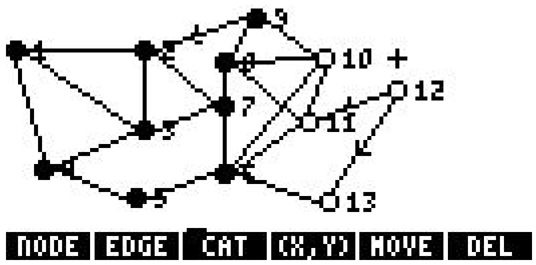
LIB 557 (22Dh) – GRAPHWRITER

INTRODUCTION



GRAPHWRITER is a full-featured graph editor for the HP 49G calculator. It allows you to create and edit a graph, represented in adjacency list form, in a graphical environment. The graph is displayed on the calculators' graphics screen (PICT), using a 2D coordinate system which can be configured by the user. There are two main entry points: GRW creates a new graph, while GRED edits the graph on stack level 1. Graphs may contain node and edge labels (as tagged values), and may have multiple edges and loops. A number of user-level operations for use in programming and on the command line are also provided.

Graph theory and algorithms play a major role in countless applications such as network design and optimisation, traffic planning, routing, scheduling and visualization, and so graphs have become one of the cornerstones of applied discrete mathematics and computer science. GRAPHWRITER makes this interesting area of mathematics come to life on your 49G calculator. To my knowledge it is the first and only graph editor available for *any* calculator.

Acknowledgements While writing this program, I have slowly become a more or less regular member of the HP 48/49 user community in the comp.sys.hp48 newsgroup, and I do not know whether I could have finished this project without the many tips and advices from knowledgeable HP users all over the world. Since the first release of GRAPHWRITER, I also received a lot of encouraging comments and suggestions for further improvements. Moreover, I'd like to thank Eduardo Kalinowski for his comprehensive book on system RPL programming, Carsten Dominik for his 49G entry point and choose engine documentation (and HP49 Emacs of course!), and John H Meyers for generously offering his help to a hopeless 49G newbie and for his never-failing sense of humour. All this helped me a lot to complete my educational journey through the strange land of system RPL programming. And of course thanks are due to Jean-Yves Avenard, Cyrille de Brebisson and all the other talented developers of HP49 fame at HP's former (alas!) Australian calculator division, ACO.

GRAPH DATA STRUCTURE

To make full use of this program, it is necessary to become familiar with the basic notions of graph theory, so let us first introduce the relevant terms and concepts. *Graphs* are abstract mathematical objects consisting of *nodes* and *edges*, with the latter connecting the former. Each edge *st* has two nodes at its ends, the *source node s* and the *target node t*, and we say that *s* and *t* are *adjacent* to each other, and that the edge *st* is *incident* to both *s* and *t*. Edges may also go from a node back to itself (i.e., $s = t$) in which case they are called *loops*. In general, a graph may have several different edges *st* between the same pair of nodes *s* and *t*. If a graph contains such multiple edges it is also called a *multigraph*; otherwise we say that the graph is *simple*.

Graphs generally come in one of two flavours, namely as *directed* and *undirected* graphs. In directed graphs (or *digraphs*, for short), edges are oriented from the source to the target node, while the edges of an undirected graph always constitute bidirectional connections, and thus an edge *st* connects both *s* to *t* and *t* to *s*. In practice, undirected edges *st* are usually implemented using pairs of matching directed edges *st* and *ts*. This is also the approach taken in the GraphWriter library. Thus an undirected graph is represented using a corresponding *bidirected* graph, which is a directed graph in which for each "out-edge" *st* there is a corresponding "in-edge" *ts*.

Graphs are usually depicted using diagrams in which nodes are drawn as points, and edges as (undirected) line segments or (directed) arrows connecting these points. This is exactly the way that graphs are displayed in GraphWriter. For this purpose, the graph is *embedded* in the plane by associating each node with a point (represented as a complex number).

The GraphWriter data structure has been designed with the goal to provide maximum flexibility with minimum overhead, keeping in mind the tight resources on a calculator. The graph is therefore stored in an (asymmetric) *adjacency list* form, i.e., for each node there is a list in which each entry specifies the target node of an outgoing edge, possibly along with other information. Thus a graph is a list of lists in the GraphWriter implementation. The adjacency list data structure provides both a compact representation and fast access to the outgoing edges of each node; incoming edges can be accessed efficiently using the graph's reversal, which can be computed using the REVGRAPH command, see PROGRAMMING. Another advantage of this data structure is that it can represent multigraphs just as well as simple graphs. Both nodes and edges can be associated with additional data, namely node and edge *labels* (besides the node embeddings we already mentioned). In the GraphWriter data structure the labels are specified as arbitrary tagged values. The syntax of the data structure is summarized in Fig. 1.

graph	::=	{ *node-info*? … }
node-info	::=	{ *point*? *label*? … *edge-info*? … }
edge-info	::=	*node* | { *node label*? … }
label	::=	tagged value
node	::=	untagged real
point	::=	untagged complex

Fig. 1 BNF-style grammar rules for the graph data structure. { … } denotes lists, ? optional elements, and | separates different alternatives.

That is, a *graph* is a (possibly empty) list of *node-info* structures, where each *node-info* is a list consisting of an optional *point* (encoded as a complex number), followed by a (possibly empty) sequence of node *labels*, followed by a (possibly empty) sequence of *edge-info* structures. An *edge-info*, in turn, is either a simple *node* index, encoded as a real value, or a list consisting of the *node* index followed by a (possibly empty) sequence of edge *labels*. Each node index must point to an existing node, i.e., it must be in the range 1..SIZE(*graph*). Finally, a *label* is simply an arbitrary tagged value.

The syntactic validity of a graph structure is always enforced when the graph is built using the GraphWriter operations. The syntax of graph objects entered directly by the user can be checked with the GRAPH? predicate, see PROGRAMMING.

Examples: {}, the empty list, also denotes the empty graph, {{}{}{}} is a graph with three nodes and no edges, and {{2. 3.} {1. 3.} {1. 2.}} is the "complete" bidirected graph with three nodes, in which all pairs of distinct nodes are connected. Node embeddings and labels are simply added at the beginning of each node list, as in {{(.5,1.) ::"ONE" 2. 3.} {(.94,.26) ::"TWO" 1. 3.} {(.06,.26) ::"THREE" 1. 2.}}, while edge labels are added at the end of an *edge-info* list, as in {{{1. ::X}}} which is a graph consisting of a single node and a loop labelled with the variable 'X'.

Labels allow you to store arbitrary data with the nodes and edges of a graph structure. This is extremely useful if a graph algorithm must work with certain node and edge "properties", such as the edge "lengths" in the shortest path problem (see the example at the end of this manual). Labels can also be used to add a short description to a node or edge, to ease identification. As indicated, labels must always be tagged values, although the tag may be empty, as in ::*value*. The tags are used to distinguish different labels, employing them as a kind of "field identifier". For this purpose, the GraphWriter library also provides the GETLABEL and PUTLABEL operations to directly access the value associated with a given tag, see PROGRAMMING.

INVOCATION

Once the GraphWriter library has been installed, its operations can be accessed as usual, either through the catalog (CAT), the library menu (LIB GRW), or by typing the command names using the alphabetic keys. If you are working a lot with this library, it is best to have the library menu (557 MENU) assigned to a key; I have it on ↰-hold-MTRW . This can be done conveniently using Wolfgang Rautenberg's Keyman program.

To start GraphWriter with a fresh and empty graph, simply execute the GRW command with no arguments. To edit an existing graph, put it into stack level 1 and invoke the GRED command. (We generally assume that the calculator is operated in RPN mode. In algebraic mode, you would have to enter the command GRED(*graph*) to start the editor.) To exit GraphWriter and place the edited graph on the stack press the ENTER key. CANCEL backs out abandoning your changes (the editor asks for confirmation when you are about to do that and the graph has been modified).

Your graph may contain unembedded nodes, i.e., nodes which do not have coordinates assigned to them; these will be embedded automatically. Currently GraphWriter uses random points for missing embeddings, which is not very pretty, but you can easily rearrange the nodes to your heart's content using the MOVE operation once the graph has been loaded into the editor. Or you can write a small program to automate this task. Of course, when the graph has gone through the editor it will always have a complete embedding attached to it.

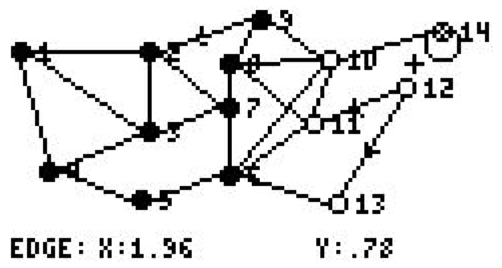
As of version 2.0, GraphWriter can show the node and edge labels in your graph, but you should note that, for the sake of efficiency and to reduce clutter on the small LCD screen, the graphical display is currently restricted to the *first* label of each node or edge (also called the *default label*), and, in the case of multigraphs, only the default label on the first edge between a given pair of nodes is displayed. Moreover, only the label values will be shown, without the tags, and label values are truncated to at most seven characters. GRAPHWRITER also allows you to turn off the node or edge label display, or both, which is useful when the screen becomes too crowded. As in earlier versions, the complete set of node and edge labels can be accessed in the *browser*, the integrated textual editor providing a choose/inform box interface.

MENU OPERATIONS

When GRAPHWRITER starts, it displays its own menu from which you can invoke the following operations:

F1 [illegible]: Enters NODE mode, in which you can place nodes at the current cursor position. ENTER places a node (unless there already is a node at the current position, in which case the editor beeps to signal an error), CANCEL exits from NODE mode.

F2 [illegible]: Enters EDGE mode, in which you can connect nodes with edges. Pressing [illegible] marks the current node (indicated by a tiny cross), after which you can create edges to other nodes by putting the cursor on the target node and pressing ENTER. ENTER creates a bidirected edge between source and target mode, while ↰ ENTER and ↱ ENTER produce directed edges going to the source or target node, respectively. New nodes are created automatically if the cursor is not over an existing node when any of these operations is executed. CANCEL exits from EDGE mode.



EDGE: X:1.96 Y:.72

We remark that EDGE mode deliberately forbids you to create a new edge on top of an existing one (the small screen and the lack of visual cues would allow this to happen much too easily by accident), so all graphs created with this operation alone will be simple. To create multigraphs you must hence use the edit operations in the browser (discussed below).

↱ F2 [illegible]: This is a variation of the EDGE command which enters so-called PATH mode. While normal EDGE mode allows you to draw edges between a single source and multiple targets, placing an edge in PATH mode causes the target node to become the new source. You can use this to produce a chain of edges (called a *path* in graph theory) very quickly.

F3 [illegible]: Invokes the “browser” choose box, which allows you to perform selections, and view, locate and edit nodes and edges in a textual format, using inform boxes for data entry. See closer description below.

F4 [illegible]: Enters CURSOR mode, in which the current cursor coordinates are shown in the menu line, as in the other modes. ENTER places the current position in the stack (above the edited graph), as a complex value. CANCEL exits CURSOR mode.

F5 [illegible]: Move either the current node or the selection to a new position. Pressing F5 [illegible] marks the current node, after which you can move the cursor to the desired target position. ENTER completes the move, CANCEL aborts it.

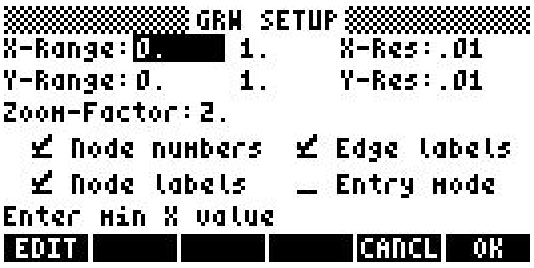
F6 DEL: Delete the current node or the selection (as well as all incident edges), after prompting for confirmation.

The remaining operations are on the second menu page. As usual, you can switch between the menu pages with the NXT and PREV keys.

F1 CLEAR: Delete the entire graph, with confirmation.

F2 SETUP: Enters the setup form, which allows you to configure the coordinate system (range of X and Y coordinates, X and Y resolution in units/pixel), the factor for zoom operations, various display options controlling the display of node numbers and labels and edge labels, and "entry mode". The latter option, when checked, causes the editor to prompt you with an input form when new nodes and edges are created. This is useful if you want to enter the node and edge labels right away when adding nodes and edges, instead of doing that later in the browser.

The settings made in SETUP are stored in the GRWPAR variable when the form is completed with F6 OK or ENTER. The editor looks for the settings in this variable at startup. If the variable is not found in the current or some parent directory, some hard-coded defaults are used, as shown in the picture on the right.



```
GRW SETUP
X-Range:0.     1.     X-Res:.01
Y-Range:0.     1.     Y-Res:.01
Zoom-Factor:2.
 ✓ Node numbers   ✓ Edge labels
 ✓ Node labels    _ Entry mode
Enter min X value
EDIT                    CANCL  OK
```

F3 PICT→: Places a copy of the current graph display on the stack.

F4 NODE→: Places the number of the node below cursor on the stack.

F5 SEL→: Places the current selection on the stack, as a list of node numbers.

F6 →SEL: Sets the selection from a node list on the stack.

On the third menu page you find the obligatory F1 HELP function which displays some help screens and copyright information.

KEYBOARD OPERATIONS

CURSOR MOVEMENT: The cursor keys ◀ ▲ ▼ ▶ move the cursor around on the screen as usual, with scrolling if necessary. The shifted cursor keys also have their usual meaning (↰: move cursor to the edges of the current window, ↱: move to the edges of the entire picture). These keys work in all modes.

ZOOMING: + zooms in, − zooms out, and the × key reverts to the default settings, as given by the GRWPAR variable. Moreover, ↰ × redraws the screen using the current display configuration, which comes in handy if the display looks garbled. The right-shifted variations of these keys can be used to zoom to various types of "full-screen" displays: ↱ − makes the largest and ↱ + the smallest of the dimensions fit into the display, while keeping the same resolution for X and Y. ↱ × makes the whole display fit on the screen (possibly using different resolutions for X and Y). These keys work in all modes.

MAKING SELECTIONS: The +/- key toggles the selection status of the node under the cursor. If no node is under the cursor, +/- deselects all nodes. Selected nodes are drawn as circles, while unselected nodes are drawn as solid bullets. Nodes can also be selected in the browser, see below.

MODE: Keyboard shortcut for the SETUP menu operation.

VAR: Toggles the node and edge label display. This key works in all modes, and is used to quickly enable or disable the label display.

CAT: Keyboard shortcut for the CAT menu operation. This key works in all modes. However, when in one of the special modes (NODE, EDGE, CURSOR, etc.), only a restricted form of the browser is available, which allows you to browse the graph and locate nodes, but not to edit the graph.

DEL / CLEAR : Keyboard shortcuts for the DEL/CLEAR menu operations.

STO▶ / RCL : STO▶ puts the subgraph “induced” by the selected nodes on the stack, i.e., the graph consisting of all selected nodes and incident edges; if no nodes are selected then the entire graph is copied. RCL is used to retrieve a graph from the stack and place it to the right and below the current cursor position; it also sets the selection to the newly created nodes. These operations provide a convenient means to copy and paste subgraphs between different graphs.

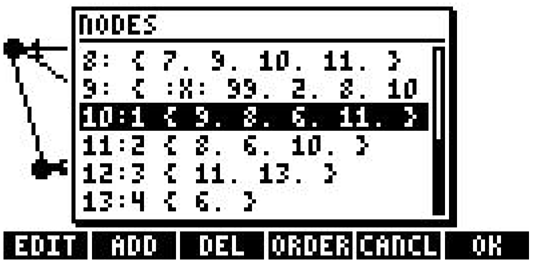
HIST: Visit the stack. This allows you to inspect, rearrange and edit the objects in the stack (excluding the edited graph) while the editor is running, which is useful in conjunction with STO▶, RCL and the other stack commands in the second page of the main menu, like F5 [illegible] and F6 [illegible]. If the stack is currently empty, the editor beeps, indicating an error condition.

THE BROWSER

The editor also provides a choose/inform box interface in which you can view and edit the graph. This environment, called the *browser*, is started with the CAT menu command. The browser can also be invoked from the special modes (NODE, EDGE, CURSOR, etc.) using the CAT key. However, in these modes the browser only provides a restricted set of operations (no selection and editing).

NODE BROWSER

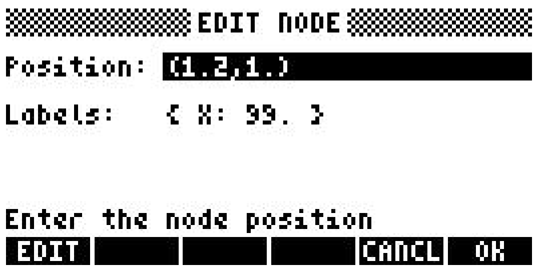
When started, the browser first shows a choose box with all nodes of the graph (with the current node selected, if any). This part of the browser is also called the *node browser*, in which you can choose a node and perform any of the following operations:



```
NODES
8:  { 7.  9.  10.  11. }
9:  { :X: 99.  2.  8.  10
10:1 { 9.  8.  6.  11. }
11:2 { 8.  6.  10. }
12:3 { 11.  13. }
13:4 { 6. }
```

F1 EDIT: Edit the position and labels of the current node. The labels are entered as a list of tagged values. You can also enter a single (non-list) value, which will be converted to a label list automatically (in this case a null tag will be added if the tag is missing).

F2 ADD: Create a new node.



```
EDIT NODE
Position: (1.2,1.)
Labels:   { X: 99. }

Enter the node position
EDIT                CANCL  OK
```

F3 DEL: Delete the current node or the selection (with confirmation).

F4 ORDER: Reorder the nodes according to the current selection order. This works analogous to the 49G filer, i.e., the selected nodes are placed first, in the order in which they were selected, followed by the unselected ones. The browser shows the current selection order by numbering the selected nodes accordingly.

F5 CANCL / CANCEL : Exit the browser without changing the cursor position.

F6 OK / ENTER : exit the browser, and put the cursor on the selected node.

▶ : Bring up another choose box for the edges of the selected node (see the discussion of the *edge browser* below).

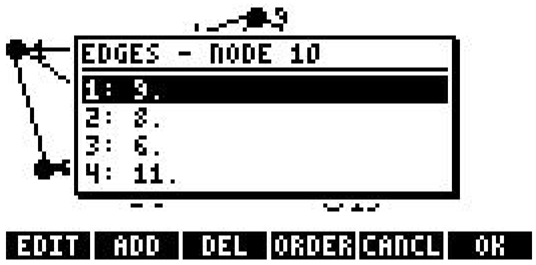
+/- : Select/deselect the current node.

STO▶ : Put the current node (i.e., its *node-info* structure) on the stack.

HIST : Visit the stack.

EDGE BROWSER

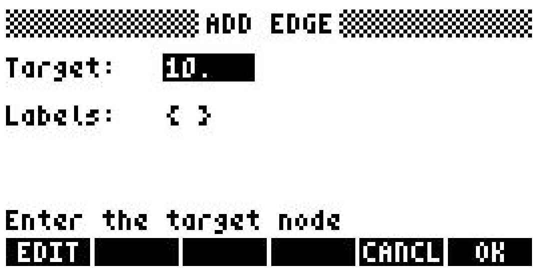
Just like the node browser, the edge browser allows you to edit or delete existing and create new entries, and change the order in which the entries are listed. CANCL / CANCEL and OK / ENTER also work as in the node browser, but OK sends the cursor to the *target* node of an edge. The edge browser provides the following operations:



```
EDGES - NODE 10
1: 9.
2: 8.
3: 6.
4: 11.
EDIT  ADD  DEL  ORDER CANCL  OK
```

F1 EDIT: edit the target node and labels of the current edge.

F2 ADD: create a new edge. With this operation you can also create multiple edges to the same target node, which is not possible with the main menu's EDGE operation.



```
ADD EDGE
Target:   10.
Labels:   { }

Enter the target node
EDIT                CANCL  OK
```

F3 DEL: delete the current edge or the selection (with confirmation).

F4 ORDER: reorder the edges according to the current selection order.

F5 CANCL / CANCEL : exit the browser without changing the cursor position.

F6 OK / ENTER : exit the browser, and put the cursor on the target node of the selected edge.

◀ : Return to the source node of the current edge in the node browser.

▶ : Show the target node of the current edge in the node browser.

+/- : Select/deselect the current edge.

STO▶ : Put the current edge (i.e., its *edge-info* structure) on the stack.

HIST : Visit the stack.

By default, the EDIT/ADD/DEL operations of the edge browser will operate on bidirected edges, meaning that the editor will create or update both the outgoing source-target edge and its reversal (i.e., the corresponding incoming target-source edge). More precisely, the ADD operation will create edges in pairs of matching in- and out-edges, while the EDIT and DEL operations modify and delete both the current out-edge and its reversal (if any).

(We must clarify what is meant by "the" reversal of an edge here, which is not obvious in the case of a multigraph; for instance, consider the case that there is a single out-edge *st* and two matching in-edges *ts*. GraphWriter adopts the convention that in- and out-edges must not necessarily be in 1-1 correspondence, so any matching in-edge will do. In the current implementation the editor simply picks the first matching in-edge, i.e., the first edge going from target to source which has the same label list as the out-edge. If you want to enforce that in- and out-edges *are* in 1-1 correspondence, you can always achieve this by adding suitable edge labels.)

If you want to create or modify only the outgoing edge, use the right-shifted softkeys instead. Thus, ↱ F2 ADD creates a directed edge going out of the current node, and ↱ F1 EDIT / ↱ F3 DEL only changes/deletes the outgoing edge but not its reversal.

PROGRAMMING

The GRAPHWRITER library also provides the following commands for creating, inspecting and manipulating a graph from the command line or through a RPL program. We give the stack diagram of each operation together with a short description. For a definition of the various types of parameters please refer to Fig. 1 in GRAPH DATA STRUCTURE. In addition, *nodes*, *labels* and *edge-infos* are used to denote (possibly empty) lists of node indices, labels and edge-info structures, respectively.

For the sake of efficiency, most operations perform syntax checking of the graph data structure in a "lazy" manner. This means that a syntactic error in the graph data structure will generally not be caught until an operation actually inspects that part of the data structure, in which case a #203 "Bad Argument Value" error is generated. You can also verify the syntactic validity of a graph beforehand using the GRAPH? predicate.

You will find that in the library menu the operations have mostly been arranged into groups of related operations on different menu pages, following the categories listed below.

SYNTAX PREDICATES

The following predicates are used to check the syntax of graphs and their components.

GRAPH? *graph* → 0 / 1

Check whether *graph* forms a valid graph structure.

NODE? *node-info* → 0 / 1

Check whether *node-info* forms a valid node structure.

EDGE? *edge-info* → 0 / 1

Check whether *edge-info* forms a valid edge structure.

GRAPH CONSTRUCTION

These operations are for generating graphs with a given number of nodes. In the following, *n* denotes a positive integer, and *p* a probability value ($0 \le p \le 1$).

NGRAPH *n → graph*

Creates an edgeless graph with *n* nodes.

RANGRAPH *n p → graph*

Creates a random simple digraph with *n* nodes and edge probability *p*.

RANGRAPH2 *n p → graph*

Creates a random simple bidirected graph with *n* nodes and edge probability *p*.

OPERATIONS ON NODES AND EDGES

These operations allow you to create, delete and modify nodes and edges in a graph.

GETNODE *graph node → node-info*

Returns the *node-info* structure for the given node.

PUTNODE *graph node node-info → graph′*

Replaces the *node-info* structure of the given node. The *node-info* structure must be in the format described in GRAPH DATA STRUCTURE.

ADDNODE *graph node-info → graph′*

Adds a new node to the graph. The *node-info* structure must be in the format described in GRAPH DATA STRUCTURE. The new node is added at the end of the graph, so its number will be SIZE(*graph*)+1. If any edges are specified, their target nodes must exist in the resulting graph, i.e., they must be in the range from 1 to SIZE(*graph*)+1.

ADDNODE2 *graph node-info* → *graph′*

Like ADDNODE, but also creates the reversals of the given edges. This is to be used when a bidirected graph is to be constructed.

DELNODE *graph node* → *graph′*

Deletes the given node and all incident edges. The remaining nodes are renumbered accordingly.

ADDEDGE *graph node edge-info* → *graph′*

Creates a directed edge emanating from *node* with the target and labels specified in *edge-info*, which must be in the format described in GRAPH DATA STRUCTURE. The edge will be added at the end of the current list of edges for *node*.

ADDEDGE2 *graph node edge-info* → *graph′*

Creates a bidirected edge (given edge plus its reversal, if not a loop).

DELEDGE *graph* $node_1$ $node_2$ → *graph′*

Deletes the (directed) edge from $node_1$ to $node_2$. If more than one edge exists between the given nodes, the first such edge is deleted.

DELEDGE2 *graph* $node_1$ $node_2$ → *graph′*

Deletes an edge and its reversal (if present).

GENERAL GRAPH OPERATIONS

Some general operations for graph manipulation.

NODES *graph* → *nodes*

Returns the list of all nodes of the graph, i.e., { 1 … SIZE(*graph*) }.

REVGRAPH *graph* → *graph′*

Reverses a graph (makes the source of each edge its target and vice versa).

SORTGRAPH *graph* → *graph*´

Sorts a graph. The edges of each node are sorted in ascending order with respect to the target node numbers. This operation does stable sorting, hence edges with the same target node will retain their relative positions.

SUBGRAPH *graph nodes* → *graph*´

Computes the subgraph induced by the given nodes. The nodes will be numbered in the order in which they are listed in the *nodes* list (which must not contain duplicates). Hence this operation can also be used to reorder the nodes of a graph.

ADDGRAPH graph_1 graph_2 → *graph*

Disjoint union of two graphs. Adds graph_2 to graph_1, renumbering the nodes of graph_2 accordingly. The resulting graph has SIZE(graph_1) + SIZE(graph_2) nodes.

JOINGRAPH graph_1 graph_2 → *graph*

Joins two graphs. Merges graph_2 into graph_1 by combining the labels and edges of corresponding nodes in both graphs. The resulting graph has MAX(SIZE(graph_1), SIZE(graph_2)) nodes.

OPERATIONS ON NODE AND EDGE STRUCTURES

The following operations are used to create node and edge structures and break up existing structures into their component values. No syntax checking is performed on the structure or component values, but you can check the structures explicitly with the NODE? and EDGE? predicates (see above).

NODE→ *node-info* → *point labels edge-infos*

Breaks up a node structure into its components.

→NODE *point labels edge-infos* → *node-info*

Creates a node structure from its components.

EDGE→ *edge-info* → *node labels*

Breaks up an edge structure into its components.

→EDGE *node labels* → *edge-info*

Creates an edge structure from its components.

NODE AND EDGE ATTRIBUTES

These operation provide quick access to the various node and edge attributes.

ADJ *node-info* → *nodes*

Returns the list of adjacent nodes (EDGES with labels stripped).

ADJ? *node-info node* → 0 / 1

Check whether there is an edge to the given node.

EDGES *node-info* → *edge-infos*

Returns the list of all edges.

POINT *node-info* → *point*

Returns the embedding of a node (NOVAL if none).

TARGET *edge-info* → *node*

Returns the target node of an edge.

The following operations work on both node and edge structures. The *tag* argument may be specified either as an identifier or a string.

LABELS *node-info* → *labels*
edge-info → *labels*

Returns the labels of a node or edge.

GETLABEL *node-info tag* → *value*
edge-info tag → *value*

Returns the value of the first label with the given tag (NOVAL if not found).

PUTLABEL *node-info tag value* → *node-info′*
edge-info tag value → *edge-info′*

Sets the value of the first label with the given tag.

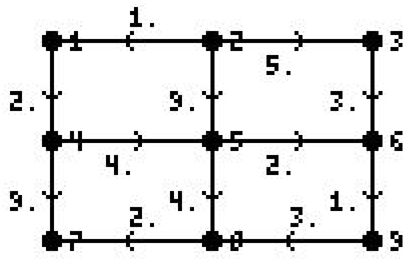
DELLABEL *node-info tag* → *node-info′*
edge-info tag → *edge-info′*

Deletes all labels with the given tag.

EXAMPLE – DIJKSTRA'S ALGORITHM

The program given below is a straightforward implementation of Dijkstra's classical shortest path algorithm in RPL, using the data structure and some of the operations of the GRAPHWRITER library. The present implementation is not fully optimised in that it repeatedly scans the entire "queue" (which is simply a RPL list) of unprocessed nodes for the node which currently is at the smallest distance from the source node. A more sophisticated version might use a kind of "heap" data structure for speeding up this part of the algorithm. A second table data structure (again implemented as a list) records the current distances from the source node, together with the predecessor nodes on a shortest path. This table thus encodes the "shortest path tree". (For an explanation of these concepts, please refer to the description of Dijkstra's algorithm in any good textbook on graph or network algorithms.)

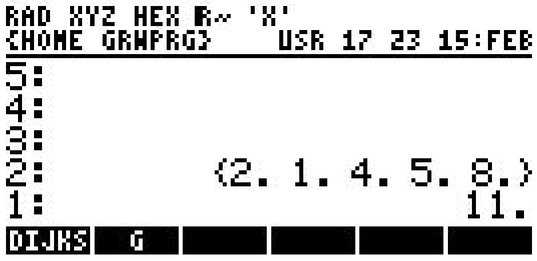
Note that this algorithm only works correctly if all edge lengths are nonnegative. Edge lengths are encoded as :L: labels on the edges of the graph, and may be infinite (these edges are effectively treated as nonexistent) or missing (in which case 1 is assumed as the default length). As implemented, the algorithm searches for a single path from a given source to a given target node, but the modifications for computing shortest paths to *all* nodes of the graph in a single run are fairly straightforward, and are left as an exercise to the interested reader.



You can find the RPL source of the algorithm in the GRAPHWRITER package, along with the sample graph shown on the right, which is ready to be used with the program. Having transferred the program and the graph to your calculator, you can give it a try. For instance, what is the shortest path from node 2 to node 8?

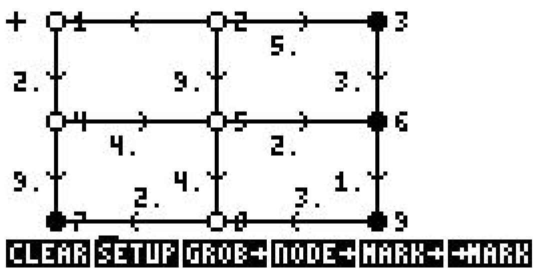
To solve this problem, first recall the contents of the graph variable 'G', enter the source and target node indices, and finally execute the DIJKSTRA program. Assuming that the program and the graph can be accessed from the VAR menu using, say, the F1 and F2 keys, we can type:

F2 2 ENTER 8 ENTER F1



```
RAD XYZ HEX R~ 'X'
{HOME GRWPRG}    USR 17 23 15:FEB
5:
4:
3:
2:            {2. 1. 4. 5. 8.}
1:                         11.
DIJKS  G
```

The program now starts searching for the shortest path, and displays the nodes considered during the search in the top display line. Finally, the result {2. 1. 4. 5. 8.} will be displayed along with the length of the path, 11.



Now is the time to take the editor for a spin and have a look at the solution! Press ◀ once to delete the path length value, F2 to recall the graph to stack level 1, and invoke the GRED command. In the editor you can highlight the nodes on the path we just computed simply by recalling the selection from the stack with the NXT F6 menu function.

RPL Program DIJKSTRA

```
« → G s t «

  s 0. RND 's' STO t 0. RND 't' STO
  @ if you like, check parameters here...

  @ list of nodes yet to be processed
  G NODES

  @ initialize the queue of {dist node pred} triples
  DUP 1. « ∞ SWAP 0. 3. →LIST » DOLIST
  s 0. s 0. 3. →LIST PUT     @ set source node distance to 0

  @ initialize the shortest path tree (table of {pred dist} pairs, a
  @ zero pred value indicates that the node has not been reached yet)
  { 0. 0. } G SIZE NDUPN →LIST

  s                          @ initial closest node

  → U Q SP pos «

    WHILE pos 0. ≠ REPEAT

      @ get the closest node and remove it from the queue and the node list
      Q pos GET
      Q 1. pos 1. - SUB Q pos 1. + Q SIZE SUB + 'Q' STO
      U 1. pos 1. - SUB U pos 1. + U SIZE SUB + 'U' STO
      OBJ→ DROP → dist node pred «

        @ give some feedback while we're searching
        "Searching... " node + "↵" + 1. DISP

        @ update the shortest path tree
        'SP' node pred dist 2. →LIST PUT

        IF node t == THEN
          @ we reached the target, set pos to 0. to indicate that we're
          @ done
          0. 'pos' STO
        ELSE
          @ iterate over all edges of the current node
          G node GETNODE EDGES
          IF DUP {} == THEN DROP         @ empty edge list
          ELSE
            1. «
              DUP TARGET DUP U SWAP POS
              IF DUP 0. == THEN
                DROP DROP2               @ skip this target
              ELSE
                @ stack now: edge target target-pos
                ROT 'L' GETLABEL         @ edge length
                EVAL                     @ handle algebraics/infty
                IF DUP NOVAL == THEN
                  DROP 1.                @ default value
                END
                IF DUP ∞ < THEN          @ finite-length edge?
                  dist +                 @ source-dist + edge length
                  Q PICK3 GET HEAD       @ current target-dist value
                  → target pos newlen oldlen «
                    IF newlen oldlen < THEN
                      @ update queue entry
                      'Q' pos newlen target node 3. →LIST PUT
                    END @ newlen < oldlen?
                  »
                ELSE DROP DROP2
                END @ edge length < ∞?
              END @ target node not in U?
            » DOLIST
          END @ empty edge list?
          @ locate the next-closest node in the queue
          0. 'pos' STO ∞ → min «
            Q 1. «
              IF HEAD DUP min < THEN
                'min' STO NSUB 'pos' STO
              ELSE DROP
              END
            » DOSUBS
          »
        END @ at target node?
      »

    END @ while pos<>0

    @ construct the path from the shortest path tree; we start at the
    @ target node and travel back along the predecessor links in the
    @ SP tree

    t SP OVER GET OBJ→ DROP 1. → pred len count «
      WHILE pred 0. > REPEAT
        pred 1. 'count' STO+
        SP pred GET OBJ→ DROP2 'pred' STO
      END
      IF DUP s == THEN
        count →LIST REVLIST len
      ELSE
        count DROPN ∞
      END
    »

  »
»
»
```