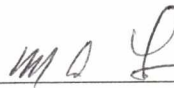


To the Graduate Council:

I am submitting herewith a thesis written by Carolyn M. Aebischer entitled "Algorithms and Interface Enhancements for GraphEd". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



Dr. Michael Langston, Major Professor

We have read this thesis
and recommend its acceptance:



Brad Vander Zanden

Accepted for the Council:



Associate Vice Chancellor
and Dean of the Graduate School

ALGORITHMS AND INTERFACE ENHANCEMENTS FOR GraphEd

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Carolyn M. Aebischer

December, 1996

Acknowledgments

I would like to thank my advisor, Dr. Michael Langston, for making sure a wayward graduate student finally finished. Your help and encouragement is appreciated! Thanks to Dr. Bradley Vander Zanden and Dr. David Straight for being on my committee. Thanks to Dr. Langston's research group, especially Kevin Hart, Barbara Plaut, and G. Rajeev, for their patience and invaluable input into this project.

I would also like to thank the formidable group of Sharon Lewis, Carianne Meystrik, and Ember Constantin who gave me both encouragement and a mild beating when I needed it. Thanks to Chris Jepeway, Chris Meystrik, Tim Martin, Michelle Baldwin, Reed Wade, and Jan Jones. Thanks to my parents for their support and humor through all of this. And special thanks to Robert Manchek for all of his help, love, and patience!

Abstract

Graph editors can be a powerful visualization tool for research. A good editor allows the user to create, display, and manipulate graphs. Researchers demand many different things in such a tool. It may be difficult to find an implementation that is flexible enough and simple enough to assist in the research process.

Over the past year I have worked with graph algorithm researchers to see what features would make a graph editor most useful to them in their work. The researchers chose an existing graph editor package called GraphEd to experiment with. GraphEd was developed at the University of Passau, Germany, by Michael Himsolt. They liked some of its features, but came up with a wish list that would make it more beneficial to their work. This wish list included more algorithm modules and a more flexible interface that would allow the researcher to integrate his/her own algorithms with a minimum of reprogramming.

This paper describes these enhancements, and indicates some possible directions for future work.

Contents

1	Introduction	1
1.1	About GraphEd	2
1.2	About Sgraph	3
1.3	Project Enhancements	4
2	Application Modules for GraphEd	5
2.1	Series-Parallel Graph Recognition	6
2.2	Chordal Graph Recognition	10
2.3	Outerplanar Graph Recognition	16
2.4	Pathwidth	21
3	User Enhancements	27
3.1	Color Selection	28
3.2	Format Converters	30
3.3	Random Graph Generators	36

3.4	Extended User Interface	38
4	Conclusions and Future Work	43
4.1	Future Application Module Additions	44
4.2	Future Interface Enhancements	45
	Bibliography	47
	Appendices	51
A	Application Module Output Examples	52
B	User Manual for Interface Extensions	57
	Vita	62

List of Tables

3.1	GraphEd color index	29
B.1	GraphEd color index	59

List of Figures

3.1	DIMACS format example	31
3.2	GraphEd format example	33
A.1	Series-Parallel Graph Recognition Output	53
A.2	Chordal Graph Recognition Output	54
A.3	Outerplanar Graph Recognition Output	55
A.4	Pathwidth Output	56
B.1	Sample .graphed_user_menu file	60

-

Chapter 1

Introduction

This thesis describes research done on a graph editing package called GraphEd. The purpose of the research was to enhance the package to make it more useful and more effective for other researchers. After experimenting with GraphEd and learning its full capabilities, I began to meet with a group of graph algorithm researchers, Barbara Plaut and G. Rajeev, who had originally selected this package for possible use. They liked certain features of the package and saw it was moving in the right direction. But they felt it could be enhanced in ways that would make it more useful for the work they were doing [BGLR96, LP95, FL88]. They drew up a wish list that became the basis of my research. The paper will discuss the changes that were made, particularly in the areas of application modules and user interface enhancements, and will also explore enhancement possibilities for the future.

1.1 About GraphEd

GraphEd was developed at the University of Passau, Germany, by Michael Himsolt. Some packages concentrate on graph editing capabilities; others focus on graph algorithm implementations. GraphEd is an attempt to combine both of these functions into one comprehensive tool which consists of a graph editing interface, application modules, and an application interface[Himb]. Examples pictures of this tool can be seen in Appendix A.

As described in the GraphEd User Manual[Hima], the graph editing interface includes both “create” and “edit” modes, as well as support for graph grammars, which allow certain kinds of graphs to be produced more efficiently. In create mode, the user can set a number of edge and node attributes, and then create a graph in the drawing window. The graph can then be saved or any necessary modifications can be made using the basic editing functions provided.

Once a set of graphs has been created, the user can choose from a large selection of graph application modules including connectivity, planarity, and triangulation algorithms. Output from these modules can be displayed in a variety of ways by changing node and edge attributes (labels and colors) and by sending messages to the console window.

Additional tools can be added to any menu using the GraphEd application interface which includes the Sgraph, Slist and Attributes data structures. These

structures allow the user to develop new modules and integrate them into GraphEd. These structures will be discussed further in Section 1.2.

1.2 About Sgraph

As described earlier, the application interface supplies data structures that allow the user to develop and add application modules to the GraphEd interface. Sgraph is the primary data structure and is described by M. Himsolt in the Sgraph 3.1 Programming Manual[Himc]. The manual defines a graph as a list of nodes using the data structures Sgraph, Snode, and Sedge. In a directed graph, each node maintains a list of in-going edges and a list of out-going edges. In an undirected graph, each node maintains only one list of edges and an edge is included in the list for both end nodes.

The graph structure the GraphEd application uses is more complex than the Sgraph structure because it must include additional information such as node and edge coordinates necessary for drawing the graph on the canvas. The application interface makes sure the two structures are kept synchronized. When a user chooses one of the application modules, the internal GraphEd data structure for the selected graph is copied to the Sgraph data structure. As the application is executed, changes are made only to the Sgraph structure. When the application is completed or whenever a request to redraw the graph is made, the interface will

determine what changes have been created in the Sgraph structure and transfer these changes to the internal structure. Then the graph can be updated. Since this can be a costly procedure, it is wise to update only when needed.

1.3 Project Enhancements

As discussed in the introduction, the graph algorithm researchers drew up a wish list of improvements to GraphEd that addressed a wide variety of issues. First on their list was the addition of several graph algorithms. Most of these were graph recognition algorithms; one was a pathwidth algorithm. Color was another important consideration for them. Currently GraphEd appears only in black and white. Although nodes and edges could be highlighted in black, the researchers really wanted to be able to change the colors of nodes and edges for enhanced visualization. Other interface modifications requested by the group included the ability to display graphs in other formats and to generate random graphs without having to draw them. They also wanted a more flexible user interface that would allow them to integrate algorithms they had written without using the native Sgraph data structures. The following chapters will discuss how each of these issues was addressed.

Chapter 2

Application Modules for GraphEd

As discussed in the introduction, Sgraph is a large graph data structure that can be used in implementing graph algorithm modules for GraphEd [Himb]. The GraphEd package includes several of these algorithms. This paper discusses four more algorithms that were specified by the research group I was working with: series-parallel graph recognition, chordal graph recognition, outerplanar graph recognition, and pathwidth of a tree. They were developed using Sgraph and were added to the user menu under “carolynizations.” Included in the Sgraph data structure are node structures, node list structures, edge structures, node and edge attribute structures and many other structures needed to represent graphs. When a change is made to the Sgraph graph structure during the execution of

an application module, these changes are copied back to the graph data structure used by GraphEd.

To implement these four application modules, I have used several of the internal functions defined in the GraphEd and Sgraph sources to manipulate, alter, and step through the Sgraph data structures. For the node data structures, `set_nodelabel()`, `set_node()`, and `for_all_nodes()` are the most frequently used functions: `set_nodelabel()` is used to add or change the label on a node, `set_node()` is used to modify any other node attributes, and `for_all_nodes()` is a macro that can be used to step through each node structure in a graph. For edge data structures, there are the comparable functions: `set_edgelist()`, `set_edge()`, and `for_edgelist()`. The latter is a macro used to step through all edge data structures associated with a particular node. Other internal functions were also used as needed. Examples of output from these four algorithms can be seen in Appendix A.

2.1 Series-Parallel Graph Recognition

The first application module that was added was a series-parallel graph recognition algorithm. A series-parallel graph is an undirected graph that contains no subgraph homeomorphic to K_4 . A K_4 is a complete graph of four vertices in which all pairs of the four vertices are adjacent to each other [ST92]. In order for

a subgraph to be homeomorphic to a K_4 graph, a K_4 graph must be obtainable from the subgraph through any series of the following actions: “replacement of a dangling edge uv with u , replacement of two series edges uv and vw with uw , replacement of a loop vv with v , and replacement of two parallel edges uv and $u'v'$ with uv [GL80, p.597]. If this can be done with any subgraph, then the original graph is not series-parallel.

To determine if a graph is series-parallel, I have implemented an algorithm developed by Liu and Geldmacher that searches a graph to determine if there is a subgraph homeomorphic to K_4 [GL80]. If a K_4 is discovered, a message is written to the console indicating that the graph is not series-parallel, the four nodes of the K_4 are highlighted in red, and the unique paths connecting the pairs of nodes in the K_4 are highlighted. If no K_4 is discovered, the graph is left unchanged and a statement to the console window reports the graph to be series-parallel.

The algorithm consists primarily of two depth-first searches. The first search prepares the graph for the second search; the second search does the work of finding a K_4 . The first search transforms the original graph from an undirected graph to a directed graph with forward and back edge sets, but before this can be done several initializations must take place. First the number of nodes in the graph must be determined. Using the `for_all_nodes()` macro, a running count can be kept of the total number of nodes. It is also necessary throughout the algorithm to maintain two graph orderings: GraphEd’s internal ordering, which must be

used to reference the internal data structure of the node, and the numbering that indicates the order in which the nodes are examined in the depth-first search. It is this latter ordering that will be used in the execution of the algorithm. An array must be initialized to map each internal node number to its depth-first node number. Finally, an adjacency structure must be initialized. During the first depth-first search, this adjacency structure will be built for use in the second part of the algorithm. Calculations will be made to determine the order of nodes on the adjacency lists.

In the initial depth-first search, each node is marked as "visited", and then given an ordering. The unvisited nodes adjacent to this node will then be recursively examined. During this recursive step the adjacency list for a node, which will be used in the second part of the algorithm, is constructed. In doing this, each node's LOW value must be determined. The LOW value for a node i is the lowest ordered target node reached by an edge in the subtree rooted at i . Target nodes are added to a source node's adjacency list in increasing order of the value of ϕ which is a function of this LOW value. If the target node was visited in the search after the source node (a forward edge), then the value of ϕ , is twice the LOW value of the target minus one. If the target node had been visited before the source node (a back edge), then the value of ϕ is twice the order number of the target node. This forward and back edge information is also stored in the adjacency list records. With the adjacency lists constructed, the graph is ready

for the second depth-first search.

The second depth-first search finds a K_4 if one exists by following the depth-first path until a back edge is traversed. A stack is maintained to store these paths. If a path is generated and there are no other paths on the stack, the path is simply pushed onto the stack. If there is already a path on the stack, the newly generated path from node s to node f and the top path on the stack from node s_t to node f_t are compared. If the end points of the paths alternate in number, that is if $s > s_t > f > f_t$ or $s_t > s > f_t > f$, a K_4 has been found and the graph is reported as non-series-parallel. If the search is completed and no alternating paths have been found, there is no K_4 and the graph is reported as series-parallel.

When a K_4 is found, information for the graphics display must be sent back to the GraphEd repainting functions. A flag is set to indicate that a group of nodes must be highlighted. This group must be presented in an Slist structure and consists of the four end points of the alternating paths. In addition to this, the edges of the last generated path as well as the edges of the paths still on the stack are colored red. These edges make up the six independent routes connecting the pairs of nodes in the K_4 . This proved to be the most difficult part of displaying the results. Ideally each of the six distinct paths should be displayed in a different color. But isolating these distinct paths out of the collection of edges proved to be a difficult puzzle. Attempts to use greedy algorithms to determine these paths were unsuccessful. Paths could be found between the first few pairs, but if

these were chosen incorrectly the needed edge segments were no longer available to connect the last pairs. For now all the edges are highlighted in the same color and the distinct paths can be found by hand. Future work would concentrate on constructing an algorithm to determine these paths.

2.2 Chordal Graph Recognition

The second application module that was added is a chordal graph recognition algorithm. A chordal graph is a graph in which each cycle of length four or more has an edge connecting any two non-consecutive vertices in the cycle. The recognition algorithm I have implemented was developed by Tarjan and Yannakakis[TY84], which is a simplified version of an algorithm developed earlier by Rose, Tarjan, and Lueker [RTL76].

Consider a graph G with vertices V and edges E . This graph has a total ordering, α . The **fill-in**, $F(\alpha)$ of this ordering is defined as follows:

$$F(\alpha) = \{\{v,w\} \mid \{v,w\} \notin E \text{ and there is a path from } v \text{ to } w \text{ containing only } v, w, \text{ and vertices ordered before both } v \text{ and } w\} \text{ [TY84, p.567]}$$

This ordering is considered to be **zero fill-in** if the fill-in set is empty. A graph G is chordal if it has an ordering that is zero fill-in. So, to test if a graph is chordal there are two steps that must be taken. First, a total ordering must be computed. Second, the fill-in of this ordering must be computed. If this set is found to be

empty, then G is chordal.

To satisfy the first step of this test, the total ordering must have the following property:

If $u <_{\alpha} v <_{\alpha} w$, $\{u,w\} \in E$, and $\{v,w\} \notin E$, then there is a vertex x such that $v <_{\alpha} x$, $\{v,x\} \in E$ and $\{u,x\} \notin E$. [TY84, p.568]

A maximum cardinality search produces an ordering which satisfies this property. In this search, vertices are ordered from n (the number of vertices) down to one. The next vertex in an ordering is chosen on the basis of the number of previously ordered vertices it is adjacent to.

Once an ordering has been computed, the fill-in can be determined for this ordering. There are two approaches to this second step. The entire fill-in can be computed and then evaluated, or the algorithm can be slightly adjusted to test directly for zero fill-in. The second approach will run faster since it can stop as soon as it has found a fill-in edge. Since we only need a zero fill-in test to check for chordality, the second option was chosen. If a fill-in edge is detected, the algorithm stops and the graph is reported to be nonchordal. But if the entire algorithm completes without a fill-in being detected, the graph is reported to be chordal.

The only other definitions needed to understand this part of the algorithm are those of the **follower** of a vertex, and the **index** of a vertex. Suppose we have a

graph G with the elimination graph of $G(\alpha)=(V,E \cup F(\alpha))$. The **follower** of a vertex w in this graph is the lowest ordered vertex v that is adjacent to w and that has a numbered ordering that is greater than the vertex w . To test for zero fill-in, the follower must be computed for every vertex. The **index** of a vertex v is the maximum of the set of v and all vertices adjacent to v that have already been processed through the algorithm.

As a vertex w is processed through the zero fill-in test, each adjacent vertex v ordered earlier than w is found. If the follower of this vertex v has been computed and the index of this follower is less than the index of w , then we have found a path from w to the follower of v that only includes these two vertices and vertices numbered lower than these two vertices. If there is not already an edge between w and the follower of v , a fill-in edge has been detected and the algorithm can stop.

Discussion of the implementation of these steps must begin with the maximum cardinality search. Before the search can start, several arrays must be initialized. The array **vset** consists of doubly linked lists. Each **vset(i)** is a doubly linked list containing all unnumbered vertices that are adjacent to i numbered vertices. Initially, all of these lists are set to NULL except for **vset(0)** which contains all of the vertices. The arrays **alpha** and **alpha_inv** are both arrays of integers. The array **alpha** maps the internal node number to the ordering number in the search while **alpha_inv** maps the order back to the internal identity number.

To begin the search, a loop is started that will number n nodes from n down to one. At each iteration, a node is chosen from the largest indexed linked list that is not NULL. The node is removed from the beginning of this linked list and given an ordering, which is recorded in both the **alpha** and **alpha_inv** arrays. Before the next iteration can begin, the linked lists must be readjusted. Every vertex that was adjacent to the vertex that was just ordered must be removed from its linked list in **vset(i)** and added back to the linked list in **vset(i+1)**. The order in which vertices are chosen from a particular **vset** list is not important, so they can be removed and added in an arbitrary way. In this case, nodes are simply added to and deleted from the beginning of a linked list.

After the ordering has been completed, initializations can be made for the zero fill-in test. An array of integers is allocated for both array **f** and array **indx**. These will hold the followers of each vertex and the index of each vertex, respectively. In addition, an array is initialized that will hold the numbers of all the vertices in the chordless cycle if the graph is found to be nonchordal. At this point, a loop is started that will cycle through the ordered nodes searching for a fill-in edge. At the i th iteration the i -ordered node **w** is retrieved from **alpha_inv**. To initialize, the follower of **w** is set to itself and the index of **w** is set to i . Next, the loop goes through each node **v** that is adjacent to but ordered before **w**. For each of these vertices, the new index and follower of **v** is computed. After this is completed, the loop of vertices adjacent to **w** is traversed again. If a vertex **v** is found that has an

`index(follower(v))` less than the current value of `i`, a fill-in edge has been found and the search for the chordless cycle begins. If the entire loop of `i` is completed with no fill-in edges found, a message is printed to the GraphEd console that the graph is chordal.

In the event that the graph is not chordal, there is a complex process for finding the chordless cycle. When a fill-in edge is found, you then have two vertices on the chordless cycle. The challenge is to find the rest of the vertices so that the entire cycle can be highlighted on the graph. After many trials and errors, it was decided to use two iterations of a shortest-path algorithm. The algorithm used was Floyd's algorithm [BB88]. The idea is to create a matrix **D** which gives the length of the shortest path between any two nodes. **D** is allocated as an array of integer arrays. To initialize this matrix, each pair of nodes (i,j) is given the value zero if i is equal to j , one if (i,j) is an edge, or infinity if (i,j) is not an edge in the graph. In addition, a path matrix **P** is maintained. When the algorithm has completed, this path matrix will identify the intermediate node k on the shortest path between two nodes. If this value is zero, there is a direct path from i to j . If not zero, the intermediate node on the path from i to k and from k to j must be found. This must continue recursively until the entire path can be pieced together. **P** will also be allocated as an array of integer arrays, and all values will be initialized to zero. Both of these initializations are also done for a second set of matrices **D2** and **P2**, which will be used for the second iteration of the shortest-

path search.

The first shortest-path procedure is executed to find the first section of the chordless cycle. The central idea behind this procedure is that if node k is found along the shortest path from i to j , then the paths from i to k and from k to j must be the shortest paths as well. To find these paths, it is necessary to execute a loop of n iterations. At each iteration k of the loop, every possible pair of nodes (i,j) must be checked to see if there is a path from i to j , and passing through k , that is shorter than the length of the path currently stored in the matrix. When all iterations have been completed the optimal path length for every pair of nodes will have been found. Three nested loops representing all possibilities for i , j , and k are used for this procedure. In the heart of the three loops, there is a check to see if $D[i][k]+D[k][j]$ is less than the current value of $D[i][j]$. If so, $D[i][j]$ is set to the value of $D[i][k]+D[k][j]$ and the path matrix $P[i][j]$ is set to k .

Once these iterations have been completed, the shortest path between the two nodes of the fill-in edge can be determined. The recursive procedure `find_mid()` takes two nodes (i,j) as arguments and finds the intermediate node k on the path between them by retrieving this value from the P matrix. If there is an intermediate node, `find_mid()` then calls itself once for the node values of (i,k) , then again for the node values of (k,j) . This is done until the entire path between i and j has been reconstructed.

To find the other half of the chordless cycle, it is necessary to run the shortest-

path algorithm again. This time the second set of matrices, **D2** and **P2**, are used. Since there is no point in finding the same path again, each node that was selected in the first path was marked as unusable during the **find_mid()** procedure. The algorithm is run precisely as it was the first time, but an extra check must be made to be sure that a path being examined does not have an unusable node as an intermediate node. When this iteration has been completed, **find_mid()** is used once again to determine the path of the rest of the cycle.

In the final step, a flag is set to indicate there is a group of nodes to be highlighted, an Slist is constructed from the nodes found in the two shortest-path iterations, and the graph is repainted to reflect the changes.

2.3 Outerplanar Graph Recognition

The third application module I have added is an outerplanar graph recognition algorithm. The implementation is based on an algorithm developed by Sandra Mitchell [Mit79]. An outerplanar graph "is a graph which can be embedded in a plane in such a way that all vertices lie on the exterior face" [Mit79]. If adding an edge between two vertices that are not adjacent makes the graph non-outerplanar, then the graph is maximal outerplanar.

The algorithm I implemented is for recognizing outerplanar graphs but the properties of maximal outerplanar graphs must be understood as well to be able

to understand the algorithm. A maximal outerplanar graph, G , with M vertices must have exactly $2M-3$ edges. It also must have at least two vertices of degree two, and if one of these vertices of degree two is removed, the resulting graph must also be maximal outerplanar. An outerplanar graph, G , with M vertices can have no more than $2M-3$ edges but, like a maximal outerplanar graph, it must have at least two vertices of degree two. An outerplanar graph is simply a polygon with chords, and since it is also planar, these chords break the polygon up into smaller polygons, which can then each be triangulated. Lemma 4 in Mitchell states:

Lemma 4. A graph G is outerplanar if and only if it can be transformed to a maximal outerplanar graph by triangulation. [Mit79, p.231]

Finally, any edge of an outerplanar or maximal outerplanar graph may not lie on more than two triangles.

The algorithm described in the paper was originally developed to test for maximal outerplanar graphs, but if the differences between maximal outerplanar and outerplanar graphs are known, it is easy to modify the algorithm to test for simple outerplanarity. It is a straightforward algorithm and begins by testing the first two properties of a maximal outerplanar graph. It first tests to make sure that there are $2M-3$ edges. In the outerplanar algorithm this test makes sure that there are no more than $2M-3$ edges. Next, a list is built containing all vertices in the graph that have degree two. For both maximal outerplanar and outerplanar tests, the

length of this list must be at least two. The remainder of the algorithm proceeds by iteratively selecting a node from the list of degree-two vertices and removing it and its edges from the graph. Every time a node is removed from the graph, the list of degree-two vertices must be adjusted to reflect the new graph. If the length of the list is ever shorter than two, the algorithm stops because the graph is no longer outerplanar. If this length is not less than two, then the algorithm proceeds and two nodes that were adjacent to the deleted node are added to a list PAIRS. If the graph is maximal outerplanar, the polygon is triangulated and there should be an edge between this pair of nodes. At the end of all iterations, there should not be any pairs in the PAIRS list that are not also on the edges list. If there are, then the graph is not maximal outerplanar.

To make this algorithm check for outerplanarity only, a few small changes had to be made. Recall that a graph that is outerplanar can be triangulated to become maximal outerplanar. The first change, as mentioned earlier, is to check that the number of edges is no more than $2M-3$ instead of exactly that number. Next, each time a pair of nodes is added to the list of pairs it is recorded as a triangulation edge and this edge is added to the adjacency list of each node in the pair. Finally, if one of these triangulation edges joins a deleted node to one of its adjacent nodes and this edge was not found in the original graph, the edge is added to the list of graph edges. All other details remain the same.

The implementation of this algorithm is straightforward, but it involves a

lot of list traversals. Several initializations must be made before the body of the algorithm can be executed: the number of nodes must be determined using the `for_all_nodes()` macro, and many data structures must be initialized. The `edglist` and `pairs` lists will be doubly linked lists. An item in the `edglist` list will contain the source and target node of an edge. An item in the `pairs` list will contain the two nodes that were adjacent to the deleted node as well as a count of the number of nodes adjacent to this pair. This will help check for the requirement that no edge can lie on more than two triangles. There will be a `deg2list` that is an array of node structures containing all nodes of degree two and a `deg2index` that will keep a running count of the number of degree-two vertices. Finally, an adjacency list must be built. Although this information is available from the Sgraph data structures, it is necessary to construct a new list that can be modified to remove edges for the test or add triangulation edges. The adjacency list is an array of structures. Each of these structures contains a node `i`, the current degree of the node, and a pointer to a double linked list of node items. Each of these node items contains a node `j` adjacent to node `i` as well as a flag indicating whether or not the edge from `i` to `j` was added to the original graph as a triangulation edge. The `for_edgelist()` macro is used to initialize this adjacency list, the `edglist`, and the `deg2list`.

Once all of these initializations have been completed, the work of the algorithm can begin. The easy checks are done first. If there are more than $2M-3$ edges or

if there are not at least two degree-two vertices, an error message is printed in the console and the algorithm exits. Otherwise the iterative loop proceeds. At each iteration, a node **i** is chosen from the list of degree-two nodes. Next, the adjacency list for this node is traversed to get two adjacent nodes, **near** and **next**. These two nodes are then added to **pairs**. If either **(i,near)** or **(i,next)** was a previously added triangulation edge, the pair **(near,next)** is also added to **edglist**. Adjacency lists are updated to include the edge **(near,next)**. The node **i** and its edges are then removed from the adjacency list and **deg2list** and **deg2index** are updated to reflect the new graph. If there are still at least two degree-two vertices left, the next iteration can be started. When all iterations have been completed, there must be a comparison of **edglist** and **pairs**. A pointer is set at the head of the pairs list and edges list. For each member of the pairs list, the entire edges list is traversed to make sure there is a match. If there is a match, the comparison can proceed with the next item in the pairs list, but if there is not a match, or if the pair being matched has a count greater than two (meaning that the pair lies on the edge of more than two triangles), then an error message is printed to the console and the algorithm terminates. But if all matches are made and there are no items in the pairs list that are not in the edges list, the graph is outerplanar and all of the nodes of the plane are highlighted.

2.4 Pathwidth

The final application module I have implemented for GraphEd is a pathwidth test for trees. The algorithm I used for this was developed by P. Scheffler [Sch90]; however, the author states at the end of the paper that similar results had already been obtained by Ellis, Sudborough, and Turner in their research of the vertex separation of graphs [EST87].

First, definitions of pathwidth and path decomposition must be presented.

Def. 1: [Sch90, p.613]

A path-decomposition of a graph G is a sequence of subsets of the node set $\ell = (X_1, \dots, X_r)$ such, that holds

- 1) The $\cup_{i=1}^r X_i = V(G)$.
- 2) For any edge $(u, v) \in E(G)$ of the graph there is a set $X_i (1 \leq i \leq r)$ containing both ends of the edge ($\{u, v\} \subseteq X_i$).
- 3) For all triple $1 \leq i < j < k \leq r$ holds $X_i \cap X_k \subseteq X_j$.

The width of a path-decomposition is $p(G, \ell) := \max_{1 \leq i \leq r} |X_i| - 1$.

The pathwidth of a graph is $p(G) := \min_{\ell} p(G, \ell)$.

The algorithm for finding the pathwidth of a tree is based on a theorem proved in the Scheffler paper:

Theorem 4: [Sch90, p.614]

For any integer $p \geq 1$ and any tree T holds:

$p(T) \geq p + 1$ iff there exists a node $t \in V(T)$ with at least

three branches $T_{tv}, T_{tu}, \text{ and } T_{tw}$ such that

$p(T_{tv}) \geq p, p(T_{tu}) \geq p \text{ and } p(T_{tw}) \geq p.$

The central function of the algorithm is **merge**, a recursive procedure which determines the pathwidth of a tree given the pathwidth of its two subtrees. But more information is needed before this pathwidth can be calculated. The path-decomposition vector, containing three elements, must also be determined to get the complete picture. The first element of the vector $p_1(t)$ denotes the pathwidth of a subtree of T rooted at t . The value of $p_2(t)$ is zero if there is a path-decomposition where t is a member of the first set X_1 ; otherwise, the value is one. Finally, $p_3(t)$ will hold the number of branches of t that have maximal pathwidth. This vector is initialized to $(0,1,2)$. The leaves will retain this value. The parents of the leaves will always have the value $(1,0,0)$. For the rest of the nodes, the vector must be calculated using the rules specified in Lemma 2 of the Scheffler paper.

Lemma 2: [Sch90, p.617]

Let T^t be an oriented tree with root t and with the path-decomposition-vector $\bar{p} = (p_1, p_2, p_3)$. Then holds:

(i) If $p_3 = 0$, then also $p_2 = 0$ and T^t has a path-

decomposition of width p_1 with $t \in X_i$ for all i .

(ii) If $p_2 = 0$, then there is no node v in the tree T^t

possessing two sons of pathwidth $p_1(s) = p_1$.

(iii) If $p_3 = 2$, then $p_2 = 1$.

(iv) If $p_2 = 1$, then there is exactly one node v in the

tree T^t possessing two sons of pathwidth p_1 , all other

nodes have at most one such son.

The path-decomposition vector $\mathbf{n}(t)$ for a tree $\mathbf{T}$ can then be determined given the path-decomposition vectors of subtrees T_t and T_s . These subtrees are joined by the edge (s,t) . The order in which these merges will take place is driven by the **minpathwidth** procedure.

To implement this procedure, **minpathwidth** is called at the start, and its first job is to make all initializations. The total number of nodes is determined using **for_all_nodes()**, and an array that will hold path-decomposition vectors for every node is allocated. In addition, an integer array is initialized that will hold the number of the father(parent) of each node and an array is allocated that will serve as the queue for nodes waiting to be evaluated.

Since finding the vector values of the leaves is trivial, they will be dealt with first. To find a leaf, **for_edgelist()** is used on each node. If there is only one edge associated with a node, then it is a leaf node. If it is a leaf node, it is first

labeled as having been examined. The position in the array **father** for that node is then set to the only node adjacent to it. The path-decomposition vector for the parent of the leaf node is set to $(1,0,0)$. $p_1(t)$ would be one since the pathwidth at the leaf node is obviously one, $p_2(t)$ would be zero since the leaf would be a member of the first set of the path decomposition, and $p_3(t)$ would be zero since there are no branches off of the parent that have maximal pathwidth. If, at this point, the parent of the leaf has only one unlabeled adjacent node (its parent), then the subtree rooted at the parent node is ready to be merged and is put in the queue.

To evaluate the inner nodes, a loop is begun that will continue as long as there are still unexamined nodes. In each loop a node s is removed from the queue array. It is marked as examined and its parent t is determined from the array **father**. The **merge** function is then called to merge the path-decomposition vector of the child s with the current path-decomposition vector of the parent t . Once the merge has been completed, if all subtrees of t have been examined and $p_1(t)$ is the pathwidth of the whole subtree at t , then the parent of t can be found and t is put in the queue. When all iterations have been completed, the path-decomposition vector of the root node should contain the pathwidth of the tree.

The function **merge** simply follows the rules in Lemma 3, as described in Scheffler, which give the path-decomposition vectors of the merged trees rooted at s and t given three possible cases: either the pathwidth of s is greater than

that of t , the pathwidths are equal, or the pathwidth of t is greater than that of

s :

Lemma3: [Sch90, p.617]

Let the tree T be obtained from the subtrees T^t and T^s joined by the edge (s,t) . Let $\bar{p}(t)$ and $\bar{p}(s)$ be the path-decomposition-vectors of the two subtrees. Then the new path- decomposition-vector $\bar{n}(t)$ of the whole tree T is as follows:

1.case: $p_1(s) > p_1(t)$

$$1a) p_2(s) = 0 \implies \bar{n}(t) := (p_1(s), 0, 1)$$

$$1b) p_3(s) = 2 \implies \bar{n}(t) := (p_1(s), 1, 1)$$

$$1c) p_2(s) = p_3(s) = 1 \implies$$

$$\bar{n}(t) := \begin{cases} (p_1(s) + 1, 0, 0) , & \text{if } p_1(T \setminus T^v) = p_1(s) \\ \bar{p}(s), & \text{otherwise} \end{cases}$$

2.case: $p_1(s) = p_1(t)$

$$2a) p_2(t) = 1 \text{ or } p_2(t) = 1 \implies \bar{n}(t) := (p_1(s) + 1, 0, 0)$$

$$2b) p_2(s) = p_2(t) = 0 \text{ and } p_3(t) = 0 \implies \bar{n} := (p_1(s), 0, 1)$$

$$2c) p_2(s) = p_2(t) = 0 \text{ and } p_3(t) = 1 \implies \bar{n} := (p_1(s), 1, 2)$$

3.case: $p_1(s) < p_1(t)$

$$3a) p_2(t) = 0 \implies \bar{n}(t) := \bar{p}(t)$$

$$3b) p_3(t) = 2 \implies \bar{n}(t) := \bar{p}(t)$$

$$3c) p_2(t) = p_3(t) = 1 \implies$$

$$\bar{n}(t) := \begin{cases} (p_1(t) + 1, 0, 0) , & \text{if } p_1(T \setminus T^v) = p_1(t) \\ \bar{p}(t), & \text{otherwise} \end{cases}$$

To display the results, the value of the pathwidth is printed in the console window and the root node of the tree is highlighted.

Chapter 3

User Enhancements

The second half of this project dealt with making experimental changes to GraphEd's user interface. Some of the enhancements were visual. Red highlighting and the ability to change the colors of nodes and edges made graph algorithm results easier to see. The rest of the enhancements made the interface more flexible for the researchers so that the algorithms and graphs they were already working with could easily be folded into the GraphEd environment. Some of these changes worked well and some may need a different approach, but the experiments provided a good testing ground for determining what kinds of changes were useful and what kinds of changes just made the interface more confusing and harder to use.

3.1 Color Selection

Color enhancement was at the top of the list when I asked researchers what changes they wanted in the application interface. In the version of GraphEd that I was working with, there were hooks in the code for color usage, but it had never fully been implemented. Highlighting of nodes was just done in black.

The first change was to make the default highlighting color red. Highlighting is done by coloring a rectangle around the borders of a node. This color was previously black and did not provide enough contrast from the rest of the graph to stand out. This change was a simple one and merely required creating a graphics context to replace the default. The graphics context defines foreground color, background color, line style, and other details of how graphics are drawn. The new graphics context is defined with a red foreground. The border rectangle can then be drawn and filled in with the new graphics context.

The second change involved setting up an array of color selections that could be used in an application module to change the colors of nodes and edges for highlighting or grouping purposes. There are hooks in the code for color usage in the GraphEd source, but in the version I am working with the feature had not yet been fully implemented. Using `set_node()` to change a color attribute has no effect. To set up the color selection, a colormap segment including a wide range of colors was created. A colormap segment is a subset of cells in an X server's

colormap. From this colormap segment an array of graphics contexts are created; each uses a different color from the colormap segment as the foreground color. In the original application the color index from a `set_node()` function was passed but never used; now the number is directly used to index the correct graphics context in the array and the node or edge color can be changed. The new color index can be seen in Table 3.1. The creation of this array requires one other change. Previously, the colors in the graph were either black or white, so the depth of the drawable only needed to be one and the function `XCOPYPlane` could be used to copy the drawable to the canvas when repainting. With the addition of color, this depth is increased and each instance of the function `XCOPYPlane` must be changed to the function `XCOPYArea`. The function `XCOPYArea` is capable of copying a drawable with a depth greater than one to a color canvas.

Table 3.1: GraphEd color index

white	0	orange	10
black	1	brown	11
green	2	olive	12
blue	3	dark blue	13
cyan	4	turquoise	14
yellow	5	gold	15
magenta	6	violet red	16
red	7	dark red	17
purple	8	dark magenta	18
gray	9	dark grey	19

3.2 Format Converters

Format converters were another crucial issue for the researchers. It is easy to draw graphs in the GraphEd environment, but it is a time-consuming activity, especially when the graph already exists in a different graph description format. The researchers frequently used a format based on graph formats specified for the Implementation Challenges at the Center for Discrete Mathematics & Theoretical Computer Science (DIMACS). They wanted to be able to convert the files they had in this DIMACS description format to the GraphEd description format. They also wanted a function that would convert a graph they had just drawn in the GraphEd environment back to the DIMACS format. The scripts **dimacs2graphed** and **graphed2dimacs** were written for this purpose. These scripts, along with the random graph generator described in the next section, are not built into the GraphEd executable. They are included in the distribution for use outside of the main executable.

The DIMACS format is capable of describing both simple graphs and multi-graphs. Each line in the format is preceded by a one-letter code. A 'c' represents a comment line. A line preceded by 'p' is required at the beginning of the description. It is followed by two space-separated integers. The first integer describes the number of nodes in the graph; the second represents the number of edges. The remaining lines all begin with an 'a' and represent the various arcs in the

```

c ....DIMACS graph.... <-comment line
p 5 7                  <-number of nodes and edges
a 1 2 1
a 1 3 5                <-arc line with multiple edges between nodes
a 1 4 2
a 2 3 1                <-arc line with only one edge between nodes
a 2 4 4
a 3 4 3
a 5 2 1

```

Figure 3.1: DIMACS format example

graph. An 'a' is followed by three space-separated integers. The first integer is the source node of the arc, the second is the target node, and the third describes how many arcs run between these two nodes. There must be a separate line for each arc. The only change I made to this format is the addition of a 'd' code to represent graph direction. The 'd' is followed by a zero or a one. A zero indicates that the graph is undirected. A one indicates a directed graph. Figure 3.1 gives an example of this format.

The GraphEd format seems more complicated at first, but can be simplified for the purpose of format conversions. A graph description file that has been created by GraphEd contains a lot of information including all of the internal information for the nodes and edges. But all of this information is not needed to read a graph. If internal information such as font, node type, or edge type are not supplied in a description file, the defaults are used. To create a GraphEd graph from a DIMACS description file, a simpler format, using only the essential

attributes, can be followed. The beginning of the description gives the direction of the graph. The rest of the graph is adjacency-list based. Each node is given its own description line with node internals specified in braces and a node label quoted at the end of the line. In this case, the only node internal that must be given is the node position. On the lines following each node description is a list of its adjacent nodes followed by a quoted edge label. For this application the node label represents the internal node number and the edge label represents the multiplicity of the arc. Figure 3.2 gives an example of this format.

Since GraphEd requires a coordinate position for each node in order to display the graph, some random layout scheme had to be chosen. To do this, a central pixel position in the drawing window was determined. Based on the number of nodes in the graph, the nodes are laid out in a square around this central position. Each row is offset from the ones on either side of it by a node's width. The scheme is simple but seemed to produce fairly good results and caused a minimum of edge overlap.

Each converter is made up of two scripts, a driver program and an awk script. The driver program reads the command line options and calls the awk script, with the proper arguments, that will read the input file and process the information. As discussed earlier, the **dimacs2graphed** converter takes a DIMACS graph description file and produces a graph description file that can be read and displayed by GraphEd. The command line arguments for the converter are an input file (in

GRAPH "" = UNDIRECTED

{ \$

\$ }

1 { \$ NP 1770 2170 \$ } "1"

2 "1"

3 "5"

3 "5"

3 "5"

3 "5"

3 "5"

4 "2"

4 "2"

;

2 { \$ NP 1830 2170 \$ } "2"

3 "1"

4 "4"

4 "4"

4 "4"

4 "4"

;

3 { \$ NP 1790 2230 \$ } "3"

4 "3"

4 "3"

4 "3"

;

4 { \$ NP 1770 2290 \$ } "4"

;

5 { \$ NP 1850 2230 \$ } "5"

2 "1"

;

END

Figure 3.2: GraphEd format example

DIMACS format) and an output file (by convention this should end in '.g'). Both of these arguments are optional. In the absence of an input file, the converter takes its input data from the standard input. This means the user will have to type in each line of the DIMACS file and terminate by typing a **Control-D**. In the absence of an output file, the converter will print the new graph to standard output. The script **dimacs2graphed** evaluates these command line options, then runs the **dimacs.awk** script with the appropriate arguments.

The **dimacs.awk** script consists of three sections: commands that are done before lines are read from the input file, commands done as each line is read, and commands done after all lines have been read. The first section requires only two initializations. The central pixel position must be initialized, and a flag indicating if the next line of nodes should be offset is initialized to zero. The main body of the awk script is made up of a series of **if** statements, which are actions for each possible one-letter code read from the current line. Information from this section is stored and organized so that the output file can be written out all at once in the last section of the script after all of the input lines have been read. The actions for 'd' and 'p' lines are fairly simple. When a 'd' line is read, the direction parameter can be stored, and when a 'p' line is read, the number of nodes and edges is stored and the node position parameters can be determined. Storing the information from the arc lines is a more complicated process. As arc information is read, adjacency lists are built for each node. An array holds each adjacency

list as a string. Each adjacency list is initialized to NULL. When an arc line is read, the adjacency list for the source node is examined. If the adjacency list is NULL, then this is the first time the node has been seen as a source node and the adjacency list is initialized with the node description: the node number, followed by the node position in braces, followed by the node label. Then the target node description is appended to the string: a new line, followed by the node number, followed by the arc label. Whenever this node appears as a source node again, it will only be necessary to append the new target node description to the end of the adjacency list. When all of the arcs have been read, any adjacency lists that are still empty are given node description strings. These nodes were standalone nodes or just target nodes. In the final section of the awk script, each of the adjacency lists, along with the other graph information, can simply be printed to the output file.

The driver script for **graphed2dimacs** is similar to that of **dimacs2graphed**, and the awk script is once again broken up into three sections. In the first section, some simple initializations are made. The central part of the awk script once again includes a series of if statements. By reading the first argument in each line, it can be determined what action to take and what information to store. From a line beginning with "GRAPH", the direction of the graph can be determined and either "d 0" or "d 1" can be printed to the output file. A running total of edges and nodes is kept throughout the script so that the 'p' code can be written in

the final section. The arc, or 'a', lines are a bit more difficult to determine. A number as the first argument in a line indicates the beginning of an adjacency list. A semicolon indicates the end of an adjacency list. As the lines between these delimiters are read, the arc list for this source node is built and appended to a string. An arc list string like this is maintained for each node. When a new node is encountered, the first arc for this node can be appended to its string. The arc line is made up of an 'a' followed by a source node, a target node, and the multiplicity of the arc. The source node has just been determined from the current line, the target node will be determined by the next line read, and the multiplicity will be determined by the number of times this same target node is read. Once a new target node is read, a new arc line can be constructed. When a semicolon is read, the end of the current adjacency list has been reached and the output string is complete. Once all adjacency lists have been read in this manner, the arc lists can be printed to the output file in the proper order.

3.3 Random Graph Generators

The researchers with whom I spoke wanted to be able to display large random graphs in GraphEd without having to go through the lengthy process of drawing them. To accomplish this I wrote a short C program called **generate**, which allows for the quick generation of a random graph in DIMACS format. The generated

graph can be used in applications or can be converted for viewing in a GraphEd environment. There are several command line options available for this program. The first two specify the number of nodes and edges wanted in the graph. These parameters are not optional. The next parameter specifies if the graph should be directed or undirected. If this parameter is not set, it will default to being a directed graph. The '-f' parameter allows you to set the output file, the default being standard output, and the '-h' parameter will print out the program usage specification.

After all of the command line parameters have been evaluated, initializations are done. A matrix of integers is allocated to represent all possible arc connections. `connect[i][j]` is initialized to be zero unless `i` is equal to `j`. In this case, the position is initialized to two, indicating there can be no arc in that position. At this point the random edge selection begins. For every edge that must be created, a random source node and a random target node are selected. The calculation for this is `(random()%(number of nodes)+1)`. If there is not yet an arc between these two nodes, the position is marked with a one. If the graph is undirected, the reverse edge is marked as unavailable. If this edge has already been chosen, the selection process will continue until a free arc is found. This process will create only simple graphs, not multigraphs.

Once all edges have been selected, the matrix is traversed and every arc position marked with a one is printed to the outfile in DIMACS format. Specifications of

how to use all of these scripts can be found in Appendix B.

3.4 Extended User Interface

The most useful additions to the GraphEd interface were the extensions to the user interface. The researchers found many of the application modules, those that came with the package and those that I added, to be useful. But they would also like to be able to use GraphEd to visualize applications they had written that were not included in the collection of application modules. They would like to be able to change colors and labels of graphs from their own applications if doing so did not require a lot of code changes. I have taken an experimental approach toward these tasks.

There is a difference between how I have added application modules to GraphEd and how the researchers will add application modules. First of all, when I implemented my application modules I used GraphEd methods. The functions I used to traverse nodes and edges and to change node and edge labels were functions defined in the GraphEd source. The data structures I used to represent graphs were those defined by the Sgraph environment. Considerable recoding would be required for them to add their applications to GraphEd using the same methods I used. Second, the manner in which I added these application modules made them part of the GraphEd executable. Instructions in the manual show the user

how to add an application to the appropriate place in the source tree, recompile, and have the application listed as a permanent menu item. The researchers would add an application dynamically, without recompiling the entire GraphEd source. While this method is more limiting in some ways, it would require only minimal work.

The user will need to create a `.graphed_user_menu` file and a `graphed_user` directory. The `.graphed_user_menu` file will contain a line for each new application. This line will have the name of the application, as it will appear on the menu, followed by the command to execute the application. All executables listed in the `.graphed_user_menu` file will be placed in the `graphed_user` directory so that GraphEd can find them and execute them when chosen. These executables are enhanced with the new library calls and have been recompiled. When GraphEd starts up, it will read the `.graphed_user_menu` file and add the appropriate items to the menu. The user can then load a graph into the window and choose an application to run. When one of the dynamic menu items is chosen, the executable is spawned and a communication pipe is set up between this executable and the GraphEd process. Whenever a library function is called, to color a node or an edge, or to change a label, the parameters of this call are sent through the pipe to GraphEd and the changes are made to the currently displayed graph. The graph is redrawn with each change. When the child process has completed, the pipe is closed and another process can be selected. One difficulty in this imple-

mentation is that any node renumbering will not be paralleled in the currently displayed graph. If node renumbering is necessary, the application must maintain a mapping from old to new or the wrong nodes or edges may be affected.

The process I have outlined required that the source code handling user commands be altered. The first change was in the code for adding items to the GraphEd user menu. The original `init_user_menu()` function only added a usage statement under the user menu. To enable dynamic additions to this menu, a couple of arrays are initialized. An array `cmenu_items` of strings is allocated to hold all strings read from the `.graphed_user_menu` file. Another will hold the path name to this file. When the GraphEd executable is started, the path to the user's configuration file is built and each line of the file is scanned and the relevant information is stored in `cmenu_items`. Once all items have been scanned, they are added to the menu using the `add_to_user_menu()` function. The callback function for each of these items will be `user_fork()`. It is this function that will spawn the appropriate executable when a dynamically added item is chosen from the menu.

As discussed earlier, to be able to make changes to a graph from a user function a pipe of communication must be set up between the user process and the GraphEd process. Once this link is set up, user library functions can send data to their counterparts on the GraphEd process side so that the change can be executed. On the user side, there are currently four functions in the user library `glib.a`:

`graphed_color_node()`, `graphed_color_edge()`, `graphed_label_node()`, and `graphed_label_edge()`. Each of these functions work in much the same way. The color functions accept as arguments a node, or a pair of nodes if an edge is being colored, and a color index. (The color list is displayed in Appendix B.) At this point, the pipe has already been set up by the GraphEd process and an environment variable has been set to indicate the descriptor for the pipe. The user function must read the environment variable to get this descriptor before sending its node information to GraphEd through the pipe. It will send an index to indicate the function being performed, followed by node information, followed by the color index. The label functions work similarly but send a label string rather than a color index.

On the GraphEd side, when a menu item has been selected, the `user_fork()` function is called. This function sets up the pipe and forks off a process. Since information only needs to flow through the pipe in one direction, the parent closes off the outgoing descriptor of the pipe and the child closes off the incoming descriptor of the pipe. The child process copies the file descriptor for the pipe to the user's environment so that it can be read by the process on the other side of the pipe. Then the child executes the user executable. The GraphEd process now waits to be notified that there is information in the pipe. The `read_it()` function is called when there is something to read on the pipe. The first argument is an index that will tell which library function has been called. This will let the `read_it` process

know what else it needs to read from the pipe. If the `graphed_color_node()` function has been called, a node number and a color index must be read. For the `graphed_color_edge()`, a source node, a target node, and a color index must be read. For `graphed_label_node()` and `graphed_label_edge()`, the node or nodes and a string must be read. Once all information has been read from a library function, the partner function on the GraphEd side is called. This partner function will traverse the graph for the matching node or edge. If the library call was for a color change, `node_set()` or `edge_set()` is called to change this node or edge attribute. If the library call was for a label change, then `set_nodelabel()` or `set_edgelabel()` is used. Once the appropriate action has been completed, the graph is redrawn and the process waits for new input. After the user process has exited, the pipe is closed and a new menu item can be selected.

With these changes, the user can visualize their own algorithms with a minimum of recoding. A user manual for these new features can be found in Appendix B.

Chapter 4

Conclusions and Future Work

The development of enhancements to the GraphEd package proved to be a valuable experiment for many reasons. The project tied together several disciplines, including GUI interface work, system programming, and application development. The system and interface work were more familiar to me and allowed me to apply skills that I had already developed. But the graph algorithm work was new and proved to be both the biggest challenge and greatest learning experience in the project. To implement an algorithm from a research paper would not necessarily require understanding the algorithm, but to come up with the best possible implementation, this understanding was essential.

This project also provided a good opportunity to work in a team environment. It would be easy as a programmer to sit down and implement changes that seem like a good idea. But it is far more challenging to sit down with the end users of a

product and discuss what would be useful to them and make compromises based on programming limitations. We came up with some good ideas in our meetings. Some may or may not be used in the future, but it often takes a few examples of what does not work to come up with the right solution. It is my hope that this project will serve as a good basis for future improvements.

4.1 Future Application Module Additions

Several issues could be addressed in future work. The application modules that I have added are solid additions, but further modifications could be made. As stated earlier, the series parallel test is unable to show the distinct paths in the K_4 at this time. The output display of this application would be greatly enhanced if this were possible. A deeper understanding of how the algorithm works might enable a solution to this problem. Second, while there is a coded solution to finding the chordless cycle in a nonchordal graph, it has not been conclusively proved. Research devoted to proving this solution or finding a better one could make this application more efficient. Third, a lot could be done to enhance the pathwidth test. The test currently implemented only finds the pathwidth and reports the value. Future work could focus on finding the path decomposition of the graph. This decomposition could be demonstrated by the use of different colors. The test could also be rewritten to handle graphs other than trees. Finally,

there are many new applications that could be added to the package. One that we can already see a need for is a layout algorithm that could group graph nodes according to their color.

4.2 Future Interface Enhancements

In the area of interface enhancements there are several possibilities for future projects. For example, the format conversion of graph files is now done as a separate action outside of the executable. If these formats could be read automatically and converted as a graph is loaded, the process would be invisible to the user. Many improvements could also be made in the use of color in the create and edit modes of GraphEd. Finally, the user library and dynamic menu addition features need to be evaluated. What is currently in place is a good start, but as the new interface features are tested, researchers may discover functions that need to be added to the user library or they may find a different approach to the problem as the University of Passau continues their research.

A graph editor can be a very powerful tool but with the help of end users it can be developed into an even more useful one. With this sort of interaction, testing can be done throughout development to make sure enhancements are helpful and don't make the package harder to use. As a result of the work outlined in this paper, researchers have more application modules at their disposal, and they are

able to visualize more of their existing work with GraphEd due to the color and interface improvements.

Bibliography

Bibliography

- [BB88] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [BGLR96] H.D. Booth, R. Govindan, M.A. Langston, and S. Ramachandramurthi. Fast algorithms for k_4 immersion testing. Technical report, Department of Computer Science, University of Tennessee, 1996.
- [EST87] J.A. Ellis, I.H. Sudborough, and J.S. Turner. Graph separation and search number. Technical Report DCS-66-IR, Department of Computer Science, University of Victoria, 1987.
- [EST94] J.A. Ellis, I.H. Sudborough, and J.S. Turner. The vertex separation and search number of a graph. *Information and Computation*, 113:50–79, 1994.
- [FL88] M.R. Fellows and M.A. Langston. Nonconstructive tools for proving polynomial-time decidability. *Journal of the ACM*, 35:727–739, 1988.

- [FL94] M.R. Fellows and M.A. Langston. On search, decision and the efficiency of polynomial-time algorithms. *Journal of Computer and Systems Sciences*, 49:769–779, 1994.
- [GL80] R.C. Geldmacher and P.C. Liu. Algorithm for finding a subgraph homeomorphic to k_4 . *Congressus Numerantium*, 29:597–609, 1980.
- [Hel91] D. Heller. *XView Programming Manual for XView Version 3*, volume 7. O'Reilly and Associates, Inc., 1991.
- [Hima] M. Himsolt. *GraphEd User Manual*. University of Passau, Passau, Germany.
- [Himb] M. Himsolt. *Implementing Graph Algorithms with GraphEd (Extended Abstract)*. University of Passau, Passau, Germany.
- [Himc] M. Himsolt. *Sgraph 3.1 Programming Manual*. University of Passau, Passau, Germany.
- [Kin92] N.G. Kinnersley. The vertex separation number of a graph equals its path-width. *Information Processing Letters*, 42:345–350, July 1992.
- [LP95] M.A. Langston and B.C. Plaut. Algorithmic applications of the immersion order. In *Slovenian International Conference on Graph Theory*, Lake Bled, Slovenia, June 1995. (Invited, Proceedings).

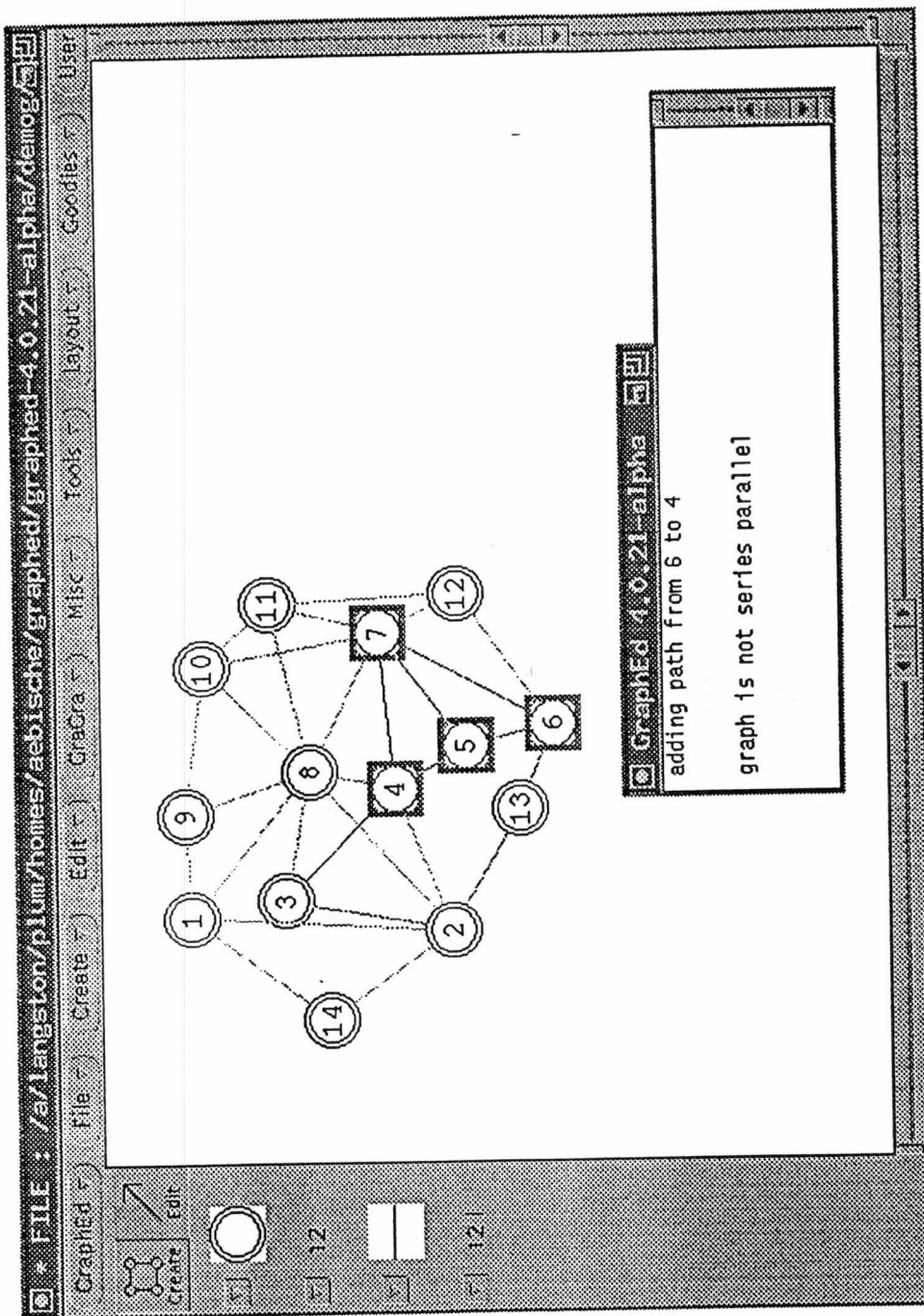
- [Mit79] S.L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, December 1979.
- [RTL76] D.J. Rose, R.E. Tarjan, and G.S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comp.*, 5:266–283, 1976.
- [Sch90] P. Scheffler. A linear algorithm for the pathwidth of trees. *Topics in Combinatorics and Graph Theory*, pages 613–620, 1990.
- [ST92] M.N.S. Swamy and K. Thulasiraman. *Graphs: Theory and Algorithms*. John Wiley and Sons, Inc., New York, 1992.
- [Tur85] D.S. Turner. The np-completeness column: An ongoing guide. *Journal of Algorithms*, 6:434–451, 1985.
- [TY84] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comp.*, 13(3):566–579, August 1984.

Appendices

Appendix A

Application Module Output

Examples



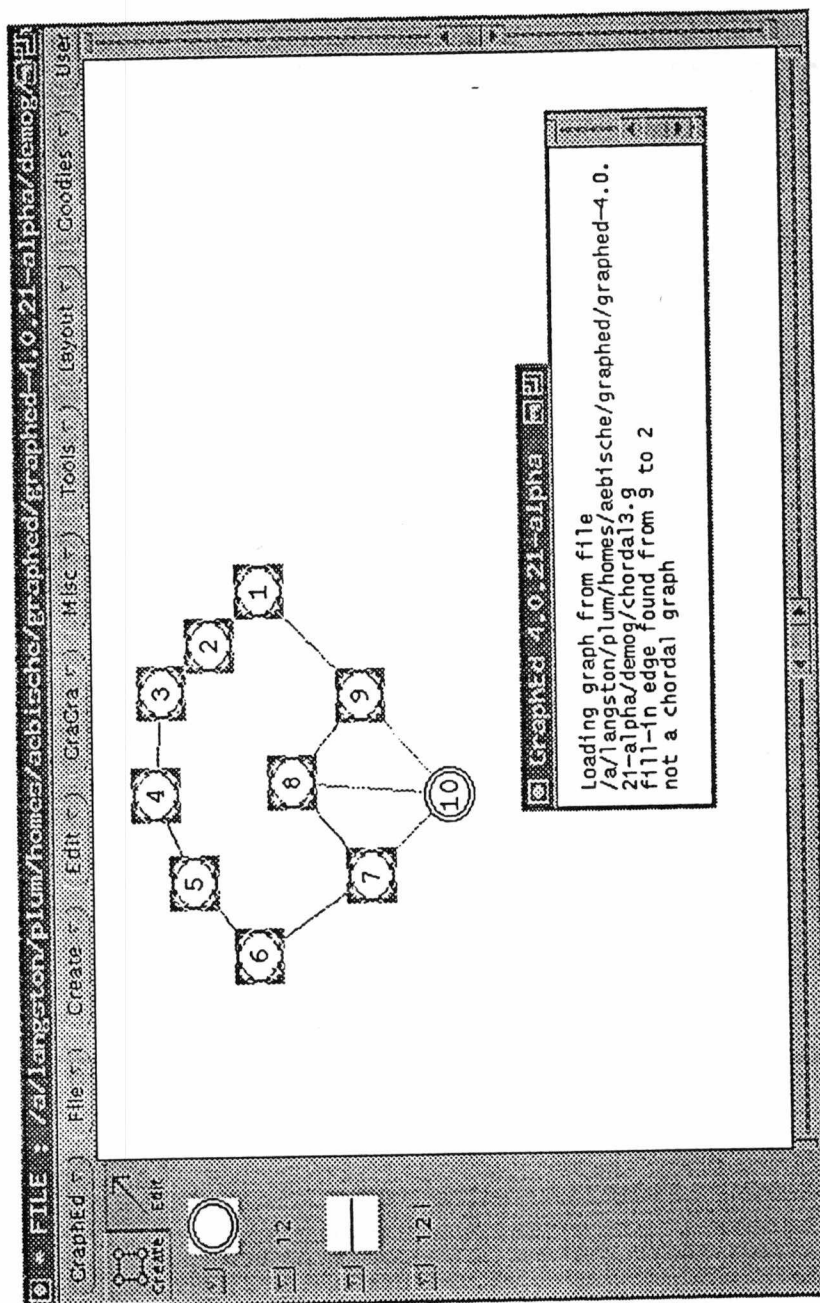


Figure A.2: Chordal Graph Recognition Output

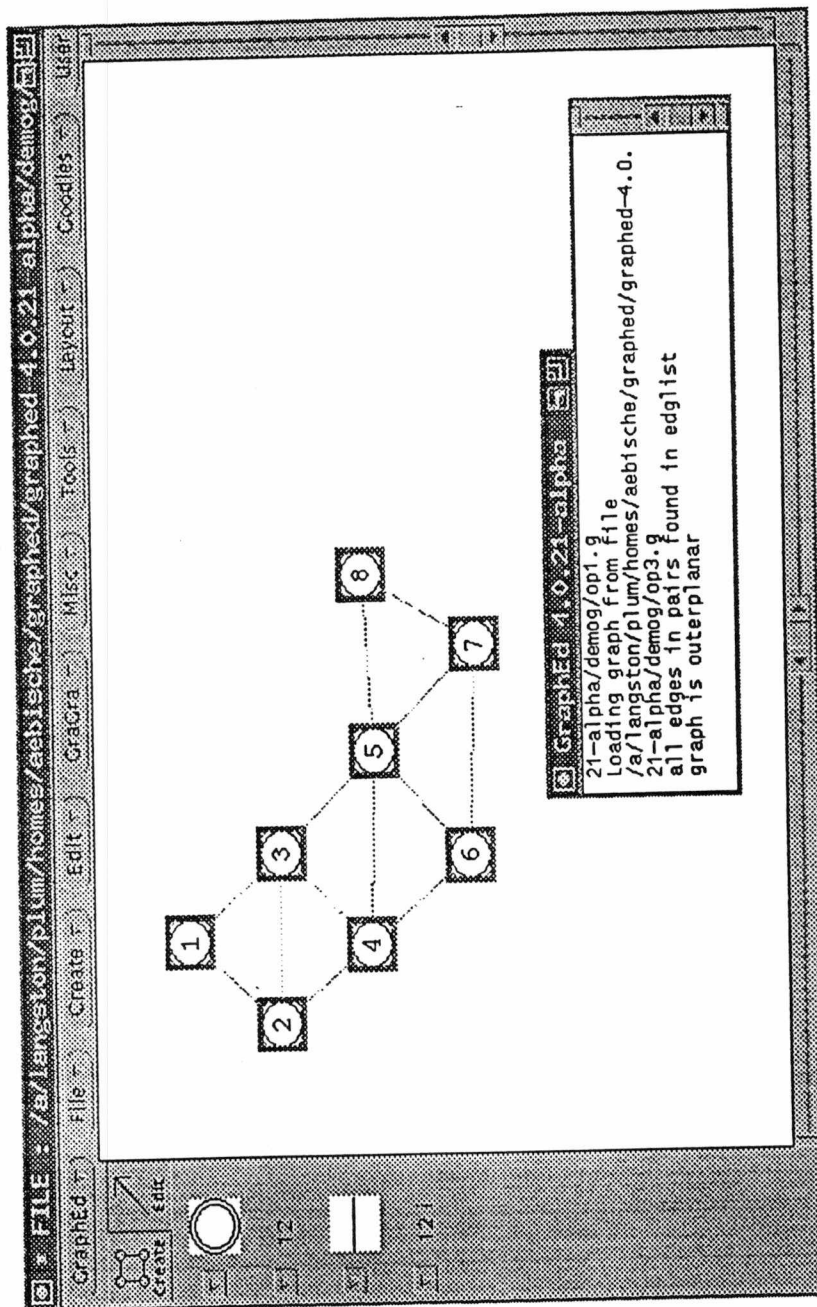


Figure A.3: Outerplanar Graph Recognition Output

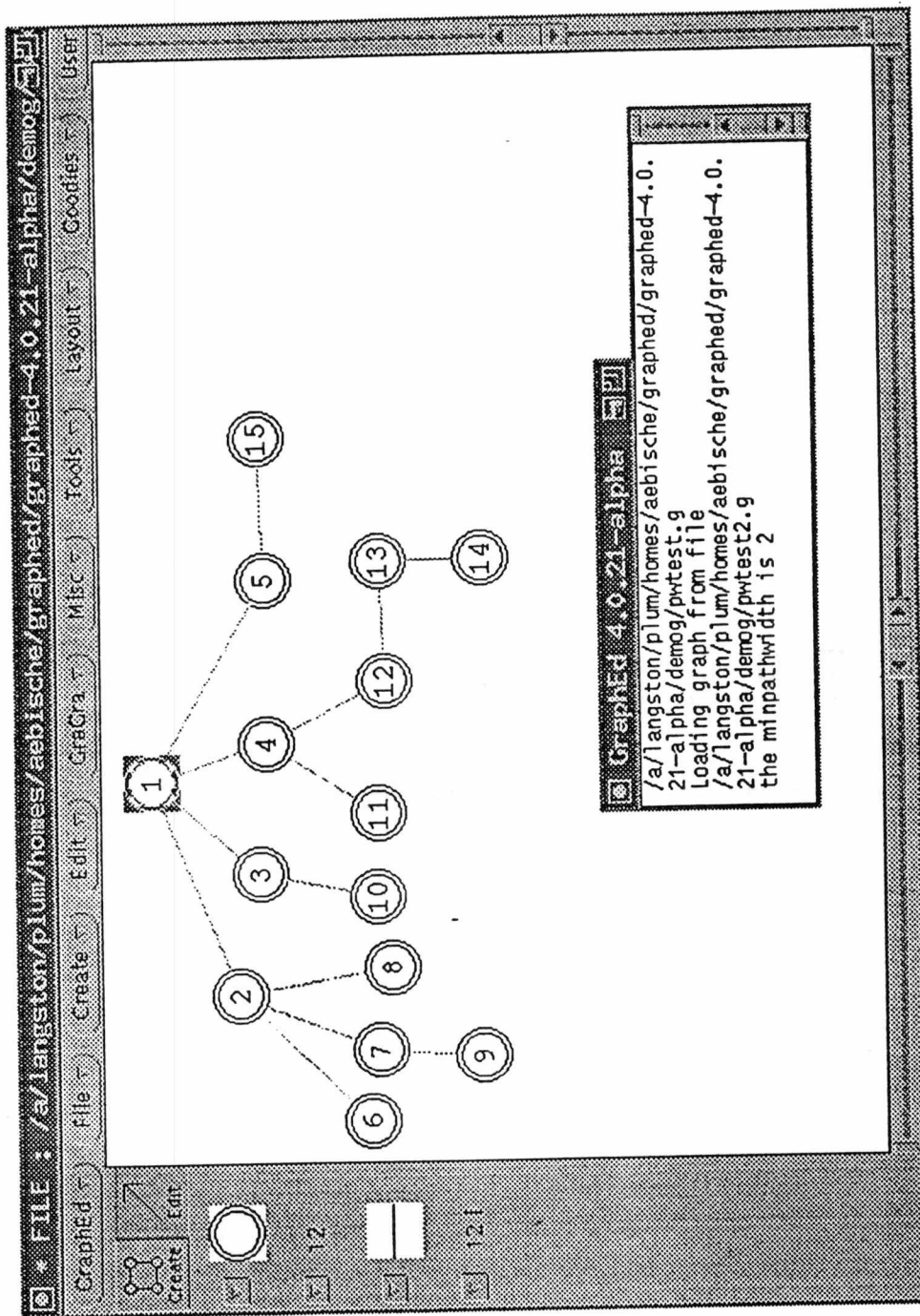


Figure A.4: Pathwidth Output

Appendix B

User Manual for Interface

Extensions

As discussed in the paper, the extended user interface allows the user to add library calls to existing graph code. These library calls can change the color of a node or edge and change the node and edge labels of a displayed GraphEd graph. Once these additions have been made and the code has been recompiled, the user can then dynamically add this code to the GraphEd user menu for execution. To take advantage of these features it is necessary to follow these steps:

1. Set environment variable `$GRAPHED_HOME` to `/home/aebische/graphed/graphed-4.0.21-alpha`.

2. Add new library calls wherever needed in the code. The available functions are:

- `graphed_color_node(node_num, color_index)` - changes the color of a node.

`node_num` An integer representing the node number.

`color_index` An integer representing a color from the color index. (Figure B.1).

- `graphed_color_edge(node_src, node_tgt, color_index)` - changes the color of an edge.

`node_src` An integer representing the number of the source node.

`node_tgt` An integer representing the number of the target node.

color_index An integer representing a color from the color index. (Figure B.1).

- **graphed_label_node(node_num, label)** - sets the label of a node.

node_num An integer representing the node number.

label A string containing the new label.

- **graphed_label_edge(node_num, label)** - sets the label of an edge.

node_src An integer representing the number of the source node.

node_tgt An integer representing the number of the target node.

label A string containing the new label.

NOTE: Remember that any internal renumbering of nodes in your code will not be paralleled on the displayed graph so it will be necessary to keep a map to the original numbering in order to reference the correct node for future color and label changes.

Table B.1: GraphEd color index

white	0	orange	10
black	1	brown	11
green	2	olive	12
blue	3	dark blue	13
cyan	4	turquoise	14
yellow	5	gold	15
magenta	6	violet red	16
red	7	dark red	17
purple	8	dark magenta	18
gray	9	dark grey	19

3. Recompile code with the library found in `$GRAPHED_HOME/user_lib/user_lib.a`.
4. Create directory `~/graphed_user` and place the new executable, and any other executables you have prepared, in this directory.
5. Create the file `~/graphed_user_menu` and place on each line the name of your program as you'd like it to appear on the menu (this needs to be one word), followed by the actual executable name with any arguments that are required for it to run. See Figure B.2 for an example of this.

```
demo1      demo <arg1>
demo2      demo <arg1> <arg2>
color_node  cnode <arg1>
color_edge  cedge <arg1> <arg2>
```

Figure B.1: Sample `.graphed_user_menu` file

6. Start a GraphEd process, your list of executables should appear under the 'user' menu.
7. Load a graph you have prepared for one of these executables and select the desired process from the user menu. The color and label changes you have added should appear on the graph as your process executes.

Finally, a list of specifications for the **dimacs2graphed**, **graphed2dimacs**, and **generate** scripts located in the `$GRAPHED_HOME/dimacs_converters` directory:

- **dimacs2graphed** [inputfile] [outputfile]
 - **inputfile** - DIMACS file to convert. Default is standard input.
 - **outputfile** - GraphEd file to write. Default is standard output.
- **dimacs2graphed** [inputfile] [outputfile]
 - **inputfile** - GraphEd file to convert. Default is standard input.
 - **outputfile** - DIMACS file to write. Default is standard output.
- **generate -n <#nodes> -e <#edges> [-d 0 | 1] [-f outfile] [-h]**
 - **-n** - number of nodes to generate. required.
 - **-e** - number of edges to generate. required.
 - **-d** - 0 for undirected graph. 1 for directed graph. default is 1.
 - **-f** - output file for DIMACS output. default is standard output.
 - **-h** - print out help information.

Vita

Carolyn Aebischer was born in Northampton, Massachusetts but grew up in Oak Ridge, Tennessee. She attended Oak Ridge High School and graduated in 1986. From here she attended Wake Forest University where she majored in Mathematics and minored in Computer Science, completing her degree in 1990. Since then she has been dragging out her graduate education at the University of Tennessee and working as a systems coordinator at Oak Ridge National Laboratory.