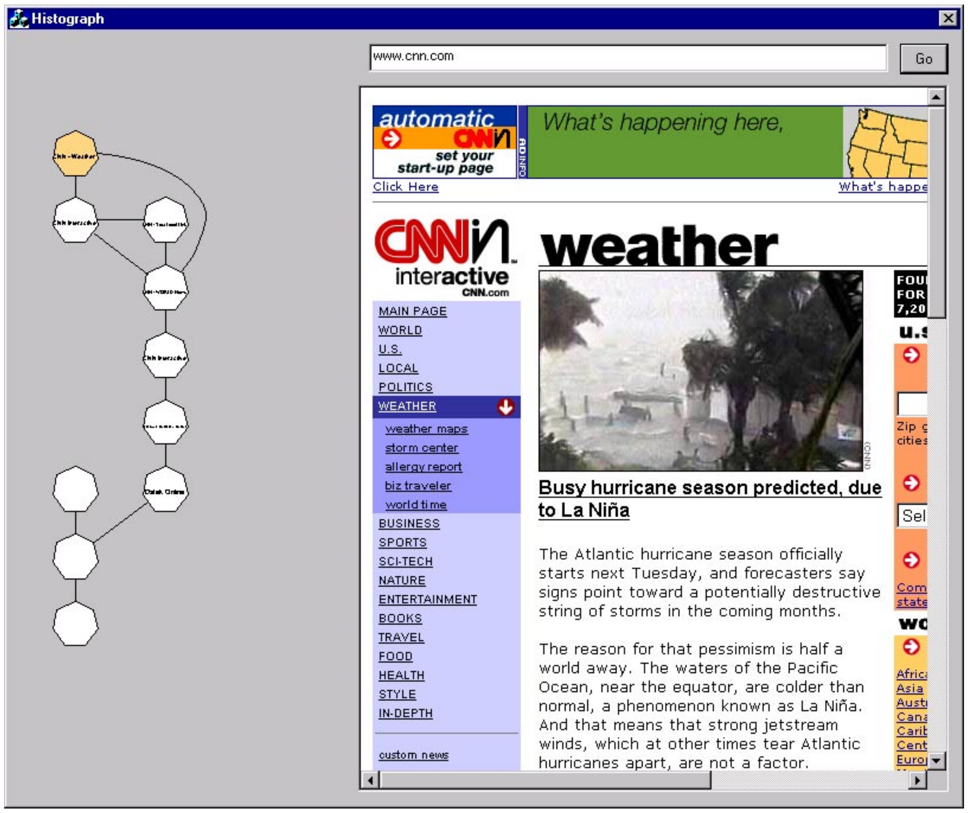
Encouraged by some success in applying graph drawings in user interfaces and software engineering documentation, we considered how to integrate graph tools with applications in Microsoft Windows. Although Lefty and Tcldot run in MS-Windows, they do not use its native graphics toolkits, nor do they communicate well with document editors such as Word. Even simple operations, such as placing a static graph layout in a document, are not intuitive.

In MS-Windows, interactive graphical programs can communicate using a protocol called ActiveX (formerly called Object Linking and Embedding, or OLE [29], that descended from a cut-and-paste facility). This protocol supports embedding of graphical clients inside containers (servers), by defining object types and messages to handle program activation, graphical events, persistence, etc.

After studying what would be needed to build an ActiveX layer for programs like Lefty, we decided instead to write a new front-end for MS-Windows, building on native graphics and inter-application communication toolkits. This front end is named Montage‡. It acts either as a client or top-level frame, and allows a dynamic layout manager to control the placement of embedded objects, that may have their own frames (such as nodes, that may also be ActiveX applications), or are frameless (as with edges in graph layouts). Montage

‡Written by Gordon Woodhull, formerly at U.C. Berkeley.

also supports application toolbars, and there is a way for external programs to access graph objects. Thus, Visual Basic programs should eventually fill the role that Lefty scripts play in Dotty. Figure 9 shows a sample session in Montage, in which a graph records the sites visited and the links traversed in a web browser. By clicking on a node, the user causes the browser to go to the corresponding site.



Busy hurricane season predicted, due to La Niña

The Atlantic hurricane season officially starts next Tuesday, and forecasters say signs point toward a potentially destructive string of storms in the coming months.

The reason for that pessimism is half a world away. The waters of the Pacific Ocean, near the equator, are colder than normal, a phenomenon known as La Niña. And that means that strong jetstream winds, which at other times tear Atlantic hurricanes apart, are not a factor.

Figure 9. A Montage Internet Explorer session

Graph Filters

The toolkit provides a collection of functions for manipulating and analyzing graphs as filters, thus providing something of an algebra of operators on graph objects. We view this as one of the main design features of our toolkit. Having a good variety of flexible filters available, the user can construct many applications in a high-level scripting language, simply combining these and other general-purpose filters.

We describe some of the more interesting filters below.

Gpr

Gpr is a graph filter, modeled after Unix's AWK and SED utilities[30, 31]. For each input graph, Gpr selects a subgraph and then generates the output graph as a function of this subgraph and the input graph.

A selection is found by iterating over all nodes and edges, testing a fixed predicate on each. Predicates are given as command-line arguments. This gives Gpr an interface that is convenient for graph-transforming commands and scripts. Also for convenience, a command line option can select the whole input graph.

Gpr's predicate evaluator has regular expression matches and arithmetic computation. It is based on Fowler's Libexpr[32] that provides these features. Primitive values in expressions may be constants (strings and numbers), graph object attributes, or one of the pseudo-attributes listed in Table III. Pseudo-attributes are actually built-in functions that refer to local syntactic properties such as graph object names or degrees. These are useful properties, but not immediately available from Libgraph's string attribute interface.

Gpr's next step generates an output graph, taking the input graph and the selected subgraph as operands. The permitted operations are:

- add induced edges to the selection
- update attributes of nodes or edges in the selection
- contract the selected subgraph into a node
- contract paths in the input graph into edges
- replace the selection with its transitive reduction

These operations are convenient for pre-processing graphs with Gpr before layout. Some examples may clarify Gpr's usage. To double the weight of all blue edges:

```
gpr -e color='blue' -i -E "weight = weight * 2"
```

To perform path compression, as when simplifying data or control flow graphs:

```
gpr -n "(indegree != 1) || (outdegree != 1)" -p
```

To find the transitive reduction of a graph:

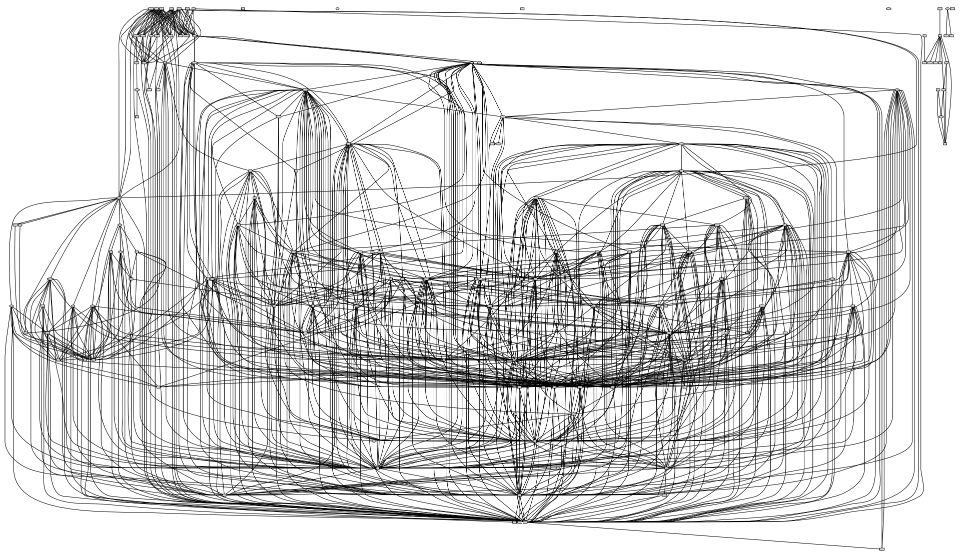
```
gpr -c -t
```

Applying transitive reduction[33] is a plausible way to attack complexity in large, dense graphs, whose layouts can be hard to read. The argument for applying this algorithm as a filter is that removing edges implied transitively by other paths does not change reachability (at least in acyclic graphs, where the transitive reduction is unique). It would be interesting to find evidence to prove or disprove whether this is an effective reduction for specific tasks involving large graphs. Figure 10 shows the effect of running this algorithm on a module dependency graph. The graph has 241 nodes and was reduced from 791 to 255 edges. Although both drawings are the same size, the latter is much cleaner and provides a much better view of the graph structure.

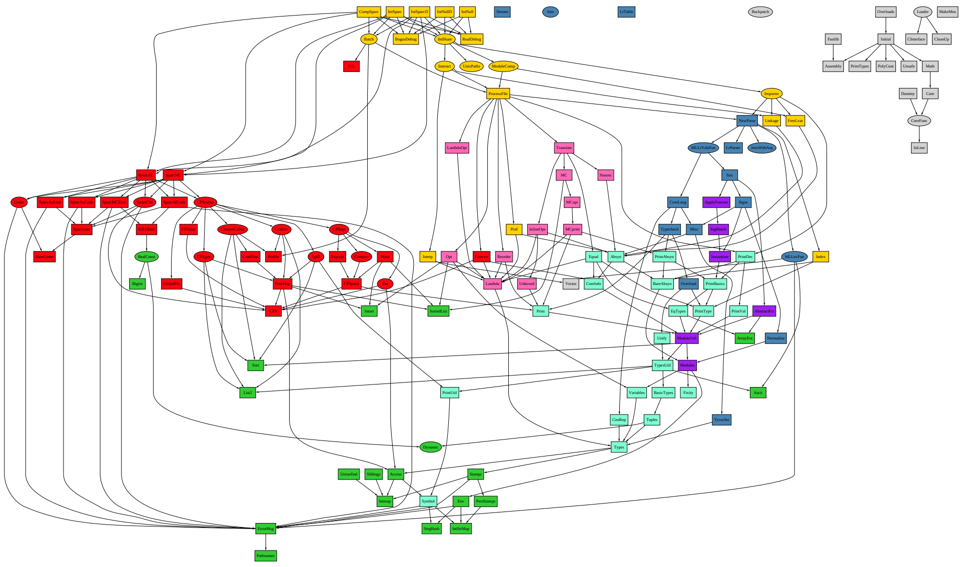
Gpr's design favors convenience (for a few operations) over flexibility. To give Gpr the power of language-based utilities like AWK and Perl, it might be changed to embody a language in which a programmer could set the sequence and composition of filtering operations. This seems to be a promising direction for further development. Another, perhaps complementary, approach is suggested by Tarjan's path expressions[34], a depth-first-search framework in which certain common graph algorithms can be succinctly defined.

Sccmap

Another approach to attacking the complexity of large graphs is to decompose them into smaller graphs. In some situations, decomposition into strongly connected components



(a)



(b)

Figure 10. `gpr -t` *applied to a module dependency graph*

(SCC's) is helpful. The Sccmap filter reads a graph stream, breaking each input graph into a list of its SCC's along with the reduced graph whose nodes correspond to the SCC's of the original graph and with the induced edge set. Figure 11 is a before-and-after example. Part (a) shows the original graph; part (b) shows the graph consisting of the four strongly connected components. For certain applications, this much smaller graph exhibits most of the essential

Table III. Pseudo-attributes

nodes		
	name	(string)
	indegree	(number)
	outdegree	(number)
edges		
	tail	(string)
	head	(string)
	tailindegree	(number)
	tailoutdegree	(number)
	headindegree	(number)
	headoutdegree	(number)

structure of the original.

Colorize

In graph layouts, graphical attributes such as node color and shape can encode relationships other than the one represented by the graph's edges, which usually determine geometric position. For example, a drawing of the call graph of a profiled program could use color to encode the time spent in each function. This typifies the use of color to get more information into a layout by encoding additional variables. Appropriate choice of graphical styles can also reinforce the primary edge relation, which also seems desirable since redundant encodings may ease some graphical tasks (cf. Earnshaw[35]). In many situations, of course, color simply makes graphical designs seem more interesting and appealing. This is not necessarily bad, but caution is needed since unnecessary complexity can conflict with accurate, informative visualization[36].

To assist in the application of color to graph visualization, Colorize is a stream tool that assigns node colors. It expects input graphs to have at least a few nodes with assigned colors. Taking these as seed values, it applies breadth-first-search to "push" colors across edges, applying them to initially uncolored nodes. Colors blend where flows meet. This technique can combine visual reinforcement of edge relations with the encoding of some other relationship through choice of seed colors. Further, Colorize can also ramp color saturation smoothly across levels in a layered graph. This way of applying colors emphasizes level assignment.

An example of Colorize output is shown in Figure 12. This is a diagram of a family of software projects, showing interdependencies. The major family divisions are architecture, software configuration management, databases and visualization, represented by the four nodes at the bottom of the graph, each with its own color. As nodes interconnect up the graph, the colors blend, giving a visual clue (even on the monochrome figure here) as to where an application fits among the family divisions.

Unflatten

The Unflatten filter adjusts a directed graph to improve its layout aspect ratio. If a graph has many leaves or disconnected nodes, the layout produced by Dot will generally be very wide or tall. Unflatten adjusts a graph to improve layout compaction. In particular, it attempts

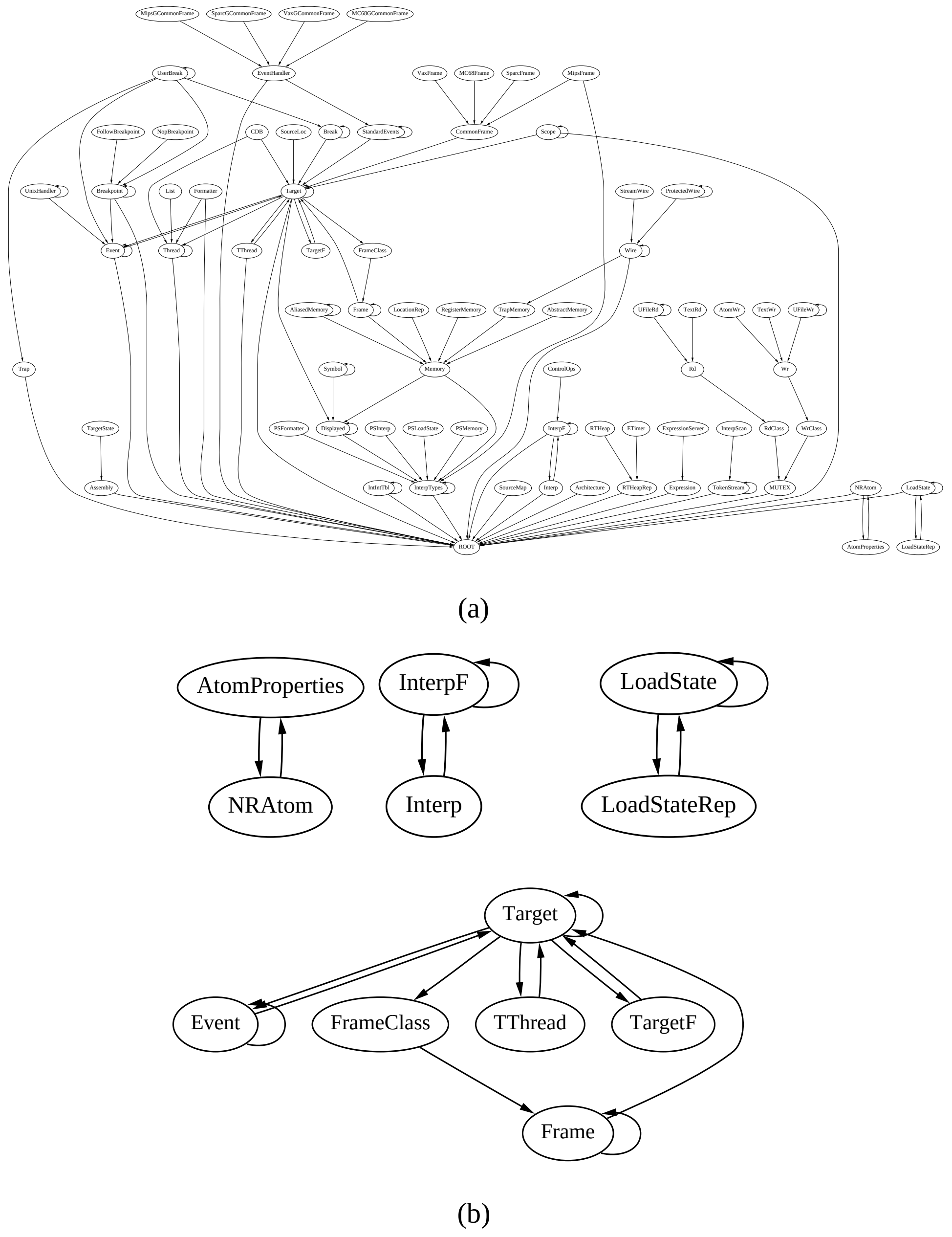


Figure 11. Strongly connected component analysis

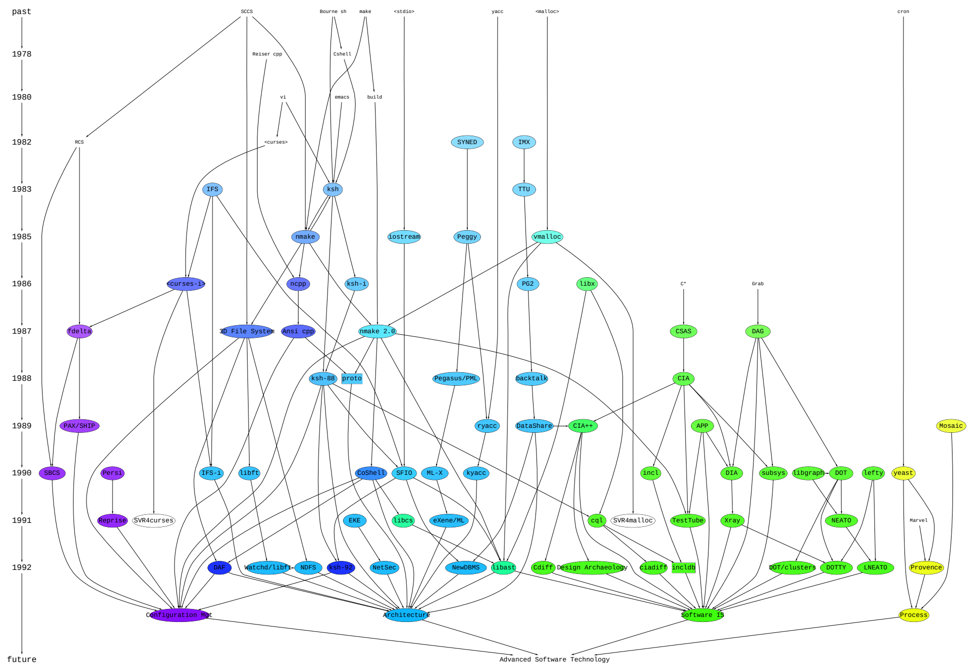


Figure 12. Colorize applied to a graph of software packages

to stagger leaf nodes and disconnected nodes by inducing (invisible) edges between them.

Experiences

Reliance on simple, concise interfaces and an extendible architecture promotes rapid development of new applications by integrating existing external components with the graph system. Prototypes can often be built in a day or two, then refined incrementally.

The range of programs and projects that have used graph filtering and visualization components tends to validate the flexibility of the system architecture and the usefulness of the individual tools. In the domain of software engineering, parts of our toolkit have been used to create tools for specification, testing, distribution, reverse engineering and software process management. The toolkit has also been used for projects in network planning and engineering, security and web page analysis.

In this section, we describe in detail a few applications built using our graph analysis and visualization components.

Structure Repositories

One framework that has proven powerful is that of a structure repository. The basic idea is that, for a given class of structured documents, e.g., program source code in a given language or web pages in HTML, one can create a model of the key structural entities, their attributes and the relations between them. One then builds a document analyzer that parses a collection of documents belonging to the given class, and generates a repository capturing the modeled components and associations.

Once the repository is built, a user can filter a repository to create derived views, to display views, and to perform queries on interesting structures within views. Although views can be presented in tabular form, one can take advantage of the duality between a relation and a graph to provide directed graph views. A graph presentation greatly helps in understanding the relationships between entities. In addition, the graph can be made an active part of the user interface, as a convenient mechanism for navigation and querying.

The relation-graph duality arises also when querying and filtering views. Often, the desired filtering is best posed as a graph manipulation. For example, to simplify a view, one may want to see the transitive reduction of the graph, removing all relations that can be derived from transitive closure. Conversely, in a repository modeling a program, one may desire to construct the transitive closure of a "uses" relation, starting from the program entry point. Anything not occurring in the closure can be considered dead code.

This framework has been implemented[37] and has been applied in many settings, as will be described below. For a given document class, someone must construct a model for the class, essentially an entity-relationship database schema, and write an analyzer that extracts the required information from a document. Our toolkit provides the core facilities for graph manipulation and display across all instances of the framework. To create a visualization subsystem tailored to a given document class, the user provides the schema, the graphical attributes (e.g., fonts, colors, shapes) of nodes and edges corresponding to entities and relationships in the schema, and the appearance and actions of the graphical interface. An instance compiler then takes these specifications and generates the complete visualization environment[§].

Software Engineering

Graphs are a ubiquitous data representation in computer science and therefore not surprisingly, in software engineering. They occur throughout software development, from requirements and specifications to testing and maintenance. They are used to specify system structure and semantics, provide a graphical view of source code structure and relations, and describe the processes, both human and machine, that compose the building and execution of software. They form the basis of the models used in most object-oriented analysis and design methodologies (cf. UML[38]).

Components of our toolkit have been used in tools supporting most aspects of software engineering. Although some of these tools have remained research prototypes, others have been used in production systems, handling hundreds of files and hundreds of thousands of lines of code.

Reverse Engineering

Reverse engineering is the task of discovering the structure and semantics of a program from implementation artifacts. Typically, artifacts include program source, and sometimes no supporting documentation. One cannot rely on the original system designers or implementors, as it is not unusual for them to be unavailable or to have forgotten their design and coding decisions.

One approach to this task is to analyze source code for certain language constructs, such as variables, functions and files that form the program, and determine the relations between

[§]The user also supplies certain database information and the instance compiler generates all of the query facilities as well.

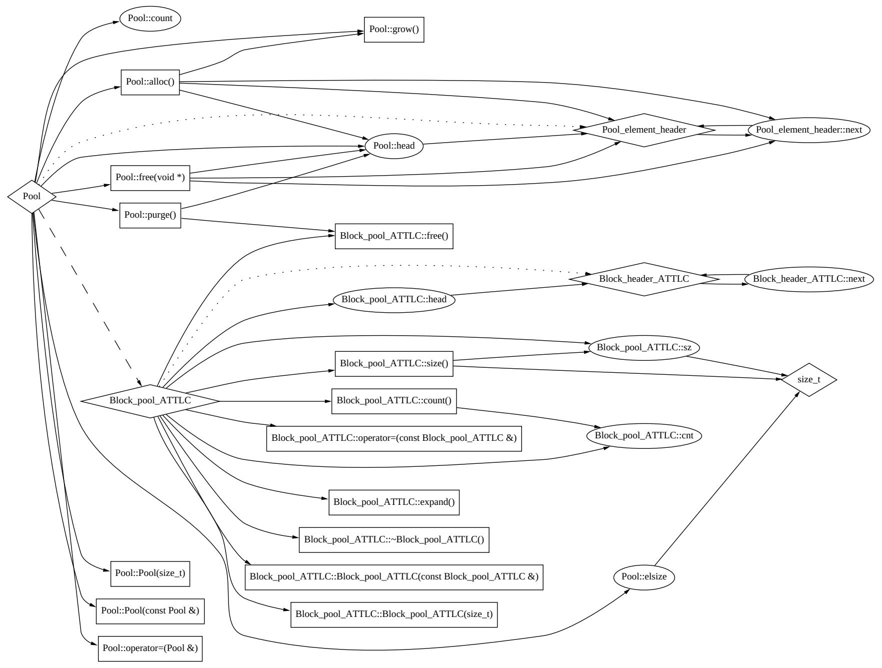


Figure 13. The reachability graph of the class `Pool`

these constructs, such as function call and data access dependency. Based on this information, one can deduce higher-level program structure and discern how the program works.

For programs and libraries written in C and C++, the Acacia[39] system provides support for this type of analysis. Acacia is an instance of the repository framework described above. It blends a special-purpose parser and semantics analyzer for C and C++, to produce the repository, with parts of the toolkit tailored to graphs of program entities. The user can employ these to explore the program structure graphs, looking at type dependencies, the include file hierarchy, or, in the case of C++, the class inheritance graph. A typical graph view from Acacia is shown in Figure 13, representing those entities reachable from a given C++ class.

Certain graph-based analyses are common and important enough that Acacia provides built-in tools for performing them. These include forward and reverse reachability analysis for determining subsystem dependencies and unnecessary include file detection.

When exploring a software system, one sometimes wants to know the set of components that some other collection of components depends on, for example, in order to decompose a system into subsystems or libraries. In particular, this information is necessary in order to extract a subsystem for separate inclusion elsewhere. Alternatively, one may need to learn what components could be affected by changing some given component. These functions are provided in Acacia as semantics-based transitive closure operators on the program graph.

A common problem in legacy C programs in the accumulation of unnecessary include files. These usually arise when code structure is altered without re-evaluating which include files are still required. They also occur from programmers naively adding include directives until compilation succeeds. This problem has only been exacerbated by C++, which requires complete prototypes for all functions used. It is not uncommon for 10-35 percent of include files to be unused.

Unnecessary include files also complicate reverse engineering, as they increase the number of components and relations, and imply associations where none actually exist beyond the inclusion. This problem can also affect the software build process. As the initial phases of compilation, especially preprocessing and lexical analysis, often predominate, removal of unneeded include files decreases build time. Again, in C++ environments that rely on nested include files, the gain is even more dramatic.

Acacia provides the necessary graph analyses[40] to partition the set of include files into three categories. An include file can be necessary for compilation; unused; or incidental, in that its immediate contents are not necessary, but it includes, directly or indirectly, a necessary include file.

Acacia is only one instantiation of the structure repository framework. Others have been built for Java[41], the Korn shell scripting language and other programming and system languages[¶].

Debugging

When looking for a bug or analyzing the dynamic behavior of code, it can be helpful to see a trace of the function calls taking place during execution. This facility is provided by Xray[42]. It instruments the code, executes it to gather a trace log, and then combines this data with the static information derived from Acacia. The execution trace can then be animated, using a function call graph view. Basic graph edges indicate that one function calls another. The color of an edge indicates the currently active functions on the call stack, plus how frequently the particular function call has been made. At one end of the spectrum, this highlights which edges are never traversed during the execution. At the other end, the system highlights edges of heavy activity, as determined by a user-supplied threshold, indicating points that might greatly benefit from further analysis and optimization. The tool has been effective in uncovering inefficiencies and exposing bugs in a variety of programs.[‖]

Specification

Graphs can also play a role at the other end of the software development process, during the requirements and specification phases. One example of this occurs in telephone switching equipment. The switching community has found it advantageous to describe the interaction of switching components using formal methods. This provides a high-level model in which the system can be analyzed and verified during analysis. One way of specifying system semantics is by a language such a LOTOS[44], which provides a process algebra of event sequences.

To assist software engineers, who are sometimes unfamiliar with formal techniques, a prototype programming environment was constructed for a subset of the LOTOS language. A system description written in LOTOS corresponds to a collection of finite-state machines. Thus, as with the structure repository, the environment presents both a text view and a graph view. The latter provides a more natural, familiar representation to an engineer of how the system will act. In addition, it can provide a view of a system simulation, in which the current state of each component is highlighted. Interestingly, this environment was written in

[¶]`http://www.research.att.com/˜ ciao/`

[‖]The recent Deet debugger[43] builds extensively on the high-level toolkit approach advocated here, and has been integrated with Dotty to display linked data structures and other program information.

SML[45], using the eXene[46] graphics library while relying on the graph toolkit for layout and other graph algorithmic support. A more closed architecture would have greatly restricted such mixing of implementation languages.

Process

Auxiliary to the main stream of software development are various activities supporting the development process itself. In these activities, graphs modeling processes provide a unifying theme. Our toolkit has been used to develop tools to model these activities.

A software process is a set of tasks, partially ordered (by time), that must be completed as part of the development of a software system. The tasks include the main development stream (from analysis and design to coding to testing) but also involve related activities such as documentation, staffing and business case analyses. The tasks may involve multiple subgroups within an organization, and may involve both humans and machines. Often, the software process reflects a chosen development methodology.

Improvise[47, 48] is a system for modeling and analyzing software processes. Since a software process is represented as a partially ordered set, the user interface to Improvise is naturally the corresponding attributed graph. A user can interact with the graph, adding annotations to the nodes and edges to describe the process or log progress. The annotations are not limited to text, but can include multimedia information (e.g., a video showing how a task should be accomplished) or executable actions. The user can also specify static and dynamic constraints on the node and edge attributes.

At the machine process level, it is useful to have a tool to monitor system process execution over multiple machines in real-time. This is provided by the VPM tool[49], which displays relevant processes, their host machines and their use of system resources. Displaying a graph, VPM uses nodes to represent processes, open files, plus special interprocess communication mechanism such as pipes and sockets. Resources restricted to a single machine are clustered together. VPM does not use a logging mechanism or specially instrumented code to gather its data, but traces system calls directly. For system debugging purposes, however, it can generate a log, which can be played back and analyzed off-line.

Other areas

Sometimes, applications for the toolkit come from unexpected sources. Ship[50] is a system for managing the distribution of (possibly mutually dependent) software components as the components change over time. For each client site, it maintains a record of the versions at that site. When a new version of some software is available, Ship determines what is the most effective way to deliver the software to a client, depending on how much software has changed and the transport mechanism (uucp, rcp, ftp, etc.) used to deliver it. Typically, this involves sending a collection of incremental updates.

Managing the software versions contained in the Ship data base is simplified by the Shipview tool[51]. This presents a collection of systems, versions and files as a directory hierarchy, with node color indicating the current status of a package version vis-a-vis its integration into the database. A user can browse the data base structure, exposing or hiding directory detail. Clicking on a file node brings up a view of the file's contents, the viewing method based on the file's type. Using parts from the toolkit, plus other readily available software tools, Shipview was built in a couple of hundred lines of code in about an hour.

Network Planning

Applications increasingly rely on web browser interfaces. An advantage is that database operations and analysis can be performed on a central server that provides consistency, reliability and security, while basic user interface functions can be performed locally to ensure good interactive response. A key architectural problem is how application-specific data is exchanged between clients and servers.

Figure 14 shows a typical display from a network design application created with stream graph tools. The initial HTML page of the application asks users to enter two network endpoints on a form; upon receipt the server invokes a network analysis utility (here, a max flow algorithm) on a network stored in the server. The result is displayed graphically in another HTML page. In this context it is clear that back-end graph processors, run from CGI programs, are always stream based.

To explore an alternative to Grappa's relatively heavyweight clients, this application's display and user interface are programmed as a concise Tcl/Tk script. The script is interpreted in the browser by a Tcl/Tk plug-in, that must be installed separately. The graph to be displayed is hard-wired into the script. In our computing environment, the plug-in interface starts up several times faster than Grappa's on networks like the one shown in the figure. Such lightweight scripts are possible because the Tcl/Tk environment provides many built-in graphics features; on the other hand more elaborate features (such as pan-and-zoom, multiple views and editable graphs) would need better data structures and substantially more code. In summary, lightweight GUI scripts have a valuable role, but a class library seems a better foundation for sophisticated user interfaces.

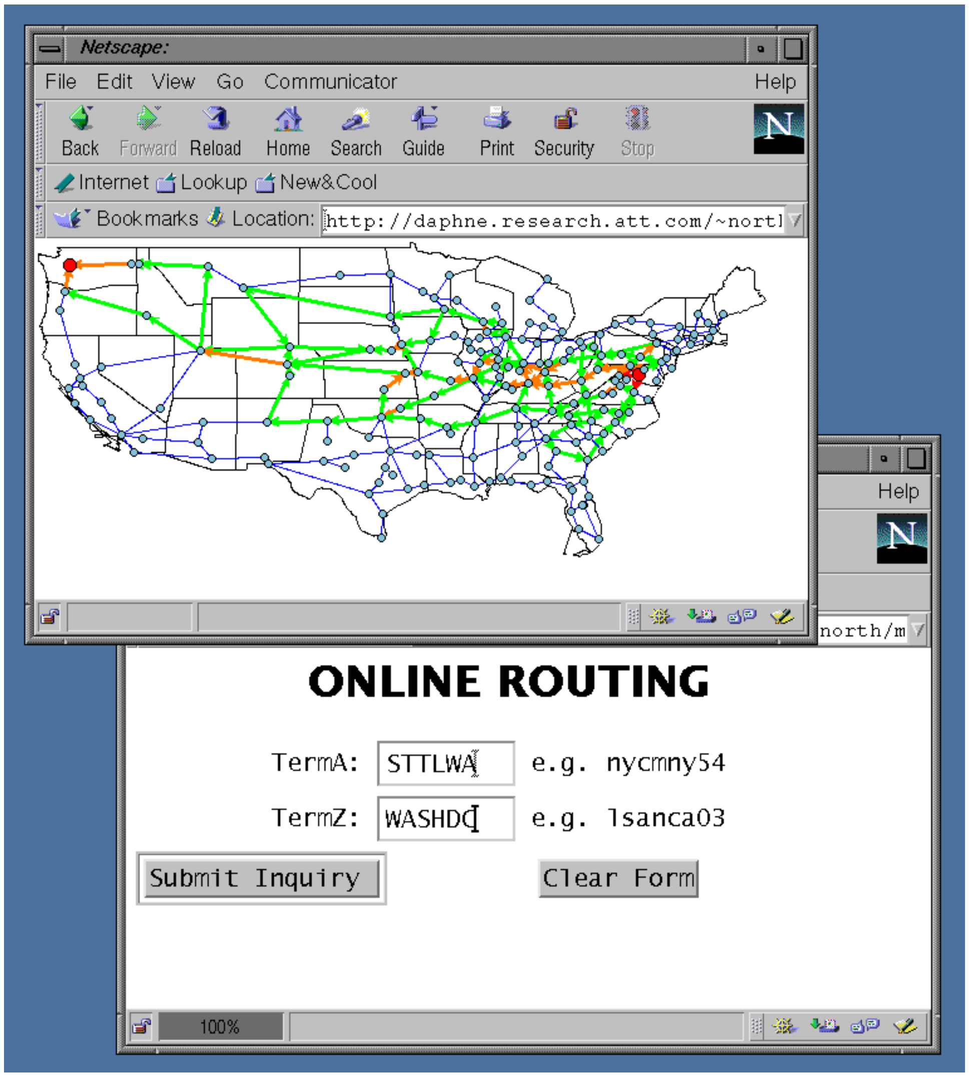
Graphing the Internet

The growth of the Internet and web pages has provided a new lode of graphs to be analyzed and displayed. These graphs arise in various contexts. For an Internet service provider, there is the obvious graph representing network and network traffic. There are also the graphs representing connections from one URL to another. These graphs can be useful in restructuring web sites to facilitate user access to important pages, for example, by providing more direct links, and to remove little used pages.

A problem associated with the rapid expansion of the web, coupled with the newness of the technology and the dynamic nature of the web, is the constant churn of web pages. Web users will visit a page one day, and when they return the next day, the page may be entirely rewritten and redesigned. One system for tracking these types of changes is WebCiao[52]. This system is another instance of the structure repository, this time for the HTML language. To capture the evolution of a web site, the system uses two analyses of the site at different times, constructs a view representing the differences between the two repositories, and presents a graph reflecting these changes. In this manner, a user can quickly spot additions, deletions and modifications of the URL links. In its WebGUIDE[53] embodiment, the system relies on an analysis component that supplies both textual and structural changes. If the user is only concerned about the static structure of a collection of web pages, WebCiao can be applied to a single repository rather than the difference of two repositories.

Availability

We have described some of the uses to which our toolkit has been applied, focusing mainly on those within our local environment. The package is freely available on the Web


Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Internet Lookup New&Cool
Bookmarks Location: http://daphne.research.att.com/~nortl
north/m
100%

ONLINE ROUTING

TermA:	STTLWA	e.g. nycmny54
TermZ:	WASHDC	e.g. lsanca03

Submit Inquiry   Clear Form

Figure 14. A network design application in Tcl/Tk

at `http://www.research.att.com/sw/tools/graphviz/`. Some of the other tools mentioned above can be found at `http://www.research.att.com/sw/tools/`.

Conclusions

We have surveyed the design of a set of compatible graph processing tools, and the toolkit's application to software engineering. The tools include filters and interactive programs that can easily be combined. They provide users with an assortment of implements for manipulating graphs, and new ones can be introduced easily. In particular, the toolkit's emphasis on filters allows many tasks to be handled by combining toolkit components using a scripting language.

Most inter-tool communication relies on a graph data language that has been implemented in several common programming environments. This has endowed the tool set with flexibility and independence from a specific language or system platform.

Based on experience with applications, it is clear that while much progress has been made in processing and visualizing graphs and networks, many interesting problems remain. From the standpoint of algorithm engineering, some relevant problems are:

- Which algorithms are the best filters for attacking the complexity of large graphs?
- How can higher-order graph objects be accommodated without compromising convenience and efficiency?
- What sort of interactive systems are best for prototyping programs that combine abstract graphs, geometric computation, graphics and external tools?

ACKNOWLEDGEMENTS

The system described in this paper is the synthesis of the work of many people. In particular, the authors would like to thank Yih-Farn Chen, Eleftherios Koutsofios, John Mocenigo and Kiem-Phong Vo of AT&T Labs - Research, and John Ellson of Lucent Corporation, for their crucial input of time, ideas and software.

REFERENCES

1. B. W. Kernighan and R. Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, N.J., 1984.
2. F. Newbery Paulish and W.F. Tichy, 'EDGE: An extendible graph editor', *Software–Practice and Experience*, **20**(S1), 1/63–S1/88 (1990).
3. M. Himsolt, 'The Graphlet system', *Proc. Symp. Graph Drawing, GD '96*, volume 1190 of *Lecture Notes in Computer Science*, Berlin, 1996, pp. 233–240. Springer-Verlag.
4. L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan, 'A browser for directed graphs', *Software–Practice and Experience*, **17**(1), 61–76 (1987).
5. M. Himsolt, 'GraphEd: An interactive graph editor', *Proc. STACS 89*, volume 349 of *Lecture Notes in Computer Science*, Berlin, 1989, pp. 532–533. Springer-Verlag. http://www.uni-passau.de/˜himsolt/-GraphEd/graphed.
6. B. Madden, P. Madden, S. Powers, and M. Himsolt, 'Portable graph layout and editing', F. Brandenburg (ed.), *Proc. Symp. Graph Drawing, GD '95*, volume 1027 of *Lecture Notes in Computer Science*, Berlin, September 1995, pp. 385–395. Springer-Verlag.
7. M. Frohlich and M. Werner, 'Demonstration of the interactive graph visualization system da Vinci', R. Tamassia and I. Tollis (eds.), *Proc. Symp. Graph Drawing, GD '94*, volume 894 of *Lecture Notes in Computer Science*, Berlin, September 1997, pp. 266–269. Springer-Verlag.
8. K. Mehlhorn and S. Näher, 'LEDA: A platform for combinatorial and geometric computing', *Comm. ACM*, **38**(1), 96–102 (1995).
9. D. E. Knuth, *The Stanford GraphBase*, Addison-Wesley, Reading, Mass., 1993.
10. J. Berry, N. Dean, M. Goldberg, G. Shannon, and S. Skienna, 'Graph drawing and manipulation with LINK', *Proc. Symp. Graph Drawing, GD '97*, September 1997.
11. D.M. Beazley, 'SWIG : An easy to use tool for integrating scripting languages with C and C++', *Proc. 4th USENIX Tcl/Tk Workshop*, 1996, pp. 129–139.
12. K.-P. Vo, 'Libcdt: A general and efficient container data type library', *Proc. Summer '97 Usenix Conf.*, 1997.
13. D. Korn and K.-P. Vo, 'Sfio: Safe/fast string/file IO', *Proc. Summer '91 Usenix Conf.*, 1991, pp. 235–256.
14. Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine, 'The generic graph component library', *Proc. OOPSLA'99*, 1999. To appear.
15. E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, 'A technique for drawing directed graphs', *IEEE Trans. Software Engineering*, **19**(3), 214–230 (1993).

16. E. R. Gansner, S. C. North, and K.-P. Vo, 'DAG - a program that draws directed graphs', *Software - Practice and Experience*, **17**(1), 1047–1062 (1988).
17. J. Warfield, 'Crossing theory and hierarchy mapping', *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-7**(7), 502–523 (1977).
18. M.J. Carpano, 'Automatic display of hierarchized graphs for computer aided decision analysis', *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-10**(11), 705–715 (1980).
19. K. Sugiyama, S. Tagawa, and M. Toda, 'Methods for visual understanding of hierarchical systems', *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-11**(2), 109–125 (1981).
20. S. C. North, 'Drawing ranked digraphs with recursive clusters', *Proc. ALCOM International Workshop PARIS 1993 on Graph Drawing and Topological Graph Algorithms*, 1993. ftp /pub/papers/compgeo/gd93-v2.tex.Z from wilma.cs.brown.edu.
21. T. Kamada and S. Kawai, 'An algorithm for drawing general undirected graphs', *Information Processing Letters*, **31**(1), 7–15 (1989).
22. J. B. Kruskal and J. B. Seery, 'Designing network diagrams', *Proceedings of the First General Conference on Social Graphics*, Washington, D.C., July 1980, pp. 22–50. U. S. Department of the Census. Bell Laboratories Technical Report No. 49.
23. E. Koutsofios and D. Dobkin, 'Lefty: A two-view editor for technical pictures', *Graphics Interface '91, Calgary, Alberta*, 1991, pp. 68–76.
24. J. K. Ousterhout, *Tcl and the Tk Toolkit*, O'Reilly and Associates, 1996.
25. B. Bederson, J. Hollan, K. Perlin, J. Meyer, D. Bacon, and G. Furnas, 'Advances in the Pad++ zoomable graphics widget', *J. Visual Languages and Computing*, **7**, 3–31 (1996).
26. J. Ellson and S. North, 'TclDG - a Tcl extension for dynamic graphs', *Proc. 4th USENIX Tcl/Tk Workshop*, 1996, pp. 37–48.
27. R. E. Wagner, R. C. Alferness, A. M. Saleh, and M. S. Goodman, 'MONET: Multiwavelength optical networking', *J. LightwaveTechnology*, **14**(6), 1349–1355 (1996).
28. W. Lee, N. Barghouti, and J. Mocenigo, 'Grappa: A graph package in Java', *Proc. Symp. Graph Drawing, GD '97*, September 1997.
29. K. Brockschmidt, *Inside OLE*, Microsoft Press, 2 edition, 1995.
30. A. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Mass., 1988.
31. L. E. McMahon, 'SED - a non-interactive text editor', *UNIX Programmer's Manual - 7th Edition*, volume 2, Bell Telephone Laboratories, Murray Hill, New Jersey, 1979.
32. G. Fowler, D. Korn, and K.-P. Vo, 'Libraries and file system architecture', in B. Krishnamurthy (ed.), *Practical Reusable UNIX Software*, John Wiley & Sons, New York, 1995, chapter 2, pp. 25–51.
33. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
34. R. E. Tarjan, 'A unified approach to path problems', *J. ACM*, **28**(3), 577–593 (1981).
35. A. MacEachren, *How Maps Work*, Guilford Press, New York, 1995.
36. J. Bertin, *Graphics and Graphic Information Processing*, de Gruyter, Berlin, 1981.
37. Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach, 'Ciao: A graphical navigator for software and document repositories', *Proc. Intl. Conf. Software Maintenance*, October 1995, pp. 66–75.
38. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.
39. Y.-F. Chen, E. R. Gansner, and E. Koutsofios, 'A C++ data model supporting reachability analysis and dead code detection', *Proc. 6th Euro. Software Engineering Conf. ESEC '97*, September 1997, pp. 414–431.
40. K.-P. Vo and Y.-F. Chen, 'Incl: A tool to analyze include files', *Summer 1992 USENIX Conference*, June 1992, pp. 199–208.
41. J. Korn, Y.-F. Chen, and E. Koutsofios, 'Reverse engineering of java applets', *Technical Report TR 98.40.1*, AT&T Laboratories, 1998.
42. Y.-F. Chen, E. Koutsofios, and D. Rosenblum, 'Intertool connections', in B. Krishnamurthy (ed.), *Practical Reusable UNIX Software*, John Wiley & Sons, New York, 1995, chapter 11, pp. 327–332.
43. J. Korn, 'Abstraction and visualization in graphical debuggers', *Ph.D. Thesis*, Department of Computer Science, Princeton University, Princeton, NJ, 1999.
44. M. Ardis, 'Lessons from using basic LOTOS', *Proc. 16th Intl. Conf. Software Engineering*, May 1994, pp. 5–14.
45. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML (revised)*, The MIT Press, Cambridge, Mass, 1997.

46. E. R. Gansner and J. H. Reppy, ‘A multi-threaded higher-order user interface toolkit’, in L. Bass and P. Dewan (eds.), *User Interface Software*, John Wiley & Sons, New York, 1993, pp. 61–80.
47. N. S. Barghouti, E. Koutsofios, and E. Cohen, ‘Improvise: Interactive, multimedia process visualization environment’, *Proc. 5th Euro. Software Engineering Conf. ESEC ’95*, September 1995, pp. 28–43.
48. N. Barghouti and B. Krishnamurthy, ‘Provence: A process visualization and enactment environment’, *Proc. 4th European Software Engineering Conf., ESEC ’93*, September 1993, pp. 451–465.
49. G. Fowler, D. Korn, E. Koutsofios, and S. North, ‘Intertool connections’, in B. Krishnamurthy (ed.), *Practical Reusable UNIX Software*, John Wiley & Sons, New York, 1995, chapter 11, pp. 299–315.
50. G. Fowler, ‘Configuration management’, in B. Krishnamurthy (ed.), *Practical Reusable UNIX Software*, John Wiley & Sons, New York, 1995, chapter 3, pp. 91–106.
51. E. Koutsofios and S. North, ‘Intertool connections’, in B. Krishnamurthy (ed.), *Practical Reusable UNIX Software*, John Wiley & Sons, New York, 1995, chapter 11, pp. 299–315.
52. Y.-F. Chen and E. Koutsofios, ‘WebCiao: A website visualization and tracking system’, *Proc. WebNet97*, October 1997.
53. F. Douglis, Y.-F. Chen T. Ball, and E. Koutsofios, ‘WebGUIDE: Querying and navigating changes in web repositories’, *Computer Networks and ISDN Systems*, **28**, 1335–1344 (1996).