GENERATIVE ASSEMBLY PROCESS PLANNING

by

LUC LAPERRIÈRE, B. Eng., M. Eng.

A Thesis

Submitted to the School of Graduate Studies

In Partial Fulfillment of the Requirements

For the Degree

Doctor of Philosophy

McMaster University

January 1992

GENERATIVE ASSEMBLY PROCESS PLANNING

DOCTOR OF PHILOSOPHY (1992) (Mechanical Engineering)	McMASTER UNIVERSITY Hamilton, Ontario

TITLE:	Generative Assembly Process Planning
AUTHOR:	Luc Laperrière B. Eng. (Université du Québec à Trois–Rivières) M. Eng. (McMaster University)
SUPERVISOR:	Dr. Hoda A. ElMaraghy Professor and Director Centre for Flexible Manufacturing Research and Development
NUMBER OF PAGES:	xx, 292

ABSTRACT

The benefits of automating assembly sequence generation include: *1) insuring that no potentially good assembly sequence is overlooked, 2) reducing planning costs, 3) accelerating the analysis of the economical impact of different design solutions, 4) standardizing and improving the quality of the produced plans, and 5) contributing to achieving autonomous assembly systems.*

Previous research in assembly planning focussed on the generation and evaluation of all possible assembly plans for the product under consideration.

This thesis presents a graph–theoretic approach for simultaneously generating and evaluating products' assembly / disassembly sequence alternatives and producing an optimum assembly plan according to predefined criteria. It aims at improving the efficiency of the assembly planning process and producing optimal assembly / disassembly plans. The developed graph–theoretic approach enables the determination of assembly sequences which transform any arbitrary initial state of the product into any arbitrary final state. Practically, this means many different types of assembly problems to be handled uniformly.

A product is described in terms of its components and the assembly relationships between them. This description lends itself to a graph representation, where vertices correspond to the set C of assembly components and edges correspond to the set R of assembly relationships.

For a product with "n" components, the generation of an assembly sequence is mapped into the problem of finding a sequence of "n–1" mutually exclusive cutsets in the graph model and its subgraphs. Each cutset corresponds to a disassembly operation of the physical product which produces two smaller subassemblies.

Assembly sequences are encoded in a directed graph of assembly states, representing the search space. Geometric feasibility and accessibility constraints have been developed to help reduce this combinatorial search space. Assembly–related criteria which guide the search to an optimal solution are described. They are:

1) the number of re–orientations,
2) parallelism among assembly operations,
3) stability of the subassemblies, and
4) clustering of similar assembly operations.

Integrating the evaluation of these criteria as the search graph gets expanded, enables the direct generation of an optimal disassembly sequence of a given product with respect to these criteria. Standard search methods, including breadth first, depth first, best first, A* and hill climbing, are used to guide the search towards a single and optimal assembly sequence. The A* method can generate optimal solutions without explicitly generating the whole directed graph of assembly states.

An interactive computer tool, based on the above approach, was developed. GAPP – a Generative Assembly Process Planner uses various search methods to incrementally construct the directed graph of assembly states and generate optimal assembly / disassembly sequences. Examples of real products are included to demonstrate GAPP's use and potential for assessing assembly, disassembly, repair, maintenance, assembly of multiple products and assembly error recovery procedures.

ACKNOWLEDGEMENTS

The author would like to express his deepest appreciation for the valuable support provided by his supervisor, Dr. H. A. ElMaraghy.

Special thanks are extended to systems analyst Todd Pfaff for his programming assistance. Many thanks are also due to friends and colleagues for their advice and encouragement.

The financial support of Université du Québec a Trois–Rivières, in the form of post–graduate scholarships to the author and the research scholarship from Dr. H. A. ElMaraghy's research funds, including MRCO and NSERC, are also greatly appreciated.

This work is dedicated to my wife, Annie, whose confidence and moral support made it all possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This first chapter introduces GAPP, a Generative Assembly Process Planner developed in this thesis. The first section of this chapter gives a brief overview of process planning, as it is generally perceived in a manufacturing environment. The second section describes the increasing role of the computer in process planning and the two approaches for implementing computer–aided process planning. The third section describes the assembly planning problem in more details and outlines the particular difficulties in automating assembly planning. The fourth section summarizes the motivations for implementing GAPP. The last section is a brief overview of GAPP's structure.

1.1 PROCESS PLANNING

Process planning is the task which transforms the product's design data into the instructions from which it can be manufactured (figure 1.1). A record of the results of this transformation on paper or in computer memory is called a process plan.

Process planning generally involves the following activities:

1– identification of product components and subassemblies,

2– determination of operations and their order of execution,

3– selection of machines, tools and fixtures,

4– calculation of process time,

5– calculation of cost.

Underlying some of these activities is their impressive combinatorial complexity. Various decisions for the second and third activities above can lead to a seemingly infinite number of process plan combinations. Activities 4 and 5 are directly influenced by these various decisions. Their output also serves as a metric to assess the goodness of a particular process plan. Therefore, it is imperative that the decisions in activities 2 and 3 be the result of a careful and rational decision–making process.

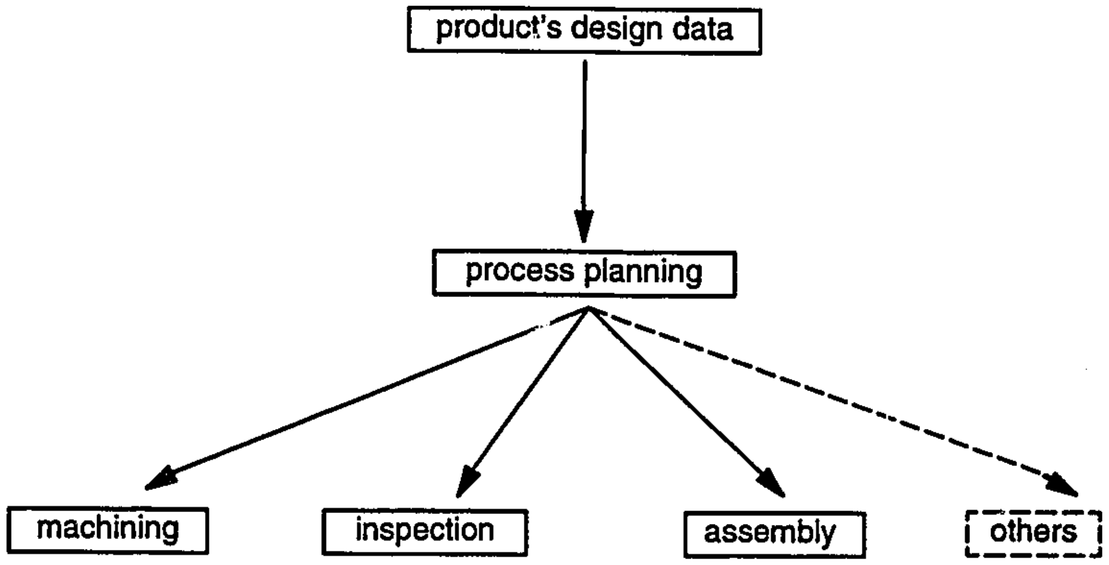


Fig. 1.1: Central role of process planning in a manufacturing environment.

The logic behind this decision–making process is very complex, due to the large size of the knowledge involved, and very subjective, due to a lack of formalization. As a result, the quality and the number of process plans generated in a plant are highly dependent upon the experience, preference and skills of the various planners who were assigned the task to generate them. This is illustrated by a striking example, provided by Groover and Zimmers [15]:

> *... a total of 42 different routings were developed for various sizes of a relatively simple part called an "expander sleeve". There were a total of 64 different sizes and styles, each with its own part number. The 42 routings included 20 different machine tools in the shop. The reason for this absence of process standardization was that many different individuals had worked on the parts: 8 or 9 manufacturing engineers, 2 planners, and 25 NC part programmers. Upon analysis, it was determined that only 2 different routings through 4 machines were needed to process the 64 part numbers.*

1.2 COMPUTER–AIDED PROCESS PLANNING (CAPP)

The recent years have seen many attempts to provide computer tools which assist planners in their complex task. The key idea is to capture part of the logic involved in the process planning activity. More rational and consistent plans can be produced as a result. These tools are generally implemented using one of the following two approaches:

1– variant process planning approach,

2– generative process planning approach.

1.2.1 Variant CAPP systems

Underlying variant CAPP systems is the concept of Group Technology (GT). This technique consists of grouping different parts into families, where each family is characterized by common design and / or manufacturing attributes of the parts within it. If the families are formed with common manufacturing attributes in mind, all the parts within a family are likely to share similar process plans. A master process plan can be associated with the family and used for any new part determined to belong to this family.

Part commonalties in a family are usually expressed in the form of a code associated with the family. When the process plan of a new part is required, a code is first determined for the part. If this code is identical to an existing family code, the master process plan for the family is retrieved and serves as the basis for establishing the process plan of the new part. If no match is found, the process plan of a family with a similar code is retrieved and adjusted to suit the new part. Two major benefits of this approach are as follows:

1– it saves the planner from always starting the process plan of a new part from scratch, as the process plan for a similar part can always be retrieved and adjusted;

2– it provides a certain level of standardization of the process plans generated, as new process plans are always constructed from previously existing ones.

On the other hand, an important drawback of the variant approach is the implicit assumption that the retrieved process plans are good ones to begin with. The underlying GT concept is also very time consuming in its implementation.

Another drawback is related to the correctness of the retrieved process plans for a new part: unless extreme care is taken when classifying and coding the parts, the retrieved process plan for a new part might be inappropriate.

The variant approach represents a significant improvement over the traditional manual process planning. It is used in many existing CAPP systems, far too numerous to be mentioned here. Nevertheless, it does not solve the root problem, i.e. it does not formalize the decision logic behind the process planning activity.

1.2.2 Generative CAPP systems

Generative automated process planning is an attempt to capture the decision logic of the human planner, formalize it, and implement it on a computer. The input to the generative CAPP system ideally contains a comprehensive model of the product. The system uses its knowledge about the process planning activity to reason upon the model and generate an optimal (or at least near–optimal) process plan automatically, in reasonable time, and without any human intervention. This approach is depicted in figure 1.2.

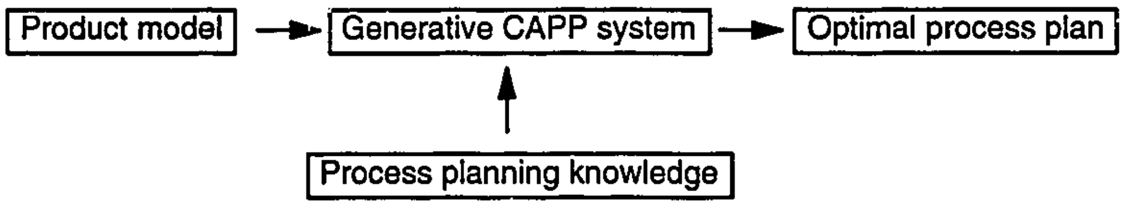


Fig. 1.2: Structure of an ideal generative CAPP system.

Such a simplistic view of a generative CAPP system hides the real complexity behind its implementation. A detailed illustration of this complexity is provided here, for the machining process of the simple plate in figure 1.3.

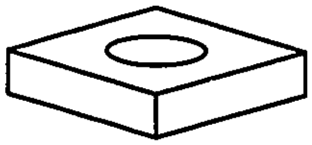


Fig. 1.3: A simple plate to be machined.

The first process planning activity consists of determining from which raw part will the plate be machined. In this case, a forging, casting or a sheet metal can be considered. The choice depends on factors such as the part's material, its dimensions and production volume.

Next comes the choice of the machining operations to be performed. If the thickness of the plate is small enough and a sheet metal is chosen for the blank, the hole in the plate can be punched. For a larger thickness the blank would probably be a casting in which case a centering and drilling operations would be more appropriate. Depending on the dimensional accuracy, surface finish and function of the hole, further operations such as internal boring, internal grinding, reaming or honing could also be added. Similar decisions on the operations to be performed are required for every other functional surface of the plate.

Once the operations have been identified, their ordering needs to be determined. For "n" operations to be performed, there are n! ways of ordering them. In practice, not all of the permutations are feasible, due to geometric and technological constraints. All such constraints must be identified and the unfeasible operation sequences eliminated. For the ones that remain, additional criteria such time and cost must be used in order to justify the choice of a particular sequence.

Next comes the choice of the machines, tools and fixtures. Assuming that the plate requires a drilling operation, a drilling machine with the proper bit and fixture need to be selected. This selection depends heavily on the size, material, accuracy and production volume of the plate, as some combinations of these parameters might be incompatible with some combinations of the drills, bits and fixtures on the shop floor. Availability is also an important decision parameter at this stage. Pieces of equipment being repaired or maintained must not be part of the selection. To complicate the problem even further, if many plates or any other part that requires drilling operations are released simultaneously for production, competition for available resources becomes important. Maximum machine utilization and avoidance of bottlenecks become a major concern.

Estimating the time required by each operation constitutes the next activity. Tool feeds and speeds need to be determined for drilling the hole. These must be carefully chosen to optimize the tool life. If the hole needs further operations such as centering or boring, tool change and setup times also become part of the equation.

The last activity consists of estimating the production cost of the part. Every decision in any of the previous activity has a direct impact on this parameter. This justifies the importance of making the right decisions at any previous stage of the process planning activity.

This simple example should suffice to show how complex the process planning activity really is, and how even more complex it becomes to formalize the decision logic behind it and implement this logic in a generative CAPP system. This is reflected by the actual state of the research in this area: most systems implemented to date simply isolate each activity and try to find optimal solutions for each one of them. Since the decisions in each activity affect each other, this approach does not guarantee optimality for the overall process plan generated. The commercial potential of generative CAPP systems is such that their development is an active area of research.

1.3 ASSEMBLY PLANNING

CAPP systems have mainly been developed for the machining manufacturing process. It is not until very recently that such systems have started to emerge for the assembly process. Some of the reasons for this are outlined below.

Any assembly operation consists of establishing the physical contacts between: a) two parts, b) a part and a subassembly or, c) two subassemblies. For the establishment of a single connection between two parts, any of the above three situations can occur, depending on the stage in the assembly sequence this connection is established. Unike machining, the characteristics[1] of an assembly operation and the resources[2] required to perform it can be very different for all three situations. This is an important source of complication in the generation of an assembly process plan.

Another reason is the larger number of parts to be treated. In machining, all the process planning activities focus on a single part or, at most, on a family of similar parts. In assembly, all the parts and subassemblies that make the product, which

1. Such characteristics include the forces involved, the stability of the subassemblies, the graspability and fixturability of the subassemblies, and so on...
2. Such resources include machines (or manual operators), tools, fixtures and fasteners.

can have quite dissimilar characteristics, have to be analyzed simultaneously in the generation of the assembly process plan.

Still another reason is the complexity of most assembly tasks. To a certain extent, an assembly task is a material handling and manipulation task. Unlike traditional material handling tasks in which a part is simply moved from one location to another, assembly tasks involve complex interactions between the two parts being joined, between the parts and tools and between the tools and fixtures. As a result, assembly operations can require a considerable amount of handling and orienting the parts in order to mate them properly. The main effect of these complications is to limit the automation of assembly operations. They most certainly have some other effects on the generation of the assembly process plan.

1.4 MOTIVATIONS FOR AUTOMATING ASSEMBLY PLANNING

1.4.1 Combinatorial complexity

In assembly, an important part of the process planning activity is the determination of the product's assembly sequence. This consists of determining a feasible order of combining the components together to construct the product. This task is usually performed by a human planner, who carefully examines the assembly drawing and generates the assembly sequences by mentally disassembling the product and reasoning about the various constraints [78].

The constraints alone rarely lead to a single assembly sequence. Industrial products of ten to twenty components typically present thousands of feasible assembly sequences [41]. It is then very likely that the human planner, as expert as he may be, will not have the ability to generate and evaluate all the many feasible assembly sequences, specially for a non trivial product. The human planner usually considers only a very small subset of them when making the decision of which one to use in production. This means that potentially good assembly sequences might be overlooked. Systematic and automated procedures for generating all possible assembly sequences of any given product would help overcome this problem.

1.4.2 Economical justification

Traditionally, assembly is either performed manually or using hard automation mass production devices. With the emergence of short life cycle products, flexible automation using programmable assembly robots has become a particularly attractive alternative.

Many companies are still reluctant to incorporate assembly robots into their assembly plant. One major reason for this is that assembly planning is still being done by hand. For small series production, which is the category in which a large majority of companies fall, this means that high manual planning costs must be amortized over the relatively small number of units to be produced during the relatively short life cycle of the product [22]. Automatic planning procedures could significantly reduce planning costs and provide financial opportunities for automation investments such as implementing assembly by robots.

Parallel to this is the fact that even minor changes in the product design can significantly affect assembly options and costs. Assembly costs are known to account for up to 60% of the total product cost. It is then essential that many design alternatives be evaluated before actually starting production. Procedures for automatically generating assembly plans could greatly speed up this analysis of the economical impact of different design solutions.

1.4.3 Standardization and quality of the generated plans

Assembly process plans generated by a computer are much more consistent than those generated manually. In the case of a variant system, a first level of consistency is achieved by the fact that all planners use the same software. Every new plan in a variant system is also constructed from previously existing ones. Still, some of the key decisions in the generation of a new process plan are subject to the personal experience and skills of the planner.

A generative CAPP system pushes the consistency even further, since the decision logic encoded in the software is consistent. A generative assembly planner is also likely to produce more optimal plans than its manual or variant counterpart. This is due to the speed advantage of the computer, which provides it with the capability of generating and comparing a large number of different assembly plans, maybe even all possible ones, before taking the final decision as to which one to use in production.

1.4.4 Applications in task level robotic languages

A great deal of research has been concerned with the development of high level languages for robotic assembly in order to facilitate the robot programming of complex assembly tasks [47] [56] [63] [68]. It is generally agreed that these languages should consist of an ordered list of high level commands relating to the operations to be performed. For the 3 blocks in figure 1.4, such commands would be of the form:

1– place plate against base,

2– fit pin into plate and base.

Given these high level commands, the robot would reason upon a three dimensional description of its surrounding world in order to infer the lower level details underlying each one of them. For the first command above, this includes identifying the position and orientation of the plate in the workspace, moving towards it without collision, grasping it properly, moving it towards the base without collision, and so on.

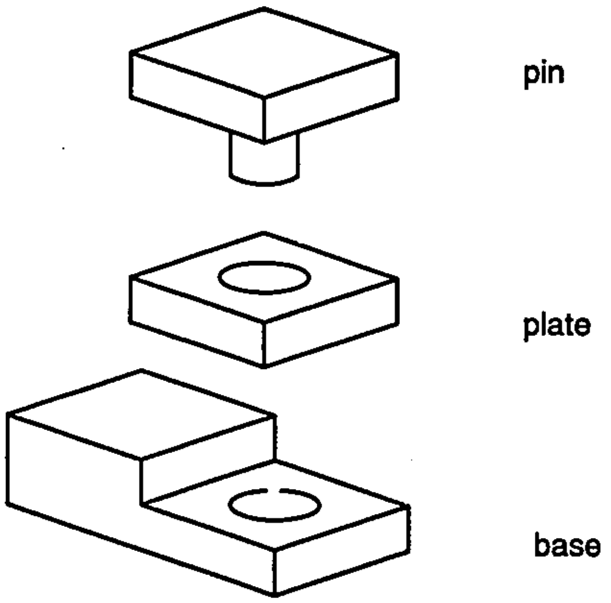


Fig. 1.4: Three blocks to be assembled.

Part of the activities of an automated assembly planner is exactly the automatic determination of the high level operations (or commands) to be performed, along with the identification of the parts and subassemblies that they involve and their order of execution. Integrating such a planner into the robot controller can simplify the programming task even further. That is, instead of specifying an ordered sequence of task level operation to be performed as above, the programming task could now be reduced to a single statement of the form:

assemble blocks.

1.4.5 CAD / CAM link

Historically, both CAD and CAM systems have evolved rather independently of each other. This translates into the existence of highly specialized but also very isolated islands of automation. For example, most major manufacturing companies have both a CAD system and some NC machine tools. Very few of them have the facilities for generating their NC programs directly from the CAD database.

The potential of integrating CAD with CAM is enormous. Unfortunately, the complexities involved are such that a true and complete integration is still many years away. It has long been recognized that CAPP is a key starting point for achieving this integration. The development of CAPP systems for either machining, inspection or assembly is then well justified in this respect.

1.4.6 Benefits in other domains

Process planning is an activity shared by many different domains, including construction, manufacturing, medicine or even restoration. A study of the requirements for automating process planning for the particular case of manufacturing assembly is likely to be beneficial to many other domains.

1.5 OVERVIEW OF GAPP

Figure 1.5 shows the block diagram of the developed GAPP. At the top level are the various inputs, which include the graph model of the product to be assembled and the various environment parameters.

At the heart of the system is the assembly process planner. It has 3 fundamental characteristics: it is generative by nature; it generates assembly sequences systematically; and the assembly sequences that it generates are optimal[3].

The generative characteristic means that GAPP does not need to know previously generated assembly plans of a similar product in order to come up with an assembly sequence of a new product. GAPP considers every problem it is faced with to be a new one and the solution to every problem is started from first principles.

The systematics of GAPP have two important implications. First, they ensure that no potentially good assembly sequence of a product is ever overlooked. Second, they ensure that the results are always consistent. This means that running GAPP twice for the same product with the same parameters[4] necessarily produces the same assembly sequence. This can be contrasted with two different human planners who are likely to come up with different assembly sequences of the same product.

Finally, GAPP has the power to enumerate all assembly sequences of a given product. It can also generate directly the optimal assembly sequence for that product without exhaustive search.

3. Whether the assembly sequence rendered by GAPP is optimal or not depends on the search method. This issue will be covered in more details in chapter 7.
4. Such parameters include search constraints, relative weight of search criteria and search method.

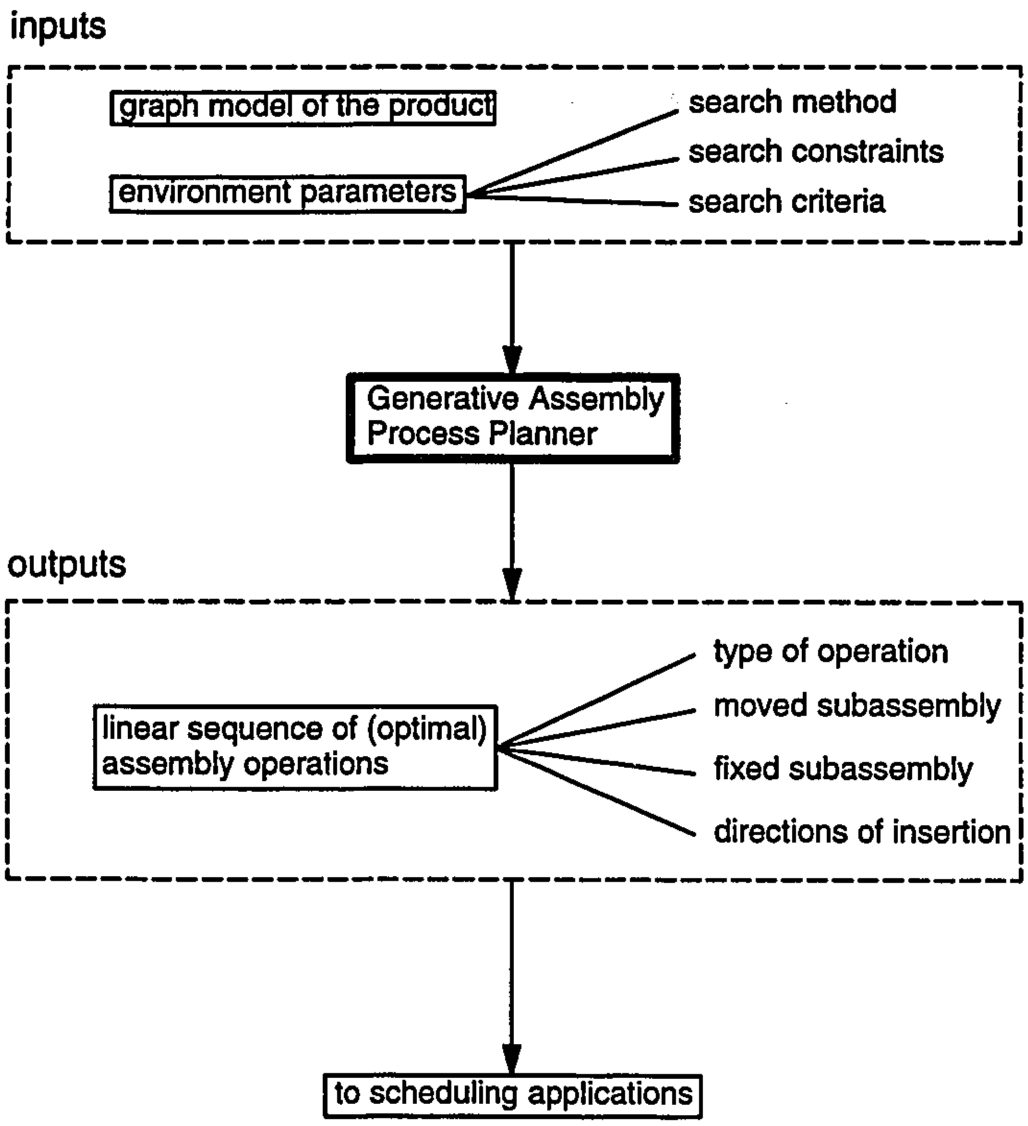


Fig. 1.5: Block diagram of GAPP.

CHAPTER 2

LITERATURE REVIEW

This chapter reviews some of the important works which emerged in the field of assembly planning. A detailed description of five of these works is provided in the first section. A less detailed description of other relevant works is provided in the second section. The third section highlights important aspects of assembly planning which still need to be researched, based on the characteristics of existing works reported in the first two sections. The fourth section describes the scope of this research. The last section outlines the remainder of this thesis.

2.1 RELEVANT RESEARCH

The works described in this first section have been carefully chosen by the author. It is believed that their detailed description should give a good insight into the essence of the assembly planning problem, as well as a better understanding of some of the tools that have now been developed to tackle this problem.

2.1.1 The work of Bourjault

Bourjault [4] is among the first ones to propose a systematic approach to assembly planning, which ensures the exhaustive generation of all the assembly sequences of a given product.

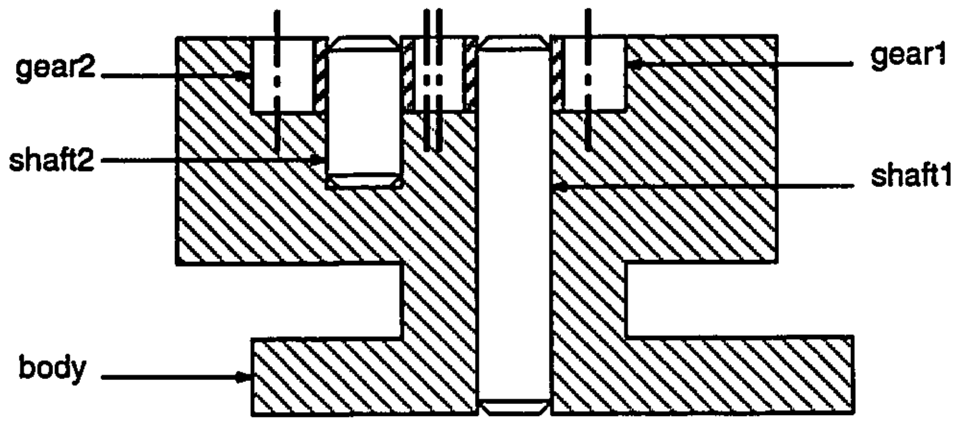


Fig. 2.1: An oil pump assembly [6].

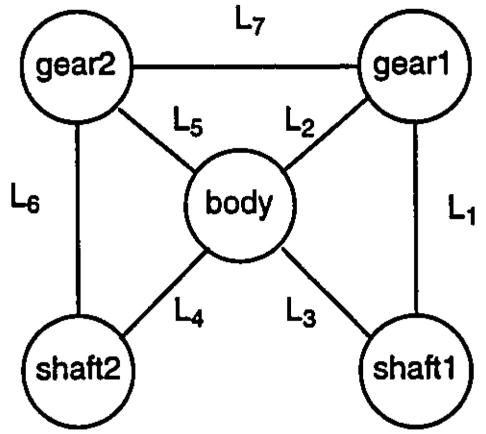


Fig. 2.2: *Graphe des liaison fonctionnelles* (graph of connections) for the oil pump in figure 2.1 [6].

His approach first consists of modelling the assembly as a simple[5] and connected[6] graph, called *graphe des liaison fonctionnelles* (graph of functional connections, or simply graph of connections). Each vertex in this graph represents one component in the assembly. Each edge, also called *liaison*, reflects the presence of a physical connection between two components in the product. Figure 2.2 shows the *graphe des liaison fonctionnelles* for the oil pump in figure 2.1.

Once the *graphe des liaisons fonctionnelles* has been generated, a structured set of symmetric questions is asked to the user. Each question must be answered by either "yes" or "no". If L_i, L_j and L_k are *liaisons* in a product, a typical question has the form:

1– Can L_i be established if L_j and L_k have already been established?

The symmetric question has the form:

2– Can L_i be established if L_j and L_k have not been previously established?

The result of this systematic questioning process is a list of precedence constraints expressed in the form of binary relations R and S, defined as follows:

$$R = \{\ (L_i, L_j)\ \mid\ (L_i,L_j) \in LxL\ \wedge\ L_i \text{ cannot be established if } L_j \text{ is already established }\} \tag{2.1}$$

$$S = \{\ (L_i, L_j)\ \mid\ (L_i,L_j) \in LxL\ \wedge\ L_i \text{ cannot be established if } L_j \text{ has not been established }\} \tag{2.2}$$

where L denotes the set of all *liaisons* of a product. If more than two liaisons are involved, these binary relations can be easily converted into "n–ary" relations:

5. A simple graph is one in which any two vertices are connected by at most one edge.
6. A connected graph is one in which there is a path between any two of its vertices.

$$R = \{ (L_i, L_j, \ldots, L_n) \mid (L_i, L_j, \ldots, L_n) \in L x L \ldots x L \wedge$$
$$L_i \textit{ cannot be established if } L_j \wedge \ldots \wedge L_n \textit{ are already established} \} \tag{2.3}$$

$$S = \{ (L_i, L_j, \ldots, L_n) \mid (L_i, L_j, \ldots, L_n) \in L x L \ldots x L \wedge$$
$$L_i \textit{ cannot be established if } L_j \wedge \ldots \wedge L_n \textit{ have not been established} \} \tag{2.4}$$

These precedence relations simply express the fact that the establishment of a particular *liaison* of a product generally requires the presence or absence of other *liaisons* of that product.

Bourjault also introduces the notion of the *state* of an assembly. In his work, these states are translated into logical functions. A binary variable f_i is first associated with each *liaison* L_i in the graph, which determines the state of the corresponding *liaison* at a particular point in time in the assembly process:

$$f_i = \begin{cases} 1 & \text{if } L_i \textit{ is established} \\ 0 & \text{if } L_i \textit{ is not established} \end{cases} \tag{2.5}$$

The assembly state S_k at any point in time of the assembly process of a product with "n" *liaisons* can be characterized by a binary vector of length "n", obtained by simply concatenating the binary variables f_i:

$$S_k = \{ f_1, f_2, \ldots, f_n \} \tag{2.6}$$

The logical function associated with this vector is:

$$\Phi_k = \tilde{f}_1^k \cdot \tilde{f}_2^k \cdot \ldots \cdot \tilde{f}_n^k \tag{2.7}$$

where $\tilde{f}_i^k = f_i^k$ *or* $\bar{f}_i^k$, according to whether L_i is established or not in the kth state.

Bourjault then reconstructs for each *liaison* the list of all the states permitting its establishment. The logical function associated with the set of all such states for a *liaison* L_i is denoted by C_i and is called *condition de réalisabilité* (establishment condition) for *liaison* L_i. If *liaison* L_i has "s" states authorizing its establishment, the *condition de réalisabilité* C_i for this *liaison* is given by:

$$C_i = \sum_{k=1}^{s} \Phi_k = \sum_{k=1}^{s} \left(\prod_{i=1}^{n} f_i^k \right) \tag{2.8}$$

As an example, the *condition de réalisabilité* for *liaison* L_1 in figure 2.2 is as follows:

$$\begin{aligned} C_1 = {} & \bar{f}_1 f_2 \bar{f}_3 \bar{f}_4 \bar{f}_5 \bar{f}_6 \bar{f}_7 + \bar{f}_1 \bar{f}_2 f_3 \bar{f}_4 \bar{f}_5 \bar{f}_6 \bar{f}_7 + \bar{f}_1 f_2 \bar{f}_3 f_4 \bar{f}_5 \bar{f}_6 \bar{f}_7 + \bar{f}_1 f_2 \bar{f}_3 \bar{f}_4 \bar{f}_5 f_6 \bar{f}_7 \\ & \bar{f}_1 \bar{f}_2 f_3 f_4 \bar{f}_5 \bar{f}_6 \bar{f}_7 + \bar{f}_1 \bar{f}_2 f_3 \bar{f}_4 f_5 \bar{f}_6 \bar{f}_7 + \bar{f}_1 \bar{f}_2 f_3 \bar{f}_4 \bar{f}_5 f_6 \bar{f}_7 + \bar{f}_1 \bar{f}_2 f_3 \bar{f}_4 \bar{f}_5 \bar{f}_6 f_7 \\ & \bar{f}_1 f_2 \bar{f}_3 \bar{f}_4 f_5 \bar{f}_6 f_7 + \bar{f}_1 \bar{f}_2 f_3 f_4 \bar{f}_5 \bar{f}_6 f_7 + \bar{f}_1 \bar{f}_2 f_3 \bar{f}_4 \bar{f}_5 f_6 f_7 + \bar{f}_1 \bar{f}_2 f_3 f_4 f_5 f_6 \bar{f}_7 \\ & \bar{f}_1 f_2 \bar{f}_3 f_4 f_5 f_6 f_7 \end{aligned} \tag{2.9}$$

Such *conditions de réalisabilité* are obtained by analyzing the precedence relations obtained from the questioning process. For example, an affirmative answer to the question:

Can L_i be established if L_j has already been established?

means that no C will contain expressions including $\bar{f}_i f_j$ and that C_j will not contain expressions including $\bar{f}_i\bar{f}_j$ unless L_i and L_j can be established simultaneously.

The systematic exploitation of the *condition de réalisabilité* of each *liaison* in a product leads to the generation of all its feasible assembly sequences. To illustrate how this can be achieved, assume that function C_1 of *liaison* L_1 of some product has the totally disassembled state $\bar{f}_1\bar{f}_2 \ldots \bar{f}_n$ in it. By the definition of C, this means that there exists at least one assembly sequence which can start by the establishment of this *liaison.* Assume now that function C_2 of *liaison* L_2 of the same product has the state $f_1\bar{f}_2 \ldots \bar{f}_n$ but not the state $\bar{f}_1\bar{f}_2 \ldots \bar{f}_n$ in it. This says that although no assembly sequence can start by establishing L_2, this *liaison* can be established as soon as L_1 has been established, meaning that the first 2 operations in at least 1 assembly sequence of that product consists of establishing L_1 and L_2, successively. If another *liaison* L_k for that product had $f_1\bar{f}_2 \ldots \bar{f}_n$ in its C_k function, another partial assembly sequence starting by L_1 and L_k would be generated. It should be clear how a systematic repetition of this process for all L_i and all C_i can lead to the generation of all feasible assembly sequences of a given product.

This systematic generation process lends itself to a forest structure to represent all the feasible assembly sequences of a product. The root node of each tree in this forest is one of the *liaisons* by which the assembly process can start. Any path from the root node of a tree to any of its leaf nodes is an assembly sequence, where each node in the path represents the *liaisons* that are established. The total number of assembly sequences is given by the number of leaf nodes in every tree in the forest.

The oil pump in figure 2.1 has six trees in the forest representing all its assembly sequences. Figure 2.3 shows one such tree, contributing to 8 of the 54 feasible assembly sequences of this product.

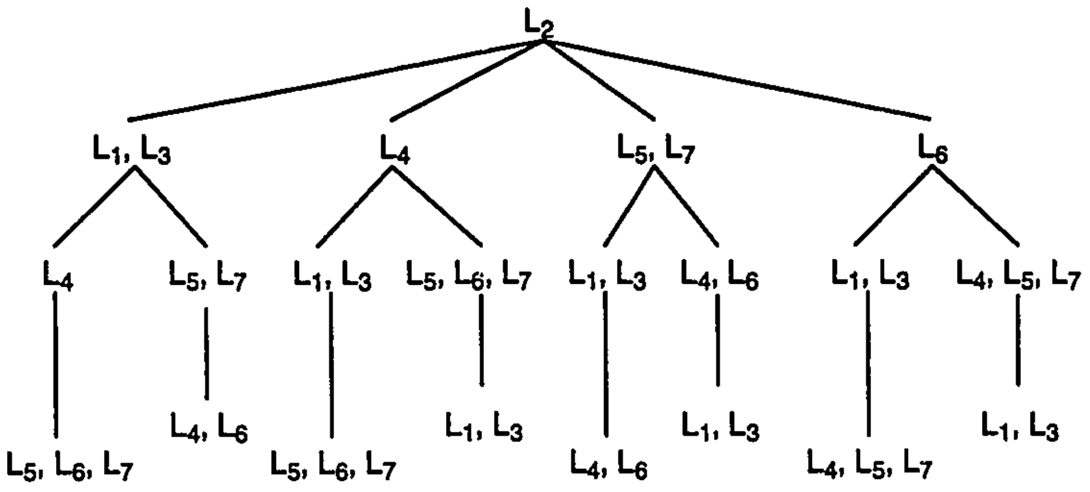


Fig. 2.3: One of the six trees in the forest of assembly sequences for the oil pump in figure 2.1 [6].

One offspring of this initial work by Bourjault includes the automatic determination of subassemblies from an analysis of the assembly sequence trees [6]. Another offspring is the representation of the assembly sequences using Petri nets from an analysis of the R and S relations [5].

2.1.2 The work of Whitney et al.

The work by Whitney et al. [79] is an adaptation of the earlier effort of Bourjault. Their approach is essentially the same as that of Bourjault. However, it presents some important improvements which make the developed methods more applicable to products with a larger number of components.

The approach starts by representing the product as a *graph of connections.* This graph is directly equivalent to Bourjault's *graphe des liaisons fonctionnelles.* Figure 2.5 shows the graph of connections of the ball–point pen in figure 2.4.

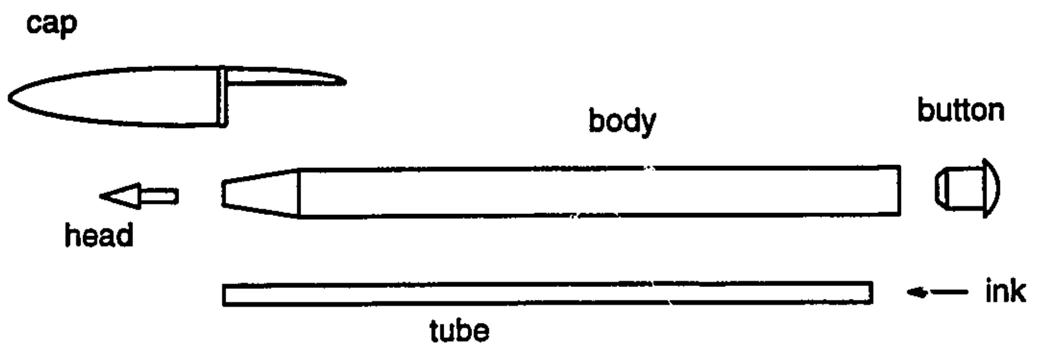


Fig. 2.4: A ball–point pen to be assembled [12].

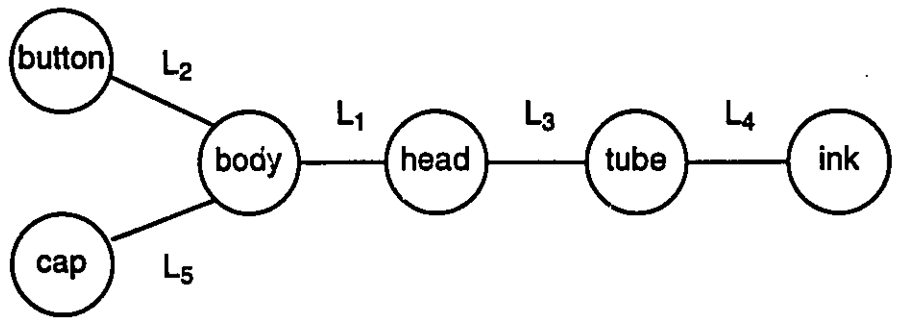


Fig. 2.5: Graph of connections for the ball–point pen in figure 2.4 [12].

Then, as is done by Bourjault, the user is asked a series of questions for obtaining the precedence relations. An important improvement lies in the number of such questions that the user must answer. For an assembly consisting of "n" liaisons, De Fazio and Whitney [12] showed that the number of questions "Q" to be asked using Bourjault's approach necessarily falls into the interval:

$$n2^n > Q \geq 2(n^2 + n) \qquad \text{for } n \geq 3 \tag{2.10}$$

For 25 liaisons, which is typical of a product with 10 to 15 parts, this means that at least 1,300 and at most 800 million questions need to be asked. This makes Bourjault's approach truly inapplicable for products with a larger number of parts[7].

The basis of their improvement is to phrase the questions so that fewer of them are needed. Although the form of these questions is similar to that of Bourjault's, their answer, instead of simply being "yes" or "no", now consists of Boolean expressions involving liaisons. Using this new approach, the number of questions can be reduced to:

$$Q = 2n \tag{2.11}$$

That is, two questions for each liaison defined in the graph of connection[8]. The two questions for any liaison L_i always have the form:

$Q1_i$: What liaison(s) must be established before L_i can be established?

$Q2_i$: What liaison(s) must be left undone so that L_i can be established?

7. This important limitation of Bourjault's approach is evident from the example products he uses to demonstrate his results: these products always have very few parts (5 or so) with very few connections (10 or so).
8. Chen [11] describes an approach in which only one question is required for each liaison in the graph of connections.

For liaison L_i, a typical answer to both questions is:

$$A1_i:\ (L_j \vee (L_k \wedge L_m)) \rightarrow L_i \tag{2.12}$$

$$A2_i:\ L_i \rightarrow (L_r \vee (L_s \wedge L_t)) \tag{2.13}$$

where $\rightarrow$ means "must precede" and L_j, L_k, L_m, L_r, L_s and L_t are other liaisons in the product.

Using the rules of Boolean algebra, the answers are merged into a set of precedence relations (PR) of the form:

$$(L_j \vee (L_k \wedge L_m)) \rightarrow L_i \tag{2.14}$$

Each PR is divided into a left hand side (LHS) and a right hand side (RHS). The LHS consists of a Boolean expression involving liaisons which must be established prior to the establishment of the liaison on the RHS. There is always at most one liaison on the RHS of any PR and no two different PR can have the same liaison as their RHS. In other words, one and only one PR is defined for each RHS liaison[9]. A liaison that has been established holds the value TRUE (or 1) in any of the PR in which it appears, FALSE (or 0) otherwise.

Like Bourjault, Whitney et al. also use the concept of the *state* of an assembly. In their work, the state of the assembly at any point in the assembly process is characterized by a binary vector. There are as many positions in that vector as there are liaisons in the graph of connections. A value of zero in a particular position of the vector means that the corresponding liaison in the graph has not yet been established. A value of one means that the liaison has been established. In

9. It is reported in [79] that this restriction to a single liaison on the RHS of each PR greatly simplifies the algorithm developed. In particular, it simplifies the translation of the PR into the actual code in which they are manipulated and evaluated.

particular, a vector of zeros corresponds to the completely disassembled state and a vector of ones corresponds to the completely assembled state.

Starting with the completely disassembled state at rank 0, the generation of all the assembly sequences of a product consists of determining all the next possible states that can result from the establishment of some executable liaisons. These new states are placed at rank 1 (see figure 2.6). Then, for all the states at rank 1, all the next possible states that can result from the establishment of some further executable liaisons are determined and placed at rank 2. For a product with "n" parts, this process is repeated until the completely assembled state at rank "n–1" has been generated.

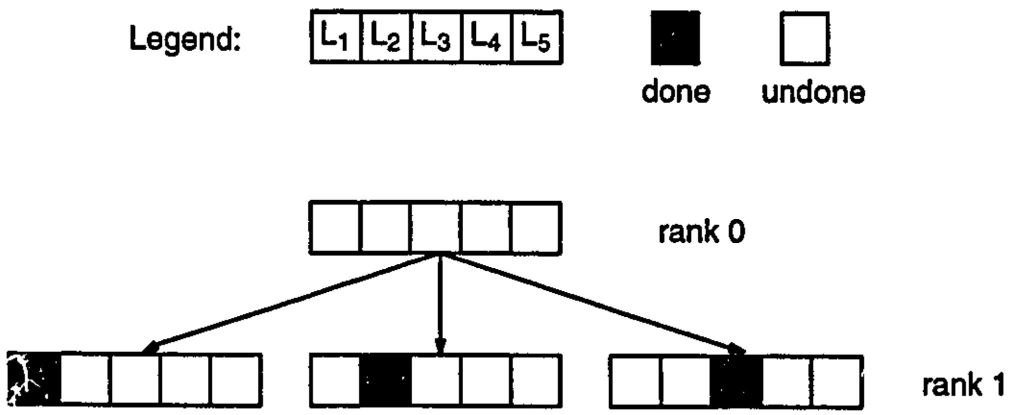


Fig. 2.6: The first two ranks in the search graph of the ball–point pen in figure 2.4. The first rank is the completely disassembled state (no liaison established). The second rank shows the establishment of the three liaisons by which assembly can start.

To determine if a particular liaison is executable, it is first assumed to be established. Therefore, the RHS of its corresponding PR becomes TRUE and the PR is triggered for evaluation. If the Boolean expression on the LHS of this PR also evaluates to TRUE, then the liaison is executable. A new vector state is created with a "1" in the position of the newly established liaison. The value TRUE is also set for this liaison in the LHS of all the PR's in which it appears. This enables the LHS of some subsequently triggered PR to evaluate to TRUE in the next ranks.

Whenever a new state is generated in a rank, it is compared to all existing states in this rank to see if it is indeed a new state before adding it to the rank. The resulting structure to encode all assembly sequences of a product is called a *directed graph of assembly states.* This constitutes another improvement to Bourjault's work, as the directed graph of assembly states is much more compact than Bourjault's trees. Figure 2.7 shows the directed graph of assembly states representing the 12 assembly sequences of the ball–point pen in figure 2.4

It is worth mentioning that although Whitney et al.'s work represents an important improvement in the number of questions asked for acquiring the precedence knowledge, it also requires much more reasoning on the part of the user to supply the answers. Bourjault's questions are answered out of context. This means that one has to reason upon only the parts involved in the liaisons in a question, without having to consider if the remainder of the assembly can be completed or not. This is not the case for the modified questions of Whitney et al. Perhaps the best way to describe this limitation is to use the authors' own words [13]:

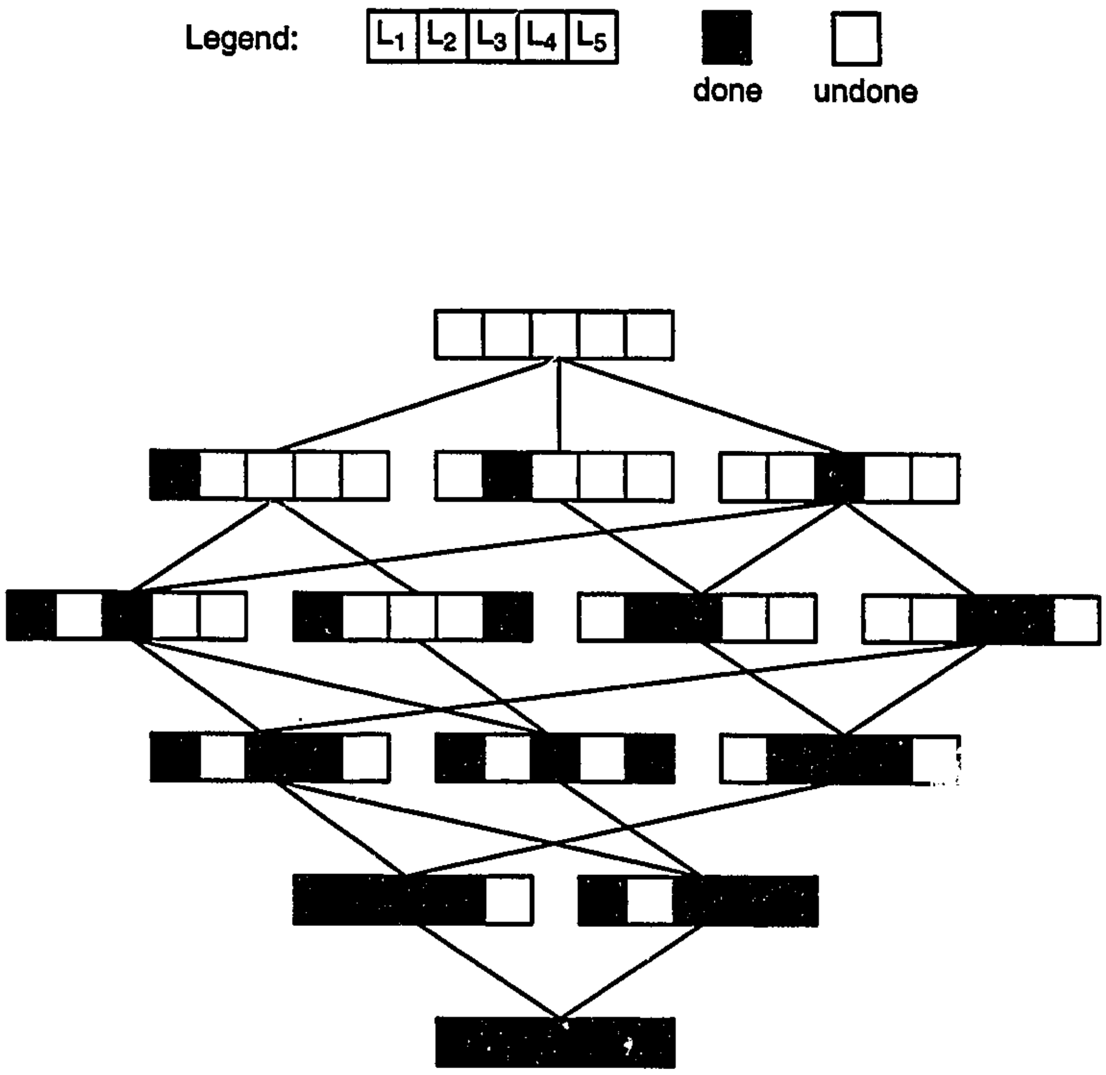


Fig. 2.7: Directed graph of assembly states for the ball–point pen in figure 2.4 [12]. Every path from the root node (top) to the leaf node (bottom) is a feasible assembly sequence. Although this graph is directed, downward pointing arrow heads have been neglected for simplicity.

"Questions cannot be answered by simply considering interference between two parts–sets named in the question. Physical reasoning is involved. The user must identify and consider subsequent states. The price is evident but there are benefits: question count is reduced, perhaps greatly."

The work of Whitney et al. has recently matured into an interactive computer aid specially designed to assist a user in the determination of the assembly sequence(s) of a new product [13]. A solid modeller is used to help the user visualize the geometric constraints involved in the questioning process. When all questions have been answered and the precedence constraints determined, the directed graph of assembly states representing all assembly sequences satisfying these constraints is displayed to the user. The sequence count can then be reduced to a reasonable size by interactively editing the graph and screening all possible solutions. The user reasons upon mechanical criteria, such as suitable parts–mating, jigging or parts orientation and interactively eliminates undesirable states, arcs or entire assembly sequences from the graph.

2.1.3 The work of Homem De Mello and Sanderson

Another significant work in assembly planning was done by Homem De Mello and Sanderson [24] [25] [26]. Their approach starts by the construction of a relational model of the assembly. The graph of connection introduced by Bourjault is a simple subgraph of the relational model of the assembly and it can be easily obtained. Figure 2.9 shows this correspondence between the relational model and the graph of connections for the product in figure 2.8.

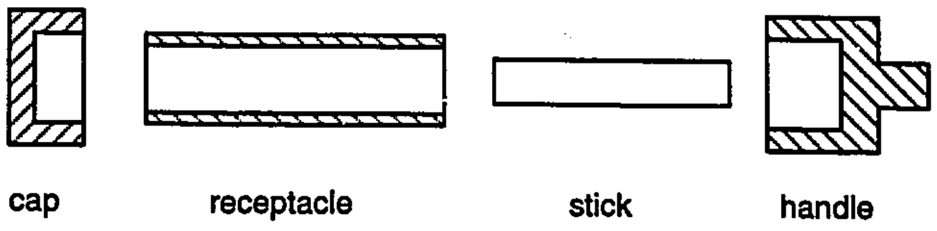


Fig. 2.8: A simple product to be assembled [25].

One important characteristic of Homem De Mello and Sanderson's work is that it uses a decomposition approach. That is, they transformed the problem of generating all mechanical assembly sequences of a product into the problem of generating all disassembly sequences for the same product[10]. Under the assumption that no assembly operation involves parts deformation, any assembly sequence is just the reverse of the equivalent disassembly sequence.

The generation of all possible disassembly sequences starts by obtaining the graph of connection from the given relational model of the assembly. All the cutsets of this graph are generated. The set of all cutsets represent all possible ways of splitting the graph. Physically, this represents all the ways the assembly can be split into two subassemblies. The same process is then repeated for the new subassemblies, for the sub–subassemblies, and so on, until only single parts are left.

10. The work of Sekiguchi et al. [72] is one of the earliest ones that could be traced by the author which uses this disassembly approach.

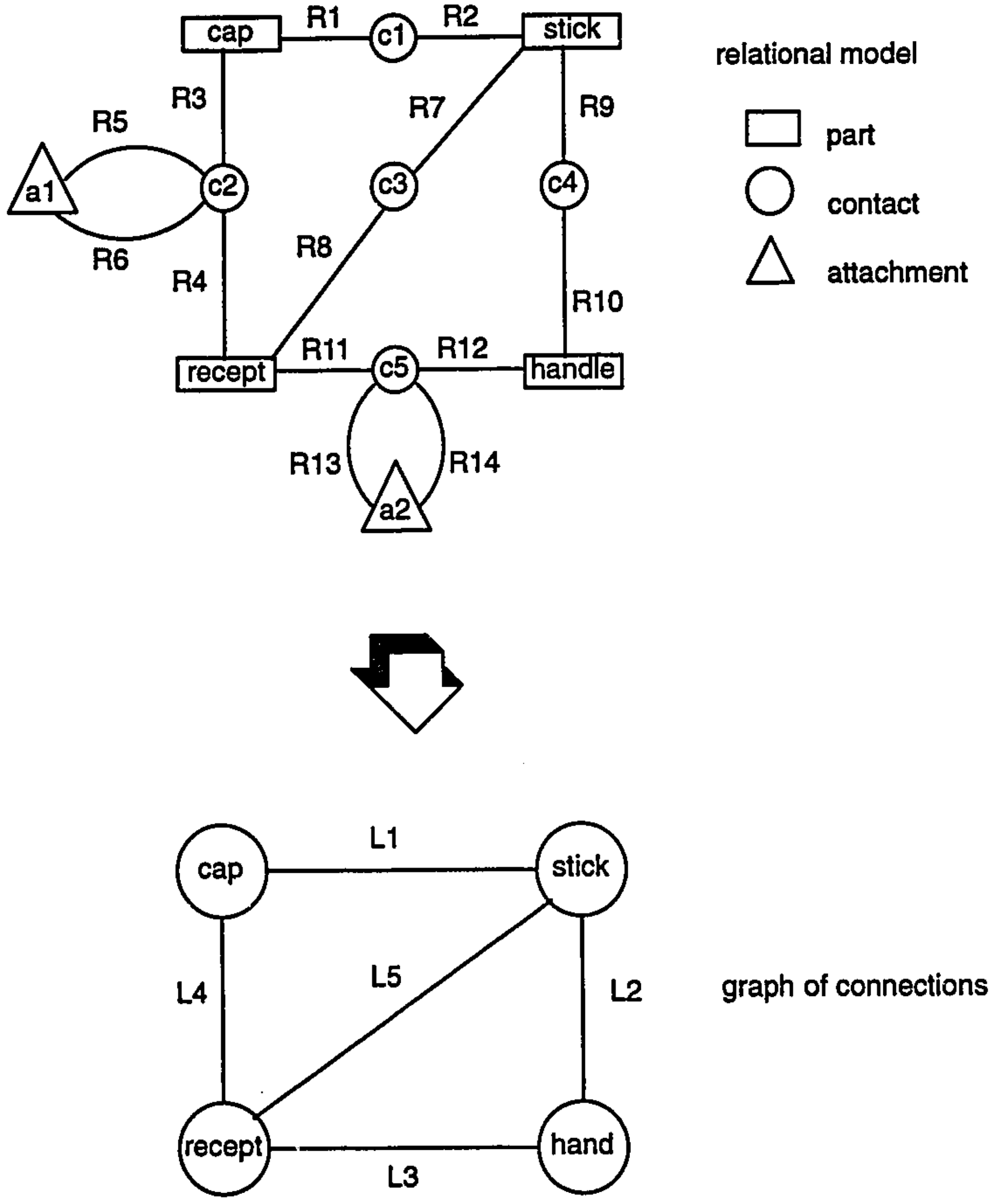


Fig. 2.9: Correspondence between the relational model (top) and the graph of connections (bottom) for the product in figure 2.8 [25].

In practice, only a subset of the set of all cutsets corresponds to feasible decompositions of the assembly into subassemblies. The determination of which cutsets correspond to feasible decompositions is based on the computation of three predicates: geometric feasibility, mechanical feasibility and stability.

The computation of the geometric feasibility predicate is separated into two stages: local analysis and global analysis. The local analysis consists of determining if there exists a direction in which a component (or subassembly) can be moved infinitesimally, with all remaining components of the assembly being fixed. This is achieved by computing the polyhedral convex cone of motion for the component (or subassembly) under investigation. Any component (or subassembly) whose polyhedral convex cone is not a point can be moved infinitesimally in some direction. The global analysis consists of determining if a component (or subassembly) which can move infinitesimally can also be moved to infinity without colliding with any of the remaining components of the assembly.

The mechanical feasibility predicate is TRUE if the attachments acting on the contacts of a decomposition are not blocked in the resulting assembly, and are not present in either one of the subassemblies. Figure 2.10 illustrates this concept: although it is geometrically feasible to remove the pin, the access to screw1 is blocked by the cap. Therefore this operation is mechanically unfeasible.

The stability predicate is TRUE if the parts in either physical subassembly of a decomposition maintain their relative position and do not break contact spontaneously. The algorithms for computing this predicate were not discussed in the published literature.

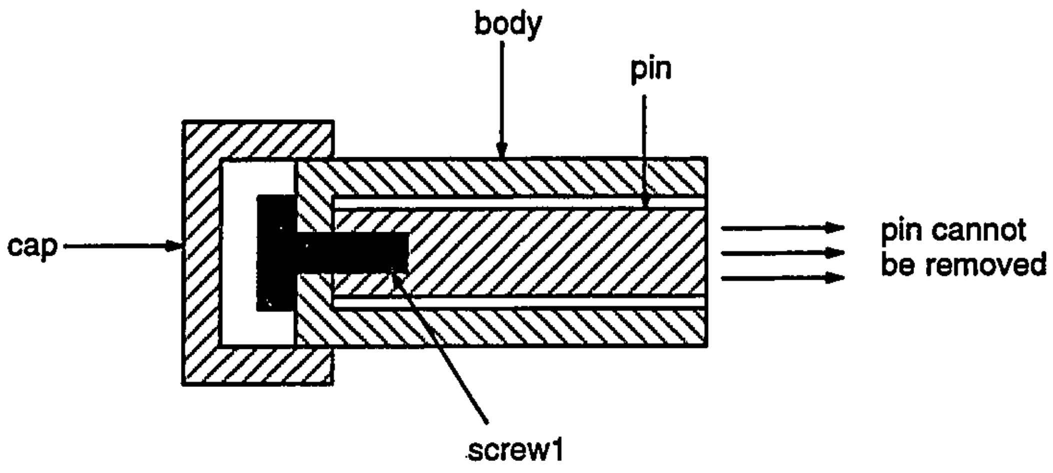


Fig. 2.10: A mechanically unfeasible operation:
part screw1 is blocked by the cap [25].

The decomposition approach lends itself to an AND / OR graph representation of assembly sequences, where the root node corresponds to the complete assembly, terminal nodes correspond to single parts, and intermediate nodes correspond to subassemblies. Every node (except terminal nodes) is split in pairs of children nodes, each corresponding to a feasible decomposition of the parent node. A parent node and any of its pairs of children nodes are linked by hyperarcs in the graph. Figure 2.11 shows the AND / OR graph of the product in figure 2.8.

Compared to both works described earlier, the disassembly approach used in this work is clearly distinguished. A more subtle difference lies in the targeted use of the developed planning systems. Whitney et al. focus on assembly sequence design, where assembly sequence issues are considered during the design stage of a new product. The issue here is the economical impact of different assembly sequences on the assembly system which will be used to assemble the new product in large quantities.

On the other hand, Homem De Mello and Sanderson focus more on low–volume, flexible robotic assembly of existing products and address more particularly the problem of opportunistically scheduling assembly operations. The typical scenario is a product to be assembled (or disassembled or repaired) in a robotic cell where parts arrive in random order. The issue here is the flexibility of the plans produced to avoid idle time in waiting for particular parts to arrive.

Offsprings of Homem De Mello and Sanderson's work include the determination of two criteria from which the various assembly plans in the AND / OR graph can be evaluated and the best one selected[11], using standard graph searching techniques [26]. An interactive system for generating and evaluating assembly plans, called *PLEIADES*[12], has also been recently proposed, as a practical implementation of some of the techniques described in this work [67].

11. The first criteria is the maximization of the flexibility of sequencing the assembly tasks, while the second is the minimization of the assembly time through parallel execution of assembly tasks.
12. PLEIADES stands for PLanning Environment for Integrated Assembly system DESign.

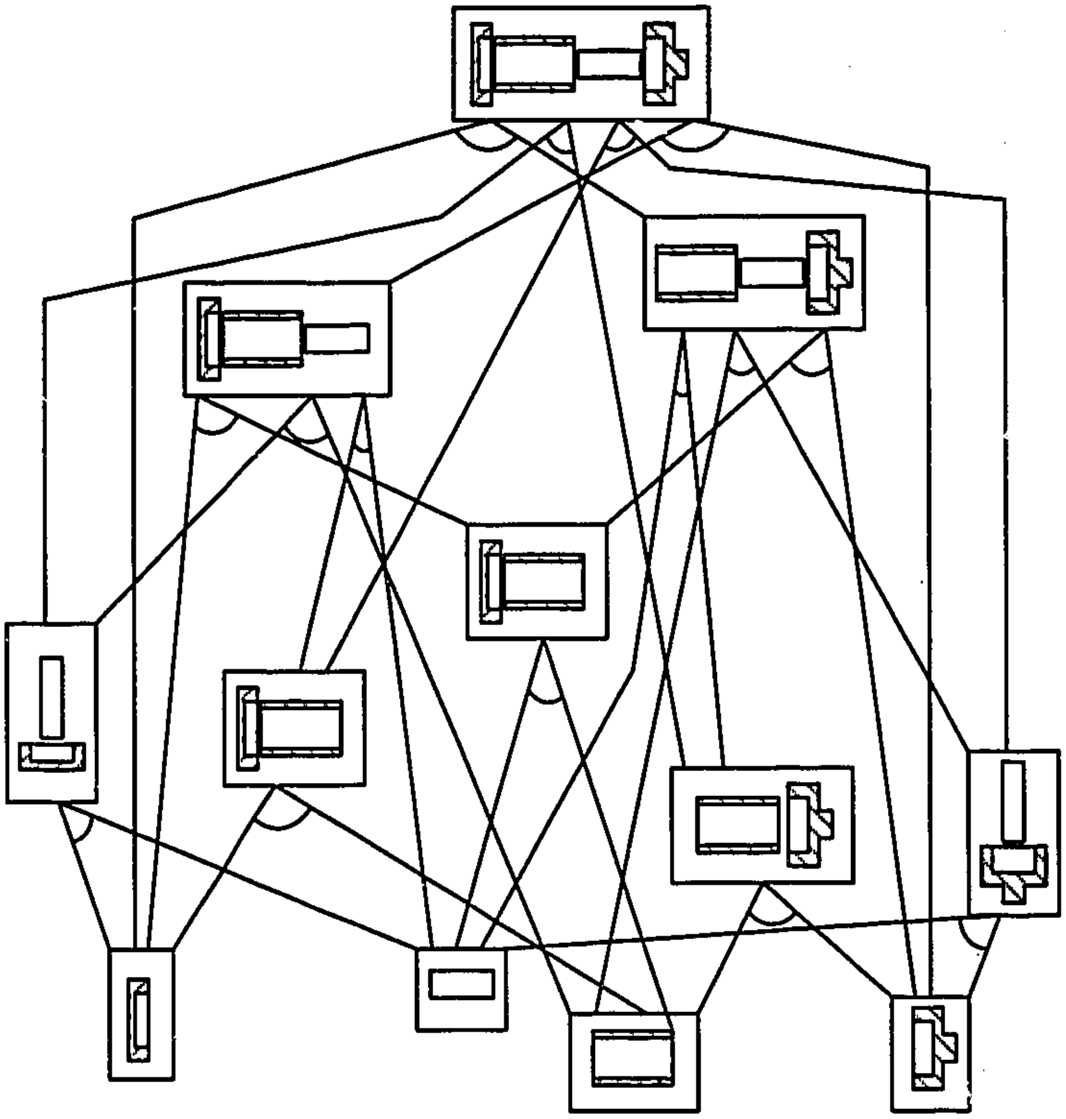


Fig. 2.11: The AND / OR graph which represents all disassembly sequences of the product in figure 2.8 [25].

2.1.4 The work of Huang and Lee

Huang and Lee [31] [32] [33] describe a *knowledge–based* assembly planning system. The assembly is described by an undirected graph called *Feature Mating Operation Graph* (*FMOG*). Two different types of vertices are present in the graph: those that represent components of the assembly (set S) and those that represent feature mating operations between components (set C). Each vertex in S is connected to at least one vertex in C. Each vertex in C is connected to at least two vertices in S. Vertices in C are never connected with each other; neither are vertices in S. Figure 2.13 shows the FMOG of the tilt mechanism in figure 2.12.

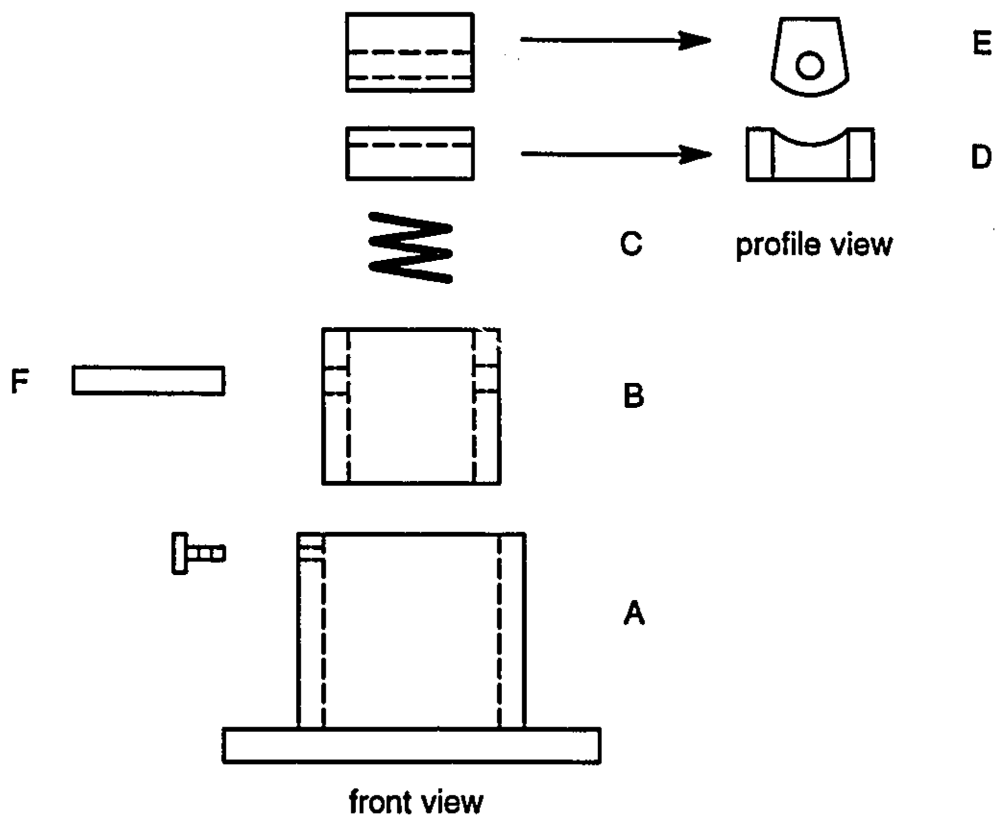


Fig. 2.12: A tilt mechanism to be assembled [31].

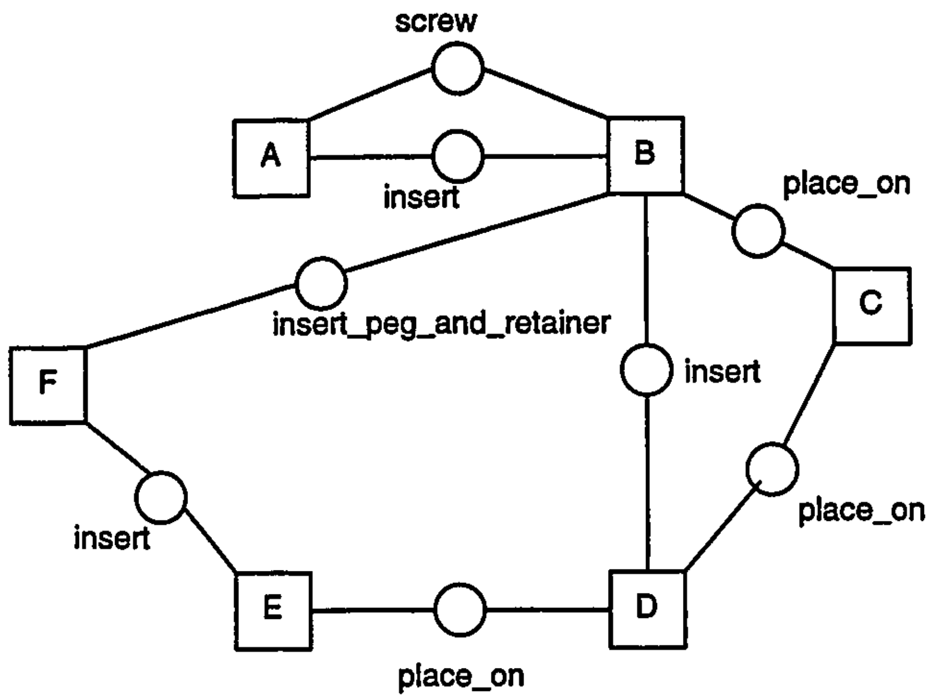


Fig. 2.13: Feature Mating Operation Graph of the tilt mechanism in figure 2.12 [31].

The authors propose a symbolic representation of the precedence knowledge of an assembly. Two predicates,

$$MP\ (\ P_i,\ P_j\ ) \tag{2.15}$$

$$NL\ (\ P_i,\ P_j\ ) \tag{2.16}$$

are defined. The former reads "Must Precede" and the latter "No Later than". Clearly, these are equivalent to the R and S relations defined earlier in Bourjault's work, respectively. However, one important distinction lies in the representation of the state operators P_i and P_j. In Bourjault's work, each operator L_i and L_j in the R and S relations could only represent a single state. The operators P_i and P_j in the above predicates can actually represent a set of states. Huang and Lee describe in details how this compression can be achieved using the concept of the *product of two states* coupled with the rules of Boolean algebra [31].

The product is studied from a disassembly point of view. The authors recognized that a questioning process similar to that of Whitney et al. could be used to acquire the precedence knowledge. Only the format of the two questions asked would have to be changed and adapted to the disassembly approach. The questions would have the form:

$Q1_i$: What is the set of operations which must be undone before and no later than undoing operation i?

$Q2_i$: What is the set of operations such that operation i must be undone before and no later than that set of operations?

Instead of this questioning process, the authors use geometric reasoning techniques to acquire the precedence knowledge about geometric constraints automatically[13]. A *Geometric Mating Graph* (*GMG*) is constructed, which is a completely connected and undirected graph. Vertices in this graph are the mating components or an additional part required for a mating operation in the assembly (such as a fastener). Edges are divided into two subsets: solid and dotted. Solid edges represent geometric relationships between components that are in physical contact, while dotted edges represent geometric relationships between components not in physical contact.

13. The authors refer to geometric constraints as *hard constraints*. Other types of constraints which can influence precedence knowledge, such as ease of assembly or even personal preference, are referred to as *soft constraints*.

Two procedures, *find_pre_conditions()* and *find_post_conditions()*, process the GMG and find automatically the pre–condition and post–condition for each mating operation in FMOG. The pre–condition of an operation is the set of all possible states of the assembly which must occur before or no later than performing this operation. Similarly, the post–condition of an operation is the set of all possible states of the assembly which have to appear after or no earlier than finishing this operation.

The generated precedence constraints become part of the *static knowledge base*, along with the assembly structural knowledge and the resources constraints. A set of production rules (three of them to be exact) is maintained in the *assembly rule base*. An assembly plan consists of an ordered list of instances of those rules. The *inference mechanism* implements a graph search algorithm which searches through a large space of assembly plans to find the optimal assembly plan based on the criteria specified by the user, or on the default value set up by the system.

2.1.5 The work of Wolter

Wolter [83] describes *XAP / 1*, which stands for *eXperimental Assembly Planner*, version *1*. This is a *rule–based* system which has the ability to generate a single and optimal assembly sequence that best satisfies some criteria.

The approach starts by proposing insertion trajectories for each of the parts in the assembly, for example "screw down" or "insert from above". For each proposed trajectory of each part, all other parts which would block that trajectory if they were already assembled are determined. Clearly, if a part cannot be moved in any of its proposed trajectories without colliding with some other parts already assembled, then this part must be assembled before these other parts. This geometric information is modelled using rules relating trajectory usage to part sequencing, for example:

if part A uses trajectory X then it must assembled before part B.

Such rules are generated for every possible collision that can occur during the assembly process.

For convenience, an *Assembly Constraint Graph* (*ACG*) is used initially to represent graphically the set of all possible collisions. Sequencing rules such as the one above then become easier to extract from the graphical representation. The nodes in the ACG are the various parts to be assembled. A directed edge from node A to node B means that part A collides with part B along some trajectory, thus requiring part A to be assembled before part B. As parts may have many proposed trajectories, two nodes in the graph may be linked by more than one directed edge, i.e. the ACG is not a simple graph. Figure 2.15 shows the ACG of the scissors in figure 2.14.

The set of all sequencing rules forms a rule network. An assembly plan consists of an ordered list of instances of these rules. The inference process can start by either supplying a *trajectory assertion*, such as:

part A uses trajectory X

or by supplying a *sequencing assertion*, such as:

part A must be assembled before part B.

The former are matched against the left hand side of the rules in the rule network, while the latter are matched against their right hand side. In any case, the inference engine finds all other assertions which arise as consequences. Rule chaining results in the generation of an assembly plan.

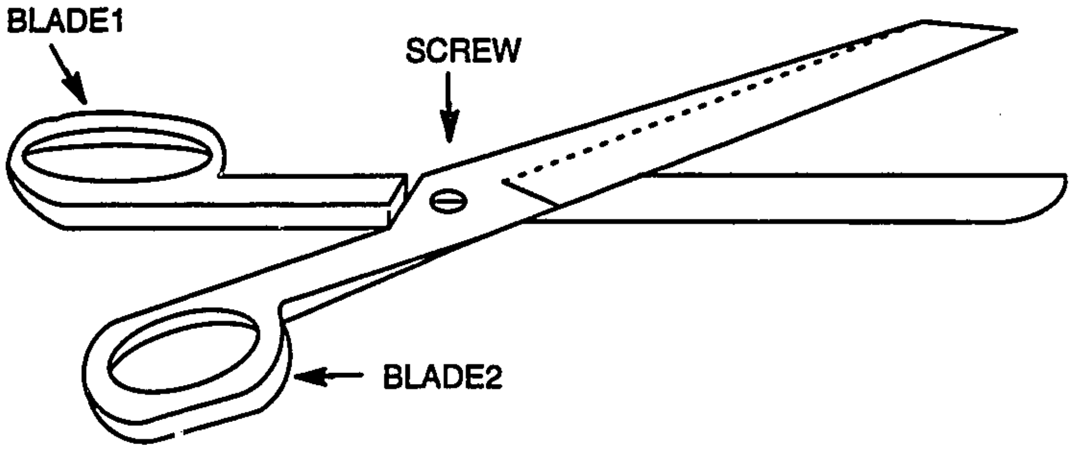


Fig. 2.14: Scissors to be assembled [83].

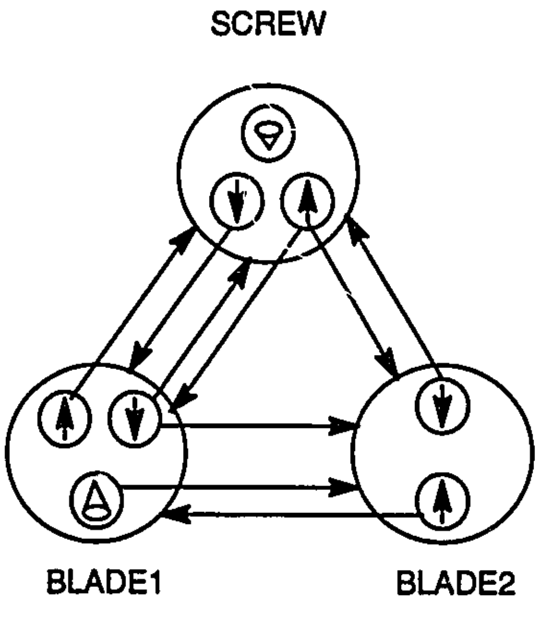


Fig. 2.15: Assembly Constraint Graph for the scissors in figure 2.14 [83].

The search through the rule network is mapped into the expansion of a binary search tree. A branch–and–bound method is used to find the optimal path in this tree, which corresponds to an optimal plan. The method basically consists of always selecting the leaf node of the search tree with the best rating and applying a new assertion to this node, yielding two new child nodes: one for which the assertion was applied and one for which the negation of the assertion was applied. Ratings for both new children are computed using criteria such as directionality of the operation, manipulability of the parts involved and fixture complexity. The ratings are always computed locally, in the sense that they evaluate the effects of the current assertion without considering any cascading effects at deeper levels in the search tree.

Wolter's work is innovative, as it focuses on both the automatic generation and evaluation of assembly plans. Nevertheless, an important limitation of his work lies in the linearity characteristic of his planner: XAP / 1 cannot handle operations where more than one part is moved and assembled with another part or subassembly. In other words, out of the two subassemblies involved in an assembly operation, at least one must be a single part.

2.2 OTHER RELEVANT WORKS

Khosla and Mattikali [33] describe a system capable of generating an assembly sequence from a 3–D model. They first model the product using the *NOODLES* solids modeller [33]. The graph of connection (called *component graph* in their work) is generated automatically from the 3–D model. The component graph is then split in various feasible ways, simulating a disassembly process[14]. One characteristic of this work is that if more than one split is possible for a particular

14. Although this "splitting" process is fundamentally the same as the decomposition process introduced by Homem De Mello and Sanderson, the details of its implementation seem to be different.

subassembly, heuristics are used to evaluate and order them. An example of such a heuristic is that a split involving a top–down operation would be better than another involving a sideways operation. Unlike Homem De Mello and Sanderson's work, where all feasible assembly sequences are generated and encoded in an AND / OR graph, a consequence of this evaluation process is that a single, most preferable disassembly plan can be directly generated and encoded in an AND / OR tree. Another interesting characteristic of their work is that it proposes to model the assembly facilities in order to determine if a particular product can indeed be assembled from them.

Heemskerk and Van Luttervelt [21] model the product in a *part relation network*, which is similar to the graph of connection. An important characteristic of this network is that some of its edges represent relationships among components which are not in physical contact, such as *enclosing* or *blocking*. Highlights of the work include the use of a clustering algorithm, which iteratively scans the part relation network and checks for part groups that meet a cluster specification. This pre–processing of the network helps identify subassemblies apriori and therefore reduces the combinatorial complexities involved. Heemskerk and Van Luttervelt also propose the use of heuristics (in the form of accessibility and stability checks) to select good disassembly sequences among clusters. The generated assembly sequences are encoded into an *Assembly State Transition Diagram* (*ASTD*), which is similar to the directed graph of assembly states.

Lee and Gil Shin [46] also use a clustering technique for extracting potential subassemblies of the product a priori. An *Attributed Liaison Graph* is first constructed for the product, whose vertices are the product parts and whose edges represent direct contacts or near contacts among the parts. A *Weighted Abstract*

Liaison Graph (*WALG*) is then obtained by merging sets of mutually inseparable vertices of the *Attributed Liaison Graph* into a single vertex in the WALG. The edges in the WALG are given some weight based on the strength of the corresponding connection and on the cost of the corresponding assembly operation. Preferred subassemblies are then extracted from the WALG, based on the computation of stability and structural preference indices. The recursive extraction of preferred subassemblies results in a single disassembly plan. This plan is encoded in a *Hierarchical Partial Order Graph* (*HPOG*), which is in fact an AND / OR tree that may contain more than one assembly sequence.

Delchambre [16] [17] creates an approximate geometric model of the assembled product using judiciously disposed parallelepipeds. A list of parts which obstruct the removal of each component in each direction, as well as a list of liaisons among the assembly components, are automatically generated by reasoning upon the geometric model. Precedence orders due to geometric, mechanical, stability and technological constraints are then obtained for each liaison. Precedence graphs that satisfy the precedence orders are generated to represent all the feasible assembly sequences of the product.

The author [37] [39] [40] describes a programming system capable of automatically generating robotic assembly sequences. It is a generative robotic assembly process planner. A geometric model of the product to be assembled is first defined in a feature–based product database. Two types of assembly relationships (namely *physical connections* and *spatial constraints*) among components are modelled interactively in graphical relation diagrams. An initial and final relation diagram is used to describe the initial and final state of the world, respectively. The validity of the physical connections defined in the final relation diagram is checked by analyzing the information contained in the feature–based product database. A single robotic assembly sequence is automatically generated, from an analysis of the relational data defined in the final relation diagram. Subassemblies are identified automatically. An interesting characteristic of this system is that the component (or subassembly) to be moved at each step of the sequence is explicitly provided in the assembly sequence formulation.

The approach taken by Ko and Lee [38] to automatically generate assembling procedures starts with the creation of a mating graph of components, where vertices are the component in the assembly, and edges correspond to various mating conditions (such as *fit* or *against*) between components. The component in that graph which has the greatest number of connections with other components is then chosen as a base component, and it is located at a specific vertex of a hierarchical tree. Any components connected to the base, and which do not move relative to it, are then grouped and located in the tree as children vertices of the base vertex. The latter thus becomes the root of a subtree. This process is recursive and uses each grouped component as a new base component. Any component having a relation with a base but which moves relative to it is not connected to the base in the hierarchical tree. It is rather placed at the same level as that base, thus forming the root of another subtree. Part of the assembly sequence is explicitly provided by the hierarchical tree, since it is assumed that any base should be assembled after its descendants. The sequence of assembly among the children of a base vertex, and among bases at the same level in the tree remains to be determined. Both objectives are achieved using interference checking. The final assembly sequences are coded in the form of a standard precedence diagram.

Chang and Wee [10] describe a *knowledge–based* planning system for mechanical assembly using robots. It is divided into two parts: a knowledge base and a control structure. The knowledge base is divided into workpiece structures, assembly operations and assembly principles. The workpiece structures contains all the physical information about the workpieces to be assembled (shape, material, contents...). The assembly operations describe robot actions that change the world state (assemble_new_part, puton, fasten_fixture...). The assembly principles consist of assembly–process–expert rules of thumb (if grasped surface is smooth, then use rubber gripper). The second part of the system, the control structure, uses the information in the knowledge base to formulate an assembly plan. There are two phases in this process: structure analysis and plan generation. The structure analyzer uses the assembly principles to determine an assembly sequence. Here, an assembly sequence simply consists of an ordering of the workpieces to be assembled, without any reference to the operations involved in assembling them. The structure analyzer also posts some constraints on the to–be–determined assembly operations. These constraints are determined based on some of the properties of the workpieces in the assembly sequence. Finally, the plan generator selects the appropriate operations for mating the assembly sequence's workpieces, making sure that the constraints posted by the structure analyzer are satisfied for each operation.

There exists many other works on assembly sequence generation, each having their own advantages and limitations. For example, Miller and Hoffman [57] focussed on assembly planning with fasteners. Hoffman [23] used ray casting techniques in order to find the successive translations from which a component can be disassembled from another. Shpitalni, Elber and Lenz [73] analyzed connectivity graphs to determine possible subassembly candidates for disassembly, with multiple disassembly directions allowed for each candidate. More recently, Zussman, Lenz and Shpitalni [87] described a relational model for the product to be assembled, along with methods by which this model could be automatically generated for a 3–D geometric description. Wilson and Rit [81] described a system which computes automatically the separability of pairs of 2–D subassemblies. They maintain geometric dependencies among parts such that once their removability has been computed in a parent subassembly in which they appear, it does not need to be re–computed in children subassemblies in which they might also appear. The reader can refer to the bibliography at the end of this thesis for references to further works in assembly planning and related research areas.

2.3 NEED FOR RESEARCH

2.3.1 Selection of assembly plans

Research on assembly planning to date has involved four basic issues:

1– how to represent the product to be assembled;

2– how to generate an assembly plan, which involves the determination of precedence constraints;

3– how to represent the resulting assembly plans;

4– how to select the most suitable assembly plan(s).

The first three issues can now be considered a solved problem. The last issue is very complex and remains a research challenge. This complexity stems from the difficult formalization of the decision logic involved in trying to select a particular assembly plan among a list of possible ones.

To illustrate this, consider the two feasible plans in figure 2.16. It is rather easy for the human planner to determine that plan 1 is better than plan 2, since in plan 2 parts B and C in the last operation are very difficult to access. Surprisingly, this notion of "accessibility", so obvious to the human planner, is very hard to model by a computer.

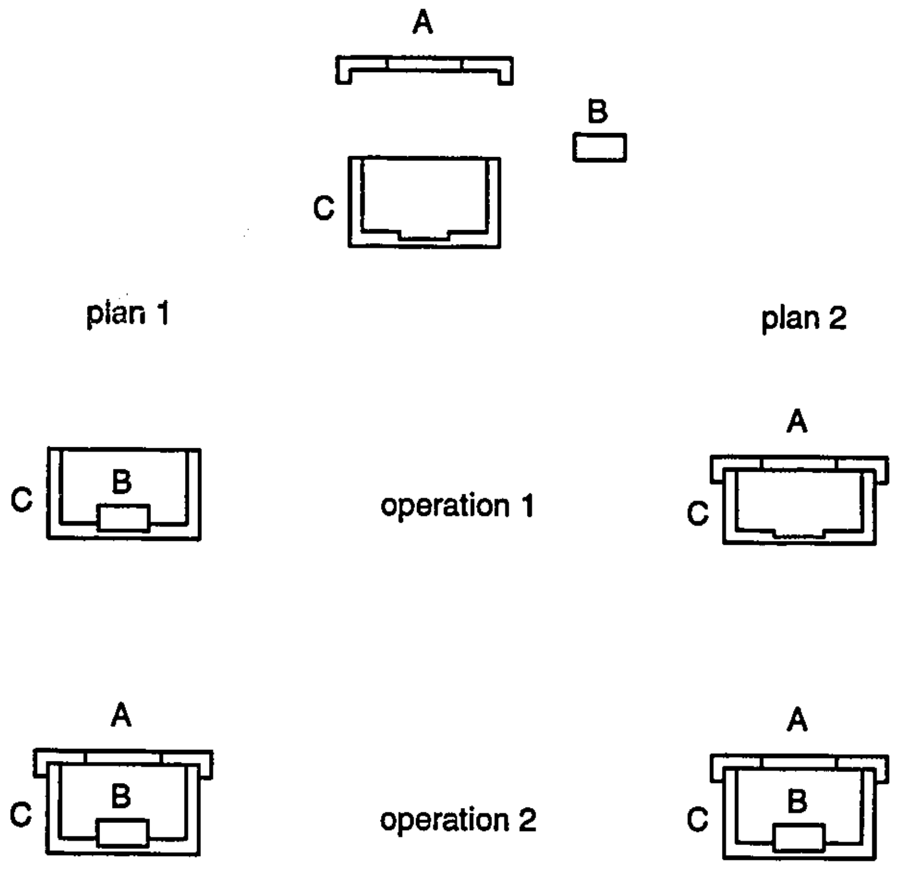


Fig. 2.16: Two feasible plans of a hypothetical product: plan 1 is better with respect to accessibility.

Consider now the two feasible plans in figure 2.17. Once again, it is rather easy to determine that plan 1 is better than plan 2. This time, the choice is motivated by stability considerations, as the first operation in plan 2 would result in a rather unstable configuration. Stability is another notion which is usually very obvious to the human planner but which is still very hard to model by a computer.

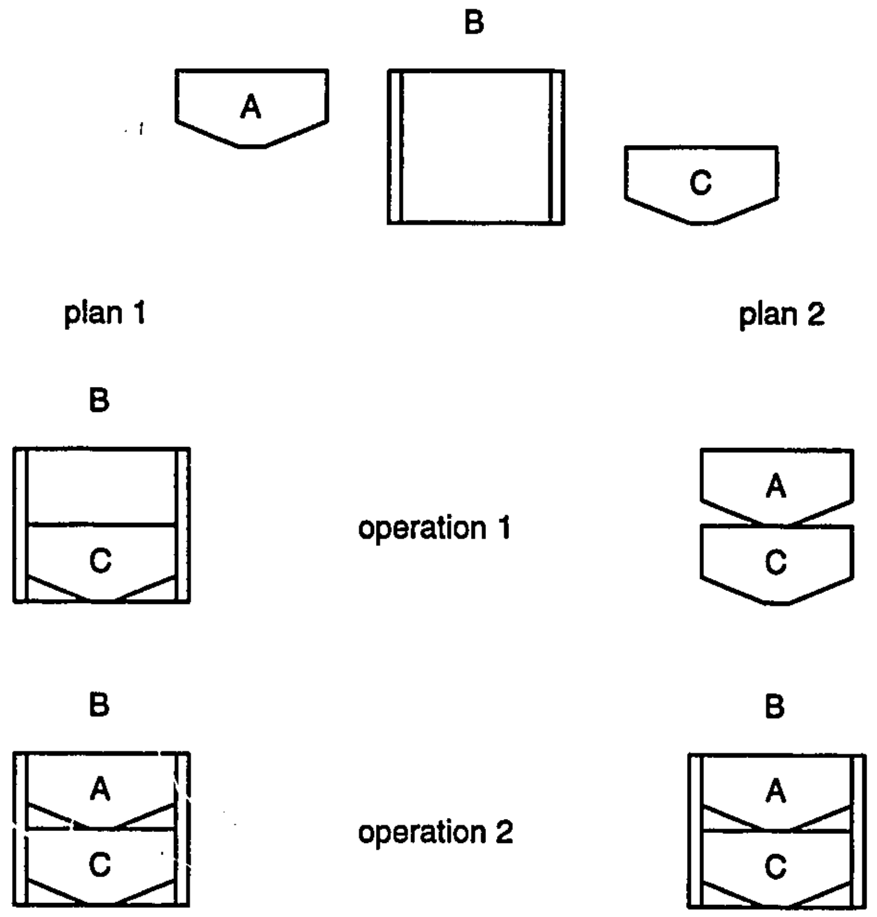


Fig. 2.17: Two feasible plans of a hypothetical product: plan 1 is better with respect to stability.

An important area of research is the identification and formalization of such criteria which directly influences a planner's choice of assembly plans to use in production.

2.3.2 Merging the generation and selection processes

Most research works that addressed the issue of selecting assembly plans are characterized by two different activities:

1– the exhaustive generation of all possible plans, and

2– the selection of the most suitable one(s).

In [13], the set of all possible solutions is generated automatically. Then the user makes all the important decisions for selecting good assembly plans by examining and editing all possible solutions to a smaller size. In [28], the set of all possible solutions is first generated, then the best one selected automatically by evaluating some relevant criteria.

Because all solutions need to be generated prior to the selection, a major problem with this approach is that the medium by which these solutions are represented can grow quite large[15]. The memory required for storing this medium prior to the selection process might not be negligible.

An important refinement is to merge the generation and selection processes into a single generation process. Good (possibly optimal) assembly plans should be generated directly, as opposed to being selected from an exhaustive set of previously generated plans. The advantages of this generative approach are evident savings in both time and memory. Developing the methods for implementing this approach is an important area of research in assembly planning.

15. For example, such mediums include Bourjault's trees, directed graphs of assembly states and AND / OR graphs.

Wolter, in [80], made a first attempt in the adoption of this generative approach. However, Wolter's planner suffers from the important limitation that it cannot handle subassemblies. Better methods for implementing the generative approach must be developed.

2.3.3 Handling arbitrary initial and final assembly states

Most assembly planners use the implicit assumption that the initial state of the problem corresponds to the completely disassembled (or assembled) product, while the final or goal state corresponds to the completely assembled (respectively disassembled) product. Although this assumption is certainly reasonable for the most general case, there are typical applications where an assembly planner must violate this assumption.

One such application consists of determining a disassembly plan for repairing a part or subassembly of a product. In this case, the initial state is the completely assembled product, which complies with the general case. The goal state, however, consists of a partially disassembled product[16]. The planner must have the ability to find the set of operations that leads to this partially disassembled state.

Another typical application is the determination of an error recovery plan for assembly robots. In this case the goal state remains the same, i.e. the product still needs to be completely assembled. The starting point of the new plan, however, is the unpredicted state resulting from the error in executing the assembly plan, which consist of partially assembled product[17]. The planner must have the ability

16. It is assumed here that the repair does not require the total disassembly of the product.
17. It is assumed that the resulting unpredicted state is neither the completely assembled nor completely disassembled product.

to find the new set of operations that leads from this new partially assembled state to the goal.

Methods for handling homogeneously arbitrary initial and final states of the product need to be determined.

2.3.4 Handling multiple products

The near future should see mono and multi–agent flexible assembly cells with the capability of assembling simultaneously two or more products of the same family, or even totally different products. The success of these cells will rely heavily on an efficient sequencing of the respective operations of the different products being worked on. In particular, a typical sequence for one agent would consist of a mix of operations on different products within the cell, for example two consecutive operations on product A, then an operation on product B, then another operation on product A, and so on.

In terms of planning, these cells will require that the planner be able to analyze multiple products simultaneously and interleave their respective required operations in the produced plan. Methods for achieving this type of functionality must be developed.

2.3.5 Identification of the moved and fixed subassemblies

A standard result of the research in assembly planning is that disassembly plans should be generated then reversed to obtain a corresponding assembly plan. In doing so, the disassembly of a large assembly into two smaller ones becomes a problem of finding whether a subassembly possesses a collision–free trajectory along which it can be removed from the larger assembly. Denoting this removeable subassembly by S_1 and the larger assembly by S, the second subassembly S_2 resulting from the disassembly operation is simply given by:

$$S_2 = S - S_1 \tag{2.17}$$

Clearly, if S_1 is removeable along some trajectory, S_2 is also removeable in the opposite trajectory. In the execution of the assembly operation which brings both subassemblies together, generally one of the two subassemblies will be moved while the other will remain fixed.

Most assembly planners do not make the distinction as to which of S_1 or S_2 should be moved or fixed during assembly. This kind of knowledge finds its proper use in robotic and automatic assembly, where switching of the moved and fixed subassemblies in an operation can have a significant effect on its reliability. Methods must be developed to make this important distinction for each assembly operation in an assembly plan.

2.3.6 Elimination of restrictive assumptions

Research in assembly planning is characterized by a list of frequently used assumptions. Among the most widely used is the *dichotomy* of the plans produced. Basically, a plan is dichotomic if each operation in it never involves the simultaneous assembly of more than two subassemblies, where subassembly is defined as a single part or a set of connected[18] parts. In other words, every operation in the plan consists of mating at most two parts, or a part and a set of connected parts, or two sets of connected parts.

Two instances of "products" requiring non–dichotomic operations have been identified by Wolter [84]. The first instance, figure 2.18a, requires the simultaneous

18. "Connected" here refers to the notion of being in physical contact.

and coordinated motion of at least two parts in different directions towards a third one. The second instance, figure 2.18b, requires either the simultaneous and coordinated motion in different directions of subassemblies {B, B1} and {C, C1, C2} towards part A, or the motion of part A towards unconnected subassemblies {B, B1} and {C, C1, C2}. In any case, the dichotomic assumption is violated, since once again at least three subassemblies must be mated in the same operation.

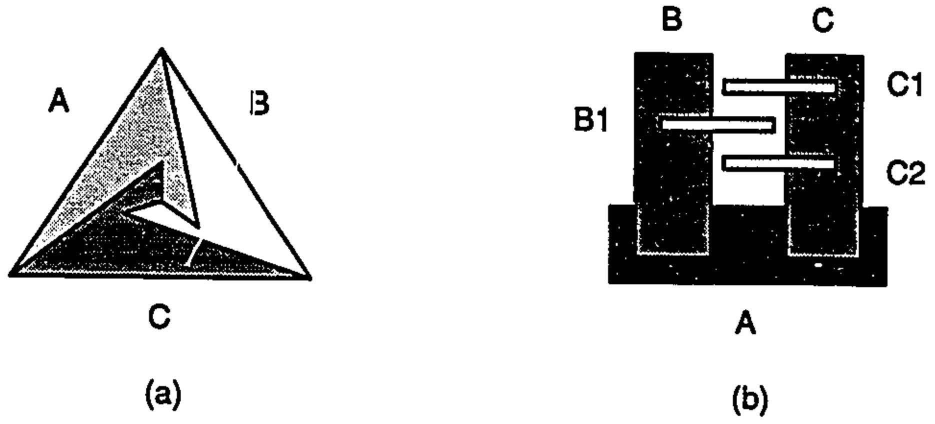
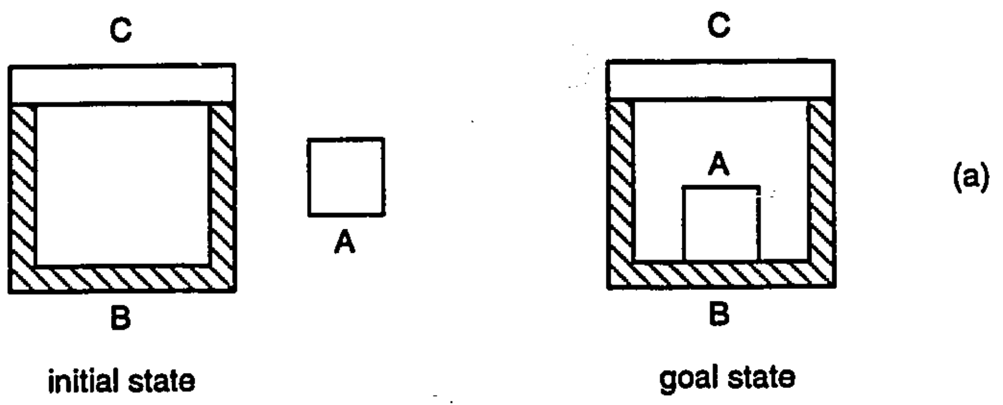
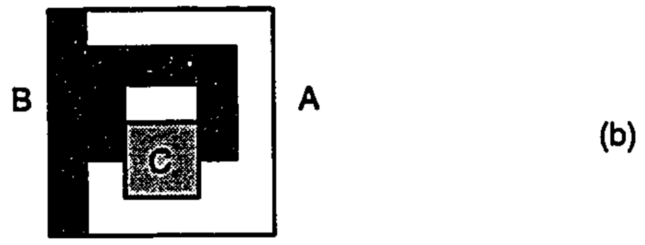


Fig. 2.18: Two products which cannot be built by a dichotomic planner [84].

Most assembly planners are also *monotone*. A plan is considered monotone if the moved parts or subassemblies involved in each of its operations can be moved directly, i.e. using a single translation, to their final position without ever being moved again in subsequent operations. Three consequences of this limitation are given below.





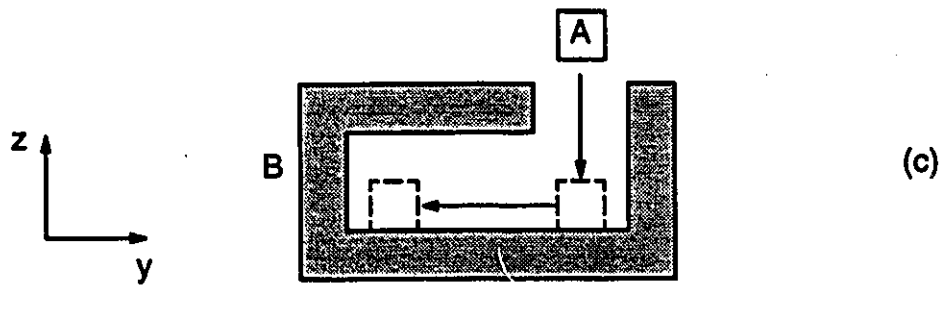


Fig. 2.19: Three example problems which cannot be solved by a monotone assembly planner. Example (b) is taken from [84].

First, a monotone plan never presents the undoing of already done operations. Figure 2.19a shows the initial and goal states of some "product". The transformation of the former state into the latter involves removing part C from part B and later undoing this operation, i.e. place part C back on part B after part A has been inserted into part B. The monotonic assumption is violated in this plan because part C is moved twice in two different operations.

A second consequence of the monotone assumption is illustrated in figure 2.19b. To assemble this "product", part C must first be placed in a temporary position into B, then A is inserted, then C is moved again in to lock part A. The monotonic assumption is violated in this plan as part C is once again moved twice in two different operations.

A third consequence is illustrated in figure 2.19c. In this example, placing part A into its desired position involves a first translation along z– then another translation along y–. This "product" violates the monotonic assumption, since part A cannot be moved directly to its final position by a single translation[19].

Methods of overcoming any of the above two restrictive assumptions without introducing new ones still need to be developed.

2.4 SCOPE OF THE RESEARCH

This research focusses on the first five issues discussed in the previous section, with an accent on the first two, i.e. the formalization of the decision logic involved in the selection of assembly sequences and the merging of the generation and selection processes.

19. To the author's knowledge, only Hoffman's [23] and Sedas and Talukdar's [71] assembly planning systems do not make use of the monotone assumption. The former uses ray casting techniques which take considerable processing time, even for simple problems. The latter is restricted to two dimensional objects.

The formalization of the decision logic first consists of identifying the metrics (called *criteria* in this work) which enable the planner to assess the goodness of the various assembly sequences. Four criteria are proposed and used in this work. They are:

1– number of re–orientations,

2– parallelism,

3– stability, and

4– clustering.

The first three criteria above have already been proposed in the literature. The last one is introduced in this research. Methods of computing automatically the cost associated with these criteria for different asembly sequences is an important objective of this research.

The merging of the generation and selection processes consists of incrementally expanding a search graph until the best solution path is found, instead of expanding all solutions and then traverse them to find the best one. This is accomplished through the use of standard search algorithms which emerged from the AI field. Initially, these algorithms have been developed for efficiently traversing rule networks in expert systems problems. They have been adapted to suit the assembly planning problem tackled by this research.

2.5 THESIS ORGANIZATION

The remaining chapters in this thesis are organized as follows. The graph model of the product, which constitutes the primary input of GAPP, is described in the next chapter. How the problem of generating an assembly process plan can be mapped into a graph search problem is described in chapter 4. Chapter 5 presents some assembly related search constraints which have for effect to reduce the search space. The various criteria by which GAPP can assess the goodness of the assembly sequences it generates are presented in chapter 6. The search methods by which GAPP can sail through the search space are covered in chapter 7. Chapter 8 discusses the results obtained from GAPP using various industrial products. Conclusions and future work are covered in the last chapter. Appendix A presents the product description files generated for the example products used throughout this thesis. Appendix B describes Kruskal's algorithm for finding a maximim spanning tree in a graph. Appendix C describes GAPP's windowing interface. Appendix D describes GAPP's data structures.

CHAPTER 3

GRAPH MODEL OF THE PRODUCT

The primary input to GAPP is a description of the product to be assembled, in terms of the components from which it is made and the assembly relationships between them. This description lends itself to a graph representation of the product to be assembled, which is the focus of this chapter. The first section provides some definitions which are the prerequisite for a mathematical definition of the graph model of the product. The second section describes the role of the developed graph model. The third section describes how different parts of the product can sometimes be merged into a single vertex of the graph model.

3.1 DEFINITIONS

Component is a solid element added to the assembly during the assembly process and which is not a subassembly. A component can be flexible or rigid. The set of all components of a product is denoted by C. Using the air cylinder shown in figure 3.1, a possible component set is as follows: {bearing_o_ring, bearing, body, piston_rod, piston, piston_screw, piston_o_ring, cover_o_ring, cover, cover_screws[20]}.

20. Although there are 8 screws, all of them have been combined into the single component called "cover_screws". Section 3.3 will discuss the benefits and implications of merging different parts into a single component.

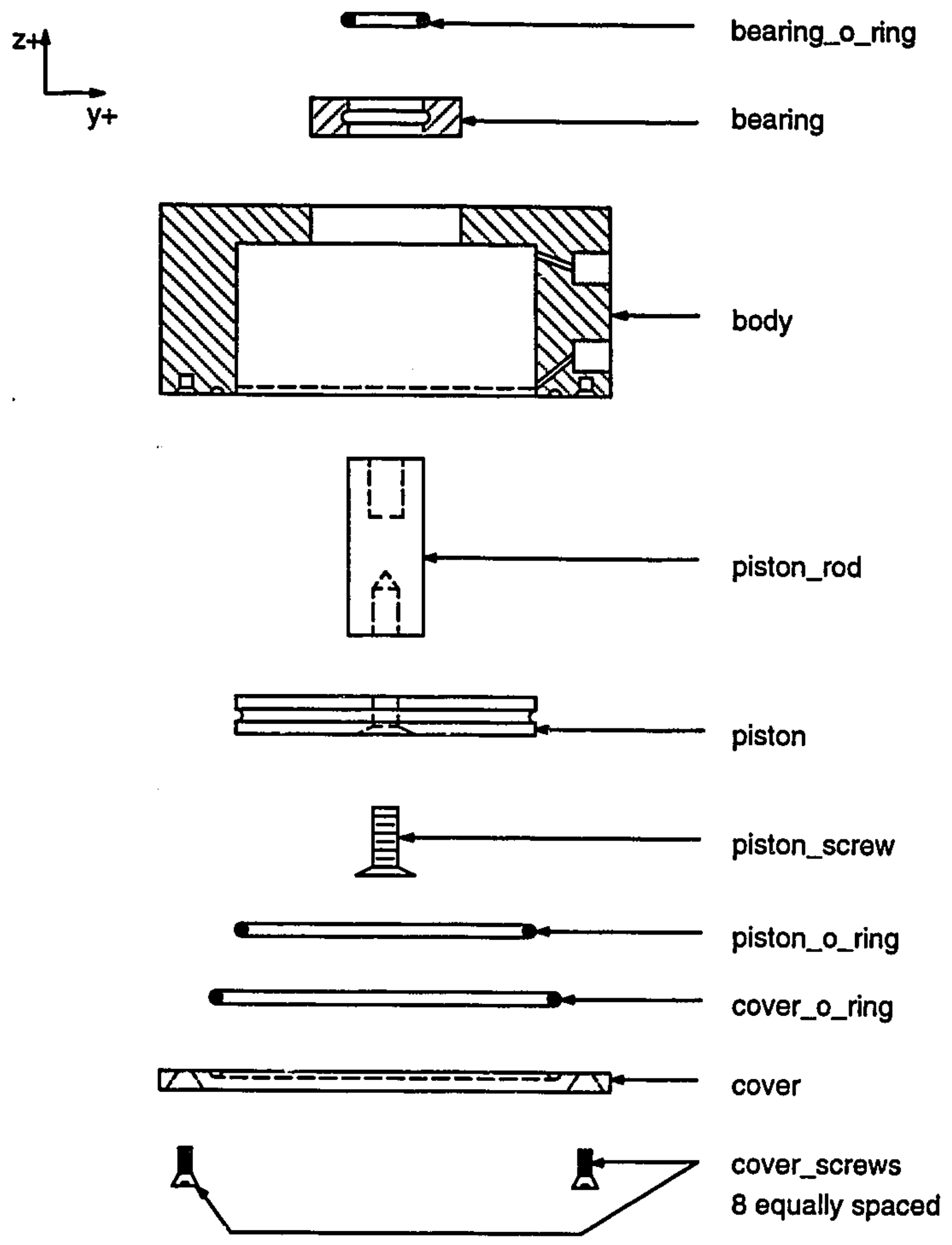
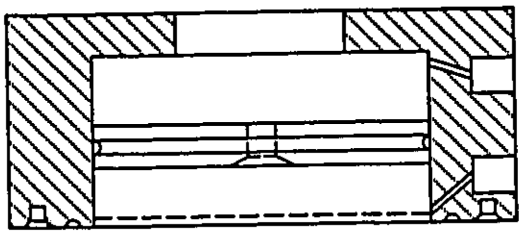
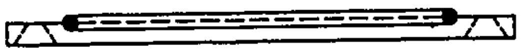


Fig. 3.1: Exploded view of an air cylinder assembly.

Two components have a **contact** relationship between them if they are in constant physical contact in the assembled product[21] and if, when holding one of them while the other remains moveable, there exist an orientation which will cause the moveable component to lose its physical contact with the fixed one. Contact is a symmetric binary relation on C. Therefore the two incident components in a contact relation are necessarily an element of $C \times C$ where $\times$ denotes the cartesian product of two sets. The set of all contacts of a product is denoted by C'. Figure 3.2 shows examples of some air cylinder components which have a contact relationship between them.



piston and body

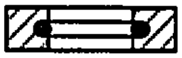


cover and cover_o_ring

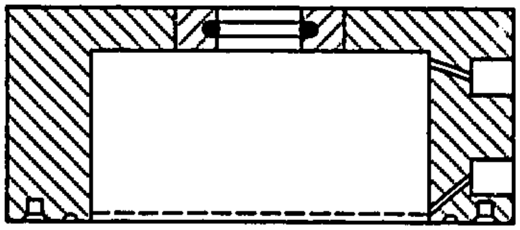
Fig. 3.2: Examples of air cylinder components having a contact relation between them.

21. "Constant physical contact" means that contact is maintained throughout the normal functioning of the product. For example, the piston and cover of the air cylinder do not have a contact relationship between them since their physical contact is intermittent.

Two components have an **attachment** relationship between them if they are in constant physical contact in the product, if they do not move relative to each other during the normal functioning of the product and if, when holding one of them while the other remains moveable, there does not exist any orientation which will cause the moveable component to lose its physical contact with the fixed one. Attachment is a symmetric binary relation on C. Therefore the two incident components in an attachment relation are necessarily an element of $C \times C$. The set of all attachments of a product is denoted by A. Figure 3.3 shows examples of air cylinder components which have an attachment relationship between them.



bearing and bearing_o_ring

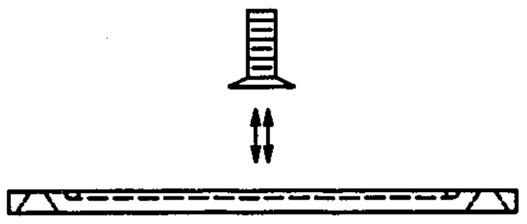


bearing and body (press fit)

Fig. 3.3: Examples of air cylinder components having an attachment relation between them.

Two components have a **blocking** relationship between them if they are not in constant physical contact in the product and if a linear translation of one of them in one of its possible disassembly directions[22] results in a collision with the other. Blocking is a symmetric binary relation on C. Therefore the two incident components of a blocking relation are necessarily an element of $C \times C$. The set of all blockings of a product is denoted by B. Figures 3.4, 3.5, 3.6 and 3.7 will help clarify this most important definition.

Figure 3.4 shows a typical example of a blocking relation using two air cylinder components. These components never enter in physical contact in the normal functioning of the product[23]. A possible disassembly direction of the piston_screw is z–. Moving this component in this direction results in a collision with the cover. Therefore these two components have a blocking relationship between them.



piston_screw and cover

Fig. 3.4: A typical example of two components having a blocking relation between them.

22. The set of possible disassembly directions of a component must necessarily be a subset of the set {x+, x–, y+, y–, z+, z–}.
23. That the piston_screw and cover never enter in physical contact can be verified in figure 3.8.

Figure 3.5 shows two examples of air cylinder components which have an intermittent contact between them in the normal functioning of the product. Therefore these components have a blocking relationship between them.

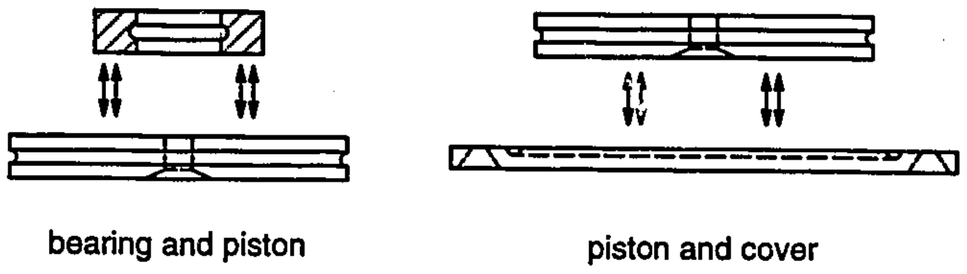


Fig. 3.5: Examples of air cylinder components having a blocking relation due to an intermittent contact between them.

Figure 3.6 shows further examples of air cylinder components having a blocking relation between them.

Figure 3.7 shows examples of air cylinder components *not having* a blocking relationship between them. In the first example, the piston_screw and cover_o_ring are not in physical contact. Linearly translating any of these two components in one of their possible disassembly directions does not result in a collision with the other. In the other example, the piston_rod and body are not in physical contact. Linearly translating the piston_rod in either x+, x–, y+, or y– would result in a collision with the body. However these directions are not possible disassembly directions of the piston_rod.

The important role of blocking relations will become clear in chapter 5 when the notion of the geometric feasibility of an assembly operation will be discussed.

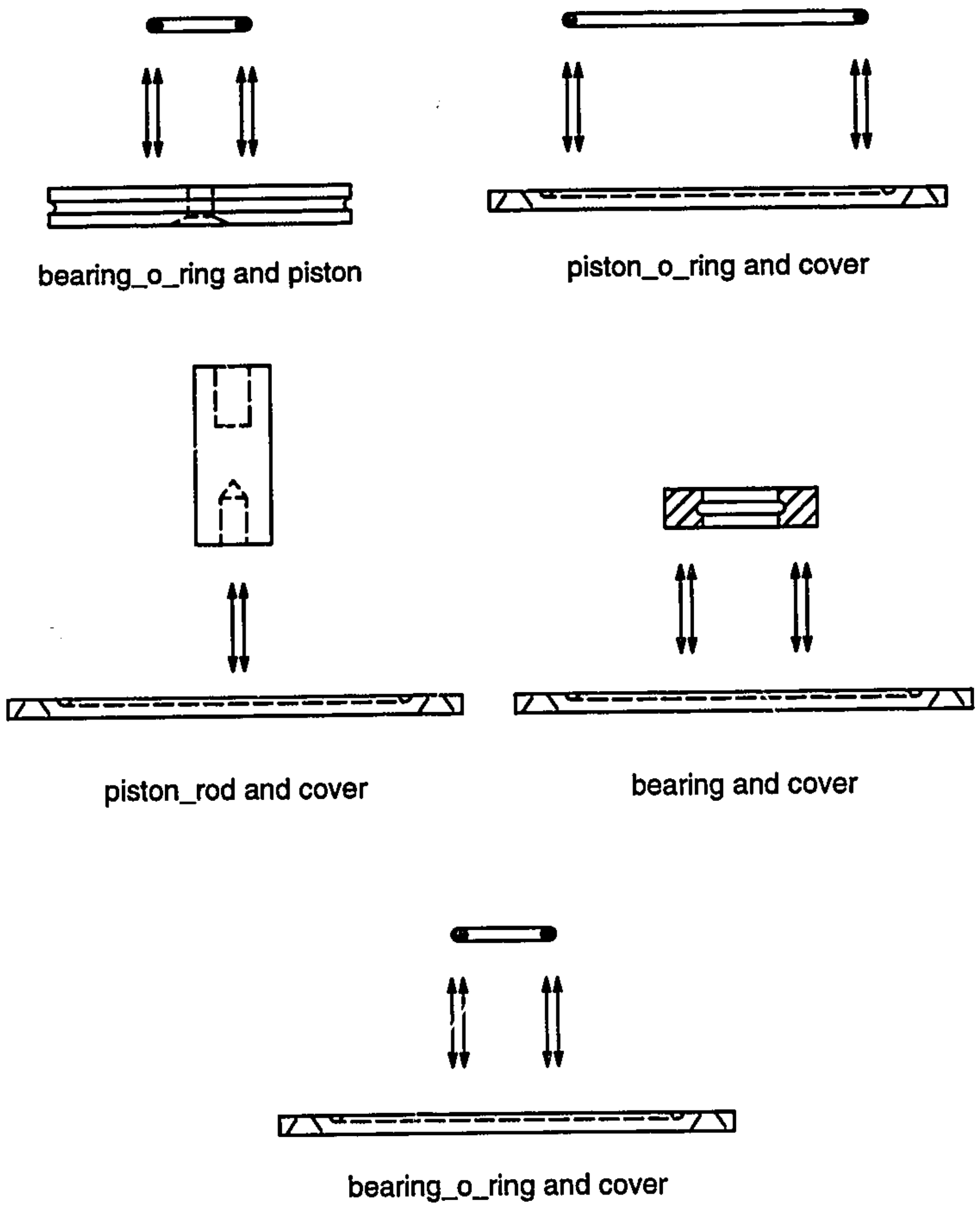
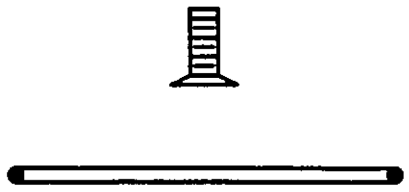
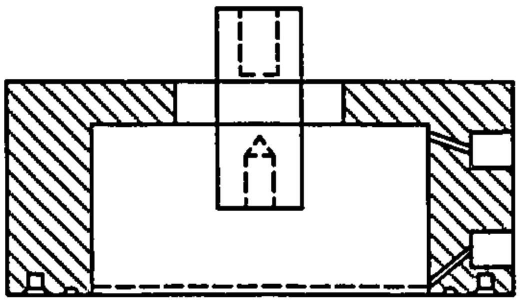


Fig. 3.6: Further examples of air cylinder components having a blocking relation between them.



piston_screw and cover_o_ring



piston_rod and body

Fig. 3.7: Examples of air cylinder components *not having* a blocking relation between them.

Two components have a **relation** between them if they either have a contact, attachment or blocking relationship between them. The set of all relations of a product is denoted by R and must satisfy:

$$R = C' \bigcup A \bigcup B. \tag{3.1}$$

$\Psi()$ is an incidence function that associates each relation with its two incident components. The domain of this function is R and the range are elements of $C \times C$. For example, if the relation in figure 3.4 is denoted by "r":

$$\Psi(r) = (piston\_screw, cover). \tag{3.2}$$

From these definitions, the graph model of the product can now be defined as the simple graph $G = \{ C, R, \Psi_G \}$. Figure 3.9 shows the air cylinder's graph model.

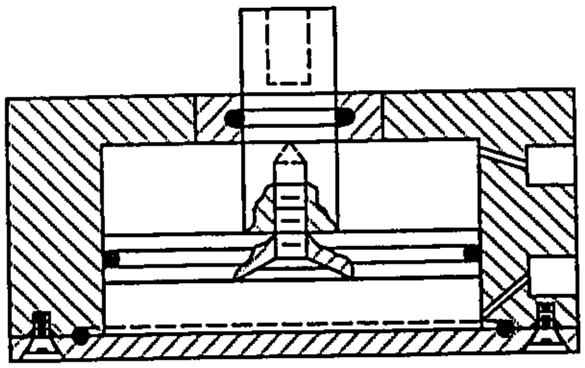


Fig. 3.8: Air cylinder assembly drawing.

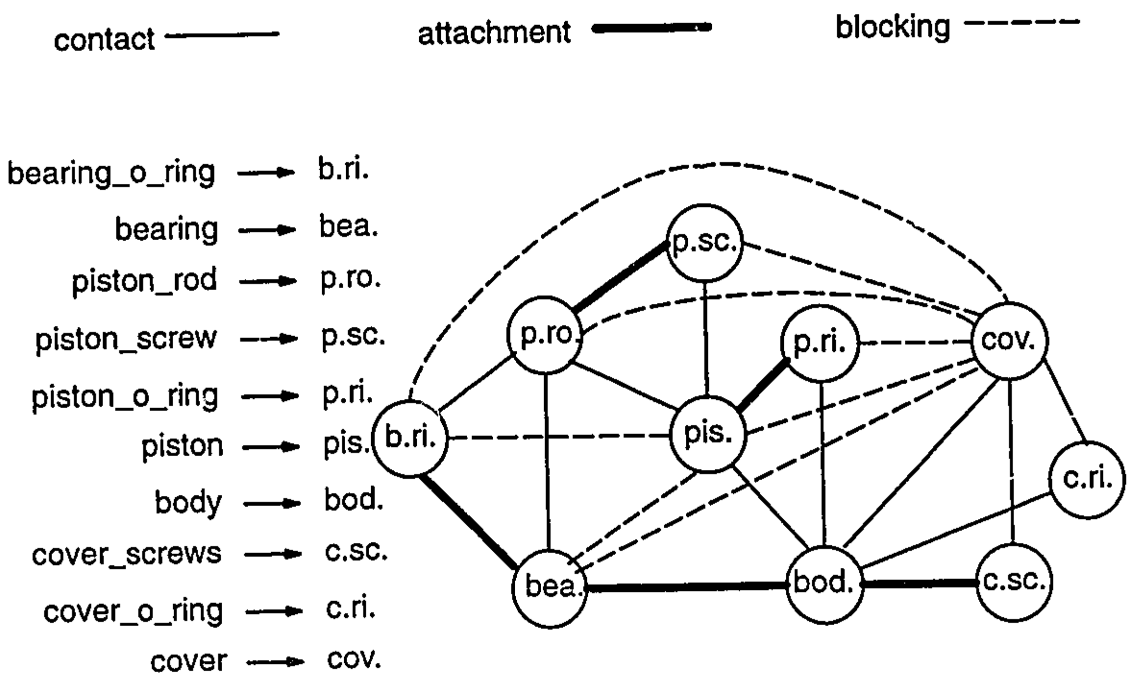


Fig. 3.9: Air cylinder graph model.

The product's graph model, as defined so far, is similar to most other models used by other researchers in assembly planning. Following are some further definitions, which help distinguish GAPP's graph model from other models in the literature.

Name() is a function that associates a name with a component. The domain of this function is C and the range is any combination of the letters in the alphabet and the numbers 0 through 9.

Goal() is a function that associates a goal flag with a relation. The domain of this function is R and the range is the set { yes, no }. If the goal flag of a particular relation is set to "yes", this means that this relation should remain unbroken in the final state resulting from the disassembly process applied by GAPP. This feature will be discussed in more details in chapter 4.

Directions() is a function that associates an incident component of a relation with the list of all its possible disassembly directions. The domain of this function is $R \times C$ and the range is any combination of { x+, x–, y+, y–, z+, z– }. For the contact relation between the piston and body in figure 3.2:

$$directions(\ (piston,\ body),\ piston\ ) = \{\ z-\ \} \qquad (3.3)$$

$$directions(\ (piston,\ body),\ body\ ) = \{\ z+\ \} \qquad (3.4)$$

Type() is a function that associates a type with a relation. The domain of this function is R and the range is the set { contact, attachment, blocking }.

Operation() is a function that associates an assembly operation with a relation. The domain of this function is R and the range is the set { against, fit, fit_and_twist, press, crimp, screw, rivet, weld, solder }

Restricted() is a function that associates a list of restricted components with a relation. A restricted component "c" for a relation "r" is one which should not be part of any of the two subassemblies involved in the establishment of "r", otherwise it would greatly complicate, or even prevent, the establishment of "r". The domain of this function is R and the range is $\mathcal{P}(C)$ where $\mathcal{P}$ denotes the power of a set, i.e. the set of all subsets of a set. Chapter 5 will discuss this feature in more details when the notion of constraints is introduced.

Moved() is a function that associates a moved component in a relation. The domain of this function is R and the range is C. Assume a product consists solely of the two incident components of a relation. Then the moved component is generally the smaller of the two components in the relation or the one which has the best grasping characteristics of the two.

Fixed() is a function that associates a fixed component in a relation. The domain of this function is R and the range is C. Assume a product consists solely of the two incident components of a relation. Then the fixed component is generally the larger of the two components in the relation or the one which has the best fixturing characteristics of the two.

Ambiguous() is a function that associates an ambiguity flag with a relation. The domain of this function is R and the range is the set { yes, no }. If a relation has its ambiguity flag set to "yes", it means that its moved and fixed component can be interchanged. In other words, there is no clear distinction as to which component should be moved and which should be fixed when establishing the relation between them.

To distinguish between the moved and fixed component of a relation, every relation in the graph model is converted into a directed edge. Such an edge is called an **arc** and is represented by an arrow in the product's graph model. By convention, the fixed component is the one with the incoming arrow head. The set of all arcs is denoted by R'.

From these further definitions, the product's graph model can now be re–defined as the simple and directed graph D = { C. R', Ψ_D}. From a graph–theoretic point of view, G is the *underlying graph* of D and D is an *orientation* of G.

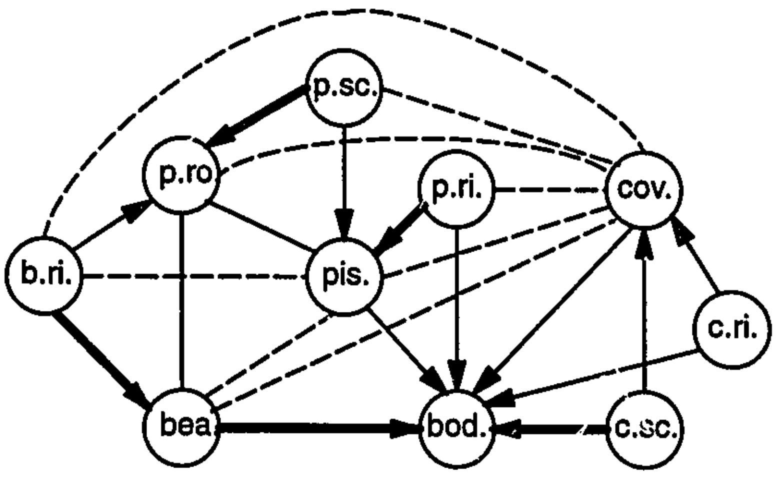


Fig. 3.10: Air cylinder directed graph model.

Figure 3.10 shows the the air cylinder's directed graph model. The directed arc from vertex labelled "b.ri." to vertex labelled "bea" means that the bearing_o_ring should be moved and assembled with the fixed bearing[24], in the case where the product would consist solely of these two components. If a decision cannot be taken as to which component should be moved and which should be fixed in a relation, the arc's direction is irrelevant. These arcs are pictorially represented as edges with no

24. Note that it would not make much sense to do the reverse, i.e. fixing the bearing_o_ring and moving the bearing towards it to establish the connection between both.

arrow heads. When the ambiguity flags of each arc in D is set to "yes", D becomes isomorphic to G and both representations can be used equivalently.

3.2 ROLE OF THE GRAPH MODEL

Given some graph model of a product as defined in the previous section, GAPP tries to find a totally ordered set of assembly operations which results in the establishment of all the relations in the graph model. Therefore, underlying the formal definition of the product's graph model is the important assumption that it must represent the state[25] of the product resulting from the assembly process.

Most of the time, this state is the one represented on the assembly drawing like the one in figure 3.8 for the air cylinder. However, it is important to understand that the product's graph model need not necessarily be tied to its assembly drawing, where all connections among components are established. For example, there might be applications where the result of the assembly process should be a set of distinct subassemblies[26] instead of a complete product. An unconnected[27] graph model which reflects this situation can be generated instead.

Figure 3.11 illustrates a hypothetical final configuration of the air cylinder resulting from some assembly process. Figure 3.12 shows the corresponding graph model. Feeding this new graph to GAPP, assembly operations which assemble two components each in a different subassembly, for example the cover and the body in figure 3.11, will not be generated. The graph model of the product is used to reflect some final configuration of the product resulting from some assembly process, in

25. The concept of the state of an assembly will be formally defined in chapter 4.
26. A subassembly will be formally defined in chapter 4.
27. Connection will be formally defined in chapter 4.

terms of the components' relations established. Whether this model reflects a totally or partially assembled configuration is not a problem for GAPP.

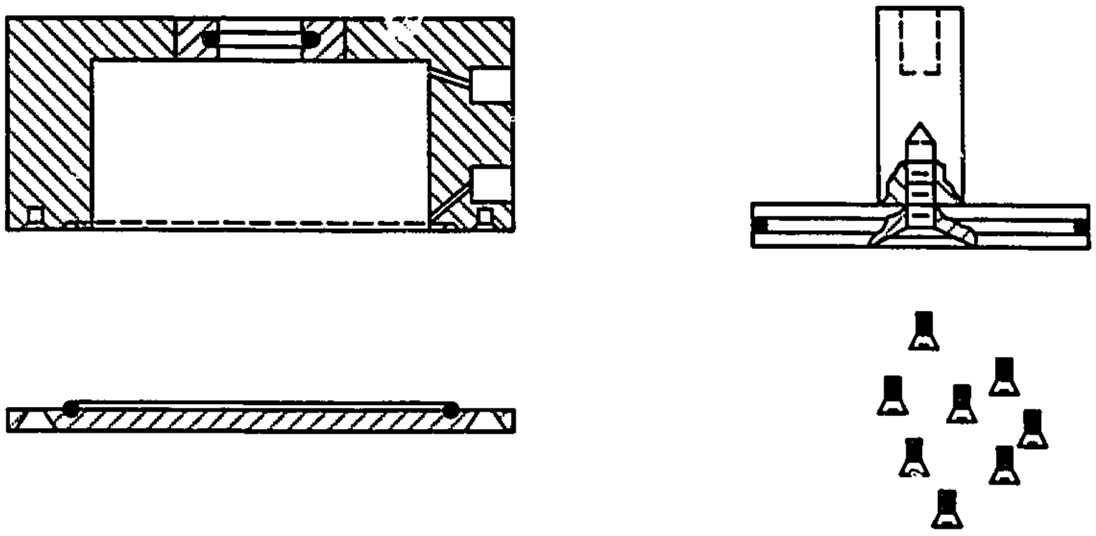


Fig. 3.11: A hypothetical final configuration of the air cylinder.

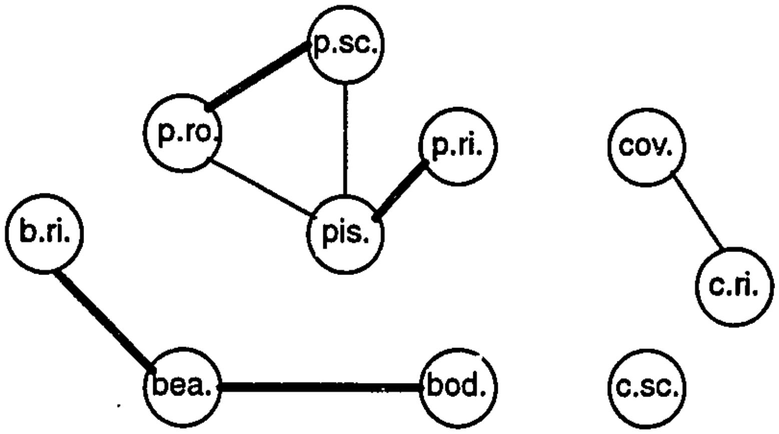


Fig. 3.12: Graph model corresponding to the final configuration in figure 3.11.

3.3 MERGING COMPONENTS

Although the identification of the components of a product is a fairly straightforward process, it is still worth presenting. Consider the parts which are listed in figure 3.1 for the air cylinder example. These parts constitute the components of the product. According to the definition of a component, it is understood that all these parts are individually mated at some point in the assembly process. Each part is also associated with a specific vertex in the air cylinder graph model.

Assume for example that the piston, piston_screw, piston_rod and piston_o_ring are pre–assembled and enter the assembly process as a whole. In this case, one is not concerned with the sequence of assembly of these parts. If the same graph model as in figure 3.10 is kept, the assembly sequence generated by GAPP will contain some operations which concerns the order of assembly of these four parts.

To remedy this situation, one solution is to merge these parts into a single component (call it piston_sub) and associate it with a single vertex in the graph model. Figure 3.13 shows the new graph model, in which the vertex labelled "p.su" represents the piston subassembly[28]. As a consequence, GAPP looks at the four parts in the piston subassembly as a single component and does not find their sequence of assembly[29].

28. From a graph–theoretic point of view, the edges between the piston_rod, piston_o_ring, piston_screw and piston in figure 3.10 that are not part of figure 3.13 are said to have been "contracted". As such, merging two incident vertices is equivalent to contracting the edges between them.

29. As mentioned earlier, this merging concept had already been applied to the model in figure 3.10, where the eight cover_screws had been combined into a single component labelled "c.sc." in the graph. As a result, GAPP will not try to find the individual sequence of assembly for these eight screws.

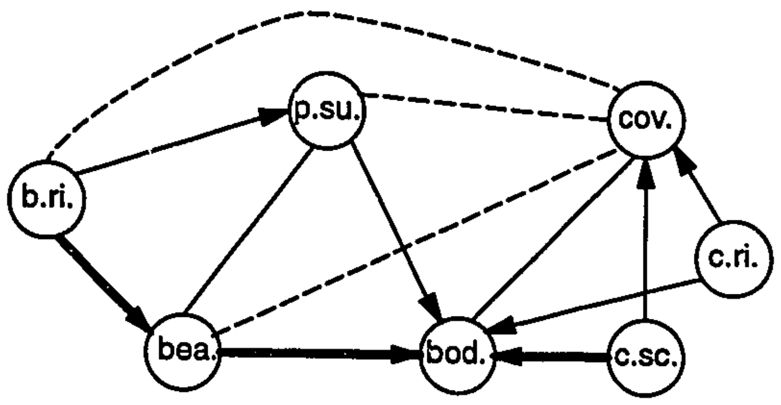


Fig. 3.13: Air cylinder's directed graph model in the case where the piston, piston_rod, piston_screw and piston_o_ring have been combined into a single component.

This type of merging reduces the component count, which in turn significantly reduces the combinatorics of generating assembly sequences. However, one must keep in mind that the order of assembly of merged components is not returned by GAPP.

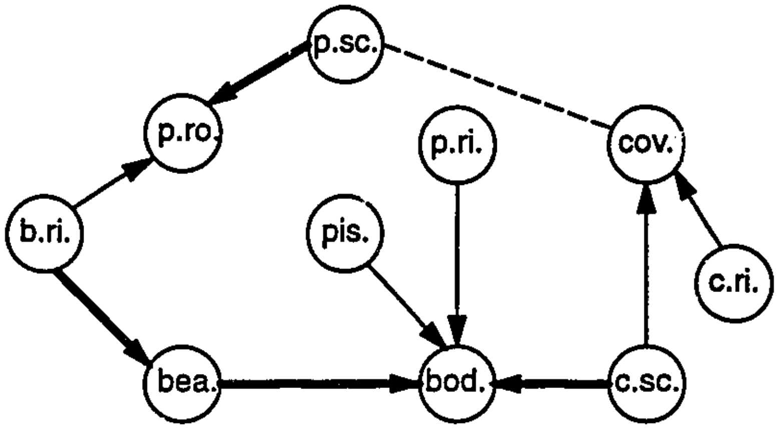
CHAPTER 4

PROBLEM FORMALIZATION

This chapter describes how the generation of an assembly sequence can be mapped into a graph search problem. The first section provides some definitions which are the prerequisite to a formalization of this mapping. The second section formalizes the problem. The third section justifies the use of a disassembly approach to the assembly planning problem. Section four describes how the cutsets of the product's graph model are generated and updated. Section five highlights some important benefits of the adopted approach.

4.1 DEFINITIONS

Let $D = \{ C, R', \Psi_D \}$ be the directed graph model of some product; let H be a graph with vertex set V, edge set E and incidence function Ψ_H. H is a **subgraph** of D if $V \subseteq C$, $E \subseteq R'$ and Ψ_H is the restriction of Ψ_D to E. Note that the use of $\subseteq$ implies that H is also considered to be a subgraph of D for the special case where $H = D$. Figure 4.1 shows two subgraphs of the air cylinder's graph model in figure 3.10.



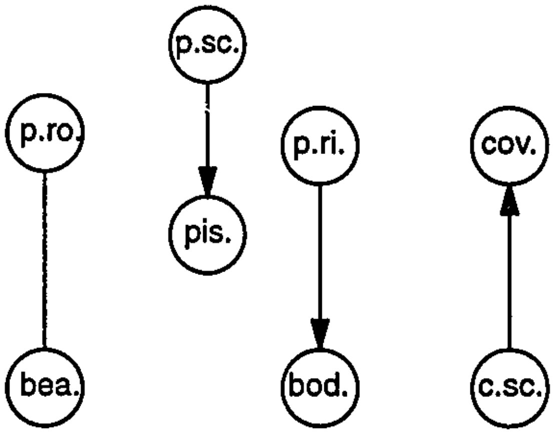


Fig. 4.1: Two subgraphs of the air cylinder graph model.

Let H = { C, R', Ψ_H } be any subgraph of D and let H' be a subgraph of H with vertex set C, edge set R'–B and incidence function $\Psi_{H'}$ where $\Psi_{H'}$ is the restriction of Ψ_H to R'–B. H' is called the **non–blocking subgraph** of H and H is called the **underlying graph** of H'. Figure 4.2 shows the non–blocking subgraph of the air cylinder's graph model.

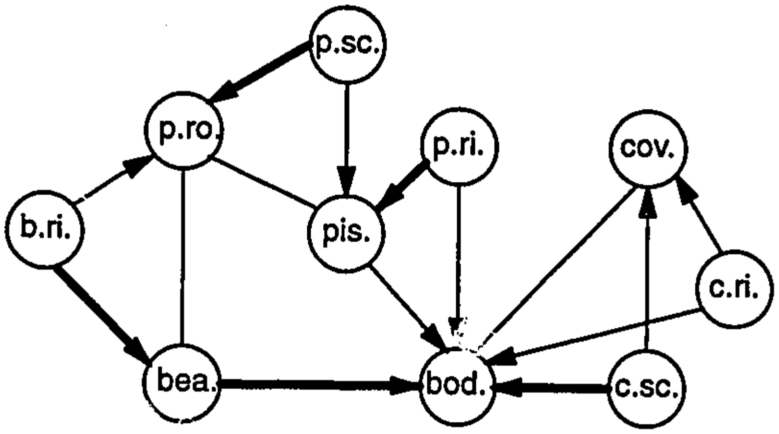


Fig. 4.2: Air cylinder's non–blocking subgraph.

Let H = { C, R', Ψ_H } be any subgraph of D and let H' = { C, R'–B, $\Psi_{H'}$ } be the non–blocking subgraph of H. Two vertices of H are **connected** if and only if there exists a path between them in H'. Connection is an equivalence relation on C which partitions this set into non–empty mutually exclusive subsets[30]. Two vertices are connected if and only if they belong to the same subset. If all the vertices of H' are connected, then H' and H are also connected. It can be readily verified that all the vertices in the non–blocking subgraph in figure 4.2 are connected. Therefore this graph is itself connected, and by definition so is its underlying graph, the graph

30. Two sets are mutually exclusive if their intersection is empty.

model of the air cylinder in figure 3.10. Figure 4.3 shows an unconnected subgraph of the air cylinder's graph model (top). This subgraph is unconnected because its non–blocking subgraph is unconnected (bottom).

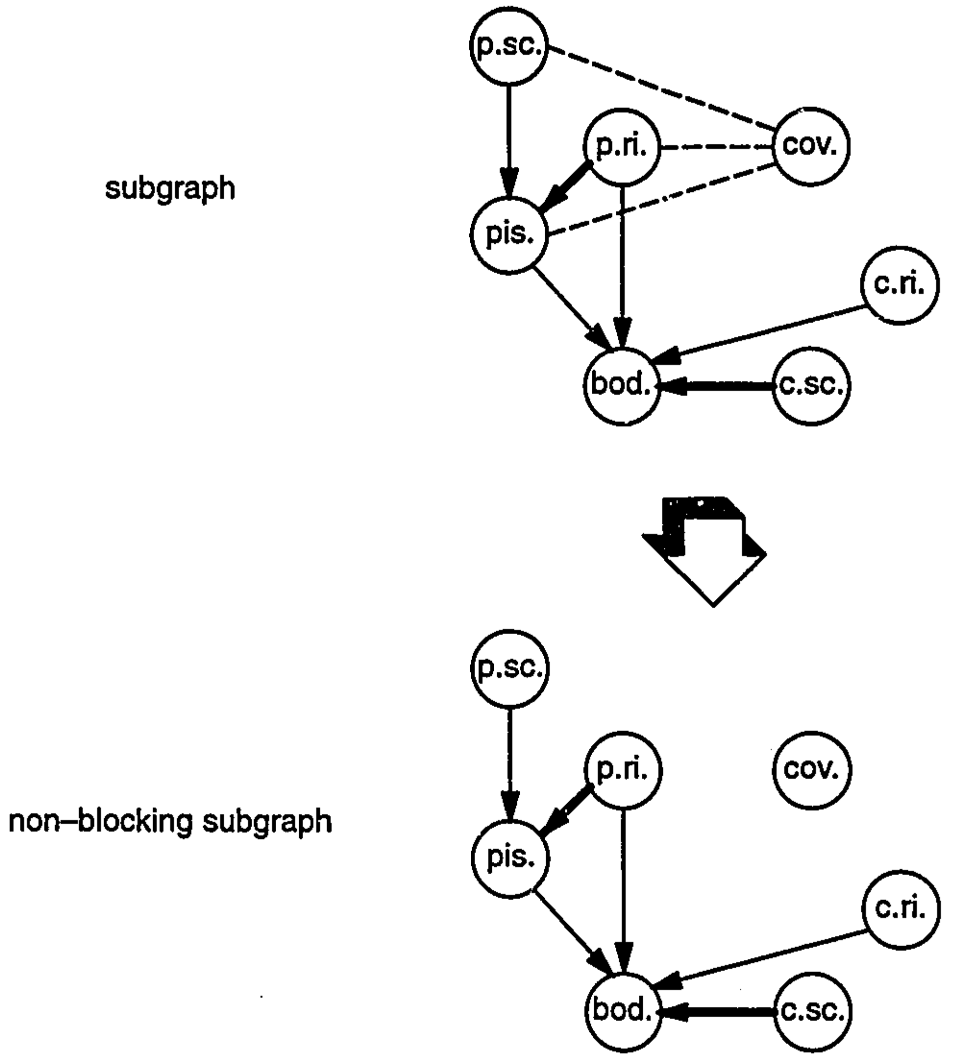


Fig. 4.3: An unconnected subgraph of the air cylinder's graph model (top) along with its unconnected non–blocking subgraph (bottom).

Let D = { C, R', Ψ_D } be the graph model of some product; let V be a subset of C; and let H be the subgraph of D whose vertex set is V and whose edge set is the set of those edges of D that have both ends in V. H is called the **induced subgraph** of D, denoted by D[V], if and only if H is connected. Figure 4.4 shows two induced subgraphs of the air cylinder's graph model. Figure 4.5 shows two other subgraphs which are not induced subgraphs.

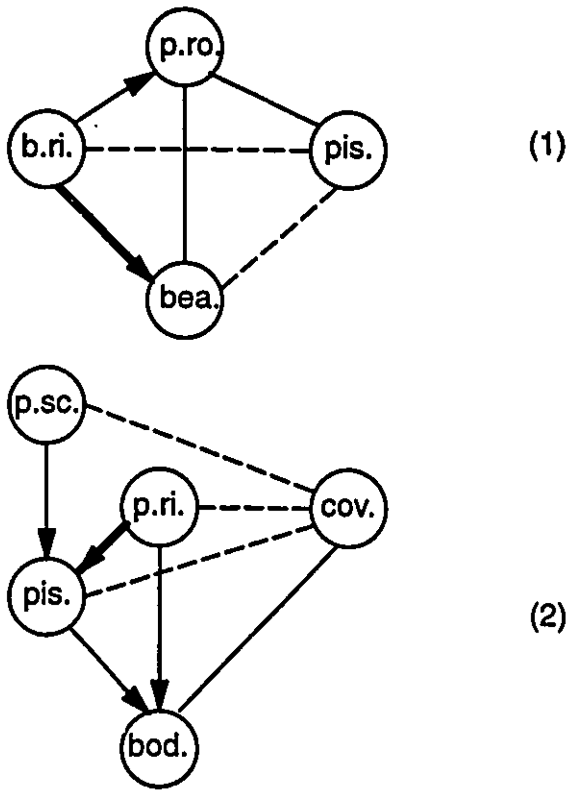
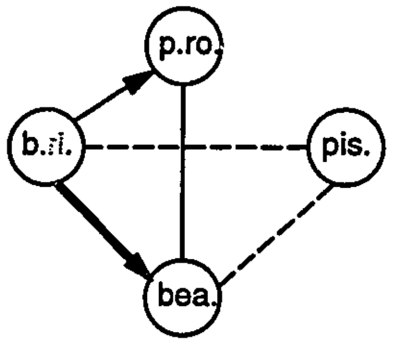
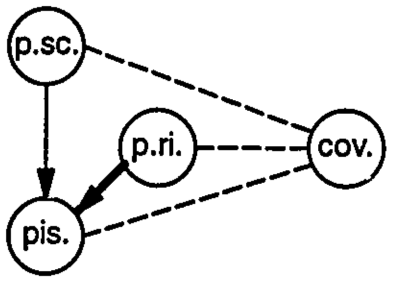


Fig. 4.4: Two induced subgraphs of the air cylinder's graph model: D[bearing_o_ring, bearing, piston_rod, piston] and D[piston_screw, piston, piston_o_ring, body, cover].



(1)



(2)

Fig. 4.5: Two subgraphs of the air cylinder's graph model which are not induced subgraphs: the top one has a missing edge (piston_rod, piston), while the bottom one is not connected.

Let D = { C, R', Ψ_D } be the graph model of some product and let W = { H_1, H_2, ..., H_n } be a set of mutually exclusive induced subgraphs[31] of D with component sets C_1, C_2, ..., C_n, respectively. W is an **assembly state** of the product represented by D if and only if:

$$C_1 \cup C_2 \cup \ldots \cup C_n = C. \tag{4.1}$$

Figure 4.6 shows two possible air cylinder assembly states.

31. Two subgraphs are mutually exclusive if the intersection of their vertex set is empty.

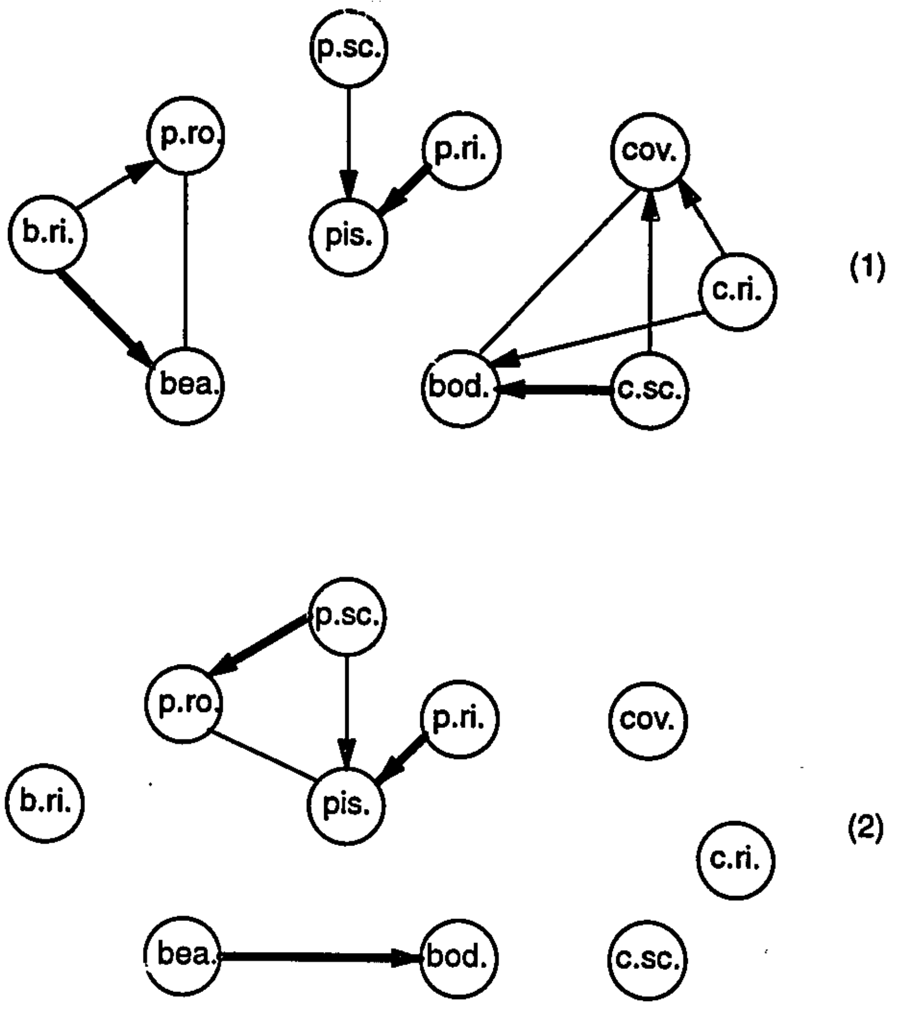


Fig. 4.6: Two possible air cylinder assembly states.

A **cutset** is a set of edges of an assembly state, the removal of which yields a new assembly state containing one and exactly one more induced subgraph than the original state.

Figures 4.7, 4.8 and 4.9 will be used to clarify this most important definition.

In figure 4.7, a set of seven edges has been removed from the air cylinder's completely assembled state which contains one induced subgraph. This removal yields a new state with two induced subgraphs: D[bearing, bearing_o_ring] and D[piston_rod, piston, piston_screw, piston_o_ring, body, cover, cover_screws]. Therefore this set of edges is a cutset.

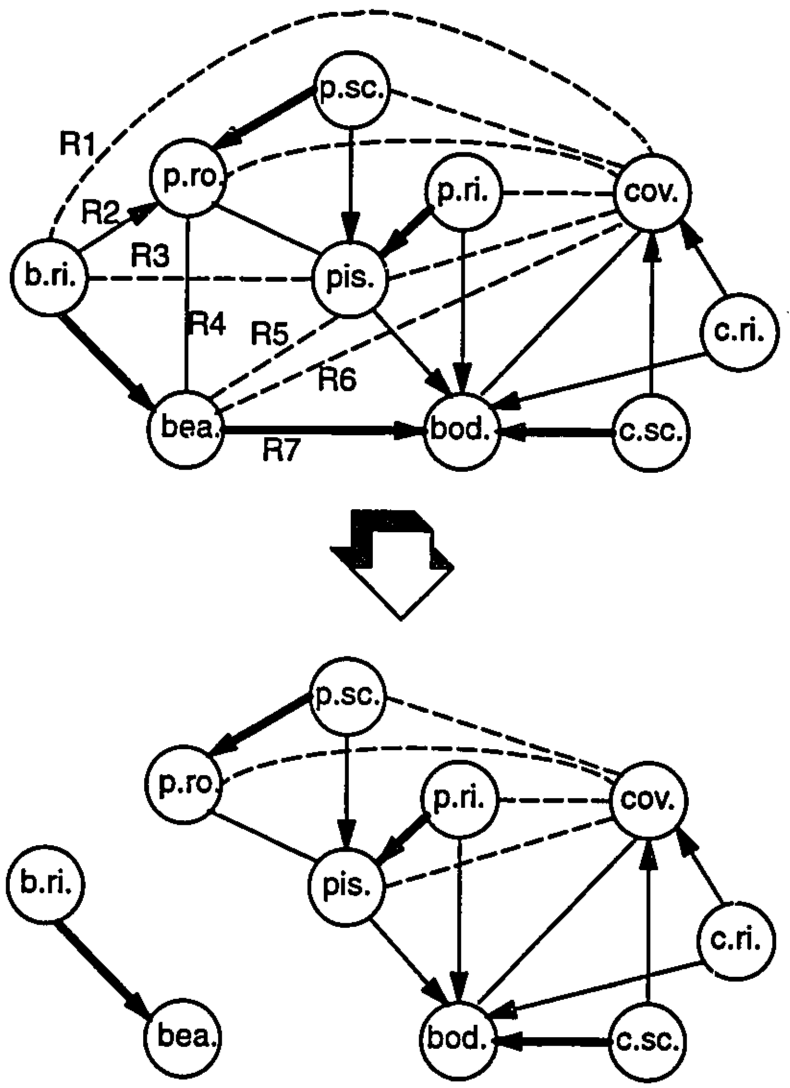


Fig. 4.7: A cutset in the air cylinder's completely assembled state: {R1, R2, R3, R4, R5, R6, R7}.

In figure 4.8, a set of seven edges have been removed from an air cylinder's arbitrary state which contains two induced subgraphs. The removal of this set of edges yields a new state with three induced subgraphs: D[bearing_o_ring, bearing], D[piston_rod, piston, piston_o_ring, piston_screw, body] and D[cover, cover_o_ring, cover_screws]. This set of edges is therefore a cutset.

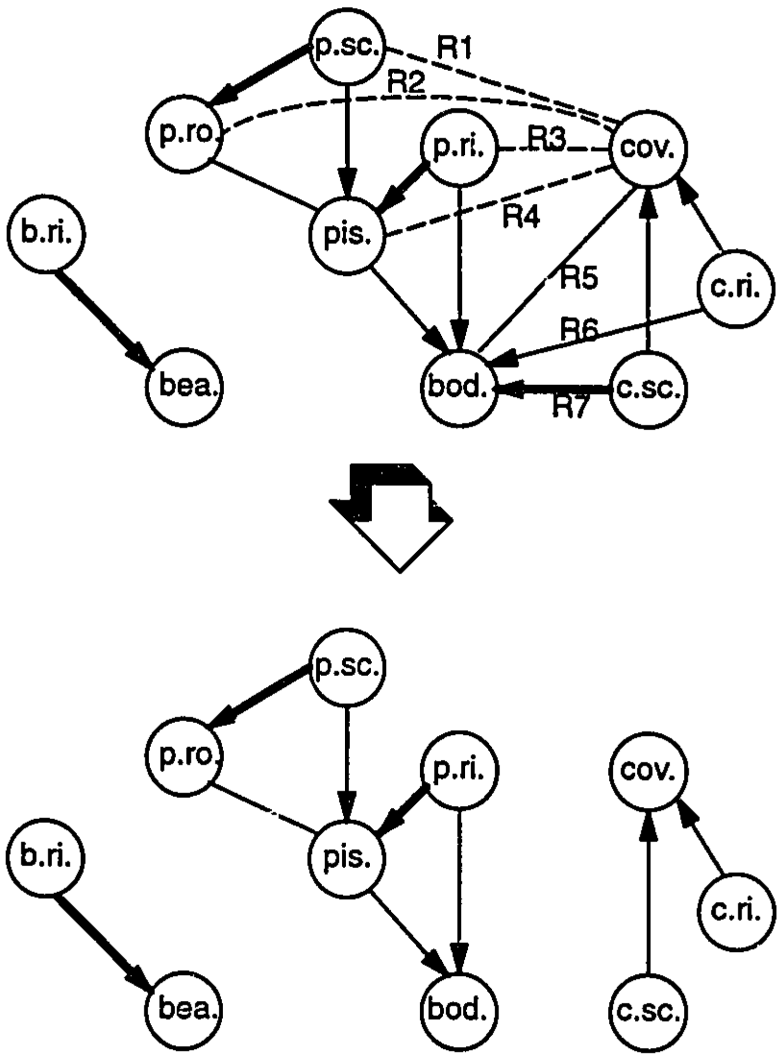


Fig. 4.8: A cutset in an air cylinder's arbitrary state: {R1, R2, R3, R4, R5, R6, R7}.

In figure 4.9, a set of six edges has been removed from the air cylinder's completely assembled state which contains one induced subgraph. The removal of this set yields a new state with three mutually exclusive induced subgraphs: D[bearing_o_ring], D[bearing, body, piston, piston_o_ring, piston_rod, piston_screw, cover, cover_o_ring] and D[cover_screws]. This violates the definition. Therefore, this set is not a cutset.

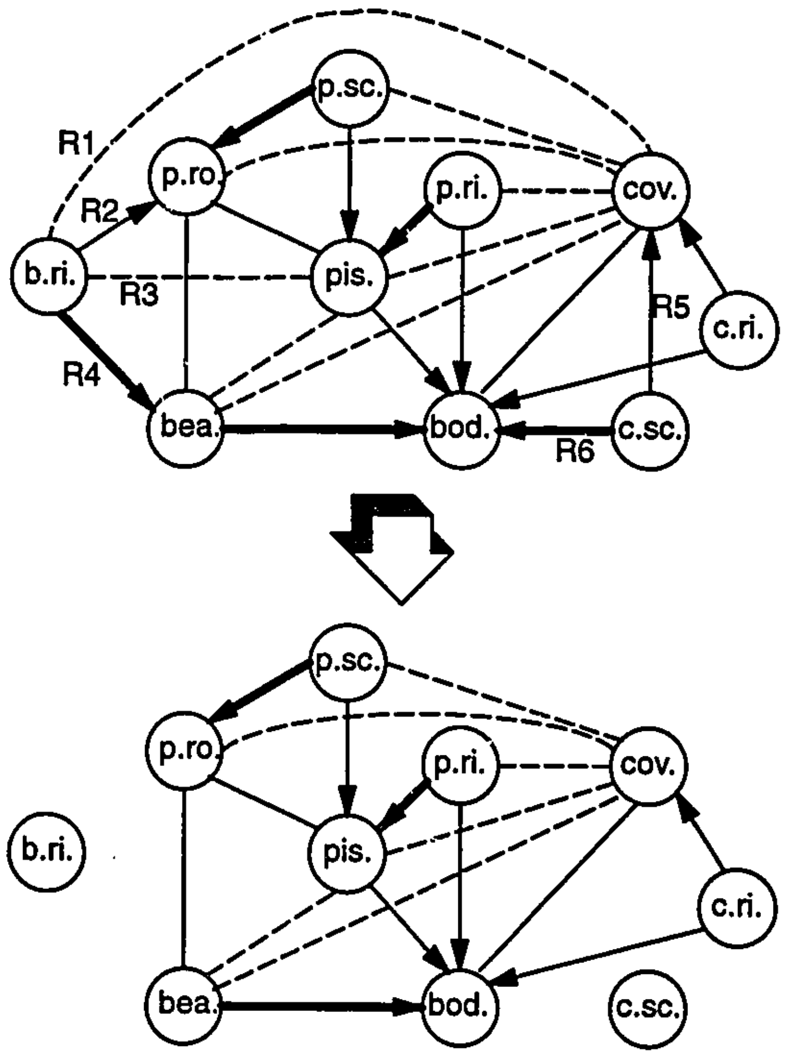


Fig. 4.9: A set of edges which is not a cutset: three induced subgraphs are resulting from their removal.

Following are some further definitions.

A **subassembly** is a set of vertices in an induced subgraph. Note that every component is also a subassembly.

An **assembly operation** is an action in the assembly process which results in the complete and definitive establishment of all the relations in a cutset. Physically, an assembly operation is an action which brings two and exactly two subassemblies together to form a larger subassembly.

A **disassembly operation** is an action in the disassembly process which results in the complete and definitive breaking of all the relations in a cutset. Physically, a disassembly operation is an action which splits a subassembly into two and exactly two smaller subassemblies[32].

An **assembly sequence** is a totally ordered set of assembly operations which results in the establishment of all relations in the graph model of the product.

A **disassembly sequence** is a totally ordered set of disassembly operations which results in the breaking of all relations in the graph model of the product.

An **assembly plan** is a partially ordered set of assembly operations, any linear extension of which is an assembly sequence.

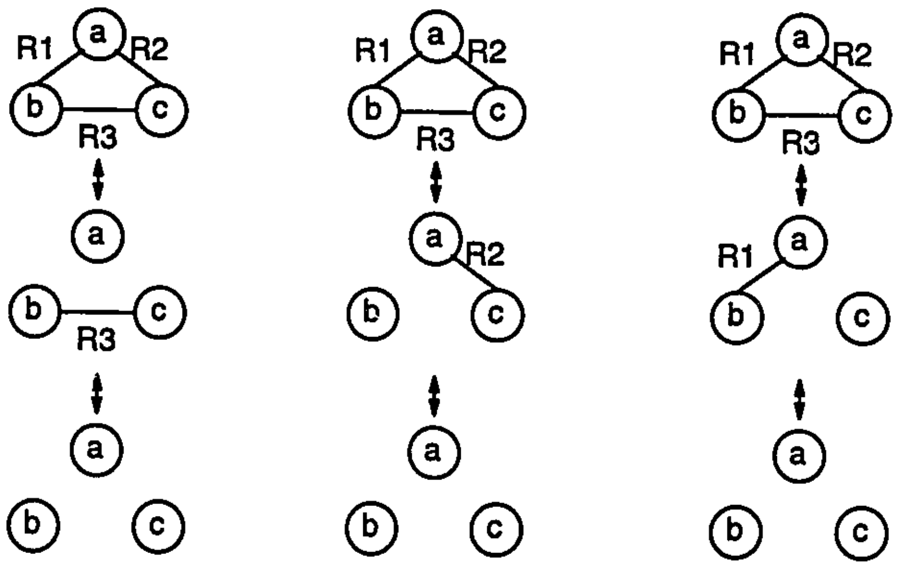
A **disassembly plan** is a partially ordered set of disassembly operations, any linear extension of which is a disassembly sequence.

4.2 FORMAL DESCRIPTION OF THE PROBLEM

Assume the graph model of a product of "n" components is connected. Using the definitions above, it can be readily verified that an assembly sequence for that

32. As there is always an assembly or disassembly operation required to establish or remove the edges in a cutset, the notion of a cutset and that of an assembly or disassembly operation are interchangeable in this work.

product contains exactly n–1 assembly operations. Therefore the generation of an assembly sequence is formally equivalent to finding a sequence of n–1 mutually exclusive cutsets in the product's graph model and its induced subgraphs. Figure 4.10 depicts this formal approach to assembly sequence generation, where the graph model of a hypothetical product with 3 components named "a", "b", and "c" is used.



Sequence 1 Sequence 2 Sequence 3

Fig. 4.10: Formal approach to assembly sequence generation. From top to bottom, sequence 1 involves cutsets {{R1, R2}, {R3}}, sequence 2 involves cutsets {{R1, R3}, {R2}} and sequence 3 involves cutsets {{R2, R3}, {R1}}.

A closer look at figure 4.10 leads to an important observation: all 3 assembly sequences have common top and bottom nodes. This suggests that the 3 assembly sequences could be more compactly represented if their common nodes were merged. Figure 4.11 shows a graph representation of this reduction. The nodes (or vertices) in this graph are the various assembly states of the product. The edges correspond to the assembly (or disassembly) operations that transform two incident states from one to another. The graph itself is appropriately called a *graph of assembly states* and was introduced earlier in chapter 2.

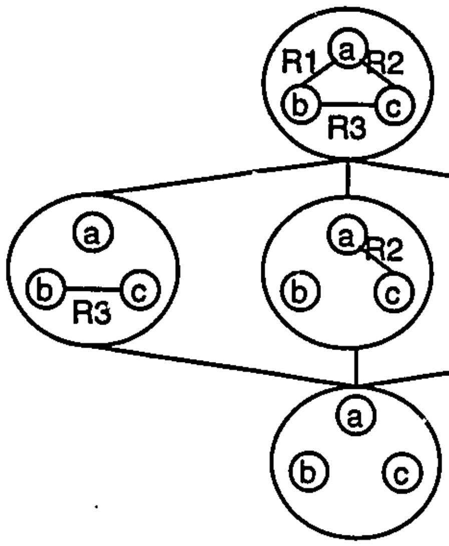


Fig. 4.11: Merging the common nodes of the 3 assembly sequences in figure 4.10 into a graph of assembly states.

Recall that any path from the root node to the leaf node (top down) corresponds to a disassembly sequence. Conversely, any path from the leaf node to the root node (bottom up) corresponds to an assembly sequence. Also note in figure 4.11 that since n = 3, i.e. 3 vertices in the product's graph model, and since the graph model is connected, then any path from the root node to the leaf node contains n–1 = 2 edges, i.e. 2 assembly or disassembly operations required in any assembly or disassembly sequence[33].

33. Section 4.5.1 will discuss the special case where the graph model of the product is not connected.

4.3 SEARCH DIRECTION

The problem of generating an assembly sequence for a product has just been mapped into the problem of finding a path into the graph of assembly states. At this point, the question is in which direction should the graph be searched: bottom up or top down? Because we are dealing with the generation of assembly (as opposed to disassembly) sequences, it would seem more logical to search the graph in the direction of assembly (as opposed to disassembly) operations, i.e. bottom up. There are several good reasons that justify the use of a disassembly approach instead.

First, an important Artificial Intelligence principle stipulates that any graph should always be searched in the direction of the lower branching factor [66]. Generally, there are much less ways of initially disassembling a product than there are ways of initially assembling its disassembled components. The disassembly approach, which yields the lower branching factor, should therefore be used.

Note that this statement is not true for a totally unconstrained product, i.e. a product for which the set of all cutsets in its graph model and any of its subgraphs is assumed to correspond to feasible assembly operations. To illustrate this, consider a totally unconstrained product of "n" components which graph model is K_n, the complete graph with "n" vertices. Mathematically, there are $2^{n-1}-1$ ways of initially disassembling this product, while there are $n(n-1)/2$ ways of initially assembling it. It can be readily verified that the former expression is always larger than the latter for $n > 3$. Consider now another totally unconstrained product, also of "n" components, but which graph model is a tree. There are $n-1$ ways of initially disassembling this product, as there are $n-1$ ways of initially assembling it. In any case the assembly approach yields the lower branching factor.

Practically, such totally unconstrained products do not exist so this argument does not hold. That there are less ways of initially disassembling a product than there are ways of initially assembling its disassembled components is also evident by looking at the product's graph of assembly states: the node corresponding to the completely assembled state always has less connections with other nodes than the one corresponding to the completely disassembled state. This can be verified using the directed graph of assembly states of the ball point pen showed earlier in figure 2.7.

A second good reason is given by Homem De Mello and Sanderson [27]: the disassembly approach requires no backtracking. Using the assembly approach, backtracking may be necessary because there are assembly operations that are physically feasible but which do not lead to a solution. To illustrate this, consider again the three blocks shown earlier on page 12. It is certainly possible to start the assembly by fitting the pin into the base. However, this renders the insertion of the plate unfeasible and therefore requires backtracking.

Clearly, the disassembly approach should be used, meaning that the graph of assembly states should be searched in the top down fashion. It is this strategy that has been implemented in GAPP. The graph of assembly states in figure 4.11 becomes the *directed* graph of assembly states in figure 4.12, where each edge has now been given a direction.

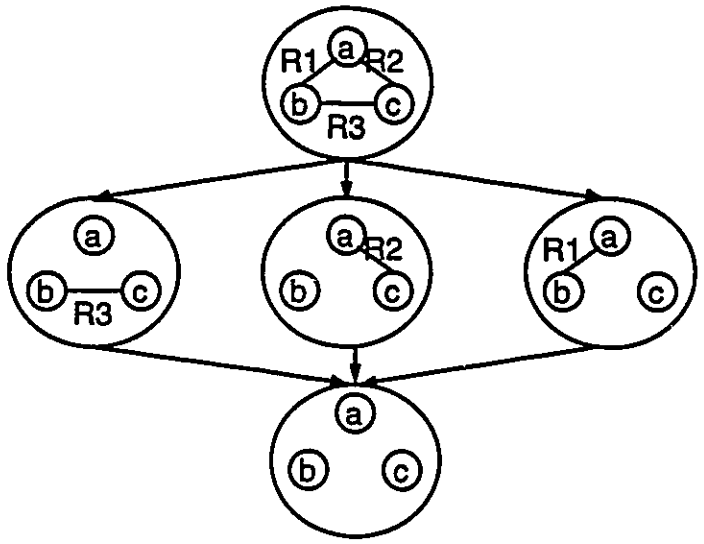


Fig. 4.12: Directed graph of assembly states.

Through the remaining of this thesis, the vertices in the directed graph of assembly states will be equivalently referred to as nodes or states. The root node (top) will be equivalently referred to as initial state, and the leaf node (bottom) as final or goal state. The directed edges in this graph will be equivalently referred to as disassembly operations, cutsets or links. For two incident nodes, the one with the incoming edge will be called the child and the one with the outgoing edge will be called the parent. Note that any node in the search graph is always the child of at least one node and the parent of at least another node. The only exceptions are the root node, which is a parent only, and the leaf node, which is a child only. The arrow heads of the directed edges will also be neglected for simplicity.

4.4 GENERATION OF THE CUTSETS

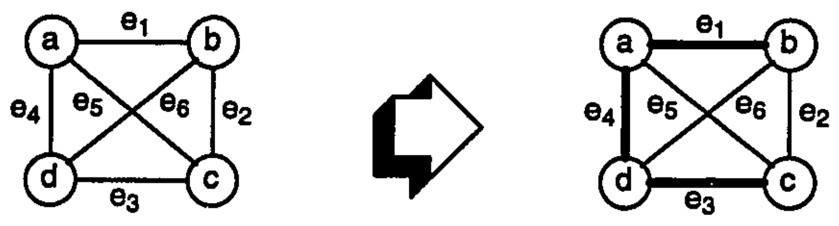
4.4.1 An algebraic approach

Let D = { C, R', Ψ_D} be the directed graph model of some product. It has been shown that $\mathcal{P}(R')$ is a vector space over the field of integers modulo two, that the set of all cutsets of D is a subspace of that space, and that the fundamental system of cutsets relative to a spanning tree or spanning forest of D is a basis of the cutset subspace [2] [48].

To generate all the cutsets of D, a spanning tree[34] of D is first selected using Kruskal's algorithm[35]. The fundamental system of cutsets relative to this spanning tree is then computed. The set of all cutsets in the fundamental system provides a basis for the cutset subspace. Linear combinations of the cutsets in the fundamental system give the set of all cutsets of the graph.

For a graph with "e" edges, a vector in the cutset subspace has "e" positions, one for each edge of the graph. Each position in the vector has a value of either 1 or 0, the two elements in the field of integers modulo two. The values 1 or 0 are associated to an edge which is part of a cutset or not, respectively. Using this vectorial representation, linear combinations of the cutsets are obtained by simply performing a position–wise addition of the corresponding vectors carried out in the field of integers modulo two. Figure 4.13 will be used to illustrate these concepts.

34. If D is not connected, a spanning forest is selected instead.
35. Kruskal's algorithm is presented in appendix B.



(a) (b)

	e_1	e_2	e_3	e_4	e_5	e_6
cutset 1	1	1	0	0	0	1
cutset 2	0	1	1	0	1	0
cutset 3	0	1	0	1	1	1

(c)

cutset 1 + cutset 2	1	0	1	0	1	1
cutset 1 + cutset 3	1	0	0	1	1	0
cutset 2 + cutset 3	0	0	1	1	0	1
cutset 1 + cutset 2 + cutset 3	1	1	1	1	0	0

(d)

Fig. 4.13: (a) a graph model; (b) a spanning tree in the graph model; (c) the three vectors in the fundamental system of cutsets relative to the spanning tree; (d) four vectors resulting from four linear combinations of the three vectors in (c).

Figure 4.13a presents the graph model of some hypothetical product with four components[36]. Without loss of generality, it is assumed that this graph is connected and that all the edges in it are of the contact type, i.e. every component has a contact relationship with every other component. Given this graph model, the first thing to do is to select one of its spanning trees using Kruskal's algorithm. In figure 4.13b, the selected spanning tree corresponds to bold edges in the graph.

Next, the cutset associated with each spanning edge in the graph is computed. For a particular spanning edge, this first consists of adding this edge to the set M of marked edges. One of the two incident vertices of this edge is selected and placed into the set S. A recursive loop examines each spanning edge of the graph in turn, except the ones in the marked set M. For any such spanning edge being examined, two situations may arise:

1– one of its incident components is included in S while the other is not, in which case this other incident component is added to S and this edge is added to M;

2– none of its incident components is included in S, in which case another spanning edge is selected for examination.

This algorithm is presented in figure 4.14. It stops when there are no more spanning edges that satisfy the conditions of the first case above.

36. Note that this graph is the complete graph with 4 vertices, K_4.

```
procedure make_subtree( component, spanning_edge )
    add component to S
    add spanning_edge to M
    for all spanning_edges in the graph model
        if spanning_edge is in M
            pick another spanning_edge
        if spanning_edge has one incident component in S
            add other incident component to S
            add spanning edge to M
            restart the loop
        if spanning_edge has no incident component in S
            pick another spanning_edge
```

Fig. 4.14: Procedure "make_subtree()".

It should be clear that, given a spanning edge "e" in the graph and given one of its incident components "a", this algorithm constructs the set S of all components connected to "a" via spanning edges only, in the case where "e" was removed from the initial graph. By running this algorithm twice, once for each incident component of "e", two component sets S_1 and S_2 get generated. The cutset associated with "e" is easily identified as the set of edges which have one incident component in S_1 and the other in S_2, or vice–versa.

Using spanning edge e_1 and incident component "a" in figure 4.13b, the sets are initially set up as $M = \{e_1\}$ and $S_1 = \{a\}$. A first iteration of the loop examines spanning edge e_3[37]. None of its two incident components are included in S_1, so a

37. Edges e_1 and e_2 are not examined prior to e_3, as the former has been marked and the latter is not a spanning edge.

next spanning edge is picked for examination, e_4 in this case. This edge is such that its incident component "a" is included in S_1 while its incident component "d" is not. Component "d" is therefore added to S_1, which now becomes $S_1 = \{a, d\}$, and edge e_4 is added to M, which now becomes $M = \{e_1, e_4\}$. A next iteration of the loop examines all unmarked spanning edges of the graph again and tries to add new components to the set S_1. It can be readily verified that the result of the algorithm for edge e_1 and component "a" are the sets:

$$S_1 = \{a, d, c\} \text{ and}$$
$$M = \{e_1, e_4, e_3\}.$$

Similarly, for edge e_1 and component "b", the resulting sets are:

$$S_2 = \{b\} \text{ and}$$
$$M = \{e_1, e_4, e_3\}.$$

The edges in the cutset associated with spanning edge e_1 can now be easily identified as being those which have one incident component in S_1 and the other in S_2, respectively. The cutset associated with e_1 is then:

$$\{e_1, e_2, e_6\}.$$

Repeating this process for every spanning edge of the graph leads to the identification of the fundamental system of cutsets corresponding to the selected spanning tree. For the spanning tree in figure 4.13b, the fundamental system of cutsets is as follows:

$$\{e_1, e_2, e_6\},$$
$$\{e_2, e_3, e_5\} \text{ and}$$
$$\{e_2, e_4, e_5, e_6\}.$$

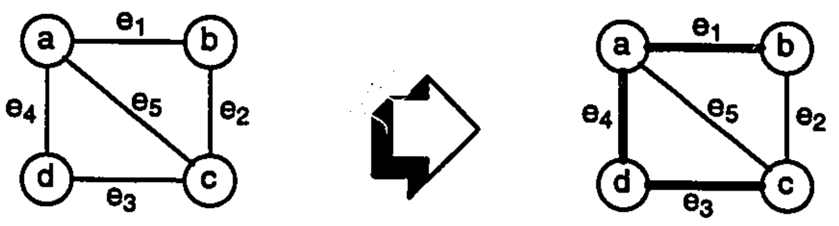
This fundamental system of cutsets is given the vectorial representation in figure 4.13c. The set of all cutsets of the graph is obtained from the linear combinations of these vectors. Three such combinations are trivially given by the three vectors themselves. Any other combination involving more than one vector can be computed nicely by the simple position–wise addition of the vectors in it, carried out in the field of integers modulo two. Figure 4.13d shows the results of all possible linear combinations involving more than one of the vectors in figure 4.13c. Together, figures 4.13c and 4.13d show the vectorial representation of all seven cutsets of the graph in figure 4.13a.

4.4.2 Non–complete graphs

A spanning tree of a connected graph with "n" vertices contains "n–1" edges. This implies that there are also "n–1" cutsets in the fundamental system of cutsets associated with this spanning tree, and that there are $2^{n-1}-1$ possible linear combinations of the cutsets in this fundamental system[38].

A complete graph has $2^{n-1}-1$ cutsets. Therefore every possible linear combination of the cutsets in the fundamental system of cutsets relative to any of its spanning trees gives one of the cutsets in this graph. The complete graph K_4 in figure 4.13a is an example of such a case: there are $2^{4-1}-1 = 7$ cutsets in this graph, each corresponding to a linear combination of the three cutsets in the fundamental system in figure 4.13c.

38. For "n–1" cutsets, there are 2^{n-1} ways of combining them. The expression $2^{n-1}-1$ is obtained by omitting the non–combination, i.e. the combination resulting from a selection of none of the cutsets.



(a) (b)

	e_1	e_2	e_3	e_4	e_5
cutset 1	1	1	0	0	0
cutset 2	0	1	1	0	1
cutset 3	0	1	0	1	1

(c)

cutset 1 + cutset 2	1	0	1	0	1	
cutset 1 + cutset 3	1	0	0	1	1	
cutset 2 + cutset 3	0	0	1	1	0	
cutset 1 + cutset 2 + cutset 3	1	1	1	1	0	⇨ not a cutset

(d)

Fig. 4.15: (a) a graph model; (b) a spanning tree in the graph model; (c) the three vectors in the fundamental system of cutsets relative to the spanning tree; (d) four vectors resulting from four linear combinations of the three vectors in (c): the last combination is not a cutset of the graph.

The graph model of most products is not complete. Therefore such a graph has less than $2^{n-1}-1$ cutsets in it. This implies that some of the linear combinations of the cutsets in some fundamental system of cutsets relative to one of its spanning trees does not correspond to a cutset in this graph. In particular, the removal of some sets of edges resulting from some linear combinations of cutsets splits the graph into more than two induced subgraphs, thus violating the cutset definition.

Figure 4.15d illustrates this. One of the vectors obtained from a linear combination of three others in figure 4.15c is not a cutset, as it splits the graph into the three induced subgraphs with vertex sets: {a, c}, {b} and {d}.

A test is performed to determine if a linear combination of cutsets results in a set of edges which is also a cutset. The algorithm which implements this test is presented in figure 4.16. Note that this algorithm is very similar to that of figure 4.14. Three differences are that the set M initially contains all the edges in the set of edges resulting from a linear combination, that non–spanning edges are also included for examination, and that a further check in the loop is performed to see if the examined edge has both incident components in S, in which case this edge is added to M.

```
procedure make_subgraph( component, list_of_edges )
    add component to S
    add list_of_edges to M
    for all edges in the graph model
        if edge is in M
            pick another edge
        if edge has one incident component in S
            add other incident component to S
            add edge to M
            restart the loop
        if edge has no incident component in S
            pick another edge
        if edge has both incident components in S
            add edge to M
            pick another edge
```

Fig. 4.16: Procedure "make_subgraph()".

It should be clear that, given a set E of edges resulting from the linear combination of some cutsets of a graph and given an incident component "a" of an arbitrary edge in E, this algorithm constructs the set S of all components connected to "a", in the case where E was removed from the initial graph. By running this algorithm twice, once for each incident component of an arbitrary edge in E, two component sets S_1 and S_2 get generated. The set E of edges is a cutset if and only if each edge in this set has one incident component in S_1 and the other in S_2, or vice–versa.

4.4.3 Updating the cutsets

One way of generating the directed graph of assembly states is as follows: generate all the cutsets in the initial state; for each cutset, create a new state of the directed graph in which all edges of this cutset have been removed; generate the cutsets of each new state; for each cutset of each new state, create a new state of the directed graph in which all edges of this cutset have been removed, and so on.

Although this approach is certainly feasible, it requires the computation of a new set of cutsets for each new state added to the directed graph. This leads to a lot of computations and the time taken for generating the directed graph of assembly states of even fairly simple products might not be negligible.

In GAPP, only the set of all cutsets in the initial state of the directed graph of assembly states is ever generated. Any other cutset in any other state is obtained by simply updating this set of cutsets in the initial state. Figure 4.17 depicts the essence of the approach. After the six cutsets of the root node have been computed, six new states[39] of the directed graph are generated. The six cutsets of the root node are inherited by each new child node. A deletion of some edges of the inherited cutsets, not part of the new child node, must first be performed. For example, child1 was obtained from cutset $\{e_1, e_3, e_5\}$. The edges in this cutset must not be part of any of child1's cutsets. They are therefore deleted from the inherited set, yielding the new cutsets:

39. Only 4 of these six new states are shown in figure 4.17, due to a lack of space. This should not influence the demonstration that follows.

1– {e_2},

2– {e_2},

3– {e_4},

4– {e_4},

5– {}, and

6– {e_2, e_4}.

Out of this new list, the first and second, as well as the third and fourth, are combined as they both represent the same cutset. The fifth, which was removed from the root node to generate child1, is eliminated as it became empty. Therefore, the list becomes:

1– {e_2},

2– {e_4},

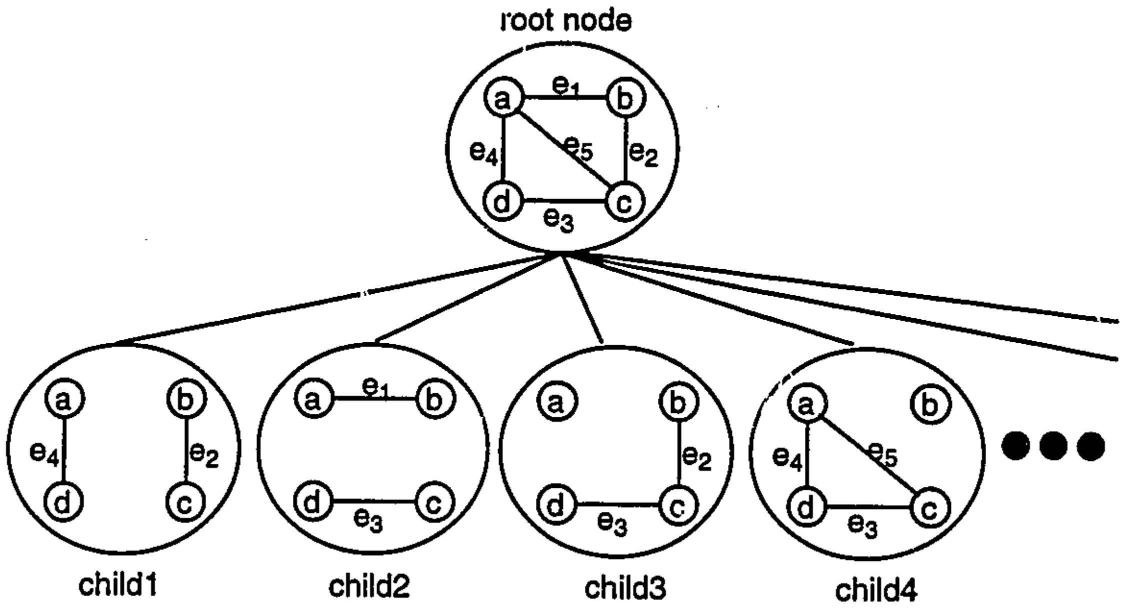
3– {e_2, e_4}.

Algorithm make_subgraph() showed in figure 4.16 is then used to check if the remaining sets of edges are indeed cutsets. Running the algorithm for the above sets of edges leads to the elimination of the last one, as this set does not satisfy the definition of a cutset. That is, by removing {e_2, e_4} from child1, a new state with more than one induced subgraph than child1 would be generated. The cutsets of child1 are then:

1– {e_2} and

2– {e_4}.

Repeating this process for every new child node leads to the determination of their cutset from a simple analysis of the ones inherited from their parent.



cutsets of root node

⇩

$\{e_1, e_2\}, \{e_2, e_3, e_5\}, \{e_3, e_4\}, \{e_1, e_4, e_5\}, \{e_1, e_3, e_5\}, \{e_2, e_4, e_5\}$

⇩ inherited by child1

$\{e_1, e_2\}, \{e_2, e_3, e_5\}, \{e_3, e_4\}, \{e_1, e_4, e_5\}, \{e_1, e_3, e_5\}, \{e_2, e_4, e_5\}$

⇩ remove edges not in child1

$\{e_2\}, \{e_2\}, \{e_4\}, \{e_4\}, \{\}, \{e_2, e_4\}$

⇩ combine similar sets, remove empty sets

$\{e_2\}, \{e_4\}, \{e_2, e_4\}$

⇩ eliminate non–cutsets

$\{e_2\}, \{e_4\}$ cutsets of child1

Fig. 4.17: Determination of the cutsets of a new child node from an analysis of the ones inherited from its parent.

4.5 BENEFITS OF THE ADOPTED APPROACH

4.5.1 Arbitrary initial state specification

So far, it has been assumed that the product's graph model, and therefore the initial state of the directed graph of assembly states, was connected. As mentioned in chapter 3, there might be applications where the final configuration of a product resulting from some assembly process might consist of different unconnected subassemblies[40]. In GAPP, such situations translate into the generation of an unconnected product graph model.

The approach developed for generating and updating cutsets, essential for the expansion of the directed graph of assembly states, is directly applicable to unconnected graph models. As far as GAPP is concerned, the only difference between a connected or unconnected graph is that a spanning tree must be selected in the former, a spanning forest in the latter. No further artifacts or special rules are required to handle both cases, as one is just the generalization of the other. Apart from giving GAPP the flexibility to handle assembly problems represented by both types of graphs, the developed approach also provides for their unified processing.

4.5.2 Multiple products

The process of generating and updating the cutsets in a state is totally independent of the particular physical subassemblies represented in that state. This provides GAPP with a most powerful feature: the subassemblies in a state need not be part of the same product.

40. An example of such an application is a product which is only partly assembled in the factory and which remaining assembly is left to the consumer.

Extending the discussion in section 4.5.1, this means the graph model of two or more different products can be combined into a single but unconnected graph model in the initial state, and that GAPP can generate a single assembly sequence where two or more consecutive operations apply to two or more different products. A practical application of this is the determination of an assembly plan for a robot assembling two products of the same family (or even two totally different products) simultaneously.

4.5.3 Arbitrary goal state specification

Another assumption used so far is that the product is completely disassembled as a result of the disassembly process, i.e. the goal state consists of all unconnected components. Once again, there are many applications which violate this assumption, for example repair planning and error recovery planning in robotic assembly.

For the former application, the part or subassembly to be repaired might not require the complete disassembly of the product. Thus the goal state need not be the completely disassembled state. For the latter application, it is quite possible that the goal state should consist initially of the completely disassembled product. However, should an error occur in the execution of the plan, this error could result in an unpredicted state. This new state, which might not be the completely disassembled state, becomes the new goal state of the planner[41].

Dealing with such problems in GAPP simply requires the appropriate setting of the goal flag of each relation in the graph model, which was briefly discussed in chapter 3. Given some graph model of the product, any relation of that model which has its goal flag set to "yes" will never be broken in the disassembly operations

41. Chapter 8 will show examples of using GAPP for repair planning and error recovery planning.

generated by GAPP. In other words, these relations are "sealed off" from the disassembly process. This implies that the goal state will present some induced subgraphs containing these relations, i.e. the goal state will not correspond to the completely disassembled product.

In repair planning, the disassembled state which permits the access to the faulty parts or subassemblies is identified and the goal flag of the remaining relations in that state is turned on. A disassembly plan which transforms an initial state of the product, usually the completely assembled state, into this disassembled state is generated by GAPP.

Errors occur often during assembly for many reasons. Such errors cannot be predicted in advance and hence the state of assembly, from which replanning to complete the assembly should start, would not be known apriori. In this case of error recovery planning, the connections among components in the unpredicted state resulting from an error are identified by the programmer and their goal flag turned on. A disassembly plan which transforms an initial state of the product into this unpredicted state is generated by GAPP. The reverse of this disassembly plan gives the assembly plan required to recover from the unpredicted assembly failure state and complete the assembly of the product.

In terms of generating the cutsets of the graph model, this type of functionality is provided by eliminating all the cutsets which contain at least one relation with its goal flag turned on. The remaining cutsets in the initial state and their updated version in subsequent children states ensure that no goal state will ever be reached in which a "sealed" relation does not appear.

It is interesting to note that the same functionality can be achieved by merging the components of each induced subgraph of the goal state into a single component, instead of turning on the goal flag of the relations in these subgraphs. This implies the generation of a new graph model with different vertex and edge sets for every different goal configuration being dealt with. Turning the goal flag of the relations on and off is a much better approach, as the same graph model can be used for different goal configurations.

4.5.4 Computation speed

Once the fundamental system of cutsets relative to some spanning tree of a graph has been determined, all possible linear combinations of these cutsets must be computed in order to generate all other cutsets of the graph. To perform this computation, the cutsets in the fundamental system are labelled and associated with a position in a binary vector, where each position holds a value of either 0 or 1. For "n" cutsets in the fundamental system, the binary vector is incremented from 1 to 2^{n-1}. Each increment corresponds to one possible linear combination, given by including in this combination all the cutsets whose corresponding position in the vector is not 0.

Figure 4.18 illustrates this concept. There are six cutsets in the graph. Therefore, the binary vector has six positions. Starting with a value of 0, a first increment of the vector corresponds to the combination of cutset "c_6" and no other cutset. A second increment corresponds to the combination of cutset "c_5" only. A third increment corresponds to the combination of cutsets "c_5" and "c_6", and so on. In turn, each cutset in the fundamental system of cutsets is associated with a binary vector, as shown earlier in figure 4.13c. The vectorial representation of a new cutset resulting from the linear combination of two or more cutsets is obtained from the position–wise addition of their corresponding vectors, carried out in the field of integers modulo two. This addition directly corresponds to the bitwise XOR operator of the C^{++} language, in which GAPP is implemented.

As will be seen in the next three chapters, the expansion of the directed search graph of assembly states involves much computations. For example, the feasibility of the disassembly operations associated with each edge in the graph must be computed. An evaluation of the goodness of these operations with respect to some criteria must also be computed. Housekeeping computations, such as identifying new generated nodes that already exist in the graph and keeping track of the best path to a node, must also be constantly performed.

The developed approach for generating the cutsets of a graph is such that the cutsets computations need to be performed only once, prior to the expansion of the directed search graph of assembly states. The cutsets of any state of the graph are obtained from a simple update of the ones inherited from their parent states. This relieves the main search algorithm from performing these computations at each new state of the search graph and enables it to focus more on the feasibility, evaluation or housekeeping computations. Furthermore, as far as the apriori computation of the cutsets is concerned, the approach of incrementing a binary vector to find all possible combinations of cutsets and using bitwise operators to determine the result of a particular combination both lead to very fast computations.

$c_1 = \{ R1, R2 \}$

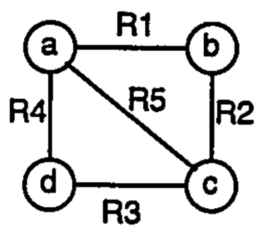
$c_2 = \{ R2, R3, R5 \}$

$c_3 = \{ R3, R4 \}$

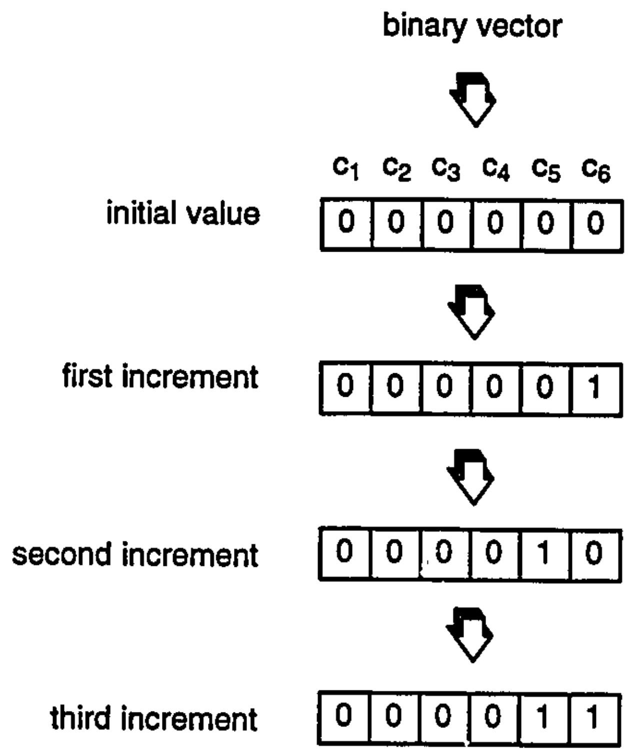
$c_4 = \{ R1, R4, R5 \}$

$c_5 = \{ R1, R3, R5 \}$

$c_6 = \{ R2, R4, R5 \}$



(a)



(b)

Fig. 4.18: (a) a graph model with six cutsets; (b) incrementing the binary vector to find all possible combinations of cutsets.

CHAPTER 5

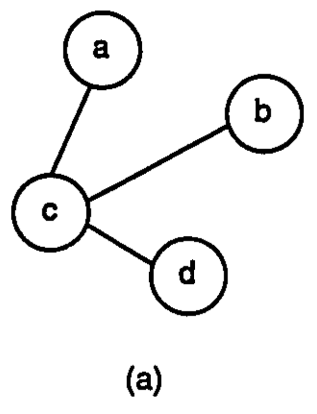
OPERATIONS CONSTRAINTS

This chapter describes the various constraints on the assembly operation associated with a cutset, which have the effect of eliminating some of the edges, vertices and paths in the directed graph of assembly states. The first section looks at the theoretical combinatorial complexity underlying the generation of this graph, in terms of the number of vertices, edges and paths that it may contain. The second section introduces the notion of constraints on assembly operations. The third and fourth sections describe the geometric feasibility and accessibility constraints, respectively.

5.1 COMBINATORIAL COMPLEXITY

This section looks at the combinatorics involved in the automatic generation of the directed graph of assembly states. The analysis is performed for three important parameters of this graph: the number of vertices in the graph, which corresponds to the number of states to be generated; the number of edges in the graph, which corresponds to the number of disassembly operations whose feasibility needs to be analyzed; and the number of paths from the initial state to the final state in the graph, which corresponds to the number of assembly sequences.

Two cases are analyzed for each parameter: a connected product whose graph model is a tree and a connected product whose graph model is complete (figure 5.1).



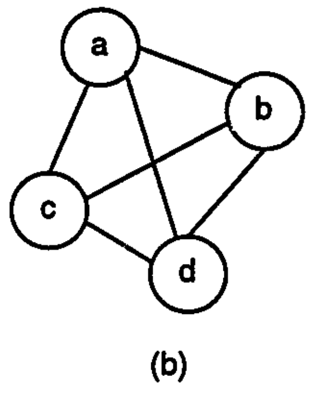


Fig. 5.1: (a) a four components product with a tree graph model; (b) a four components product with a complete graph model.

The product is assumed to be totally unconstrained. This means that the disassembly operation associated with each edge in the directed graph of assembly states is always assumed to be physically feasible.

5.1.1 Tree graph model

For a totally unconstrained product of "n" components with a tree graph model, the number "x" of nodes in the directed graph of assembly states is given by:

$$x_n = 2^{n-1} \tag{5.1}$$

The number "y"of edges in the directed graph of assembly states is given by:

$$y_2 = 1 \tag{5.2}$$

$$y_n = 2y_{n-1} + 2^{n-2} \tag{5.3}$$

The number "z" of paths from the root node to the goal node is given by:

$$z_n = (n-1)! \qquad (5.4)$$

Table 5.1 shows the values of these parameters for products with 10 to 15 components.

Table 5.1
Statistics of the directed graph of assembly states for products with a tree graph model.

number of components	number of nodes (eq. 5.1)	number of edges (eqs 5.2 & 5.3)	number of paths (eq. 5.4)
10	512	2,304	362,800
11	1,024	5,120	3,628,800
12	2,048	11,264	39,916,800
13	4,096	24,576	479,001,600
14	8,192	53,248	6,227,020,800
15	16,384	114,688	87,178,291,200

Figure 5.2 shows the unconstrained directed graph of assembly states for the product in figure 5.1a.

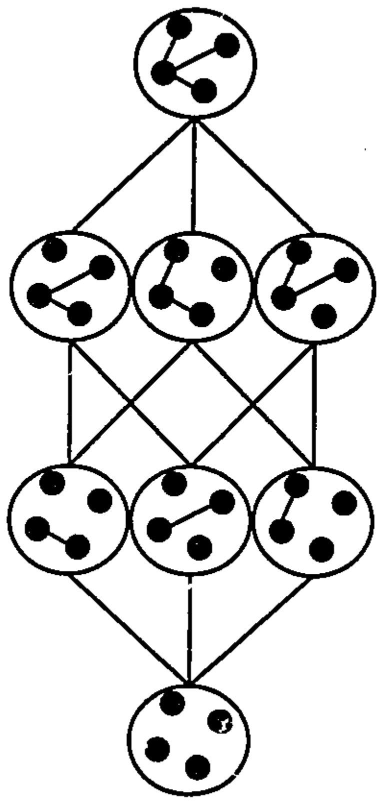


Fig. 5.2: Unconstrained directed graph of assembly states for the product in figure 5.1a. There are 8 vertices, 12 edges and 6 paths.

5.1.2 Complete graph model

For a totally unconstrained product of "n" components with a complete graph model, the number "x" of nodes in the directed graph of assembly states is given by:

$$x_0 = x_1 = 1 \tag{5.5}$$

$$x_n = \sum_{i=0}^{n-1} \binom{n-1}{2} x_{n-i-1} \tag{5.6}$$

The number "y"of edges in the directed graph of assembly states is given by:

$$w_2 = 2 \tag{5.7}$$

$$w_n = n! + \sum_{i=2}^{n-1} w_i \tag{5.8}$$

$$y_n = w_n - n + 1 \tag{5.9}$$

The number "z" of paths from the root node to the goal node is given by:

$$z_n = \prod_{i=2}^{n} \binom{i}{2} \tag{5.10}$$

Table 5.2 shows the values of these parameters for products with 10 to 15 components.

Table 5.2
Statistics of the directed graph of assembly states for products with a complete graph model.

number of components	number of nodes (eqs 5.5 & 5.6)	number of edges (eqs 5.7, 5.8 & 5.9)	number of paths (eq. 5.10)
10	115,975	4,101,559	2.571 E9
11	678,570	44,491,126	1.414 E11
12	4,213,597	528,067,061	9.336 E12
13	27,644,437	6.804 E9	7.282 E14
14	190,899,322	9.455 E10	6.626 E16
15	1.382 E9	1.409 E12	6.958 E18

Figure 5.3 shows the unconstrained directed graph of assembly states for the product in figure 5.1b.

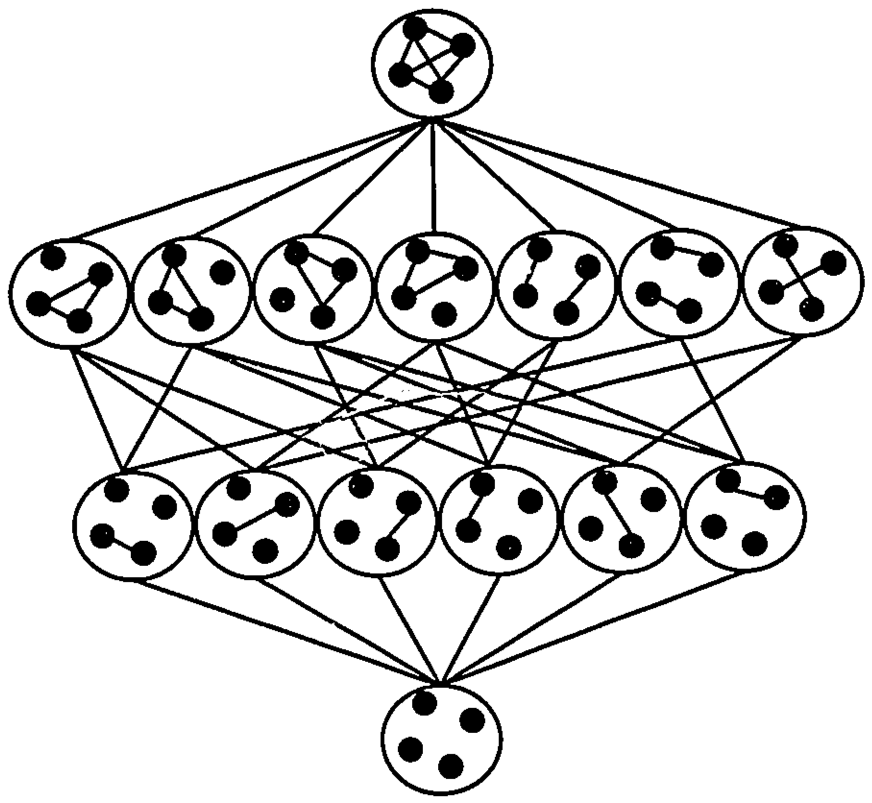


Fig. 5.3: Unconstrained directed graph of assembly states for the product in figure 5.1b. There are 15 vertices, 31 edges and 18 paths.

Tables 5.1 and 5.2 clearly show the combinatorial explosion of the number of nodes in the directed graph of assembly states as the number of components in the product and the number of connections among them gets larger. Practically, the figures in table 5.2 are not realistic since no product is ever represented by a complete graph model as shown in this example. Even the figures in table 5.1 are way above the average statistical values that are generally obtained for the directed graph of assembly states of real industrial products with a corresponding number of components. The reason for this is simply that these figures were obtained assuming that the product was totally unconstrained. Although not realistic, the figures were still necessary at the initial development stage of GAPP, in order to compare its results with those predicted by the theory. It was verified that the results obtained from GAPP using hypothetical unconstrained products are always in agreement with those predicted by theory.

5.2 CONSTRAINTS ON ASSEMBLY OPERATIONS

Consider the 3–blocks product along with its graph model in figure 5.4. Note that this graph model is isomorphic to that of the hypothetical product shown earlier on page 86. From a purely mathematical point of view, the directed graph of assembly states for the three blocks should be identical to that of the hypothetical product (page 90). There should be a disassembly sequence of the three blocks corresponding to every directed path in this graph. Practically however, this may not be true because some of the disassembly operations associated with the edges of the directed graph are physically unfeasible.

In particular, it is initially impossible to remove block "b" from the total assembly. This results in the elimination of an edge in the search graph, which in turn results in the elimination of an entire disassembly sequence. This is illustrated in figure 5.5.

In general, the consideration of the various feasibility constraints of the disassembly operations associated with each edge in the directed graph of assembly states considerably reduces the size of this graph.

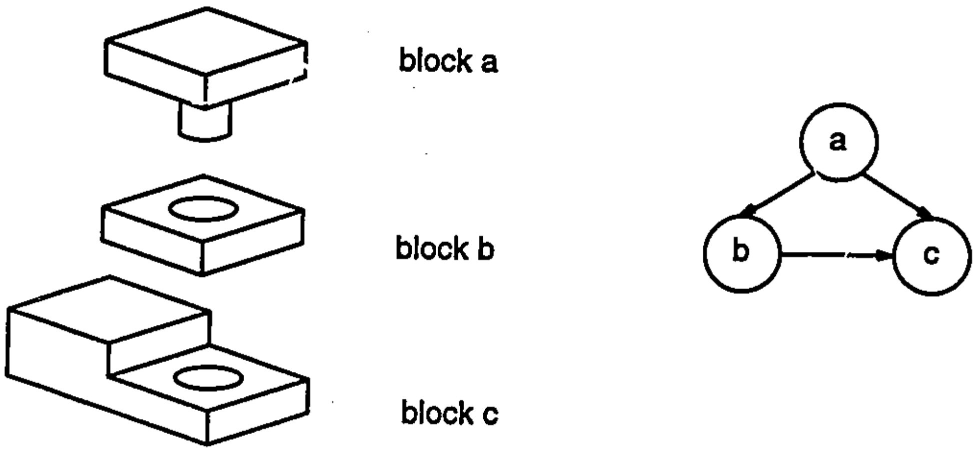


Fig. 5.4: Three blocks to be assembled (left) along with their graph model (right).

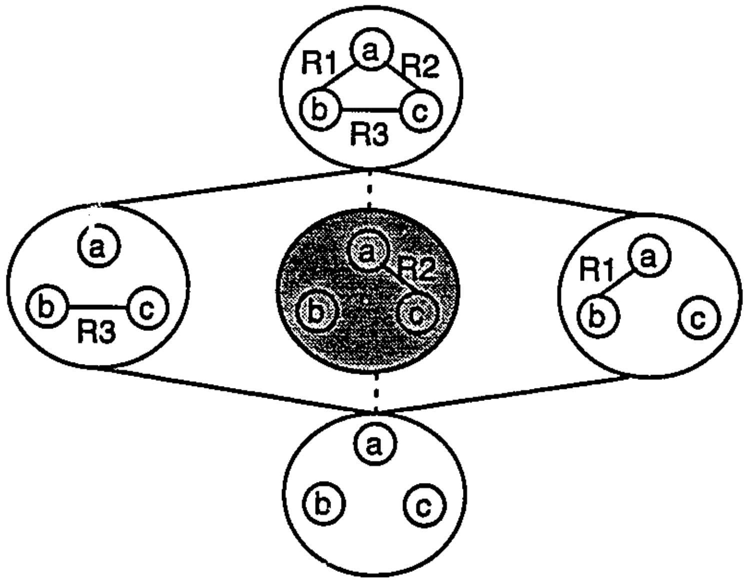


Fig. 5.5: Illustration of the effect of adding feasibility constraints to the directed graph of assembly states of the 3 blocks in figure 5.4: the shaded node and dotted edges are eliminated, leaving only 2 disassembly sequences.

In this work, two types of constraints on assembly operations are used:

1– geometric feasibility constraints, and

2– accessibility constraints.

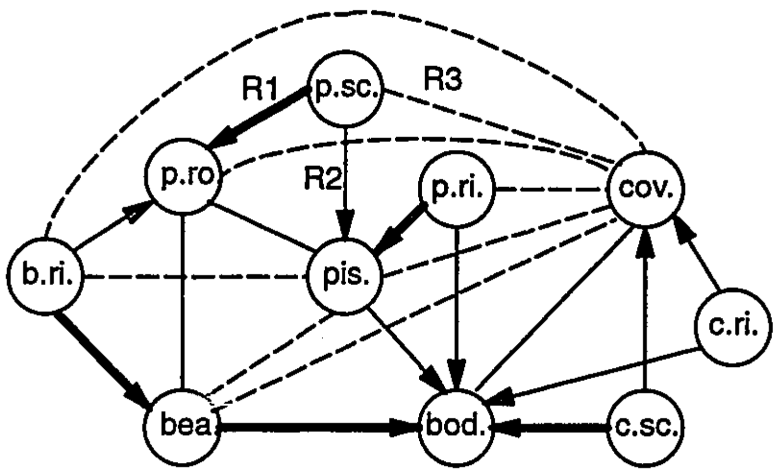
Each one is discussed individually in the following sections.

5.3 GEOMETRIC FEASIBILITY CONSTRAINT

The contact and attachment relations defined in the product's graph model restrict the degrees of freedom of the components in the assembly. For example, a peg that loosely fits into a hole has only two degrees of freedom: one translation in the direction of the hole axis and one rotation about the hole axis.

One way to test the geometric feasibility of an assembly operation is to determine if a component or subassembly of the product possesses a local degree of freedom in some direction, by analyzing the contacts and attachments in the graph model. However, a subassembly that effectively possesses a degree of freedom in some direction might still be impossible to remove, because it might collide in that direction with some other components in the assembly with which it is not in physical contact. This is the reason for defining blocking relations in the graph model of the product. These relations help determine whether a subassembly that possesses a local degree of freedom in some direction is also removable in that direction without colliding with the remaining components of the assembly.

Figure 5.6 will be used to illustrate this concept. It is desired to determine if an operation that would split the air cylinder into the two subassemblies {piston_screw} and {bearing_o_ring, bearing, body, piston_rod, piston, piston_o_ring, cover, cover_o_ring, cover_screws} is geometrically feasible.



R1 — piston_screw along (z–)
R1 — piston_rod along (z+)

R2 — piston_screw along (z–)
R2 — piston along (z+)

R3 — piston_screw along (x+, x–, y+, y–, z+)
R3 — cover along (x+, x–, y+, y–, z–)

Fig. 5.6: A cutset in the air cylinder's graph model, along with the disassembly directions of the incident components of the relations in this cutset.

Associated with this operation is the cutset {R1 R2 R3}. The disassembly directions for the incident components of the relations in this cutset, which are also shown in figure 5.6, are used by GAPP in order to compute the geometric feasibility of this operation.

Attachment R1 and contact R2 reveal that the piston_screw could be removed along z–. However, blocking relation R3 reveals that the piston_screw can be moved in any direction *except* z– in order to avoid a collision with the cover. This operation is not geometrically feasible since there does not exist any disassembly direction from which the piston_screw can be removed. It is important to note the role of the blocking relation, which helped determine that although the piston_screw could be moved locally along z–, it could not be completely removed from the product in this direction.

Generally, not every component in each subassembly resulting from a cutset is incident to the relations in this cutset. In the above example, the component in the {piston_screw} subassembly is necessarily incident to all relations in the cutset since it is the sole component in this subassembly. However, only components piston_rod, piston and cover of other subassembly {bearing_o_ring, bearing, body, piston_rod, piston, piston_o_ring, cover, cover_o_ring, cover_screws} are incident to the relations in the cutset. The components of a subassembly which are incident to the relation(s) in the cutset from which this subassembly was formed will be called the *matched* components.

From this, GAPP formally concludes that an operation is geometrically feasible if the intersection of all the disassembly directions of all matched components in either one of the two subassemblies resulting from a cutset is not empty. For the example above, this means that either one of the following two expressions must be satisfied:

$$dir(R1, piston\_screw) \bigcap dir(R2, piston\_screw) \bigcap dir(R3, piston\_screw) \neq \{\} \tag{5.11}$$

$$dir(R1, piston\_rod) \bigcap dir(R2, piston) \bigcap dir(R3, cover) \neq \{\} \tag{5.12}$$

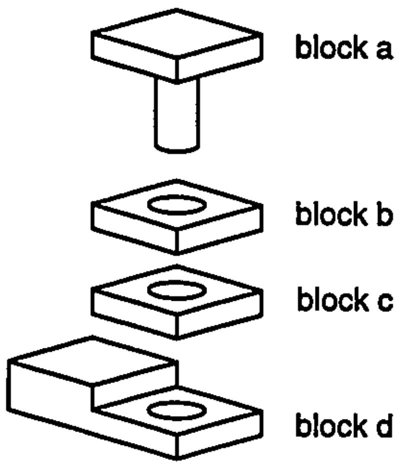
where *dir()* stands for the *directions()* function described earlier in chapter 3.

Evaluating the expression given by (5.11) yields:

$$\{z-\} \bigcap \{z-\} \bigcap \{x+\ x-\ y+\ y-\ z+\} = \{\}. \tag{5.13}$$

Therefore the disassembly operation corresponding to the cutset {R1, R2, R3} in figure 5.6 is geometrically unfeasible. The consequence of this is the elimination of all the state transitions of the air cylinder's unconstrained directed graph of assembly states which correspond to this cutset. Considering the geometric feasibility of all other possible cutsets in all other possible state transitions in this graph leads to a reduction in the number of state transitions (edges) in it, which in turn results in a significant reduction in the number of assembly sequences (paths). A smaller reduction in the number of nodes (vertices) is also usually obtained.

In the unconstrained case, the air cylinder's directed graph of assembly states has 2,678 nodes, for a total of 6,519,744 assembly sequences (paths). By including the computation of the geometric feasibility constraint during the generation process, the graph is reduced to 2,322 nodes and 646,380 assembly sequences. This kind of reduction has been observed to vary from one product to another, with strongly geometrically constrained products exhibiting the most significant reductions. Figures 5.7, 5.8 and 5.9 illustrate this reduction for a simpler product.



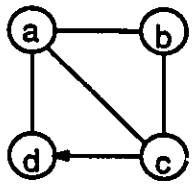


Fig. 5.7: Four blocks to be assembled (left) along with their graph model (right).

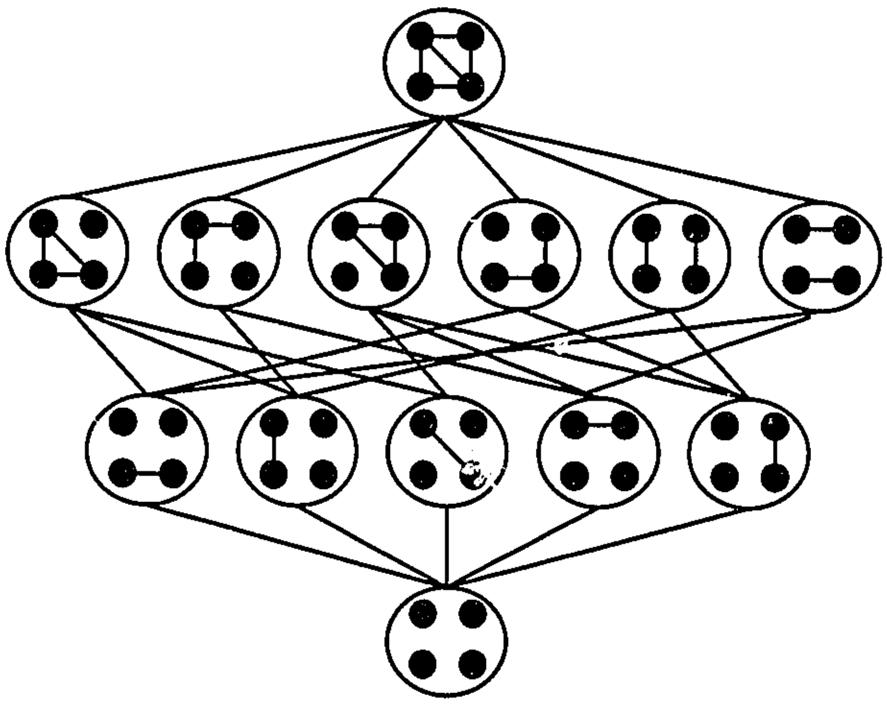


Fig. 5.8: Unconstrained directed graph of assembly states for the product in figure 5.7. There are 13 vertices, 25 edges and 14 paths.

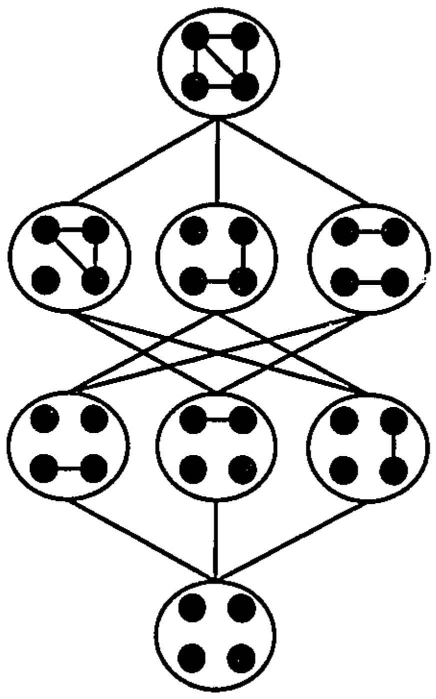
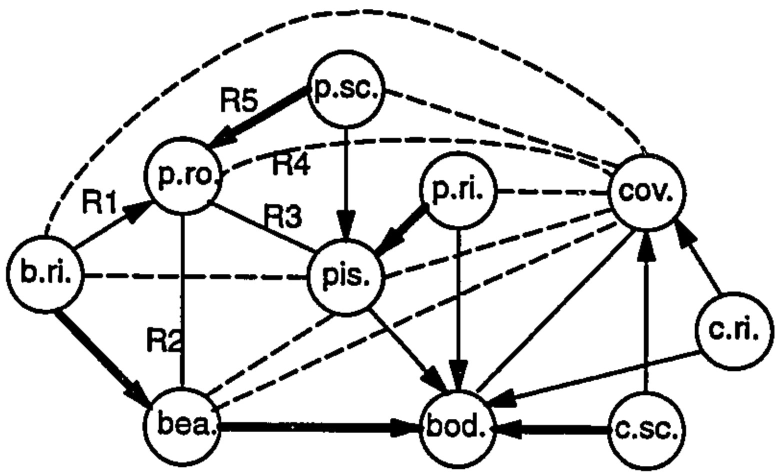


Fig. 5.9: Directed graph of assembly states for the product in figure 5.7 after considering the geometric feasibility constraint. There are now 8 vertices, 12 edges and 6 paths.

5.4 ACCESSIBILITY CONSTRAINTS

5.4.1 Illustrative examples

Geometric feasibility constraints result in a major reduction in the number of edges, and therefore paths, in the directed graph of assembly states and eliminates most of the unfeasible disassembly operations. Nevertheless, geometrically feasible disassembly operations might still be impossible to execute for accessibility reasons. Consider for example figure 5.10. It is desired to determine the feasibility of an operation that would split the air cylinder into the two subassemblies {piston_rod} and {bearing, bearing_o_ring, piston_screw, piston_o_ring, piston, body, cover, cover_screws, cover_o_ring}.



R1 — piston_rod along (z+, z–)
bearing_o_ring along (z+, z–)

R2 — piston_rod along (z+, z–)
bearing along (z+, z–)

R3 — piston_rod along (x+, x–, y+, y–, z+)
piston along (x+, x–, y+, y–, z–)

R4 — piston_rod along (x+, x–, y+, y–, z+)
cover along (x+, x–, y+, y–, z–)

R5 — piston_rod along (z+)
piston_screw along (z–)

Fig. 5.10: A cutset in the air cylinder graph model, along with the disassembly directions of the incident components of the relations in this cutset.

The relations in the cutset show that z+ is a possible disassembly direction for removing the piston_rod:

$$\begin{aligned} &dir(R1, piston\_rod) \bigcap dir(R2, piston\_rod) \bigcap dir(R3, piston\_rod) \\ &\quad \bigcap dir(R4, piston\_rod) \bigcap dir(R5, piston\_rod) = \{z+\}. \end{aligned} \tag{5.14}$$

Considering geometry alone, the corresponding disassembly operation is feasible. Practically however, it is clearly impossible to perform this operation because one of the matched components in the cutset[42], i.e. the piston_screw, is inaccessible. That is, no tool can access this component to hold against the screwing torque applied to the piston_rod.

Another example of such a situation for the air cylinder is the removal of the {bearing, bearing_o_ring} subassembly from the complete assembly (figure 5.11): although this subassembly is geometrically feasible to remove along the z+ direction, the lack of access for a tool to hold this subassembly in trying to remove it makes the corresponding disassembly operation impossible to execute.

Figure 5.12 shows a generic example of a component which is geometrically feasible to remove but due to access restriction the corresponding disassembly operation could be considered unfeasible.

42. The definition of a matched component was given in section 5.3.

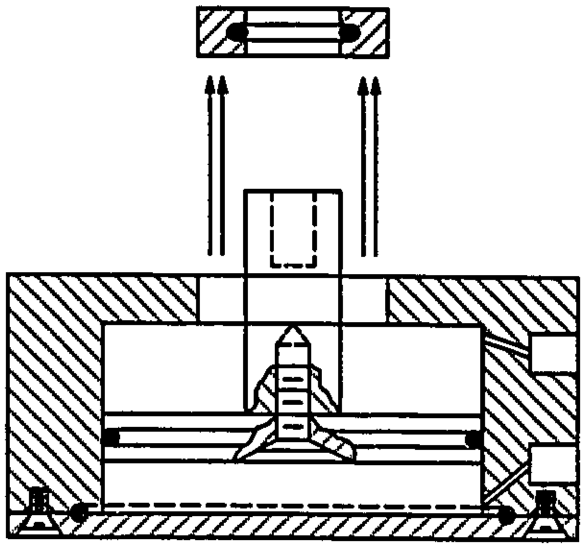


Fig. 5.11: A geometrically feasible disassembly operation where the moved subassembly {bearing, bearing_o_ring} is inaccessible.

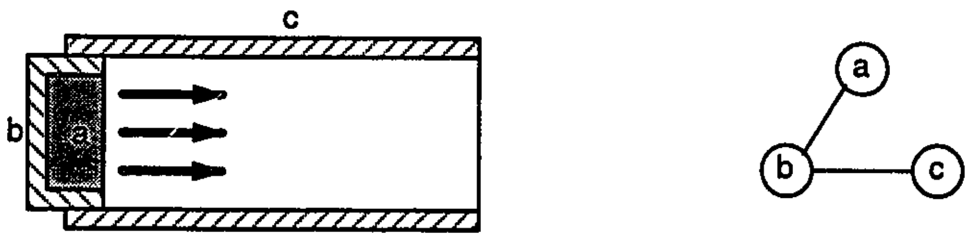


Fig. 5.12: A typical example of an inaccessible component (a) in a geometrically feasible disassembly operation.

This notion of accessibility of the matched component has not been implemented in GAPP, nor has an investigation been conducted to determine the complexity involved in its formalization. It promises to be very hard to compute automatically and constitutes an interesting area for future work.

5.4.2 Restricted components

To cope with accessibility constraints, GAPP relies on the notion of imposing restricted components in relations. For the example in figure 5.10, making the cover a restricted component for the attachment relation between the piston_rod and piston_screw prevents GAPP from generating the unfeasible operation. This specification expresses the intuitive fact that the establishment of the relation between the bearing and body is impossible if the piston_rod is part of any two subassemblies involved in the establishment of this relation. Similarly, solving the situation in figure 5.11 requires making the component piston_rod a restricted component in the relation between the bearing and the body. Solving the generic case shown in figure 5.12 requires making component "c" a restricted component in the relation between components "a" and "b".

5.4.3 Other constraints

The concept of imposing restricted components in relations can be used to model other types of (sometimes) more subtle constraints. Consider for example the situation depicted in figure 5.13. It is geometrically feasible to remove the bearing_o_ring along the z+ direction. The internal diameter of the body can also be assumed to be large enough to avoid any accessibility problems for the tool used to remove the bearing_o_ring. Assume that for some reason, the {bearing, bearing_o_ring} must be disassembled first, and only then can the bearing_o_ring be removed from the bearing. To model such a constraint, the body can be made a restricted component in the relation between the bearing and bearing_o_ring.

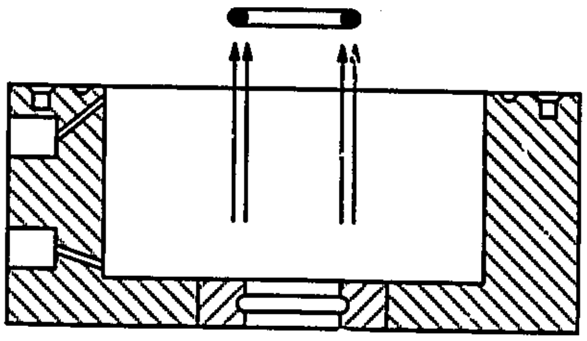


Fig. 5.13: A feasible disassembly operation to be avoided.

5.4.4 Transforming restricted components into forbidden states

The specification of restricted components for some relations of the graph model, used to express accessibility and other constraints, are translated into forbidden states in the directed graph of assembly states.

Let "r" be a relation in the graph model of some product; let "c" be a restricted component of "r"; let $K = \{ k_1, k_2, \ldots , k_i\}$ be the set of all cutsets of the unconstrained directed graph of assembly states which include "r"; let $S_{1i} = \{c_1, c_2, \ldots , c_j\}$ and $S_{2i} = \{c_1, c_2, \ldots , c_j\}$ be the set of components in the two subassemblies obtained from cutset k_i, respectively; and let $O = \{o_1, o_2, \ldots, o_n\}$ be the set of all assembly states containing any of the S_{1i} or S_{2i} in which "c" is included. The specification of restricted component "c" in relation "r" induces a constraint precluding all states in O from being generated in the directed graph of assembly states.

Consider a new state with two new subassemblies obtained from the removal of the edges in some cutset. If the new subassemblies in the state include any of the restricted components specified for any relation in the cutset, then the new state becomes part of O and is therefore forbidden.

Formally, if two new subassemblies are denoted by S_1 and S_2, and if the set of all restricted components in all relations of the cutset by which these two subassemblies were obtained is denoted by RES, an operation is feasible under the constraints induced by restricted components if:

$$S_1 \bigcup S_2 \bigcap RES = \{\}. \quad (5.15)$$

A single relation may have many restricted components. A single relation may also be part of many different cutsets. Therefore, there might exist many states of the directed graph which do not satisfy eq. (5.15) above. Consequently, the specification of restricted components in relations can drastically reduce the number of states in the directed graph of assembly states.

For the air cylinder, a total of 7 restricted components were specified for 6 relations in the graph model. Without considering geometric feasibility constraints, the graph was reduced to 151 nodes and 10,432 assembly sequences (paths). Considering both geometric feasibility constraints and constraints induced by the specification of restricted components on relations, the graph has 85 nodes with 728 feasible assembly sequences.

CHAPTER 6

EVALUATION CRITERIA

The previous chapter showed how various constraints can be used to greatly reduce the size of a product's directed graph of assembly states, compared to the size of the corresponding unconstrained graph. Nevertheless, even with the consideration of all constraints, the directed graph of assembly states might still be quite large and contain numerous feasible disassembly sequences.

This chapter describes four criteria used to evaluate the goodness of the disassembly operations associated with each edge of the directed graph of assembly states. As a result, among all possible paths of this graph, those which correspond to "better" disassembly sequences can be identified and selected.

The first four sections respectively describe the following four implemented criteria:

1– the number of re–orientations in the assembly sequence,

2– parallelism among assembly operations,

3– stability of the subassemblies, and

4– clustering of similar assembly operations.

The last section describes how conflicts among criteria may arise and how these conflicts are resolved by GAPP.

6.1 NUMBER OF RE–ORIENTATIONS

The choice of an assembly sequence which requires the least number of re–orientations helps reduce the assembly cycle time by eliminating re–orientation operations. It also avoids the design of complex assembly tools and fixtures often required for performing re–orientations. This section describes how re–orientations are computed in GAPP and how these computations can be used to rate the different paths in the directed graph of assembly states.

6.1.1 The need for re–orientations

The dichotomic assumption used in this work implies that every assembly operation consists of bringing two and exactly two subassemblies together to form a larger subassembly. The orientation of these subassemblies prior to the execution of assembly operation and the desired orientation of the new structure resulting from the assembly operation play an important role in determining whether a re–orientation is required.

Figure 6.1 illustrates this concept. In figure 6.1a, starting from the initial state of the parts, the goal configuration can be achieved without any re–orientation. In figure 6.1b, the initial orientation of the pin is such that achieving the goal configuration first requires its re–orientation. In figure 6.1c, a re–orientation of both parts is required. By simply changing the goal orientation in figure 6.1c to that of figure 6.1d, no re–orientation of the parts is now required.

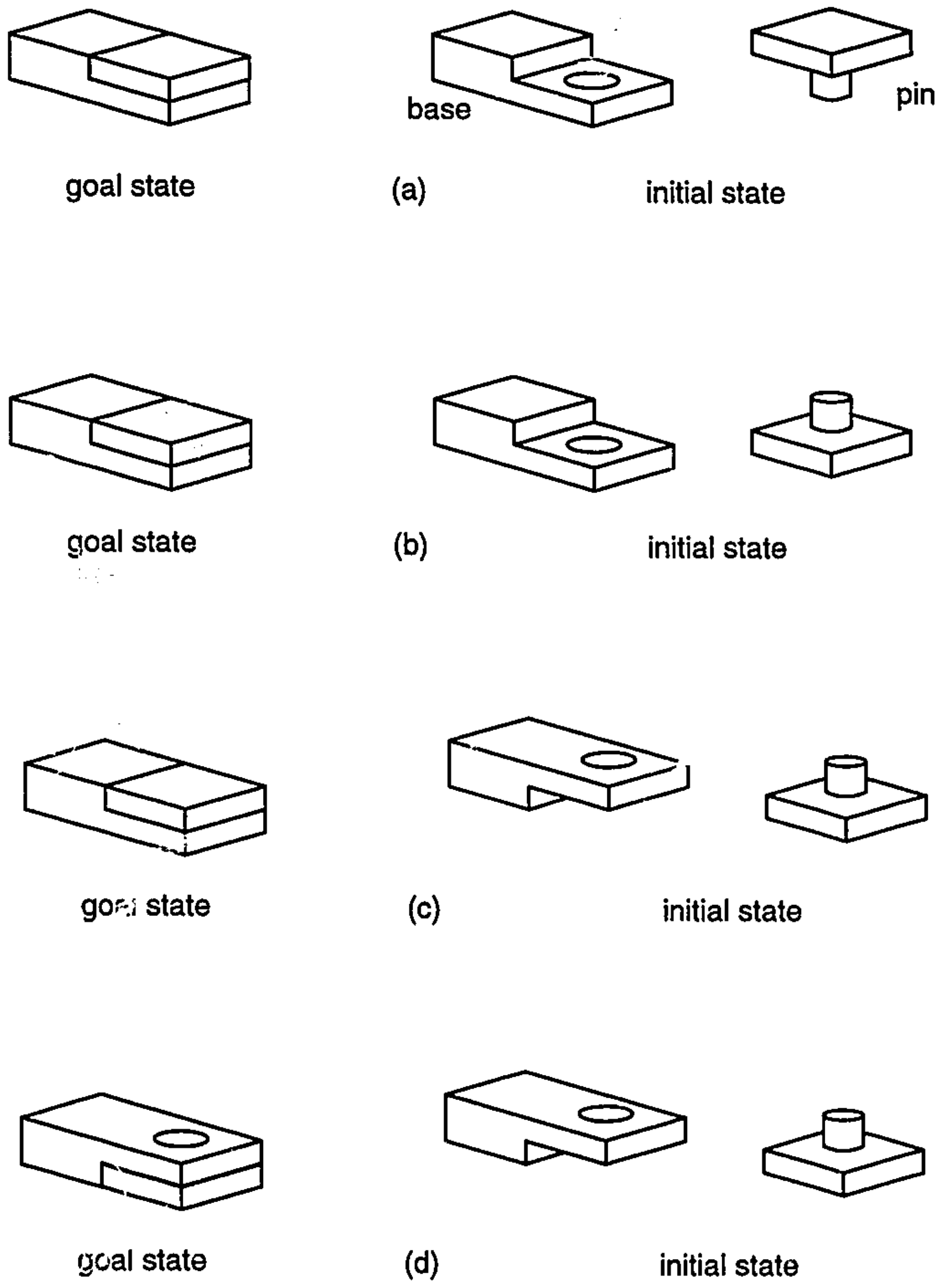


Fig. 6.1: Illustration of a the effects of constraining the initial and goal orientations of the parts in an assembly operation.

Although the same operation of mating the pin to the plate is performed, special constraints on the initial and goal configurations of these parts may require re–orientation. An important observation is that some re–orientation is definitly required if the initial orientation of the parts is different than their orientation in the final configuration. If all parts have the same orientation in both their initial and goal configurations, it is easily conceivable that there must exist a plan to assemble them without any re–orientation, as depicted in figures 6.1a and 6.1d.

Consequently, given some known orientation of the product in its assembled state, GAPP can always find a disassembly plan in which no re–orientation is required, i.e. in which the resulting orientation of all disassembled components is the same as their original orientation in the assembled product. To the author's knowledge, every assembly planner uses this fundamental assumption. The following sections describe concepts which enable GAPP to generate disassembly plans in which re–orientations can occur.

6.1.2 The notion of the moved and fixed subassemblies

Various properties of the two subassemblies involved in an assembly operation are generally such that a distinction can be made between which subassembly should be moved and which should be fixed in its execution. A typical example of this is provided by Haynes and Morris [18]: in an operation that mates a cup of coffee and a dining table, the cup is moved and placed on the table, and not vice–versa[43].

43. There are several criteria which can be used to decide which part is best moved or fixed in a given assembly operation. Examples of such criteria are: size, weight, flexibility, loose components, etc. For every relation defined in the product's graph model, the user specifies the component to be moved as part of the relation's definition.

One can apply this concept to the example in figure 6.1 and decide that the pin should be most naturally moved to the fixed base in executing the assembly operation that mates them both. In doing so, the situation depicted in figure 6.1a does not present much of a problem: simply pick the pin and insert it from above into the base. However, an important complexity arises in the situation depicted in figure 6.1d: the pin must be inserted from underneath. The question to be considered is whether such an operation should be considered feasible or not.

The clearance between the subassembly to be removed from underneath and the closest object with which it would collide when removed from that direction play an important role in trying to answer the above question. In figure 6.1d, if one imagines the assembled parts to be lying on a flat table, then the removal of the pin from the base using the downwards direction is clearly impossible because there is no clearance between the pin and the table. For some parts like those in figure 6.2, the clearance between part "a" and the "table" is large enough to enable the disassembly of part "a" from underneath.

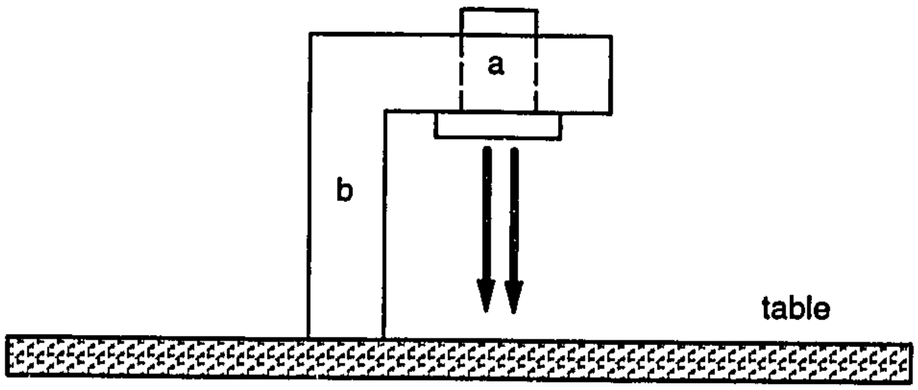


Fig. 6.2: Illustration of a disassembly operation where a part can be removed from underneath because of sufficient clearance.

Assuming that sufficient clearance exists to remove a part from below, the kinematic capabilities of the resource(s) used to perform such an operation also impose restrictions on its feasibility. A human operator can easily perform the operation in figure 6.2. With a little programming effort, a robot with a jointed–arm configuration could probably succeed in performing this operation as well. Special workheads and fixtures have also been used in automatic assembly which enable insertions of parts from below. Nevertheless, a resource like a SCARA robot could never perform such an operation, nor would any other resource which is restricted to top–down or sideways movements.

Due to the complexities involved in assembling (or disassembling) a part from underneath, GAPP assumes such operations to be unfeasible when the re–orientation criteria is being considered. The concept of the moved and fixed subassembly of an assembly operation, along with the assumption that the moved subassembly cannot be inserted from underneath, enable GAPP to generate disassembly plans in which the orientation of the components in their disassembled configuration might be different from that in their assembled configuration. In other words, GAPP generates plans in which re–orientations can occur.

Before actually describing how re–orientations are computed in GAPP, a few things need to be clearly understood. First, only the orientation of the product in its completely assembled state can be constrained. In generating the product's graph model, an orientation must be chosen. This orientation of the product as reflected by the graph model is implicitly part of the input of GAPP. On the other hand, the orientation of the completely disassembled components cannot be constrained. This orientation is decided by GAPP as it generates the disassembly sequences of the product. Therefore, although one always knows the orientation of the product prior to disassembly, one cannot predict the orientation of the completely disassembled components at the end of the disassembly process applied by GAPP. In particular, if the re–orientation criterion is turned on, the orientation of the disassembled components will be such that the least number of re–orientations will have been required in the disassembly sequence.

Second, it is important to understand that any re–orientation generated by GAPP applies to the subassembly before it is split into two smaller ones. For example, maintaining the reasonable assumption that the pin should be the moved subassembly, the {pin, base} subassembly in figure 6.1d would first be re–oriented, then the pin removed from above in the new orientation. In this case the resulting orientation of both disassembled components would be different from that in their assembled configuration.

Finally, it must be clear that a single component is never re–oriented by itself. Therefore a plan like the one in figure 6.1d cannot be solved by GAPP. This relates to the fact that the orientation of the disassembled components cannot be constrained, and that a component can only be re–oriented as part of a subassembly.

In the following sections, the subassembly to be split into two smaller ones, {pin, base} in figure 6.1 and {a, b} in figure 6.2, will be referred to as the *parent* subassembly.

6.1.3 Computing re–orientations in GAPP

Determining whether a disassembly operation requires a re–orientation of the parent subassembly is rather easy. GAPP first examines the relations in the cutset. It uses algorithm *make_subgraph()* presented earlier on page 99 to construct the two subassemblies S_1 and S_2 resulting from the removal of the relations in the cutset. It then picks either one of these two subassemblies and adds all its matched components to the set M. For each matched component in this set, it counts how many are moved and fixed, respectively, by analyzing the user–defined moved and fixed components defined apriori in the grap. model. Assuming that S_1 was chosen for examination, three outcomes are possible:

Case 1: the number of matched components in S_1 specified to be moved is greater than the number of matched components specified to be fixed,

#moved > #fixed;

In this first case, GAPP concludes that S_1 is the moved subassembly and S_2 the fixed subassembly in the disassembly operation that separates them. If the orientation of the parent subassembly is such that S_1 can only be disassembled along z–, GAPP further concludes that a re–orientation of the parent subassembly is required prior to removing S_1. As a result, S_1 is removeable along z+.

Case 2: the number of matched components in S_1 specified to be fixed is greater than the number of matched components specified to be moved,

#moved < #fixed;

In this second case, GAPP concludes that S_2 is the moved subassembly and S_1 the fixed subassembly in the disassembly operation that separates them. If the orientation of the parent subassembly is such that S_2 can only be disassembled along z–, GAPP further concludes that a re–orientation of the parent subassembly is required prior to removing S_2. As a result, S_2 is removeable along z+.

Case 3: the number of matched components in S_1 specified to be moved is equal to the number of matched components specified to be fixed,

#moved = #fixed.

In this last case, GAPP does not know which of S_1 or S_2 should be moved or fixed. The decision is made by looking at the possible disassembly directions of each. In particular, if one subassembly can be removed along z+, it is selected as the moved one. Otherwise S_2 is selected as the moved one by default.

This algorithm is implemented in the procedure presented in figure 6.3. The call to this procedure each time that a new node is generated in the directed graph of assembly states helps determine if a re–orientation is required in the disassembly operation which transformed the parent node into the new child node. Keeping track of how many such re–orientations are required in all paths from the root node to the goal node permits the selection of the path with the smallest number of required re–orientations.

```
procedure find_re_orient( $S_1$, $S_2$, cutset )
    put matched components of $S_1$ in M
    for all relations in cutset:
        if this relation is ambiguous
            pick another relation
        if the moved component of this relation is in M
            moved++
        else
            fixed++
    if moved > fixed
        $S_1$ is the moved subassembly
        if disassembly directions of $S_1$ = z– only
            return re–orientation required
    if fixed > moved
        $S_2$ is the moved subassembly
        if disassembly directions of $S_2$ = z– only
            return re–orientation required
    if fixed = moved
        if $S_1$ has z+ disassembly direction
            $S_1$ is the moved subassembly
        else
            $S_2$ is the moved subassembly
    return no re–orientation required
```

Fig. 6.3: Procedure "find_re_orient()".

6.1.4 An example

Figures 6.4 and 6.5 will be used to illustrate the concepts developed in the previous section.

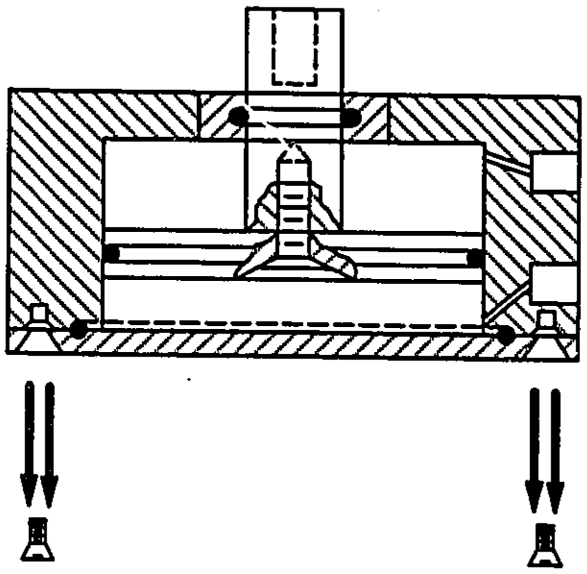


Fig. 6.4: A disassembly operation which consists of removing the cover_screws from the air cylinder.

Figure 6.4 depicts a possible initial disassembly operation of the air cylinder which consists of removing the cover_screws. Figure 6.5 shows the graph representation of this disassembly operation. Two relations, R1 and R2, are included in the cutset. This cutset splits the initial assembly into the two subassemblies S_1 = {cover_screws} and S_2 = {bearing, bearing_o_ring, piston_rod, piston, piston_o_ring, piston_screw, body, cover_o_ring, cover}. The two sets of matched components in S_1 and S_2 are {cover_screws} and {body, cover}, respectively. Using the former set, relations R1 and R2 reveal that the sole matched component {cover_screws} has been specified as moved one twice. Using the latter set, relations R1 and R2 reveal that both matched components {body, cover} have been specified as fixed.

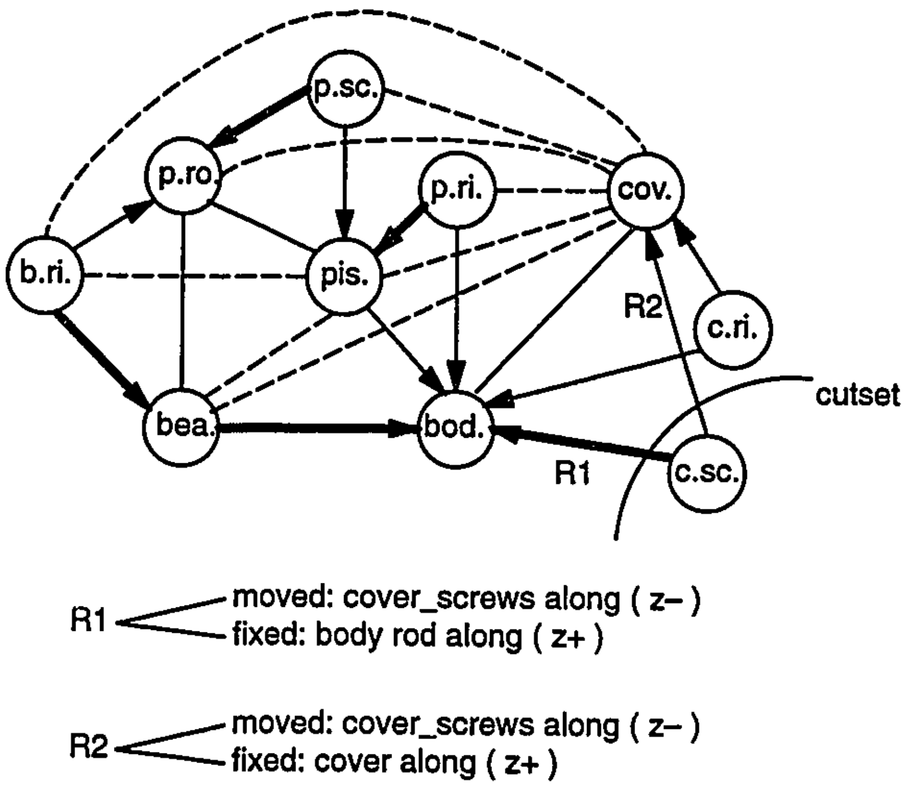

p.sc.
p.ro.
p.ri.
cov.
b.ri.
pis.
R2
c.ri.
cutset
bea.
bod.
R1
c.sc.
R1 — moved: cover_screws along (z–)
fixed: body rod along (z+)
R2 — moved: cover_screws along (z–)
fixed: cover along (z+)

Fig. 6.5: Graphical representation of the disassembly operation in figure 6.4.

At this point, GAPP concludes that in an assembly operation which mates S_1 = {cover_screws} with S_2 = {bearing, bearing_o_ring, piston_rod, piston, piston_o_ring, piston_screw, body, cover_o_ring, cover}, S_1 is the moved subassembly and S_2 the fixed one.

The orientation of the air cylinder in figure 6.4 is such that the moved cover_screws must be removed along z–. This represents the only possible disassembly direction of this subassembly. GAPP further concludes that a re–orientation of the air cylinder is required prior to this disassembly operation. As a result of this re–orientation, the cover_screws can be removed along z+.

6.1.5 Effects on assembly states

The decision by GAPP of which subassembly should be moved and which should be fixed is made by analyzing the initial user specifications of the moved and fixed components in each relation in the graph model. The decision of whether a re–orientation is required or not is based on the assumption that the moved subassembly cannot be removed from underneath. In themselves, these decisions can be easily computed at each state transition of the directed graph of assembly states. A non–trivial complication resulting from a re–orientation is that the new orientation of the subassemblies involved must be updated in the new state after the disassembly operation.

Performing the disassembly operation in figure 6.4 yields the new configuration in figure 6.6. The corresponding new child state is depicted in figure 6.7. Because of the re–orientation, the list of disassembly directions of each component in each of the new child state's relation must be updated. Every disassembly direction along z– becomes one along z+, and vice–versa. For example, edge R1 in figure 6.7 originally specified {x+, x–, y+, y–, z–} and {x+, x–, y+, y–, z+} as the possible disassembly directions of its incident components cover and body, respectively. In figure 6.6, these disassembly directions must be updated to become {x+, x–, y+, y–, z+} and {x+, x–, y+, y–, z–}, respectively. This type of update must be performed for all relations in any two subassemblies resulting from a disassembly operation requiring a re–orientation. This dependency of a state on the orientation of the subassemblies that it contains may slightly increase the number of nodes in the directed graph of assembly states.

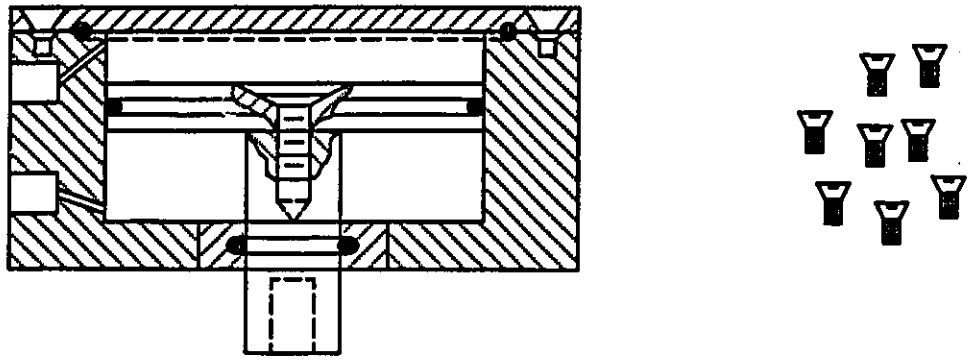


Fig. 6.6: Configuration resulting from the disassembly operation and re–orientation in figure 6.4.

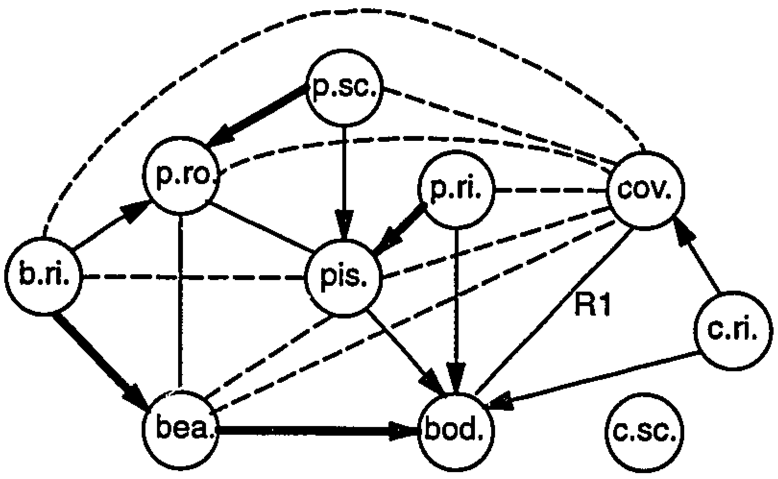


Fig. 6.7: Assembly state corresponding to the configuration of the air cylinder in figure 6.6.

6.1.6 Search cost associated with re–orientations

If the re–orientation criterion is turned on, GAPP associates a cost to the new state of the directed graph of assembly states resulting from a disassembly operation which required a re–orientation. The value "re" of this cost is obtained from the constant function:

$$re = \begin{cases} w_{re} & \text{if } \textit{re–orientation required} \\ 0 & \text{if } \textit{no re–orientation required} \end{cases} \qquad (6.1)$$

where "w_{re}" is the relative weight of the re–orientation criterion as specified by the user. For example, if the user specifies a unit weight of 25 (on a scale of 0 to 100) for this criterion, a cost of 25 units is added to every new child state resulting from a disassembly operation requiring a re–orientation.

6.1.7 Underestimating re–orientation's remaining search cost

The next chapter will discuss the use of various search methods for traversing the directed graph of assembly states in order to find an optimal disassembly sequence of some product. Among these methods is the well known A^* algorithm. This algorithm guarantees finding the optimal solution in an OR graph, provided that the following three conditions are satisfied:

1– the cost of getting from the root node to a particular node of the search graph can be determined precisely,

2– the cost of getting from the particular node to some goal node can be reasonably estimated, and

3– the cost in 2 is never over–estimated.

This section deals with conditions 2 and 3 above, i.e. methods for finding good underestimations of the cost of re–orienting. This problem is depicted in figure 6.8: given a newly generated node in the directed graph of assembly states, an underestimation of the remaining cost associated with re–orientations in the best path from this new child to the goal must be obtained.

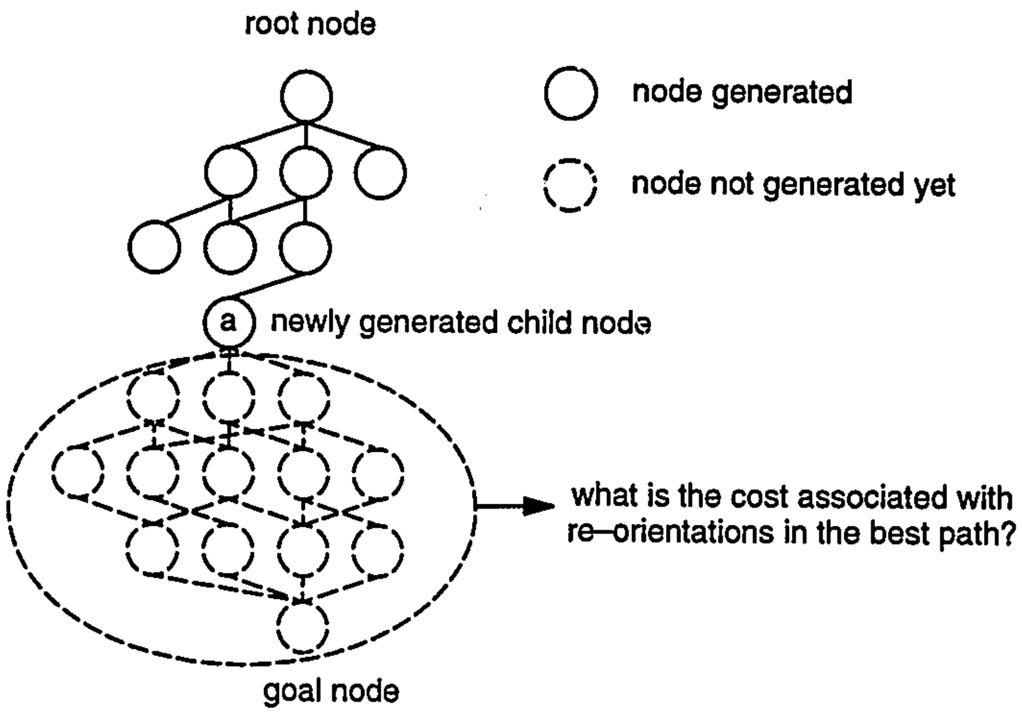

root node
node generated
node not generated yet
a
newly generated child node
what is the cost associated with re–orientations in the best path?
goal node

Fig. 6.8: At one instance in the generation of the directed graph of assembly states: node "a" has just been generated and an underestimation of the cost associated with re–orientations in the best path from this node to the go[illegible] must be computed.

An analysis of the cutsets in the new child state can lead to a good underestimation of the cost associated with re–orientations in the best path between this child and the goal. Before describing the algorithm which performs this analysis, it is shown that at best, each induced subgraph of an assembly state requires at least zero and at most one re–orientation in the disassembly operations that splits this subgraph into individual components.

To illustrate this, refer to figure 6.9. It shows a simple plate with a pin to be removed from the top and two pins to be removed from underneath. Without loss of generality, the graph model of these parts can be considered an induced subgraph of some larger graph and can therefore be seen as one of many subassemblies of the state in figure 6.9c. It is now desired to compute an underestimation of the number of re–orientations required to disassemble this subassembly.

An analysis of cutset {R1} reveals that moved component pin1 must be removed from underneath, which requires a re–orientation. So does moved component pin2, by analyzing cutset {R2}. On the other hand, cutset {R3} specifies that moved component pin3 can be removed from above. A total of six permutations in the order of execution of the disassembly operations associated with each of these cutsets is possible, as follows:

1– R1, R2, R3

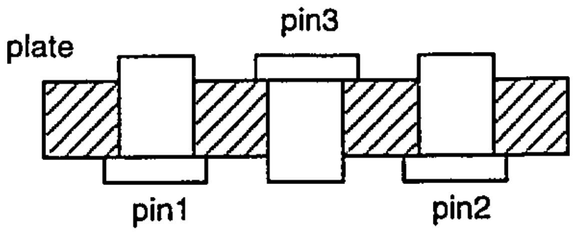
2– R1, R3, R2

3– R2, R1, R3

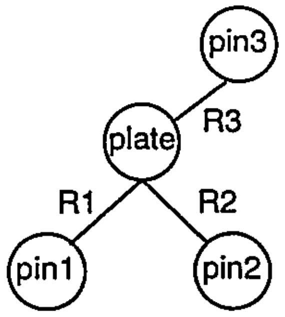
4– R2, R3, R1

5– R3, R1, R2

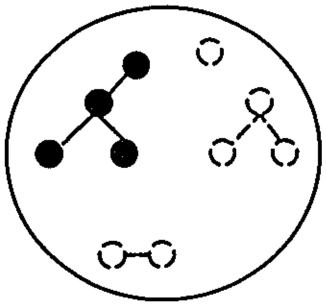
6– R3, R2, R1.



(a)



(b)



(c)

R1 — moved: pin1 along (z–)
R1 — fixed: plate along (z+)

R2 — moved: pin2 along (z–)
R2 — fixed: plate along (z+)

R3 — moved: pin3 along (z+)
R3 — fixed: plate along (z–)

(d)

Fig. 6.9: (a) a subassembly to be disassembled; (b) induced subgraph of (a); (c) assembly state containing this induced subgraph; (d) two cutsets by which the subassembly can be disassembled.

Out of the six possible permutations, 5 and 6 require a single re–orientation; 1 and 3 require two re–orientations; 2 and 4 require three re–orientations. The last four permutations result in disassembly sequences in which more than one re–orientation is required. Therefore, they are not optimal in this respect.

The generalization of these results is as follows: let A be the set of all pins that can be removed from above; let B be the set of all pins that can be removed from below; let "min" be the minimum number of re–orientations required to remove all pins from the plate; and let "max" be the maximum number of re–orientations required to remove all pins from the plate; then it can be readily verified that:

$$min = \begin{cases} 0 & if\ |B| = \{\} \\ 1 & otherwise \end{cases} \tag{6.2}$$

$$max = \begin{cases} |A| + |B| & if\ |A| = |B| \\ |B| + 1 & if\ |A| > |B| \\ 2 \times |A| + 1 & if\ |B| > |A| \end{cases} \tag{6.3}$$

For the example in figure 6.9:

$$|A| = 1, \quad |B| = 2, \quad \min = 1, \quad \max = 2 \times |A| + 1 = 3. \tag{6.4}$$

These results can apply to any arbitrarily induced subgraph consisting of other components than plates and pins: the set A above becomes the set of all subassemblies of this induced subgraph which can be removed from above; B becomes the set of all subassemblies of this induced subgraph which can be removed from below; "min" becomes the minimum number of re–orientations required in the disassembly operations that splits this subgraph into individual components; the mapping for "max" is not relevant.

An underestimation of the cost associated with re–orientations in the best path between a new child node and the goal node is obtained from an analysis of the cutsets of this new child. For each cutset, a first check determines if the cutset results in a feasible disassembly operation. If not, a next cutset is selected for investigation. If so, a check is performed to see if any of the incident components of the relations in this cutset are included in set POOL. If so, a next cutset is selected for investigation. If not, the two subassemblies which would result from its removal are computed using procedure *make_subgraph()* shown earlier on page 99.

The two subassemblies are then passed as argument to procedure *find_re_orient()* in figure 6.3. As a result of this call, GAPP knows if the cutset under investigation would result in a re–orientation in a later disassembly operation. If a re–orientation would indeed be required, a re–orientation counter is incremented by one. As a re–orientation is found for the induced subgraph to which this cutset applies, and as either zero or one such re–orientation of the subassembly represented by this induced subgraph is required in the best possible case, any other remaining unprocessed cutset which applies to the same induced subgraph need not be examined. This is achieved by adding all components of the induced subgraph to which the cutset applies to the set POOL. This algorithm stops when there are no more cutsets to be investigated. The procedure by which this algorithm is implemented is presented in figure 6.10.

Using the value of the counter which gets increment in the procedure, an underestimation "u_re" of the cost associated with re–orientations in the best path from a new child node to the goal is simply given by:

$$u\_re = w_{re} \times counter \qquad (6.5)$$

It is worth mentioning that it is absolutely necessary to compute the feasibility of the disassembly operation associated with the cutset under investigation. To illustrate why, assume there exists a cutset of an induced subgraph whose corresponding disassembly operation is unfeasible and requires a re–orientation. Also assume that the disassembly operations associated with all other cutsets of this induced subgraph are feasible and do not require any re–orientation. Processing this "unfeasible" cutset in procedure *underestimate_re_orient()* would mistakenly increment the counter by one. This would result in an overestimation of "u_re" in eq. (6.5), which would in turn violate the condition for ensuring optimality in performing an A^* search over the directed graph of assembly states.

```
procedure underestimate_re_orient( new_child )
    u_re = 0
    POOL = {}
    for all cutsets of new_child:
        if any incident component of the relations in this cutset
        is included in POOL
            pick another cutset
        $S_1$, $S_2$ = make_subgraph( incident_component, cutset)
        flag = find_re_orient( $S_1$, $S_2$, cutset )
        if flag = TRUE
            add $S_1$ and $S_2$ to POOL
            u_re++
        pick another cutset
    return u_re
```

Fig. 6.10: Procedure "underestimate_re_orient()".

6.2 PARALLELISM AMONG ASSEMBLY OPERATIONS

The selection of an assembly plan which allows parallelism leads to significant reductions in total assembly cycle time but may require additional equipment. In contrast, an assembly plan in which no parallelism can occur is likely to require a longer cycle time but can theoretically be performed using a single flexible resource (for example a robot). It is important to analyze the economic trade–off between longer assembly time and the use of additional equipment early in the product design and manufacturing planning phase, to ensure cost effectiveness. This section describes how parallelism is computed in GAPP and how these computations can be used to rate the different paths in the directed graph of assembly states.

6.2.1 Type1 versus type2 parallelism

Type1 parallelism refers to the simultaneous execution of two or more assembly operations in which the resulting subassemblies have no component in common. In other words, if operation1 mates subassemblies S_1 and S_2, and if operation2 mates subassemblies S_3 and S_4, then the simultaneous execution of these two operations presents type1 parallelism if:

$$(S_1 \bigcup S_2) \bigcap (S_3 \bigcup S_4) = \{\}. \qquad (6.6)$$

The simultaneous fitting of the piston_o_ring to the piston and of the bearing_o_ring to the bearing in the air cylinder product is an example of type1 parallelism; so is the placing of the cover against the body simultaneously with the screwing of the piston_screw to the piston_rod.

Type2 parallelism refers to the simultaneous execution of two or more assembly operations in which the resulting subassemblies do have components in common. In other words, if operation1 mates subassemblies S_1 and S_2, and if operation2 mates subassemblies S_3 and S_4, then the simultaneous execution of these two operations presents type2 parallelism if:

$$S_1 \bigcap (S_3 \bigcup S_4) = S_1 \qquad \text{or} \qquad S_2 \bigcap (S_3 \bigcup S_4) = S_2. \tag{6.7}$$

The simultaneous fitting of the piston_screw and piston_o_ring to the piston is an example of type2 parallelism; so is the placing of the cover against the body simultaneously with the fitting of the bearing to the body. These two examples have the piston and body as a common component, respectively.

6.2.2 Parallelism in GAPP

GAPP can evaluate assembly plans only with respect to type1 parallelism, as any operation executed in type2 parallelism violates the dichotomy assumption.

To determine how good a disassembly operation is with respect to type1 parallelism, a count of the number of component in the two subassemblies resulting from the disassembly operation is performed. The smaller the difference between the component count in each subassembly, the better the operation with respect to the parallelism criterion.

At one extreme, if a feasible plan exists in which every disassembly operation always splits a parent subassembly into two subassemblies with an equal number of components, then maximum type1 parallelism is achieved[44]. At the other extreme, if every operation in any feasible plan always consists of removing a single component at a time, then no type1 parallelism exists.

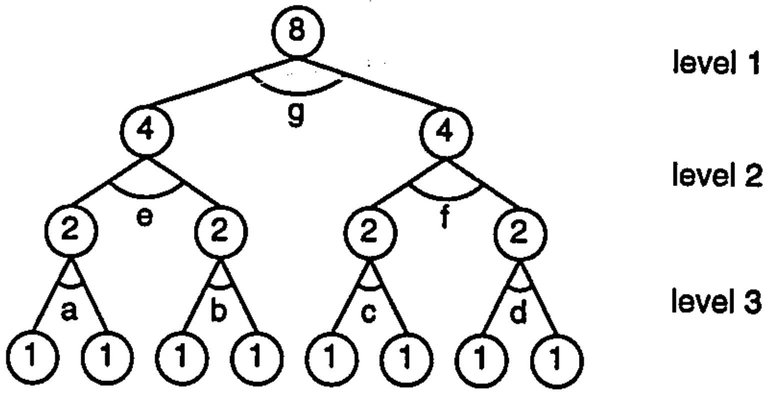


Fig. 6.11: An AND / OR tree representing one possible assembly plan of a hypothetical unconstrained product of 8 components.

To illustrates this, refer to figure 6.11. It shows the AND / OR tree representation of an assembly plan for a hypothetical product with 8 components. The nodes in this graph represent various subassemblies. Hyperarcs represent disassembly operations that split a parent subassembly into two smaller subassemblies. The number of components in each subassembly at each node is labelled in the node.

In this plan, every disassembly operation splits a parent node into two child nodes with an equal number of components. As a result, operations "a", "b", "c" and

44. Note that this can be achieved only if the parent subassembly has an even number of components. For a parent subassembly with an odd number of components, the best that can be achieved is to split this subassembly into two subassemblies whose respective number of components differs by one.

"d" can be performed in parallel, then operations "e" and "f" in parallel, then operation "g" by itself. Altogether, 6 operations can be performed in type1 parallelism in this plan.

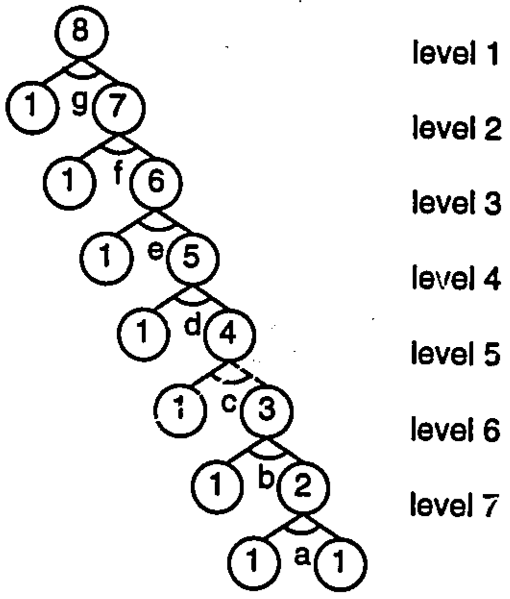


Fig. 6.12: An AND / OR tree representing another possible assembly plan of a hypothetical unconstrained product of 8 components.

Consider now figure 6.12. It shows another AND / OR tree for the same 8–components product. However in this case every disassembly operation consists of removing a single component from the parent subassembly, yielding a child node with one component and another with one less the number of components of its parent node. No type1 parallelism is ever possible in this case, as no two hyperarcs ever appear in the same level in the tree.

Clearly, the "shallowest" the AND / OR tree is, the more hyperarcs can appear at each level in the tree, and the more type1 parallelism is possible [29]. Therefore a necessary condition for maximizing parallelism among disassembly operations is that their corresponding AND / OR tree representation be the shallowest of all the trees representing all possible partial orders of these operations.

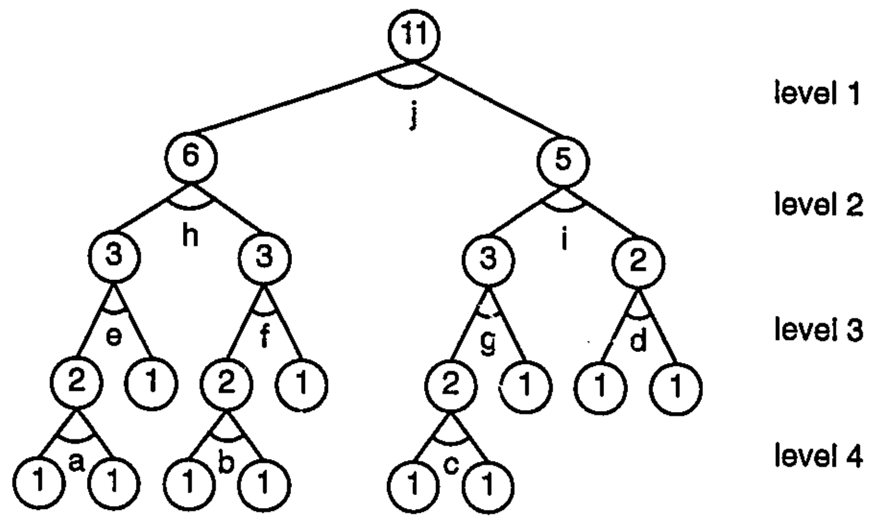
The depth "d" of the shallowest AND / OR tree for a totally unconstrained product of "n" components can be computed as follows:

$$d = \lceil \log_2 n \rceil \tag{6.8}$$

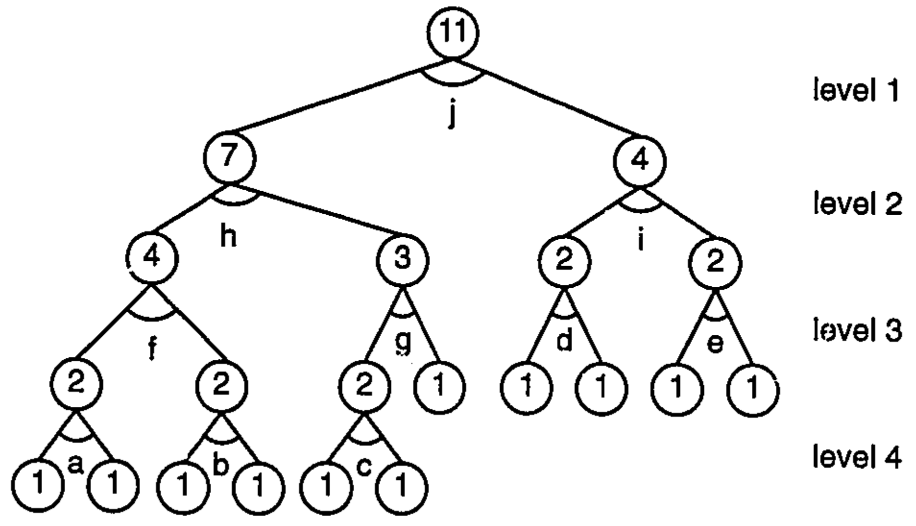
where $\lceil \; \rceil$ denotes the ceiling operator, in this case the smallest integer larger than $\log_2$ n. For example, the depth of the shallowest AND / OR tree of a totally unconstrained product of 11 components is 4, as $\log_2 11 = 3.32$ and $\lceil 3.32 \rceil = 4$.

Still, there can be more than one tree of equal shallowest depth for products with a certain number of components. For example, figure 6.13 shows two shallowest trees of depth 4 for an 11–components product. Note that the one which splits the parent nodes into two child nodes with the smallest difference in their component count, i.e. figure 6.13a, is the one in which the larger number of operations can be performed with type1 parallelism, i.e. 9 compared to 8 for figure 6.13b.

Therefore, a sufficient condition for maximizing parallelism among disassembly operations is that these operations always consist of splitting the parent subassembly into two subassemblies with the smallest difference in their component count. This difference is "zero" if the parent subassembly has an even number of components, and "one" otherwise.



(a)



(b)

Fig. 6.13: Two shallowest AND / OR trees of a hypothetical unconstrained product of 11 components: in (a), operations {a, b, c, d}, then {e, f, g}, then {h, i} can be performed in parallel, for a total of 9; in (b), operations {a, b, c, d, e}, then {f, g, i} can be performed in parallel, for a total of 8.

6.2.3 Search cost associated with parallelism

To avoid making the distinction between a parent subassembly with an even or odd number of components, both possible values in this best case situation are merged into a single one using some convention. In particular, if "dif" denotes the difference in the component count of the two subassemblies and "n" denotes the number of components in their parent subassembly, then the convention value $\overline{dif}$ used by GAPP is given by:

$$\overline{dif} = \begin{cases} 0 & \text{if } dif = 0 \text{ and } n \textit{ is even} \\ 0 & \text{if } dif = 1 \text{ and } n \textit{ is odd} \\ dif & \textit{otherwise} \end{cases} \tag{6.9}$$

In the worst case, it can be readily verified that the maximum difference "d" between the component count of two subassemblies resulting from splitting a parent subassembly is simply given by:

$$d = n - 2 \tag{6.10}$$

For example, in the worst case, a parent subassembly with 11 parts can be split into two subassemblies where one has 1 component and the other 10, for a difference between their component count of 10–1 = 9, which is also equal to 11–2 as given by eq. (6.10) above.

The cost associated with parallelism can now be defined as the linear function:

$$pa = \frac{w_{pa} \times \overline{dif}}{d} \tag{6.11}$$

where "w_{pa}" is the relative weight of the parallelism criterion as specified by the user, $\overline{dif}$ is the value of the difference in the component count of the two subassemblies as given by eq. (6.9), and "d" is the maximum value of this difference as given by eq. (6.10).

For example, if the user specifies a weight of 25 (on a scale of 0 to 100) for the parallelism criterion, the operation associated with hyperarc "j" in figure 6.13a and 6.13b results in unit costs of 0 and 7.33, respectively. Note that if $\overline{dif} = d$, then $pa = w_{pa}$ which is the maximum possible value of the cost associated with type1 parallelism for any possible disassembly operation.

If the parallelism criterion is turned on, meaning that the user has not specified a value of "0" as its relative weight "w_{pa}", this cost is computed and added to the new child state resulting from a disassembly operation in the directed graph of assembly states.

6.2.4 Underestimating parallelism's remaining search cost

As in the case for the re–orientation criterion, given a newly generated node in the directed graph of assembly states, an underestimation of the remaining cost associated with parallelism in the best path between this new child and the goal must be obtained if the A^* search method is used.

Recall that in the case of the re–orientation criterion, a fairly good underestimation could be obtained from an analysis of the cutsets in each induced subgraph of the new child state. As each state knows about its cutsets, the information that was reasoned upon to obtain the underestimation was local to the child state.

Unfortunately, such a good underestimation for the parallelism criterion is not as straightforward to obtain as in the case of the re–orientation criterion. To explain why, first note that for any new child node, there must exist an unconstrained path from that node to the goal in which the cost associated with type1 parallelism as given by eq. (6.11) has the value "0" for every disassembly operation in this path. One such path is presented in figure 6.14 for a hypothetical new child state.

It follows directly that a lower bound for the cost associated with parallelism in the best path from the new child to the goal is "0" and this value can always be used as an underestimation for this criterion[45]. Although this value is certainly an underestimation for this criterion, it is most probably not a good one. That is, this value assumes that the product is totally unconstrained. By adding constraints on disassembly operations in the remaining paths, it is very likely that this lower bound will never be achieved. For example, it is possible that the disassembly operations resulting in node "B" in the unconstrained path in figure 6.14 might be physically unfeasible.

45. The value "0" is actually a trivial underestimation for any criterion.

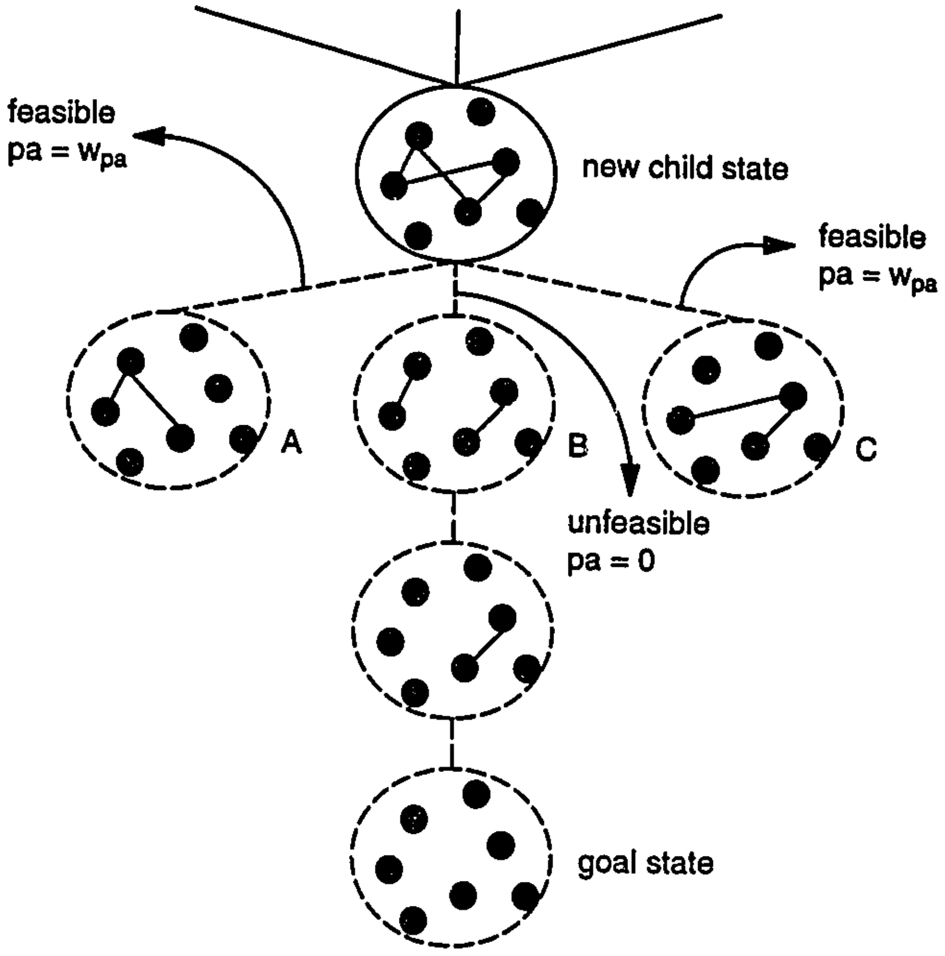


Fig. 6.14: An unconstrained path from a new child state to the goal which is optimal with respect to type1 parallelism.

A much better underestimation could be obtained by considering only the feasible remaining paths from the new child to the goal, i.e. paths in which every disassembly operation is feasible. However, since these paths have not been generated at the time when the underestimation is needed, one must always assume that an optimal unconstrained path like the one in figure 6.14 will turn out to be a feasible one, which means that the value "0" is actually the best possible underestimation that one can get for this criterion.

As will be shown in the next chapter, such a trivial value for an underestimation of any criterion is not very helpful in trying to reduce the combinatorics involved in the expansion of the directed graph of assembly states. The strategy used to overcome this problem for the parallelism criterion consists of "looking one step ahead" of the new child node for which an underestimation is required. That is, all the children of this new child are temporarily generated and the cost associated with type1 parallelism as given by eq. (6.11) is computed for each corresponding disassembly operation. The minimum cost of all temporary children is kept as an underestimation "u_pa" for the original child node. That is:

$$u\_pa = min\left(\frac{w_{pa} \times \overline{dif}}{d}\right) \quad for\ all\ tc_i \tag{6.12}$$

where "tc_i" is the ith temporary child of the new child for which an underestimation is required.

For example, assuming that the disassembly operation resulting in node "B" in figure 6.14 is indeed unfeasible, the new improved underestimation would be the maximum cost "w_{pa}", as computed for the disassembly operations resulting in nodes "A" and "C".

Obviously, as this new value is obtained by considering only feasible disassembly operations, on average it will represent a much better underestimation than the trivial "0" value in which every remaining operation is assumed to be feasible. The only exception is the case where one of the feasible "look ahead" disassembly operation also resulted in a value of "0" for the cost associated with parallelism.

If the new child for which an underestimation is required is actually the goal state, no temporary child is ever generated and the value returned by eq. (6.12) is "0", which is a perfect estimation. If there is only one remaining disassembly operation to transform the new child into the goal state, this goal state is generated as the only temporary child and a perfect estimation of "0" is once again obtained from eq. (6.12).

If there are "n" remaining disassembly operations to transform the new child into the goal state, an arbitrary number of temporary child states of this new child can be generated. The cost of the best path from the new child to the goal consists of the sum of the costs of each disassembly operations in this path. Eq. (6.12) returns the lowest of all cost values computed for every disassembly operation resulting in every temporary child. This minimum value is the cost of the first disassembly operation in the best path from the new child to the goal. As any other remaining disassembly operations in this best path either adds to the cost or all have a cost of zero, eq. (6.12) is guaranteed to be an underestimation of the cost of the best path from the new child to the goal.

It is interesting to note that "looking ahead" of the new child node for which an underestimation is required can be extended deeper. In the extreme case, one could generate the whole portion of the directed graph of assembly states from the new child node to the goal and obtain a perfect estimation of the cost associated with parallelism in the best path from this new child to the goal. However, the computational expense of obtaining such a perfect estimation would not be justified.

6.3 STABILITY OF THE SUBASSEMBLIES

6.3.1 Introduction

The choice of an assembly sequence with highly stable subassemblies improves the reliability of assembly operations. This reduces assembly cycle time by preventing errors during execution. Moreover, it reduces the need for tools and fixtures, greatly simplifies their design and drastically reduces their cost. The flashlight in figure 6.15 will be used to illustrate how the execution of different disassembly operations might influence the stability of the two resulting subassemblies.

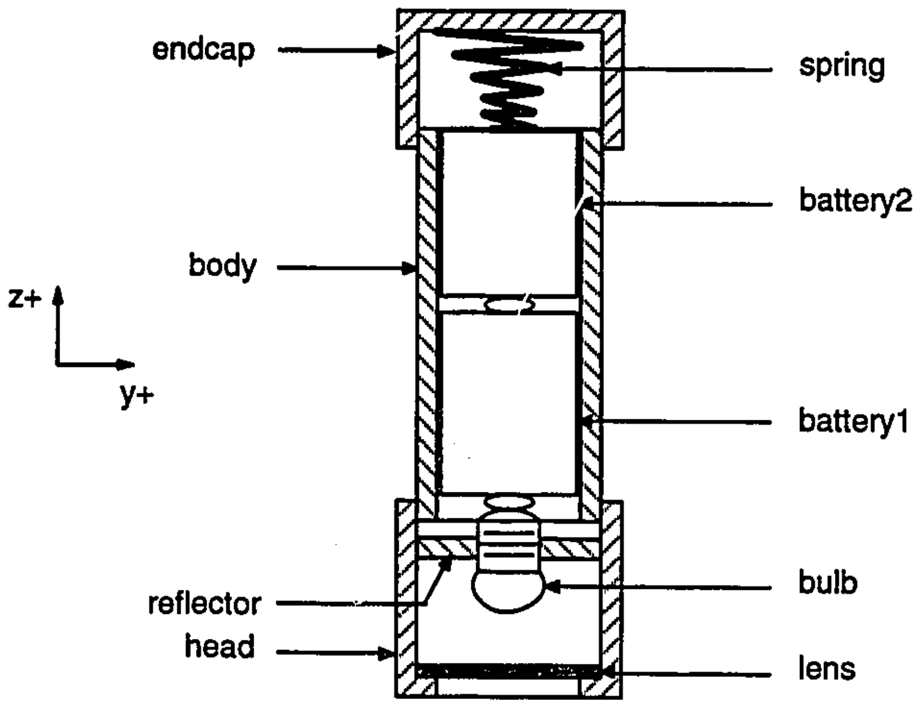


Fig. 6.15: Simplified geometric model of the flashlight product.

Figure 6.16 shows three feasible disassembly operations that can be applied to the completely assembled flashlight. Knowing that the spring is attached to the endcap, that both batteries can slide freely inside the body, and that the area of contact between battery2 and battery1 as well as between battery1 and the bulb is very small, the first operation in figure 6.16 is clearly the best one, as far as the stability of the two resulting subassemblies is concerned.

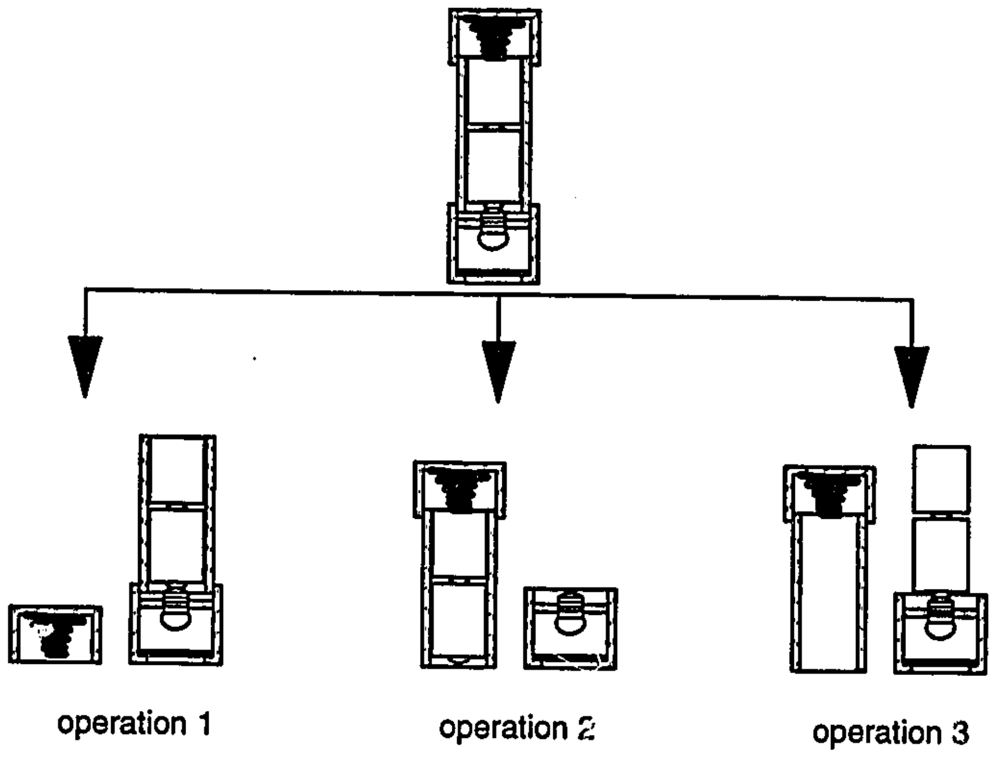


Fig. 6.16: Three initial disassembly operations that can be applied to the completely assembled flashlight.

In particular, in the second operation both batteries are likely to slide out of the body and fall when the {endcap, spring, body, battery1, battery2} subassembly is removed from above. In the third operation, both batteries become highly unstable and fall out. In the fist operation, the removed spring and endcap remain attached together while at the same time their removal leaves the {battery1, battery2, body, head, lens, reflector, bulb} subassembly in a stable state.

By measuring the stability of the two subassemblies resulting from the disassembly operations in any path of the directed graph of assembly states, one can select the path in which the overall stability is maximum. The next section describes how this notion of subassembly stability can be evaluated from the computation of the degrees of freedom of the components in this subassembly, and how maximizing subassembly stability maps nicely into minimizing the number of degrees of freedom of the components in a subassembly.

6.3.2 Degrees of freedom

A component in 3–D space has a maximim of 6 degrees of freedom: 3 translations and 3 rotations. In GAPP, a distinction is made between the actual direction of a translation or rotation involved in a degree of freedom, which gives a total of 12 "degrees of freedom" for this same component: {x+, x–, y+, y–, z+, z–, X+, X–, Y+, Y–, Z+, Z–}. The lower case and upper case letters denote translations and rotations, respectively.

As GAPP does not consider rotations, only 6 possible translations remain. These are: {x+, x–, y+, y–, z+, z–,}. Each correspond to the 6 disassembly directions that can be specified for a relation in the product's graph model (chapter 3). Therefore the degrees of freedom in GAPP are actually "translational" degrees of freedom.

When two components are in physical contact, the degrees of freedom that they present are reduced depending on the type of the contact. For example, placing a square block on top of another leaves 5 degrees of freedom for each block. Fitting a cylindrical peg into a hollow cylinder open at both ends leaves only 2 translational degrees of freedom for each part. The less degrees of freedom that two physically contacting components have, the more constrained the relative motion between them is, and the more stable these two components are.

The computation of the cost associated with the stability criterion is based on an evaluation of the degrees of freedom of each component in the new state resulting from a disassembly operation. The degrees of freedom in a state is simply the sum of the degrees of freedom of the components in this state. By convention, any unconnected component in a state is assumed to have zero degree of freedom. A direct implication of this is that a goal state in which all components are unconnected has zero degree of freedom.

For a new child state, the components in this state are processed sequentially and the computation of their individual degrees of freedom is performed. This computation of the degrees of freedom of a single component "c" is as follows: identify the set R of all contact and attachment relations of the new child state in which this component is involved. For each such relation "r_i" in R, identify the corresponding possible disassembly direction "d_i" of this component, as returned by the function:

$$d_i = directions(r_i, c) \tag{6.13}$$

Identify the directions which are common to all "d_i" and add them to the set D. The analysis of these common disassembly directions determine the degrees of freedom of "c". Two cases can occur:

Case 1: if D is empty, component "c" has 0 degree of freedom. This can happen either if "c" is unconnected or if its contacts and attachments with other components in the state are such that they completely immobilize it.

Case 2: if D is not empty and does not contain "z–" as one of its elements, count the number of elements in it. This number corresponds to the degrees of freedom of "c" in the new state.

Figure 6.17 will be used to illustrate these concepts. It shows two possible disassembly operations that can be applied to the three blocks shown earlier.

In the state resulting from operation 1, the pin has 0 degree of freedom because it is not connected. The plate has 4 degrees of freedom due to its contact with the base, namely {z+, x+, x–, y+}. Similarly, the base has 4 degrees of freedom due to its contact with the plate, namely {z–, x+, x–, y–}. In total, the state resulting from operation 1 has:

$$0 + 4 + 4 = 8 \text{ degrees of freedom} \tag{6.14}$$

In the state resulting from operation 2, the base has 0 degree of freedom because it is not connected. The pin has 1 degree of freedom, namely {z+}, as a result of its connection with the plate. Similarly, the plate has 1 degree of freedom, namely {z–}, as a result of its connection with the pin. In total, the state resulting from operation 2 has:

$$0 + 1 + 1 = 2 \text{ degrees of freedom} \tag{6.15}$$

The state resulting from operation 2 in figure 6.17 is, therefore, more stable than the one resulting from operation 1, as it is has the less degrees of freedom.

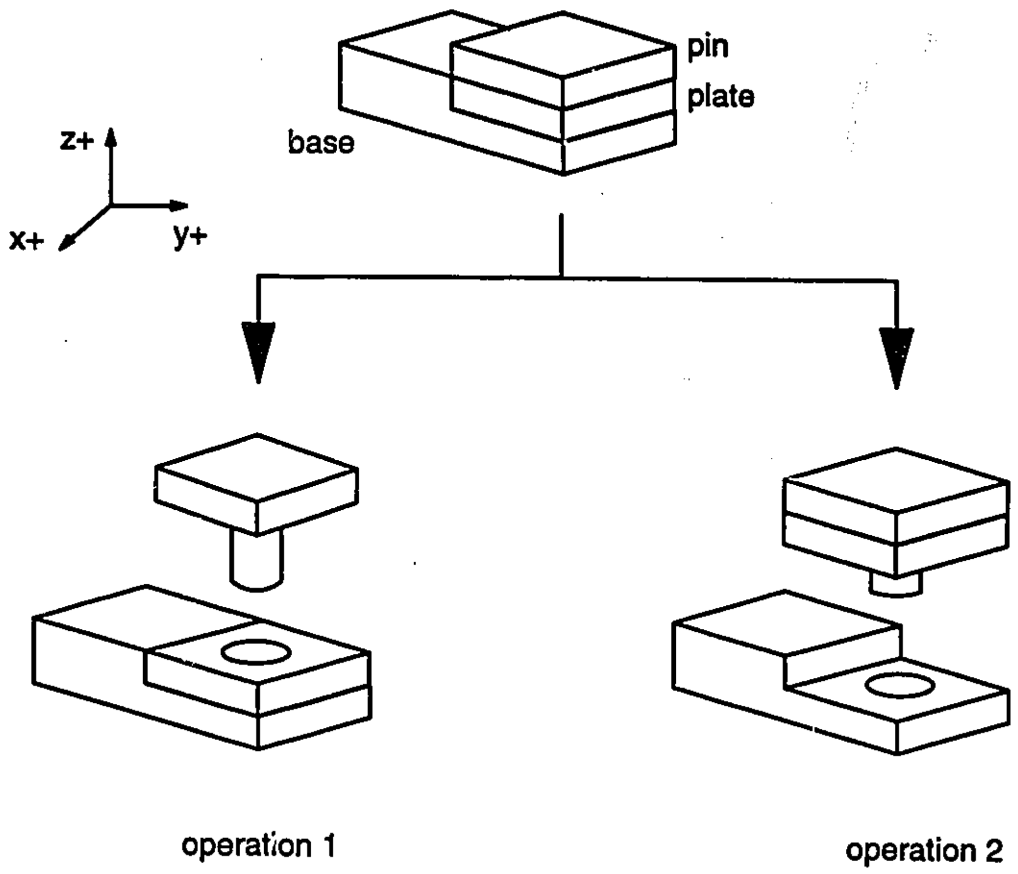


Fig. 6.17: Two possible initial disassembly operations of the three blocks.

6.3.3 Adding penalty degrees of freedom

Although operation 2 in figure 6.17 is the one with less degrees of freedom, it is interesting to note that intuitively one would rather consider the subassemblies resulting from operation 1 to be more stable, even if they present more degrees of freedom. Such a conclusion is directly influenced by the fact that the loose plate, part of the moved subassembly in operation 2, is likely to fall during the execution of this operation. If the plate was attached (for example press fitted) to the pin, then its chances of falling during the execution of operation 2 would vanish.

Apart from a quantitative examination of the degrees of freedom of each component in a state, it seems important to also include as well a qualitative examination under three aspects:

1) what are the particular degrees of freedom of a component,

2) which of the moved or fixed subassembly is the component part of, and

3) whether the component is attached or not to some other component(s) in the state.

Including these further notions helps make the difference between an unattached component, part of the moved subassembly with a z– degree of freedom, and another attached component part of the fixed subassembly with an x+ degree of freedom. Although quantitatively both components have only one degree of freedom, the former is clearly much more unstable than the latter.

In GAPP, these notions are modelled by simply adding penalty degrees of freedom to the components in a state. Once the common disassembly directions have been identified in the set D, two further cases are provided which result in penalizing the components satisfying these two cases:

Case 3: if D is not empty, and it contains "z–" as one of its elements, and component "c" is not attached to any other component in this state, and component "c" is part of the fixed subassembly of this state, then give "c" the maximum degrees of freedom, i.e. 6, to reflect that it is highly unstable in the fixed subassembly in this new state.

Case 4: If D is not empty, and it contains "z–" as one of its elements, and component "c" is not attached to any other component in this state, and component "c" is part of the moved subassembly of this state, then give "c" twice as many degrees of freedom as the maximum possible case, i.e. 12, to indicate that it is highly unstable in the moved subassembly in this new state.

In the state resulting from operation 1 (figure 6.17), the pin has 0 degree of freedom because it is not connected. The plate has 4 degrees of freedom due to its contact with the base. However the base is part of the fixed subassembly, it has "z–" as one of its degrees of freedom in this state and it is not attached to any other component. Therefore case 3 above applies and the base is given 6 degrees of freedom. In total, the state resulting from operation 1 now has:

$$0 + 4 + 6 = 10 \text{ degrees of freedom} \tag{6.16}$$

In the state resulting from operation 2, the base has 0 degree of freedom because it is not connected. The pin has 1 degree of freedom as a result of its connection with the plate. However, the plate is part of the moved subassembly, it has "z–" as one of its degrees of freedom in this state and it is not attached to any other component. Therefore case 4 above applies and the plate is given 12 degrees of freedom. In total, the state resulting from operation 2 now has:

$$0 + 1 + 12 = 13 \text{ degrees of freedom} \tag{6.17}$$

Using these new heuristics, the state resulting from operation 1 in figure 6.17 is now considered more stable than the one resulting from operation 2.

Using the same technique, it can be verified that the number of degrees of freedom in the state resulting from operations 1, 2 and 3 for the flashlight in figure 6.16 are 8, 18 and 16, respectively. Therefore operation 1 in figure 6.16 is the one which results in the most stable state.

6.3.4 Search cost associated with stability

The cost associated with stability can now be defined as the linear function:

$$st = \frac{w_{st} \times dof}{cc \times 3} \tag{6.18}$$

where "w_{st}" is the relative weight of the stability criterion as specified by the user, "dof" is the number of degrees of freedom in the new state resulting from the execution of the disassembly operation, "cc" is the number of connected components in the new state, and "3" represents an average value of the number of degrees of freedom of a component in a state. The denominator "cc x 3" then represents the reasonable estimation that in the worst case, every connected component in a state would have 3 degrees of freedom.

For example, if the user specifies a value of 25 (on a scale of 0 to 100) for "w_{st}", the first and second disassembly operations in figure 6.17 result in unit cost values of 41.66 and 54.16, respectively.

If the stability criterion is turned on, meaning that the user has not specified a value of "0" as its relative weight "w_{st}", this cost is computed and added to the new child state resulting from a disassembly operation in the directed graph of assembly states.

6.3.5 Underestimating stability's remaining search cost

Stability is another criterion for which a good underestimation is hard to obtain from a simple analysis of whatever data is available in the new child state. The technique of "looking ahead" of the new child described earlier for the parallelism criterion was found to be a good way for obtaining a reasonably good underestimation for this criterion.

Given a new child state, an underestimation "u_st" of the cost associated with stability in the best path from this child to the goal is obtained by first generating temporarily all the feasible children states of this new child. The cost associated with stability for each disassembly operation resulting in each temporary child state is then computed using eq. (6.18) and the minimum value for all children is kept as an underestimation for the original child. That is:

$$u\_st = min\left(\frac{w_{st} \times dof}{cc \times 3}\right) \quad for\ all\ tc_i \qquad (5.19)$$

where "tc_i" is the ith temporary child of the new child for which an underestimation is required.

If the new child for which an underestimation is required is the goal state, no temporary child is ever generated and the value returned by eq. (6.19) is "0", which is a perfect estimation. If there is only one remaining disassembly operation to transform the new child into the goal state, this goal state is generated as the only temporary child and a perfect estimation of "0" is once again obtained from eq. (6.19)[46].

46. Recall that, by convention, the goal state consisting of all disassembled components has zero degree of freedom.

If there are "n" remaining disassembly operations to transform the new child into the goal state, an arbitrary number of temporary child states of this new child can be generated. The cost of the best path from the new child to the goal consists of the sum of the costs of each disassembly operations in this path. Eq. (6.19) returns the lowest of all cost values computed for every disassembly operation resulting in every temporary child. This minimum value is the cost of the first disassembly operation in the best path from the new child to the goal. As any other remaining disassembly operations in this best path either adds to the cost or has a cost of "0" if it is the last operation, eq. (6.19) is guaranteed to be an underestimation of the cost of the best path from the new child to the goal.

6.3.6 Stability: criterion versus constraint

Homem De Mello and Sanderson [25] suggested that a disassembly operation resulting in an unstable configuration should be considered unfeasible[47]. This implies that stability could become one of the constraints introduced earlier in chapter 5.

In terms of the directed graph of assembly states, the important distinction between considering stability as a constraint or a criterion is that in the former case, a node in which unstable subassemblies exist would never be generated, while in the latter case such a node could be generated but be associated with a high cost with respect to the stability criterion.

GAPP's philosophy of considering stability as a criterion instead of a constraint is justified by three good reasons. First, any unstable configuration can always be made stable by using proper clamping or fixturing. Although this may

47. The authors define an unstable configuration as one in which the parts do not maintain their relative position and break contact spontaneously.

increase tooling costs, this situation can still be very justifiable, specially if it results in a significant reduction in cycle time or an important increase in productivity. Second, there may exist some products which cannot be assembled without running into unstable configurations. The candelabra in Miller and Hoffman [57] is an example of such a product. Third, although nodes with unstable configurations can be generated, providing a sufficiently high relative weight to the stability criterion can prevent any such node from being selected in the disassembly sequence returned by GAPP, which is practically equivalent to not generating these nodes at all.

6.4 CLUSTERING OF SIMILAR OPERATIONS

6.4.1 Benefits of clustering

The last criterion used in GAPP relates to the clustering of similar disassembly operations into successive operations in the final assembly sequence. The underlying motivation for implementing this criterion lies in the benefits that such a clustering can provide.

Take for example the simple product in figure 6.18. The disassembly of this product simply consists of removing both screws and the block from the plate along the z+ direction. Altogether, there are six possible permutations in the order of execution of these three disassembly operations. They are:

1– remove screw1, screw2, block

2– remove screw1, block, screw2

3– remove screw2, screw1, block

4– remove screw2, block, screw1

5– remove block, screw1, screw2

6– remove block, screw2, screw1

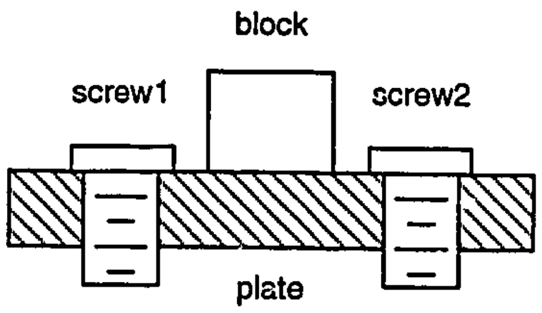


Fig. 6.18: A simple product to be disassembled.

Whether a robot or human performs the disassembly of this product, it is obvious that the tool to be used to remove both screws is different from that used to remove the block. It can be readily verified that disassembly sequences 1, 3, 5 and 6 above require a single tool change and that disassembly sequences 2 and 4 above require two tool changes. The selection of either one of disassembly sequences 1, 3, 5 or 6 would be most desirable as they minimize the number of tool changes. Note that these 4 sequences are the ones in which the maximum number of similar operations could be performed consecutively, i.e. two consecutive unscrewing operations.

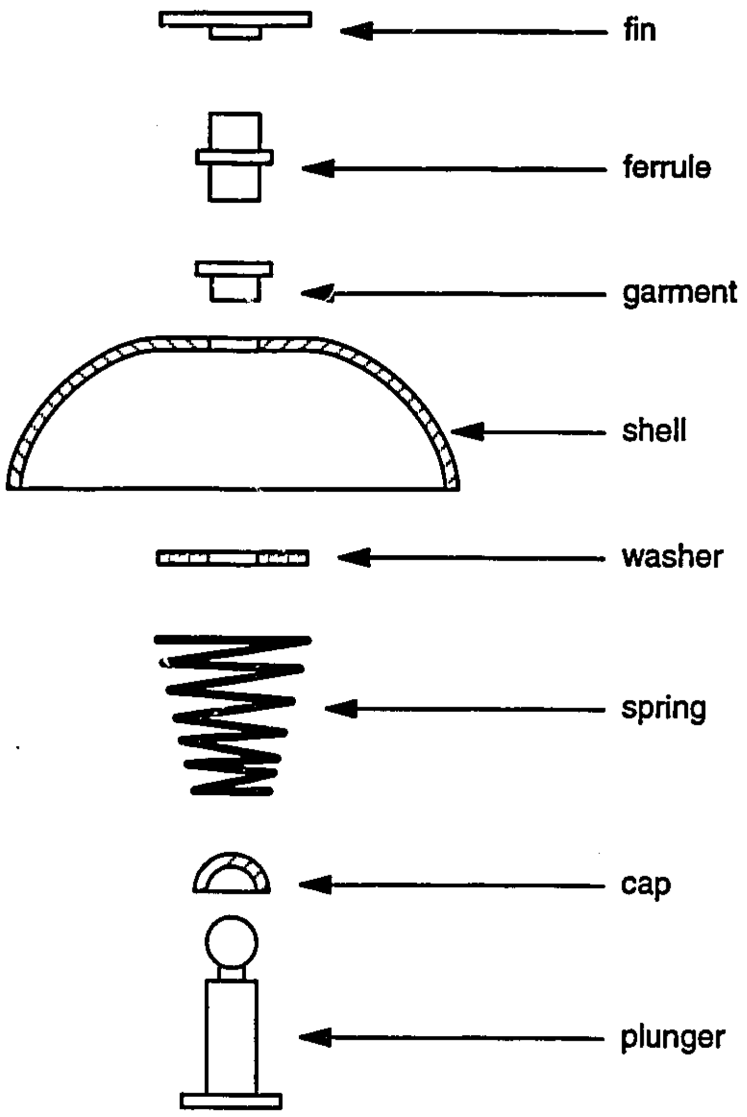


Fig. 6.19: Shell subassembly of a heat detector device.

Consider now the shell subassembly of a heat detector device shown in figure 6.19. Knowing that the fin is pressed to the ferrule, that the ferrule is pressed into the garment, and that the garment is pressed into the shell, the clustering of these similar operations into successive operations in the assembly sequence may not only reduce the number of tool changes for the resources which execute the corresponding assembly operations, but may also suggest the merging of these similar operations into a single operation using a single resource. As a matter of fact, 2 of these 3 operations are known to be executed simultaneously in a single pressing operation by the company producing the heat detectors to which these shell subassemblies belong.

Clearly then, important benefits can be obtained from the clustering of similar operations into successive ones in the assembly sequence. The next section describes how similar disassembly operations generated by GAPP can be recognized.

6.4.2 Identifying similar operations in GAPP

The identification of similar disassembly operations in the directed graph of assembly states first requires the labelling of the cutsets with particular operation names. Checking the similarity among disassembly operations in this graph then becomes the simple problem of checking whether their associated cutsets have similar operation names.

For a cutset with a single relation, the operation name for the cutset is simply the one returned by the *operation()* function applied to its relation. For example, the operation name of cutset R3 in figure 6.9 is "fit". However, when a cutset consists of more than one relation, the operation specified for each relation might be different. For example, the cutset shown earlier in figure 6.5 consists of 2 relations. The operations associated with relations R1 and R2 in this cutset are screwing and fitting, respectively. The problem arises as to which name should be specified for the operation associated with this cutset.

GAPP solves this problem as follows: the operation specified for each relation in the product's graph model is given some weight corresponding to the strength of the connection resulting from the execution of this operation. The stronger the connection, the larger the weight. For example, a relation between two components which is established by performing a screwing operation is stronger than another relation established by simply placing the two involved components against each other. The weight associated with the former operation is larger than the one associated with the latter.

From this, the disassembly operation associated with a cutset becomes the operation of the relation with largest weight among all relations in this cutset. The screwing operation associated with relation R1 is stronger than the fitting operation associated with relation R2 in the example in figure 6.5. The former operation is given a larger weight than the latter and the operation name associated with the two relations in this cutset is "screw".

This simple technique for associating a single disassembly operation among all possible operations specified for the relations in a cutset can be generalized as follows:

$$operation(cutset) = \max\ operation(r_i) \quad \text{for } all\ r_i\ in\ cutset \qquad (6.20)$$

where "operation()" is the *operation()* function described earlier in chapter 3 and r_i is the i^{th} relation in the cutset.

Once all cutsets have been given an operation name in this way, the identification of similar operations in the directed graph of assembly states consists of matching similar names for the cutsets associated with the operations in this graph.

The cutset associated with the disassembly operation which transforms some parent node into some child node will be referred to as the *generative* cutset of the child node in the section that follows.

6.4.3 Search cost associated with clustering

Given the operation name of the generative cutset of some parent node and given the operation name of the generative cutset of one of its child nodes, the cost "cl" associated with clustering can now be defined as the constant function:

$$cl = \begin{cases} 0 & \text{if } operation(child) = operation(parent) \\ w_{cl} & \text{if } operation(child) \neq operation(parent) \end{cases} \tag{6.21}$$

where "w_{cl}" is the relative weight of the clustering criterion as specified by the user, "operation(child)" is the name of the operation of the child's generative cutset and "operation(parent)" is the name of the operation of the parent's generative cutset.

For example, if the user specifies a weight of 25 (on a scale of 0 to 100) for this criterion, a cost of 25 units will be added to every new child state which generative cutset operation name is different than that of its parent. By convention, the cost associated with this criterion for every child of the root node in the directed graph of assembly states is assumed to be zero. In other words, the value of *operation(parent)* where *parent* is the root node is always assumed to be equal to *operation(child)* where *child* is one of the root node's children, resulting in a cost of zero in eq. (6.21).

6.4.4 Effects on assembly states

An important assumption for a heuristic function that computes the cost of a criterion in a state of the directed graph of assembly states is called *history–independence* [84]. In essence, this assumption specifies that the computation of the cost associated with a criterion depends only on the local information available in the new state for which this cost is being computed. Without this important assumption, the directed graph of assembly states is not guaranteed to be a suitable representation for sets of optimal assembly sequences.

Unlike any other criteria discussed so far, the computation of the cost associated with clustering for a new child node, as given by eq. (6.21), violates the history–independence assumption. That is, the computation of this cost also requires the knowledge of the operation that was applied in the generation of its parent, i.e. the computation does not depend only on the local information available in the new state which cost is being computed.

The next chapter will present a method for keeping track of the best path to a node. This will be an appropriate time for presenting a method for ensuring that optimal paths can be represented in the directed graph of assembly states even when the history–independence assumption is violated.

6.4.5 Underestimating clustering's remaining search cost

The technique of "looking ahead" of the new child described earlier for the parallelism and stability criteria is used again here for obtaining a reasonably good underestimation for the clustering criterion.

Given a new child state, an underestimation "u_cl" of the cost associated with clustering in the best path from this child to the goal is obtained by first generating temporarily all the feasible children states of this new child. The cost associated with clustering for each disassembly operation resulting in each temporary child state is then computed using eq. (6.21) and the minimum value for all children is kept as an underestimation for the original child. That is:

$$u\_cl = \min\ cl \qquad for\ all\ tc_i \tag{6.22}$$

where "tc_i" is the ith temporary child of the new child for which an underestimation is required.

If the new child for which an underestimation is required is actually the goal state, no temporary child is ever generated and the value returned by eq. (6.22) is "0", which is a perfect estimation. If there is only one remaining disassembly operation to transform the new child into the goal state, this goal state is generated as the only temporary child and a perfect estimation is once again obtained from eq. (6.22).

If there are "n" remaining disassembly operations to transform the new child for which an underestimation is required into the goal state, an arbitrary number of temporary child states of this new child can be generated. The cost of the best path from the new child to the goal consists of the sum of the costs of each disassembly operations in this path. Eq. (6.22) returns the lowest of all cost values computed for every disassembly operation resulting in every temporary child. This minimum value is the cost of the first disassembly operation in the best path from the new child to the goal. As any other remaining disassembly operations in this best path either adds to the cost or has a cost of "0", eq. (6.22) is guaranteed to be an underestimation of the cost of the best path from the new child to the goal.

6.5 COMPETITION AMONG CRITERIA

Given a disassembly operation in the directed graph of assembly states, and given eqs (6.1), (6.11), (6.18) and (6.21) to compute the goodness of this operation with respect to the four corresponding criteria, this operation is likely to be good with respect to some criteria and not as good with respect to others. An obvious example of this is disassembly operation 1 for the flashlight in figure 6.16: although this operation is good with respect to stability, it is not so good with respect to parallelism. On the other hand, operation 2 in the same figure is very bad with respect to stability but is best with respect to parallelism. Operation 3 is still very bad with respect to stability but is better than operation 1 with respect to parallelism.

Clearly, in trying to determine which of these three operations is the best with respect to both criteria, the relative importance among these criteria plays an important role. For example, if one considers stability to be more important than parallelism, operation 1 will be selected. For the reverse case where parallelism is considered to be more important, operation 2 would now be selected as the best one. An overall measure is the summation of all four defined criteria.

In GAPP, such conflicts are resolved by providing some weight to the various criteria used in the evaluation process. These weights are supplied interactively at the click of the mouse by the user prior to the disassembly sequence generation process. Any criterion whose relative weight has been set to "0' is ignored by GAPP in the computations. The higher the weight of a criterion, the more influence it has in deciding which disassembly operation is the best among different possible ones.

As an illustrative example, assume that the relative weight of the re–orientation and clustering criteria have been set to the value "0", and that a weight of 25 and 50 has been set for the stability and parallelism criteria, respectively. With this particular set up, it can be verified that the cost of the three operations in figure 6.16 will be as in table 6.1. Note that because operation 2 is the best one with respect to parallelism, which is the criterion with the largest weight, this operation becomes the one with the lowest total cost, even if this operation is the most unstable of the three. Similarly, by switching the weight values for both criteria, stability becomes predominant and operation 1 now has the lowest total cost, as shown in table 6.2.

Table 6.1
Search cost of the three operations in figure 6.16 for relative weights of 25 and 50 for the stability and parallelism criteria, respectively.

Operation #	Stability search cost (eq. 6.18)	Parallelism search cost (eq. 6.11)	Total search cost
1	7.40	35.71	43.11
2	16.66	0	16.66
3	14.81	21.4	36.21

Table 6.2
Search cost of the three operations in figure 6.16 for relative weights of 50 and 25 for the stability and parallelism criteria, respectively.

Operation #	Stability search cost (eq. 6.18)	Parallelism search cost (eq. 6.11)	Total search cost
1	14.81	17.85	32.66
2	33.32	0	33.32
3	29.62	10.7	40.32

It will be shown in chapter 8 how a variation of the relative weight of the various assembly planning criteria causes different optimal assembly sequences to be generated by GAPP.

CHAPTER 7

SEARCH METHODS

This chapter describes the various methods by which GAPP incrementally constructs the directed graph of assembly states and generates an optimal disassembly sequence, which when reversed gives an optimal assembly sequence. The first section provides some definitions. The second section describes how to keep track of the best path to a node. The third section describes the effects of violating the history independence assumption. Section four describes two exhaustive search methods, namely breadth first and depth first. The fourth, fifth and sixth sections describe the best first, A^* and hill climbing search methods, respectively. These 5 search methods are all available in GAPP.

7.1 DEFINITIONS

The process of generating the directed graph of assembly states will now be referred to as the expansion of a **search graph.**

A node of the search graph has been **expanded** if all its children nodes resulting from the feasible disassembly operations corresponding to some of its cutsets have been generated.

The set of all nodes of the search graph which have been generated but not yet expanded is called the **open set.** Initially, only the root node corresponding to the product's graph model is on the open set.

The set of all nodes of the search graph which have been generated and expanded is called the **closed set.** If the whole search graph gets expanded in the search process, no more nodes remain in the open set and all nodes are included in the closed set.

Assuming that the root node of the search graph has already been placed in the open set, the **basic search procedure** used by GAPP is as in figure 7.1.

```
1   while the open set is not empty
2       pick one node from the open set
3       remove this node from the open set and add it to the closed set
4       find all feasible disassembly operations that can be applied to the
        subassemblies in this node by analyzing its cutsets
5       for each feasible disassembly operation:
            generate a corresponding new node of the search graph
            if the new node already exists on the open set:
                delete it
                make parent point to existing node
            if the new node already exists on the closed set:
                delete it
                make parent point to existing node
            else
                add the new node to the open set
                make parent point to it
```

Fig. 7.1: GAPP's basic search procedure.

Variations of the above basic search procedure define the 5 search methods that have been implemented in GAPP. A discussion of the use of particular search methods is presented in chapter 8.

7.2 KEEPING TRACK OF THE BEST PATH TO A NODE

7.2.1 Cost function

The introduction of the various criteria in chapter 6 gives GAPP the ability to evaluate the many different disassembly sequences in the directed graph of assembly states. Integrating the evaluation of these criteria into the search procedure as the search graph gets expanded, enables the direct generation of an optimal disassembly sequence of a given product with respect to these criteria. Depending on the search method used, the optimal solution can even be obtained without exhaustive generation of the whole search graph.

To achieve this type of functionality, each node of the search graph is associated with a pointer to its best known parent. Since every node has only one such parent, the backward path through the best parent links between any two nodes in the search graph represents the best path between these two nodes. In particular, the backward path from the goal node to the root node represents the optimal assembly sequence of the product.

Inherent in the concept of the best path to a node is the measure of a path's goodness. In GAPP, this is accomplished by associating a cost to each new node that gets generated in the search graph. This cost is denoted by "g" and is computed using the recurrence formula:

$$g = g(parent) + cg \qquad (7.1)$$

where "g(parent)" is the cost of the best path from the root node of the search graph to the parent of a new node, and "cg" is the cost of the disassembly operation transforming the parent of the new node into the new node itself.

The value of "cg" is in turn obtained from the functions which compute the values of the various criteria:

$$cg = re + pa + st + cl \tag{7.2}$$

where the values for "re", "pa", "st" and "cl" are obtained from eqs (6.1), (6.11), (6.18) and (6.21), respectively. Figure 7.2 illustrates the cumulative nature of the cost function given by eq. (7.1) for an arbitrary path.

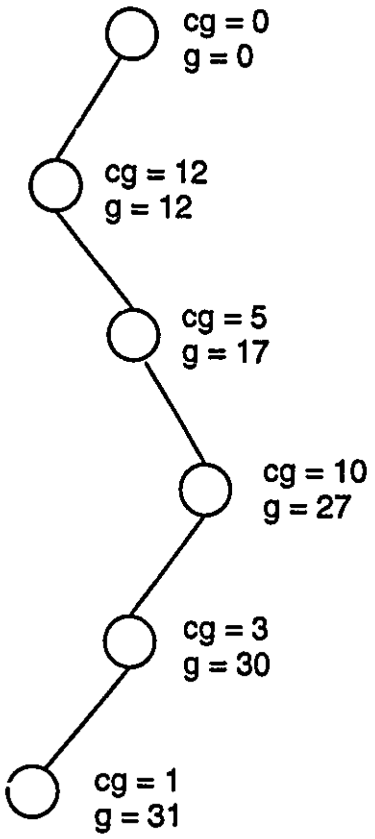


Fig. 7.2: Accumulation of cost to the nodes in an arbitrary path.

7.2.2 Resetting the best parent of a node in the open set

To illustrate how the cost function is used to keep track of the best path to a node, consider figure 7.3. It shows the state of the search graph for a hypothetical 4–components product after a few iterations of the basic search procedure outlined in figure 7.1. The bold edges in figure 7.3 correspond to a best known parent link. All other edges correspond to the usual parent–child link. Bold nodes correspond to nodes in the closed set. All other nodes are in the open set[48].

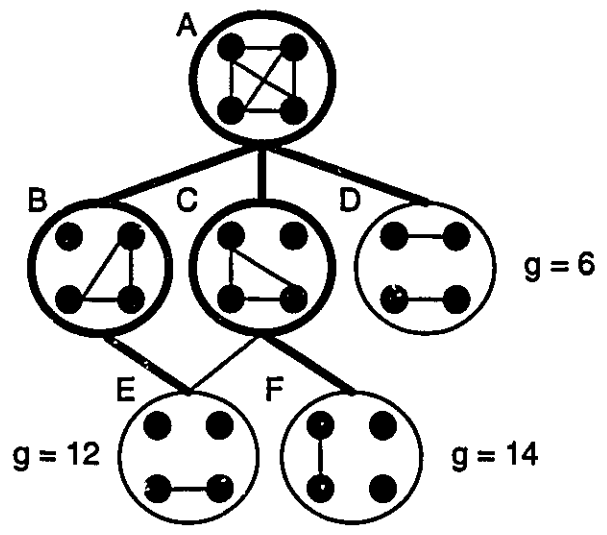


Fig. 7.3: State of the search graph of a 4–components product after a few iterations of the search procedure in figure 7.1.

First note that every node has a single pointer to its best known parent. For example, out of the two parents of E, only B is identified as its best known parent. Working our way through the best parent links, it is very easy to identify that, out of

48. These conventions will be used through the remaining of this chapter.

the two possible paths to E, the path {A, B, E} is actually the best one generated so far. Similarly, the path {A, C, F} is the best known path to F.

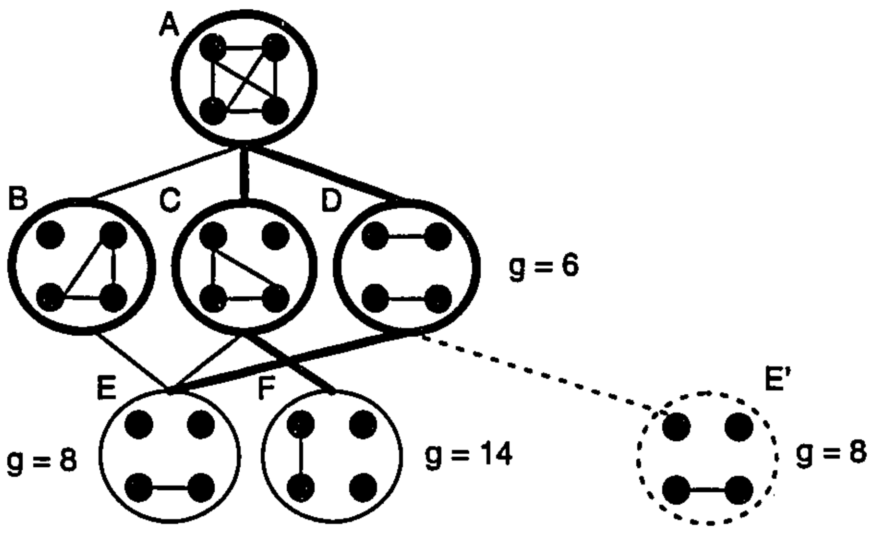


Fig. 7.4: Resetting the best parent of node E.

Assume now that in the next iteration of the basic search procedure, node D is selected from the open set and expanded to give node E', as depicted in figure 7.4. Note that this new node is the same as E already in the open set. Therefore E' is deleted and D is set to point to E instead. Because the cost of this new path to E is lower than the path {A, B, E} in figure 7.3, the pointer to the best parent of E is reset to D and the best path to E is now {A, D, E}.

7.2.3 Resetting the best parent of a node in the closed set

Consider now figure 7.5. It shows the state of the search graph for the same 4–components product after a few more iterations of the basic search procedure.

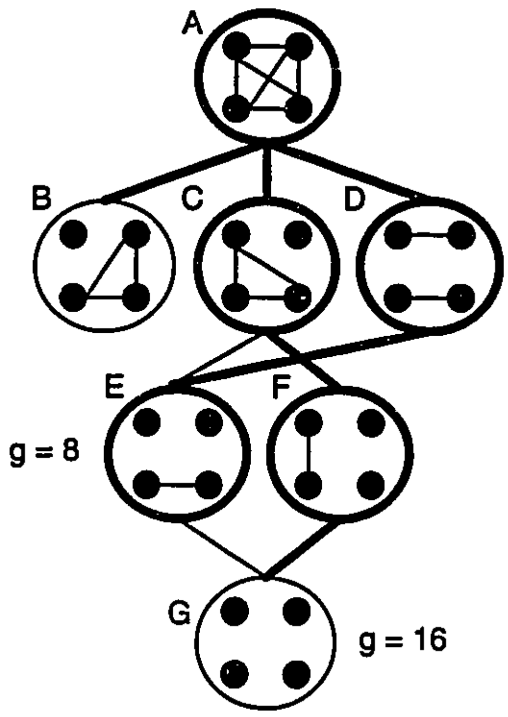


Fig. 7.5: State of the search graph of the 4–components product after a few more iterations of the search procedure.

Assume that node B is selected for expansion in the next iteration to yield node E', as depicted in figure 7.6. Once again, node E' is the same as node E, so E' is deleted and B is set to point to E instead. The resulting new cost to node E through path {A, B, E} is now lower than the earlier one through path {A, D, E}. Therefore the best parent of E is reset to B. However this time node E is in the closed set. Due to the cumulative nature of the cost function, this implies that the cost of the best path to its child G has also been improved[49]. Therefore this new lower cost must be propagated to G[50]. In performing these computations, the new cost to G

49. If G also had children, their cost would also be improved, as would the cost of their children, of the children of their children, and so on.

50. In GAPP, when the best path to a node "n" on the closed set has been reset, the new improved cost is propagated in a depth first fashion starting at "n" through the whole state of the search graph.

is also lower than the earlier one through path {A, C, F, G}. The best known parent of G is then reset from F to E, yielding the new best path {A, B, E, G}.

Although this kind of housekeeping could be avoided by expanding a search tree instead of a search graph, the important reduction in memory consumption that this approach provides greatly justifies its usage. This will be discussed in more detail in chapter 9.

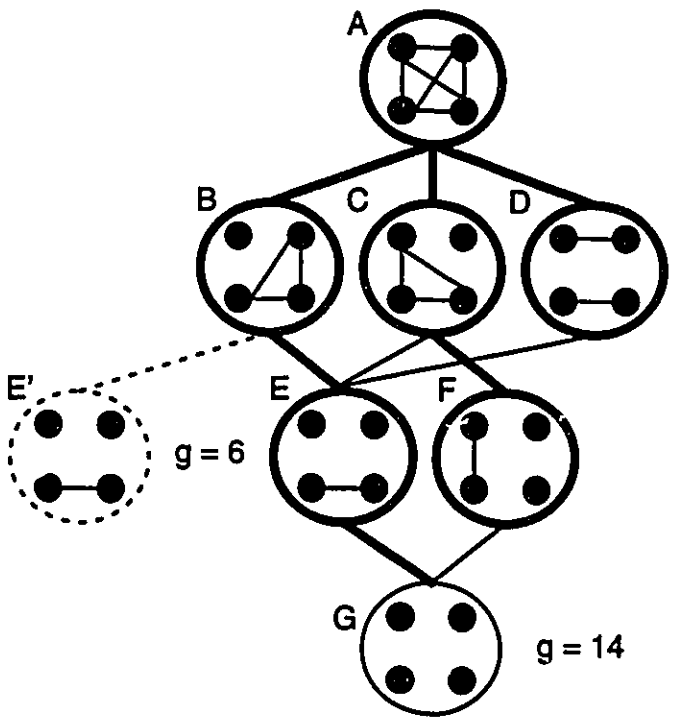


Fig. 7.6: Resetting the best parents of nodes E and G.

7.3 VIOLATING THE HISTORY INDEPENDENCE ASSUMPTION

The repetition of the above process until the goal node gets generated ensures that the optimal path to the goal is always known, i.e. the optimal assembly sequence has been generated. Nevertheless, a complication may arise when the history independence is violated in the computation of the cost associated with a criterion such as clustering. In particular, the optimal path of the search graph might be missed. This is demonstrated in the next two sections.

7.3.1 Failure to reset the best parent pointer

Figure 7.7 shows an instance during the generation of the search graph for a hypothetical 4–components product. Each edge in the graph is labelled by an arbitrary name corresponding to some disassembly operation. A bar above an operation name means that this operation also requires a re–orientation of the parent subassembly before it can be executed[51]. The top and bottom values beside each node represent the values of "cg" and "g" in eq. (7.1), respectively. These values are obtained for the case where relative weights of 75 and 50 have been specified for the re–orientation and clustering criteria, respectively. It is assumed that parallelism and stability have been turned off.

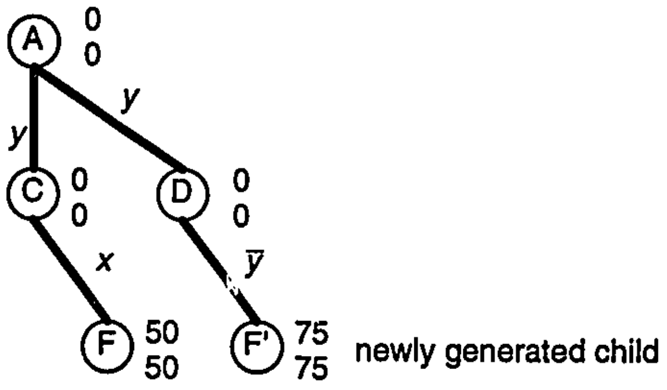


Fig. 7.7: An instance in the search graph: node F' has just been generated.

51. A parent subassembly was defined at the end of section 6.1.2.

In order to decide whether new child node F' is equal to already existing node F, the subassemblies in these two states as well as their orientation must be compared. Assuming that nodes F and F' indeed represent the same subassemblies in the same orientation, then these two nodes are the same. Therefore node F' can be deleted and node D can point to already existing node F. One must also determine if the resulting new path to F is better than the current one by comparing the cost values. As the cost to F' is larger than the best one to F in figure 7.7, the best parent pointer of F is not reset. This is illustrated in figure 7.8.

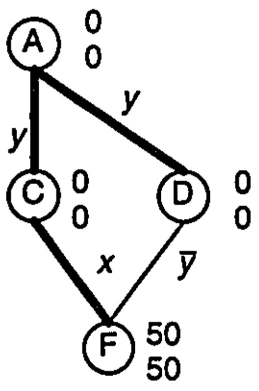


Fig. 7.8: Resetting node D to point to node F without changing the best path to node F.

Assuming that node F gets expanded in the next iteration to yield goal node G, the best path from the root to the goal which passes through F has a cost of 100. This is depicted in figure 7.9.

Consider now the path from the root to the goal in figure 7.10. This new path results from a change of the best parent of node F from node C to node D. Note that the total cost to the goal in now smaller than that in figure 7.9.

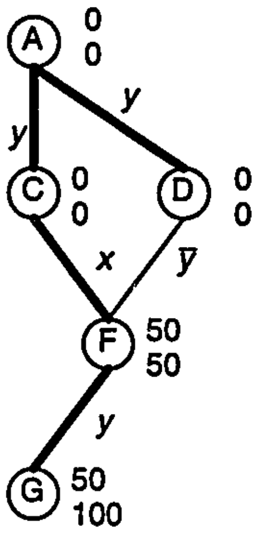


Fig. 7.9: Expanding node F to yield goal node G.

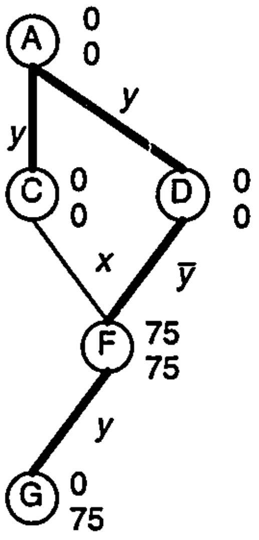


Fig. 7.10: Changing the best path to node F and then expanding it.

What has just been shown by this simple example is that the true optimal path of the search graph might be missed by a failure to reset the best parent pointer of some node. This may happen when some criteria which violate the history–independence assumption are used in the computation of the cost function.

7.3.2 Mistakenly resetting the best parent pointer

Figure 7.11 shows an instance in the generation of the search graph for another hypothetical 4–components product where the relative weight of the re–orientation and clustering criteria have been interchanged and the operation transforming node F into node G has been changed.

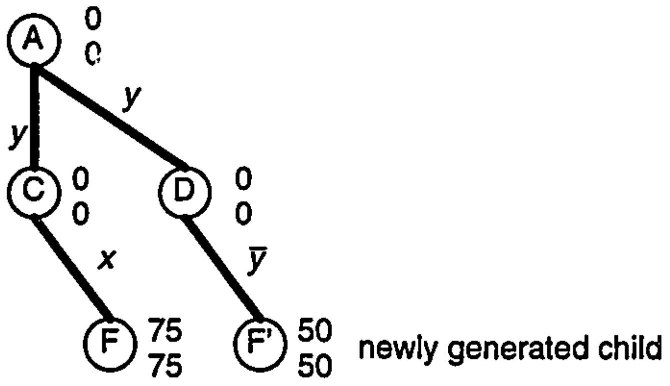


Fig. 7.11: An instance in the search graph: node F' has just been generated.

In this case, assuming that nodes F and F' represent the same subassemblies in the same orientation, the search graph in figure 7.12 is now obtained. Expanding node F to goal node G results in an optimal path with a total cost of 175, as depicted in figure 7.13. But then, in figure 7.14 the optimal path obtained by not resetting the best parent of F has a lower cost of 125.

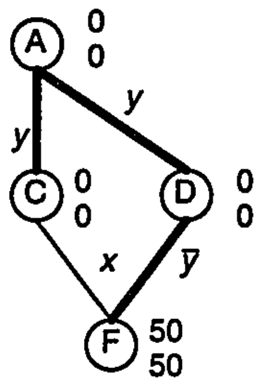


Fig. 7.12: Resetting node D to point to node F and changing the best path to node F.

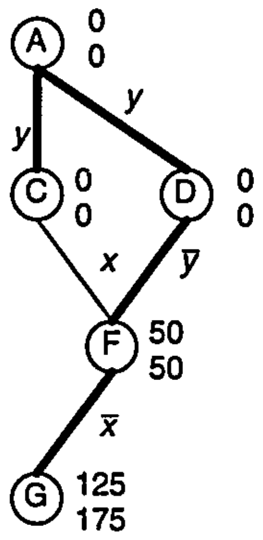


Fig. 7.13: Expanding node F to yield goal node G.

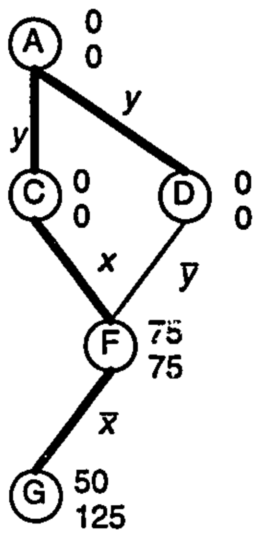


Fig. 7.14: Not changing the best path to node F and then expanding it.

This example shows that the true optimal path of the search graph might be missed by mistakenly resetting the best parent pointer of some node. Once again, this may happen when some criteria which violate the history–independence assumption are used in the computation of the cost function.

To solve this problem, the operation name associated with the generative cutset of a node must also become part of its description. Two nodes are equal only if their generative cutset[52] have the same operation name, apart from having the same subassemblies in the same orientation. Using these concepts, the expansion of the search graphs for the examples in sections 7.3.1 and 7.3.2 results in the new search graphs in figures 7.15 and 7.16, respectively. Note that both nodes F and F' are now part of the graphs, and that the optimal path from the root to the goal has been correctly recorded in both cases.

52. A generative cutset was defined at the end of section 6.4.2.

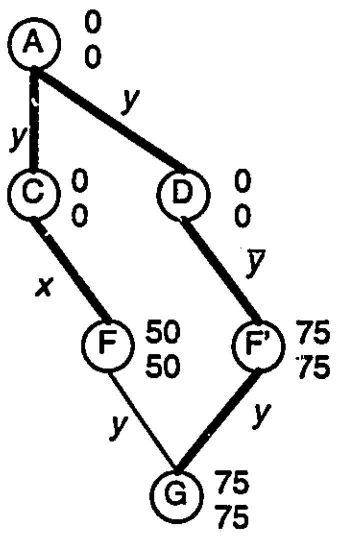


Fig. 7.15: Modified search graph for the example in section 7.3.1.

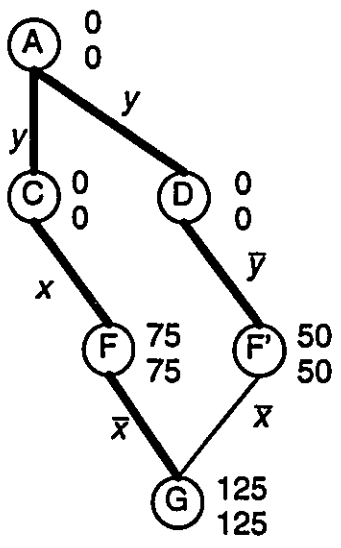


Fig. 7.16: Modified search graph for the example in section 7.3.2.

7.3.3 Multiple goal nodes

In incorporating the operation from which a node was generated into the node's description, an important implication is that the search graph may now present multiple goal nodes. For example, in figure 7.17 nodes G and G' may represent the same subassemblies in the same orientation. But since they were not obtained from the same disassembly operation, they are not equal and cannot be merged into a single node.

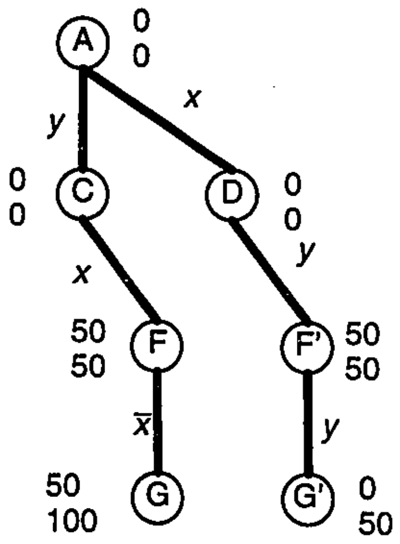


Fig. 7.17: Multiple goal nodes in a search graph.

The best path to every such goal is obtained from the backward traversal of the graph through the best parent pointers starting at each goal node. In the example above, two such best paths are {A, C, F, G} and {A, D, F', G'}. Among all possible best paths, the overall optimal path is simply the one with lowest cost. In the example above, the optimal path is {A, D, F', G'}.

7.4 BREADTH FIRST AND DEPTH FIRST EXHAUSTIVE SEARCH

Given the basic search procedure in figure 7.1 together with the notion of the best path to a node described in the previous sections, two exhaustive methods available in GAPP for generating the optimal assembly sequence of a product can now be presented.

1– If the basic search procedure is applied until the open set is empty, and if the node selected for expansion in step 2 is always the earliest of all the nodes added in the open set, then the whole search graph is generated in a breadth first fashion. The path through the best parent pointers from the best goal node to the root node represents the optimal assembly sequence with respect to the chosen criteria.

Using this search strategy, the order of expansion of the nodes in the search graph of figure 7.18 is {A, B, C, D, E, F, G, H} and the optimal path is {H, G, C, A}.

2– If the basic search procedure is applied until the open set is empty, and if the node selected for expansion in step 2 is always the latest of all the nodes added in the open set, the whole search graph is generated in a depth first fashion. The path through the best parent pointers from the best goal node to the root node represents the optimal assembly sequence with respect to the chosen criteria.

Using this search strategy, the order of expansion of the nodes in the search graph of figure 7.18 is {A, D, F, H, C, G, B, E}. The optimal path is still {H, G, C, A}.

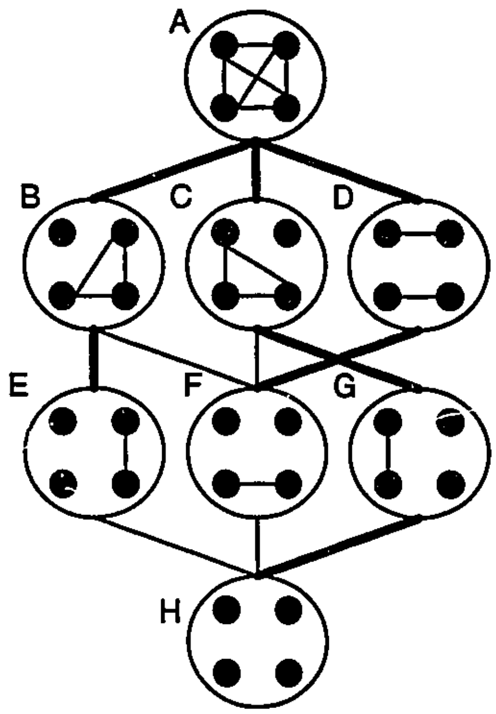


Fig. 7.18: A hypothetical search graph for a 4–components product.

It is interesting to note that the sufficient condition which guarantees the exhaustive generation of the whole search graph is to keep on expanding new nodes until the open set is empty. Expanding the earliest added (breadth first search) or latest added (depth first search) node in this set is just a means of insuring a systematic search. The whole search graph is eventually expanded and the optimal assembly sequence returned even if the selection is not done systematically.

It is also very important to understand at this point that GAPP actually searches through an *implicit search graph*, meaning that the graph is being generated as it is searched. This is in contrast to a search through an explicit search graph, in which case the search is performed through an already existing graph.

7.5 BEST FIRST SEARCH

The breadth first and depth first search strategies guarantee the generation of the optimal assembly sequence of a product. Nevertheless, these methods become inefficient for products with a large number of components, due to the significant combinatorial explosion of the number of nodes in the search graph (section 5.1). The best first search method is a possible way of overcoming this problem.

Unlike exhaustive search methods, which are characterized by a systematic selection of either the earliest or latest node added to the open set until this set becomes empty, the best first search method is first characterized by its ability to select the best node on the open set, i.e. the one with the lowest value of the cost function given by eq. (7.1), and by its ability to stop the search as soon as the goal node is selected from this set. This implies that some nodes might remain in the open set at the end of the search process.

If a path of the search graph is substantially better than all others, this method finds this path directly. This is illustrated in figure 7.19. In a first iteration of the search procedure, node A gets expanded, yielding nodes B, C and D. In the second iteration, node B, with the lowest cost, is expanded to yield node E. The low cost of node E leads to its expansion in the third iteration to yield node F, then node F in the fourth iteration, then node G in the last iteration. Clearly, because one path is substantially better than all others, this path gets explored further and further and the optimal solution is found directly.

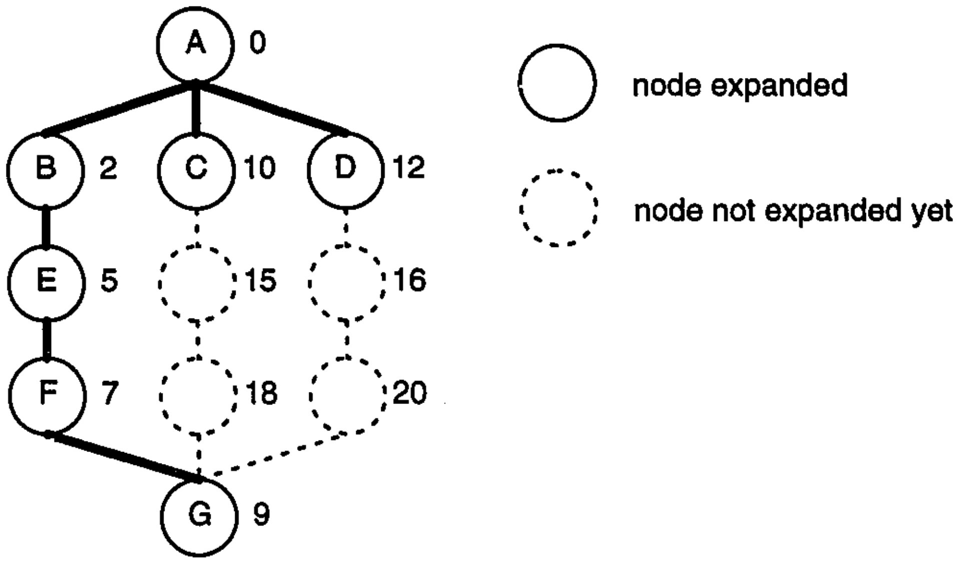


Fig. 7.19: A search graph with a solution path substantially better than all others. The number beside a node is the cost of the best path to this node.

If the difference between the cost of every operation in the search graph is small, then the best first search method approaches an exhaustive search. This is illustrated in figure 7.20. In this case, a first iteration expands node A and a second one node B. In the third iteration, node C is expanded instead of E. Similarly, in the fourth iteration node D is expanded instead of either E or F, and so on. Because of of the similar cost value of each operation, expanding the search graph using this search method does not provide a substantial improvement compared to exhaustive methods. However, the optimal path is still selected in the end.

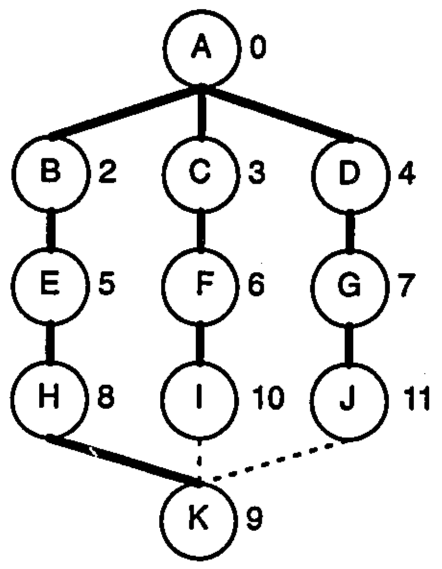


Fig. 7.20: A search graph in which no solution path is substantially better than all others.

7.6 A* SEARCH

The problem with the best first search strategy is that, since the cost function is cumulative, the nodes closer to the goal generally have a higher cost value than those farther from it. This leads to a selection of the farther nodes to be expanded in the next iteration, which translates to a search pattern close to that of exhaustive search.

The A* algorithm [59] [66] is the tool used by GAPP to generate optimal assembly sequences more efficiently. The main difference between the best first search strategy and the A* search is that the cost associated with a node also includes an estimate of the cost of the best possible path from this node to the goal. As a result, the nodes closer to the goal need not have a higher cost value than those farther from it, which leads to a more direct generation of the best path in the search graph.

In the A* formalism, the cost associated with a new child node is given by:

$$f = g + uh \tag{7.3}$$

where "f" is the estimated cost of the best path from the root node to the goal node which passes through the new child node, "g" is the cost of the best path from the root node to the new child node as given by eq. (7.1), and "uh" is the underestimated cost of the best path from the new child node to the goal as given by:

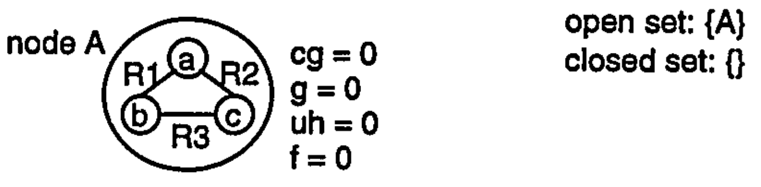
$$uh = u\_re + u\_pa + u\_st + u\_cl \tag{7.4}$$

where the values for "u_re", "u_pa", "u_st" and "u_cl" are obtained from eqs (6.5), (6.12), (6.19) and (6.22), respectively.

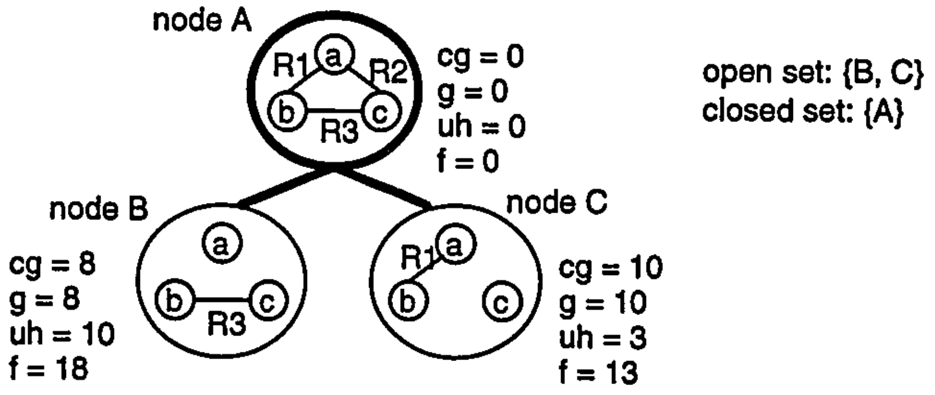
As mentioned earlier, the A* algorithm is guaranteed to find the optimal path of the search graph, provided that the value of "uh" is an underestimation of the cost of the best path from the new child node in the search graph to the goal node. This condition was shown to hold true in the computation of "u_re", "u_pa", "u_st" and "u_cl" in chapter 6. In eq. (7.4) above, the value of "uh" is the simple addition of these 4 parameter values. It follows that the A* algorithm is guaranteed to find the optimal assembly sequence in the directed graph of assembly states using the values of "uh" as defined above.

Figure 7.21 will be used to show how an A* search works. The figure consists of a series of snapshots, each corresponding to the state of the search graph after each iteration of the A* algorithm, which is a variant of the basic search procedure in figure 7.1.

A hypothetical 3–components product is used in this example. The cost of the operation which transforms a parent node into a new child node, "cg", the cost of the best path from the root node to a new child node, "g", the underestimated cost of the best path from a new child node to the goal node, "uh", as well the cost estimate of the best path from the root node to the goal node which passes through a new child node, "f", are specified beside each node.

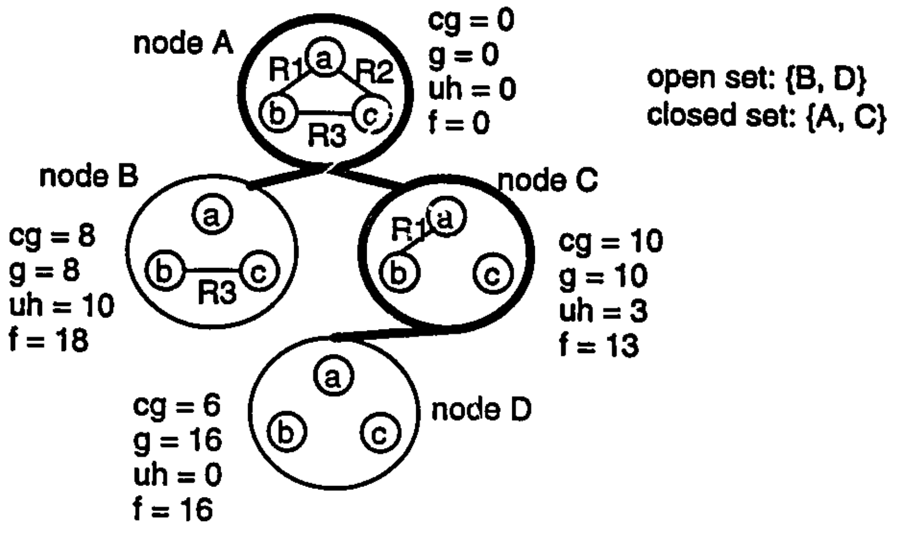


Before first iteration

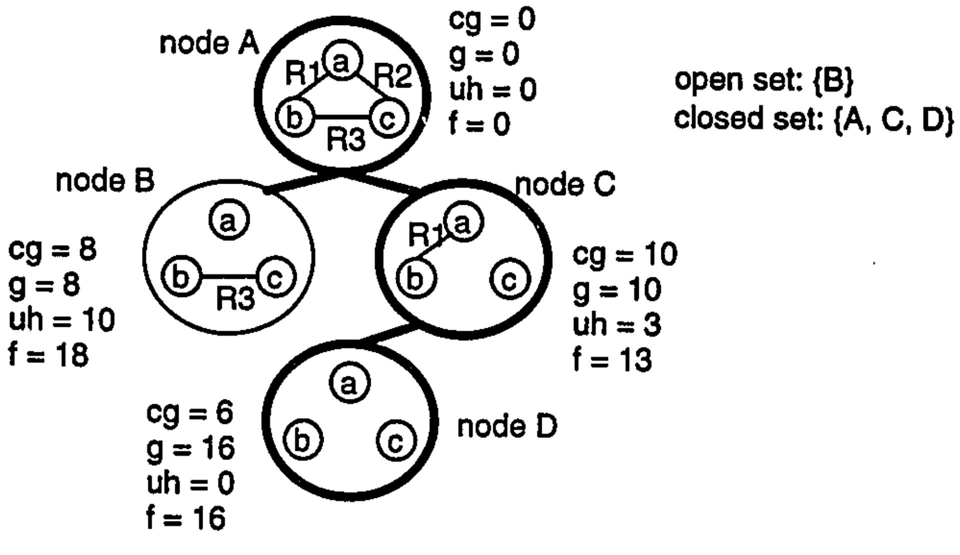


After first iteration

Fig. 7.21: State of the search graph after each iteration of the A* algorithm for a hypothetical 3–components product.



After second iteration



After third iteration

Fig. 7.21 (continued).

The first snapshot corresponds to the state of the search graph prior to any iteration of the search algorithm. Only the root node has been generated at this stage. The cost of this node, as reflected by the "f" value, is of course 0.

The second snapshot shows the state of the search graph after a first iteration of the search procedure. Node A, which was the only one on the open set, was expanded to yield nodes B and C. These two nodes were then added to the open set and node A was removed from the open set and added to the closed set.

In the third snapshot, node C was expanded and added to the closed set. Node D was generated, its various costs computed and added to the open set. Note that the cost of the best path to C, reflected by the "g" parameter, is actually higher than that of the best path to B. However, because of the high underestimation of the remaining cost in the best path from node B to the goal, node C was selected instead.

In the fourth snapshot, node D is now removed from the open set. As this node is recognized to be the goal node, the algorithm stops at this point. Using the best parent pointers, the optimal path from node D to node A gives the optimal assembly sequence:

1– assemble (a) with (b)

2– assemble (a, b) with (c).

If the actual cost of the path from node B to the goal was below 7 instead of the estimated 10, then the best path in the graph would be {A, B, D}, with a cost of 15 and lower, compared to 16 for {A, C, D}. Because of the overestimation of "uh" at node B, this path would never be found. This shows the importance of supplying an underestimation for the "uh" parameter in order to guarantee optimality in an A* search.

7.7 HILL CLIMBING SEARCH

The A* algorithm is very powerful, as long as good values for "uh" can be found. At one extreme, if the values for this parameter are perfect estimations, there is no search, i.e. the optimal path is found directly. At the other extreme, if the best underestimation is always 0, the A* algorithm becomes the same as the best first search method.

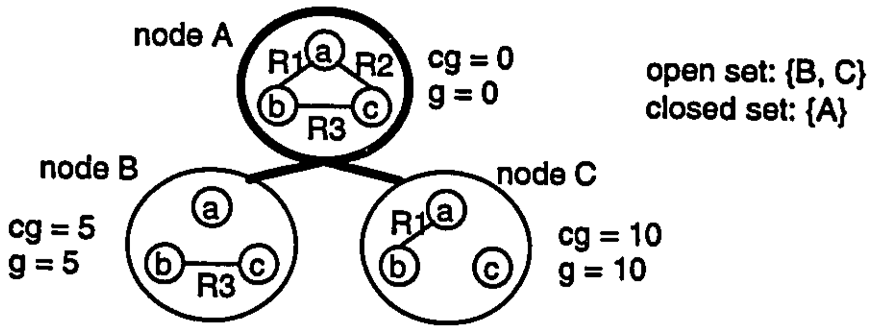
In GAPP, the heuristic functions used to compute values for "uh" are such that the number of nodes to be expanded before finding the optimal solution using an A* search is always much smaller than when using breadth first, depth first or best first search methods. Nevertheless, for products with a large number of components, the combinatorics of the problem are such that in spite of the significant reduction provided by the A* search, the number of nodes to be expanded may still be unacceptably large. In such cases, a hill climbing search can be performed instead.

Like the A* and best first search methods, the hill climbing search method is characterized by its ability to remove the best node on the open set and to stop the search as soon as the goal node is selected from this set. However, unlike any other methods discussed so far, the hill climbing method removes all other nodes on the open set once the best one has been selected from it. This makes backtracking to previously generated nodes impossible.

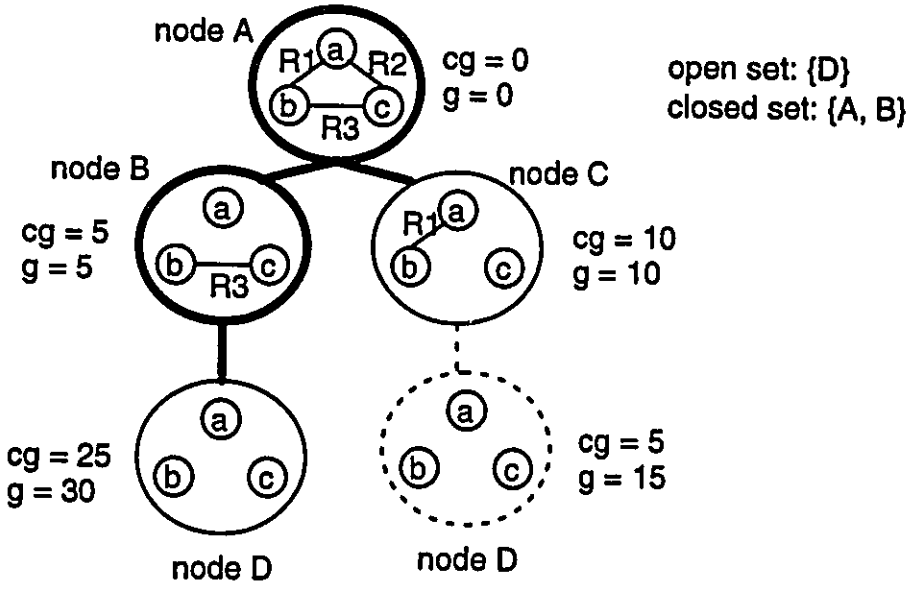
On one hand, the hill climbing search method ensures the generation of a good assembly sequence, due to its ability to always select the best node on the open set. Also, for a product of "n" components, it ensures that the assembly sequence is generated in polynomial time, i.e. in exactly n–1 expansions, since nodes are not accumulated on the open set. On the other hand, a limitation of this method is that it is does not guarantee the optimality of the found assembly sequence.

To illustrate this, consider figure 7.22. It shows the state of the search graph prior and after the second iteration of the hill climbing search procedure for the hypothetical 3–component product. Out of the two possible nodes on the open set that can be expanded in this second iteration, node B with lower cost is selected to yield node D. Node D is then added to the open set and node B removed from it and added to the closed set. As part of the hill climbing formalism, *node C is also removed from the open set.*

Clearly, in the next iteration of the algorithm, there is no other choice than to select node D for expansion, as this node is the only one on the open set. However, note how high the "g" value of node D is, compared to that of node C. This suggests that perhaps node C should have been expanded instead and a less expensive path to node D be found. Assume that effectively, the cost of the operation that transforms node C into node D is 5 units of cost. This , yields a "g" value of 15 for node D. What becomes obvious from this example is that the assembly sequence generated by the hill climbing search method was not the optimal one.



Before second iteration



After second iteration

Fig. 7.22: State of the search graph before (top) and after (bottom) the second iteration of the hill climbing search method for the hypothetical 3–components product.

CHAPTER 8

RESULTS

This chapter presents some of the results obtained from GAPP. Sections one, two and three analyze the effects of the search constraints, search criteria and search methods, respectively. Sections four, five and six show applications of GAPP in repair planning, error recovery planning and multiple product's planning, respectively. Section seven presents GAPP's output for the base subassembly of a heat detector device consisting of 17 components. GAPP's optimal solution for this assembly is also compared to the actual plan used in production. A typical screendump of GAPP's environment is provided at the end of this chapter.

8.1 EFFECTS OF THE SEARCH CONSTRAINTS

The main effect of the geometric feasibility constraint and those induced by restricted components is to reduce the search graph size. Depending on the particular product being analyzed, this reduction varies for each particular type of constraints.

Geometric feasibility constraints contribute to the elimination of edges in the directed graph. Therefore, expanding the graph while considering this type of constraints usually leads to a significant reduction in the number of edges and consequently in the number of paths in it, compared to the unconstrained graph. This is evident from the results in tables 8.1 and 8.2. The geometric feasibility constraint led to reductions of 91% and 96% in the number of assembly sequences (paths) for the air cylinder's and flashlight's directed graphs, respectively.

Table 8.1
Number of nodes and number of assembly sequences in the directed graph of assembly states of the air cylinder as a function of the constraints being used.

geometric feasibility	restricted components	# nodes expanded	# of assembly sequences
off	off	2,678	6,519,744
on	off	2,322	646,380
off	on	151	10,432
on	on	85	728

Table 8.2
Number of nodes and number of assembly sequences in the directed graph of assembly states of the flashlight as a function of the constraints being used.

geometric feasibility	restricted components	# nodes expanded	# of assembly sequences
off	off	858	333,792
on	off	331	12,896
off	on	414	66,024
on	on	192	5,425

On the other hand, the specification of restricted components in relations is transformed into forbidden states of the search graph. Therefore, the use of such constraints contributes to a reduction in the number of nodes of the directed graph of assembly states, compared to the unconstrained graph. In particular, when considering constraints induced by restricted components, tables 8.1 and 8.2 show reductions of 94% and 52% in the number of nodes in the air cylinder's and flashlight's directed graphs, respectively. The smaller reduction for the flashlight is easily explained by the fact that only 2 restricted components in 2 relations were specified for this product, compared to 7 restricted components in 6 relations for the air cylinder. There were therefore a lot more forbidden states generated for the latter product.

8.2 EFFECTS OF THE SEARCH CRITERIA

Once the search space has been reduced through the computation of constraints, the remaining solutions must be evaluated, using the four criteria, and the best one selected[53]. This section highlights the consequence of specifying the relative weights of the four assembly planning criteria defined in this thesis. In particular, different weight assignments lead to different optimal assembly sequences being generated for the same product. Figures 8.1, 8.2, 8.3 and 8.4 will be used for illustration.

In figure 8.1, an assembly sequence of the flashlight was obtained without using any of the four criteria. Although this assembly sequence is feasible, it may not be optimal. In fact, it is easy to see that the operations in lines 6 through 10 lead to highly unstable assembly configurations.

53. If no criterion is specified, an arbitrary solution is generated by GAPP.

1– *fit (lens) (head) from z+*

2– *screw (bulb) (reflector) from z+*

3– *RE–ORIENT 180 DEGREES*

4– *screw (reflector bulb) (head lens) from z+*

5– *screw (body) (head lens reflector bulb) from z+*

6– *against (battery1) (battery2) from z+x+y+x–y–*

7– *against (battery2 battery1) (spring) from x–y+x+y–z+*

8– *fit and twist (spring battery2 battery1) (endcap) from z+*

9– *RE–ORIENT 180 DEGREES*

10– *screw (endcap spring battery2 battery1) (body head lens reflector bulb) from z+*

Fig. 8.1: A feasible assembly sequence of the flashlight obtained without using any criterion.

1– *screw (bulb) (reflector) from z+*

2– *RE–ORIENT 180 DEGREES*

3– *fit (lens) (head) from z+*

4– *screw (reflector bulb) (head lens) from z+*

5– *screw (body) (head lens reflector bulb) from z+*

6– *fit (battery1) (body head lens reflector bulb) from z+*

7– *fit (battery2) (body head lens reflector bulb battery1) from z+*

8– *fit and twist (spring) (endcap) from z+*

9– *RE–ORIENT 180 DEGREES*

10– *screw (endcap spring) (body head lens reflector bulb battery1 battery2) from z+*

Fig. 8.2: Optimal assembly sequence of the flashlight with respect to the stability criterion.

To remedy this situation, a new solution was generated in figure 8.2, using the stability criterion with a weight of 50 units. This time, the generated assembly sequence is optimal with respect to this criterion. However, it might be poor with respect to any of the other three criteria which have not been used. For example, a little investigation of this sequence shows that it is not very good with respect to parallelism, as shown by lines 5, 6, 7 and 10.

At this point, it would seem like a good trade off to consider both the stability and parallelism criteria in the search process and provide them with equal relative weight, for example 50. The solution corresponding to this strategy is shown in figure 8.3. Note how inefficient this solution became with respect to the re–orientation criterion, which was not considered. Also note how the operations in lines 9 through 12 are still highly unstable, in order to provide better parallelism in the plan.

This example shows the importance of the selection of the relative weight of the various criteria. In this research, these relative weights are provided by the user using common sense and experience. For example, providing more weight to the stability criterion than any other is a recommended practice. A formal solution to the important issue of selecting relative weights is not covered by this research and promises to be a challenging area for future work. Figure 8.4 shows another solution where all criteria with equal weight of 50 units have been used.

Clearly, the variation of the relative weight of the various evaluation criteria leads to the generation of different optimal assembly sequences for the same product. This feature can be exploited in concurrent engineering where alternate assembly sequences obtained by changing the importance of various criteria can be analyzed and compared.

1– fit and twist (spring) (endcap) from z+
2– RE–ORIENT 180 DEGREES
3– screw (bulb) (reflector) from z+
4– RE–ORIENT 180 DEGREES
5– fit (lens) (head) from z+
6– fit (battery2) (body) from z+z–
7– screw (endcap spring) (body battery2) from z+
8– RE–ORIENT 180 DEGREES
9– against (battery1) (bulb reflector) from x–y+x+y–z+
10– screw (reflector bulb battery1) (head lens) from z+
11– RE–ORIENT 180 DEGREES
12– screw (battery1 bulb reflector head lens) (battery2 body spring endcap) from z+
13– RE–ORIENT 180 DEGREES

Fig. 8.3: Optimal assembly sequence of the flashlight with respect to the stability and parallelism criteria with equal weight.

1– fit and twist (spring) (endcap) from z+
2– RE–ORIENT 180 DEGREES
3– fit (lens) (head) from z+
4– fit (battery1) (body) from z+z–
5– fit (battery2) (body battery1) from z+
6– screw (endcap spring) (body battery1 battery2) from z+
7– screw (bulb) (reflector) from z+
8– RE–ORIENT 180 DEGREES
9– screw (reflector bulb) (head lens) from z+
10– screw (body battery1 battery2 spring endcap) (head lens reflector bulb) from z+

Fig. 8.4: Optimal assembly sequence of the flashlight with respect to all four criteria with equal weight.

8.3 EFFECTS OF THE SEARCH METHODS

For any of the five available search methods except hill climbing, GAPP is guaranteed to return the optimal assembly sequence with respect to any of the four criteria which have been turned on. However, because different search methods explore different paths of the search graph first, different optimal assembly sequences can sometimes be returned by different search methods, in terms of the operations to be performed and their order of execution. Of course, the total cost of these assembly sequences will necessarily be the same. Differences in the sequences obtained from different search methods simply express the fact that there can be more than one optimal assembly sequence for a given product.

Another effect of the different search methods relates to the search time and space required to generate an optimal solution. Both the breadth first and depth first methods, which are exhaustive, must expand the whole search graph in order to find the optimal solution. On the other hand, the best first and A^* methods can sometimes generate the optimal solution rather directly. Finally, the hill climbing method always finds a solution in polynomial time, with no guarantee of optimality.

Table 8.3 summarizes these facts for the air cylinder product. All four criteria were used and given an equal weight of 50 units of cost. Note the important reduction of nodes obtained from the A^* search. Also note that the hill climbing method found a solution which cost is very close to that of the optimal solutions returned by all the other four methods[54]. The solution returned using the first four methods in table 8.3 was the same and it is presented in figure 8.5. The solution returned using the hill climbing method is presented in figure 8.6.

54. That the hill climbing method results in a solution almost as good as the optimal one is a particularity of this example and cannot be generalized.

Table 8.3
Comparison of the search space size required to find an optimal solution for the air cylinder using five different search methods.

search method	# nodes on open	# nodes on closed	total # of nodes	total search cost
breadth first	0	122	122	553.10
depth first	0	122	122	553.10
best first	39	65	104	553.10
A^*	34	38	72	553.10
hill climbing	0	10	10	553.24

1– against (piston) (piston_rod) from z+

2– screw (piston_screw) (piston_rod piston) from z+

3– fit (bearing_o_ring) (bearing) from z+z–

4– fit (cover_o_ring) (body) from z+

5– fit (bearing bearing_o_ring) (body cover_o_ring) from z+z–

6– fit (piston_o_ring) (piston piston_rod piston_screw) from z+z–

7– fit (piston_o_ring piston piston_rod piston_screw) (body bearing bearing_o_ring cover_o_ring) from z+

8– fit (cover) (bearing_o_ring bearing piston_rod body piston_o_ring piston cover_o_ring piston_screw) from z+

9– screw (screws) (cover body bearing bearing_o_ring piston_rod piston_o_ring piston cover_o_ring piston_screw) from z+

10– RE–ORIENT 180 DEGREES

Fig. 8.5: Optimal assembly sequence of the air cylinder obtained using the first four search methods in table 8.3.

1– against (piston) (piston_rod) from z+

2– screw (piston_screw) (piston_rod piston) from z+

3– fit (piston_o_ring) (piston piston_rod piston_screw) from z+z–

4– fit (bearing_o_ring) (bearing) from z+z–

5– fit (cover_o_ring) (body) from z+

6– fit (bearing bearing_o_ring) (body cover_o_ring) from z+z–

7– fit (piston_o_ring piston piston_rod piston_screw) (body bearing bearing_o_ring cover_o_ring) from z+

8– fit (cover) (bearing_o_ring bearing piston_rod body piston_o_ring piston cover_o_ring piston_screw) from z+

9– screw (screws) (cover body bearing bearing_o_ring piston_rod piston_o_ring piston cover_o_ring piston_screw) from z+

10– RE–ORIENT 180 DEGREES

Fig. 8.6: Assembly sequence of the air cylinder obtained using the last search method in table 8.3 (hill climbing).

8.4 GENERATION OF REPAIR PLANS

Designing a product for ease of assembly is a well established concept. Recently, concepts such as design for disassembly, maintainability and repairability have also been introduced. For example, some components within a product may be expected to fail during service and be replaced periodically. The efficiency of the disassembly sequences enabling access to this part may vary with different design solutions. GAPP enables an analysis of various disassembly sequences required for repairs for particular design solutions.

Consider the air cylinder's bearing_o_ring. This is an example of a part which is likely to be replaced several times during this product's life. To reach this faulty part, the cover_screws must be removed first, then the cover and cover_o_ring, and finally the subassembly made up of the piston, piston_rod, piston_o_ring and piston_screw (figure 8.7). In order for GAPP to generate such a plan, all that is required is the identification of those relations of the graph model which should not be broken in the repair disassembly plan. In this particular example, these relations are as follows:

(bearing, bearing ring)
(bearing, body)
(piston, piston_rod)
(piston, piston_o_ring)
(piston, piston_screw)
(piston rod, piston_screw).

The goal flag of these relations must then be set to the value "yes" in the product description file read by GAPP. These relations will not be part of any cutset used in the search process. This ensures that the generated disassembly plan will preserve the above relations. The optimal disassembly sequence returned by GAPP for this example is presented in figure 8.8.

Once again, different solutions can be generated by changing the relative weights of the criteria. Also, given a fixed set of weight assignments, the cost of optimal disassembly sequences obtained for different design solutions can be compared, and the design yielding the lowest disassembly cost can be selected.

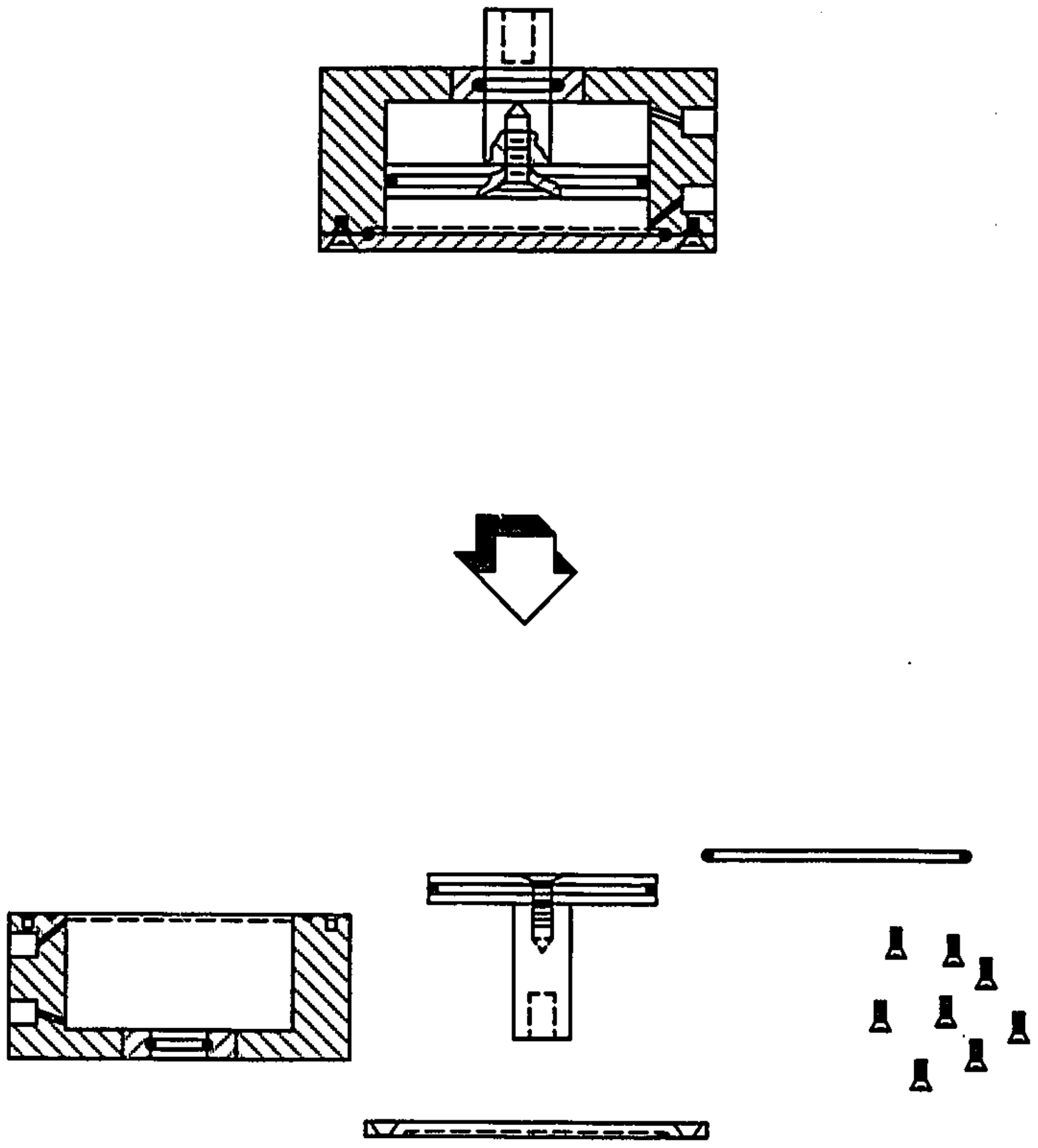


Fig. 8.7: Transforming the air cylinder's completely assembled state into a state enabling access to the faulty bearing_o_ring.

1– RE–ORIENT 180 DEGREES

2– unscrew (screws) (cover body bearing bearing_o_ring piston_rod piston_o_ring piston cover_o_ring piston_screw) from z+

3– unfit (cover) (bearing_o_ring bearing piston_rod body piston_o_ring piston cover_o_ring piston_screw) from z+

4– unfit (cover_o_ring) (body bearing bearing_o_ring piston_rod piston_o_ring piston piston_screw) from z+

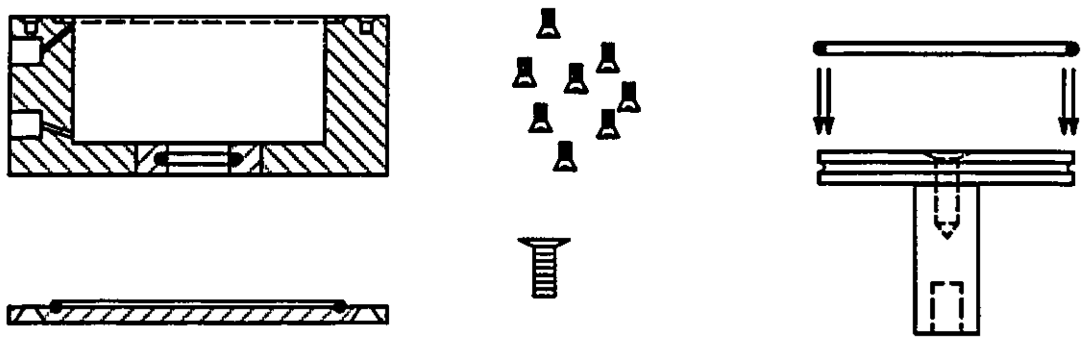
5– unfit (piston_o_ring piston piston_rod piston_screw) (body bearing bearing_o_ring) from z+

Fig. 8.8: Optimal repair plan generated by GAPP to access the faulty bearing_o_ring.

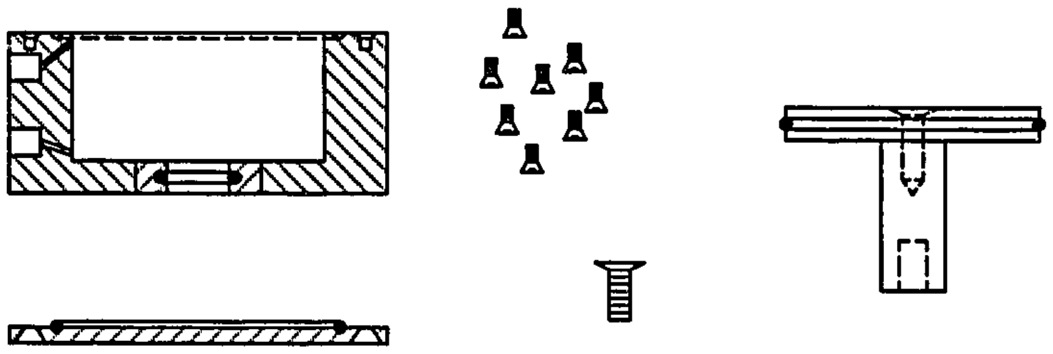
8.5 RECOVERING FROM ASSEMBLY ERRORS

An important characteristic of any assembly planner is its ability to generate plans for recovering from errors which occur during the execution of assembly (disassembly) operations. In particular, the planner must have the ability to generate a new set of operations to transform the unpredictable assembly state due to the error into the goal configuration of the product. The following example illustrates how such functionality is achieved in GAPP.

Consider an assembly operation of the air cylinder which changes the state depicted in figure 8.9a into that of figure 8.9b by placing the piston_o_ring on top of the {piston, piston_rod} subassembly. Due to the instability of this last subassembly, assume that during the execution of this operation, the piston falls from the piston_rod such that the new state depicted in figure 8.10 is obtained as a result. In order to recover from this error, a new optimal plan starting at this unpredictable state must be generated. This requires the simple identification of all the relations of the resulting state which have already been established in previous successful assembly operations.



(a)



(b)

Fig. 8.9: Transforming the state in (a) into that of (b) by placing the piston_o_ring on top of the {piston, piston_rod} subassembly.

For this example, these established relations are simply:

(bearing, bearing_o_ring),

(bearing, body), and

(cover, cover_o_ring).

Setting the goal flag of these relations to the value "yes" enables GAPP to generate a new assembly plan, *possibly including different operations than those in the original plan*, and which does not involve the re–establishment of those relations which have already been established before the error occurred. Figure 8.11 shows the recovery plan generated by GAPP for this particular example.

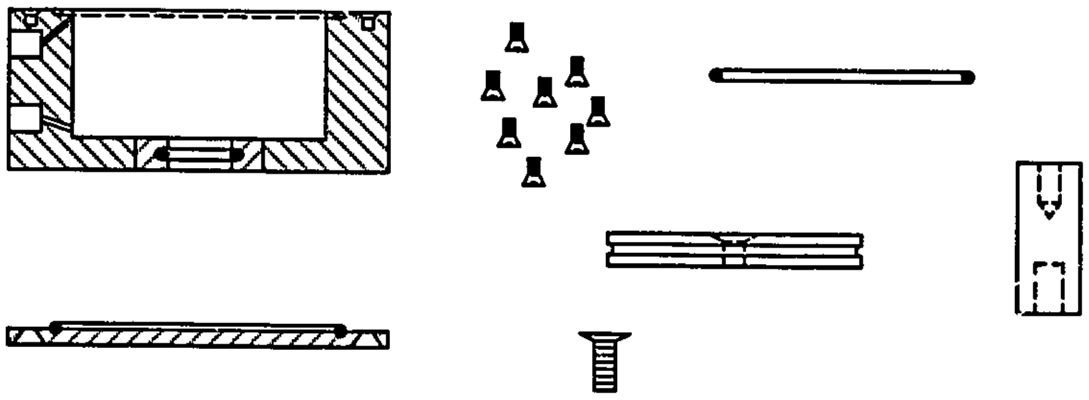


Fig. 8.10: Unexpected state resulting from an error in the execution of the operation in figure 8.9.

1– *against (piston) (piston_rod) from z+*

2– *screw (piston_screw) (piston_rod piston) from z+*

3– *fit (piston_o_ring) (piston piston_rod piston_screw) from z+z–*

4– *fit (piston_o_ring piston piston_rod piston_screw) (body bearing bearing_o_ring cover_o_ring) from z+*

5– *fit (cover) (bearing_o_ring bearing piston_rod body piston_o_ring piston cover_o_ring piston_screw) from z+*

6– *screw (screws) (cover body bearing bearing_o_ring piston_rod piston_o_ring piston cover_o_ring piston_screw) from z+*

7– *RE–ORIENT 180 DEGREES*

Fig. 8.11: An error recovery plan generated by GAPP.

8.6 MULTIPLE PRODUCTS PLANNING

Consider the identical products in figure 8.12. It was demonstrated in section 6.4.1 that at least one tool change is required to disassemble one of them. Assume a robot must assemble these two products consecutively. Assume also that the strategy used to accomplish this task is to repeat the execution of the optimal plan obtained for a single product. One possible optimal plan consists of removing the top block, changing the tool, then removing both screws. If this plan is to be executed for the disassembly of both products, then 3 tool changes will be required: one before removing both screws of the first product, one before removing the block of the second product, and one before removing the screws of the second product.

Alternatively, a better solution would be to change the plan for the second product and take advantage of the current tool mounted on the robot after the disassembly of the first product. For example, after the first tool change to remove both screws on the first product, the same tool could be kept and used again to remove the screws on the second product. A second tool change is still required to remove the top block from the second product.

The best solution consists of removing both blocks, changing the tool, then removing all 4 screws. Although obvious, this solution is characterized by *consecutive operations on different products.* In other words, both products must be analyzed simultaneously in order to come up with the optimal plan to disassemble them both. The graph–theoretic approach to assembly planning developed in chapter 4 enables GAPP to handle such an important class of assembly problems. The graph model of each product (whether identical or different) is simply considered an unconnected graph model. They are considered collectively and the optimal solution generated by GAPP for this problem is presented in figure 8.13. Note that only one tool change is required. Also note that GAPP chose to start with the screwing operations rather than the placement of blocks. This is still an optimal solution.

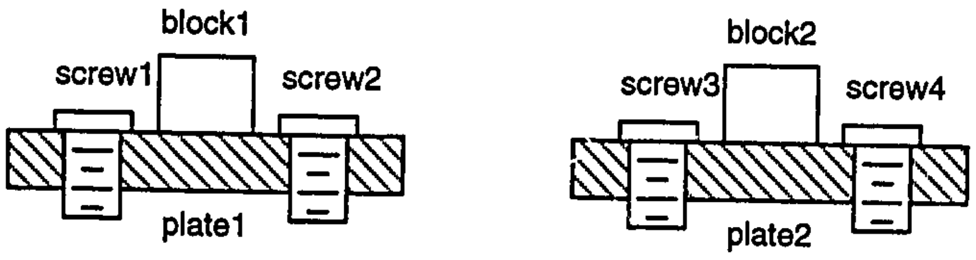


Fig. 8.12: Two products to be disassembled.

1– screw (screw4) (plate2) from z+
2– screw (screw3) (plate2 screw4) from z+
3– screw (screw2) (plate1) from z+
4– screw (screw1) (plate1 screw2) from z+
5– against (block2) (plate2 screw3 screw4) from z+x+y+x–y–
6– against (block1) (plate1 screw1 screw2) from z+x+y+x–y–

Fig. 8.13: Optimal multiple products plan generated by GAPP.

8.7 A MORE COMPLEX PRODUCT

So far, the air cylinder (10 components) and flashlight (9 components) products were used to demonstrate some of the key concepts developed in this thesis. This section shows the optimal solution generated by GAPP for the base subassembly of a heat detector device which consists of 17 components. An exploded view of this product is shown in figure 8.14. The corresponding product description file is shown in figure 8.15. The actual manual assembly plan used by the company which produces this device is shown in figure 8.16. The solution generated by GAPP is shown in figure 8.17. This solution was generated using equal relative weights of 50 units for each of the four defined criteria. The A^* search method was used. There were 343 and 1,810 nodes in the open set and closed sets, respectively, at the time when the optimal solution was found. Real elapsed time to get this solution was about 9 minutes on a Sun SPARCstation2.

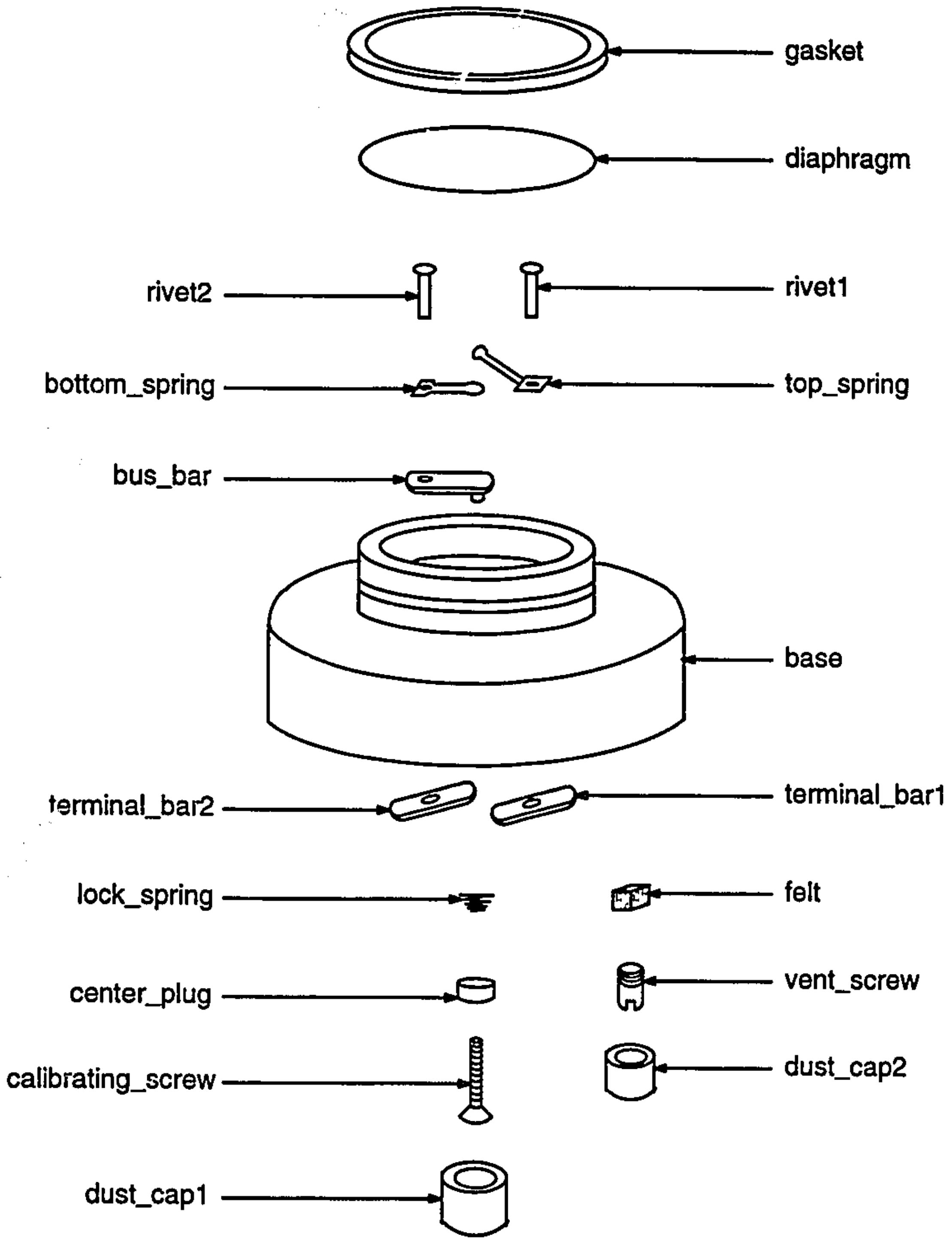


Fig. 8.14: Exploded view of the base subassembly of a heat detector device (17 components).

```
component gasket
component diaphragm
component top_spring
component bottom_spring
component bus_bar
component rivet1
component rivet2
component base
component terminal_bar1
component terminal_bar2
component lock_spring
component center_plug
component cal_screw
component dust_cap1
component felt
component vent_screw
component dust_cap2

relation no gasket diaphragm yes z+ z– attach press
relation no diaphragm base no z+x+x–y+y– z–x+x–y+y– attach press gasket

relation no bus_bar base no z+ z– contact fit rivet1 bottom_spring top_spring
relation no bottom_spring bus_bar no z+ z– contact against rivet1
relation no top_spring base no z+ z– contact against rivet2

relation no bottom_spring diaphragm yes z– z+ blocking
relation no top_spring diaphragm yes z– z+ blocking
relation no bus_bar diaphragm yes z– z+ blocking
relation no rivet1 diaphragm yes z– z+ blocking
relation no rivet2 diaphragm yes z– z+ blocking
```

Fig. 8.15: Product description file generated for the base subassembly of a heat detector device.

```
relation no rivet1 bottom_spring no z+ z– contact against
relation no rivet1 terminal_bar1 no z+ z– attach rivet
relation no rivet2 top_spring no z+ z– contact fit
relation no rivet2 terminal_bar2 no z+ z– attach rivet
relation no lock_spring base no z– z+ contact fit center_plug cal_screw
relation no center_plug base no z– z+ contact fit
relation no center_plug cal_screw no z+ z– attach fit lock_spring
relation no cal_screw base no z– z+ attach screw
relation no cal_screw bus_bar yes z– z+ contact against
relation no dust_cap1 base no z– z+ attach fit
relation no cal_screw dust_cap1 yes z+ z– blocking
relation no center_plug dust_cap1 yes z+ z– blocking
relation no lock_spring dust_cap1 yes z+ z– blocking

relation no felt base no z– z+ attach fit vent_screw
relation no vent_screw felt yes z– z+ contact against
relation no vent_screw base no z– z+ attach screw
relation no dust_cap2 base no z– z+ attach fit
relation no dust_cap2 vent_screw yes z– z+ blocking
relation no dust_cap2 felt yes z– z+ blocking

relation no terminal_bar1 base no z– z+ contact against rivet1
relation no terminal_bar2 base no z– z+ contact against rivet2
```

Fig. A.5 (continued).

1– against terminal bar1 and base

2– against terminal bar2 and base

3– RE–ORIENT base

4– fit bus bar on base

5– against bottom spring and bus bar

6– against top spring and base

7a– rivet bottom spring and bus bar to terminal bar1

7b– rivet top spring to terminal bar2

8– RE–ORIENT base

9– fit center plug into calibrating screw

10– RE–ORIENT calibrating screw

11– fit lock spring to base

12– fit calibrating screw to base

13– fit dust cap1 to base

14– fit felt to base

15– fit vent screw to base

16– fit dust cap2 to base

17– RE–ORIENT base

18a–press diaphragm to base

18b–press gasket to base

Fig. 8.16: Assembly sequence of the base subassembly used by the company in the actual production.

1– fit (felt) (base) from z+

2– fit (center_plug) (cal_screw) from z+

3– RE–ORIENT 180 DEGREES

4– fit (lock_spring) (base felt) from z+

5– screw (cal_screw center_plug) (base lock_spring felt) from z+

6– screw (vent_screw) (base lock_spring center_plug cal_screw felt) from z+

7– against (terminal_bar2) (base lock_spring center_plug ...) from z+

8– against (terminal_bar1) (base lock_spring center_plug ...) from z+

9– fit (dust_cap2) (base lock_spring center_plug ...) from z+

10– fit (dust_cap1) (base lock_spring center_plug ...) from z+

11– RE–ORIENT 180 DEGREES

12– fit (bus_bar) (base lock_spring center_plug ...) from z+

13– against (bottom_spring) (bus_bar base lock_spring ...) from z+

14– against (top_spring) (base bus_bar bottom_spring ...) from z+

15– rivet (rivet2) (terminal_bar2 base bus_bar bottom_spring ...) from z+

16– rivet (rivet1) (terminal_bar1 base top_spring ...) from z+

17– press (diaphragm) (base bus_bar bottom_spring ...) from z+

18– press (gasket) (diaphragm base bus_bar bottom_spring ...) from z+

Fig. 8.17: Assembly sequence of the base subassembly generated by GAPP.

Compared to the plan used in production (figure 8.16), the one returned by GAPP (figure 8.17) is clearly better with respect to re–orientations. The simple explanation for this is that the bases are currently assembled manually. Therefore, due to the dexterity of the human hands, one is not too concerned about optimizing the number of re–orientations and the plan in figure 8.16 is considered satisfactory. On the other hand, if a robot or automatic assembly line were to perform the assembly of this product, re–orientations would become an important matter and a plan like the one generated by GAPP would certainly be more desirable. In particular, this plan avoids the design of complex assembly tools and fixtures often required for performing re–orientations. Furthermore, assembly cycle time is reduced by eliminating re–orientation operations.

One disadvantage of the plan produced by GAPP, compared to the one used in production, relates to the physical "proximity" of the assembly operations. In the production plan, operations were naturally grouped according to the location were they take place on the base. For example, in lines 14, 15 and 16 of figure 8.16, the felt is inserted first, then the vent screw is inserted to squeeze the felt, then the dust cap is placed to cover both parts. Clearly, as these operations focus on a specific location of the base, it was logical to group them into successive operations. This can be contrasted with GAPP's solution, where these same 3 operations now appear in lines 1, 4 and 9, respectively. This is a simple consequence of the fact that no criteria have been implemented to consider the notion of physical proximity of the assembly operations.

It is also worth mentioning that in the actual production plan, some of the operations are actually performed simultaneously, for example the riveting operations in lines 7a and 7b and the pressing operations in lines 18a and 18b in figure 8.16. GAPP cannot combine different operations into simultaneous ones due to the dichotomy assumption. Nevertheless, the clustering criterion helped make these operations successive, which may suggest their combination into simultaneous operations. This is indicated by lines 15 and 16 for the riveting operations and lines 17 and 18 for the pressing operations in figure 8.17.

The fact that the base is re–oriented only once in the plan produced by GAPP also means that all the operations on one side of the base are performed, then the base is re–oriented, then all the operations on the other side are performed. This prohibits the clustering of similar operations on different sides of the base into successive operations, which would necessarily require a re–orientation. By providing more weight to the clustering criterion and less weight to the re–orientation criterion, a plan requiring extra re–orientations but providing more clustering of similar operations on different sides of the base can be obtained. One such plan is shown in figure 8.18. Relative weights of 100 and 25 were specified for the clustering and re–orientation criterion, respectively. The other two criteria were turned off (zero weight). Note that in this new plan the base is now re–oriented twice such that many similar assembly operations on different sides could be clustered.

1– *fit (bus_bar) (base) from z+*

2– *RE–ORIENT 180 DEGREES*

3– *fit (felt) (base bus_bar) from z+*

4– *fit (center_plug) (cal_screw) from z+*

5– *RE–ORIENT 180 DEGREES*

6– *fit (lock_spring) (base bus_bar felt) from z+*

7– *screw (cal_screw center_plug) (base bus_bar lock_spring felt) from z+*

8– *screw (vent_screw) (base bus_bar lock_spring center_plug ...) from z+*

9– *fit (dust_cap2) (base bus_bar lock_spring center_plug ...) from z+*

10– *fit (dust_cap1) (base bus_bar lock_spring center_plug ...) from z+*

11– *against (terminal_bar2) (base bus_bar lock_spring center_plug ...) from z+*

12– *against (terminal_bar1) (base bus_bar lock_spring center_plug ...) from z+*

13– *RE–ORIENT 180 DEGREES*

14– *against (top_spring) (base bus_bar lock_spring center_plug ...) from z+*

15– *against (bottom_spring) (bus_bar base top_spring lock_spring ...) from z+*

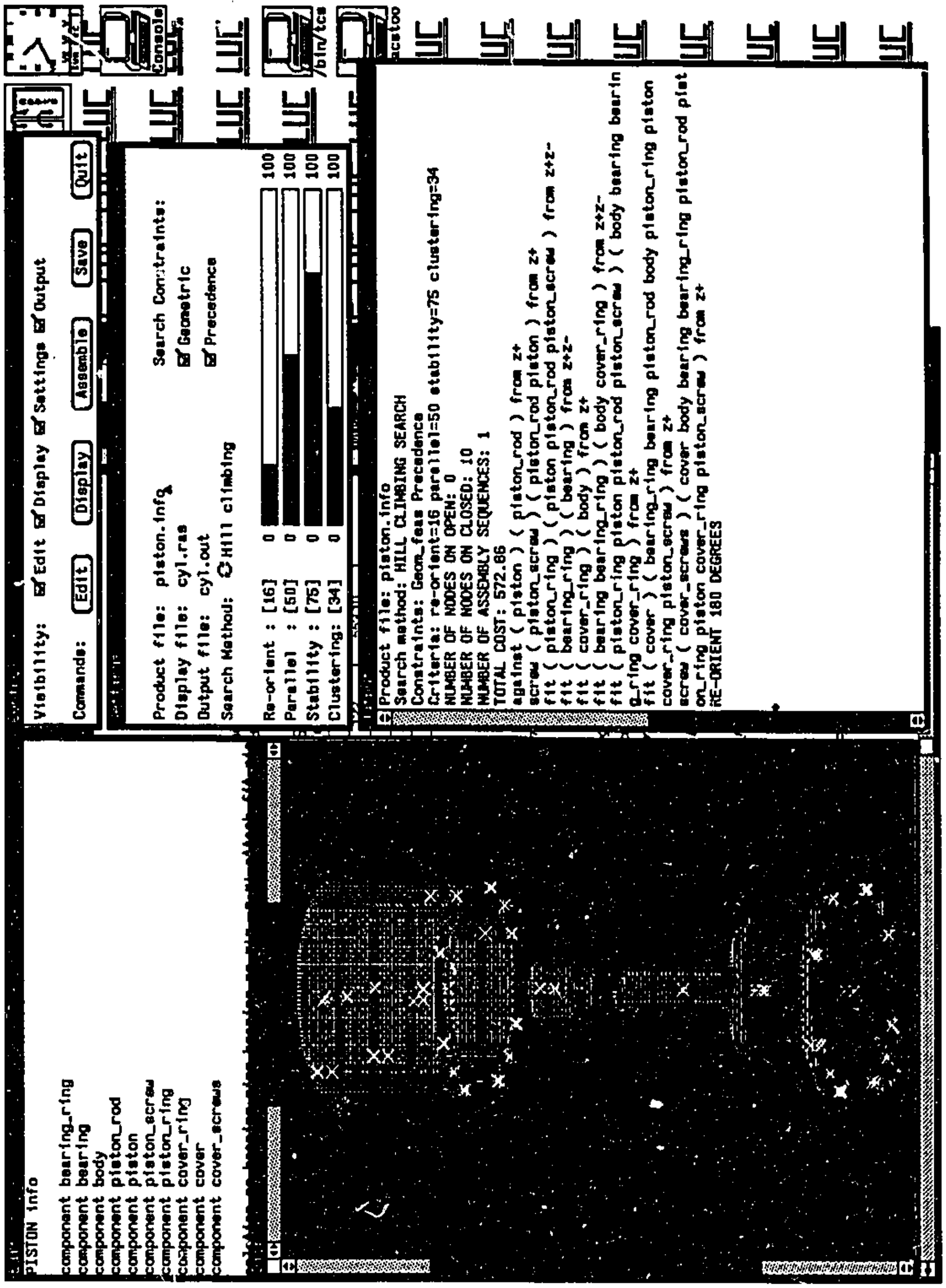
16– *rivet (rivet2) (terminal_bar2 base bus_bar bottom_spring ...) from z+*

17– *rivet (rivet1) (terminal_bar1 base bus_bar bottom_spring ...) from z+*

18– *press (diaphragm) (base bus_bar bottom_spring top_spring ...) from z+*

19– *press (gasket) (diaphragm base bus_bar bottom_spring ...) from z+*

Fig. 8.18: Assembly sequence of the base subassembly obtained by providing more weight to the clustering criterion.



```
PISTON info

component bearing_ring
component bearing
component body
component piston_rod
component piston
component piston_screw
component piston_ring
component cover_ring
component cover
component cover_screws
```

```
Visibility:  ☑Edit ☑Display ☑Settings ☑Output
Commands:  (Edit) (Display) (Assemble) (Save) (Quit)
```

```
Product file:  piston.info          Search Constraints:
Display file:  cyl.ras              ☑Geometric
Output file:   cyl.out              ☑Precedence
Search Method:  Hill climbing

Re-orient  : [16]   0 ▓▓░░░░░░░░░░ 100
Parallel   : [50]   0 ▓▓▓▓▓▓░░░░░░ 100
Stability  : [75]   0 ▓▓▓▓▓▓▓▓▓░░░ 100
Clustering: [34]    0 ▓▓▓▓░░░░░░░░ 100
```

```
Product file: piston.info
Search method: HILL CLIMBING SEARCH
Constraints: Geom_feas Precedence
Criteria: re-orient=16 parallel=50 stability=75 clustering=34
NUMBER OF NODES ON OPEN: 0
NUMBER OF NODES ON CLOSED: 10
NUMBER OF ASSEMBLY SEQUENCES: 1
TOTAL COST: 572.66
against ( piston ) ( piston_rod ) from z+
screw ( piston_screw ) ( piston_rod piston ) from z+
fit ( piston_ring ) ( piston piston_rod piston_screw ) from z+z-
fit ( bearing_ring ) ( bearing ) from z+z-
fit ( cover_ring ) ( body ) from z+
fit ( bearing bearing_ring ) ( body cover_ring ) from z+z-
fit ( piston_ring piston piston_rod piston_screw ) ( body bearing bearin
g_ring cover_ring ) from z+
fit ( cover ) ( bearing_ring bearing piston_rod body piston_ring piston
cover_ring piston_screw ) from z+
screw ( cover_screws ) ( cover body bearing bearing_ring piston_rod pist
on_ring piston cover_ring piston_screw ) from z+
RE-ORIENT 180 DEGREES
```

Fig. 8.19: Screendump of GAPP's windowing interface.

CHAPTER 9

DISCUSSION AND CONCLUSIONS

Chapter 2 described the state of the research in assembly planning and outlined some of the issues which required further research. The first section of this chapter summarizes research issues which have now been solved by GAPP. The second section includes a discussion of these achievements. The third section outlines some important issues which are still to be solved in future work.

9.1 SUMMARY OF ACHIEVEMENTS

9.1.1 Graph theoretic approach to assembly planning

Once the product has been mapped into the developed directed graph model, the expansion of the search graph first requires the determination of all the cutsets of this graph model. The feasibility of the disassembly operation corresponding to each cutset is then computed. If the cutset corresponds to a feasible operation, a new node is generated whose graph model is that of its parents minus the edge(s) of the cutset. This process is then repeated for the new child node, for its children, for its grandchildren, and so on, until a node with a completely unconnected graph corresponding to the completely disassembled state has been obtained.

A key element in the above approach is the determination of the set of all cutsets of the product's graph model. An algebraic method for doing so has been developed and presented in chapter 4. Basically, the method consists of first selecting a spanning tree of the graph model. A fundamental system of cutsets relative to this spanning tree is then obtained. The linear combinations of the cutsets in this fundamental system represent the set of all cutsets of the graph model.

The beauty of this approach is that it is applicable to any type of graph model – connected or unconnected. Physically, this means that the graph model of the product need not be restricted to a completely assembled product, nor to a single product. This enables GAPP to handle problems such as:

1) finding the sequence of assembly of a product which is to be only partly assembled leaving the remaining assembly for the consumer to complete,
2) finding the partial assembly sequence of a product resulting in the formation of subassemblies to be tested, and
3) finding the simultaneous sequence of assembly of multiple (different) products.

A method has also been developed for preventing already established relations from being re–established in the assembly plan. The method consists of eliminating all cutsets in which these relations are included. This enables GAPP to handle other interesting problems such as:

4) generating repair plans to replace faulty parts, and
5) re–planning to recover from unpredictable execution errors.

Most assembly planners developed to date are rigid in the assembly problems that they can tackle. For example, most assembly planners assume that the product is initially completely disassembled, and that the assembly process consists of completely assembling it. The graph–theoretic approach developed in this thesis enables GAPP to handle arbitrary initial and final states of the product.

This makes GAPP an innovative and very flexible assembly planner which can handle many different types of assembly problems uniformly.

9.1.2 Identification and formalization of evaluation criteria

Four criteria have been identified to assess the goodness of various assembly sequences:

1) the number of re–orientations required in executing a sequence,

2) the temporal independence of the operations in the sequence (parallelism),

3) the stability of the subassemblies involved in each operation, and

4) the clustering of similar operations into groups of successive operations.

It was shown in chapter 6 how these criteria represent metrics which can be used to assess the quality of assembly sequences. Methods of computing automatically these criteria as the search graph is expanded have also been described in chapters 6 and 7 and have been implemented in the GAPP software.

This represents an important contribution. For the first time, assembly plans can be evaluated automatically and the best one selected. This is in contrast to traditional assembly planners developed to date, which simply enumerate feasible assembly plans without any notion of their relative goodness. The importance of this novelty can be appreciated by the fact that, for any of the five assembly problems outlined in the previous section, GAPP has the ability to evaluate different solutions and render the optimal one, for example an optimal repair plan or an optimal error recovery plan, instead of simply finding an arbitrary feasible plan or enumerating all feasible plans.

9.1.3 Merging the generation and evaluation processes

The goal in assembly planning has been to develop software tools to assist (or even replace) planners in their complex task of selecting the assembly sequence to use in production. Researchers realized very early that methods of enumerating a product's numerous feasible assembly sequences were a prerequisite in the achievement of this goal. As a result, a large number of assembly planners were developed which focus on this particular problem. As better enumeration techniques emerged from these works, the focus of assembly planning naturally shifted towards the evaluation and selection of the many assembly sequences generated.

Although logical, this approach suffers from the important defect that the generation and selection processes are separated. As a result, means of storing the many possible solutions is required prior to their evaluation. It was one of GAPP's main objectives to break the established pattern:

generate then evaluate

and change it to:

evaluate as you generate.

The introduction of the various criteria in chapter 6 gives GAPP the ability to evaluate the many different disassembly sequences in the directed graph of assembly states. Integrating the evaluation of these criteria into the search procedure as the search graph gets expanded, enables the direct generation of an optimal disassembly sequence of a given product with respect to these criteria. Appropriate search methods, in particular the A^* algorithm, guide the search towards a single and optimal assembly sequence, without explicitly generating the whole directed graph of assembly states. The methods developed to achieve this integration of search and evaluation into a single process are an important novelty and represent a major contribution of this research.

9.1.4 Reduction of the combinatorics

Methods by which GAPP determines if a newly generated node already exists somewhere on the open or closed set have been developed and described in chapter 7. In terms of the search medium, these checks permit the expansion of a search graph instead of a search tree. Although these checks represent extra computational effort, their presence is well justified if one considers the important reduction in the number of expanded nodes that they provide. This can be verified by comparing the two columns in table 9.1. The former and latter represent the number of nodes resulting from performing an exhaustive breadth first search through a tree and graph structures, respectively. Both strategies returned the same optimal assembly sequence of the corresponding products.

Table 9.1
Comparison of the number of nodes generated in expanding a search tree and a search graph to find the optimal assembly sequence of three products.

product to be assembled	No. of generated nodes (search tree)	No. of generated nodes (search graph)
ball–point pen (6 components, figure 2.4)	39	14
air cylinder (10 components, figure 3.8)	2,160	122
flashlig [illegible] (9 compo [illegible] figure 6 [illegible]	15,311	421

Another concept developed to reduce the combinatorics is that of the geometric feasibility constraints and those induced by the specification of restricted components in relations. Geometric feasibility constraints contribute to the elimination of state transitions in a product's directed graph of assembly states. This usually leads to a significant reduction in the number of edges, and consequently in the number of paths, in this graph. The specification of restricted components in relations is transformed into forbidden states of the search graph. This contributes to a reduction in the number of nodes of a product's directed graph of assembly states. Together, both types of constraints can drastically reduce search time and space.

Integration of the A^* and hill climbing search methods also contributes greatly to the reduction of the search space, as shown earlier in table 8.3. In particular, the A^* method enables GAPP to generate an optimal assembly sequence without generating the whole search graph. This translates into an important reduction in the number of generated nodes for finding an optimal solution, compared to an exhaustive method like breadth first or depth first. For a product of "n" components, the hill climbing method always returns a feasible (but not necessarily optimal) assembly sequence of this product by expanding only n–1 nodes of the search graph.

Finally, the method developed in chapter 4 for updating the cutsets of a new state based on the knowledge of the cutsets of its parent state also contributes to a reduction in the combinatorial complexity of the assembly planning problem. This method avoids the total re–computation of the cutsets at each newly generated state of the search graph.

The complete integration of all the above concepts to reduce combinatorial complexity into the GAPP's software represents an important innovation provided by this thesis.

9.2 DISCUSSION

9.2.1 Problem size

Section 5.1 clearly showed the combinatorial complexity underlying assembly planning. In particular, the search space increases exponentially with the number of components and relations. Sections 5.3 and 5.4 showed how the computation of relevant constraints can help reduce the search space. This reduction varies from one product to another, depending on its geometry and other properties.

So far, the 17–components, 31–relations base subassembly of a heat detector device represents the largest problem solved by GAPP. However, this product is clearly not highly geometrically constrained by nature. This means that there probably exists more constrained products with larger number of components for which optimal assembly sequences can be generated by GAPP. Similarly, there might exist some very unconstrained products with less components for which GAPP will never find an optimal solution[55].

There are at least 3 different ways of overcoming this limitation:

1) Better underestimations can be developed for the 4 implemented criteria. The better the underestimations are, the more directly the optimal solutions using an A^* search are found. Ideally, if perfect underestimations can be obtained, then optimal solutions can be found in n–1 expansions for a

55. By not finding a solution, it is meant that the search graph grows large enough to actually exhaust available memory before any solution can be reached.

with "n" components. For example, using GAPP 16 expansions would be needed to find the optimal assembly of the base subassembly in figure 8.14 (n = 17), instead of the 1,810 expansions required using the currently implemented heuristics in GAPP.

2) A large product can be broken down into a number of smaller subassemblies. Individual optimal solutions for each subassemblies can be found at first. An optimal solution for combining the subassemblies to form the larger product can then be found.

3) While the search graph gets expanded, a concurrent process could be developed to eliminate already generated nodes, edges and paths which are not likely to be part of the optimal solution.

In the current GAPP implementation, the second solution above is directly applicable. The first and last solution would require further research.

9.2.2 Comparison with other assembly planners

To the author's knowledge, there exists only two assembly planners, apart from GAPP, provided with the ability to both generate and evaluate assembly plans automatically.

The first one is Homem De Mello and Sanderson's [26]. A fundamental difference is that this planner expands an AND / OR graph instead of a directed graph of assembly states. Benefits of this type of graph over the directed graph of assembly states include the fact that it can be substantially smaller (less nodes) and that it explicitly represents parallel operations. The smaller size implies that products with larger number of components can theoretically be handled. Nevertheless, in the published literature [25] the largest product described by the authors is the 11–parts Assembly From Industry first introduced by De Fazio and Whitney [11]. This is comparable to the air cylinder used throughout this thesis. In its actual implementation, another important difference is that this planner selects optimal assembly plans using an AO* search over the set of all plans. These plans must be exhaustively generated previously and encoded in the AND / OR graph prior to the search. In other words, the search is performed through an *explicit* search graph. GAPP integrates the generation and evaluation processes. In particular, optimal assembly sequences can be returned by GAPP without generating the whole search graph. In other words, the search is performed through an *implicit* search graph. An additional difference concerns the specificity of the generated optimal plan. GAPP returns a total ordering of assembly operations (an assembly sequence). Homem De Mello and Sanderson's planner returns only a partial order (an assembly plan).

The second assembly planner approaching GAPP's capabilities is Wolter's Xap–1 [80]. Like GAPP, this planner integrates the generation and selection processes. A branch and bound method, similar to the A* algorithm used by GAPP, helps generate optimal assembly sequences without exhaustive search. However, a binary search tree gets expanded instead of a graph structure. Most importantly, assembly operations involving subassemblies are not permitted. This last aspect is very restrictive and drastically reduces the search space by not considering a large number of plans involving subassemblies manipulation. This implies that products requiring subassemblies formation cannot be processed by this planner. In addition, although Wolter has implemented a few evaluation criteria, the important notion of parallelism among assembly operation is prohibited by this restriction. On the other hand, products with much larger number of parts can be products comply with this restriction (an example product with 37 parts which can be assembled by always adding one part at a time is presented in [80]). Finding optimal plans for such products requires much less time and space.

9.2.3 Programming environment

GAPP has been implemented with about 5,000 lines of C^{++} code. A nice user interface has also been developed using the Sunview windowing package (see figure 8.19 and also appendix C). Every part of the code dealing with the setup and control of the windowing interface has been kept in a separate file. To port GAPP on other machines not supporting Sunviews, all that is required is code compilation without linking this file. Similarly, other windowing packages can be designed and placed in a separate file to be linked during compilation.

9.3 FUTURE WORK

Any of the achievements in section 9.1 represent one more step towards the solution of the assembly planning problem. Nevertheless, some other important issues in this field remain a challenge for researchers. Among the most important ones are those outlined in the following sections.

9.3.1 Additional evaluation criteria

In its present implementation, the solutions generated by GAPP are optimal with respect to only the four criteria implemented so far. Further evaluation criteria which influence the selection of the plans need to be identified, and, most importantly, formalized. Examples include the physical proximity of assembly operation, execution time, subassembly's graspability and fixturability, and so on... This represents a natural extension of the current work. The developed methods can easily provide for such an extension.

9.3.2 Development of monetary cost functions

In the current implementation, the same assembly operation can have different cost values by simply varying the relative weights of the selected criteria. For example, an assembly operation transforming state A into state B may result in a unit cost of 115 when relative weights of 40 and 60 are used for the stability and re–orientation criteria, respectively. By simply changing the relative weights to 20 and 80, the new cost of the same operation could become 95, for example. Extending this concept to every assembly operation in a sequence, a variation of the relative importance of the criteria may actually lead to different optimal solutions being generated. As a result, different optimal assembly sequences might be generated depending on the relative weight assignments.

By using monetary costs in the computations instead of the subjective relative weights, each operation cost with respect to some criterion becomes invariant, and so is the cost of each individual assembly sequence. As a result, a single overall optimal assembly sequence can be generated, i.e. the one leading to minimal monetary assembly cost[56]. This is a most desirable feature of a planner as it eliminates subjectiveness in the selection of optimal assembly sequences.

Ideally, for a single assembly operation, individual monetary costs obtained for each criterion add up to give the total monetary cost of this assembly operation. Monetary costs of each operation in an assembly sequence add up to give the sequence's total cost.

An implementation of this concept requires the development of detailed models representing the monetary cost contribution of each relevant criterion. Due

56. It is reasonably assumed that the overall optimal assembly sequence of any product is the one which minimizes monetary costs.

to the large amount of variables to be considered in the determination of such costs, this concept promises to be a challenging area for future work in assembly planning, as well as in process planning in general.

9.3.3 Generation of assembly graph from CAD models

GAPP's graph model of the product is set up from the interpretation of the information which is manually entered in a product description file (appendix A). Although this file is fairly easy to generate even for complex products, a good idea would be to try to automate its creation. As a matter of fact, it has been recognized that the relations among components is one type of information which lends itself to being inferred from some CAD model of the product. Research in this area has already started and preliminary results seem encouraging [84].

9.3.4 Relaxing assumptions

The assembly plan of any product requiring the simultaneous assembly of two or more subassemblies towards a third one cannot be generated by a dichotomic planner; and the assembly plan of any product requiring parts to be assembled from successive translations in different directions cannot be generated by a monotone planner.

Although many products exist whose assembly plans violate one or both of these assumptions, they are used by most assembly planners developed to date, including GAPP. The relaxation of the dichotomy and monotone assumptions described in chapter 2 represent an important target for future research in assembly planning and would result in widening the scope of applications.

9.3.5 Design For Assembly feedback

GAPP is for now an open loop system, in the sense that it simply processes the product's graph model and produces the optimal assembly sequence for that product.

An interesting aspect of future work will be to provide GAPP with enough intelligence to actually recognize problematic design features of the product, i.e. features which contribute to higher assembly costs in the optimal plan. Suggestions of clever design changes to lower assembly costs could then be part of GAPP's outputs. The system developed in [36] seems to be a first move towards an implementation of this concept.

9.3.6 Link to scheduling

An important aspect of future work is to assign machines, tools, fixtures and operational times to every assembly operation generated by the planner. This constitutes an essential link between the process planning and scheduling activities. In particular, given the operations to be performed and their order of execution, and also given a list of possible resources for each operation (tools, fixtures, machines and time), a scheduler would select the resources for each operation such that machine utilization will be maximized and bottlenecks will be minimized. This promises to be a challenging issue as once again this selection is characterized by an important combinatorial complexity.

BIBLIOGRAPHY

[1] Asama H., Yokota K. and Yoshikawa H. (1988), "A Knowledge–Based Task Sequence Planning System for Maintenance Manipulators", *Preprints of Ro.man.sy '88*, pp. 1–8.

[2] Bondy J. A. and and Murty U. S. R. (1976), "Graph Theory With Applications", *North–Holland.*

[3] Bourjault A. (1984), "Contribution à une approche méthodologique de l'assemblage automatisé: élaboration automatique des séquences opératoires", *Thèse d'Etat*, Université de Besançon Franche–Comté, France.

[4] Bourjault A., Chappe D. and Henrioud J. M. (1987), "Elaboration Automatique des gammes d'assemblage à l'aide de réseaux de Pétri", *Automatique–Productique Informatique Industrielle*, Vol. 21, no. 4, pp. 323–342.

[5] Bourjault A., and Henrioud J. M. (1987), "Détermination des sous–assemblages d'un produit à partir des séquences temporelles d'assemblage", *Automatique–Productique Informatique Industrielle*, Vol. 21, no. 2, pp. 117–127.

[6] Campagne J.–P. and Caplat G. (1989), "Elaboration Automatique de gammes d'assemblage", *Automatique–Productique Informatique Industrielle*, Vol. 23, no. 1, pp. 53–68.

[7] Canny J. (1986), "Collision Detection for Moving Polyhedra", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI–8, No. 2, pp. 200–209.

[8] Chan K. C., Benhabib B. and Dai M. Q. (1990), "A Reconfigurable Fixturing System for Robotic Assembly", *Journal of Manufacturing Systems*, Vol. 9, No. 3, pp. 206–221.

[9] Chang K. H. and Wee W. G. (1988), "A Knowledge–Based Planning System for Mechanical Assembly Using Robots", *IEEE Expert*, Vol. 3, no. 1, pp. 18–30.

[10] Chen C. L. P. (1989), "Precedence Knowledge Acquisition for Generating Robot Assembly Sequences", *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, Vol. 1, pp. 71–76.

[11] De Fazio T. L. and Whitney, D. E. (1987), "Simplified Generation of all Mechanical Assembly Sequences", *IEEE Journal of Robotics and Automation*, Vol. RA–3, no. 6, pp. 640–658.

[12] De Fazio T. L. et al. (1989), "Aids for the Design or Choice of Assembly Sequences", *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, Vol. 1, pp. 61–70.

[13] De Floriani L. (1989), "A Graph Model for Face–to–Face Assembly", *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, pp. 75–78.

[14] Frommherz B. and Werling G. (1990), "Generating Robot Actions by Means of an Heuristic Search", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 884–889.

[15] Groover M. P. and Zimmers E. W. (1984), "CAD / CAM: Computer–Aided Design and Manufacturing", *Prentice–Hall.*

[16] Delchambre A. (1990), "A Pragmatic Approach to Computer–Aided Assembly Planning", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 1600–1605.

[17] Delchambre A. and Wafflard A. (1991), "An Automatic, Systematic and User–Friendly Computer–Aided Planner for Robotized Assembly", *Proceedings of IEEE International Conference on Robotics and Automation*, Sacramento, California, pp. 592–598.

[18] Haynes L. S. and Morris G.H. (1988), "A Formal Approach to Specifying Assembly Operations", *International Journal of Machines Tools and Manufacturing*, Vol. 28, No. 3, pp. 281–298.

[19] Heemskerk C. J. M. and Van Luttervelt, C. A. (1989), "The use of Heuristics in assembly sequence Planning", *The Annals of CIRP*, Vol. 38, no. 1, pp. 37–40.

[20] Heemskerk C. J. M. and Reijers L. N. (1990), "A Concept for Computer–Aided Process Planning of Flexible Assembly", *The Annals of CIRP*, Vol. 39, no. 1, pp. 25–28.

[21] Hoffman R. L. (1989), "Automated Assembly in a CSG Domain", *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, pp. 210–215.

[22] Homem De Mello L. S. (1989), "Task Sequence Planning for Robotic Assembly", *Ph.D. Thesis*, Carnegie–Mellon University, Department of Electrical and Computer Engineering, Pittsburgh, PA.

[23] Homem De Mello L. S. and Sanderson A. C. (1988), "Automatic Generation of Mechanical Assembly Sequences", *Report no. CMU–RI–TR–88–19*, The Robotics Institute, Carnegie–Mellon University, Pittsburgh, PA.

[24] Homem De Mello L. S. and Sanderson A. C. (1989), "A Correct and Complete Algorithm for the Generation of Mechanical Assembly Sequences", *Proceedings of IEEE International Conference on Robotics Automation*, Scottsdale, Arizona, pp. 56–61.

[25] Homem De Mello L. S. and Sanderson A. C. (1990), "AND / OR Graph Representation of assembly Plans", *IEEE Transactions on Robotics and Automation*, Vol. 6, No. 2, pp.188–199.

[26] Homem De Mello L. S. and Sanderson A. C. (1990), "Evaluation and Selection of Assembly Plans", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 1588–1593.

[27] Hoummady A. and Ghosh K. (1989), "Generation and Evaluation of Assembly Sequences in Computer–Automated Process Planning", *International Journal of Computer Applications in Technology*, Vol. 2, No. 3, pp. 151–158.

[28] Huang Y. F. and Lee C. S. G. (1989), "Precedence Knowledge in Feature Mating Operation Assembly Planning", *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, pp. 216–221.

[29] Huang Y. F. and Lee C. S. G. (1990), "An Automatic Assembly Planning System", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 1594–1599.

[30] Huang Y. F. and Lee C. S. G. (1991), "A Framework of Knowledge–Based Assembly Planning", *Proceedings of IEEE International Conference on Robotics and Automation*, Sacramento, California, pp. 599–604.

[31] Hutchinson S. A. and Kak A. C. (1990), "Extending the Classical AI Planning Paradigm to Robotic Assembly Planning", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 182–189.

[32] Hutchinson S. A. and Kak A. C. (1990), "SPAR: A Planner That Satisfies Operational and Geometric Goals in Uncertain Environments", *AI Magazine*, Vol. 11, No. 1, pp. 30–61.

[33] Khosla P. K. and Mattikali, R. (1989), "Determining the Assembly Sequence from a 3–D Model", *Journal of Mechanical Working Technology*, Vol. 20, pp. 153–162.

[34] Kim S. H. and Lee K. (1989), "An Assembly Modelling System for Dynamic and Kinematic Analysis", *Computer Aided Design*, Vol. 21, No. 1, pp. 2–12.

[35] Ko, H. and Lee K. (1987), "Automatic Assembling Procedure from Mating Conditions", *Computer Aided Design*, Vol. 19, no. 1, pp. 3–10.

[36] Kroll E., Lenz E. and Wolberg J. R. (1988), "A Knowledge–Based Solution to the Design–For–Assembly Problem", *Manufacturing Review*, Vol. 1, no. 2, pp. 104–108.

[37] Laperrière L. (1989), "Automatic Generation of Robotic Assembly Sequence", *Master's Thesis*, McMaster University, Hamilton, Ontario, Canada.

[38] Laperrière, L. (1990), "A Study of the Combinatorial Complexity of Automatically Generating Mechanical Assembly Sequences", *Report no. FMRD–08–01–1990*, Flexible Manufacturing Research and Development Centre, McMaster University, Hamilton, Ontario, Canada.

[39] Laperrière L. and ElMaraghy H. A. (1989), "Automatic Generation of a Robotic Assembly Sequence", *Proceedings of 1st ASME International Conference in Flexible Assembly*, Montréal, Canada, pp. 15–22.

[40] Laperrière L. and ElMaraghy H. A. (1991), "Automatic Generation of a Robotic Assembly Sequence", *International Journal of Advanced Manufacturing Technology*, Vol. 6, No. 4, pp. 299–316.

[41] Lee K. and Andrews G. (1985), "Inference of the Positions of Components in an Assembly: part 2", *Computer Aided Design*, Vol. 17, No. 1, pp. 25–29.

[42] Lee K. and Gossard D. C. (1985), "A Hierarchical data Structure for Representing Assemblies: part 1", *Computer Aided Design*, Vol. 17, No. 1, pp. 15–19.

[43] Lee S. and Gil Shin Y. (1990), "Assembly Planning Based on Subassembly Extraction", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 1606–1611.

[44] Lieberman L. I. and Wesley M. A. (1977), "AUTOPASS: An Automatic Programming System for computer controlled Mechanical Assembly", *IBM Journal of Research and Development*, Vol. 21, July issue, pp. 321–333.

[45] Lin A. C. and Chang T. C. (1989), "A Framework for Automated Mechanical Assembly Planning", *Journal of Mechanical Working Technology*, Vol. 20, pp. 237–248.

[46] Liou F. W. et al. (1989), "Design of a Flexible Fixture for Flexible Assembly – A Case Study", *Proceedings of 1st ASME International Conference in Flexible Assembly*, Montréal, Canada, pp. 85–92.

[47] Lippman S. B. (1989), "C++ Primer", *Addison–Wesley.*

[48] Liu C. L. (1968), "Introduction to Combinatorial Methematics", *McGraw–Hill.*

[49] Liu C. L. (1977), "Elements of Discrete Mathematics", *McGraw–Hill.*

[50] Liu P.–S. and Fu L.–C. (1990), "An Efficient Method of Solving Problems of Classification and Selection Using Minimum Spanning Tree in a Flexible Manufacturing System", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 2148–2153.

[51] Liu Y. and Popplestone R. J. (1989), "Planning for Assembly From Solid Models", *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, pp. 222–227.

[52] Lozano–Perez T. (1983), "Spatial Planning: A Configuration Space Approach", *IEEE Transactions on Computers*, Vol. C–32, No. 2, pp. 108–120.

[53] Mazer E. (1984), "LM–GEO: Geometric Programming of Assembly Robots", *Advanced Software in Robotics*, A. Danthine and M. Geradin (eds.), Elsevier Science Publisher B. V. (North–Holland), pp. 99–110.

[54] Miller J. M. and Hoffman R. L. (1989), "Automatic Assembly Planning With Fasteners", *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, pp. 69–74.

[55] Miller J. M. and Stockman G. C. (1990), "On the Number of Linear Extensions in a Precedence Graph", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 2136–2141.

[56] Nilsson N. (1980), "Principles of Artificial Intelligence", *Springer–Verlag.*

[57] Nnaji B. O. (1988), "CAD–BAsed Schema for an Assembly Planning Reasoner", *EXPERT SYSTEMS, Strategies and Solutions in Manufacturing Design and Planning*, Andrew Kusiak (editor), SME Publications Development Department, pp. 215–255.

[58] Nnaji B. O., Chu J.–Y. and Akrep M. (1988), "A Schema for CAD–Based Robot Assembly Task Planning for CSG–Modelled Objects", *Journal of Manufacturing Systems*, Vol. 7, No. 2, pp. 131–145.

[59] Nnaji B. O. and Liu H.–C. (1990), "Feature Reasoning for Automatic Assembly and Machining in Polyhedral Representation", *International Journal of Production Research*, Vol. 28, No. 3, pp. 517–540.

[60] Popplestone R. J., Ambler A. P. and Bellos I. M. (1978), "RAPT: A Language for Describing Assemblies", *The Industrial Robot*, September issue.

[61] Popplestone R. J., Liu Y. and Weiss R. (1990), "A Group Theoritic Approach to Assembly Planning", *AI Magazine*, Vol. 11, No. 1, pp. 82–97.

[62] Preiss K. and Shai O. (1989), "Process Planning by Logic Programming", *Robotics and Computer–Integrated Manufacturing*, Vol. 5, No. 1, pp. 1–10.

[63] Rich E. (1983), "Artificial Intelligence", *McGraw–Hill.*

[64] Rocheleau D. N. and Lee K. (1987), "System for Interactive Assembly Modelling", *Computer Aided Design*, Vol. 19, No. 2, pp. 65–72.

[65] Rondeau J.–M. and ElMaraghy H.A. (1989), "Development of a Knowledge–Based Robot Task Plannng System for Mechanical Assembly", *Proceedings of 1st ASME International Conference in Flexible Assembly*, Montréal, Canada, pp. 23–30.

[66] Rosario L. M. (1990), "Design for Assembly Analysis: Extraction of Geometric Features from a CAD System Data Base", *The Annals of CIRP*, Vol. 35, no. 1.

[67] Sanderson A. C., Homem De mello L. S. and Zhang H. (1990), "Assembly Sequence Planning", *AI magazine*, Vol. 11, No. 1, pp. 62–81.

[68] Sedas S. W. and Talukdar S. N. (1987), "A Disassembly Planner for Redesign", *Intelligent and Integrated Manufacturing Analysis and Synthesis*, The Winter Annual Meeting of ASME, Boston, Massachusets, pp. 95–100.

[69] Sekiguchi H. et al. (1983), "Study on Automatic Determination of Assembly Sequence", *The Annals of CIRP*, Vol. 32, no. 1, pp. 371–374.

[70] Shpitalni M., Elber G. and Lenz E. (1989), "Automatic Assembly of Three Dimensional Structures via Connectivity Graphs", *The Annals of CIRP*, Vol. 38, no. 1, pp. 25–28.

[71] Smith S. F., Keng N. and Kempf K. (1990), "Exploiting Local Flexibility During Execution of Pre–Computed Schedules", *Report no. CMU–RI–TR–90–13*, The Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

[72] Tönshoff H. K. and Anders N. (1989), "FLEXPLAN – A Concept for Intelligent Process Planning and Scheduling", *CIRP International Workshop on Computer Aided Process Planning (CAPP), State–of–the–Art, Future Directions, New Tools*, Hanover University, Federal Republic of Germany, September 21–22, pp 87–106.

[73] Watabe H. et al. (1989), "Inteference Recognition Among 3D Solid Models for Assembly Planning", *Proceedings of 1st ASME International Conference in Flexible Assembly*, Montréal, Canada, pp. 79–84.

[74] Wesley M. A. et al. (1980), "A Geometric Modelling System for Automated Mechanical Assembly", *IBM Journal of Research and Development*, Vol. 24, January issue, pp. 64–74.

[75] Weule H. and Friedmann Th. (1989), "Computer–Aided Product Analysis in Assembly Planning", *The Annals of CIRP*, Vol. 38, no. 1, pp. 1–4.

[76] Whitney D. E. et al. (1988), "Computer Aided Design of Flexible Assembly Systems", *Report No. CSDL R–2033*, C. S. Draper Laboratory Inc., Cambridge, Massachusetts.

[77] Wilkins D. E. (1988), "Practical Planning: Extending the Classical AI Planning Paradigm", *Morgan Kaufmann Publishers.*

[78] Wilson R. H. and Rit J.–F. (1990), "Maintaining Geometric Dependencies in an Assembly Planner", *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinnati, Ohio, pp. 890–895.

[79] Wolter J. D. (1988), "On the Automatic Generation of Plans for Mechanical Assembly", *Ph.D. thesis*, University of Michigan, Department of Computer, Information and Control Engineering.

[80] Wolter J. D. (1989), "On the Automatic Generation of Assembly Plans", *Proceedings of IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona, pp. 62–68.

[81] Wolter J. D. (1991), "A Combinatorial Analysis of Enumerative Data Structures for Assembly Planning", *Proceedings of IEEE International Conference on Robotics and Automation*, Sacramento, California, pp. 611–618.

[82] Zhang, W. (1989), "Representation of Assembly and Automatic Robot Planning by Petri Net", *IEEE Transactions on systems, man, and cybemetics*, Vol. 19, No. 2, pp. 418–422.

[83] Zozaya–Gorostiza C., Hendrickson C. and Rehak D. R. (1989), "Knowledge–Based Process Planning for Construction and Manufacturing", *Academic Press.*

[84] Zussman E., Lenz E. and Shpitalni M. (1990), "An Approach to the Automatic Assembly Planning Problem", *The Annals of CIRP,* Vol. 39, no. 1, pp. 33–36.

APPENDIX A

PRODUCT DESCRIPTION FILES

This appendix shows the Product Description Files (PDF) that were manually generated for some of the products used as examples throughout this thesis. The first section presents the air cylinder's product description file and describes how this file was manually generated. The second section describes how GAPP sets up the internal graph model of the product from an interpretation of the product description files. Sections A.3, A.4, A.5, and A.6 simply present the PDF of the flashlight, ball–point pen, multiple products and heat detector products, respectively.

A.1 THE AIR CYLINDER

A.1.1 Overview of the file format

The format of the product file is very simple. Each line in the file is either a *comment* line, a *component* line or a *relation* line. A component line starts with the keyword "component" and has 1 argument after the keyword: the component's name. A relation line starts with the keyword "relation" and has up to 9 arguments after the keyword. They are, in the required order of appearance: goal flag, moved component name, fixed component name, ambiguity flag, moved component's disassembly directions, fixed component's disassembly difrections, relation's type, operation required to establish the relation, list of restricted components. Any other line is treated as a comment line.

Figure A.1 shows the product description file that was manually generated for the air cylinder in figure 3.8. In this file, lines 1, 2, and 13 are comment lines. Lines 3 to 12 are component lines. Lines 14 to 36 are relation lines[57].

Two simple rules apply in the generation of the product file:

1– components must be defined before any of the relations in which they appear;

2– the arguments of a relation line must be supplied in the pre–defined order.

On the other hand, the order of definition of the components and relations is irrelevant, as long as the first rule above is satisfied. For example, any permutation of lines 3 to 12 and 14 to 36 in figure A.1 is permitted. In general however, the definition of both the components and the relations should follow some systematic order for better clarity, as illustrated for the air cylinder in figure A.1.

At present, the necessary information about the product is entered in the file by the user in the pre–determined format. The file is then read and interpreted by GAPP and the internal graph model of the product is set up accordingly.

The simple format of the product file makes it very easy for a GAPP user to create the file manually, even for complex products. All that is required is the identification of the various components in the product and the various relations between them. The identification of the product's components has already been discussed at the end of the third chapter. The next section discusses the identification of the relations among components in more detail.

57. Line numbers have been provided for reference only. They are not part of the actual file format.

```
1– AIR CYLINDER info
2–
3– component bearing_o_ring
4– component bearing
5– component body
6– component piston_rod
7– component piston
8– component piston_screw
9– component piston_o_ring
10– component cover_o_ring
11– component cover
12– component cover_screws
13–
14– relation no bearing_o_ring bearing no z+z– z–z+ attach fit piston_rod body
15– relation no bearing_o_ring piston_rod no z+z– z–z+ contact fit
16– relation no bearing_o_ring piston yes z+ z– blocking
17– relation no bearing_o_ring cover yes z+ z– blocking
18– relation no bearing body no z+z– z–z+ attach fit piston_rod
19– relation no bearing piston_rod yes z+z– z–z+ contact fit
20– relation no bearing piston yes z+ z– blocking
21– relation no bearing cover yes z+ z– blocking
22– relation no piston_o_ring body no z– z+ contact fit
23– relation no piston body no z– z+ contact fit
24– relation no cover body no z–y+y–x+x– z+y+y–x+x–contact against
      cover_screws
25– relation no cover_screws body no z–y+y–x+x– z+y+y–x+x– attach screw
26– relation no cover_o_ring body no z– z+ contact fit
27– relation no piston_rod piston yes z+ z– contact against piston_screw
28– relation no piston_screw piston_rod no z– z+ attach screw cover
29– relation no piston_rod cover yes z+ z– blocking
30– relation no piston_screw piston no z– z+ contact fit
31– relation no piston_o_ring piston no z+z– z–z+ attach fit body
32– relation no cover piston yes z–x+x–y+y– z+x+x–y+y– blocking
33– relation no piston_screw cover yes z+ z– blocking
34– relation no piston_o_ring cover yes z+y+y–x+x– z–y+y–x+x– blocking
35– relation no cover_o_ring cover no z+x+x–y+y– z–x+x–y+y– contact fit
36– relation no cover_screws cover no z– z+ contact fit
```

Fig. A.1: Product description file generated for the air cylinder.

A.1.2 Identifying relations

The identification of the relations among the components in the product is a fairly straightforward process. The general methodology is best described in the following algorithm:

```
1– Examine the product in its assembled state
2– Select one component (call it SOURCE)
3– For all other components :
    Determine if component SOURCE has a contact, attachment or
    blocking with another component (call it DEST.), according to the
    definitions in section 3.1
    If there is no relation between SOURCE and DEST. or if the relation
    already exists[58], pick another DEST. component
    Else define a new relation between SOURCE and DEST.
```

Mathematically, for a product of "n" components, this algorithm is known to require n (n – 1) / 2 steps. In practice, the user would be tempted to take some short cuts. For example, considering the air cylinder, if the component selected in step 2 is the cover_screws, it would not seem logical to look for relations between it and "remote" components, such as the bearing or the bearing_o_ring. Line 3 of the above algorithm could then be modified as "for all nearby components". But the danger of overlooking some relations introduced by this simplification greatly outweighs its benefit. In fact, experience with the system has shown that the above algorithm should be thoroughly followed, even if it is sometimes obvious that two remote components are not related.

58. The relation between SOURCE and DEST. might have already been defined if DEST. had previously been processed as a SOURCE component.

Once a relation has been identified using the above algorithm, its various attributes must be determined, in the pre–determined fixed order. First, the goal attribute must be set. Recall that a value of "yes" for this attribute means that the corresponding relation to which it applies will remain in the goal state, i.e. it will not be broken by any of the disassembly operations generated by GAPP. For the file in figure A.1, it was assumed that the air cylinder had to be completely disassembled, that is all relations have their goal flag set to the value "no".

Next, the moved and fixed components must be selected. For most relations, a decision can be taken as to which component in the relation should be moved and which should be fixed, if the product consisted solely of these two components. For example, it is obvious that the bearing_o_ring should be moved and assembled to the bearing and not vice–versa. In line 14 of figure A.1, this is expressed by putting "bearing_o_ring" before "bearing", and by setting the ambiguity flag to "no". There are cases where no clear decision between the moved and fixed components can be made. For such ambiguous cases, simply set the ambiguity flag to "yes". The two components involved in the relation can then appear in any order in the relation line. It should be noted that a blocking relation is always ambiguous, as it does not imply a physical contact. The notion of which component should be moved or fixed in the case of a blocking relation becomes irrelevant.

Next, the disassembly directions of both the moved and fixed components in the relation must be determined. This implies a decision on the final orientation of the product, since different orientations lead to different directions. For the file in figure A.1, the directions were determined according to the orientation shown in figure 3.8.

The first set of disassembly directions corresponds to the first component that appears in the relation line and the second set to the second component, respectively. Note that the directions of both components are always opposite. For example, if one component can be disassembled along z+, the other component must necessarily have z– as one of its disassembly directions.

The disassembly directions are determined locally, by first isolating the two components in the relation from the rest of the product and maintaining their relative position and orientation. Then, all the possible directions in which one component can be moved without colliding with the other are determined. These directions can be any combination of the elements in the set {x+, x–, y+, y–, z+, z–}.

Another attribute that must be set for a relation is its type (contact, attachment or blocking). Assigning the wrong type to a relation has some effects on the assembly sequence(s) being generated. It is therefore imperative for the user to have a good understanding of the differences between the three types. The definitions and examples in section 3.1 are believed to be clear enough to avoid any kind of confusion.

The type of the relation does not say much about the assembly operation that it involves. This is the purpose of the operation attribute. The value for any relation's operation is always the answer to the following question: what is the operation that must be performed in order to bring the two components of the relation to their assembled state, if the product consisted solely of these two components? For the products that have been worked on so far, the answer to that question is always one of the 9 pre–defined operations: against, fit, fit_and_twist, press, crimp, screw, rivet, weld, solder. This set is not exhaustive and can be extended to suit other products.

Note that it is irrelevant to specify an operation for a blocking relation (lines 16, 17, 20, 21, 29, 32, 33 and 34 in figure A.1).

The final attribute of a relation is actually an optional one. It consists of a list of restricted components which presence in either one of the two subassemblies involved in the establishment of a relation would complicate or even prevent the execution of the corresponding assembly operation. Lines 14, 18, 24, 27, 28 and 31 in figure A.1 are examples of relations with restricted components. Implications of this last attribute of a relation have been discussed at the end of chapter 5.

A.2 INTERPRETATION OF THE INPUT FILE BY GAPP

A.2.1 Setting up the graph model

In order to create the graph model, GAPP reads the product file sequentially, one line at a time. For lines which start with the keyword "component", GAPP allocates memory for a new component object (call it c) and adds it to the set C of D. Then it reads the first and only argument after the keyword[59] (call it arg1) and applies the function:

$$name(c) = arg1.$$

For lines which start with the keyword "relation", GAPP allocates memory for a new relation object (call it r) and adds it to the set R' of D. Then it reads the 9 arguments after the keyword[60] (call them arg1 to arg9, respectively) and applies the following functions:

59. Recall that this argument is the component's name.

60. Recall that these arguments are, in order: goal flag, moved component name, fixed component name, ambiguity flag, moved component's disassembly directions, fixed component's disassembly directions, relation's type, operation required to establish the relation, list of restricted components.

goal(r) = arg1

moved(r) = arg2

fixed(r) = arg3

ambiguous(r) = arg4

direction(r, arg2) = arg5

direction(r, arg3) = arg6

type(r) = arg7

operation(r) = arg8

restricted(r) = arg9.

A.2.2 Error checking

Some checks are performed to ensure that the file read by GAPP is consistent. A first of these checks ensures that the names of the moved and fixed components of a relation correspond to the names of components defined earlier in the file. Another check is concerned with the values of the goal(), ambiguity(), directions(), type() and operation() functions. In particular, any of the values for these functions must be one of those included in their respective pre–defined range (section 3.1). The optional list of restricted components is also checked, i.e. the names supplied for these components must match those of existing components defined earlier in the file. Finally, there must be at least 1 component and 1 relation defined in the file. If any of those restrictions is violated, a pop–up error message that describes the erroneous situation is displayed to the user.

A.3 PDF OF THE FLASHLIGHT

```
component head
component lens
component reflector
component bulb
component battery1
component battery2
component body
component spring
component endcap

relation no lens head no z+ z– attach fit
relation no lens bulb yes z–x+x–y+y– z+x+x–y+y– blocking
relation no lens reflector yes z–x+x–y+y– z+x+x–y+y– blocking
relation no lens body yes z–x+x–y+y– z+x+x–y+y– blocking
relation no lens battery1 yes z–x+x–y+y– z+x+x–y+y– blocking
relation no lens battery2 yes z–x+x–y+y– z+x+x–y+y– blocking
relation no lens spring yes z–x+x–y+y– z+x+x–y+y– blocking
relation no lens endcap yes z–x+x–y+y– z+x+x–y+y– blocking
relation no bulb reflector no z– z+ attach screw
relation no bulb battery1 yes z–x+x–y+y– z+x+x–y+y– contact against
relation no reflector head no z+ z– attach screw body
relation no reflector body yes z–x+x–y+y– z+x+x–y+y– blocking
relation no reflector battery1 yes z–x+x–y+y– z+x+x–y+y– blocking
relation no reflector battery2 yes z–x+x–y+y– z+x+x–y+y– blocking
relation no reflector spring yes z–x+x–y+y– z+x+x–y+y– blocking
relation no reflector endcap yes z–x+x–y+y– z+x+x–y+y– blocking
relation no head body yes z– z+ attach screw
relation no battery1 body no z+z– z+z– contact fit
relation no battery1 battery2 yes z–x+x–y+y– z+x+x–y+y– contact against
relation no battery2 body no z+z– z+z– contact fit
relation no spring battery2 yes z+x+x–y+y– z–x+x–y+y– contact against
relation no spring endcap no z– z+ attach fit_and_twist body
relation no spring body no z+ z– blocking
relation no endcap body yes z+ z– attach screw
```

Fig. A.2: Product description file generated for the flashlight.

A.4 PDF OF THE BALL–POINT PEN

```
component cap
component head
component body
component tube
component ink
component button

relation no ink tube no y+ y– contact fit
relation no cap body yes y– y+ attach fit
relation no tube head no y+ y– attach fit ink
relation no cap head yes y– y+ blocking
relation no head body yes y– y+ attach fit
relation no button tube yes y+ y– blocking
relation no button ink yes y+ y– blocking
relation no button body no y+ y– attach fit
```

Fig. A.3: Product description file generated for the ball point pen.

A.5 PDF OF THE MULTIPLE PRODUCTS

```
component plate1
component block1
component screw1
component screw2

relation no block1 plate1 no z+x+x–y+y– z–x+x–y+y– contact against
relation no screw1 plate1 no z+ z– attach screw
relation no screw2 plate1 no z+ z– attach screw

component plate2
component block2
component screw3
component screw4

relation no block2 plate2 no z+x+x–y+y– z–x+x–y+y– contact against
relation no screw3 plate2 no z+ z– attach screw
relation no screw4 plate2 no z+ z– attach screw
```

Fig. A.4: Product description file generated for the multiple products.

A.6 PDF OF THE BASE SUBASSEMBLY OF A HEAT DETECTOR DEVICE

```
component gasket
component diaphragm
component top_spring
component bottom_spring
component bus_bar
component rivet1
component rivet2
component base
component terminal_bar1
component terminal_bar2
component lock_spring
component center_plug
component cal_screw
component dust_cap1
component felt
component vent_screw
component dust_cap2

relation no gasket diaphragm yes z+ z– attach press
relation no diaphragm base no z+x+x–y+y– z–x+x–y+y– attach press gasket

relation no bus_bar base no z+ z– contact fit rivet1 bottom_spring top_spring
relation no bottom_spring bus_bar no z+ z– contact against rivet1
relation no top_spring base no z+ z– contact against rivet2

relation no bottom_spring diaphragm yes z– z+ blocking
relation no top_spring diaphragm yes z– z+ blocking
relation no bus_bar diaphragm yes z– z+ blocking
relation no rivet1 diaphragm yes z– z+ blocking
relation no rivet2 diaphragm yes z– z+ blocking
```

Fig. A.5: Product description file generated for the base subassembly of a heat detector device.

```
relation no rivet1 bottom_spring no z+ z– contact against
relation no rivet1 terminal_bar1 no z+ z– attach rivet
relation no rivet2 top_spring no z+ z– contact fit
relation no rivet2 terminal_bar2 no z+ z– attach rivet

relation no lock_spring base no z– z+ contact fit center_plug cal_screw
relation no center_plug base no z– z+ contact fit
relation no center_plug cal_screw no z+ z– attach fit lock_spring
relation no cal_screw base no z– z+ attach screw
relation no cal_screw bus_bar yes z– z+ contact against
relation no dust_cap1 base no z– z+ attach fit
relation no cal_screw dust_cap1 yes z+ z– blocking
relation no center_plug dust_cap1 yes z+ z– blocking
relation no lock_spring dust_cap1 yes z+ z– blocking

relation no felt base no z– z+ attach fit vent_screw
relation no vent_screw felt yes z– z+ contact against
relation no vent_screw base no z– z+ attach screw
relation no dust_cap2 base no z– z+ attach fit
relation no dust_cap2 vent_screw yes z– z+ blocking
relation no dust_cap2 felt yes z– z+ blocking

relation no terminal_bar1 base no z– z+ contact against rivet1
relation no terminal_bar2 base no z– z+ contact against rivet2
```

Fig. A.5 (continued).

APPENDIX B

KRUSKAL'S ALGORITHM

This appendix describes Kruskal's algorithm [2], which is a standard graph theoretic algorithm used to select a minimum spanning tree in a graph. The first section presents Kruskal's algorithm. The second section presents an example of its application.

B.1 THE ORIGINAL ALGORITHM

Assume a graph G exists which consists of a set $V = \{ v_1, v_2, ..., v_m \}$ of vertices and a set $E = \{ e_1, e_2, ..., e_n \}$ of edges. Also assume that some function $w(v_i, v_j)$ exists which can compute the weight of the edge between vertices v_i and v_j. By applying this function to every edge in the graph, G becomes a weighted graph.

Many real-life problems translate into the selection of a minimum spanning tree in a weighted graph, i.e. a spanning tree which edges have the lowest possible weight. Kruskal's algorithm was developed for this particular purpose. The algorithm is as follows:

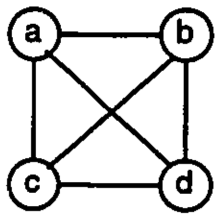
```
1– R = {}, S = {}.
2– Assign each vertex of the graph with a different number.
3– Choose an edge $e_i$ of E not already in R or S with smallest $w(e_i)$;
4– If $e_i$ is such that $G[S + e_i]$ is acyclic:
        add $e_i$ to the set S of selected edges.
        Assign both vertices of this edge with the larger number of the two and
        propagate this larger value to all connected vertices in S.
   else:
        add $e_i$ to the set R of rejected edges.
5– Repeat steps 3 and 4 and stop when they cannot be implemented further.
```

Fig. B.1: Kruskal's algorithm.

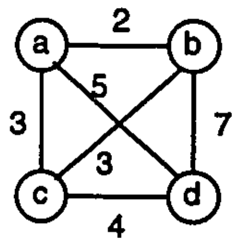
In the fourth step, one must determine if the edge e_i which has the smallest weight of all edges, not already selected in S or R, can be added to the spanning tree without generating any cycles. To check for such a condition, the numbers of both incident vertices of this edge are compared. If these numbers are the same, then the addition of this edge to the building spanning tree will generate a cycle and therefore this edge must be rejected. This should become clearer from the example in the following section.

B.2 EXAMPLE

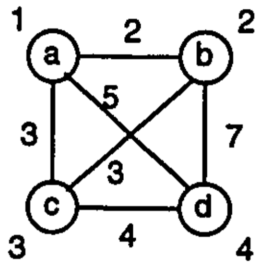
The graph in figure B.2a will be used to illustrate the use of Kruskal's algorithm. First apply the weight function to every edge in this graph. In GAPP, given a product's graph model, this weight function returns a number proportional to the strength of the connection between the two incident components of the corresponding relation. For example, a "screw" relation is given more weight than an "against" relation. Hypothetical weight assignments are shown in figure B.2b. Once this is done, a number is associated with each vertex in the graph. The numbers can hold any value, as long as they are different for every vertex. An arbitrary number assignment is shown in figure B.2c.



(a)



(b)



(c)

Fig. B.2: (a) some graph; (b) weight assignment to the edges of this graph; (c) number assignment to the vertices of this graph.

At this point, the graph can be processed through Kruskal's algorithm. In the first iteration, edge (a, b) is selected and added to S. Number 1 of vertex "a" becomes 2 and the new graph is as in figure B.3.

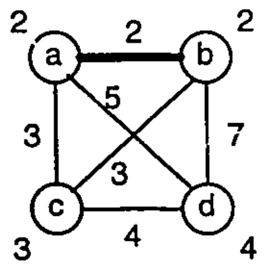


Fig. B.3: New graph after the first iteration.

In the second iteration, edge (a, c) is selected and added to S. Number 2 of vertex "a" becomes 3 and this new value is propagated to the incident vertices of all edges selected so far. This results in changing number 2 of vertex "b" to the value 3 as well. The new graph at this point is as in figure B.4.

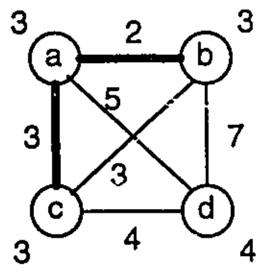


Fig. B.4: New graph after the second iteration.

In the third iteration, edge (b, c) is the one with lowest weight which is not already in S or R. However, the numbers of the incident vertices of this edge are the same, i.e. 3. This signals that an addition of this edge to the building spanning tree would result in a cycle, violating the definition of a spanning tree. This edge is therefore added to the set R of rejected edges.

In the fourth iteration, edge (c, d) is the one with lowest weight which is not already in S or R. Number 3 of vertex "c" becomes 4 and this new value is propagated to the incident vertices of all edges selected so far. This results in changing number 3 of vertices "a" and "b" to the value 4 as well. The new graph at this point is as in figure B.5.

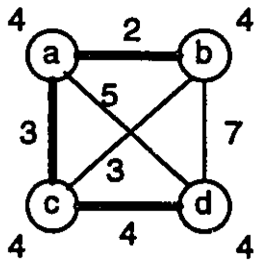


Fig. B.5: New graph after the fourth iteration.

The next two iterations of Kruskal's algorithm will process remaining edges (a, d) and (b, d). Both will fail the conditions for being added to the spanning tree and will therefore be added to the set R. The set S contains the edges which constitute the maximal spanning tree of the graph. This spanning tree corresponds to the bold edges in figure B.5.

APPENDIX C

WINDOW INTERFACE

Starting the execution of GAPP brings up five windows on the computer screen, as depicted by figure C.1. A temporary window also gets displayed while GAPP is searching for an optimal solution. Each window and their functionalities are described individually in the following sections.

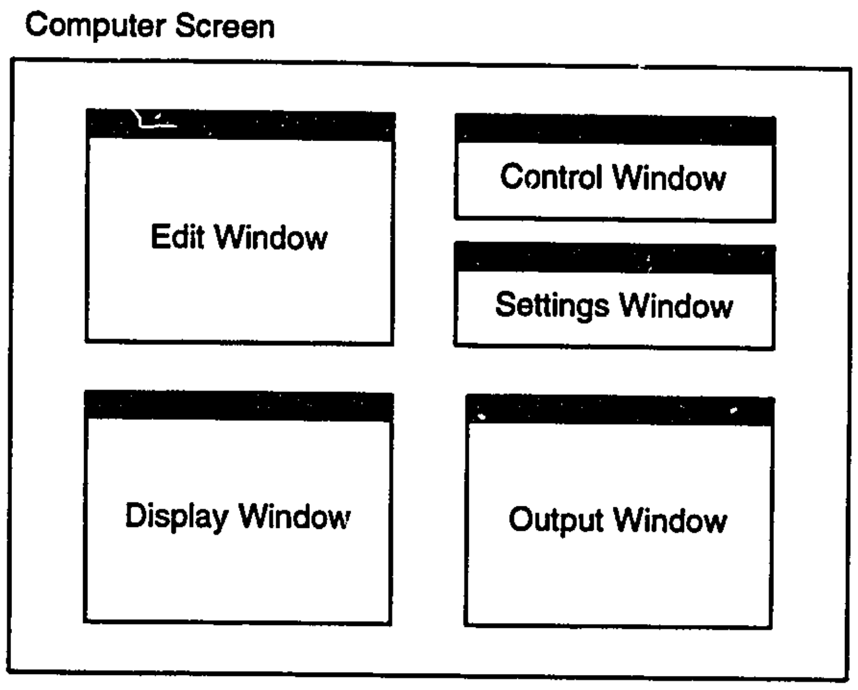


Fig. C.1: Default position of GAPP's windows on screen.

C.1 CONTROL WINDOW

The detailed control window is shown in figure C.2. When running GAPP, it appears in the upper right corner of the screen. This window performs two main functions:

1– it controls the visibility of all other windows in the interface, and

2– it controls the interactions between the user and GAPP (what the user wants to do and when).

By default, all windows in figure C.1 are visible when GAPP is first invoked. Clicking left in the square boxes in the line labelled "visibility" changes the visibility (on or off) of the corresponding window. Of course, the visibility of the control window itself cannot be turned off.

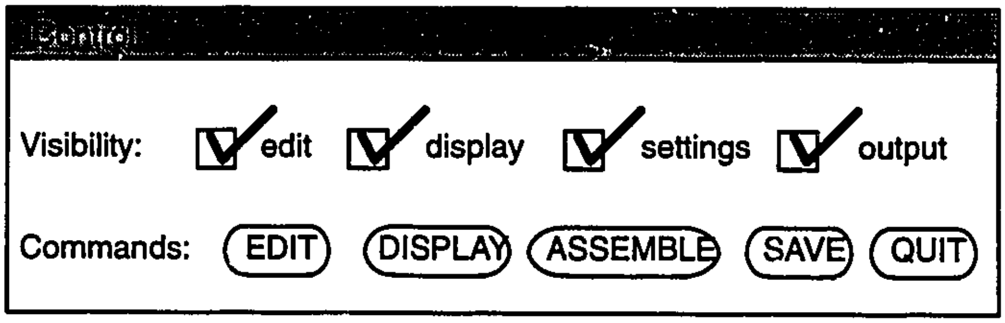
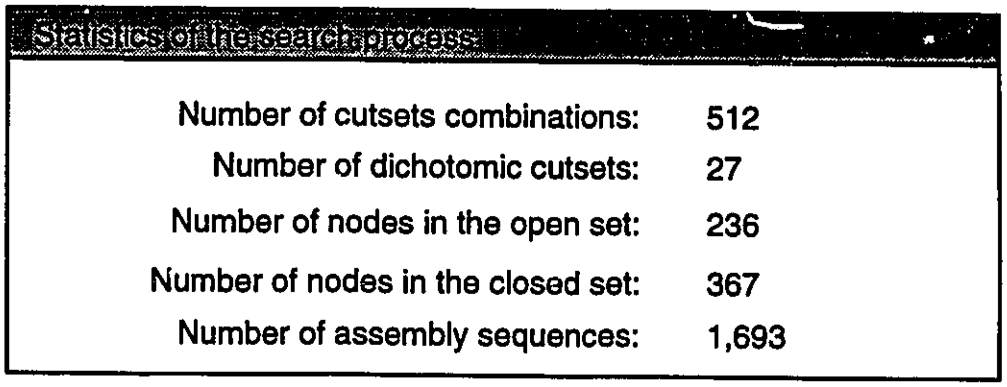


Fig. C.2: The control window.

Clicking left on the buttons in the line labelled "commands" starts the execution of the process labelled on the button. The EDIT button is used to edit a file in the edit window. The file that gets edited is the one in the line labelled "product file" in the settings window (figure C.4). The DISPLAY button is used to display a raster file, usually a geometric model of the product to be assembled, in the display window. The file that gets displayed is the one in the line labelled "display file" in the settings window.

The ASSEMBLE button starts the execution of the search process. The product's directed graph of assembly states gets expanded until a solution is found. Statistics of the search process also get displayed in a temporary pop–up window (figure C.3). Once the search process is activated using this button, no other commands will be accepted in the window interface until a solution is found. The pop–up window disappears when the search process is terminated.



```
Statistics of the search process

     Number of cutsets combinations:   512
        Number of dichotomic cutsets:   27
      Number of nodes in the open set:   236
    Number of nodes in the closed set:   367
        Number of assembly sequences:   1,693
```

Fig. C.3: The pop–up window displaying statistics of the search process.

The SAVE button saves all textual output which appears in the output window in a file. The file which receives this output is the one in the line labelled "output file" in the parameters setting window. If the file already exists, a pop–up window asks for a permission to overwrite the existing file. Finally, the QUIT button stops GAPP and terminates all processes associated with GAPP's windowing interface.

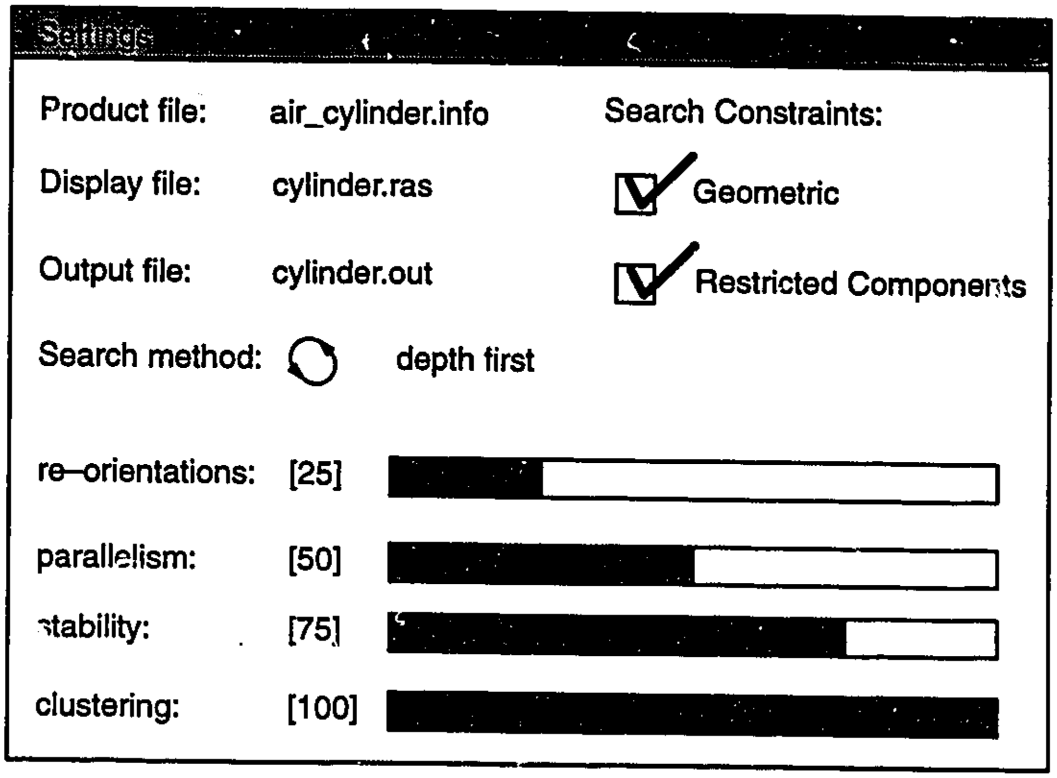
C.2 PARAMETERS SETTING WINDOW

The detailed parameters setting window is shown in figure C.4. When running GAPP, it appears in the middle right corner of the screen, below the control window. This window is first used to specify the file names required by the EDIT, DISPLAY and SAVE buttons of the control window. Most importantly, it is also used to set some important parameters which directly influence the search process and the solutions generated by GAPP.

The line labelled "product file" in this window is used to specify the name of the product description file. When GAPP is first invoked, the blinking caret is already at the beginning of the field used for typing this file name. If a name must be typed with the caret positioned somewhere else in the window, left clicking over this field will bring the caret back. Pressing the return key successively will also cycle through the various text fields in this window. A typed file name must exist in the directory from which GAPP was invoked and cannot have more than 32 characters. Upon left clicking on the EDIT button, this file can be edited in the edit window. Upon left clicking on the ASSEMBLE button, this file will get interpreted by GAPP and the internal graph model of the product will be set up accordingly.

The line labelled "display file" is used to specify the name of a raster file containing the geometric model of some product. The blinking caret can be positioned over this field as described above.

The line labelled "output file" is used to specify the name of a file which will receive all the textual information displayed in the output window throughout a consulting session.


Settings

Product file: air_cylinder.info

Display file: cylinder.ras

Output file: cylinder.out

Search method: depth first

Search Constraints:

✓ Geometric

✓ Restricted Components

re–orientations: [25]

parallelism: [50]

stability: [75]

clustering: [100]

Fig. C.4: The settings window.

The line labelled "search method" is used to select one of five search methods available in GAPP. These methods determine the way GAPP sails through the directed graph of assembly states in order to find a solution. They are:

1) breadth first,

2) depth first,

3) best first,

4) A^*, and

5) hill climbing.

The first two methods ensure solution's optimality but are exhaustive. The next two also ensure solution's optimality without necessarily expanding the whole search graph. The last one always finds a good but not necessarily optimal solution in polynomial time. The choice of any of the methods is performed by left clicking on the cycle symbol (little circle with arrows) until the desired one appears on the left hand side.

The line labelled "search constraints" is used to turn search constraints on and off, by left clicking in the corresponding square box. These constraints and the search space reductions they may imply were fully described in chapter 5.

The lines labelled "re–orientations", "parallelism", stability" and "clustering" are used to specify the relative weight of the corresponding criteria. A criterion which relative weight has been set to zero is not considered in the search process. The criterion with the largest weight is the one which GAPP tries to optimize the most.

C.3 EDIT WINDOW

The edit window is actually a shell like any other shells in Suntools. This shell's default directory is the one from which GAPP was first invoked. By positioning the mouse over this window, the user can execute any shell commands, like changing the directory, executing programs, editing files, and so on. This window is called the "edit window" simply because the file at the text caret in the settings window gets automatically edited under "vi" in this window when the EDIT button of the control window is clicked. For example, if the text caret is at the "product file" line of the settings window, left clicking the EDIT button of the control window edits the product description file in the edit window.

C.4 DISPLAY WINDOW

The display window is used to display the raster file specified at the "display file" line of the settings window. The display occurs when the DISPLAY button of the control window is clicked. The display window has been provided with both horizontal and vertical scrolling capabilities.

C.5 OUTPUT WINDOW

When GAPP reaches the solution to some problem, this window displays the results of the search process. These include overall statistics of the search process and the solution itself. Every line of the solution has the following format:
operation (moved subassembly) (fixed subassembly) from disassembly_directions

If all ambiguity flags have been set to the value "yes", the order of appearance of both subassemblies in the line becomes irrelevant. Also, no disassembly directions are returned when the geometric feasibility constraint is turned off.

An output obtained using the air cylinder is shown in figure C.5. The first line specifies the name of the product description file from which GAPP constructed the graph model. The second line specifies the search method used. The third line specifies the constraints that were used. The fourth line specifies the relative weight of the criteria. The fifth, sixth and seventh line are statistics about the directed graph of assembly states that was expanded. The eighth line specifies the total unit cost of the optimal solution. The remaining lines represent the actual assembly sequence generated by GAPP.

```
Product file: air_cylinder.info
Search method: BREADTH FIRST SEARCH
Constraints: Geometric_feasibility Restricted_components
Criteria: re–orient=50 parallel=50 stability=50 clustering=50
NUMBER OF NODES ON OPEN: 0
NUMBER OF NODES ON CLOSED: 122
NUMBER OF ASSEMBLY SEQUENCES: 728
TOTAL COST: 553.10
against ( piston ) ( piston_rod ) from z+
screw ( piston_screw ) ( piston_rod piston ) from z+
fit ( bearing_o_ring ) ( bearing ) from z+z–
fit ( cover_o_ring ) ( body ) from z+
fit ( bearing bearing_o_ring ) ( body cover_o_ring ) from z+z–
fit ( piston_o_ring ) ( piston piston_rod piston_screw ) from z+z–
fit ( piston_o_ring piston piston_rod piston_screw ) ( body bearing bearing_o_ring
cover_o_ring ) from z+
fit ( cover ) ( bearing_o_ring bearing piston_rod body piston_o_ring piston
cover_o_ring piston_screw ) from z+
screw ( screws ) ( cover body bearing bearing_o_ring piston_rod piston_o_ring
piston cover_o_ring piston_screw ) from z+
RE–ORIENT 180 DEGREES
```

Fig. C.5: Air cylinder's solution in the output window.

APPENDIX D

DATA STRUCTURES

This appendix describes the C++ data structures on which GAPP is based. A first section describes the class "List". A second section describes the class "Item". Advantages of the developed data structures are presented in the third section.

D.1 THE CLASS "LIST"

Expanding the directed graph of assembly states and representing the actual subassemblies at each different state is all done by list processing. A general class called List has been defined as follows:

```
class List {
        public:         List();
                        List( Item* );
                        virtual ~List();
                        void add( Item* );
                        void remove( Item* );
                        int find( Item* );
                        void purge_primary();
                        void purge_all();

                        Item *head;

        };
```

This class is actually a superclass, meaning that its above definitions are inherited by four sub–classes, namely:

Cutset,

Subassembly,

Open, and

Closed.

A Cutset is a list of relations, a Subassembly is a list of components, Open and Closed are both lists of nodes of the search graph.

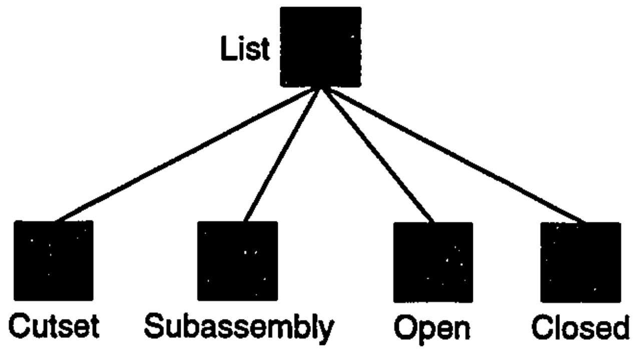
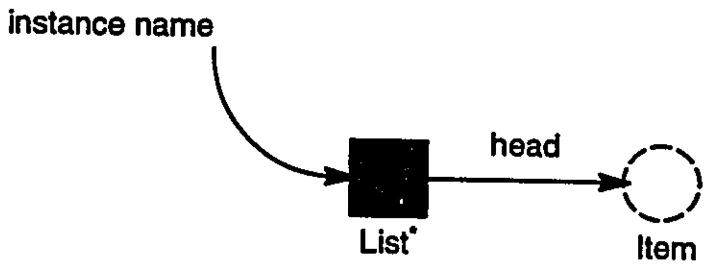


Fig. D.1: Hierarchy of list classes.

The member functions of the List superclass contain the usual definitions for the constructors and destructors. Various processing functions such as adding an item, removing an item, finding an item, etc. are also added.

The List superclass presents only one data member: "head". This member is of type Item, which is also a superclass (it will be defined in the next section). This data member is a pointer to the first item in the list. It is inherited by all sub–classes.

The schematic representation of any instance in the List hierarchy is as in figure D.2.



*can be any of List, Cutset, Subassembly, Open or Closed

Fig. D.2: Schematic representation of a List instance.

D.2 THE CLASS "ITEM"

General properties of lists are captured in a List superclass. Similarly, the general properties of the items in a list are captured in a second superclass called Item. Its definition is as follows:

```
class Item {
        public:             Item();
                            Item( Item* );
                            virtual ~Item();

                            Item *item;
                            Item *next;
        };
```

Three sub–classes inherit the general definitions of the Item superclass:

Component,

Relation, and

Node.

The only member functions in the Item superclass are the required constructors and the destructor.

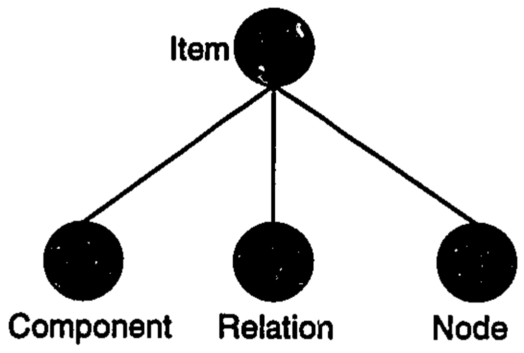
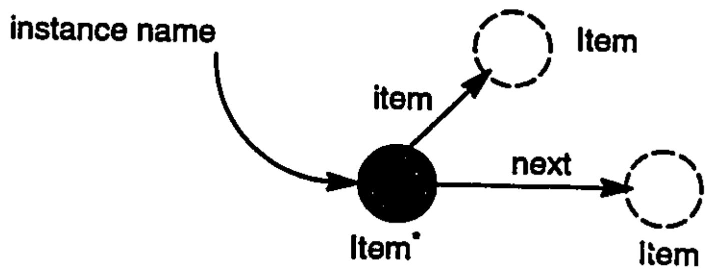


Fig. D.3: Hierarchy of item classes.

Two data members are also defined: "item" and "next". Both are of type Item. This provides the schematic representation in figure D.4 for any instance in the Item hierarchy.



*can be any of Item, Component, Relation, Node

Fig. D.4: Schematic representation of an Item instance.

D.3 ADVANTAGES

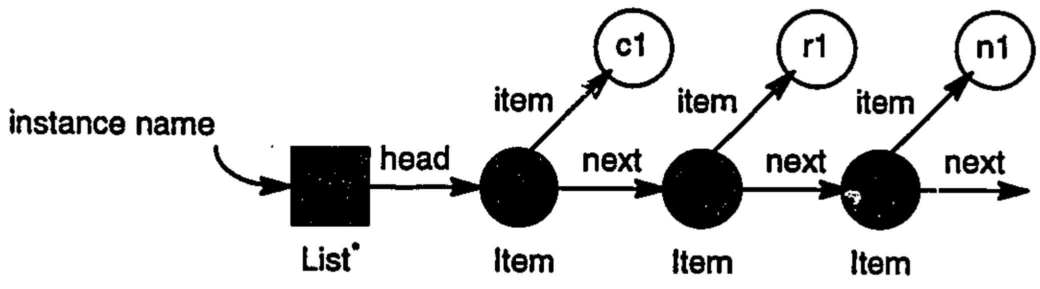
An important advantage of the above data structure design is that lists of heterogeneous objects can be built. That is, the head of any instance in the List hierarchy can point to any instance in the Item hierarchy. Thus if "c1" is a Component instance, "r1" a Relation instance and "n1" a node instance, one can have the list:

{ c1, r1, n1 }

which has the schematic representation shown in figure D.5.

A second advantage is related to the way that lists are set up. When GAPP reads the product description file, Component and Relation instances are constructed. The "item" and "next" data members of these instances, inherited from the Item superclass, are set to null. The same applies when a Node instance is constructed during the search graph expansion. Thus no instance in these three subclasses ever point to each other to form a list. Rather, each instance is added to a list through the "add" member function defined in the List superclass. This function constructs an instance of the Item superclass whose "item" data member points to the actual instance to be added in the list. As a result, the chaining through the head and next pointers in any list always passes through "dummy" instances of the Item superclass (figure D.5). The actual Component, Relation or Node instance that was added to the list can be reached through the "item" pointer of the dummy Item instance.

The rationale for such a design is as follows. For each node in the directed graph of assembly states, the required lists of relations and components must be generated. Because each node has different relations and components, such lists are different for each node. If lists had been built by making components and relations point to each other, duplication of all components and relations would have been required at each node, to correctly maintain the pointers among them. Due to the size of the directed graph of assembly states, such duplications would have resulted in serious memory limitations. One can now appreciate the advantage of inserting a "dummy" instance of the Item superclass to build a list, as it requires little memory and prevents duplication of the actual Component or Relation instances at each node.



*can be any of List, Cutset, Subassembly, Open or Closed

Fig. D.5: Schematic representation of an heterogeneous list.