AN INTERACTIVE SOFTWARE TOOL FOR EDITING AND EVALUATING MECHANICAL ASSEMBLY SEQUENCES BASED ON FIXTURING AND ORIENTATION REQUIREMENTS

by

THOMAS EDWARD ABELL

S. B. Mechanical Engineering, MIT

1988

Submitted to the Department of Mechanical Engineering

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1989

© Thomas Edward Abell, 1989

The author hereby grants to M.I.T. and The Charles Stark Draper Laboratory, Inc.
permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author __________________________________________________________

Department of Mechanical Engineering

August 11, 1989

Certified by ____________________________

Daniel E. Whitney
Thesis Supervisor

Accepted by ____________________________

Ain A. Sonin
Chairman, Department Committee

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

MAR 13 1990
LIBRARIES
ARCHIVES
DISCLAIMER

This research is supported by the National Science Foundation. It represents the opinions of the author only, and is offered for the stimulation and exchange of ideas. It does not represent the policies of the National Science Foundation, the Charles Stark Draper Laboratory, or the Massachusetts Institute of Technology.
AN INTERACTIVE SOFTWARE TOOL FOR
EDITING AND EVALUATING MECHANICAL ASSEMBLY SEQUENCES
BASED ON FIXTURING AND ORIENTATION REQUIREMENTS

by
Thomas Edward Abell

Submitted to the Department of Mechanical Engineering
on August 11, 1989 in partial fulfillment of the
requirements for the Degree of Master of Science at the
Massachusetts Institute of Technology

ABSTRACT

For products to be competitive, we must consider manufacturing issues, including
assembly, early in the design process. The choice of assembly sequence is an important
factor in the design of products. Through Liaison Sequence Analysis, we can generate all
the possible assembly sequences for given mechanical products. To evaluate assembly
sequences early in product design, we need computer tools that make the evaluation
process faster and easier. This thesis describes the implementation of a computer tool that
allows efficient evaluation and editing of assembly sequences. It is based on a variety of
assembly sequence editing strategies including the need to minimize the number of fixturing
and orientation changes during assembly.

The tool's facilities for editing assembly sequences include several functions based
on deleting individual assembly states and assembly moves. Many of the previously known
editing strategies like precluding multiple subassemblies, eliminating particular sets of
simultaneous part matings, and specifying strict ordering of assembly moves, have been
incorporated as simple interactive commands. The software also gives us the option to
delete all arborescent assembly sequences that represent the same parts tree. Evaluating
assembly sequences based on fixturing and orientation requires us to enter information
about how each part and unique subassembly is fixtured and oriented. The program will
then use one of two shortest path algorithms to find assembly sequences that require
relatively few fixturing and orientation changes. Other features of the editing tool are the
use of part drawings to illustrate assembly states and moves and the close integration of this
tool with the assembly sequence generation tools.

Thesis Supervisor: Dr. Daniel E. Whitney, Lecturer, Mechanical Engineering
ACKNOWLEDGEMENTS

I would first like to thank Dr. Daniel Whitney, who proved to be the perfect thesis advisor. He provided just the right amount of guidance and advice. I want to thank Dr. Tom De Fazio for providing direction and enthusiasm for my work. I would also like to thank all the other people at Draper I had the pleasure to work with. Ava Seekins and Jane Winter were always helpful answering questions when I had problems.

My experience at Draper was made particularly enjoyable by all the students I worked with. Dan Baldwin was a great office mate. I want to congratulate him on his wedding and wish him the best of luck in his pursuit of a PhD here at MIT. I thank Max Lui for helping me get started on the project and for continuing to provide support from California. All the other students, Eugene Tung, Suguna Pappu, Guillaume Amblard, Tom Kurfess, Russ Whipple, Trish Roxas, Alex Denner, and Curt Cooprider have been good friends. I thank all of you and wish you the best of luck.

This work was funded by the National Science Foundation under Grant DMC-8715800.
TABLE OF CONTENTS

Section ................................................................. Page

ABSTRACT........................................................................ 3

ACKNOWLEDGEMENTS...................................................... 4

TABLE OF CONTENTS....................................................... 5

LIST OF FIGURES........................................................... 8

1.0 INTRODUCTION......................................................... 11
    1.1 Goals of Thesis..................................................... 14
    1.2 Background on Liaison Sequence Analysis..................... 14
        1.2.1 Generation of Precedence Constraints..................... 15
        1.2.2 Representing Assembly Sequences......................... 18
        1.2.3 Evaluating Assembly Sequences......................... 23

2.0 OVERVIEW OF CURRENT SOFTWARE SYSTEM.................... 25

3.0 SEQUENCE GRAPH DATA STRUCTURE AND DISPLAY.............. 31
    3.1 Sequence Graph Data Structure................................ 31
        3.1.1 Rank Structure............................................ 32
        3.1.2 State Structure........................................... 33
        3.1.3 Parent Transition Structure............................. 36
        3.1.4 Child Transition Structure.............................. 38
        3.1.5 Subassembly Structure................................. 38
    3.2 Sequence Graph Display......................................... 39

4.0 SEQUENCE GRAPH EDITING BASED ON STATES AND TRANSITIONS... 42
    4.1 Screen Editing: Viewing and Deleting Individual States and Transitions... 42
    4.2 Editing State Transitions Based on Multiple Liaison Establishment........ 48
    4.3 Requiring One Liaison to Immediately Follow Another................... 53
    4.4Editing States Based on the Number of Subassemblies..................... 56
    4.5 Editing Based on Redundant Arborescent Sequences.................... 58
5.0 EVALUATING ASSEMBLY SEQUENCES BASED ON FIXTURING AND ORIENTATION....................................................... 67

5.1 Background on Fixturing and Orientation................................. 67
  5.1.1 Fixturing................................................................. 67
  5.1.2 Orientation............................................................. 69
  5.1.3 The Relationship Between Fixturing and Orientation............ 71
  5.1.4 How to Use Fixturing and Orientation Information............. 71

5.2 Shortest Path Approach...................................................... 72
  5.2.1 Background on Shortest Path Algorithms.......................... 72
  5.2.2 Applying Shortest Path Approach to Sequence Graph.......... 75

5.3 Representation of Fixturing and Orientation............................ 79
  5.3.1 Representation of Data.............................................. 80
  5.3.2 Associating Fixturing and Orientation With Assembly States... 82
  5.3.3 Acquiring Fixturing and Orientation Data........................ 83
  4.3.4 A Small Example.................................................... 85

5.4 Expanding Sequence Graph................................................... 93
  5.4.1 Creating Expanded Sequence Graph Data Structure............ 93
  5.4.2 Calculating Fixturing and Orientation Cost..................... 98

5.5 Implementation of Shortest Path Algorithms............................ 108
  5.5.1 K-Shortest Paths Algorithm...................................... 108
  5.5.2 Array Shortest Paths Algorithm................................. 111
  5.5.3 Comparison of the Two Algorithms................................ 120

5.6 AFI Example........................................................................ 124

6.0 RECOMMENDATIONS FOR FUTURE RESEARCH.............................. 128

6.1 Directly Entering Precedence Relations..................................... 128

6.2 General Logical Functions...................................................... 128

6.3 Undoing Editing Moves.......................................................... 129

6.4 Improving Fixturing and Orientation Algorithm........................ 130
6.4.1 Attaching Fixturing and Orientation to Individual Parts ............. 130
6.4.2 Including Grippers as well as Fixtures .................................. 131
6.4.3 Expansion of Sequence Graph .............................................. 132
6.4.4 Other Improvements ................................................................ 135

6.5 Acquiring Fixturing and Orientation Data .................................... 136
6.5.1 Automatically Generating Data ................................................... 136
6.5.2 Entering Fixturing and Orientation Data .................................... 137

6.6 Other Evaluation Criteria ............................................................ 139

7.0 CONCLUSIONS ............................................................................... 141

8.0 REFERENCES .................................................................................. 142

9.0 APPENDICES .................................................................................. 146

   Appendix A. Data Structure Declarations and External Variables .......... 146
   Appendix B. EDIT Function Directory ................................................. 152
   Appendix C. Procedures for Using EDIT ............................................. 157
   Appendix D. Creating Part Drawings .................................................. 163
LIST OF FIGURES

1-1. Example assembly and liaison diagram. 15
1-2. Ball-point pen assembly drawing and liaison diagram. 19
1-3. Parts trees representing all assembly sequences of ball-point pen. 20
1-4. And-Or graph representation of ball-point pen assembly sequences. 22
1-5. Liaison sequence diagram for ball-point pen assembly. 23
2-1. How EDIT fits into Liaison Sequence Analysis Software. 26
2-2. EDIT overall program structure. 27
2-3. Example of EDIT's screen display. 28
3-1. Rank data structure. 33
3-2. An example state structure and its parent, child, and subassembly structures. 34
3-3. Example illustrating the construction of subassembly tables (subasby_tbl)
in the parent transition data structures of a state with two parents. 37
3-4. Illustration of data structures and variables used in the display of the
sequence graph. 40
4-1. EDIT screen display showing AFI assembly drawing. 43
4-2. Example of SHOW STATE function. 45
4-3. Example of SHOW TRANSITION function. 46
4-4. Illustration of simultaneous mating of liaisons 8, 9, and 10. 49
4-5. Eliminating the simultaneous establishment of liaisons 8, 9, and 10. 50
4-6. Flowchart of function to eliminate specified sets of simultaneous liaisons. 52
4-7. Flowchart of function to require one liaison to immediately precede another. 54
4-8. Example of immediately precede function. 55
4-9. Example of function to eliminate multiple subassemblies. 57
4-10. 8-part parts tree. 59
4-11. Sequence graph for 8-part parts tree. 59
4-12. 16-part parts tree. 60
4-13. Assembly sequences of 16-part parts tree. 61
4-14. Two assembly sequences of 4-part parts tree only represent different order
of subassembly creation. 62
4-15. Six assembly sequences on 6-part parts tree can be reduced to one by
building one subassembly first. 63
4-16. Flowchart of function to remove redundant arborescent assembly sequences. 65
4-17. Example of 'x' command to eliminate redundant sequences. 66
5-1. Network for shortest path algorithm example.
5-2. Network after one iteration of shortest path algorithm.
5-3. Network after two iterations of shortest path algorithm.
5-4. Network after three iterations of shortest path algorithm.
5-5. Network after final iteration of shortest path algorithm.
5-6. State transition where one orientation change occurs.
5-7. State transition where three fixturing changes occur.
5-8. How orientations are defined for example assembly.
5-9. How fixturing surfaces are defined (named and numbered)
    for example assembly.
5-10. Two fixturing and orientation options for example assembly.
5-11. How mating surfaces are blocked from fixturing after mate is established.
5-12. Liaison diagram and part drawings for example assembly.
5-14. Example of defining additional part surfaces for fixturing.
5-15. Example of defining relevant orientations.
5-16. Specifying blocked surfaces and menu of fixturing and orientation options.
5-17. Final three fixturing and orientation options for example assembly.
5-18. State where subassembly fixturing and orientation has already been entered.
5-19. Single state expanded into three.
5-20. One state with two subassemblies expanded into six states.
5-21. Data structure of expanded sequence graph.
5-22. Example of assembly move interfering with a fixture.
5-23. Use of function to specify that a liaison interferes with a fixture.
5-24. Changing the cost to make and/or break fixture on a particular surface.
5-25. Illustration of make/break costs for all surfaces.
5-26. Flowchart for calculating cost of a transition where the parent has more
    subassemblies than the child.
5-27. Flowchart for calculating cost of a transition where the parent and child
    have the same number of subassemblies.
5-28. Flowchart for calculating cost of a transition where the child has more
    subassemblies than the parent.
5-29. Example of a state's K shortest subpaths table.
5-30. How K shortest subpath table is calculated from parent states.
5-31. Use of K shortest path function.
5-32. Example of how to view individual paths.
5-33. First state of individual path.
5-34. Second state of individual path.
5-35. Final state of individual path.
5-36. Illustration of array shortest path concept.
5-37. Example array for array shortest path algorithm.
5-38. Use of array shortest path function.
5-39. Illustration of command to find all paths of a given cost.
5-40. Example of how to view individual paths.
5-41. How orientations and fixturing surfaces are defined for AFI example.
5-42. Example array for AFI example.
6-1. Example how a gripper is used to make an assembly move.
6-2. Example where the difficulty of attaching the gripper depends on the fixturing of the subassembly.
6-3. One unexpanded state transition into thirty-six expanded state transitions.
6-4. Eliminating unnecessary expanded state transitions leaving twelve of original thirty-six.
C-1. Example EXPN.DAT file.
C-2. How to execute EDIT.
C-3. How to save current sequence graph data.
C-4. How to retrieve previously saved files.
D-1. Drawing an assembly in SUNDRAW.
D-2. Saving a part as a raster image.
D-3. How to specify the raster files in EDIT.
1.0 INTRODUCTION

Manufacturing is becoming increasingly important to the success of products. Product designs must bring together more complicated technologies with greater precision. Historically, designers considered only functionality when developing new products. It was up to the manufacturing engineers to figure out how the designs would be produced. Unfortunately, factors such as fabrication, assembly, maintenance, and robustness are largely determined by the product design. If these ideas are not considered until after the product has been designed, then very little can be done about them. These are the problems we try to avoid through the practice of concurrent design. Early in the development process, designers must understand how each design decision will affect all the parameters of the product not just its functionality.

One of the most important consequences of the design of a product is how it will be assembled. Assembly is where parts first come together to perform a function. It is here that poorly fitting parts and poorly thought out designs are exposed. Designers often do not fully consider the relationship between parts and how they will be assembled. So assembly must be carefully considered during the design of a product to ensure the product's success. Identifying assembly as the key integrative activity in manufacturing is the idea behind the Strategic Approach to Product Design [1], [2] where assembly is the focus of concurrent design.

An important parameter of an assembly system is the choice of assembly sequence. In addition to influencing the cost and equipment selection of the assembly system, the choice of assembly sequence influences part orientation and fixturing requirements. It also determines whether particular subassemblies will be used which influences where in-process testing of components can be done. The choice of assembly sequence also affects the overall system layout and thus influences material and labor requirements.
Because the choice of assembly sequence is determined by the product design, the designer must consider assembly sequences as part of the design process. Deriving the optimal assembly system for a product from the product's design has not been done. To make the task possible as part of the design process, the designer needs software tools that help do the job. These software tools should enable the designer to generate all possible assembly sequences and then help him to edit the sequences to find those that will lead to the best possible assembly system for the product. The designer will then be more able to optimize a product for assembly as well as for functionality.

The first component of an assembly sequence design tool is the generation of all possible assembly sequences. We must be able to consider all possible assembly sequences before we can guarantee to find one that is optimal. Traditionally, industry has relied on first choosing a base part then examining ways to complete the assembly by adding successive parts or subassemblies. This idea does not consider all assembly sequences and thus may overlook good sequences. Now methods by Bourjault [3], De Fazio and Whitney [4], and Homem De Mello [5] are available which enable a designer to generate all possible assembly sequences for a given product.

After all the possible assembly sequences are generated, the designer should have software tools that will help distinguish good sequences from bad ones. Most products have many thousands of possible assembly sequences, so it is usually difficult to manually select the best candidates. Work has already been done to address this problem. Lui's assembly sequence generation software [6] allows the user to eliminate all sequences that contain multiple subassemblies. This powerful constraint usually reduces the number of assembly sequences by an order of magnitude or more. However, blindly eliminating all sequences that have multiple subassemblies is not necessarily a good way to find the best assembly sequences, and when multiple subassemblies are eliminated, there can still be thousands left to consider.
Lui's software also provides a way to edit the assembly sequences directly by eliminating single assembly states or state transitions. Editing individual states and state transitions is a powerful way to reduce the number of sequences. Unfortunately, this leaves it up to the designer to use all the knowledge he has about the assembly to find bad assembly states and to decide whether these states should be eliminated from consideration. New editing tools should help the designer find states and transitions that are undesirable, and when editing states and transitions is not practical or desirable, tools should be available that allow evaluation of actual assembly sequences.

Ultimately, the best assembly sequence is the one that minimizes total product cost. We have no good ways to measure total product cost, but if an easy way to calculate unit assembly cost were available, a computer could just pick the one sequence that has the lowest assembly cost. Unfortunately determining unit assembly cost is very difficult too. Software tools such as Assembly System Design Program (ASDP) [7] can design a nearly optimal assembly system, but they need an extensive set of data about assembly machines and operations as well as the particular assembly sequence that was chosen. So the designer must narrow the choice of assembly sequences before one of these tools can be conveniently used. New tools would be useful if they would help the designer bridge the gap between all possible assembly sequences and a few "good" ones that can be considered more rigorously.

In many assembly processes changing the orientation or fixtures of the assembly is an expensive operation. Changing fixtures usually requires using different grippers or sometimes even different machines. The extra fixtures automatically increase costs. Each new orientation may require a new fixture, changing orientation can risk part damage. Both of these operations at least use up some of the cycle time in the operation. So identifying sequences that have excessive reorientations and fixture changes may be a good way to eliminate undesirable assembly sequences.
1.1 Goals of Thesis

The goal of this thesis is to create a software tool that allows us to interactively edit assembly sequences and evaluate assembly sequences based on fixturing and orientation requirements. Through a program called EDIT, we have achieved this goal. We can edit assembly sequences with a variety of editing functions based on precluding individual states of assembly and individual assembly moves. We can evaluate assembly sequences through the use of a shortest path approach for finding assembly sequences that require relatively few fixturing and orientation changes.

We begin editing with a representation of all possible assembly sequences for a given product. We can then use our software to view the states of assembly and the assembly moves. Some of the editing tools we can apply include precluding all instances of a particular assembly move that we consider undesirable, constraining the assembly process to a sequential assembly line, and forcing one assembly move to occur immediately before another. These tools are based on evaluation of individual states of assembly and individual assembly moves. We have the capability to compare actual assembly sequences using requirements for fixturing and orientation.

To evaluate assembly sequences based on fixturing and orientation, we must first enter feasible fixturing and orientation combinations for all the possible subassemblies of the product. The software can then determine where all the changes in fixturing and orientation occur. A shortest path algorithm can then be used to find assembly sequences that require the fewest fixturing and orientation changes.

1.2 Background on Liaison Sequence Analysis

Liaison Sequence Analysis (LSA) is a methodology for evaluating assembly sequences based on connections between parts instead of just the parts themselves.
Bourjault called the connection between parts a liaison, and he represented an assembly as a liaison diagram where parts are nodes and the connections between parts are arcs. An example assembly and corresponding liaison diagram are shown in Figure 1-1.

FLASHLIGHT ASSEMBLY DRAWING

FLASHLIGHT LIAISON DIAGRAM

Figure 1-1. Example Assembly and Liaison Diagram

1.2.1 Generation of Precedence Constraints

Using the idea of a liaison as a mate between parts and the liaison diagram as a network representing an assembly, Bourjault formulated an algorithm for generating all
possible assembly sequences. He made the assumption that parts are rigid and that parts
mate are unique geometric relationships between parts. His algorithm was then based on
the following pair of questions:

Question 1: Can liaison i be established if liaisons j-k have already been
established? (Liaison i is one of the assembly's liaisons, and
liaisons j-k are one or more of its remaining liaisons.)

Question 2: Can liaison i be established if liaisons j-k have not already been
established?

Answering these two questions for all liaisons i and all combinations of liaisons j-k
will yield all the precedence constraints of the assembly. The precedence constraints can
then be used to generate all the possible assembly sequences. A precedence constraint is a
constraint on the ordering of assembly operations. Precedence constraints can be expressed
in mathematical form as precedence relations. The following expression is an example of the
kind of precedence relations that Bourjault's algorithm generates:

Liaison 1 >= Liaison 2 and Liaison 3

This states that liaison 1 must precede or concur with liaisons 2 and 3. The operator
"must precede or concur with", represented as >=, is elicited by answering no to question
1. Answering no specifies that liaison i can never follow liaisons j-k, therefore it must
precede liaisons j-k or occur simultaneously with them.

De Fazio and Whitney realized the usefulness of Bourjault's method, but decided
that the number of questions that had to be answered was too high for typical products. In
response to this, they proposed a different set of questions as follows:

Question 1: What liaisons must be done prior to doing liaison i?
Question 2: What liaison must be left to be done after doing liaison i?

These questions, instead of having yes-no answers, elicit logical expressions like:

- Liaison 1 and Liaison 2 or Liaison 3 and Liaison 4 \(\geq\) Liaison 5
- Liaison 3 and not Liaison 2 \(>\) Liaison 1

Where the operator, \(\geq\), means "must precede or concur with", and the operator, \(>\), means "must precede".

The advantage of this approach is that if only required answering 2N questions where N is the number of liaisons in the assembly. For some assemblies, however, the questions can be difficult to answer.

Bourjault also realized that the question count was an important constraint in his method. He introduced some simplifications that greatly reduced the number of questions that needed to be asked. His simplifications were based on the fact that: 1) his second question was generally unnecessary, and 2) the answers to many questions can be inferred from answers to previous questions. Inferring the answers of questions is explained by the so-called subset and superset rules. The subset rule states that if two subassemblies can be assembled, then after removing any parts from either subassembly they can still be assembled. For Bourjault's question 1 this means that if liaison i can be established in the presence of liaisons j-k, then liaison i can be established in the presence of any subset of liaisons j-k. So any question that is answered yes can automatically be answered yes for any subset. The superset rule states that if two subassemblies cannot be assembled, then after adding parts to either subassembly they still cannot be assembled. This means that for question 1, if liaison i cannot be established in the presence of liaisons j-k, then liaison i cannot be established in the presence of any superset of liaisons j-k. So any question that is answered no can automatically be answered no for any superset. These simplifications
brought Bourjault's question count down to a reasonable number while maintaining the advantage that the questions require only yes-no answers.

Another methodology of determining precedence constraints was developed somewhat in parallel to Bourjault by Homem De Mello. Homem De Mello used the same concept of asking questions based on part mates, but instead of asking questions about various combinations of liaisons, he looked at cut-sets of the liaison diagram. This idea alone limits the question count to the number of cut-sets in the liaison diagram. Homem De Mello did not utilize the subset or superset simplification rules, and he did not explicitly derive precedence relations, but he did, however, use information about part mating directions to help answer the yes-no questions.

Baldwin [8] has combined these ideas, Bourjault's simplifications and Homem De Mello's cut-sets and mating conditions, into a software tool called SPAS that derives precedence relations. His approach also utilizes heuristics for determining the order of asking questions and part drawings to illustrate the questions. All of these ideas combined into one tool make generating precedence constraints a fast and relatively straight-forward process.

1.2.2 Representing Assembly Sequences

Along with the various methods to generate precedence constraints, there are different ways to represent the possible assembly sequences of a product. One of the earliest used ways to represent assembly sequences is the parts tree. Figure 1-2 shows the liaison diagram and part drawings of a ball-point pen assembly. Figure 1-3 shows the parts trees that represent all assembly sequences of the pen. Bourjault uses the parts tree representation in his assembly sequence generation software [9]. This representation is interesting because it shows the arborescence of each assembly sequence, but many
different parts trees are generally necessary to represent all assembly sequences of a product.

![Diagram of ball-point pen assembly](image)

Figure 1-2. Ball-point pen assembly drawing and liaison diagram.
Figure 1-3. Parts trees representing all assembly sequences of ball-point pen.
Homem De Mello uses an And-Or Graph to represent assembly sequences. The And-Or Graph representation of the ball-point pen assembly sequences is shown in Figure 1-4. This representation establishes each possible unique subassembly of the product as a node. Arcs between nodes then indicate how it is possible to combine subassemblies in assembly sequences. This representation is compact, but does not easily convey the state by state completion of assembly. The And-Or Graph is also difficult to automatically draw because of the complex arrangement of nodes and arcs.

De Fazio and Whitney introduced the idea of using a directed graph called a Liaison Sequence Diagram or Sequence Graph to represent assembly sequences. In this representation the boxes depict states of assembly where the completed liaisons in the assembly are shown as shaded squares in each box. Figure 1-5 shows the Liaison Sequence Diagram for the ball-point pen example. The sequences proceed rank by rank from the top disassembled state to the final fully assembled state. Because each state can consist of more that one subassembly, the number of states is greater than the number of unique subassemblies. So this representation may not be as compact as the And-Or Graph, but it does more obviously show how each assembly sequence progresses from beginning to end by establishing more and more part mates.
Figure 1-4. And-Or Graph representation of ball-point pen assembly sequences.
1.2.3 Evaluating Assembly Sequences

Through Liaison Sequence Analysis it has become easier to generate all possible assembly sequences for products. Tools such as Assembly System Design Program have been devised which help design and optimize assembly sequences after the actual sequence of assembly is chosen. There obviously must be some transition between generation of all possible assembly sequences and rigorous evaluation of a single assembly sequence. Some work has been done to automate the selection of "good" assembly sequences from all possible sequences.
In his program, SED (for Sequence Edit and Display), Lui allowed the user to delete individual states of assembly and individual transitions. He also provided a way to display the entire sequence graph as well as individual states of assembly. In Lui's software the user also had the option of constraining the assembly to a sequential process by deleting all states of assembly that contained more than one subassembly. These basic editing features eased the process of narrowing the choice of assembly sequence. Still more editing was made possible through the use of non-geometric precedence relations [10]. One could specify that certain combinations of mates will not occur simultaneously or require that a particular liaison immediately precede another by adding special precedence relations to Lui's software.

One of the goals of Liaison Sequence Analysis is to provide a way to evaluate assembly issues of a product early in design. For this to be successful, the tools for performing Liaison Sequence Analysis must be fast and easy to use. Evaluating and editing all the possible assembly sequences to identify assembly problems and to find the "best" assembly sequences is an important part of Liaison Sequence Analysis, and the work described in this thesis was intended to improve the automation of this process.
2.0 Overview of Current Software System

The current software system consists of a computer program called EDIT that allows the user to interactively edit assembly sequences and evaluate assembly sequences based on fixturing and orientation requirements. EDIT was written in C [11] on a SUN 3/60 computer workstation. Through the use of the SUNVIEW [12] graphics package, EDIT displays the sequence graph as well as drawings of assembly parts generated as raster images using SUN's drawing program, SUNDRAW [13].

EDIT fits into a larger Liaison Sequence Analysis software package as shown in Figure 2-1. SPAS allows us to conveniently enter assembly precedence constraints which are processed by PRED [14] and compiled into LSG. LSG then generates all possible assembly sequences in the form of a liaison sequence diagram. This liaison sequence diagram, along with raster images of the part drawings, is then input to EDIT.

The two main features of EDIT are: 1) viewing and editing assembly states and state transitions; and 2) a pair of shortest path algorithms that allow the user to evaluate assembly sequences based on fixturing and orientation requirements. Figure 2-2 shows the overall structure of the program.

EDIT's screen display is shown in Figure 2-3. Up to four windows are used to display information and gather input from the user. The Unexpanded Sequence Graph Window holds the sequence graph display and a menu panel for a few of the most important editing features. The Text Command Window receives all text input and displays printed information. The Parts Window displays the part drawings of states that we select, and both the Parts Window and the Parts After Transition Window display part drawings of the two states before and after transitions that we select.
Figure 2-1. How EDIT fits into Liaison Sequence Analysis Software.
Figure 2-2. EDIT overall program structure.

We can perform a variety of editing features based on states and state transitions. The most basic of these is eliminating individual states or state transitions. The menu items above the sequence graph in Figure 2-3. allow us to invoke several editing features. The
Figure 2-3. Example of EDIT's screen display.
"SHOW" item gives us the ability to view particular states or state transitions by clicking on them with the mouse. If we choose the "SHOW STATE" option, we can click on a state to see the following information about the state: 1) its part drawings in the Parts Window, 2) a text description in the Text Command Window, and 3) the state and all its parents and children highlighted in green in the Unexpanded Sequence Graph Window. If we choose the "SHOW TRANSITION" option, we can click on the two states associated with a transition to highlight the transition in green and to bring up part drawings of the states before and after the transition. After we have highlighted a state or state transition, we can delete the state or transition by selecting "DEL SHOWN".

If we wish to simply delete particular states or transitions without showing them first, we can use the "DELETE" option. This option has the following four choices: 1) "DELETE STATE", 2) "DELETE TRANSITION", 3) "UNDELETE STATE", and 4) "UNDELETE TRANSITION". Selecting one of these items will activate the mouse to perform the indicated function on selected states. Deleting an item will cause the item to be highlighted in red. Invoking the "REDRAW" item allows us to remove all the deleted items from the display.

EDIT allows us to perform several more complicated editing features by entering text commands in the text window. These editing features allow elimination of all assembly states that have multiple subassemblies, elimination of all state transitions where a particular set of simultaneous mates is made, or specifying that a particular assembly move must immediately precede another. These features are powerful editing tools; invoking them can quickly bring the number of sequences down to a reasonable few.

If we cannot sufficiently narrow the choice of assembly sequence using state and transition based editing, then we need a way to edit or evaluate individual assembly sequences. EDIT gives us this capability by allowing comparison of assembly sequences.
based on the number of required fixturing and orientation changes. We would generally like to minimize these changes during assembly.

The program allows us to invoke two different shortest path algorithms to search the sequence graph for sequences that require relatively few fixturing and orientation changes. This shortest path approach is possible if we make the assumption that fixturing and orientation changes occur only in state transitions. Before we invoke the shortest path algorithm, we must enter possible fixturing and orientation options for each assembly state in the sequence graph. Fixturing options are entered as possible surfaces on a subassembly where fixtures can be attached, and orientation options are entered as the possible orientations of a subassembly. These orientations are taken from a finite set of feasible orientations that we define. Once these options are entered, the program generates a new sequence graph (called "expanded" in Figure 2-2) in which every set of fixturing and orientation options for a state is made into a unique state. This is necessary because the shortest path searches require that a unique fixturing, orientation pair exists for each state. Once the new state space is generated, the number of fixturing and orientation changes in each state transition can be determined, and the shortest path algorithm can find those sequences that require relatively few changes. The program will then plot the individual sequences and let us step through the part drawings of each state in a given sequence.
3.0 Sequence Graph Data Structure and Display

We chose to represent all possible assembly sequences in the sequence graph format, so the focus of our evaluation and editing of assembly sequences is on the sequence graph. Because of this, the sequence graph is the most important element of a software tool for editing and evaluating assembly sequences. This chapter illustrates how we store the sequence graph as data structures in the computer, and how we display the sequence graph.

3.1 Sequence Graph Data Structure

The sequence graph data structures in EDIT are similar to the data structures Lui used in SED. Both programs read the same input files generated by the liaison sequence generation programs, LSG and PRED. Because the sequence graph editing capabilities in EDIT are more extensive than in SED, EDIT's data structures contain more information.

In EDIT the sequence graph is stored as a hierarchy of data structures. The set of all ranks is at the top of this hierarchy. The ranks are the rows of the sequence graph. The first rank (labelled zero) contains the unassembled state, and the last rank contains the totally assembled state. The number of ranks is equal to the number of parts in the assembly, and each assembly sequence passes through each rank once and only once. Each rank structure contains a pointer to the list of states that exist in that rank. Below the state structures are the parents, children, and subassemblies. The parents of a state are the states in the previous rank that are connected to the state. Similarly, a state's children are all the states in the next rank that connect to the state. The parent and child transition structures contain information about the transitions between the states and each of their parents and children, and the subassembly structures contain information about the fixturing and orientation.
possibilities and completed liaisons of each subassembly. We refer to this set of structures as the main sequence graph data structure. It contains all the basic information about the sequence graph, and it is relatively static. When we delete states or transitions during editing, nothing is removed from the main data structure. We can only change the status of elements in the structure. This way, accessing the data is a well defined process, and it is easier to undo any mistakes we make during editing.

In addition to the main sequence graph data structure, three other sequence graph data structures that in EDIT. These are separate because they represent the sequence graph in different ways. One is for the sequence graph display, and two are for shortest path algorithms. The sequence graph display is constantly changing during the editing process. States and transitions can be removed or added to the display at any time, so a separate data structure (described in 3.2) is used to keep track of where each displayed state is located on the screen. The shortest path algorithms that find sequences with few fixturing and orientation changes require expanded versions of the main sequence graph data structure. These data structures (described in 5.1 and 5.2) are created using the main structure and are eliminated when we complete the shortest path search.

3.1.1 Rank Structure

The purpose of the rank structure is to indicate which states exist in the rank. As shown in Figure 3-1, the rank structure contains a pointer (first_state_ptr) to the first state in the rank and a count (num_states) of the number of states in the rank. The program uses this information to index through all the states in a given rank.
3.1.2 State Structure

The state structure is the most important element of the main sequence graph data structure. It contains information about each state's liaisons, status, and location in the rank. The state structure also stores the number of parents, children, and subassemblies of the state as well as pointers to each of these elements. Figure 3-2 illustrates an example state structure as well as the parent, child, and subassembly structures that belong to it.

The variables, just_est_liai_tbl, liai_tbl, and next_liai_tbl store the state's liaison information in a binary table format. just_est_liai_tbl is a table of liaisons that were completed in the transitions between the state and its parents, liai_tbl is a table of completed liaisons of the state, and next_liai_tbl is a table of liaisons that are completed in the transitions from the state to its children. The variables, col_num and disp_col_num, store the state's position in the rank. col_num is the state's permanent index in the rank of the main sequence graph structure, and disp_col_num is the state's index of position in the displayed rank. A state's disp_col_num is set to -1 when it is removed from display.
Figure 3-2. An example state structure and its parent, child, and subassembly structures.
The variable, \textit{delete\_status}, stores the current edit status of the state. During editing a state can have one of six levels of delete status which indicate whether the state has been deleted and what caused it to be deleted. If a state is on (or undeleted), its \textit{delete\_status} is 0. In Figure 3-2, we use "on" to mean 0. All states that are deleted have \textit{delete\_status} greater than 0. States that are deleted and highlighted because they depend on other deleted states have \textit{delete\_status} of 1, and states that were specifically deleted and highlighted have \textit{delete\_status} of 2. Similarly, states that are deleted and removed from display because they depend on other undisplayed states have \textit{delete\_status} of 3, and states that were specifically deleted and removed from display have \textit{delete\_status} of 4. So as the \textit{delete\_status} gets higher, the level of the delete becomes more severe. The highest delete level (when \textit{delete\_status} is 5) occurs when a state is deleted because it is a dead-end state or a dependent of a dead-end state. The sequence graph that LSG produces can sometimes contain dead-end states. These dead-end states and their dependents are not valid states because they are not part of any valid assembly sequences, so before EDIT displays the sequence graph, it sets their \textit{delete\_status} to 5 which means they will never be displayed and they will never be evaluated as valid states.

The rest of the variables in the state structure store information about the state's parents, children, and subassemblies. \textit{num\_parents} is the state's number of parents, \textit{num\_children} is the state's number of children, and \textit{num\_subasbys} is the states number of subassemblies. \textit{parent\_struct\_ptr} is a pointer to the first element of a set of parent transition structures. These parent structures contain information about each transition from the state to its parents. Similarly, \textit{child\_struct\_ptr} points to the first element of a set of child transition structures, and \textit{subasby\_ptr} points to the first of element of the state's set of subassembly structures.
3.1.3 Parent Transition Structure

Parent transition structures (as shown in Figure 3-2) are used to store information about the transitions from a state to its parent states. The structure contains three elements: 
\textit{parent\_ptr}, \textit{delete\_status}, and \textit{subasby\_tbl}. \textit{parent\_ptr} is a pointer to the data structure of the parent state that the transition is from. \textit{delete\_status} stores the delete status of the transition between the state and the parent state. This \textit{delete\_status} is similar to the \textit{delete\_status} of the state structure except that it only has three delete levels. These are: 0 for on, 2 for specifically deleted and highlighted, and 4 for specifically deleted and removed from display. We do not need to use transition \textit{delete\_status} values for dependency or dead-ends because EDIT assumes that the \textit{delete\_status} of a transition is the same as the highest delete level of the two states that it connects unless it was specifically set to a higher value. When we delete a state and remove it from display, its transitions should automatically be removed as well, so there is no need to specifically change the \textit{delete\_status} of a transition unless we want the delete level of the transition to be higher than the delete level of the states that it connects.

The element, \textit{subasby\_tbl}, is a pointer to an integer table that stores information about which subassemblies in the parent correspond to the subassemblies in the state. The table contains one integer for each subassembly in the state. Each number in the table is the index of the subassembly in the parent that corresponds to a subassembly in the state. So for example, if the second subassembly in the state is a child of the first subassembly in the parent, then the second integer in the table will be zero because that parent subassembly has an index of zero (index numbers start with zero not one). Figure 3-3 shows an example of the subassembly tables for a state with two parent states. For the case where a subassembly has no parent subassembly, the subassembly table value is set to -1, and when a
subassembly has two parent subassemblies, its table value is set to one of the two parents arbitrarily.

Figure 3-3. Example illustrating the construction of subassembly tables (subasby_tbl) in the parent transition data structures of a state with two parents.
3.1.4 Child Transition Structure

Child transition structures (as shown in Figure 3-2) are the same as parent transition structures except that they store information about the transitions from a state to its children not its parents. The three elements in the child transition structure are: child_ptr, delete_status, and subasby_tbl. child_ptr is a pointer to the child state's data structure, delete_status stores the current delete status of the transition exactly as the delete_status of the parent transition structure. subasby_tbl is a pointer to an integer table that indicates which subassemblies in the child correspond to the subassemblies in the state. The subassembly table is set up exactly like the subassembly table in the parent transition structure. The only difference between the two is that each subassembly can have one and only one child subassembly, so there is no ambiguity about whether a subassembly has more than one child, and there are no entries of -1 to indicate that a subassembly has no children. This difference exists because, once a subassembly is created, it cannot vanish from the sequence graph; it can only grow as individual parts are added or be merged into another subassembly.

3.1.5 Subassembly Structure

Subassembly structures (as shown in Figure 3-2) contain information about the individual subassemblies of states. The purpose of the subassembly structure is to store a table of the subassembly's completed liaisons and to store information about its fixturing and orientation possibilities. The following variables exist in the subassembly structure: sub_liai_tbl, fix_or_optns, num_fix_or_optns, unavlbl_surfs_ptr, and num_unavlbl_surfs. sub_liai_tbl is a binary table of completed liaisons, and the rest of the variables in the structure contain fixturing and orientation data that is explained in Chapter 5.
3.2 Sequence Graph Display

To display the sequence graph, EDIT creates a new data structure from the main sequence graph structure. This new structure, called the display data structure, holds information about where each displayed state is located on the screen as illustrated in Figure 3-4. The rank structure contains the following three elements: num_states, zi_top_loc, and first_state_ptr. num_states indicates how many displayed states are in the rank, and zi_top_loc holds the rank's position in the window. This position is the distance, in pixels, from the top of the screen to the top of the rank's state boxes. first_state_ptr is a pointer to the first displayed state in the rank.

The state structure of the sequence graph display data structure contains three elements as follows: col_num, orig_state_index, and zi_xloc. col_num is the rank index of the state in the display, and orig_state_index is the rank index of the state in the main sequence graph data structure. zi_xloc holds the state's position in its rank. This position is the pixel distance from the left edge of the display window to the left edge of the state box.

Other variables that are independent of the display data structure help determine the shape and spacing of the state boxes. These are zi_state_xlen, zi_state_ylen, zi_x_space, zi_y_space, x_liai_num, y_liai_num, zi_liai_xlen, and zi_liai_ylen. zi_state_xlen and zi_state_ylen are the x and y pixel lengths of the state box. EDIT currently keeps these values equal so that the state boxes are always square. The RESIZE BOX menu item allows us to change size of the state boxes. zi_x_space and zi_y_space are the x and y spacing (also in pixels) between state boxes. EDIT automatically sets zi_x_space to half the x-length of the state boxes, and it sets zi_y_space to twice the y-length of the state boxes. x_liai_num and y_liai_num store the number of liaison boxes in the x and y dimensions of
the state box. EDIT also currently sets these two values equal. Finally, \( zi\_liai\_xlen \) and \( zi\_liai\_ylen \) store the x and y pixel lengths of the liaison boxes.

Figure 3-4. Illustration of data structures and variables used in the display of the sequence graph.

Unlike the main sequence graph data structure, the display data structure only contains information about the states that are currently displayed. So every time a state is added or deleted from the display, the display data structure must be reconstructed. Also
every time we change the size of the graph, EDIT must reconstruct the data structure because the locations of the states change.
4.0 Sequence Graph Editing Based on States and Transitions

The sequence graph consists of states and state transitions which combine to form assembly sequences. As described by De Fazio [15], the greatest gains in narrowing the choice of assembly sequences come from eliminating states. Lesser gains come from editing transitions, and still lesser gains come from editing actual assembly sequences. This is true because there are many more transitions than states in the sequence graph, and there are many more sequences than transitions, so if we eliminate a single state we will automatically eliminate at least a few transitions and many sequences.

A significant part of the assembly sequence evaluation capability of EDIT is based on editing states and transitions. This chapter covers these editing features. Figure 4-1 shows an EDIT screen with the example assembly from De Fazio and Whitney. This example, called Assembly From Industry (AFI), is a simplified drawing of an automatic transmission assembly. There are 50,748 geometrically possible assembly sequences for this product. To maintain continuity, we will use this example throughout this chapter to illustrate the different editing capabilities.

4.1 Screen Editing: Viewing and Deleting Individual States and Transitions

Perhaps the most basic way to edit the sequence graph is to evaluate each state and transition and eliminate those that are undesirable. This is the form of editing that Lui provided in his program, SED. When we edit the sequence graph in this way, the program does not help us evaluate the desirability of individual states and transitions, it just allows us to see which parts and liaisons exist in the states and which changes occur in the transitions. In EDIT the process of manually evaluating the sequence graph is simplified by the use of part drawings and printed information for displaying states and transitions and by the use of simple mouse activated commands for deleting states and transitions.
Figure 4.1: EDIG screen display showing AFI assembly drawing.
Viewing and deleting individual states and transitions are the two main parts of the manual editing process. In EDIT we control these two functions using the mouse with the menu items above the sequence graph in the Unexpanded Sequence Graph Window, seen in Figure 4-1. The menu items SHOW, DEL SHOWN, REDRAW, and DELETE give us various viewing and editing capabilities. The menu items, SHOW and DELETE, allow us to define the current mouse selection action. So if we want to view states by selecting them with the mouse cursor, we select the SHOW STATE option under the SHOW menu selection. The test status line immediately below the menu selections describes the current mouse action item and gives instructions. In Figure 4-1, the SHOW STATE option has been selected and the status line instructs us to select states that we want to show.

There are two viewing options under the SHOW menu selection. They are SHOW STATE and SHOW TRANSITION. The SHOW STATE option allows us to select states for viewing. When we select a state for viewing by clicking on its state box in the sequence graph, EDIT will display a variety of information about the state. This information consists of a display of the state's part drawings in the parts window, a listing of text information in the text window, and highlighting (in green) of the state and its parents and children states in the sequence graph. As shown in Figure 4-2, the state indicated by the cursor (rank 9, column 6) has been selected. The highlighting does not appear in the figure, but it does show the part drawings and the state's text information. The two separate drawings in the Parts Window represent the two subassemblies of the state.

Figure 4-3 illustrates the SHOW TRANSITION option. Choosing SHOW TRANSITION allows us to view transitions by selecting the two states connected by the transition. When we select a transition, the transition and the two states it connects will become highlighted in green, and the part drawings of the two states will appear. We have placed the cursor over the transition selected in the figure because the green highlighting does not show. As in the figure, drawings of the state before the transition will appear in
Figure 4.2. Example of SHOW STATE function.

RANK 9, DISPLAY COLUMN NUMBER: 6
State Completed Liasons: 1 2 6 7 11 12 13 14 15 16 17 18
Subassembly #1: Completed Liasons: 1 2 6 7 11 12 13
Parts: number name
1  A
2  C
3  D
7  B
8  E
11  F

Subassembly fixturing and orientation options:
Fixture: Orientation:
#  Num. Name (Part and Fix. Surface) Num. Name
1  37 Part A  Fix. Front-Underside 2 T-C-Down
2  30 Part D  Fix. Engine-Face 2 T-C-Down
Subassembly #2: Completed Liasons: 14 15 16 17 18
Parts: number name
4  G
9  H
10  J
5  K
16  L

Subassembly fixturing and orientation options:
Fixture: Orientation:
#  Num. Name (Part and Fix. Surface) Num. Name
1  40 Part L  Fix. Back-End-Boss 1 T-C-Up

STATE NUMBERS OF PARENT STATES: 17 18 19 20 21
STATE NUMBERS OF CHILD STATES: 8
DELETE STATUS: on
Figure 4-3. Example of SHOW TRANSITION function.
the Parts Window, and drawings of the state after the transition will appear in the Parts After Transition Window. This allows us to visualize which part mates occur in the transition. So in the figure, part L is added during the selected transition This transition connects the state in rank 9, column 1 to the state in rank 10, column 0.

Viewing allows us to more easily search for states and transitions that are undesirable. When we find undesirable states and transitions, we need to be able to delete them from the sequence graph. In EDIT, we can delete items by choosing the DEL SHOWN or DELETE menu items. If we have selected a state or transition for viewing, and then decide that we want to delete that item, we can simply choose the DEL SHOWN menu button, and the item will be deleted. Choosing DEL SHOWN will delete the currently shown state or state transition along with all the states and transitions that depend on it. When states or state transitions become deleted, they are highlighted in red on the sequence graph to indicate that they are no longer on. If we wish to remove deleted items from the display, we can use the REDRAW menu button. Selecting REDRAW removes all the deleted states and transitions from the displayed sequence graph. So all the states and transitions that are highlighted in red will disappear from the screen when we invoke the REDRAW option.

The other way to delete items in the sequence graph is through the DELETE menu button. Four choices exist under this menu item; these choices are DELETE STATE, UNDELETE STATE, DELETE TRANSITION, and UNDELETE TRANSITION. Similar to the SHOW STATE and SHOW TRANSITION menu items, choosing one of these four items will set the mouse selection action to that item. So if we choose DELETE STATE, all the states that we subsequently select with the mouse will become deleted and highlighted in red along with all the states and transitions that depend on that state. If we wish to select a deleted state from the graph to undelete, we can use the UNDELETE STATE menu item. Choosing this item will cause all the states that we select to become undeleted unless they
are dependents of states that are deleted. The DELETE TRANSITION and UNDELETE TRANSITION menu items are the same as DELETE STATE and UNDELETE STATE except that they enable us to delete and undelete transitions by selecting the two states on either side of the desired transition.

4.2 Editing State Transitions Based on Multiple Liaison Establishment

When making assembly moves, we often establish more than one part mate or liaison simultaneously. This can happen when a single part has liaisons with several other parts or when we attach two subassemblies where several parts mate at the same time. In fact if the number of liaisons in an assembly is not less than the number of parts, then there must be at least on assembly move where more than one liaison is established. Sometimes we can determine that a particular set of liaisons should not be established simultaneously. The individual mates could require extra precision such that completing them all in one move is difficult, or a part could be so delicate that we would want to mate it to each of the connecting parts separately to avoid part damage. In EDIT we can remove particular sets of simultaneous mates from the sequence graph using the text command, 'I'.

Figures 4-4 and 4-5 show an example of this editing strategy applied to our AFI assembly. The two parts windows in Figure 4-4 illustrate a particular simultaneous liaison mate where liaisons 8, 9, and 10 are all made in one move. This happens when part B is mated to parts G, H, and J. This assembly move requires three simultaneous splined-shaft to splined-bore mates with some journal-and-bore mates. Making all these mates simultaneously is difficult because we cannot reach all the parts to rotate them into place properly. As shown in Figure 4-5, we invoke the command 'I' in the Text Command Window to delete all transitions where these liaisons are made at the same time. Choosing option number 3 under 'I' allows us to enter the liaison numbers that are not to occur simultaneously. The sequence graph displayed in Figure 4-5 is what we have left after
Figure 4-4. Illustration of simultaneous mating of liaisons 8, 9, and 10.
Figure 4.5: Eliminating the simultaneous establishment of liaisons 8, 9, and 10.
invoking the command and redraw the graph. There are now only 33,294 sequences from our original 50,748. Figure 4-6 shows a flowchart of the function that deletes transitions where particular sets of liaisons are made simultaneously.

Another feature under the 'l' command will highlight all the state transitions where more than one liaison is made. We invoke this function by selecting option number 1, and it will highlight all multiple-liaison transitions in green. This feature lets us more easily search for multiple-liaison mates that are undesirable.
Figure 4-6. Flowchart of function to eliminate specified sets of simultaneous liaisons.
4.3 Requiring One Liaison to Immediately Follow Another

Another potentially useful way to edit the sequence graph is to specify that certain liaisons must immediately follow others. It is often desirable to perform one assembly operation immediately after another even though it is geometrically possible to perform them in a different order. For example, we would usually place the the cover onto a gearbox immediately after the last component goes in to prevent particulate contamination. Specifying a strict order may also be desirable to stabilize a part immediately after it is added to the assembly, or to keep a part from being damaged.

In EDIT, option number 4 under the 'I' command in the Text Command Window allows us to perform this kind of editing. When we enter this command, EDIT prompts us for two liaison numbers. It will then enforce the rule that the first liaison must immediately precede the second. Figure 4-7 shows the flowchart for this editing function. The function first enforces that the first liaison precedes the second. Then it checks every state that contains the first liaison without the second and deletes all its child transitions where the second liaison is not completed.

Figure 4-8 illustrates an example of this editing feature applied to our AFl assembly. In that assembly, part F is a cover for the bolt on part E. We may want to put that cover on immediately after attaching part E. We can express this in terms of liaisons by saying liaison 7 must immediately precede liaison 13. This was entered as shown in Figure 4-8, and it reduced the unedited sequence graph from 50,748 sequences to 26,034. The figure shows the resulting graph after selecting REDRAW to remove the deleted states and transitions from display.
Figure 4-7. Flowchart of function to require one liaison to immediately precede another.
Figure 4-8: Example of immediately precede function.

Type 'h' to edit states based on multiple subassemblies.
Type 'r' to undelete all states and transitions.
Type 'u' to return to window (if applicable).
Type 'q' to quit the program.

Enter next desired command (h for help): h

EDIT STATES BASED ON LIAISON ESTABLISHMENT
1) Highlight all transitions where multiple liaisons are made.
2) Delete all transitions where multiple liaisons are made.
3) Delete all transitions where a specified set of liaisons are made.
4) Specify that one liaison must immediately precede another.
Type the number of the desired selection or 0 to quit: 4

Enter a liaison number followed by the liaison number that must immediately follow it: 7 13

Enter next desired command (h for help): g

Number of undeleted states = 277
Number of undeleted subassemblies = 535
Number of unique undeleted subassemblies = 64
Number of undeleted state transitions = 868
Number of undeleted sequences = 29334
4.4 Editing States Based on the Number of Subassemblies

One of the earliest methods of editing the sequence graph that De Fazio and Whitney identified was to eliminate all assembly states that contain more than one subassembly. This constrains the assembly process to a sequential line. We justify this editing strategy because in industry most assembly systems are sequential. Parts are generally added one after another to some base assembly, and if subassemblies are used, they can usually be considered a single part that is assembled on a separate line.

In Lui's software, we could invoke this option in the generation of assembly sequences. In EDIT, we have made it available at any time during the editing process. The 'm' command in the Text Command Window allows us to highlight or delete states based on the number of subassemblies. Option number 2 will highlight in green all states that have more than one subassembly, and option number 3 will highlight in green all states that do not have more than one subassembly. These options let us get a rough idea of how many sequences involve sequential assembly lines.

To delete all states with more than one subassembly, we choose option number 1 under the 'm' command. This will delete all states (and dependent states and transitions) that have more than one subassembly. This feature is a very powerful editing tool. Figure 4-9 shows our AFI example after we have deleted and removed all multiple subassembly states. The number of sequences went from 50,748 to 904. From an unedited sequence graph, this option will usually reduce the number of assembly sequences by a factor of ten or more. It is, however, a heavy-handed editing move because there may be many good assembly sequences that are not purely sequential.
Figure 4-9.
Example of function to eliminate multiple subassemblies.

Type 'k' to start a shortest paths search.
Type 'p' to print state information to a file.
Type 's' to save current sequence graph data.
Type 'g' to see current graph counts.
Type 'x' to remove 'redundant' sequences.
Type 'm' to edit states based on multiple subassemblies.
Type 'l' to edit states based on liaison establishment.
Type 'u' to undelete all states and transitions.
Type 'w' to return to window (if applicable).
Type 'q' to quit the program.

Enter next desired command (h for help): m

-----------------------------------------------
EDIT STATES BASED ON MULTIPLE SUBASSEMBLIES

1) Delete all states that have more than one subassembly.

2) Highlight all states that have more than one subassembly.

3) Delete all states that do not have more than one subassembly (except first, second, and last ranks).

4) Highlight all states that do not have more than one subassembly.

Type the number of the desired selection or 0 to quit: 1

Enter next desired command (h for help): g

Number of undeleted states = 65
Number of undeleted subassemblies = 64
Number of unique undeleted subassemblies = 64
Number of undeleted state transitions = 133
Number of undeleted sequences: 984

Enter next desired command (h for help): w
4.5 Editing Based on Redundant Arborescent Sequences

Amblard [16] studied parallel or arborescent assembly systems. He used parts trees to represent arborescent assembly sequences. Translating between the parts tree representation and the sequence graph representation caused him problems because a single arborescent parts tree often represents many sequence graph assembly sequences. For example, the 8-part parts tree in Figure 4-10 corresponds to the 80 assembly sequences in the sequence graph of Figure 4-11. The 16-part parts tree in Figure 4-12 represents almost 22 million assembly sequences partially depicted as a sequence graph in Figure 4-13.

The work by Amblard led us to conclude that it is not important to represent all the assembly sequences from the same parts tree as separate sequences in the sequence graph. All the different assembly sequences that come from the same parts tree represent only different ways to order assembly operations that occur in separate subassemblies. As illustrated in Figure 4-14, the two assembly sequences that correspond to the 4-part parts tree differ only because we can change the order in which we create the two subassemblies. We can either first create subassembly A-B, then C-D, then join the two, or we can first create subassembly C-D, then A-B, then join the two. Because creating the two subassemblies are separate operations, their order generally does not matter. So we could represent the pair of sequences as only one.
Figure 4-10. 8-part parts tree.

Figure 4-11. Sequence graph for 8-part parts tree.
Figure 4-12. 16-part parts tree.

In some cases, the order of operations between separate subassemblies could be important. We could, for example, need to measure the dimensions of one subassembly before we assembled another because of tight tolerances, or we could be assembling both subassemblies using the same machine in such a way that we would want one operation to precede another. Generally, however, even if we show only one assembly sequence associated with a given parts tree, we can always realize that the order of operations in separate subassemblies may be different. So we do not really lose any information by representing an arborescent parts tree as only one assembly sequence.

As part of his thesis, Amblard devised a way to measure the arborescence of an assembly sequence. He then came up with an algorithm that would find the most arborescent assembly sequence in a given sequence graph. Part of his algorithm would remove from the sequence graph all but one of the assembly sequences that represent a single parts tree. He called this part of his algorithm the Arborescent Purge Module. We implemented a simplified version of the Arborescent Purge Module in EDIT as the text command 'x'.
Figure 4.13. Assembly sequences of 16 part parts tree.
Figure 4-14. Two assembly sequences of 4-part parts tree only represent different order of subassembly creation.

The rationale behind Amblard's Arborescent Purge Module is as follows. If we look at any state that has more than one subassembly, we will see that in the transitions from its parent states there are assembly operations being done to each of the state's subassemblies. For example, if we look at the parts tree in Figure 4-15, the final assembly move will always be at the node labelled 3 where the subassembly ABC is joined to subassembly DEF. The six different assembly sequences for this parts tree only represent different ways to order operations between the two subassemblies. If we look at any of the assembly states that contain more than one subassembly, in the transitions leading to that state we will see operations being done to both subassemblies. If we constrain the assembly sequences so that they do operations on only one subassembly at a time, then we will have only one sequence for each parts tree. So in Figure 4-15, if we choose one of the two subassemblies (either ABC or DEF) and constrain the sequence graph so that we build that subassembly completely before we start the second, then we will be left with only one sequence.
Figure 4-15. Six assembly sequences of 6-part parts tree can be reduced to one by building one subassembly first.

Amblerd implemented this idea by finding each state that contains multiple subassemblies and enforcing that only one subassembly of the state can be involved in an assembly move from the parents to the state. This is done by deleting all parent transitions where operations are done on subassemblies other than the chosen one. This forces the assembly sequences to operate on only one subassembly at a time. For the algorithm to work, the chosen subassembly of each state has to have the fewest number of parts. Choosing the subassembly in the state with the fewest number of parts guarantees that the subassembly we choose will also be the chosen subassembly in the parent states. If a subassembly has the fewest number of parts in a state, and we enforce that parent transitions only operate on that subassembly, then that subassembly will automatically have the fewest parts in the parent states because it can only get smaller while the others remain
unchanged. Therefore the algorithm will force each arborescent parts tree to be assembled such that subassemblies are built one at a time starting with the largest and finishing with the smallest.

The difference between Amblard's Arborescent Purge Module and EDIT's 'x' command is the way we resolve ties in choosing the smallest subassembly of a state. Amblard chose to resolve ties by choosing the subassembly with the most arborescent parts tree (using his measure of arborescence). This method would simplifies his calculation of the most arborescent assembly sequence. We, on the other hand, do not need to find the most arborescent assembly sequence, so when two subassemblies have the same number of parts, our function will chose the one with the highest liaison number. Figure 4-16 shows the flowchart of the 'x' editing function.

An example use of the 'x' command is shown in Figure 4-17. The figure shows our AFI example after we invoked this option and redrew the graph. Again we started from an unedited sequence graph. The function pruned the sequence graph from 50,748 assembly sequences to only 3319. For most complex assemblies we see more than a factor of ten reduction in the number of assembly sequences using this function.
Figure 4-16. Flowchart of function to remove redundant arborescent assembly sequences.
Figure 4-17. Example of 'x' command to eliminate redundant sequences.
5.0 Evaluating Assembly Sequences Based on Fixturing and Orientation

We can evaluate and edit the sequence graph using the techniques based on states and transitions as described in the previous chapter. Another way we can evaluate assembly sequences is to examine fixturing and orientation requirements. For large or heavy assemblies or for automated assembly systems, changing fixturing and orientation is usually undesirable. This provides a way to evaluate and compare whole assembly sequences, not just states and transitions. This chapter discusses how we can enter fixturing and orientation information into EDIT and then evaluate the individual assembly sequences based on the number of required fixturing and orientation changes.

5.1 Background on Fixturing and Orientation

5.1.1 Fixturing

Fixturing or jigging refers to the gripping of parts during assembly. This usually involves holding one part steady so that other parts can be added to it. Strictly speaking each part must be held by something while it is being assembled. For example, when we put the cap on a pen we are making an assembly move. We hold both the pen and the cap in our hands to perform the task, so we could say that the hands are providing fixturing for the assembly move. The idea of fixturing that we are addressing is the need to hold one part still while another is moved into position relative to it.

It is often necessary in assembly to have one part held rigidly so that other parts can be easily attached. In automated assembly, where machines are used to perform assembly steps, the parts usually need close dimensional accuracy. This can be alleviated by the use of remote center compliances, or by specifying large clearances, but often parts must be
held in precise positions for the operations to be successful. Subassemblies must also be fixed if they are automatically transported between workstations or if they are automatically reoriented.

There are many factors that influence the choice of fixturing. First, parts that are fixed must have appropriate features that allow them to be gripped by the fixturing machines. Figuring out how to attach a machine to a part is sometimes a difficult part of planning an assembly system. This must be taken into consideration early in the design of a product. Often special jiggling features must be added to parts to make them compatible with the assembly system.

The choice of fixturing should also not interfere with the assembly process. We try not to grip a part where another part has to eventually be added, and we avoid using a part for jiggling if it has to go inside another part. To minimize the number of fixturing changes, we often choose one of the base parts or outside parts of an assembly for fixturing. Sometimes, however, assemblies may not have any base parts, or if they do the base part may not enter the assembly process right away, so fixturing changes become impossible to avoid.

Another consideration in the choice of fixturing is simply that the fixed part must usually support all the parts that are added to it. When a delicate part is being attached to a rugged part, it is a good idea to avoid using the delicate part for fixturing because the fixturing process may damage the part. Sometimes, parts in assemblies are just not capable of supporting the weight of the rest of the assembly, so they cannot be considered for fixturing.

There are many different ways to represent the fixturing of parts in an assembly state. One of the simplest representations is to just specify the part to which each fixture
attaches. To be more specific, the representation could include the part and corresponding part surface, or it could specify which particular feature on the part surface is to be gripped. To go into even more detail, the representation could include what kind of gripper is used or even how the gripper attaches to and detaches from the surface of the part.

5.1.2 Orientation

Orientation refers to the directional orientation of the parts during each assembly step. The orientation can be thought of as a direction relative to some reference orientation. The reference orientation, although arbitrary, can usually be the orientation of gravity. In order to represent any of the infinite number of possible orientations in three dimensions, each orientation would need to be defined as the way to mathematically transform from the coordinate frame fixed in the reference orientation to a coordinate frame fixed to the assembly in the specified orientation.

Fortunately for many types of products only a few orientations are important to the assembly process. If the orientation of the parts has to be controlled during assembly (as in automated assembly), then usually only a few orientations need to be considered. Part symmetry and common insertion directions allow many assembly moves to be done in the same orientation.

In deciding which orientations are important, we necessarily make judgements about how orientation affects assembly. Some of the ways that orientation affects assembly are the need to minimize part damage, constraints on space, access requirements of machines or people, and part stability. The risk of part damage during assembly is an important consideration when designing an assembly system. Certain orientations may allow parts to scrape against each other during insertion. This can damage surface finishes or leave metal shavings in the product. To alleviate this problem, orientation can be chosen

69
to make the insertion direction parallel to gravity. Orientation may also be chosen to keep delicate parts from being crushed during assembly.

Limitations on space in the factory can also influence the orientation of parts during the assembly process. If parts or machines are very tall or if floor space is limited, certain part orientations may be impossible simply because the process will not fit into the factory. Orientation may also be influenced by the need to have proper access to the parts. Workers need to be able to see the operations they are performing, and they need to have the parts and tools in convenient orientations so they can do their jobs properly. It is also desirable to be able to insert parts from the top so that gravity will aid the part mating.

Space and access influence the choice of orientation mainly because of constraints on equipment and facilities, while the issue of stability is more related to the product design. For an assembly operation to be stable, the parts brought together during the operation must stay together after the operation is complete. Stability is often a problem during intermediate states of assembly where parts may only be stacked on top of each other or loosely fitted together. Many factors like motion or vibration can make assemblies come apart, but the primary issue of stability is gravity. Gravity often forces us to orient the assembly certain ways for stability. For example, a box cannot be turned upside-down before the cover is put on because its contents will fall out.

When parts are not rigidly fastened to each other, they must be in particular orientations to be stable. This is why a box must be filled with the open end up. Gravity will pull loose parts away from each other unless they are oriented appropriately. So whether the liaison between parts is a rigid connection or just plane contact determines whether stability is an issue in this sense. Therefore, two parts that are bolted together will
generally be stable in any orientation, while two parts that just stack on top of each other can be unstable in some orientations depending on how they are fixtured.

5.1.3 The Relationship Between Fixturing and Orientation

Not only are fixturing and orientation important considerations in the design of an assembly system, but they are also closely related. Knowing which part will be jigged influences which orientations are possible, and specifying the orientation influences how jigging will be done. This is why it is important to consider both factors simultaneously when designing an assembly system.

5.1.4 How to Use Fixturing and Orientation Information

Once we decide that fixturing and orientation are important considerations in the design of a particular assembly system, we need to either identify the sequences that have few fixturing and orientation changes for further consideration or identify the sequences that have too many changes so they can be deleted. It is not sufficient to simply find the single sequence that requires the fewest fixturing and orientation changes because there are many other factors that influence our choice of assembly sequence. We need the capability to obtain a variety of information about the assembly sequences. This capability would ideally be like querying a database. Some of the kinds of queries that we might want to perform are as follows:

1) All sequences that have less than 3 fixturing changes and 2 orientation changes.

2) The 100 sequences that require the fewest total fixturing and orientation changes.

3) All sequences that avoid state X and have less than 2 fixturing changes.
Having the capability to query the sequence graph data in this way lets us use fixturing and orientation information in a reasonable way. We can consider other factors that influence the desirability of assembly sequences, instead of just selecting the sequence that has the fewest fixturing and orientation changes.

5.2 Shortest Path Approach

To evaluate assembly sequences based on minimizing the number of fixturing and orientation changes, we need to be able to determine the number of changes for a given sequence, and then we need to find those sequences that require relatively few changes. Because the sequence graph consists of a well-defined network of nodes (states) and arcs (transitions), the most obvious approach to searching for individual sequences is a shortest path algorithm. If we can determine a fixturing and orientation cost for each state transition that is not path dependent, the problem is ideal for the shortest path approach.

5.2.1 Background on Shortest Path Algorithms

The shortest path algorithm is one of the most widely used network optimization techniques. The general problem that this kind of algorithm addresses is finding an optimal path in a network where each arc has a certain length or cost. The approach is based on the idea that an optimal path consists of optimal subpaths. This means that if the shortest path between nodes A and B passes through node C, then that path consists of the shortest path from A to C and the shortest path from C to B. This idea is called the Dynamic Programming Principle or the Principle of Optimality. It allows us to find an optimal path by building on successive subpaths instead of searching all possible paths.

In the example shown in Figure 5-1, the shortest path from A to I must pass through either node B or node C because all paths from A to I must pass through B or C. If
the shortest path goes through B then it consists of the shortest path from A to B and the shortest path from B to I. Likewise if it goes through C, then it must consist of the shortest path from A to C and the shortest path from C to I. We know that the shortest path from A to B has length 1, and the shortest path from A to C is of length 2, so we can ignore the other paths from A to B and A to C. Now we can simplify the graph to that shown in Figure 5-2. Using this method we can find the shortest path to each successive node. So the next iteration would appear as in Figure 5-3, where we used the information about the shortest paths from A to B and A to C to get the shortest paths from A to D, A to E, and A to F. So as shown in Figures 5-4 and 5-5, it takes two more steps to find the shortest path from A to I.

Figure 5-1. Network for shortest path algorithm example.

Figure 5-2. Network after one iteration of shortest path algorithm.
Figure 5-3. Network after two iterations of shortest path algorithm.

Figure 5-4. Network after three iterations of shortest path algorithm.

Figure 5-5. Network after final iteration of shortest path algorithm.

The shortest path algorithm thus allowed us to build up intermediate shortest paths to find the desired shortest path from A to I. As shown in Figure 5-5, the shortest path has length 3. We can look backwards at each iteration of the algorithm to see which arcs comprise the path. The arcs are A to B (cost 1), B to D (cost 0), D to G (cost 1), and G to I (cost 1).

This example is one of the most basic shortest path problems, and it is similar to the problem of finding the assembly sequence that requires the fewest fixturing and orientation changes. We want the single shortest path from one node to another in a directed, acyclic
network where all the arcs have non-negative lengths. Dijkstra [17], Dantzig [18], Bellman [19], Ford [20], Minty [21], and Moore [22] all address this problem. Most of the solution techniques differ in the way that they build optimal subpaths. The A* Algorithm by Hart, Nilsson, and Raphael [23] even builds subpaths based on a heuristic estimate of the shortest total path length from each subpath.

There are many different kinds of problems that can be solved using shortest path techniques, and there are many different solution methods. Deo and Pang [24] classify shortest path algorithms three different ways: 1) What information needs to be found from the network? For example, we may want the N shortest paths from node A to node B, or all paths of a certain length. 2) What are the characteristics of the network being evaluated? Is it sparse or dense; is it directed? 3) What solution methods are used in the algorithm? Some algorithms solve algebraic equations, while others may execute matrix operations.

Deo and Pang present an extensive classification of shortest path algorithms. They discuss a large number of algorithms that have been implemented and list references for them. Their work shows how we can evaluate a given problem in terms of the shortest path approach and where we can look for particular solution methods that will suit our needs. In the next section we discuss issues that are important in applying the shortest path approach to the fixturing and orientation problem.

5.2.2 Applying the Shortest Path Approach to the Sequence Graph

We chose a shortest path approach because we want to find the assembly sequences that generally require few fixturing and orientation changes. Because the network and the paths that we want to examine are already well defined, the most important issue is how to define the costs or lengths in the network. Shortest path algorithms are generally formulated so that costs are incurred in the arcs of the network. In the sequence graph, it is
natural to associate fixturing and orientation changes with state transitions because the assembly moves occur in the state transitions. So sequence graph states should represent subassemblies with certain fixtures in certain orientations, and sequence graph transitions should represent assembly moves and possible fixturing and orientation changes.

Our shortest path approach is based on the fact that we can obtain information about the fixturing and orientation of each subassembly, and we can use that information to generally determine where fixturing and orientation changes occur. So, for example, if a subassembly is fixed to a particular surface in one state, and in the next state it is fixed to a different surface, then we know that a fixturing change had to occur in the transition between the two states. Likewise, if a subassembly is oriented differently in the following state, then an orientation change had to occur in the transition.

So we can calculate the fixturing and orientation costs of each transition using the fixturing and orientation information about the states. The orientation cost is simply the number of orientation changes that occur in a transition or, put another way, the number of subassemblies that undergo orientation changes during the transition. Figure 5-6 illustrates a state transition where one orientation change occurs. Subassembly AD changes orientation to be mated to subassembly BC.

Fixturing cost is slightly more complicated because changing fixturing involves removing one fixture and attaching another. So we define the fixturing cost of a transition as the number of fixtures that are made or broken in the transition. So in Figure 5-7, the fixturing cost is 3 because the fixtures on part A and B in the parent state must be broken, and the fixture on part C in the child state must be made. Section 5.4 contains a more detailed description of the assignment of costs.
Figure 5-6. State transition where one orientation change occurs.

Figure 5-7. State transition where three fixturing changes occur.
We can generally determine the costs incurred during a state transition if we know the fixturing and orientation of the subassemblies in the states on either side of the transition. However, subassemblies can often have several reasonable fixturing and orientation possibilities. A shortest path algorithm requires each state to have only one fixturing and orientation option for each subassembly in order to calculate a cost for each transition. This requires us to either restrict each subassembly to one fixturing and orientation option or create separate states for each option. To allow the greatest flexibility, we chose the latter. So in order to implement a shortest path algorithm, we must first expand the sequence graph so that each state has only one fixturing and orientation option for each subassembly. We describe this state space expansion in more detail in Section 5.4.

Another shortest path issue related to costs is the fact that each transition has two different costs, fixturing cost and orientation cost. Most shortest path algorithms are formulated to handle only one cost. Loui [25] discusses the problem of multi-dimensional costs in shortest path algorithms. When the arcs of a network have multi-dimensional costs, the Dynamic Programming Principle is no longer valid for that network. We cannot assume that optimal paths consist of optimal subpaths because we do not know the form of our utility function relating the different costs.

In section 5.6, we discuss the implementation of two shortest path algorithms. These two algorithms differ in the way that they account for the multi-dimensional costs issue. The first and probably the simplest way to solve the problem is to evaluate the cost of each transition as the sum of the fixturing cost and orientation cost. This method assumes that we value one orientation change, or an orientation cost of one, the same as a fixturing cost of one, which is not necessarily the case. This method does, however, give us a useful way to handle the problem. The second approach almost completely sidesteps the problem of how to compare fixturing cost to orientation cost. It finds all the sequences
with given fixturing and orientation cost pairs instead of finding sequences with the least cost.

Most of the other issues in applying a shortest path approach to the sequence graph are easily resolved because the sequence graph is such a well-defined network. We know that all the paths go from the unassembled state to the fully-assembled state and pass through each rank along the way. We also have a data structure for the sequence graph where we can store fixturing and orientation information conveniently. One of the important issues in many shortest path algorithms is how to build the optimal subpaths. The sequence graph even makes this problem easy to solve. Each assembly state can only have as many parents and children as the number of liaisons in the liaison diagram. This means that for a shortest path algorithm each subpath through a given state only has that many possible extensions. How to build optimal subpaths is a problem because each node is often connected to many other nodes, so we have to be clever in choosing where to extend our search. So in the sequence graph, we can be relatively efficient if we build successive subpaths by stepping rank by rank through the sequence graph, and our calculation will be limited to performing only one operation on each state transition in the graph.

5.3 Representation of Fixturing and Orientation

The basic idea behind the shortest path approach to fixturing and orientation is that the user enters information about how each subassembly in the sequence graph can be fixtured and oriented, and we use this information to find fixturing and orientation changes. We then assign a fixturing cost and an orientation cost to each state transition and find paths of the least cost. So it is important to represent fixturing and orientation in a useful way. We want to be able to generally find sequences that require few fixturing and orientation
changes, but we do not want to enter a tremendous amount of information about the fixturing and orientation of each subassembly. This section describes how fixturing and orientation are represented as data in the program and how we go about entering necessary information.

5.3.1 Representation of Data

Between fixturing and orientation, the easiest to represent is orientation because we can make use of the fact that most products will have only a few important orientations for assembly. So instead of having to enter possible subassembly orientations mathematically, we define the orientations that we decide are important. We do this by giving each orientation an integer number and a name. So, for example, we may decide that the product in Figure 5-8 has two important orientations. As shown, we could define these two orientations as Up and Down.

Orientation # 1: Up  Orientation # 2: Down

![Diagram of orientations]

Figure 5-8. How orientations are defined for example assembly.

Fixturing is more complicated because there are many different ways it can be represented. We would like to represent fixturing as simply as possible to simplify the task of entering fixturing and orientation data. We would also, however, like to be as accurate and realistic and possible which leads us to a more complicated representation. As a compromise, we decided to represent fixturing as the surface of the part where a fixture
attaches. Through the liaison diagram, we already have information about parts, so we just need to extend this data to store information about surfaces on parts that may be used for fixturing.

EDIT stores information about fixturing surfaces as the name of each surface, an integer number representing the surface, and the number of the part that contains the surface. In addition, it also stores information with each part about what surfaces have been defined for the part. As with orientation, it is up to the user to decide how surfaces should be defined and what they should be named. Figure 5-9 shows how we might define the surfaces for the example product of Figure 5-8 (its part drawings and liaison diagram are shown in Figure 5-12).

Figure 5-9. How fixturing surfaces are defined (named and numbered) for example assembly.
5.3.2 Associating Fixturing and Orientation With Assembly States

Once we define all the important orientations and fixturing surfaces for each of the parts in an assembly, we must enter information about how each state of assembly can be fixed and oriented. Because fixturing and orientation are interrelated, we enter the information as pairs. So if we defined fixturing surfaces and orientations as in Figures 5-8 and 5-9, the subassembly in Figure 5-10 might have the following possibilities for fixturing and orientation:

![Figure 5-10. Two fixturing and orientation options for the example assembly.](image)

1) Fix to part A, outside surface in the Up orientation
2) Fix to part C on the top surface in the Down orientation

In defining the fixturing and orientation options for this state like this, we are specifying that we can attach fixtures to the outside surface of part A and to the top surface of part C. If we fix to A, the open end must be up to keep C from falling out, while if we fix to C, the open end must be down to keep A from falling off.

As we enter these fixturing and orientation pairs for each subassembly, they are stored in the main sequence graph data structure in the subassembly structures under each
state. As shown in Figure 3-2, the variable `fix_or_optns` is a pointer to a list of integer
fixturing and orientation pairs. The first element of each pair is the number of a fixturing
surface, and the second is the number of an orientation. So the example subassembly in
Figure 5-10 would have the following fixturing and orientation list: `((11,1), (13,2))`. The
first pair is (11,1) because the outside surface of part A is fixture surface number 11, and
B-up is orientation number 1. The second pair is (13,2) because the C-D-mate surface on
part C is surface number 13 and B-down is orientation number 2.

5.3.3 Acquiring Fixturing and Orientation Data

    After we have defined all the relevant surfaces and orientations for a product, we
must enter information about how each subassembly can be fixtured and oriented. Entering
all the fixturing and orientation data into the program is a time consuming task. The user
must specify all the feasible fixturing and orientation options for each subassembly in each
state of the sequence graph. To make this process easier, we implemented a few
simplifications in the way we enter the data.

    The first simplification comes from the fact that many states in the sequence graph
share the same subassemblies. In our AFI example, there are 277 states in the sequence
graph but only 64 different subassemblies. So we can limit the data entry problem by
entering the information for each unique subassembly only once.

    The second simplification lets us choose fixturing and orientation options from a
menu instead of having to type them in directly. When entering information for a given
subassembly, the program will bring up a menu of all the surfaces in the subassembly,
each paired with all the defined orientations. We can then simply select those pairs that are
not feasible fixturing and orientation options for the subassembly instead of typing in the
feasible pairs.
The example subassembly in Figure 5-11 illustrates another way we can simplify the process of entering fixturing and orientation data. The surfaces C-D-mate and D-C-mate are completely blocked, so there is no way that we would be able to attach fixtures to them. If we knew that these two surfaces were mating surfaces between parts C and D, then we could exclude them from the menu selections and further simplify the process of entering the data. Implementing this idea requires that we know the mating surfaces between parts. In EDIT, we have done this by having the user specify which mating surfaces correspond to each liaison in the liaison diagram. Whenever a liaison is made, the program knows that the mating surfaces corresponding to that liaison are unavailable for fixturing. In the subassembly structures of the sequence graph data structure, the program stores a list of surfaces that are unavailable for fixturing. The variable `unavlbl_surfs_tbl` in the subassembly structures of Figure 3-2 is a pointer to this list. The numbers of all the mating surfaces for each of a subassembly's liaisons are put into the list. All of the surfaces in the list are then omitted from the menu of fixturing and orientation options.

![Diagram of parts C and D with surfaces blocked](image)

The mate between parts C and D blocks the surfaces C-D-mate and D-C-mate so that they cannot be used for fixturing.

Figure 5-11. How mating surfaces are blocked from fixturing after mate is established.

The idea of recognizing that some mating surfaces are unavailable for fixturing can be extended if we consider that some surfaces that are not involved in part mating may become blocked during assembly. Once a surface becomes blocked, it will continue to be blocked because adding more parts to the assembly will not uncover it. So once a surface in
a subassembly is blocked, it cannot be used for fixturing, so we can exclude it from consideration for fixturing in all the child states.

We have implemented this strategy in EDIT. Before the program presents the menu of fixturing and orientation pairs to consider for a given subassembly, it lets us specify whether any of the surfaces in the subassembly are blocked. It adds the surface numbers of these surfaces to the subassembly's unavailable surfaces table, so in that state and all its child states, those surfaces will not be considered for fixturing.

5.3.4 A Small Example

Now that we have described how fixturing and orientation information is entered into EDIT, we will go through an example problem using the assembly from the previous figures. The liaison diagram and part drawings are shown in Figure 5-12. The first step as shown in Figure 5-13 is to enter the 'e' command in the Text Command Window to specify that we want to enter fixturing and orientation information for the assembly. The program then asks us to define the mating surfaces that correspond to each liaison, as shown in Figure 5-13, and then define any other surfaces may be used for fixturing, as shown in Figure 5-14. This gives us all the surfaces defined for the product as in Figure 5-9.

![Liaison Diagram](image1)
![Assembly Drawing](image2)

Figure 5-12. Liaison diagram and part drawings for example assembly.
Figure 5.13. Example of defining part mating surfaces.

DEFINE MATING SURFACES FOR EACH LIAISON

Liaison #1 is a mate between the following two parts: A, B

Enter the name of the surface on part A that mates to part B. If you want the mate to be defined as more than one surface on part A, type 'm':A-B-mate

Liaison #2 is a mate between the following two parts: A, B

Enter the name of the surface on part B that mates to part A. If you want the mate to be more than one surface on part B, type 'm':B-A-mate

Liaison #2 is a mate between the following two parts: B, C

Enter the name of the surface on part B that mates to part C. If you want the mate to be defined as more than one surface on part B, type 'm':B-C-mate

Liaison #2 is a mate between the following two parts: B, C

Enter the name of the surface on part C that mates to part B. If you want the mate to be more than one surface on part C, type 'm':C-B-mate
DEFINE RELEVANT FIXTURING SURFACES OF PARTS

You have already entered mating surfaces for each liaison. If there are any surfaces of parts that are not mating surfaces that you may want to specify as fixturing surfaces, you should enter them here. You can define new fixturing surfaces later, but it is more convenient to enter them now.

PART: A

You have already defined the following surfaces for this part.
# Surface
1 A-B-mate
2 A-D-mate
3 A-C-mate

If there is another surface on this part that may be used for fixturing, enter a name for it (<15 chars), or type 'q' to quit:outside

PART: A

You have already defined the following surfaces for this part.
# Surface
1 A-B-mate
2 A-D-mate
3 A-C-mate
4 X

If there is another surface on this part that may be used for fixturing, enter a name for it (<10 chars), or type 'q' to quit:

PART: B

You have already defined the following surfaces for this part.
# Surface
1 B-A-mate
2 B-C-mate

If there is another surface on this part that may be used for fixturing, enter a name for it (<10 chars), or type 'q' to quit:top.
The next step is to define the orientations that we consider important in assembling this product. Figure 5-15 shows how we enter the orientations as in Figure 5-8. After we define the important orientations and fixturing surfaces, we start entering fixturing and orientation options for all the different subassemblies. The program will step through each subassembly of each state starting with rank 1. As shown in Figure 5-16, it will draw the subassembly's parts in the Parts Window and then ask us whether any of the subassembly's surfaces are blocked. The example is subassembly 0 of state 0 in rank 1, and it consists of parts C and B. Liaison number 2 is completed, so it knows that the surfaces C-B-mate on part C and B-C-mate on part B are already blocked because we specified that those two surfaces correspond to that liaison.

After we specify that all the surfaces are unblocked by typing 'c', the program brings up the menu of available surfaces paired with each orientation. As shown in Figure 5-16 there are 10 fixturing and orientation pairs that we can consider. We can then eliminate those pairs that are not feasible by typing the number as shown in Figure 5-17. Because parts C and B are not fastened, we can only fix to C in the Up orientation, and we can only fix to B in the Down orientation. So after we delete all the fixturing orientation pairs that are not feasible, we are left with what is shown at the bottom of Figure 5-17.

The program then goes on to the next subassembly, and it steps through each subassembly state by state down the sequence graph. Figure 5-18 shows what happens when we come to a subassembly for which we have already entered fixturing and orientation information. The subassembly shown is subassembly 1 of state 1 in rank 2. This is the same subassembly of parts A and D that exists in state 2, rank 1. We already entered information for the subassembly, so the program tells us what information we entered and asks whether we want the same information to apply here.
If there is another surface on this part that may be used for fixtures, enter a name for it (< 15 chars), or type 'q' to quit: q

PART: D

You have already defined the following surfaces for this part.
    1 D-C-mate
    2 D-A-mate

If there is another surface on this part that may be used for fixtures, enter a name for it (< 15 chars), or type 'q' to quit: q

DEFINE RELEVANT ORIENTATIONS

Before you can enter fixtures and orientations for this assembly, you must specify which orientations need to be considered. It will be possible to define additional orientations later, but it is more convenient to enter them now.

Enter orientation number 1 or type 'q' to quit entering information: up

You have entered the following orientations
    1 up

Enter orientation number 2 or type 'q' to quit entering information: down

You have entered the following orientations
    1 up
    2 down

Enter orientation number 3 or type 'q' to quit entering information: q
Figure 5.16. Specifying blocked surfaces and menu of fixturing and orientation options.

NEXT STATE
Enter fixturing and orientation options for state in rank 1, column B. This state consists of the following liaisons and parts:

Subassembly #1: Completed Liaisons: 2
- Parts: number name
  - 2 A
  - 3 B

NEXT SUBASSEMBLY
SPECIFY SURFACES THAT ARE BLOCKED
The following surfaces exist in this subassembly:

<table>
<thead>
<tr>
<th>surface number</th>
<th>part name</th>
<th>surface name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>B</td>
<td>B-A-mate</td>
</tr>
<tr>
<td>12</td>
<td>B</td>
<td>top</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>C-D-mate</td>
</tr>
<tr>
<td>18</td>
<td>C</td>
<td>C-A-mate</td>
</tr>
<tr>
<td>13</td>
<td>C</td>
<td>top</td>
</tr>
</tbody>
</table>

If any of them are blocked from access, either by being covered by other surfaces, or by being enclosed within other parts, indicate that by typing the number of one of the blocked surfaces (c to continue):

RANK 1, STATE B, SUBASSEMBLY #1

<table>
<thead>
<tr>
<th>Fixture Surface:</th>
<th>Orientation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (Part and Fixturing Surface)</td>
<td>Name</td>
</tr>
<tr>
<td>Part B Fix. B-A-mate</td>
<td>up</td>
</tr>
<tr>
<td>Part B Fix. B-A-mate</td>
<td>down</td>
</tr>
<tr>
<td>Part B Fix. top</td>
<td>up</td>
</tr>
<tr>
<td>Part C Fix. top</td>
<td>up</td>
</tr>
<tr>
<td>Part C Fix. C-D-mate</td>
<td>up</td>
</tr>
<tr>
<td>Part C Fix. C-D-mate</td>
<td>down</td>
</tr>
<tr>
<td>Part C Fix. C-A-mate</td>
<td>up</td>
</tr>
<tr>
<td>Part C Fix. C-A-mate</td>
<td>down</td>
</tr>
<tr>
<td>Part C Fix. top</td>
<td>up</td>
</tr>
<tr>
<td>Part C Fix. top</td>
<td>down</td>
</tr>
</tbody>
</table>

The preceding list is a list of fixturing, orientation pairs. You must decide which of these pairs are not possible fixturing and orientation options for this subassembly.

Type the number of a pair you want to delete, or type 'g'
If all the pairs are possible options, or type 'h' for help:
RANK 1, STATE 8, SUBASSEMBLY #1

<table>
<thead>
<tr>
<th>Fixture Surface:</th>
<th>Orientation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Name (Part and Fixturing Surface)</td>
</tr>
<tr>
<td>1</td>
<td>Part B</td>
</tr>
<tr>
<td>2</td>
<td>Part B</td>
</tr>
<tr>
<td>3</td>
<td>Part C</td>
</tr>
<tr>
<td>4</td>
<td>Part C</td>
</tr>
<tr>
<td>5</td>
<td>Part C</td>
</tr>
</tbody>
</table>

The preceding list is a list of fixturing, orientation pairs. You must decide which of these pairs are not possible fixturing and orientation options for this subassembly. Type the number of a pair you want to delete, or type 'g' if all the pairs are possible options, or type 'h' for help:

---

RANK 1, STATE 8, SUBASSEMBLY #1

<table>
<thead>
<tr>
<th>Fixture Surface:</th>
<th>Orientation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Name (Part and Fixturing Surface)</td>
</tr>
<tr>
<td>1</td>
<td>Part B</td>
</tr>
<tr>
<td>2</td>
<td>Part B</td>
</tr>
<tr>
<td>3</td>
<td>Part C</td>
</tr>
<tr>
<td>4</td>
<td>Part C</td>
</tr>
</tbody>
</table>

The preceding list is a list of fixturing, orientation pairs. You must decide which of these pairs are not possible fixturing and orientation options for this subassembly. Type the number of a pair you want to delete, or type 'g' if all the pairs are possible options, or type 'h' for help:

---

RANK 1, STATE 8, SUBASSEMBLY #1

<table>
<thead>
<tr>
<th>Fixture Surface:</th>
<th>Orientation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Name (Part and Fixturing Surface)</td>
</tr>
<tr>
<td>1</td>
<td>Part B</td>
</tr>
<tr>
<td>2</td>
<td>Part B</td>
</tr>
<tr>
<td>3</td>
<td>Part C</td>
</tr>
</tbody>
</table>

The preceding list is a list of fixturing, orientation pairs. You must decide which of these pairs are not possible fixturing and orientation options for this subassembly. Type the number of a pair you want to delete, or type 'g' if all the pairs are possible options, or type 'h' for help:
Figure 5-18. State where subassembly fixturing and orientation has already been entered.
5.4 Expanded Sequence Graph

To enable us to calculate unique fixturing and orientation costs for each state transition, each state in the sequence graph must have a unique fixturing and orientation option for each subassembly. Since our sequence graph can have an arbitrary number of options for each subassembly, we must expand the sequence graph by creating a separate state for each fixturing and orientation option. The new expanded sequence graph data structure holds information about the fixturing and orientation cost of each transition which is calculated using the information in the states. This section discusses how the sequence graph is expanded, how we organize the data in the expanded sequence graph, and how the costs of each transition is calculated.

5.4.1 Creating Expanded Sequence Graph Data Structure

The expanded sequence graph contains a separate state for every fixturing and orientation option in each subassembly of the sequence graph. So, as in Figure 5-19, the single state with one subassembly that has three fixturing and orientation options is expanded into three expanded states. When a state has more than one subassembly, we must create new states for every combination of fixturing and orientation options between the different subassemblies. This is shown in Figure 5-20, where the unexpanded state has two subassemblies, one with three fixturing and orientation options and one with two. This state is expanded into six states as shown that represent all the ways to combine the options between the two subassemblies.
Figure 5-19. Single state expanded into three.
Figure 5-20. One state with two subassemblies expanded into six states.

Expanding the sequence graph in this way will cause the expanded sequence graph to have many times more states than the unexpanded sequence graph. If every state in the sequence graph becomes N states in the expanded sequence graph, then every state transition in the unexpanded sequence graph becomes N^2 transitions in the expanded sequence graph. This is true because each transition that before connected one parent to one child will expand into enough transitions to connect N parents to N children. Since the number of transitions increases by a factor of N^2, the number of sequences will increase by a factor of N^{R-1}, where R is the number of ranks in the sequence graph. So if we originally
had a sequence graph with 100 states, 500 state transitions, and 10,000 sequences; and we expanded it so that each state became 5 expanded states, we would end up with an expanded sequence graph with 500 states, 12,500 transitions, and 19,531,250,000 sequences.

Because the expanded sequence graph can potentially be so large, we made the expanded sequence graph data structure as brief as possible. The data structure is shown in Figure 5-21. It consists of ranks, states, subassemblies, and transitions like the unexpanded sequence graph data structure, except that it holds much less information. We simply refer to the unexpanded data structure when we need more information. In the expanded state structure, the variable, \textit{col_num} indicates the state's position in the expanded rank. \textit{orig_state_index} is the index of the state's unexpanded state. \textit{num_parents} is the state's number of expanded parents. \textit{parent_struct_ptr} is a pointer to the state's first expanded parent transition structure, and \textit{subasby_ptr} is a pointer to the state's subassembly information. The last variable, either \textit{k_sh_path_ptr} or \textit{array_ptr}, points to the state's shortest path information. These items are discussed in section 5.5.

The expanded sequence graph data structure does not have separate subassembly structures like the unexpanded structure. The only information that is stored for each subassembly is its single fixturing and orientation option. So each state structure has a variable called \textit{subasby_ptr} that points to a table of the fixturing and orientation options for the state's subassemblies.
Figure 5-21. Data structure of expanded sequence graph.

Because the number of state transitions increases exponentially with the number of states in the expanded sequence graph, we only store each transition once as a parent instead of twice as both a parent and a child. So each expanded state contains a pointer to its parent transition structures of its expanded parents. As shown in Figure 5-21, the parent transition structure only contains a pointer to the parent state structure called parent_ptr, and an integer cost pair called cost. This cost is an array of two integers where the first integer is the transition's fixturing cost, and the second is the transition's orientation cost. One of the shortest path algorithms requires the parent transition structures to also have a variable to use as an index to information in the parent. The use of this variable, called ksp_tbl_index, is described in section 5.5.1.
5.4.2 Calculating Fixturing and Orientation Cost

Our shortest path approach is based on the assumption that we can generally determine the number of fixturing and orientation changes in a transition using fixturing and orientation information about the states connected by the transition. This assumption is generally valid because if two subassemblies on either side of a transition are oriented differently, we know an orientation change occurred in the transition; and if they are oriented the same, then it is very likely that an orientation change did not occur in the transition. Similarly, if two subassemblies on either side of a transition are fixtured differently, we know a fixturing change occurred in the transition; and if they are fixtured the same, then it is very likely that a fixturing change did not occur in the transition.

This approach is inaccurate for complicated situations where several fixturing or orientation changes occur in a transition. It is also inaccurate when two states connected by a single transition are fixtured to the same surface but are actually fixtured differently. This may happen if an orientation change requires us to use a different fixture on the same surface. Our representation of fixturing as the surface where a fixture attaches limits us in this respect. Another problem is that this approach assigns the same cost to every fixturing change and the same cost to every orientation change. We implemented two enhancements to solve a couple of these problems.

The first enhancement to our cost model solves the problem illustrated in Figure 5-22. The subassembly is fixtured to the end of part C in the states before and after the transition, so our cost calculation will assume that no fixturing changes occurred in the transition. The addition of part B, however, interferes with that fixtured surface. Clearly, any fixture attached to that surface of part C would have to be removed before adding part B and reattached afterwards. We can handle this situation through the use of the 'i'
command in the Text Command Window by specifying that making a particular liaison interferes with a particular fixturing surface. Figure 5-23 shows how we would invoke this option for our example. Mating parts B and C involves establishing liaison 2, so we specify under the 'i' command that liaison 2 interferes with fixtures to the top surface of part C. So whenever that fixturing option exists before and after a transition, the code checks to see whether liaison 2 is established during the transition. If liaison 2 is not made, the fixturing cost of the transition is calculated normally as 0 under the assumption that no fixturing changes occurred in the transition. If liaison 2 is made in the transition, the code calculates the fixturing cost as though we had to break the fixture, attach a temporary fixture somewhere else, and then reattach the original fixture. So it assigns 3 fixturing changes to the transition.

![Diagram](image)

Figure 5-22. Example of assembly move interfering with a fixture.
Figure 5.23. Use of function to specify that a liaison interferes with a fixture.
The next enhancement involves assigning different costs for making and/or breaking fixtures to different surfaces.Assigning a cost of 1 to making or breaking a fixture on every surface assumes that all these operations are equally difficult. The user may want to specify that making or breaking a fixture on one surface is more difficult than another. The program allows us to do this by assigning a make cost and a break cost to each surface. So, for example, we can tell the program that we want the cost of attaching a fixture to a particular surface to be 2. Then whenever a fixture is attached to that surface during a transition, 2 will be added to the cost instead of 1. We can perform this option using the command 'f' in the Text Command Window as shown in Figure 5-24. The program by default sets each surface's make and break cost to 1, and these costs must be integers to be compatible with our shortest path algorithms. Figure 5-25 shows how all the make/break costs are then defined for the surfaces of our example assembly.

With these two enhancements, calculating the fixturing and orientation costs of each transition becomes fairly complicated. Flowcharts of the cost calculation are shown in Figures 5-26, 5-27, and 5-28. The routine, as divided in the flowcharts, puts each transition into one of three categories which are: 1) there are more subassemblies in the parent than in the child, 2) the parent and child have the same number of subassemblies, and 3) there are more subassemblies in the child than in the parent. If there are more subassemblies in the parent than in the child, then we know that two subassemblies were joined in the transition. So two fixtures and two orientations turn into only one. If there are the same number of subassemblies in each state, then we know that a single part was added to a subassembly. To calculate the costs of the transition, we only have to compare the subassemblies between the parent and child. If there are more subassemblies in the child than in the parent, then we know that a new subassembly was created in the transition. We currently assign no fixturing or orientation cost to creating a new subassembly because we
do not know how the incoming parts are fixtured or oriented before they come into the assembly system.
Figure 5.24. Changing the cost to make and/or break fixture on a particular surface.
Figure 5-25. Illustration of make/break costs for all surfaces.
Figure 5-26. Flowchart for calculating cost of a transition where the parent has more subassemblies than the child.
Definitions:
P(i): Parent's i-th subassembly
C(P(i)): Child subassembly that corresponds to subassembly P(i) in parent
F(S): Fixturing option of Subassembly, S
O(S): Orientation option of Subassembly, S

Figure 5-27. Flowchart for calculating cost of a transition where the parent and child have the same number of subassemblies.
Figure 5-28. Flowchart for calculating cost of a transition where the child has more subassemblies than the parent.
5.5 Implementation of Shortest Path Algorithms

Once we have expanded the sequence graph into the new data structure and have calculated the fixturing and orientation costs of each transition, we are ready to implement a shortest path algorithm to find the assembly sequences with the fewest fixturing changes. In EDIT we have implemented two shortest path algorithms. We will refer to the first one as the K shortest path algorithm. It simply finds the K best paths, where the best path is the one with the lowest total fixturing cost plus orientation cost. We call the second algorithm the array shortest path algorithm. It keeps the fixturing cost and orientation cost separate and finds all the paths associated with certain fixturing and orientation cost pairs.

5.5.1 K Shortest Path Algorithm

The K shortest path algorithm is very similar to the example algorithm described in section 5.2.1. The difference is that instead of finding only one shortest path, we find an arbitrary number of shortest paths. The user enters the number of paths, K, that he wants to see, and the algorithm will find them. The algorithm evaluates the cost of each path as the sum of the fixturing costs and the orientation costs of the transitions that make up the path.

The K shortest path algorithm basically steps through each rank from the top of the sequence graph and builds up optimal subpaths at each rank. The algorithm stores, at every state, the K shortest paths to that state from the top of the sequence graph. This way, it only has to store two ranks of subpath information at any moment. It calculates the K shortest paths to each state in a rank using the information about the K shortest paths for the previous rank. After it calculates all the subpaths for all the states in a rank, it frees the information about the subpaths from the previous rank and starts on the next rank.
The K shortest subpaths to each state are stored in a table referenced by the variable, \textit{k\_sh\_path\_ptr}, in the expanded sequence graph state structure. These tables are organized as shown in Figure 5-29. There are \( K \) rows in the table, one for each of the state's subpaths, with the shortest (best) path at the top and the longest (worst) of the K subpaths at the bottom. If the state is in rank, \( R \), then the first \( R \) entries in each row are the state column numbers of the states that make up that subpath. The last two entries in each row are the fixturing cost and orientation cost, in that order, of the subpath. So at each state, we build up a list of K optimal subpaths to use in calculating the K optimal subpaths to the states in the next rank.

<table>
<thead>
<tr>
<th>EXPANDED STATE STRUCTURE</th>
<th>STATE'S SUBPATH TABLE FOR ( K = 10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank 3, State 6</td>
<td>Path Numbers:</td>
</tr>
<tr>
<td>col_num</td>
<td>Column numbers of states in path</td>
</tr>
<tr>
<td>num_parents</td>
<td>(Path Fix. Cost, Path Ori. Cost)</td>
</tr>
<tr>
<td>parent_struct_ptr</td>
<td>0: 0, 2, 11, 6 (2,1)</td>
</tr>
<tr>
<td>orig_state_index</td>
<td>1: 0, 2, 10, 6 (2,2)</td>
</tr>
<tr>
<td>subasby_ptr</td>
<td>2: 0, 3, 11, 6 (2,2)</td>
</tr>
<tr>
<td>k_sh_path_ptr</td>
<td>3: 0, 1, 1, 6 (3,1)</td>
</tr>
<tr>
<td></td>
<td>4: 0, 7, 21, 6 (2,3)</td>
</tr>
<tr>
<td></td>
<td>5: 0, 3, 1, 6 (4,1)</td>
</tr>
<tr>
<td></td>
<td>6: 0, 1, 10, 6 (3,2)</td>
</tr>
<tr>
<td></td>
<td>7: 0, 1, 11, 6 (3,3)</td>
</tr>
<tr>
<td></td>
<td>8: 0, 3, 21, 6 (4,2)</td>
</tr>
<tr>
<td></td>
<td>9: 0, 1, 21, 6 (3,3)</td>
</tr>
</tbody>
</table>

Figure 5-29. Example of a state's \( K \) shortest subpaths table.

At every state, the table of \( K \) shortest subpaths is generated from the tables in the state's parents. To generate the subpaths for a given state, the algorithm looks at the top of the table in each of its parents. The subpaths at the top of the parent tables are the best subpaths for the parents, so one of these will extend to become the best subpath to the state. To find which of these is best, the algorithm adds the total cost of each parent subpath to the cost of making the transition. The path that has the lowest sum is put at the top of the state's subpath table. Once it finds this first subpath, the algorithm has to make sure not to find that subpath again, so it keeps an index for each parent that tells which of
the parent's subpaths have already been selected as members of the state's subpaths table. This index, called *ksp_tbl_index*, is stored in the state's parent transition structure. Figure 5-30 illustrates how a table of five shortest subpaths would be generated for a state with three parent states.

![Diagram showing the generation of subpaths](image)

Figure 5-30. How K shortest subpath table is calculated from parent states.

If there is only one expanded state in the final rank, then the K shortest paths algorithm will end when it finishes the last rank. The shortest paths table of the last state contains the K shortest assembly sequences, and the program will print out the information. If there is more than one expanded state in the last rank, then the algorithm must take one more step and extract the K shortest paths from those states.
Figure 5-31 shows how we can execute the K shortest path algorithm for our simple four-part example product. We type 'k' in the Text Command Window, and it asks us for, K, the number of shortest paths that we want to find. In the example we chose a K of 5. The program then stepped through the calculation and printed out the resulting paths in a compact form. The first set of lists tell the expanded states that comprise each path (0 7 10 0 for path 0) followed by the fixturing and orientation cost of the path (0 0 for path 0). The next set of lists describes the unexpanded and the expanded state numbers of the paths. Each unexpanded state number is followed by the fixturing and orientation numbers of the state's subassemblies. As Figure 5-31 shows, there are two paths with no fixturing or orientation cost. We have several options available for viewing the sequences using the commands in Figure 5-32. The 'p' command highlights the K sequences in blue in the sequence graph. We can use the 'i' command to view and list individual paths. Figure 5-33 shows how we can use this command to see the 0th path. This path is shown in bold in the sequence graph. The blue highlighting does not appear. The sequence starts in rank 1 with parts A and D fixed to the outside of A in the up orientation. It then continues by adding part C, if Figure 5-34, then B, in Figure 5-35, keeping the same orientation and fixturing.

5.5.2 Array Shortest Path Algorithm

The second shortest path algorithm that we implemented in EDIT uses a very different approach. Instead of finding some number of shortest paths, it finds all the paths for particular costs. It thus allows greater flexibility in presenting information. Instead of building up optimal subpaths, this algorithm creates a table at each state that contains the number of ways to complete the sequence graph for each cost. The idea is illustrated in Figure 5-36 for a network where each arc has only one cost. The algorithm starts at the second to last rank and creates a table in each state that indicates the number of ways to
Figure 5.31: Use of K shortest path function.
K-SEARCH COMMAND INSTRUCTIONS:
Type 'p' to see all k paths.
Type 'i' to see an individual path
Type 'u' to unplot paths.
Type 'r' to review unexpanded state information.
Type 'g' to see current graph counts.
Type 'c' to change the size of k.
Type 'q' to quit to main command loop

Enter your next command (h for help): 

Type the number of the path you want to see:

PATH #8: FIX COST 0 AND ORIEN COST 0

RANK 0, STATE 0 DISPLAY COLUMN 0

RANK 1, STATE 2 DISPLAY COLUMN 2
Subassembly B: Fix. to: A, Surface: outside, Orientation: up

RANK 2, STATE 2 DISPLAY COLUMN 2
Subassembly B: Fix. to: A, Surface: outside, Orientation: up

RANK 3, STATE 0 DISPLAY COLUMN 0
Subassembly B: Fix. to: A, Surface: outside, Orientation: up

Do you want to step through the state part drawings
for this sequence(y/n):

Enter your next command (h for help): 

Type the number of the path you want to see:

PATH #1: FIX COST 0 AND ORIEN COST 0

RANK 0, STATE 0 DISPLAY COLUMN 0

RANK 1, STATE 0 DISPLAY COLUMN 0
Subassembly B: Fix. to: B, Surface: top, Orientation: down

RANK 2, STATE 0 DISPLAY COLUMN 0
Subassembly B: Fix. to: B, Surface: top, Orientation: down

RANK 3, STATE 0 DISPLAY COLUMN 0
Subassembly B: Fix. to: B, Surface: top, Orientation: down

Figure 5.32. Example of how to view individual paths.
Figure 5-33. First state of individual path.
Figure 5-34. Second state of individual path.
Figure 5.35: Final state of individual path.
STEP 1
Start with the second to last rank.

The table in each state box consists of ordered pairs of cost to reach the end versus the number of ways to reach the end for that cost.

The number associated with each transition is the cost of making the transition.

For the cost 1, there is one way to reach the end from this state.

For the cost 3, there are two ways to reach the end from this state.

STEP 2
Move up the sequence graph rank by rank creating cost tables for each state.

STEP 3
After reaching the initial unassembled state we are left with a table of the cost of all the sequences.

This final table tells us that there is one sequence of cost one, one sequence of cost three, and two sequences of cost four.

Figure 5-36. Illustration of array shortest path concept.
reach the end for each possible cost. By stepping rank by rank up the sequence graph, it creates the tables in each rank using the tables in the lower rank.

After the final step, we have a table in the first state that lists the number of sequences for each cost. We can then use this table, along with the tables in all the other states, to find all the paths associated with any given cost. So this algorithm requires us to make two passes through the network to find any desired paths, one to generate all the tables, and one to find the paths.

In applying this idea to our fixturing and orientation problem, we can use two-dimensional matrices instead of tables to store each state's path information. The $x$ and $y$ indices of the matrix are fixturing and orientation cost, and the matrix value at each location is the number of paths from the state to the end of the sequence graph for that cost pair. So if a state had the matrix in Figure 5-37, there would be 3 $\text{fixturing} \times \text{orientation}$s to the end from that state for 0 fixturing cost and 0 orientation cost because the entry in the matrix at 0,0 is 3.

<table>
<thead>
<tr>
<th>Fixturing Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5-37. Example array for array shortest path algorithm.

Using matrices to store path information makes the algorithm more computationally efficient. Instead of having to find and choose the best subpaths for each state, we can just
add matrices. Starting from the second to last rank, the matrix at each state is simply the sum of the matrices of all its children, each shifted by the cost of its transition. So if \( CM_i \) is the \( i \)th child's matrix, and \( FC_i \) and \( OC_i \) are the fixturing and orientation costs of the \( i \)th child transition, then the matrix at a state, \( M(x,y) \), is calculated by:

\[
M(x,y) = \sum_{i=0}^{\text{number of children}} CM_i(x+FC_i, y+OC_i)
\]

This means that if a state's child has a 2 in its 1,0 matrix location; and the transition to the child has a cost of 1,1; then the 2 from the child's matrix will go into the 2,1 entry of state's matrix. This is true because the two paths from the child of cost 1,0 will become two paths from the state of cost 2,1 because the cost of the transition is 1,1.

One of the most convenient aspects of this algorithm is the fact that we can choose the size of the matrices arbitrarily. If we make the matrices large, we will get a lot of information about the paths, but the calculation will take longer and use more memory. So if we make the matrices 2x2, for example, we will only know the number of assembly sequences associated with fixturing cost 0 or 1 and orientation cost 0 or 1. If there happen to be no sequences for these costs, we would not be able to see any sequences without rerunning the algorithm with larger matrices.

Once we have generated all the matrices for a sequence graph, we can see the number of assembly sequences that can be completed for the particular costs. In order to find the sequences of a given cost, another pass through the network must be made. This operation is in some ways the opposite of generating the arrays. We start at the top of the sequence graph and work our way down, finding the states that complete the paths of a given cost.
To execute the array shortest path algorithm, we type 'a' in the Text Command Window as shown in Figure 5-38. The program asks us the maximum number of fixturing and orientation changes that we wish to consider. This is basically asking for the size of the matrices. In the figure, we indicated that we wish to consider all paths with up to a fixturing cost of 4 and an orientation cost of 4, so the matrices at each state are 5x5. The matrix representing all assembly sequences of the desired costs is shown.

The matrix indicates that there are two sequences that have 0 fixturing and orientation cost, which is what the K shortest path algorithm told us also. As shown in Figure 5-39, we can perform a variety of functions with the shortest path information. The figure shows how we can ask to see all the paths associated with a particular cost pair. We typed the command 'p' and entered the cost (0,0), and the program printed the two paths. We can see that these two paths are the same as the two paths of the same cost found in the K shortest path algorithm in Figure 5-31. If we wish to view either of these paths in more detail, we can invoke the 'i' command. As before, this command will print a more detailed description of the sequence, and it will let us step through the part drawings of the states in the sequence. This action is shown in Figure 5-40, where we are viewing the second state of the path described in Figures 5-33 - 5-35.

5.5.3 Comparison of the Two Algorithms

Based on the information that the two algorithms provide, the array algorithm is superior to the K shortest paths algorithm. The array algorithm is much more like making database queries. It gives us a good idea of how many different paths there are for costs that we consider reasonable. The array algorithm also does not dictate how we compare fixturing and orientation costs. The algorithm is ideal for the fixturing and orientation problem because of the way it breaks the computation down into two steps, generating the
Figure 5-38. Use of array shortest path function.
RETURNING TO ARRAY SEARCH COMMAND LOOP
Enter your next command (h for help): a

Vertical Axis: Number of Fixturing Changes

Horizontal Axis: Number of Orientation Changes

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>21</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>35</td>
<td>27</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>57</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Enter your next command (h for help): h

A-SEARCH COMMAND INSTRUCTIONS:

Type 'p' to see all paths associated with a certain number of changes in fixturing and orientation.
Type 'i' to see an individual path
Type 'u' to unplot paths.
Type 'r' to review unexpanded state information.
Type 'g' to see current graph counts.
Type 'a' to display array.
Type 'c' to change array size.
Type 'w' to return to graphics window
Type 'q' to quit to main command loop

Enter your next command (h for help): p
Enter the fix. changes, orien. changes pair that corresponds to the paths you want to see: 0 0

A-State Numbers for Path #8: B 1 1 3
Unexpanded State Numbers and (Fix. Or.) Options for Path 8: B:(12, 2) B:(12, 2) B:(12, 2)

A-State Numbers for Path #1: B 7 19 0
Unexpanded State Numbers and (Fix. Or.) Options for Path 8: 2:(11, 1) 2:(11, 1) B:(11, 1)

Enter your next command (h for help):
Figure 5.40. Example of how to view individual paths.

Type the number of the path you want to see:

PATH #1 OF FIX COST 0 AND ORIEN COST 0
RANK 0, STATE 0 DISPLAY COLUMN 0
RANK 1, STATE 2 DISPLAY COLUMN 2
Subassembly 0: Fix. to: A, Surface: outside, Orientation: up
RANK 2, STATE 2 DISPLAY COLUMN 2
Subassembly 0: Fix. to: A, Surface: outside, Orientation: up
RANK 3, STATE 0 DISPLAY COLUMN 0
Subassembly 0: Fix. to: A, Surface: outside, Orientation: up

Do you want to step through the state part drawings for this sequence (y/n): y

RANK 0, STATE 0 DISPLAY COLUMN 0

RANK 0, DISPLAY COLUMN NUMBER: 0
State Completed Liaisons:
STATE NUMBERS OF PARENT STATES:
STATE NUMBERS OF CHILD STATES: 0 1 2
Type 'n' for next state (q to quit): n

RANK 1, STATE 2 DISPLAY COLUMN 2

RANK 1, DISPLAY COLUMN NUMBER: 2
State Completed Liaisons: 0
Subassembly #1: Completed Liaisons: 0
Parts: number name
4 A
1 D
STATE NUMBERS OF PARENT STATES: 0
STATE NUMBERS OF CHILD STATES: 1 2
Subassembly 0: Fix. to: A, Surface: outside, Orientation: up
Type 'n' for next state (q to quit):
matrices and finding the sequences of certain costs. The information we get from the first step happens to be very useful for our needs, and we use this information to decide how to execute the next step.

In terms of the computational and memory requirements, the two algorithms are difficult to compare. The array algorithm is much faster than the K algorithm because generating the matrices is very efficient. We cannot make a good comparison because for a given matrix size, we cannot guarantee that we will find any sequences. If the lowest total cost sequence had both fixturing and orientation costs of 5, we would have to use 6x6 matrices to find it. The K algorithm would find it even if we set K equal to 1. Knowing that there are no sequences of lower cost is useful information in itself, but the array algorithm is useful mainly because we expect the desirable paths to lie in the low-cost area of the matrix.

The array algorithm can potentially require large amounts of memory. It has to store the arrays for every state, so if the arrays are large and there are very many states, the memory requirements can be prohibitive. The K algorithm has to store at most two ranks of shortest path information, but this information can be large if we ask for many shortest paths. Comparing the memory requirements of the two algorithms is often meaningless anyway because when the expanded sequence graph becomes large compared to the unexpanded sequence graph, the memory required for the state transitions becomes the limiting factor for both of the algorithms.

5.6 The AFI Example

The AFI example from the previous chapter is much larger than the example we have been using in this chapter. In this section we illustrate how we can evaluate the AFI assembly sequences using fixturing and orientation requirements. Because AFI is a
transmission where we add parts along one axis to the top and bottom of a case, we defined
two orientations as shown in Figure 5-41. T-C-Up means that part F is on the top. Figure
5-41 also shows the names that we gave to the AFI part surfaces for fixturing
considerations. The AFI assembly has 64 different subassemblies in its sequence graph.
Entering all the fixturing and orientation information took about two hours.

Figure 5-42 shows the final matrix after invoking the array algorithm, and as in the
small example assembly, there are two paths with no fixturing or orientation cost. Figure 5-
42 also lists the two paths of cost (0,0). These paths both represent sequences where we
begin the assembly sequence fixtured to part L in the T-C-Up orientation. They both then
build the assembly by adding individual parts from the top starting with parts J and H. The
two sequences are identical except that the order of adding parts G and K is reversed. This
example shows the convenience of the array algorithm. Instead of having to evaluate
thousands of assembly sequences, we can use the results of the algorithm and restrict our
consideration to those sequences in the top left corner of the matrix.
Figure 5-41. How orientations and fixturing surfaces are defined for AFI example.
Enter next desired command (h for help): a

Expanding A-State Space

Finished Expanding

Enter the maximum number of refixturgings that you want to consider: 4

Enter the maximum number of reorientations that you want to consider: 4

MAKING ARRAYS FOR EACH STATE

Vertical Axis: Number of Fixturing Changes
Horizontal Axis: Number of Orientation Changes

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>48</td>
<td>256</td>
<td>354</td>
<td>452</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>84</td>
<td>962</td>
<td>1378</td>
<td>1298</td>
</tr>
<tr>
<td>3</td>
<td>58</td>
<td>588</td>
<td>4954</td>
<td>5284</td>
<td>5752</td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td>630</td>
<td>4034</td>
<td>18138</td>
<td>9578</td>
</tr>
</tbody>
</table>

Enter your next command (h for help): p

Enter the fix. changes, orient. changes pair that corresponds to the paths you want to see: B B

A-State Numbers for Path #0: 8 7 15 21 29 9 18 18 7 1 3
Unexpanded State Numbers and (Fix. Or.) Options for Path
8: 0: (48, 1) 20: (48, 1) 39: (48, 1) 49: (48, 1) 18: (48, 1) 8: (48, 1) 7: (48, 1) 0: (48, 1) B: (48, 1) 0: (48, 1)

A-State Numbers for Path #1: 8 7 15 22 29 9 18 18 7 1 3
Unexpanded State Numbers and (Fix. Or.) Options for Path
8: 0: (48, 1) 20: (48, 1) 49: (48, 1) 42: (48, 1) 18: (48, 1) 8: (48, 1) 7: (48, 1) 0: (48, 1) B: (48, 1) 0: (48, 1)

Enter your next command (h for help):
6.0 Recommendations for Future Research

In this chapter, we discuss ways to enhance and extend the research described in this thesis. This discussion will concentrate on the two main topics of the thesis, state and transition-based editing and evaluating assembly sequences based on fixturing and orientation requirements.

6.1 Directly Entering Precedence Relations

Precedence relations are powerful assembly sequence editing tools. In addition to representing all the physical assembly constraints, many different editing moves can be stated in terms of precedence relations. A useful addition to EDIT would be to allow the user to type in precedence relations interactively. The current software package requires us to regenerate the sequence graph using PRED and LSG every time we want to add precedence relations. The main difference between precedence relations and the different editing functions in EDIT is that precedence relations require the code to evaluate logical expressions that the user inputs. The logical operators, and, or, and not must be considered as well as the different precedence operators, must precede, must precede or concur with, and possibly must immediately precede. Functions for evaluating and imposing precedence relations in the sequence graph have already been written for the codes PRED and LSG. The main issue in applying this work to EDIT is making the process interactive.

6.2 General Logical Functions

Once we implement functions for precedence relations that can interactively evaluate logical expressions based on the completed liaisons of assembly states, we could easily add new editing functions. These new functions could evaluate not only states but state transitions and subassemblies based on their completed parts as well as liaisons. The user
may want to make general queries like: delete all states that contain this logical combination of parts, or display all the transitions where this logical combination of liaisons is made. This idea is even easier to implement than precedence relations because it would not involve the use of precedence operators.

The sequence graph data structure in EDIT stores bit tables of the completed liaisons of each state and subassembly. If we added a bit table representation of the parts that exist in each state and subassembly, we could write general functions that evaluate logical expressions using bit tables and apply them to either liaisons or parts. That way we could write the functions to evaluate either parts or liaisons of states or subassemblies. By performing a bitwise exclusive or operation on the bit tables of the states connected by a transition, we can have a bit table representation of the liaisons completed or the parts added during the transition. We could then also use the same logical functions to evaluate state transitions. These general queries that the user could enter would then have the form: (delete or display) all (states, transitions, or subassemblies) where (some logical combination) of (parts or liaison) are made.

6.3 Undoing Editing Moves

One of the inconveniences of the current software is that, because of all the different editing features, it is not necessarily easy to undo editing operations. We can delete and undelete individual states and state transitions using mouse actions. Other editing functions can delete many states and transitions. Often a single state may be affected by several of these editing moves, so if we want to undo a move, we cannot simply undelete all the states that the move deleted.

There are a few different strategies we could use to minimize this problem. One is to keep track of all the different editing moves that have been performed in some kind of table. This would be similar to the Delete List in Lui's SED program. He kept a list of all
the states that had been deleted. If we stored a table of all the edits we perform we could undo any particular editing move by starting over with the unedited sequence graph and reapplying all the entries in the table except the one we did not want. The difficulty in applying this approach here is that there are many different types of editing, and we would have to be able to represent all the different editing functions and the parameters that go with each.

A simpler way to allow undoing of editing moves is to store only the changes made by the most recent move. We could have a table that stores all the states and transitions that were deleted by the last edit. So if we wanted to undo that operation, the program would simply have to undelete all the entries in the table.

6.4 Improving Fixturing and Orientation Algorithm

6.4.1 Attaching Fixturing and Orientation to Individual Parts

The current software does not represent the fixturing and orientation of individual parts as they are added to the assembly. So the fixturing and orientation costs of each transition do not include the possibility that individual parts may need to be refixtureed or reoriented when they are first added to the assembly. We originally intended fixturing to represent the only base assembly that is being held rigidly while parts are added. If we wish to extend this representation to include the way we may fixture individual parts as they are being added to the assembly, we need to associate fixturing and orientation options to the individual parts. Another reason to associate fixturing and orientation with individual parts is that a part in the liaison diagram is sometimes actually a subassembly that is represented as a single part for simplicity. If we know that a single part is actually a subassembly, then we should be able to represent the fixturing and orientation of the subassembly.
We could easily enhance the existing software to have the capability of associating fixturing and orientation information to parts. When two parts come together to form a subassembly, or when a single part is added to the assembly, we could calculate an additional fixturing and orientation cost of the transition due to the refixturing or reorientation of the individual parts. It may be true that certain parts are manually assembled onto the base assembly so that the fixturing and orientation of the parts is not important. We should continue to have the option of specifying that parts are "free" to assemble in terms of fixturing and orientation.

6.4.2 Including Grippers as well as Fixtures

Our software currently models how assemblies are fixtured and where fixturing changes are necessary. Figure 6-1 illustrates the problem of gripping parts during part mating. In order to assemble subassembly B-C to subassembly A-D, we would probably grip the subassembly on the top of part C. Depending on how the subassembly is fixtured and oriented in the parent state, as shown in Figure 6-2, the problem of attaching this gripper may be easy or difficult. If the subassembly is fixtured to the gripping surface, we may need a temporary fixture or gripper to hold it when we detach the fixture.

![Figure 6-1. Example how a gripper is used to make an assembly move.](image)
Figure 6-2. Example where the difficulty of attaching the gripper depends on the fixturing of the subassembly.

The problem of modelling part mating and gripping is much more difficult than fixturing. The computer would have to handle more complicated issues like part trajectories and the stability of parts in motion. There could, however, be ways to put some of the general issues of gripping into the model without requiring too much information.

6.4.3 Expansion of Sequence Graph

The example in Figure 5-20 shows how we must expand multi-subassembly states so that unique states are created from every combination of fixturing and orientation options of the different subassemblies. This is necessary so that the shortest path algorithms
consider all the possible assembly sequences. When states have several subassemblies or when each subassembly has many fixturing and orientation options, the number of expanded states created from each unexpanded state becomes very large. The number of state transitions increases by the square of the number of the number of states. Figure 6-3 illustrates this situation. One state transition between states with two subassemblies becomes 36 expanded state transitions because each state becomes 6 states.

**SINGLE UNEXPANDED TRANSITION**

**PARENT STATE**
- Subassembly 0
  - Fix., Or. Options: (4,2), (7,1)

**CHILD STATE**
- Subassembly 0
  - Fix., Or. Options: (1,1), (9,1), (3,2)
- Subassembly 1
  - Fix., Or. Options: (4,2), (11,1)

**6 EXPANDED PARENTS**

<table>
<thead>
<tr>
<th>Sub 0: (4,2)</th>
<th>Sub 0: (4,2)</th>
<th>Sub 0: (4,2)</th>
<th>Sub 0: (7,1)</th>
<th>Sub 0: (7,1)</th>
<th>Sub 0: (7,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub 1: (1,1)</td>
<td>Sub 1: (9,1)</td>
<td>Sub 1: (3,2)</td>
<td>Sub 1: (1,1)</td>
<td>Sub 1: (9,1)</td>
<td>Sub 1: (3,2)</td>
</tr>
</tbody>
</table>

**36 EXPANDED TRANSITIONS**

**6 EXPANDED CHILDREN**

Figure 6-3. One unexpanded state transition expanded into thirty-six expanded state transitions.
We cannot eliminate any of the expanded states without losing information about possibly desirable assembly sequences, but we can eliminate some of the transitions if we make use of the fact that in every state transition there is only one assembly move. In our example, the one assembly move involves adding part C to subassembly 0. Since no assembly move is performed on subassembly 1, it is exactly the same subassembly before and after the transition, and it should have the same fixturing and orientation options. If no assembly move is performed on subassembly 1 in the transition, then there is no reason to allow any fixturing or orientation changes to occur in the transition. In the figure, this means that we can eliminate all the transitions where a fixturing or orientation change is made to subassembly 0. So each set of 6 transitions for each state can be reduced to 2 as shown in figure 6-4.

<table>
<thead>
<tr>
<th>6 EXPANDED PARENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub 0: (4,2)</td>
</tr>
<tr>
<td>Sub 1: (1,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12 EXPANDED TRANSITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub 0: (4,2)</td>
</tr>
<tr>
<td>Sub 1: (1,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 EXPANDED CHILDREN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub 0: (4,2)</td>
</tr>
<tr>
<td>Sub 1: (1,1)</td>
</tr>
</tbody>
</table>

Figure 6-4. Eliminating unnecessary expanded state transitions leaving twelve of original thirty-six.

This situation is similar to Amblard's Arborescent Purge Module described in 4.5. It is possible to change the fixturing or orientation of a subassembly without performing any assembly moves, but there is no reason to represent this in our software. For sequence
graphs with many multi-subassembly states, this simplification could greatly reduce the number of expanded state transitions. This, in turn, would then reduce the calculation time and the memory requirements of both the shortest path algorithms.

6.4.4 Other Improvements

One way that we can improve the way that EDIT compares the different assembly sequences is to change the costs associated with transitions so that they calculate more than just the number of fixtureing and orientation changes. If we knew more information about the parts and mates, we could assign more or less difficulty to different assembly moves given the weight, size, or part count of the assembly involved. If we could continue to evaluate the cost of each transition as an integer, we could continue to use the array algorithm which allows us great flexibility in the amount of information we obtain and the time needed for different calculations.

There are some other ideas that may be useful additions to the fixtureing and orientation model in EDIT. Some of these ideas are multiple fixtures on a single subassembly and temporary fixtures. It is sometimes necessary to have fixtures attached to more than one part in order to keep all the parts stable. The current software assumes that there is only one fixture for each subassembly, and that each fixture only attaches to a single part. It is possible to work around this problem for some examples. We may, for example, want a fixture to attached to two parts simultaneously so that we fixture to one part and simply stabilize the other. We could represent this in the software by specifying that the subassembly is stable fixtureed to the first part. This would allow us to consider that fixtureing option in an assembly sequence without modelling the bridging fixture.

Modelling multiple fixtures would not be impossible to add the the existing code. We would no longer have only fixtureing and orientation pairs. It would complicate the calculation of fixtureing costs, and it could make the process of entering fixtureing and
orientation information difficult. Temporary fixtures are difficult to model because we do not necessarily know when they are needed and when they must be removed. Probably the best approach to these issues is to study more actual assembly systems to see where they are important and how they affect assembly systems.

6.5 Acquiring Fixturing and Orientation Data

The most serious drawback of our method of evaluating sequences based on fixturing and orientation is the time needed to enter fixturing and orientation data. This section discusses some ways to improve this process.

6.5.1 Automatically Generating Data

Stability is one of the most important ways that fixturing and orientation affects assembly. We often fixture and orient assemblies in certain ways because they are unstable otherwise. If we could calculate whether a subassembly is stable given its fixturing and orientation, we could automatically eliminate some of the work needed to enter data. Of course, we would still have to define possible fixtures and orientations for an assembly in order to know what choices to eliminate.

There are many different approaches to finding whether a subassembly is stable given its fixturing and orientation. One is to use a solid model to determine whether unfastened parts can be removed in the direction of gravity, or whether unfastened parts have components of their removal directions in the direction of gravity.

Another way to determine whether a given subassembly is stable is to establish some sort of stability requirements of each part in terms of more basic information about the parts. For example, we know that if a part is mated to another part through some sort of fastening, the two parts should be stable. We can also determine some aspects of stability by knowing what types of features are involved in part mating. Using all this information,
we could evaluate each subassembly given its fixturing and orientation and try to determine whether it is stable. If we can conclude that the subassembly is not stable, then we can eliminate that fixturing and orientation combination from consideration.

Another issue in addition to stability that influences how parts can be fixtured and oriented is the insertion direction. We often want parts to be assembled so that the insertion direction is parallel to gravity. We also often want parts to be added from above. If we knew the insertion direction of parts and knew how these directions related to the different orientations and how they related to gravity, then we could constrain fixturing and orientation by specifying how the parts should be inserted. We could state, for example, that a certain part must be inserted from above. Knowing the insertion direction and the orientation relative to gravity would allow us to eliminate any assembly moves where that part is added to the assembly when it is in the wrong orientation.

All these ideas for automatically eliminating particular fixturing and orientation possibilities because they are unstable or because they have undesirable mating directions depend on first having some idea of how the parts can be fixtured and oriented. We could potentially make use of assembly features and geometry to help determine this information. If we knew, for example, which features are generally used for fixturing, the computer could look for those features as possibilities for fixturing. We could also have the computer generate possible orientations based on insertion directions and surface normals.

6.5.2 Entering Fixturing and Orientation Data

There are a few ways that we can improve the way that we enter the fixturing and orientation information. One issue that we could use to make the process easier is that some parts and part surfaces would never be considered for fixturing. The code currently makes us define all the mating surfaces for all the parts. It is likely that we would never consider some of these surfaces for fixturing. We should be able to specify at any time that we do
not want particular surfaces or even all the surfaces on particular parts to be considered. So from that point on, we would not have those surfaces included in our choice of fixturing and orientation options. We should also be able to delete particular states or subassemblies during the process of entering the fixturing and orientation data. During this process we are more likely to realize that particular subassemblies are undesirable.

Another way that we could improve the process of entering fixturing and orientation information is to graphically illustrate the different orientations and fixturing surfaces in the part displays. The part drawings in EDIT are bit maps, so we cannot separate the different surfaces, and we cannot easily display the parts in different orientations. By using a more sophisticated graphical representation of the parts, we could graphically illustrate fixturing and orientation. There are several different levels of complexity that we can consider for representing part geometry starting with two-dimensional stick drawings and going all the way to a features-based solid model.

The SUNVIEW graphics package that we use in EDIT to create all the windows and to draw the sequence graph and part drawings has only limited graphics capabilities. Drawing functions in SUNVIEW do not include objects like arcs, circles, or polygons; so we are limited to using lines. Even with the limitations, it could be used to represent a two-dimensional representation of parts. One way is to specify each surface of the part as a line similarly to the way we defined surfaces for our example product in Figure 5-9. If we represented each surface as a line, we could draw it in any 2-D orientation. We could also draw it in any color and attach graphical representations of fixtures to it.

One of the problems with providing a graphical part description is where the part drawings come from. Presently, we have the user draw each part in SUNDRAW and save it in a raster format with a file name that the program can recognize. If we wanted to represent some of the geometry of a part (like surfaces) in this manner, we could not save
the parts as raster images. One alternative is to read the object output of the drawing program. Every drawing program has a file format for all of its graphical objects. If we understood this format for the basic objects like lines, rectangles, and polygons, we could read the object files from the drawing program and create these objects as lines in SUNVIEW.

Probably a better approach to the problem is to use a more powerful graphics package like SUN's SUNCORE [26]. More powerful graphics packages allow us to actually define graphical objects that we can easily rotate and translate on the screen. These objects are also generally easy to manipulate using a mouse, which is not the case if we represent surfaces as lines. If we used a more powerful graphics package, we could still read the object files of drawing programs to obtain the part drawings.

A solid model would be the ideal representation of parts. Part surfaces would actually be represented as surfaces in the solid model. A features-based solid model would be even better. We could use knowledge about features to help determine fixturing options. The problem with solid models is that they are difficult to manipulate programmatically, and it is difficult to create the solid model for products, especially in the early design phases.

6.6 Other Evaluation Criteria

There are many other reasons for selecting an assembly sequences besides the number of fixturing and orientation changes required. One of these is the need to minimize the actual number of fixtures used. It is important to know how many fixtures are used in addition to the number of times they have to be changed. The current software could be enhanced so that it calculated the number of fixtures used in a given sequence as well as the number of fixturing and orientation changes.
If more information is known about the parts and the assembly process, we could include even more ways to evaluate and compare different assembly sequences. If we knew more information about geometry and part tolerances, we could propagate tolerances from the fixturing surface to the mating surface to help decide what kind of accuracy is necessary for the mating process.
7.0 Conclusions

The program EDIT provides significant improvements to our capabilities to evaluate and edit assembly sequences. We have closely integrated the editing process with assembly sequence generation tools. Several editing features that were usually difficult to perform in the past have been implemented as simple interactive commands. These include 1) precluding particular multi-liaison mates, 2) precluding multi-subassembly states, 3) enforcing that a certain liaison immediately precede another, and 4) elimination of redundant arborescent sequences. The implementation of part drawing displays for visualization of assembly states and transitions proved to be a significant improvement to the assembly sequence editing process.

The algorithms to find assembly sequences that require few fixturing and orientation changes allows us to relatively easily compare all the individual assembly sequences based on one criterion that we consider important. Implementation of the search algorithms requires us to enter general fixturing and orientation information about the states of assembly. Using some basic assumptions about when fixturing and orientations occur, we can determine a fixturing cost and an orientation cost of each state transition. Then two different shortest path algorithms allow us to search for those sequences that have few enough fixturing and orientation changes that they can be further evaluated.

Some of the important issues involved in the thesis were how to represent the sequence graph as data structures, how to model the fixturing and orientation of subassemblies, how to most easily determine where fixturing and orientation changes occur in assembly sequences, and how to organize the sequence graph for shortest path searches. The software tool described in this thesis represents a significant part of a powerful integrated software package for optimizing the choice of assembly sequence.
8.0 REFERENCES


Appendix A. Data Structure Declarations and External Variables

Most of the data structures described in this thesis are declared in the file COMMON.H which is listed below. Many of the important external variables used in EDIT are defined in the file COMMON.H which is also listed below.

```
/**-------------------------------------------------------------**/
/** File common.h updated 6/13/89 */
/**-------------------------------------------------------------**/
#define TRUE 1
#define FALSE 0
#define ON 1
#define OFF 0

/** SEARCH STATUS DEFINITIONS **/
#define SON 0 /*use state in all sequence searches */
#define AVOID 1 /*state is avoided because it depends on avoided states*/
#define FIXORAVOID 2 /*state is avoided because of fix. or. options */
#define SPECAVOID 3 /*state specified to be avoided in sequence searches*/

/** DELETE STATUS DEFINITIONS **/
#define DON 0 /*state is not deleted */
#define DELETE 1 /*dependency forces state to be deleted and hilited */
#define SPECDELETE 2 /*state is specified to be deleted and hilited*/
#define DOFF 3 /*dependency forces state to be deleted and removed from display*/
#define SPECDOFF 4 /*state is specified to be deleted and and removed from display*/
#define DEADOFF 5 /*state is deleted and removed because it depends on dead end state*/

#define LSIZE 2 /*LSIZE is the number of unsigned short integers used */
    /*for bit tables (TABLE)*/
```
typedef unsigned short int TABLE;

typedef struct subasby_structure {
    TABLE sub_liai_tbl[LSIZE]; /* table of established liaisons in subassembly */
    int num_fix_or_optns; /* number of fixturing and orientation options */
    int *fix_or_optns; /* pointer to table of fixturing and orientation options */
    int num_unavbl_surfs; /* number of fixturing surfaces that cannot be used */
    int *unavbl_surfs_ptr; /* pointer to list of unavailable fixturing surf's */
} SUBASBY, *SUBASBYPTR;

struct state_structure; /* This establishes state_structure as a pointer to a */
    /* structure so it will be legal before it is defined */

typedef struct parent_pointer_struct {
    struct state_structure *parent_ptr; /* pointer to parent state */
    int *subasby_list; /* pointer to a list that contains the */
        /* subasby numbers in the parent state that correspond */
        /* with those in the current state */
    int delete_status;
    int search_status;
} PARENT, *PARENTPTR;

typedef struct child_pointer_struct {
struct state_structure *child_ptr; /* pointer to child state */
int *subasby_list;              /* pointer to a list that contains the */
    /* subasby numbers in the child state that correspond */
    /* with those in the current state */
int delete_status;
int search_status;
} CHILD, *CHILDPTR;

typedef struct state_structure {
    TABLE just_est_liai_tb[LSIZE]; /* table of just established liaisons */
    TABLE liai_tb[LSIZE];           /* table of established liaisons in state */
    TABLE next_liai_tb[LSIZE];     /* table of next prospective liaisons */
    PARENTPTR parent_struct_ptr;    /* pointer to first of a list of parent */
        /* state pointer structures */
    CHILDPTR child_struct_ptr;      /* pointer to first of a list of child */
        /* state pointer structures */
    struct state_structure *next_state; /* pointer to next state in rank */
    int delete_status;              /* delete status of state: DON, DELETE, ... */
    int search_status;              /* search status of state: SON, AVOID, ... */
    int col_num;                    /* column number of the state in the rank */
    int disp_col_num;               /* column number in display */
    int num_parents;                /* number of parent states for the state */
    int num_del_parents;            /* number of deleted parent states */
    int num_children;               /* number of child states for the state */
    int num_del_children;           /* number of deleted child states */
    int num_subasbys;               /* number of subassemblies in the state */
    SUBASBYPTR subasby_ptr;         /* pointer to state's first subassembly */
} STATE, *STATEPTR;

typedef struct rank_structure {

148
```c
STATEPTR first_state_ptr; /* pointer to the first state (index = 0) in rank */
STATEPTR first_disp_state_ptr; /* pointer to first displayed state in rank */
int num_states; /* number of states in rank */
} RANK, *RANKPTR;

struct k_expanded_state_structure; /* these two lines establish the expanded states as */
struct a_expanded_state_structure; /* structures so they will be legal before defined */

typedef struct k_parent_pointer_structure {
    struct k_expanded_state_structure *parent_ptr; /* pointer to a parent expanded state */
    int cost[2]; /* fixting, orientation costs for this transition */
    int ksp_tbl_index; /* index of a position in the parent's path data */
} KPARENT, *KPARENTPTR;

typedef struct a_parent_pointer_structure {
    struct a_expanded_state_structure *parent_ptr; /* pointer to a parent expanded state */
    int cost[2]; /* fixting, orientation costs for this transition */
} APARENT, *APARENTPTR;

typedef struct k_expanded_state_structure { /* for k-shortest paths structure */
    int col_num; /* column number (index) of state in rank */
    int num_parents; /* number of expanded parents of state */
    KPARENT *parent_struct_ptr; /* ptr to parent structs */
    int orig_state_index; /* index in rank of unexpanded (original) state */
    int *subasby_ptr; /* pointer to list of subassembly fix. options */
    int *k_sh_path_ptr; /* pointer to list of k-shortest paths data */
} KSTATE, *KSTATEPTR;

typedef struct a_expanded_state_structure { /* for array (all) paths structure */
    int col_num; /* column number (index) of state in rank */
    int num_parents; /* number of expanded parents of state */
    APARENT *parent_struct_ptr; /* ptr to parent structures */
    int orig_state_index; /* index in rank of unexpanded (original) state */
    int *subasby_ptr; /* pointer to list of subassembly fix. options */
    unsigned int *array_ptr; /* pointer to list of array paths data */
} ASTATE, *ASTATEPTR;
```
typedef struct k_rank_structure {
    int num_exp_states; /* number of expanded states in rank */
    KSTATEPTR first_state_ptr; /* first state in expanded rank */
} KRRANK; /* KRRANKPTR;

typedef struct a_rank_structure {
    int num_exp_states; /* number of expanded states in rank */
    ASTATEPTR first_state_ptr; /* first state in expanded rank */
} ARANK; /* ARANKPTR;

typedef struct liaison_information {
    int part_x; /* index to the array part_info[] */
    int num_part_x_surfs; /* number of surfaces on part x that mate to part y */
    int *part_x_mating_surfs; /* pointer to the list of those surfaces on part x */
    int part_y; /* index to the array part_info[] */
    int num_part_y_surfs; /* number of surfaces on part y that mate to part x */
    int *part_y_mating_surfs; /* pointer to the list of those surfaces on part y */
} LIAIINFO; /* liaisons table */

struct pixrect;

typedef struct part_information {
    char abb[MAX_ABB_LEN+1]; /* abbreviated name */
    char name[MAX_NAME_LEN+1]; /* full description */
    int num_fixtures; /* the number of different fixtures associated with the part */
    int *fix_num_ptr; /* pointer to a table of the part's fixture numbers */
    char raster_fname[MAX_RAS_FILE_LEN]; /* name of part's raster file */
    struct pixrect *raster_name;
} PARTINFO; /* part name table */

typedef struct fixture_information {
    int part_num; /* the number of the part that the fixture is on */
    char fix_name[MAX_NAME_LEN+1]; /* character description of fixture */
    int cost_to_make; /* cost to establish fixture to surface */
    int cost_to_break; /* cost to disestablish fixture to surface */
}
int num_interfering_liais; /* number of liaisons that interfere with fix. to surface */
int *interfering_liais_ptr; /* list of liaisons that interfere with fix. to surface */

} FIXINFO, *FIXINFOPTR;

/*****************************************************************************/
/*
FILE extern.h updated 5/4/89
*/
/*
*****************************************************************************/

extern int num_ranks, max_num_cols;
extern int total_num_subasbys;
extern int liai_max, part_max;
extern int num_states, num_state_trans;
extern int liai_size;
extern int x_liai_num, y_liai_num;
extern int stopped_rank, stopped_state;

extern int k_state_space_memory_flag; /* These four variables are flags */
extern int a_state_space_memory_flag; /* that indicate whether memory is*/
extern int arrays_memory_flag; /* currently allocated for certain*/
extern int k_paths_memory_flag; /* search variables. They let me */
/* know whether to free memory before starting new searches.*/

extern STATEPTR view_stp;
extern int view_mk;
Appendix B. EDIT Function Directory

Due to its size, it is not feasible to provide a listing of EDIT in this thesis. EDIT consists of 13 code files as well as the COMMON.H and EXTERN.H files listed in Appendix A. The text below lists each of EDIT’s files and the functions contained in each.

EDIT.C
change_cost_to_make_break()
input_liaison_that_interferes_with_fix()
edit_states_based_on_multiple_subsc()
delete_all_states_and_transitions()
edit_states_based_on_liaison_establishment()
int get_trans_liaisons(int *, STATEPTR, STATEPTR)

EMOD1.C
int read_lsg_data()
set_initial_memory()
free_initial_memory()
memory_err()
read_part_data()
int locate_part(char *)
int save_previously_edited_sequence_data(char *)
int read_previously_edited_sequence_data(char *)

EMOD2.C
make_subassemblies()
make_child_subasby_tbls(STATEPTR, int, STATEPTR)
make_parent_subasby_tbls(STATEPTR, int, STATEPTR)
make_state_subasby(STATEPTR)
fill_in_subasby_tbl(int, TABLE*, TABLE*)
int is_liai_connected(int, TABLE*)
int offbit(TABLE *, int)
tonbit(TABLE*, int)
EMOD3.C
int is_bit_on(TABLE *, int)
int input_new_fix_or()
int make_part_num_tbl(STATEPTR, int, int *)
print_sub_liai_parts(FILE *, STATEPTR, int, int *, int)
int find_other_subasbys(STATEPTR, int, FOUNDSUBASBY**)
int ask_to_use_other_subasbys(int FOUNDSUBASBY *, int)
copy_fix_or_options(STATEPTR, int, FOUNDSUBASBY*, int)
int get_subasby_fix_or_options(STATEPTR, int)
add_match_to_found_list(STATEPTR, int, int *, FOUNDSUBASBY**)
int is_option_already_in_table(int, int, int *)
print_fix_or_options(FILE *, SUBASBYPTR)
print_fix_or_options_no_numbers(FILE *, SUBASBYPTR)

EMOD4.C
get_mating_surfs_of_liaisons()
int add_new_fix_to_fix_info(int, int)
add_fix_surf_to_part_info(int, int)
add_mate_surf_to_liai_tab(int, int, int)
get_fix_surfs_of_parts()
get_initial_orientations()
put_unavlbl_mating_surfs_into_sub(SUBASBYPTR)
put_unavlbl_parent_fixs_into_sub(STATEPTR, int)
add_unavlbl_surf_to_table(int, SUBASBYPTR)
ask_user_for_blocked_surfaces(STATEPTR, int, int, int*)
fill_tbl_with_possible_fix_or_optns(SUBASBYPTR, int, int*)
int remove_option_pair_from_sub(int, SUBASBYPTR)

EMOD5.C
get_new_fix_option(STATEPTR)
put_new_fix_option_in_previous_states(STATEPTR, int)
print_all_current_fixtures()
print_all_current_parts(FILE *)
print_fix_or_optns_of_parent_subs(STATEPTR, int)
change_state_fix_or_info(int, int)
change_subasby_fix_or_info(int, int, int)

EMOD6.C
expand_k_state_space()
set_k_graph_memory()
k_make_parent_structs()
update_search_status()
count_number_of_expanded_states_trans_subasbys()
count_states_num_exp_states()
make_options_combinations_table()
print_k_graph_info()
count_num_k_sequences()
int get_cost_due_to_interfering_liaisons(int, TABLE *, TABLE *)

EMOD7.C
expand_a_state_space()
set_a_graph_memory()
a_make_parent_structs()
print_a_graph_info()
double count_num_a_sequences()
count_num_a_sequences()

EMOD8.C
make_arrays(int, int)
print_last_array(int, int)
int get_all_a_paths_for_this_cost(int, int, int, int, int**)
int fill_in_a_paths_table(ASTATEPTR, int, int, int, int*, int, int)
int *get_k_shortest_paths(int)
int get_next_best_path(STATEPTR, int *, int, int)

EMOD9.C
int print_state_info(FILE*, STATEPTR, int, int, int, int, int, int, int, int int)
draw_sub_parts(STATEPTR, int)
draw_single_sub_parts(STATEPTR, int)
draw_state_parts(STATEPTR)
draw_second_sub_parts(STATEPTR, int)
draw_second_state_parts(STATEPTR)
update_delete_status()
double count_num_sequences()
count_num_states_subs_trans()
remove_dead_end_states_and_dependents()
remove_redundant_sequences()

EMOD10.C
perform_a_search()
perform_k_search()
print_k_path_info(int, int*)

EMOD11.C
get_fix_or_data_from_previous_problem()
r_set_initial_memory()
r_free_initial_memory()
r_memory_err()
int r_locate_part(char *)
int read_previous_graph_to_get_fix_or_data(char *)
int are_liaison_diagrams_same()
copy_fix_or_data_from_old_to_new()
int find_and_copy_subasby_fix_or_info(int, SUBASBYPTR)

EMOD12.C
go_to_graphics_window()
static void canvas_event_proc(canvas, event)
void panel_quit_proc()
void panel_go_to_text_proc(item, event)
void panel_show_state_proc(item, event)
void panel_edit_fix_or_proc(item, event)
void panel_save_to_file_proc(item, event)
void panel_redraw_graph_proc(item, event)
void panel_avoid_proc(item, event)
void panel_delete_proc(item, event)
void panel_resize_state_box_proc(item, event)
void delete_menu_delete_state_proc(item, event)
void delete_menu_undelete_state_proc(item, event)
void delete_menu_delete_trans_proc(item, event)
void delete_menu_undelete_trans_proc(item, event)
void avoid_menu_avoid_trans_proc(item, event)
void avoid_menu_unavoid_trans_proc(item, event)
make_unexpanded_graph_structure()
draw_graph()
delete_gstate(int, GSTATEPTR)
int is_state_still_on(int, GSTATEPTR)
delete_gtrans(GSTATEPTR, STATEPTR, int, GSTATEPTR, STATEPTR, int)
plot_k_path_info(int, int*)
plot_a_path_info(int, int*)
redraw_single_state(int, int)
redraw_single_state_and_parents_and_children(int, int)
redraw_single_state_and_parents_and_children_to_color(int, int, int)
redraw_state_and_trans_to_color(int, int, int)
redraw_state_box_to_color(int, int, int)
redraw_state_trans_to_color(int, int, int)
clear_graph()
set_initial_state_box_size()
draw_part(Pixrect, int)
erase_part_window()
set_part_canvas_size()
Appendix C. Procedures for Using EDIT

EDIT receives as input either a previously saved file or a pair of files generated by
the other programs in the LSA software package. This pair of files consists of files called
by convention ZAP.DAT and EXPN.DAT. These are exactly the same files required as
input to Lui's SED program. The file ZAP.DAT contains hexadecimal information on the
sequence graph and is generated by Lui's program, LSG. The other file, EXPN.DAT
contains the part names and liaison diagram as shown in Figure C-1. The liaison diagram is
represented in the first nineteen rows of the file in the figure. The first of the three columns
in each row contains the liaison number, and the second two columns contain the part
numbers of the parts that are connected by the liaison. The last row of zeros signals the end
of the liaison data. The information following the liaison diagram contains the names of the
parts following the part numbers. The semicolon at the end of each part name represents the
end of the part name.

To execute EDIT using the LSA files, ZAP.DAT and EXPN.DAT, we simply type:
EDIT  ZAP.DAT  EXPN.DAT, as shown in Figure C-2. The program then asks whether
we want to read fixturing and orientation information in from a previous problem. It is
possible that we may enter fixturing and orientation information for a given assembly and
then find that we need to regenerate the sequence graph using different precedence
relations. Many of the subassemblies in the new sequence graph will probably have existed
in the old sequence graph, and we should be able to automatically retrieve this information
instead of having to type it all by hand into the new sequence graph. This question is
asking us whether this is the case. If we do indeed want to retrieve fixturing and orientation
information from a previous example, the program will then ask us the enter the file name
of the previously saved file. This is not shown in the figure.
The next question shown in Figure C-2 is asking us to enter information about where the part raster images are stored. As described in Appendix D, the part raster images are stored in the format: *assembly name.part name.ras*. EDIT knows the part names from the liaison diagram, so we only have to give it the assembly names. The assembly name we enter in Figure C-2 is AFI/AFI, which includes the path as well as the assembly name. This means that the part raster images are stored in a directory called AFI, directly under EDIT's directory, and their assembly name is AFI.

After we specify information about the part drawings, we are free to edit the sequence graph and enter fixturing and orientation information as we please. If we wish to save the current sequence graph data, we can simply type 's' in the Text Command
If you have entered fixturing and orientation data for this
same assembly using different assembly constraints, you can
gain some time by retrieving that data to use here.
The previous data must be from a problem that has the same
liaison diagram (identical liaison and part numbers) as the
current example.
Do you want to read data from a previous problem (y/n)?

If the parts of this assembly have been drawn using
SUNDRAW and were saved in the proper format, it is
possible to use the drawings in this program. The
drawings must exist in the current directory, and they
must be named as follows: 'assembly_name.part_name.res',
where 'assembly_name' is the name of the assembly and
'part_name' is the name of the part as it is
stored in this program.

If you want to use the part drawings, type the name
of the assembly ('n' for no): AFI/AFI
Window as shown in Figure C-3. The program will then ask us for a file name under which it will save the current sequence graph information. If we want to execute EDIT later using this saved information, all we have to do is type EDIT without the two arguments, as shown in Figure C-4. The program will then ask us whether we want to continue a previous problem. If we type 'y' as shown, it will let us type in the name of the saved file, and if we type 'n' it will ask us to type in the two LSA files. After entering the file name, the program asks us for the part drawing information, and we can continue from where we executed the save command.
Figure C-3. How to save current sequence graph data.
Do you want to continue a previously edited problem (y/n)? y
Type the name of the save file
from the previous problem: AFI.out

If the parts of this assembly have been drawn using
SUNDRAW and were saved in the proper format, it is
possible to use the drawings in this program. The
drawings must exist in the current directory, and they
must be named as follows: 'assembly_name.part_name.dra',
where 'assembly_name' is the name of the assembly and
'part_name' is the name of the part as it is
stored in this program,

If you want to use the part drawings, type the name
of the assembly (‘n’ for no): AFI/AFI

Figure C-4. How to retrieve previously saved files.
Appendix D. Creating Part Drawings

The part drawings that are used in EDIT are raster images generated using SUNDRAW. Each part has its own raster image, and the program simply overlays all the raster images of the parts in a subassembly to display the entire subassembly. The procedures for generating these raster images and storing them in a way that EDIT understands is described here.

The first step is to draw the assembly in SUNDRAW as shown in Figure D-1. We must draw the assembly in the top left corner if the SUNDRAW canvas so that it appears in the proper place in EDIT's parts windows. The assembly drawing must be less than seven inches wide and four inches high in SUNDRAW in order to fit inside EDIT's windows.

After we finish drawing the assembly, we must save each part individually as a raster image as in Figure D-2. We must leave each part in the same location in the overall drawing so that all the parts overlay properly when drawn in EDIT. We do this by simply deleting all the parts except the one we want to save. So in the figure, we have deleted all the parts except A. We then save the part in raster format by selecting the Raster option in the save menu. Each part must be saved in the following format: assembly name.part name.ras. So our example is called AFI.A.ras because the assembly name is AFI and the part name is A. When we first execute EDIT, it then asks for only the assembly name because it knows the part names from the liaison diagram. Figure D-3 shows how we enter the assembly name in EDIT. In the example, we typed AFI/AFI because all the raster images for the AFI assembly were in a directory called AFI under EDIT's directory.
Figure D-1. Drawing an assembly in SUNDRAW.
Figure D-2. Saving a part as a raster image.
If you have entered fixturing and orientation data for this same assembly using different assembly constraints, you can save some time by retrieving that data to use here. The previous data must be from a problem that has the same liaison diagram (identical liaison and part numbers) as the current example.

Do you want to read data from a previous problem (y/n)?

If the parts of this assembly have been drawn using SUNDRAW and were saved in the proper format, it is possible to use the drawings in this program. The drawings must exist in the current directory, and they must be named as follows: 'assembly_name.part_name.ras', where 'assembly_name' is the name of the assembly and 'part_name' is the name of the part as it is stored in this program.

If you want to use the part drawings, type the name of the assembly ('n' for no): AFI/AFI.

Figure D-3. How to specify the raster files in EDIT.