A CELLULAR AUTOMATA REPRESENTATION FOR ASSEMBLY SIMULATION AND SEQUENCE GENERATION

by

Kontong Francisco Pahng

Bachelor of Science in Mechanical Engineering
The University of Iowa
May, 1993

Submitted to the Department of Mechanical Engineering
in Partial Fulfillment of the Requirements for the Degree of

Master of Science
in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Signature of Author _______________

Kontong Francisco Pahng
Department of Mechanical Engineering
June 5, 1995

Certified by _______________

Mark J. Jakiela
Associate Professor
Department of Mechanical Engineering
Thesis Supervisor

Accepted by _______________

Ain A. Sonin
Chairman, Departmental Graduate Committee

A CELLULAR AUTOMATA REPRESENTATION FOR ASSEMBLY SIMULATION AND SEQUENCE GENERATION

by

Kontong Francisco Pahng

Submitted to the Department of Mechanical Engineering
on June 5, 1995 in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Mechanical Engineering

ABSTRACT

Cellular automata are used to represent physical objects and their interactions for the purpose of planning the assembly process of several parts. Cells assume a vector value, made up of a material/void indicator, a set of cell boundary edge states, and an indicator of the allowable motion direction of the cell. Sets of rules that recognize parts and deduce allowable motions are described. The assembly planning is assumed to be the reverse process of its corresponding disassembly sequence and an algorithm to generate allowable disassembly sequences is developed using cellular automata. The rules and algorithms are implemented to study their performance, and several examples for the assembly sequence generation are presented. The Optimization of linear assembly sequences based on assemblability criteria is investigated using genetic algorithms and branch and bound search technique. The assemblability pertains to characteristics or attributes of an assembly sequence that can be either desirable or undesirable from a manufacturing standpoint.

Thesis Advisor: Professor Mark J. Jakiela

Acknowledgments

First of all, I would like to express my sincere gratitude to my academic adviser, professor Mark Jakiela, for his invaluable support and guidance throughout the development of this thesis as well as the research.

I would also like to thank Mr. Shunsuke Minami, formerly a visiting research engineer from the Hitachi Research Laboratory, for his professional insights and guidance along the implementation of our system.

My experience at MIT CADLAB was made particularly enjoyable by all the students and staff. I would like to thank all the people in CADLAB. Also, I cannot thank enough all the friends at MIT as well as here in Boston, in Iowa City, and far back in Korea for their support and friendship.

Finally, and most importantly, I would like to thank my family. I would like to thank my dear brother, sister, sister-in-law, and my beloved little nephew. Of course, the most special thanks go to my parents. Not only did they gave me a chance to see this wonderful world, but they also gave me the love and support to help me do my best.

Table of Contents

ABSTRACT ii
Acknowledgments iii
Table of Contents iv
List of Figures vii
List of Algorithms ix

Introduction 10
1.1 Motivation 10
1.2 Objective 11
1.3 Organization 12

Background 14
2.1 Overview 14
2.2 Geometric Representation 14
2.2.1 Boundary Representations (B-Reps) 14
2.2.2 Constructive Solid Geometry (CSG) 15
2.2.3 Spatial-Occupancy Enumeration 16
2.2.4 Comparison of Representations 17
2.2 Geometric Description in This Investigation 18
2.2.1 Cellular Automata 18
2.2.2 Cellular Automata for Physical Reasoning 19
2.3 Assembly Planning and Sequence Generation 19
2.4 Optimization Methods 23
2.4.1 Genetic Algorithms 23
2.4.2 Branch And Bound 24

Previous Work in Assembly Sequence Generation 26
3.1 Overview 26
3.2 Computer-Based Interactive Assembly Planning 26
3.2.1 Bourjault and Henrioud 27
3.2.2 De Fazio and Whitney 27
3.3 Computer-Automated Assembly Planning 28
3.3.1 Homen de Mello and Sanderson 28

3.3.2 Wilson 30
3.3.3 Wolter 31
3.3.4 Hoffman 31
3.3.5 S. Lee 31
3.4 Summary 32

Assembly Simulation With Cellular Automata 33
4.1 Overview 33
4.2 Simulation of Rigid Part Interactions 33
4.2.1 Cell State 33
4.2.2 Part Membership Rules 35
4.2.3 Part Motion Rules 36
Possible Motion Propagation Rules 37
Cell Movement Rules 40
4.2.4 Part Motion Procedure 43
4.2.5 Part Extraction Procedure 44
4.3 Disassembly Sequence Generation Algorithm 46
4.4 Examples 50

Assembly Sequence Optimization 57
5.1 Overview 57
5.2 Selecting Assembly Sequences 57
5.2.1 Introduction 57
5.2.2 Linear and Parallel Assembly Sequences 58
5.2.3 Assemblability Evaluation 58
5.2.4 Evaluation Criteria for Assembly Sequence 59
5.3 Genetic Algorithms Approach 61
5.3.1 Introduction 61
5.3.2 Implementation 61
5.3.3 GA Operators for Crossover and Mutation 62
Partial Matched Crossover and Ordered Crossover 62
Swap Mutation and Shift Mutation 64
5.3.4 Examples 65
5.4 Branch And Bound Approach 71
5.3.1 Introduction 71
5.3.2 Implementation 71

5.4.3 Examples....75

Conclusions....80

6.1 Overview....80

6.2 Capabilities of the Disassembly Planner....80

6.3 Optimization with Cellular Automata....82

6.4 Future work....83

References....85

List of Figures

Figure 2.1 An objective define by B-Reps. 15
Figure 2.2 A model defined by CSG and its tree. 16
Figure 2.3 A model define by Spatial-Occupancy Enumeration. 16
Figure 2.4 Possible transformation among geometric representations. 17
Figure 2.5 Cellular automata operation. 19
Figure 2.6 Example assembly 20
Figure 2.7 Directed graph of feasible assembly sequences 22
Figure 2.8 AND/OR Graph of feasible assembly sequences 23
Figure 2.9 Enumeration tree of branch and bound method. 25
Figure 3.1 A four-part simple product from (Homem de Mello and Sanderson 1990) 28
Figure 3.2 AND/OR graph of the simple product in figure 3.1 29
Figure 4.1 State vector of a cell. 34
Figure 4.2 Application of possible motion propagation rules. 40
Figure 4.3 Application of possible cell movement rules. 43
Figure 4.4 Example of allowable motion propagation and cell movement rules. 44
Figure 4.5 Example of a part extraction procedure. 45
Figure 4.6 Example of no possible part extraction. 46
Figure 4.7a Example assembly with four components 48
Figure 4.7b Feasible decomposition task for the given assembly 49
Figure 4.7c Feasible decomposition task for subassembly { 1, 3, 4 } 49
Figure 4.7d AND/OR graph for the example in figure 10a. 50
Figure 4.8a 4 part assembly in 10x10 domain. 51
Figure 4.8b 6 part assembly in 10x10 domain. 52
Figure 4.8c 8 part assembly in 10x10 domain. 52
Figure 4.8d CPU seconds versus part count 53
Figure 4.9a 4 part assembly in 15x15 domain. 54
Figure 4.9b 4 part assembly in 20x20 domain. 55
Figure 4.9c CPU seconds versus array size 56
Figure 5.1 Schematic representation of assembly sequences types 58
Figure 5.2 Partial matched crossover. 63
Figure 5.3 Ordered crossover 64
Figure 5.4 Swap and shift mutation 65

Figure 5.5 Example assembly: Test 1..66
Figure 5.6 Example assembly: Test 2..66
Figure 5.7 Example assembly: Test 3..67
Figure 5.8 Example assembly: Test 4..68
Figure 5.9 Population versus number of evaluation and fitness..............................69
Figure 5.10 Mutation probability vs. no. of eval. and fitness................................70
Figure 5.11 GA operators vs. no. of eval. and fitness.......................................71
Figure 5.12 Five part assembly...74
Figure 5.13 Branch and bound tree of the example assembly in figure 5.12.............74
Figure 5.14 Branch and bound tree of the example assembly in figure 5.12.............75
Figure 5.15 Search methods versus number of evaluation and fitness.....................76
Figure 5.16 Search methods versus number of evaluation and fitness.....................77
Figure 5.17 Example assembly: Horizontal stack..77
Figure 5.18 Example assembly: Horizontal-vertical stack.................................78
Figure 5.19 Example assembly: Vertical-horizontal stack.................................78
Figure 5.20 Example assembly: Vertical stack..79
Figure 6.1 A monotone assembly...81
Figure 6.2 A non-contact-coherent assembly..82

List of Algorithms

List 2.1 Procedure of the genetic algorithm. 24
List 4.1 Rules for membership.....35
List 4.2 Algorithm to determine membership. 36
List 4.3 Possible movement propagation rules.....37
List 4.4 Cell movement rules.....41
List 4.5 Algorithm to move parts. 43
List 4.6 Algorithm to generate disassembly tasks. 47
List 4.7 Algorithm to generate a AND/OR graph. 47
List 5.1 Algorithm for searching sequences using the genetic algorithm 61
List 5.2 Algorithm for evaluating a sequence 62
List 5.3 Algorithm to find the best sequence 72
List 5.4 Branch And Bound Algorithm. 72

1 Introduction

1.1 Motivation

It has been widely recognized that the final cost of a manufactured product is largely determined by the early design process, and thus, there is intense interest in developing design tools which will aid this process and make it more effective. For the manufactured product, an assembly-conscious design is desirable because it can generate significant savings in capital costs and assembly time, and thereby, a higher design quality and efficiency. The assembly process is defined as the putting together of various parts to create an end product and the term 'assembly,' in fact, designates both the process and sometimes the product (Delchambre 1992). However, assembly-conscious design is difficult because designers, in general, do not have sufficient knowledge relevant to an assembly process. Such knowledge includes information with regard to the assembly operations and planning (cost, difficulty of process, *etc.*) and design for assembly methodologies such as a classification of relevant design features, redesign techniques, experience of plant engineers, *etc.* (Boothroyd and Alting 1992). In recent years, therefore, the effective design of products to facilitate assembly and issues pertaining to assembly in general have been very active areas of research.

This research is concerned with the specific issue of *assembly planning* or *assembly sequence generation.* To understand the context of this issue, consider the mass production assembly of a product. This entire process can be sensibly divided into three phases. The first, known as part presentation, takes parts in some disorganized bulk form, separates, organizes, and orients them so that their location and orientation are known prior to assembling them to the product. Humans perform this task very well with sophisticated tactile and visual sensing capabilities; in automated settings, bowl feeders, tracks, hoppers, *etc.* are used. The third phase is the final insertion of a part into the assembly or subassembly. Again human sensory capabilities facilitate this task; advanced robotics and special purpose end effectors (*e.g.* the remote center compliance device (Nevins and Whitney 1978)) help for automated assembly.

The second phase, between part presentation and final insertion, involves the gross motion of parts between the part presentation point and the final insertion point. Humans and robots can both be used for this task. Important considerations with regard to this phase are the order in which parts are assembled into the product and the complexity of the motions that are required to move each part towards its final insertion point. The choice of a good assembly order can minimize the number of part movement operations, such as pick and place motions and inversions of subassemblies. Simplified part motions can be achieved with less sophisticated, and therefore less costly, machinery. It is clear, then, that a good *assembly plan,* the ordering of parts assembly and the design of the insertion motions, can have a large influence on the cost of an assembly process. In addition, the selection of an alternative assembly plan or sequence can have a large impact on nearly every aspect of the facility and assembly methods such as ability to automate, resource utilization, product quality, in-progress repair, assembly system configuration, and *etc.* (Klein 1987). In turn, the design of individual parts and the configuration of the entire assembly will dictate the assembly plans that are possible.

1.2 Objective

The purpose of this thesis is to present a computer-automated assembly planner which generates feasible assembly sequences using a geometric representation method based on cellular automata. Also, the optimization of linear assembly sequences is investigated with genetic algorithms and branch and bound search techniques in order to determine good sequences based on a predefined set of criteria.

The overall interest is in the total design of parts and assemblies to facilitate the creation of high quality assembly plans. In practice, the human or computational analysis of an assembly is time consuming and laborious. Analyzing one assembly of significant part count takes several minutes, requires some type of human analysis, and in general is a significant achievement. The process of iteratively modifying the assembly and its parts, and then regenerating an assembly plan, for the purpose of design optimization of the assembly, could not be practically achieved. Therefore, it is sought to develop a computer-based tool that will either fully automate this design process or significantly assist a human completing a product design. An automated assembly design process will require an automated assembly planning process. Some tools already exist (at least in research form) for all three phases of assembly. Boothroyd and associated researchers (Boothroyd, Poli et al. 1982) have distilled the results of many empirical studies (Boothroyd and Ho 1976) into charts that provide guidance on appropriate part shapes to facilitate part presentation and bulk handling. Jakiela and associated researchers have created a system that automatically evaluates a three-dimensional solid model with respect to the Boothroyd charts and provides design improvement suggestions. ((Jakiela, Papalambros et al. 1985) and (Jakiela, Papalambros et al. 1985)) Several researchers have proposed and implemented approaches to automate the generation of assembly plans, which will be discussed in chapter 3. This automatic generator of assembly sequences can be an efficient aid to the designer. Whenever he or she modifies features of the product, the influence of these modifications on the sequences can immediately be checked. This design iteration, which also considers assembly, is significant because the optimal design of individual components will not necessarily imply the optimal design of their assembly process.

As opposed to an accurate representation of part geometry, a representation was sought that allows very rapid geometric reasoning and facilitates iterative modification of part geometry, while still adequately representing the part geometry. The part representation can be more schematic in exchange for greater ease of modification and physical simulation. In this investigation, a cellular automata model of assemblies of rigid parts is proposed and demonstrated. This will be detailed in chapter 2. It will be seen that this representation allows rapid and general physical simulation at the expense of accurately modeling physical detail. Physically realistic CAD models are certainly important: their use is widespread and they facilitate the subsequent manufacture of the designed part (as they were originally intended to do). However, they are not ideally suited to the more qualitative reasoning that is required for assembly simulation and optimization. This system based on a cellular automata representation is perhaps best considered as a schematic preprocessor to the more exacting CAD modeling phase.

1.3 Organization

Chapter 2, Background, first introduces the conventional methods for representing solid geometry which can be utilized for the purpose of assembly sequences generation. Then, cellular automata ("CA") are introduced and it is explained how they can be used to simulate the interactions of rigid parts. A simple example is also provided to explain the concepts of generating assembly sequences. In chapter 3, Previous Work, the previous research in assembly sequence generation is reviewed briefly.

Chapter 4, Assembly Simulation with Cellular Automata, describes in detail the underlying rules and algorithms for the physical reasoning, which is later used for simulating the assembly process and generating assembly sequences. Examples are provided to show the performance of the implementation. In chapter 5, Assembly Sequence Optimization, the optimization of assembly sequences with genetic algorithms and branch and bound methods is explained with examples for each method.

Chapter 6, Conclusions, summarizes the research with some general conclusions and discusses recommendations for future research.

2 Background

2.1 Overview

This chapter first introduces three conventional methods for geometric representation which can be utilized for the purpose of assembly sequence generation. The properties of each representation method are discussed and compared based on important criteria. Cellular automata ("CA") are then introduced and it is described how CA can be used to simulate the interactions of rigid parts. The fundamental concept of generating assembly sequences is explained with a simple example. Finally, the optimization methods, which are used to search for optimal assembly sequences, are detailed.

2.2 Geometric Representation

There are three major methods for geometric representation in solid modeling that are able to provide the necessary information for an assembly planner (Delchambre 1992). These representations are boundary representations, constructive solid geometry, and spatial-occupancy enumeration. In the following sections, they are briefly described and compared so as to give the rationale for choosing a geometric representation in this investigation. Other common representation methods for modeling mechanical parts in the geometric database of CAD systems are wireframe models, parametrized shapes, sweep representations, and several others. For a comprehensive survey of these methods, please refer to (Foley 1993) or (Mortenson 1985).

2.2.1 Boundary Representations (B-Reps)

In B-Reps an object is described in terms of its surface boundaries: vertices, edges, and faces. These boundaries separate points inside from points outside the object or solid. Curved faces are generally approximated with polygons or represented with complex surfaces such as a Bézier surface which requires more information. Depending on the type of boundary data structure, B-Reps can be categorized into polygon-based boundary models, vertex-based boundary models, edge-based boundary models, *etc.* In general, B-Reps are useful for generating graphical output, because they readily include the data needed for driving a graphical display. It can be difficult to establish the validity of models, however, because boundary models are valid only if they define the boundary of a topologically and geometrically reasonable solid object. Figure 2.1 illustrates the basic component of a boundary model. Figure 2.1a shows an object whose surface is divided into an enclosing set of faces. These faces can be represented in terms of either bounding polygons as shown in figure 2.1b or edges and vertices as shown in figure 2.1c.

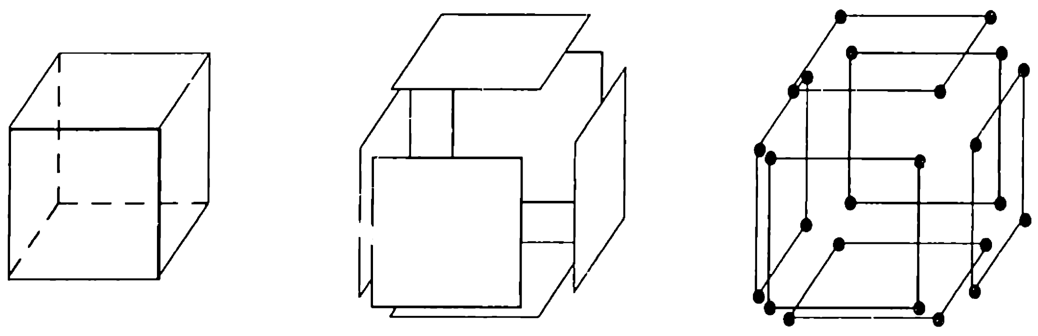


Figure 2.1 An objective define by B-Reps.

2.2.2 Constructive Solid Geometry (CSG)

CSG is a term for modeling methods that define an object by combining simple primitives or "building blocks." The user of a CSG modeler operates only on parametrized instances of these building blocks using regularized Boolean set operators that are included directly in the representation. A model is stored as a tree with operators at the internal nodes and simple primitives at the leaves. Figure 2.2 shows an example model defined by primitives such as cylinder and plate.

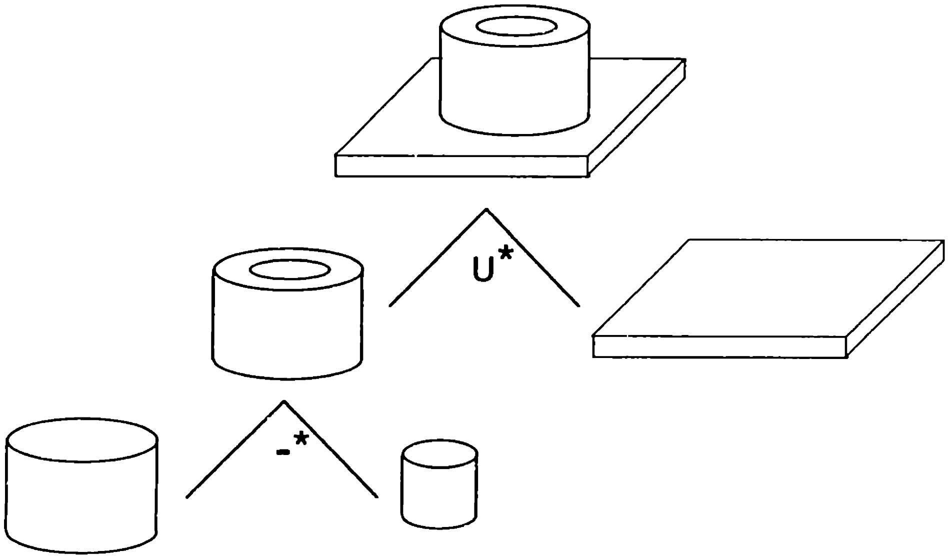


Figure 2.2 A model defined by CSG and its tree.

2.2.3 Spatial-Occupancy Enumeration

In spatial-occupancy enumeration, a solid is decomposed into identical cells arranged in a fixed, regular spatial grid. These cells are often called voxels (volume elements). The maximum resolution of the solid is determined by the size of voxels. To construct an object, it is only required to decide whether a single cell at each position is present or absent.

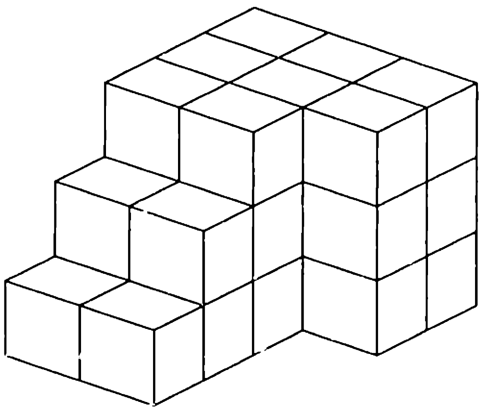


Figure 2.3 A model define by Spatial-Occupancy Enumeration.

2.2.4 Comparison of Representations

The properties of each representation method can be formally compared based on the following criteria:

Accuracy: spatial-occupancy enumeration method produces only approximated representations for most solid objects. In some applications, this is not a drawback, so long as an object is represented with an adequate resolution for its application. However, in order to obtain the higher resolution, the computation becomes costly because it requires a large amount of memory space. For applications in high-quality graphical display, because high resolution is required, CSG and B-Reps are often used.

Domain: The versatility of B-Reps depends on the complexity of faces and edges that are available, and the domain of CSG is limited by the available primitives. On the contrary, the spatial-partitioning method can represent any object with an approximation depending on the resolution.

Validity: In general, B-Reps are the most difficult representation to validate because of many vertex, edge, and face data structures, which might lead to intersection among faces or edges. In CSG, it is guaranteed to model a valid solid object, provided that the primitives are valid, while there is no checking needed for spatial-occupancy enumeration.

It is possible to transform the representation of a given object into another one. Figure 2.4 shows the possible transformations between CSG, B-Reps, and Spatial-Occupancy Enumeration based on the current algorithms that are available. The solid and dashed lines indicate the exact and approximated transformations, respectively.

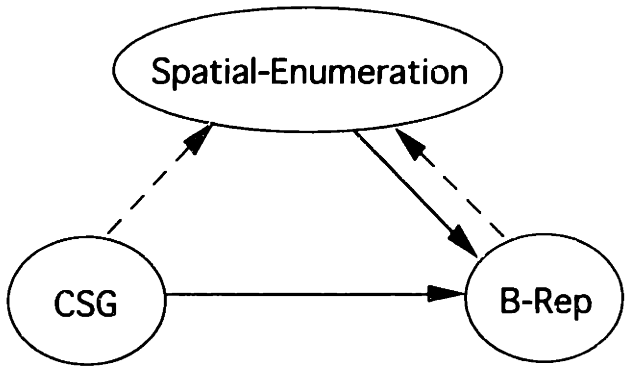


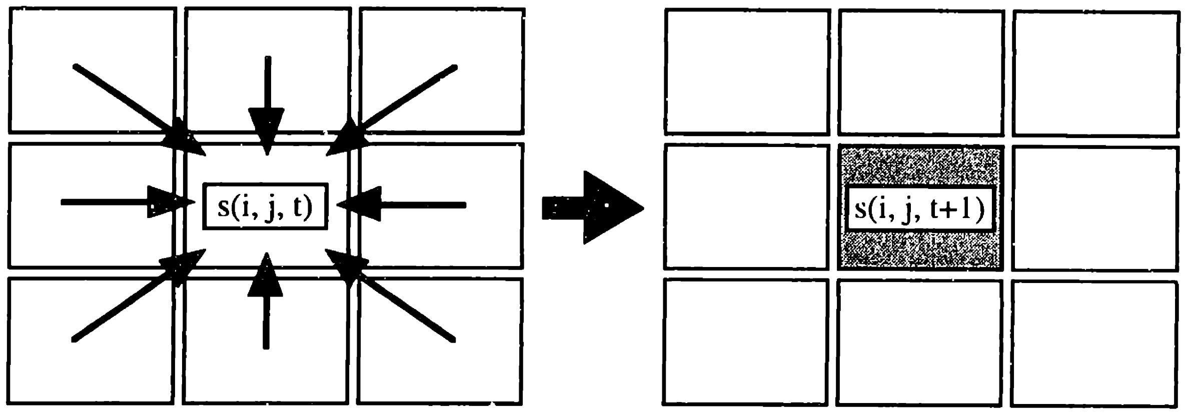
Figure 2.4 Possible transformation among geometric representations.

2.2 Geometric Description in This Investigation

In this investigation an assembly is represented by sets of cells each of which indicates a part in the assembly. Therefore, each part can be represented as the sum of a set of cells into which it can be decomposed. For this geometric representation of an assembly, all computations on these models are based on relational and logical operations, which means that the analysis algorithms are fast and can be decomposed in parallel processes. Based on this geometric description of a mechanical assembly, the physical reasoning and simulations for the purpose of generating assembly plans are performed using the cellular automata.

2.2.1 Cellular Automata

Cellular automata are arrays of elements whose states change in discrete time steps, which are called generations or iterations. A two-dimensional implementation, which is made up of an array of adjacent squares, is commonly used for image processing applications (Preston and Duff 1984). As is shown in figure 2.1, the state of a cell at location (i, j) at time t + 1, denoted as s(i, j, t+1), is a function of the state of the cell at time t, s(i, j, t) as well as the states of the neighboring cells at time t. States can assume any number of values, depending on the application under consideration. A set of local rules take the time t states as input (9 for the case shown in figure 2.1) and produce s(i, j, t+1) as output. Importantly, these local rules are *isotropic*, meaning that they are applied in the same way regardless of position in the array, and it is assumed that the operations that update the states of arrays occur simultaneously, *i.e.*, the action of all elements in the cellular array is *synchronous* (Preston and Duff, 1984, p. 12). It should be noted that the eight neighboring cells of the cell(i,j) also simultaneously update their states with respect to the states of their neighboring cells at time t including s(i,j,t).



State of a cell at time t
in the initial generation

State of a cell time t + 1
in the next generation

Figure 2.5 Cellular automata operation.

2.2.2 Cellular Automata for Physical Reasoning

The feasible motions of components in an assembly under force constraints are determined by iteratively applying cellular automata rules to the cellular arrays representing the assembly. These rules are described in detail in chapter 4. Because of the locality of the algorithm, the rules for physical reasoning do not have information regarding the global characteristics of a given assembly or a higher level representation of its parts. This implies that the information such as the number and shape of components in the assembly is unknown prior to the physical reasoning.

2.3 Assembly Planning and Sequence Generation

A simple two-dimensional example is presented to explain the concepts that are important to the approach described later. To simplify this brief introductory explanation, the use of formal notation to specify subassemblies and part motions has been kept to a minimum. When formal notation is used, however, it is consistent with that suggested by Homem de Mello and Sanderson (1991a).

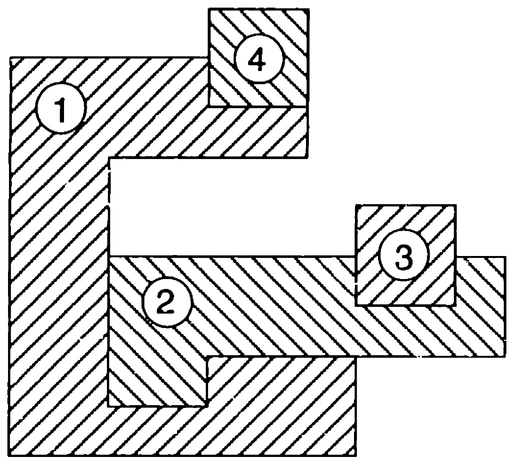


Figure 2.6 Example assembly.

Consider the assembly shown in figure 2.6. Assembly plans (again, made up of assembly orders and part motions) are generated by investigating *dis*assembly plans. Note that a disassembly plan, no matter how inefficient, will always proceed to single parts. Various assembly plans, on the other hand, can be "dead ends," meaning they can reach points from which the assembly process cannot correctly proceed. For almost all cases, the reverse of a valid disassembly plan will be a valid assembly plan. To generate disassembly plans, we assume that the assembly exists in an ideal tabletop environment. Two-dimensional parts move in the plane of the assembly and are stopped only if they collide with other parts (or fixtures) that are fixed. If they collide with other parts that are movable, they cause the movable parts to move with them. We impose an additional constraint that the parts only move in rectilinear directions, so a collision occurs when the edges of two parts, moving normal to one another, meet. Edges are assumed to be frictionless: motion is not transmitted through parallel sliding.

A simple procedure to generate disassembly plans might be as follows:

1. In turn, treat each part (in the fully assembled state) as a fixed or "base" part.

2. Try to remove each other part, in all possible orders. Note that removal of a part may move other parts, in effect creating a subassembly. Use this procedure recursively on subassemblies.

As an example, consider part 1 in figure 1 to be the base part. Part 2 could then be removed by moving it up until it collides with part 1, and then to the right until it leaves the vicinity of the assembly. Note that in its movements, part 2 has taken part 3 with it, as a two-part subassembly. Part 4 could then be removed by moving it up or to the right, and then part 3 could be removed from part 2 by moving it up. A disassembly plan (one of many possible) has thus been described by specifying the order of part removal and the motions required to move the parts.

All disassembly plans can be generated by combinatorially choosing each part as a possible base part and generating the related disassembly sequences. As there may be a very large number of plans, possibly with identical subplans, a compact efficient notation is required. We will use a set-based notation, formally presented by Homem de Mello and Sanderson (1991a). The fundamental general unit in this notation is the subassembly. A single part is the most basic subassembly and the entire assembly is the largest possible subassembly. A subassembly is denoted as a list of other subassemblies. The disassembly process outlined above would be represented as follows:

{ {1, 2, 3, 4} } :	The entire assembly is a single element list of one subassembly.
{ {1, 4}, {2, 3} } :	The subassembly comprised of parts 2 and 3 is removed, yielding two two-part subassemblies.
{ {1}, {4}, {2, 3} } :	Part 4 is separated from part 1
{ {1}, {4}, {2}, {3} } :	Part 3 is separated from part 2

Each step in this process yields a new *assembly state.*

Importantly, subassemblies are defined by contact between constituent subassemblies. As part 2 is removed with the two-step motion in the disassembly sequence described above, the subassembly {2, 3} is maintained because the movement of 2 does not break the contact between the parts 2 and 3. On the other hand, the subassembly {1, 4} is maintained so long as there is some contact between parts 1 and 4: part 4 need not be touching both faces of the corner cutout of part 1.[1]

[1]Note that here the "ideal tabletop" metaphor relaxes some constraints normally applied in assembly sequence planners (see Homem de Mello and Lee, 1991, chapters 6 - 15). It is typically required that when a contact is established between two subassemblies, it is the *final goal* contact state: the set of all physical contacts is established and this set will not be altered by subsequent assembly operations. This is not required in the explanatory example, nor in the implemented system. The additional planning capabilities this implies will be described in chapter 6.

Finally, in the above disassembly process, it is evident that the same assembly states can be reached in many different ways. Initially holding part 4 fixed and removing part 2 in the same manner described above will lead to the same state { { 1, 4}, {2, 3} }. Graph structures can be used to efficiently represent the possible transitions between assembly states. Figure 2 shows the *directed graph* of the assembly. The nodes of this graph are the possible partitions of the entire assembly into subassemblies, and the arcs represent feasible (dis)assembly tasks between the differently partitioned states. The *AND/OR graph* shown in figure 3, on the other hand, is a representation of the feasible subassemblies arranged in AND/OR trees that indicate all possible assembly sequences. AND's represent two subassemblies that can be joined to make up another subassembly, and OR's represent alternate pairs that may make up the same subassembly. Both graphs represent all possible assembly sequences, and there is a formal correspondence between the two representations (see Homem de Mello and Sanderson, 1991a, p. 141). Note that neither graph shows invalid assembly states. For example, { {1, 3}, {2, 4} } is not found.

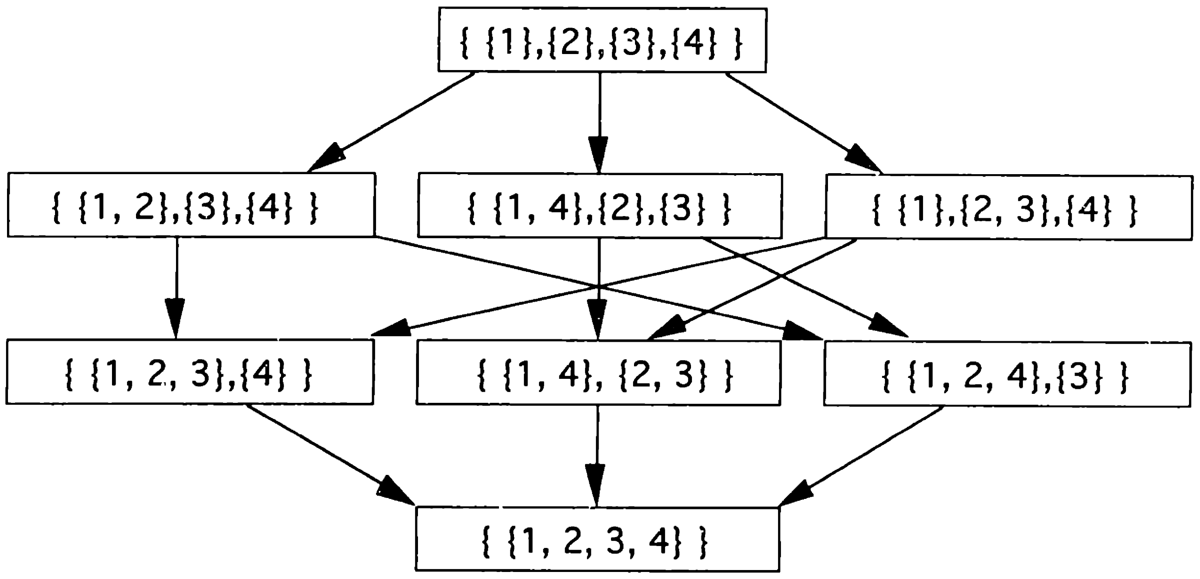


Figure 2.7 Directed graph of feasible assembly sequences for the example assembly shown in figure 1.

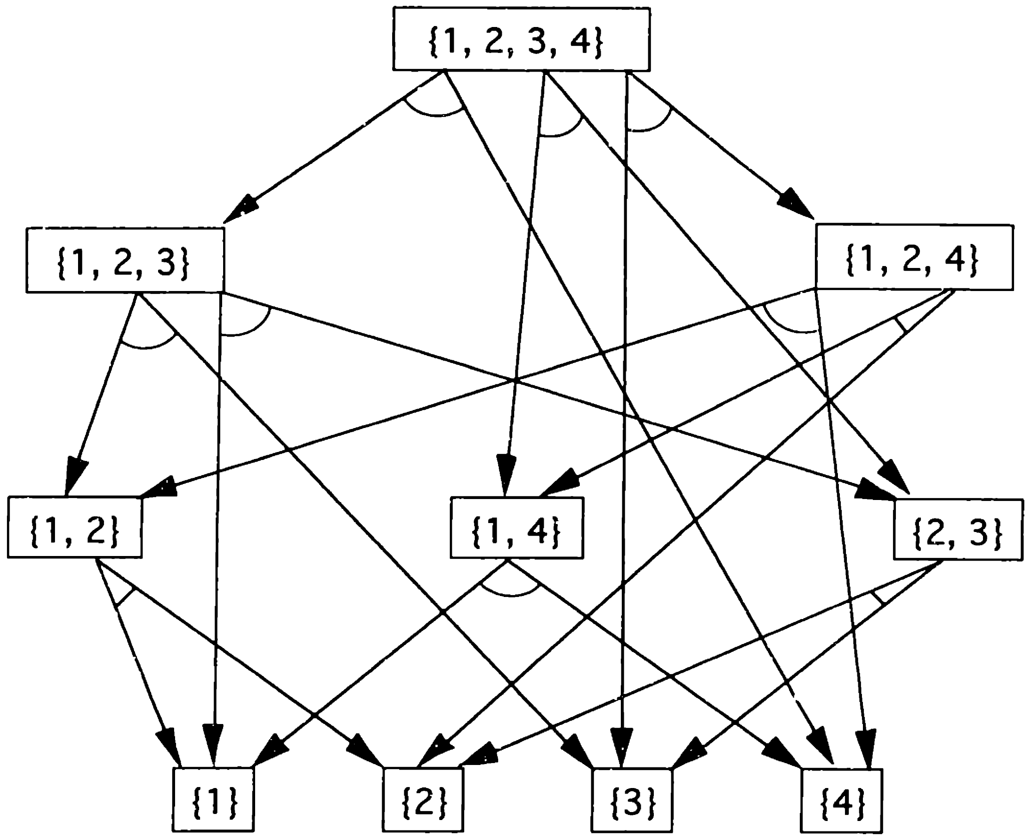


Figure 2.8 AND/OR Graph of feasible assembly sequences for the example assembly shown in figure 1.

2.4 Optimization Methods

With the same physical reasoning algorithm used for assembly sequence generation, good assembly sequences based on a predefined set of evaluation criteria are searched with optimization techniques. The criteria used in optimization are detailed in chapter 5.

2.4.1 Genetic Algorithms

The genetic algorithm (GA) is a global searching method based on the mechanics of natural selection and natural genetics (Holland 1975). Artificial organisms, each of which represents a solution point in the search space of an optimization problem, evolve over many generations in order to improve the overall quality of the population of artificial organisms. The genetic algorithm has demonstrated its robustness and versatility in number of analytical and empirical studies (Goldberg 1989).

There are a number of characteristics of the genetic algorithm which separate it from other conventional optimization methods. First of all, GAs search from a population of solution points, not from a single point. This gives GAs powerful parallelism, which is useful when multiple solutions are sought. Secondly, genetic algorithms search for the optimum by sampling, meaning a blind search. Therefore, even though there is no such information as gradient of a function available for an optimization technique to proceed for the next search point, GAs are able to perform the optimization of the function by concerning payoff. Thirdly, while other conventional techniques have deterministic transition rules, genetic algorithms utilize stochastic transition rules simulating natural systems. Lastly, genetic algorithms directly manipulate solution representations which are in form of strings at the low level to determine the similarities among high-performance solutions. This enables GAs to be robust even for problems with very complex functions.

In the genetic algorithm, a series of operations; selection, crossover, and mutation, is used to evolve the population of artificial organisms, which mimics the procedures of natural evolution. The general procedure of the genetic algorithm is shown in list 2.1. For a comprehensive introduction to the genetic algorithm, please refer to (Goldberg 1989).

List 2.1 Procedure of the genetic algorithm.

```
procedure GeneticAlgorithm
begin
    create initial population of chromosomes
    repeat
        begin
            evaluate the fitness of each chromosome
            select individuals with respect to their fitness values
            reproduce offsprings with selected parent chromosomes
            mutate the child chromosomes based on mutation probability
            replace parents with children
        end
    until maximum number of generation is reached or solution space converges
end
```

2.4.2 Branch And Bound

Branch and bound, which is also called "truncated enumeration method," is an optimization method used for integer programming problems in which there are finite number of feasible solutions which must be examined in order to determine the optimal solution (Christofides, Mingozzi et al. 1979). The branch and bound approach has proven to be the most successful in solving very special types of integer programming problems.

In a branch and bound algorithm, the total set of solutions under consideration is systematically subdivided into smaller and smaller sets instead of attempting to directly solve the given problem. These small sets or subproblems have the property that any optimal solution must be in at least one of the sets. This subdivision or partition of a given problem into subproblems is often illustrated by an 'enumeration tree' such as that of figure 2.9.

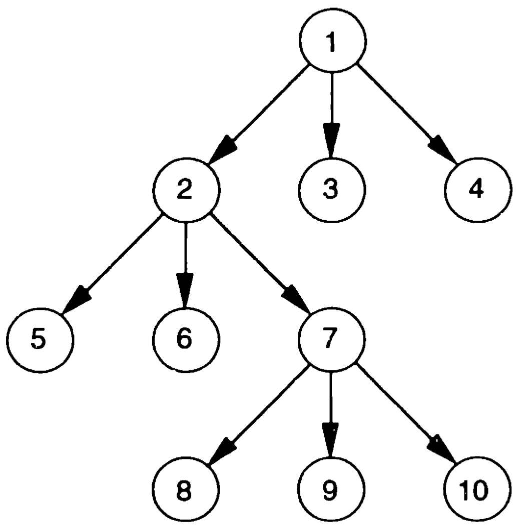


Figure 2.9 Enumeration tree of branch and bound method.

Each node of the enumeration tree shown in figure 2.9 corresponds to a subproblem. As the subdivision of nodes proceeds farther down the tree, the subproblems become smaller until it finally becomes possible to find an optimum or at least to determine whether or not it contains a potentially optimal solution. If it is determined that a node doesn't contain an optimal solution farther down the node, the node becomes pruned and excluded from the further search. Therefore, if an optimal solution is found in the early stage of a branch and bound search, the algorithm can effectively discard many subproblems without exploring further.

3 Previous Work in Assembly Sequence Generation

3.1 Overview

Generation of assembly sequences has been the central topic of much research and a number of methods have been developed to date. Some of these methods generate subsets of all feasible assembly sequences, while others generate all feasible sequences. Some methods require the user interactions, which result in partially automated systems, while others are fully automated.

To automate the process of assembly plan generation, it is necessary to automate the process of geometric reasoning and efficiently generate a representation of the possible assembly sequences. Various methodologies exist for representing mechanical assemblies. The difficulty of geometric reasoning when using a realistic representation of geometry, such as a boundary representation CAD model, has been a focal point of much of the research systems previously developed. This chapter divides the currently active approaches into two groups based on the degree of automation, and briefly overviews the emphasis of each approach.

3.2 Computer-Based Interactive Assembly Planning

Interactive assembly planning is concerned with formulating a necessary and sufficient set of questions to be answered by the user or designer. These question-and-answer operations are then used to determine or complete the precedence relationships among parts in a product. In this approach, It is crucial to minimize the number of questions required for the generation of assembly plans in order to enhance the efficiency of the system.

3.2.1 Bourjault and Henrioud

Bourjault in 1984 proposed the first systematic method for determining all feasible assembly plans available for a given assembly using precedence constraints or relations among the various assembly connections (Bourjault 1984). Bourjault named these relations and the graphical representation of them "liaison" and "liaison diagram," respectively. An initial computer-based implementation of Automatic Generation of Assembly Sequences (SAGA) was developed, which focused on the liaisons between parts and represented an assembly sequence as a series of liaisons (Bourjault 1987). In their system, the user must supply answers to a series of questions asked by the system, which are associated with the precedence constraints between liaisons.

Bourjault's group in (Henrioud and Bourjault 1991) presented a new approach which was not only based on the liaison graph, but focused on the parts by introducing additional constraints that express some reasonable strategies for the assembly process and are deduced from the product structure. They also presented a computer-based implementation of this method, called LEGA. The author mentions that the current version of LEGA is still prohibitive for products having more than ten components because it requires much time from the user and produces too many valid assembly trees. They are also investigating a partially automated evaluation of assembling operations or sequences so as to rank them. In the paper, they classified and formalized the assembly constraints into two categories; operative constraints and strategic constraints.

The operative constraints, such as geometric stability, and material constraints, are mainly used to check the feasibility of assembly operations. When the system cannot determine feasibility, it asks the user questions. The strategic constraints, which are introduced in the new approach so as to allow the determination of a reasonable number of assembly trees, include imposed subassemblies, groups of components, and linear assembly trees. These strategic constraints must be introduced before the process of assembly tree determination begins and, thereby, the choice and definition of assembly constraints are left to the user allowing subjectivity such as the user's expertise to enter into the systematic process.

3.2.2 De Fazio and Whitney

(De Fazio and Whitney 1987) presented a modification of Bourjault's method which simplifies the form and reduces the number of questions that are needed to generate all valid assembly sequences. This modified method requires $2l$ questions that are answered in a precedence-logical form, while Bourjault's methods requires $2l^2$ questions for an assembly with l parts. This simplification allows the practical extension of the techniques to assemblies with much higher parts count. The questions which evoke the set of precedence relations are of a form familiar to persons concerned with assembly.

In addition to the assembly sequence generator based on the liaison method, a computer-interactive editing system was implemented in order to reduce the sequence count to a small desirable sets ((De Fazio and Whitney 1989), (De Fazio, Abell et al. 1990), and (Baldwin, Abell et al. 1991)). The editing of assembly sequences generated by their system consists of a number of operation stages. During these stages, the user edits the sequences based on various criteria related to the production environment of a product. The system aids the user in judging the value of feasible assembly sequences for the product with on-line visual aids during the evaluation.

3.3 Computer-Automated Assembly Planning

As powerful computer-based geometric modeling and reasoning tools are developed, the automatic determination of geometric interference or assembly path can be utilized as a means of identifying precedence relationships in assembly.

3.3.1 Homen de Mello and Sanderson

Homem de Mello and Sanderson in (Homem de Mello and Sanderson 1990) proposed a compact representation of assembly plans of a product using as AND/OR graph. An AND/OR graph is used to represent the set of all assembly sequences. The nodes in the graph represent the stable subassemblies of a given product or assembly. Each enumerated arc corresponds to the geometrically and mechanically feasible assembly tasks. Figure 3.2 shows the AND/OR graph of a simple product presented by Homem de Mello and Sanderson in figure 3.1.

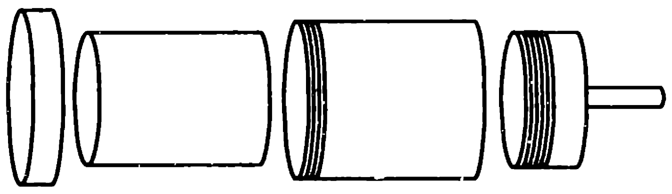


Figure 3.1 A four-part simple product from (Homem de Mello and Sanderson 1990).

In an AND/OR graph, the state of an assembly process is defined by the configuration of subassemblies, which can be described by the fixed relative position between pairs of parts. Then, the assembly plan can be seen as a sequence of these states. Using the AND/OR graph, the problem of finding how to assembly a given product can be converted to an equivalent problem of finding how the product can be disassembled. Then, this resulting backward approach to find the assembly plans of a product may be viewed as a system in which the problem of disassembling an assembly is decomposed into distinct subproblems. Then, each of these subproblems is to disassemble one subassembly. All feasible assembly or disassembly sequences can be then found, thus, by decomposing subsets of the given assembly until single part subassemblies are reached.

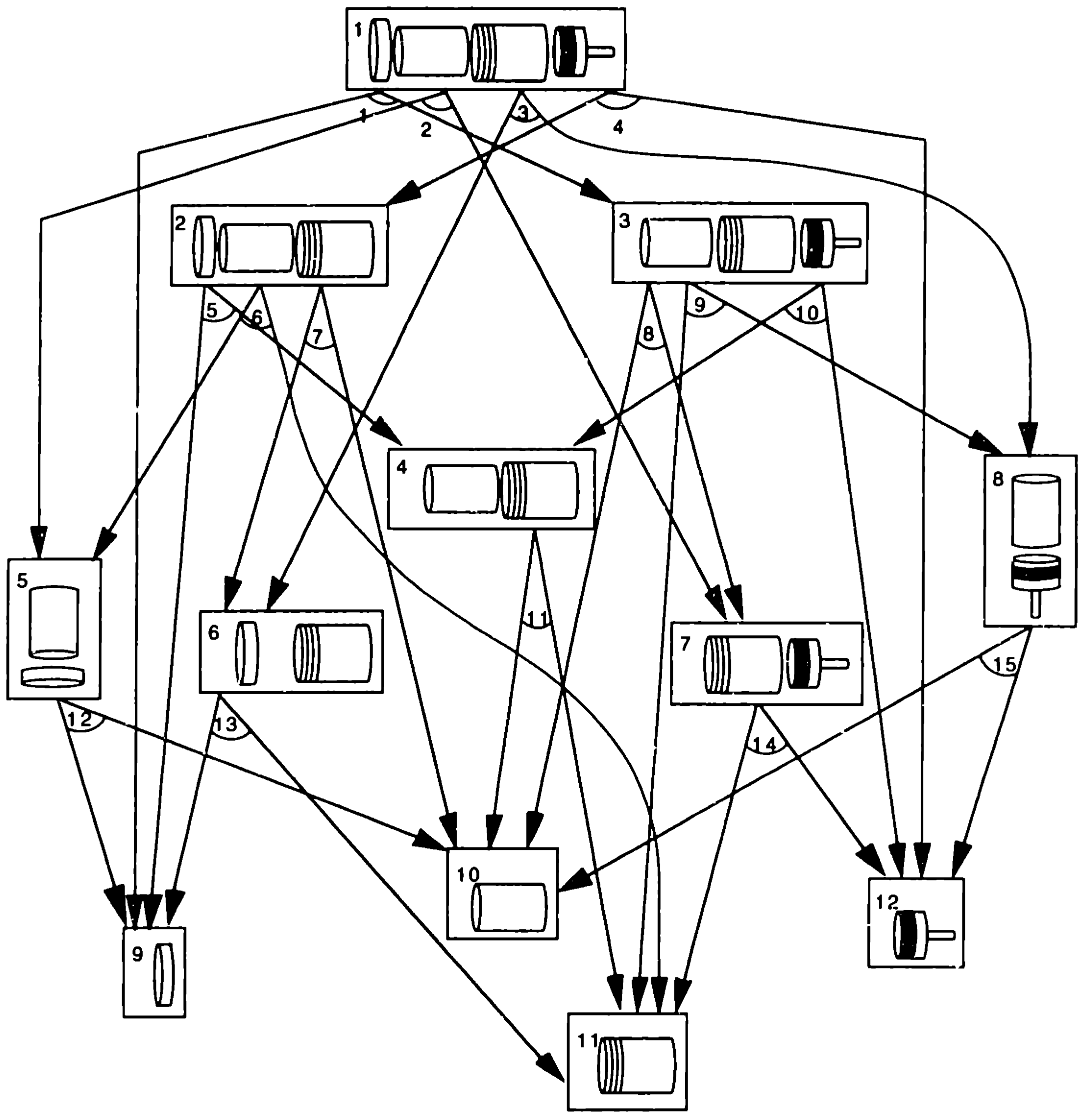


Figure 3.2 AND/OR Graph of the simple product in figure 3.1.
(adapted from (Homem de Mello and Sanderson 1990))

It was also mentioned that the AND/OR graph can be used for traversing the space of all candidate solutions or feasible assembly sequences for a problem of selecting the best assembly plan.

In (Homem de Mello and Sanderson 1990) and (Homem de Mello and Sanderson 1991), a correct and complete algorithm for the generation of all feasible sequences of a given product was presented. The precedence relations among parts in the assembly are found from allowed or precluded part-partitions. Cut-sets of the assembly's graph of connections are evaluated for possibility of decomposition without interference and the assemblability is tested as disassemblability.

3.3.2 Wilson

Wilson and Rit in (Wilson and Rit 1991) describes a method to efficiently build an AND/OR graph for the linear assembly plans for a product and presented the assembly planner GRASP, which generates assembly sequences strictly from the geometry of a product with no human input. They proposed a three level approach to reduce the number of geometric checks; (i) inheriting assemblability from a parent assembly, (ii) considering obstructing components, and (iii) clustering obstacles by assembly path. Two steps are taken to check the movability of a part. First, the local translational freedom of the part is checked by analyzing the part's contact. Then, the global validity of the disassembling path is ensured by sweeping the part along the chosen directions of the part.

(Wilson 1990) presents an extended algorithm that solves the physical partioning problem to facilitate generating non-linear assembly sequences, which produce parallel subassemblies during assembly planning.

Recently, Wilson integrated the method described above with a user interaction method such as De Fazio and Whitney's. This dual approach to the problem of generating assembly plans is described in (Wilson 1993). In this approach, most assembly operations are checked from the CAD models of the assembly's parts using the previous method and questions associated with the geometric feasibility of some assembly operations are asked of the user.

3.3.3 Wolter

Wolter presented a two step process for generating multiple assembly sequences (Wolter 1989). He used multiple assembly axis trajectories for mating part pairs to generate the precedence constraints.

This trajectory technique for generating assembly plans was then extended further such that non-monotone and non-linear assembly plans can be generated (Tsao and Wolter 1993). In non-monotone plans, a part can be moved to an intermediate position which is a non-goal position instead of moving directly from its initial position to the goal position. Non-linear or parallel plans, as described in chapter 5, are able to deal with moving a group of parts.

In (Chakrabarty and Wolter 1994), a fundamentally new approach for generating assembly plans is proposed, in which an assembly is viewed as a hierarchy of standard structures. An assembly plan is generated by merging the plans in a recursive manner.

3.3.4 Hoffman

Miller and Hoffman proposed automatic planning which takes into account fasteners that are used to connect parts or subassemblies (Miller and Hoffman 1989). A valid disassembly sequence is determined from two types of constraints imposed on a product: geometric descriptions of objects and labels for the different fastener types. The model in their system is defined in terms of Constructive Solid Geometry (CSG) primitives and nuts, bolts, and screw fastener primitives.

In (Hoffman 1990), Hoffman presented a different approach for assembly sequence generation of a product using the boundary representation (B-Rep) format with bicubic surfaces. This treats components as part of a real workcell environment by allocating sections of the work table to subassemblies and disassembled parts.

3.3.5 S. Lee

Lee presented a method for the automatic determination of an assembly partial order from a CAD database through the recursive extraction of preferred subassemblies from a liaison graph representation of an assembly ((Lee 1994), (Lee 1991), and (Lee and Shin 1990)). A set of tentative subassemblies is selected by decomposing a liaison graph into a set of subgraphs based on feasibility and aspects of difficulty of disassembly such as directionality, stability, and manipulability. To reduce the possible number of liaison graph decomposition, nodes that are mutually inseparable in the current state of disassembly are merged into a super node, forming an abstract liaison graph. Each of the tentative subassemblies are evaluated based on the subassembly selection indices, and the preferred subassemblies are extracted along with the verification of their disassemblability. Lee defined this particular method for achieving an assembly plan based on the recursive identification and selection of desirable direct subassemblies as "Backward Assembly Planning."

Lee also proposed an integrated system for assembly planning and redesign, based on "Design For Assembly (DFA)" in (Lee, Kim et al. 1993). The main objectives of the system are to generate a cost effective or preferred assembly sequence of a given product based on number of design criteria including DFA, and to generate a redesign strategy based on the DFA analysis result. When a redesign strategy is generated, a user would make appropriate changes in the CAD database based on the strategy. Then, the procedure described above can be repeated for the improved design.

3.4 Summary

Based on the solution strategies for assembly sequence generation, methods can be divided into two groups. The first approach utilizes user interaction, which extracts precedence relations between parts of the final assembly by question-and-answer sessions which deal with the feasibility of some assembly operations. The other uses the geometric and topological information of a given assembly from a CAD database, and automatically generates the precedence relations between parts by a series of computer-automated physical reasonings. These precedence relations are then used to generate the feasible assembly sequences.

4 Assembly Simulation With Cellular Automata

4.1 Overview

This chapter describes the basis for a computer-based implementation of the cellular representation of assembly and algorithms for simulating movements of parts so as to perform physical reasonings. A procedure to extract a part from an assembly is then explained. Lastly, algorithms to determine all feasible assembly sequences by generating an AND/OR graph for a given assembly is described with an example, and the performance of the algorithms are discussed.

4.2 Simulation of Rigid Part Interactions

4.2.1 Cell State

A two-dimensional rectilinear array is used to represent an arrangement of rigid planar mechanical parts. This could, for example, be a depiction of a cross-section of a cylindrical assembly. Individual cells assume a state vector, which is shown in figure 4.1, instead of a single state quantity. This is referred as the cell's *frame*. The five fields of this frame and their uses are as follows:

Material : A single label which records the solid/void distinction.

Edge : A vector of four labels for each edge of the cell. These will be called top, right, bottom, and left. Each label can assume a value of "free" or "glued," which describes the state of the edge. Free edges are a frictionless sliding interface and glued edges occur between two cells of the same part.

Part_Label : A label which can take on integer values 1, 2, 3, ... As will be seen, different parts can be identified in the cellular array. When they are, the cells of that part are given the same Part_Label number.

Movement_Defined : Parts will be moved in unit steps in one of the rectilinear directions. Each cell will be assigned a single movement direction before each unit step move. This field can take on a binary true/false value, depending on whether or not an allowable movement direction has been assigned to the cell.

Move_Direction : A vector of four labels that together indicate the allowable movement direction. If Movement_Defined is true, five values are possible:

1 0 0 0 :	Upward motion possible
0 1 0 0 :	Rightward motion possible
0 0 1 0 :	Downward motion possible
0 0 0 1 :	Leftward motion possible
0 0 0 0 :	Fixed (no motion possible)
1 1 1 1 :	Initialization value

The terminology of the investigation will be to simply use a frame field name to refer to the frame field value of a particular cell. For example material(i, j, t) indicates if cell(i, j) contains material or void at time t. Right_edge(i, j, t) indicates if the right edge of cell(i, j) is free or glued at time t. A Cartesian coordinate system is also used such that the i index increases in value from left to right and the j index increases in value from top to bottom (*e.g.* s(i, j-1, t) is the state of the cell *above* the cell under consideration).

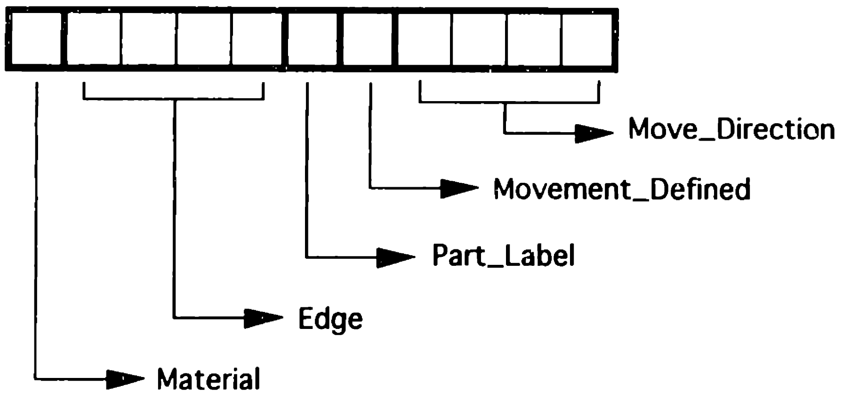


Figure 4.1 State vector of a cell.

4.2.2 Part Membership Rules

Given an assignment of material and edge types to the cellular array, a first set of rules determines a higher level representation of parts from the lower level representation of cells. In other words, the membership of cells to parts is revealed. Underlying this set of rules is the obvious fact that two adjacent cells, both filled with material, with a glued edge between them, will both be in the same part. There are four rules, one for each adjacency direction.

List 4.1 Rules for membership.

```
if
        material( i, j, t) = solid and
        part_label( i, j, t) = unlabeled and
        top_edge( i, j, t) = glued and
        part_label( i, j-1, t ) != unlabeled
then
        part_label( i, j, t+1) ← part_label( i, j-1, t)

if
        material( i, j, t) = solid and
        part_label( i, j, t) = unlabeled and
        bottom_edge( i, j, t) = glued and
        part_label( i, j+1, t ) != unlabeled
then
        part_label( i, j, t+1) ← part_label( i, j+1, t)

if
        material( i, j, t) = solid and
        part_label( i, j, t) = unlabeled and
        left_edge( i, j, t) = glued and
        part_label( i-1, j, t ) != unlabeled
then
        part_label( i, j, t+1) ← part_label( i-1, j, t)

if
        material( i, j, t) = solid and
        part_label( i, j, t) = unlabeled and
        right_edge( i, j, t) = glued and
        part_label( i+1, j, t ) != unlabeled
then
        part_label( i, j, t+1) ← part_label( i+1, j, t)
```

The algorithm to label all cells is shown in List 4.2. After executing the algorithm, each cell has a label to identify parts.

List 4.2 Algorithm to determine membership.

```
procedure DetermineMembership( domain )
begin
    while ( always )
    begin
        search for a solid and unlabeled cell
        if no cell was found
            then return
        else put a new part_label on the cell found
            do
                apply the rules for membership to domain
            until no rules are fired
    end
end
```

This initial set of rules does not move the parts; it only determines the physical meaning of the array assignment. With regard to this physical meaning, it is also important to note that physically incorrect, or at least physically inappropriate, situations can arise. Freely moving parts, for example can be enclosed in internal voids of other parts, and other special cases such as cracks, are possible.

4.2.3 Part Motion Rules.

In the cellular automata representation, part motion is achieved by shifting the material of parts into neighboring cells. Parts, and their constituent cells, can move into neighboring void cells, or into cells occupied by other parts if these other parts can in turn move out of the way. The fifth element of the state vector indicates the impending allowable movement direction of a cell. This can assume five values: up, down, left, right, and fixed (*i.e.* no movement is allowed). These values are assigned to elements with a second set of rules that uses commonsense physical reasoning to propagate allowable movement values within the cells of individual parts and between the cells of different parts across edges. An initial movement value, representing a robot grasping and moving a part for instance, is typically assigned to a single cell, and the effect of this assignment is propagated over the entire cellular array.

There are two kinds of rules to achieve part motion: possible motion propagation rules, and cell movement rules. The possible motion propagation rules determine the possible motion of all cells from given initial movement. The cell movement rules shift cells along the direction of their possible motion by one cell unit.

Possible Motion Propagation Rules

The possible motion propagation rules are shown in List 4.3, and are divided into three sets, (a) propagation within a component; (b) blocking motion; and (c) propagation beyond part boundary. If no rules from these sets are applicable, the properties of the cell are maintained for the next time step.

List 4.3 Possible movement propagation rules.

(a) Propagation within component

```
if
        material( i, j, t) = solid and
        top_edge( i, j, t) = glued and
        movement_defined( i, j-1, t) = true
then
        movement_defined( i, j, t+1 ) ← true
        move_direction( i, j, t+1) ← move_direction( i, j, t) ^
                                     move_direction( i, j-1, t)

if
        material( i, j, t) = solid and
        right_edge( i, j, t) = glued and
        movement_defined( i+1, j, t) = true
then
        movement_defined( i, j, t+1 ) ← true
        move_direction( i, j, t+1) ← move_direction( i, j, t) ^
                                     move_direction( i+1, j, t)

if
        material( i, j, t) = solid and
        bottom_edge( i, j, t) = glued and
        movement_defined( i, j+1, t) = true
then
        movement_defined( i, j, t+1 ) ← true
        move_direction( i, j, t+1) ← move_direction( i, j, t) ^
                                     move_direction( i, j+1, t)

if
        material( i, j, t) = solid and
        left_edge( i, j, t) = glued and
        movement_defined( i-1, j, t) = true
then
        movement_defined( i, j, t+1 ) ← true
        move_direction( i, j, t+1) ← move_direction( i, j, t) ^
                                     move_dir( i-1, j, t)
```

(b) Blocking motion.

```
if
```

```
        material( i, j, t) = solid and
        top_edge( i, j, t ) = free and
        move_direction( i, j, t) = up and
        ( move_direction( i, j-1, t) = fixed or
        move_direction( i, j-1, t ) = left or
        move_direction( i, j-1, t ) = down or
        move_direction( i, j-1,t) = right )
then
        move_direction( i, j, t+1 ) ← fixed

if
        material( i, j, t) = solid and
        right_edge( i, j, t ) = free and
        move_direction( i, j, t) = right and
        ( move_direction( i+1, j, t) = fixed or
        move_direction( i+1, j, t ) = left or
        move_direction( i+1, j, t ) = down or
        move_direction( i+1, j, t) = up )
then
        move_direction( i, j, t+1 ) ← fixed

if
        material( i, j, t) = solid and
        bottom_edge( i, j, t ) = free and
        move_direction( i, j, t) = down and
        ( move_direction( i, j+1, t) = fixed or
        move_direction( i, j+1, t ) = left or
        move_direction( i, j+1, t ) = up or
        move_direction( i, j+1, t) = right )
then
        move_direction( i, j, t+1 ) ← fixed

if
        material( i, j, t) = solid and
        left_edge( i, j, t ) = free and
        move_direction( i, j, t) = left and
        ( move_direction( i-1, j, t) = fixed or
        move_direction( i-1, j, t ) = up or
        move_direction( i-1, j, t ) = down or
        move_direction( i, j-1,t) = right )
then
        move_direction( i, j, t+1 ) ← fixed
```

(c) Propagation beyond part boundary.

```
if
        material( i, j, t) = solid and
        top_edge( i, j, t) = free and
        movement_defined( i, j, t) = false and
        move_direction( i, j-1, t) = down
then
        movement_defined( i, j, t+1) ← true
        move_direction( i, j, t+1) ← down
```

```
if
        material( i, j, t) = solid and
        right_edge( i, j, t) = free and
        movement_defined( i, j, t) = false and
        move_direction( i+1, j, t) = left
then
        movement_defined( i, j, t+1) ← true
        move_direction( i, j, t+1 ) ← left

if
        material( i, j, t) = solid and
        bottom_edge( i, j, t) = free and
        movement_defined( i, j, t) = false and
        move_direction( i, j+1, t) = up
then
        movement_defined( i, j, t+1) ← true
        move_direction( i, j, t+1 ) ← up

if
        material( i, j, t) = solid and
        left_edge( i, j, t) = free and
        movement_defined( i, j, t) = false and
        move_direction( i-1, j, t) = right
then
        movement_defined( i, j, t+1) ← true
        move_direction( i, j, t+1 ) ← right
```

The rules of list 4.3(a) are the movement propagation rules for cells that are adjacent within a component. For a component to move, all cells of the component must shift in the same direction. This set of rules ensures this by checking if adjacent glued cells can have the same impending movement direction. If they cannot, the cell under consideration is considered fixed: this is achieved by the logical AND operation of the Move_Direction fields of the two cells. There are four rules, one for each adjacency direction. Figures 4.2a and 4.2b show how a movement direction and a fixed designation will propagate across a glued edge.

The set of rules shown in list 4.3(b) identify if a cell from an adjacent component will cause the cell under consideration to be fixed. As is shown in figures 4.2c and 4.2d, this can occur if the adjacent cell (*i.e.* across a free boundary) is fixed or has a different impending movement direction. Again, there are four rules, one for each adjacency direction.

The final group of motion propagation rules, shown in list 4.3(c), handles the cases in which impending motion directions can propagate across a free boundary. This occurs when an impending movement direction for the cell under consideration has not yet been defined. An example of the use of the first rule is shown in figure 4.2e. Again, there are four rules, one for each adjacency direction.

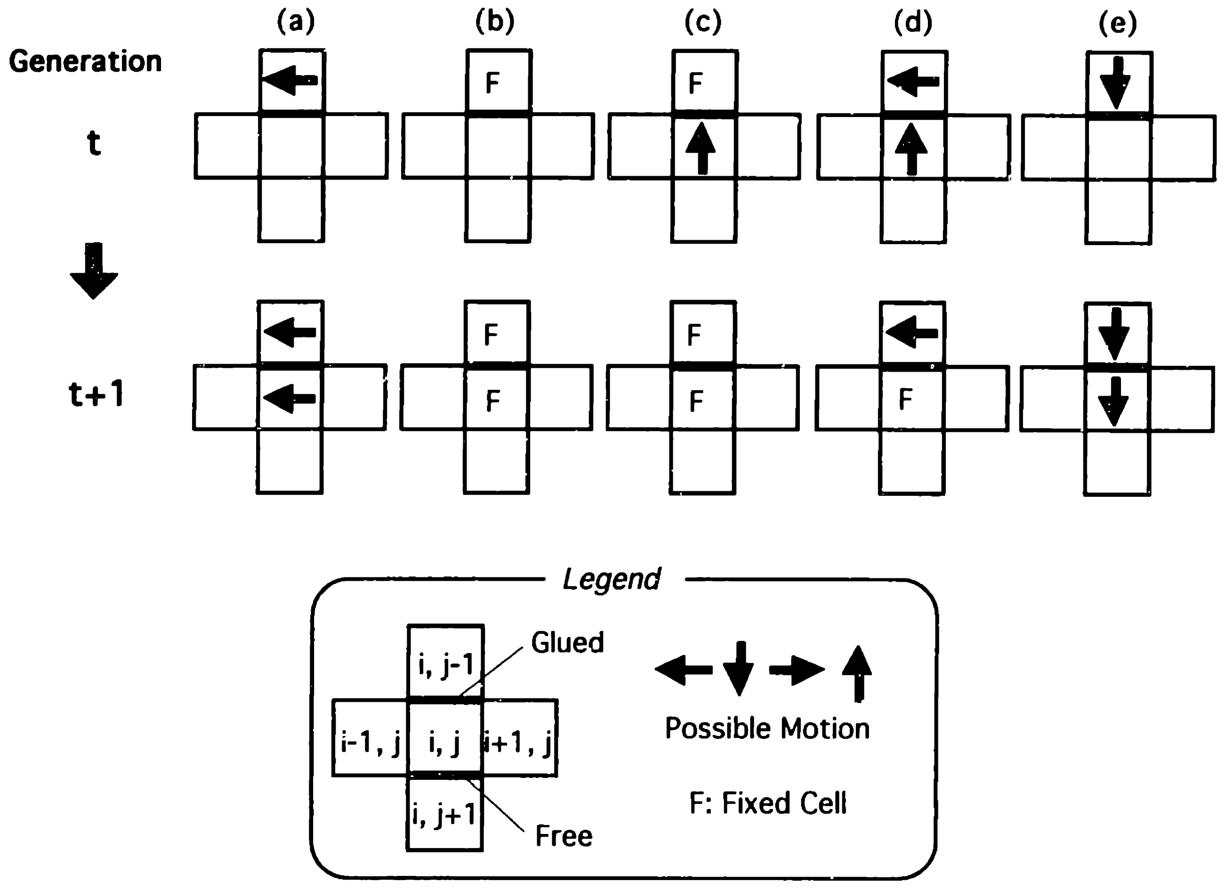


Figure 4.2 Application of possible motion propagation rules.

Cell Movement Rules

Once the possible motion propagation rules have assigned impending movement directions to all cells of the array, the components can be moved by shifting cells values in the array. This is done with the cell movement rules that are shown in list 4.4.

As is shown in figure 4.3a, it is possible that two cells adjacent to the cell under consideration would collide if they were moved according to their assigned allowable move direction. This condition is recognized by the single rule of list 4a.

Complementing the collision case, the rules of lists 4.4(b) and 4.4(c) address the situations in which material respectively moves in or moves out of the cell under consideration. In the move-in rules, the antecedents ensure that a collision will not occur and determine which adjacent cell will be shifted into the center cell. Note that, as is shown in figures 4.3b and 4.3c, a move-in is possible if the center cell is void of if the material of the center cell will be moved out of the way. The consequent clause "all-property" refers to the entire vector of a cell, and the assignment shown in the consequent indicates how the cells are shifted.

In the move-out rules, the antecedents determine if the c^ll under consideration will become void after the assigned impending motion is carried out. An example is shown in figure 4.3d.

List 4.4 Cell movement rules.

(a) Collision check rules.

```
if
        ( move_direction( i, j-1, t ) = down and
        move_direction( i, j+1, t ) = up ) or
        ( move_direction( i-1, j, t ) = right and
        move_direction( i+1, j, t ) = left ) or
        ( move_direction( i, j-1, t ) = down and
        move_direction( i+1, j, t ) = left ) or
        ( move_direction( i+1, j, t ) = left and
        move_direction( i, j+1, t ) = up ) or
        ( move_direction( i, j+1, t ) = up and
        move_direction( i-1, j, t ) = right ) or
        ( move_direction( i-1, j, t ) = right and
        move_direction( i, j-1, t ) = down )
then
        collision
```

(b) Move-in rules

```
if
        ( material( i, j, t) = void or
        move_direction( i, j, t ) = down ) and
        move_direction( i, j-1, t) = down
then
        all_property( i, j, t+1 ) ← all_property( i, j-1, t)

if
        ( material( i, j, t) = void or
        move_direction( i, j, t ) = left ) and
        move_direction( i+1, j, t) = left
then
        all_property( i, j, t+1 ) ← all_property( i+1, j, t)

if
        ( material( i, j, t) = void or
        move_direction( i, j, t) = up ) and
        move_direction( i, j+1, t) = up
then
        all_property( i, j, t+1 ) ← all_property( i, j+1, t)

if
        ( material( i, j, t) = void or
        move_direction( i, j, t ) = right ) and
        move_direction( i-1, j, t) = right
then
        all_property( i, j, t+1 ) ← all_property( i-1, j, t)
```

(c) Move-out rules

```
if
        move_direction( i, j, t ) = down and
        move_direction( i, j-1, t ) != down and
        move_direction( i+1, j, t) != left and
        move_direction( i-1, j, t) != right
then
        material( i, j, t ) = void

if
        move_direction( i, j, t ) = right and
        move_direction( i, j-1, t ) != down and
        move_direction( i-1, j, t) != right and
        move_direction( i, j+1, t) != up
then
        material( i, j, t ) = void

if
        move_direction( i, j, t ) = up and
        move_direction( i, j+1, t ) != up and
        move_direction( i+1, j, t) != left and
        move_direction( i-1, j, t) != right
then
        material( i, j, t ) = void

if
        move_direction( i, j, t ) = left and
        move_direction( i, j-1, t ) != down and
        move_direction( i, j+1, t) != up and
        move_direction( i+1, j, t) != left
then
        material( i, j, t ) = void
```

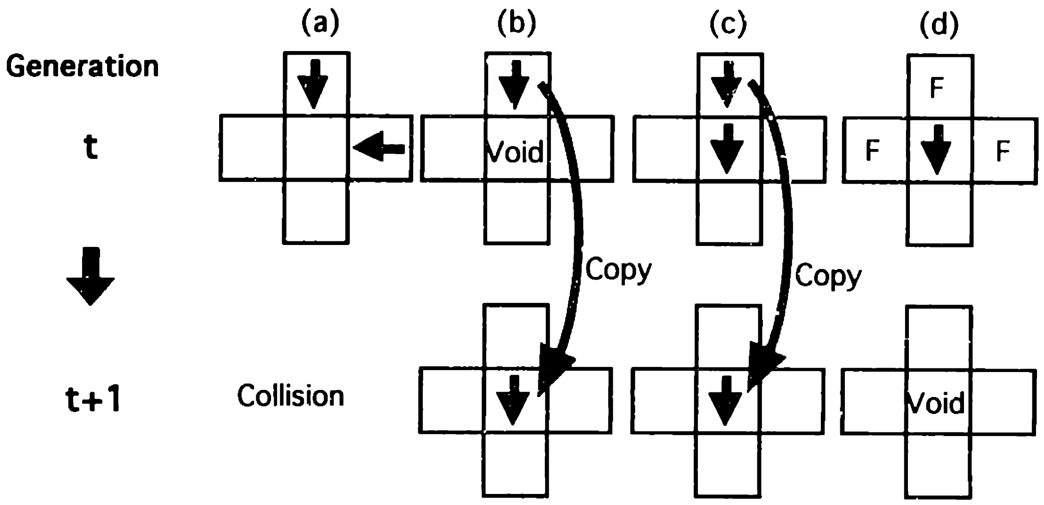


Figure 4.3 Application of possible cell movement rules.

4.2.4 Part Motion Procedure.

The basic part motion procedure moves one part (recall that other parts may move with it) in one direction until further motion is not possible, or the part leaves the assembly, which means that the part moves out of the cellular domain. This is achieved by applying the rules of lists 4.3 and 4.4 with the algorithm shown in list 4.5.

List 4.5. Algorithm to move parts.

```
procedure MovePartUntilStop
begin
    repeat
        repeat
            apply Possible Motion Propagation Rules
        until converged
        if there is no possible motion in the domain
            then return STOPPED
        apply Cell Movement Rules
        if there is collision then return ERROR
    until forever
end
```

Figure 4.4 illustrates how the algorithm *MovePartUntilStop* works. In Figure 4.4a, an upward motion at cell (2, 2) and a fixed condition at cell (2, 4) are given as an initial condition. Figures 4.4b, 4.4c, and 4.4d show how the possible motion propagation rules are applied once, twice, and three times, respectively. In Figure 4.4d, the domain is converged and will not be changed if the possible motion propagation rules are applied to it. After convergence, if there are no cells with possible motions in the domain, the program stops normally. In this case, there are some cells with possible motion, which causes the cell movement rules to be applied to the domain. Figure 4.4e shows the result is that the part moves one unit cell upward.

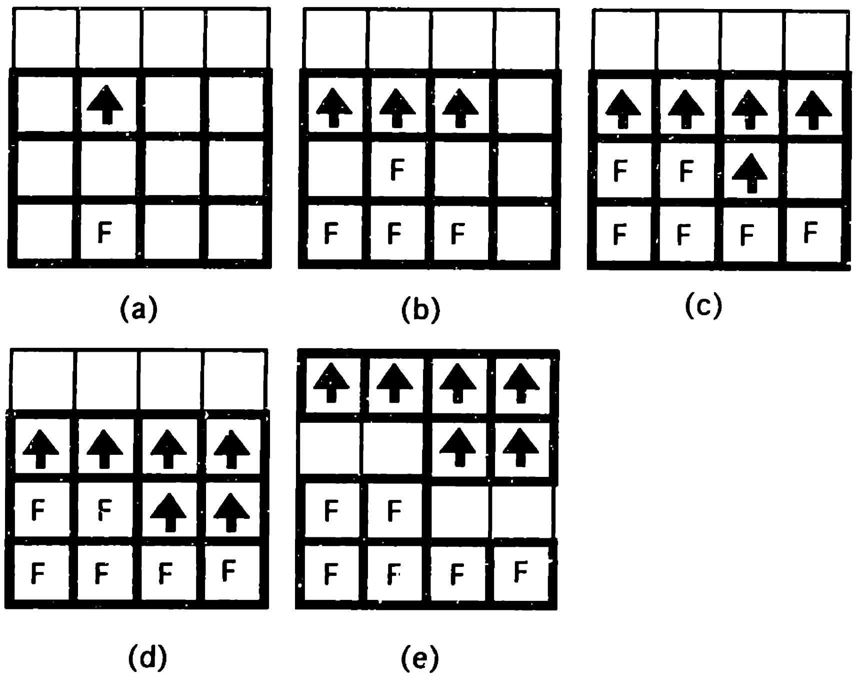


Figure 4.4 Example of allowable motion propagation and cell movement rules.

4.2.5 Part Extraction Procedure.

This fundamental part motion procedure becomes the basis for a part (subassembly) extraction procedure that is used in the assembly sequence generation. We will refer to this procedure as *ExtractSubassembly.* Figure 4.5 shows how a single part is extracted from an assembly. In Figure 4.5a, a given assembly to be disassembled is shown. First, as shown in figure 4.5b, the different parts are revealed using the part membership rules of list 4.1. Second, one of the cells in part A is fixed and an initial movement value downward is assigned to one of the cells in part B, as initial conditions, as shown in figure 4.5c. Third (figure 4.5d), the procedure MovePartUntilStop is applied to the domain. Part

B is moved until it collides with Part A. Part B could not be extracted by single motion. Therefore, another movement value, rightward, is assigned to the cell of Part B. Finally, Part B is extracted from the assembly as shown in figure 4.5f.

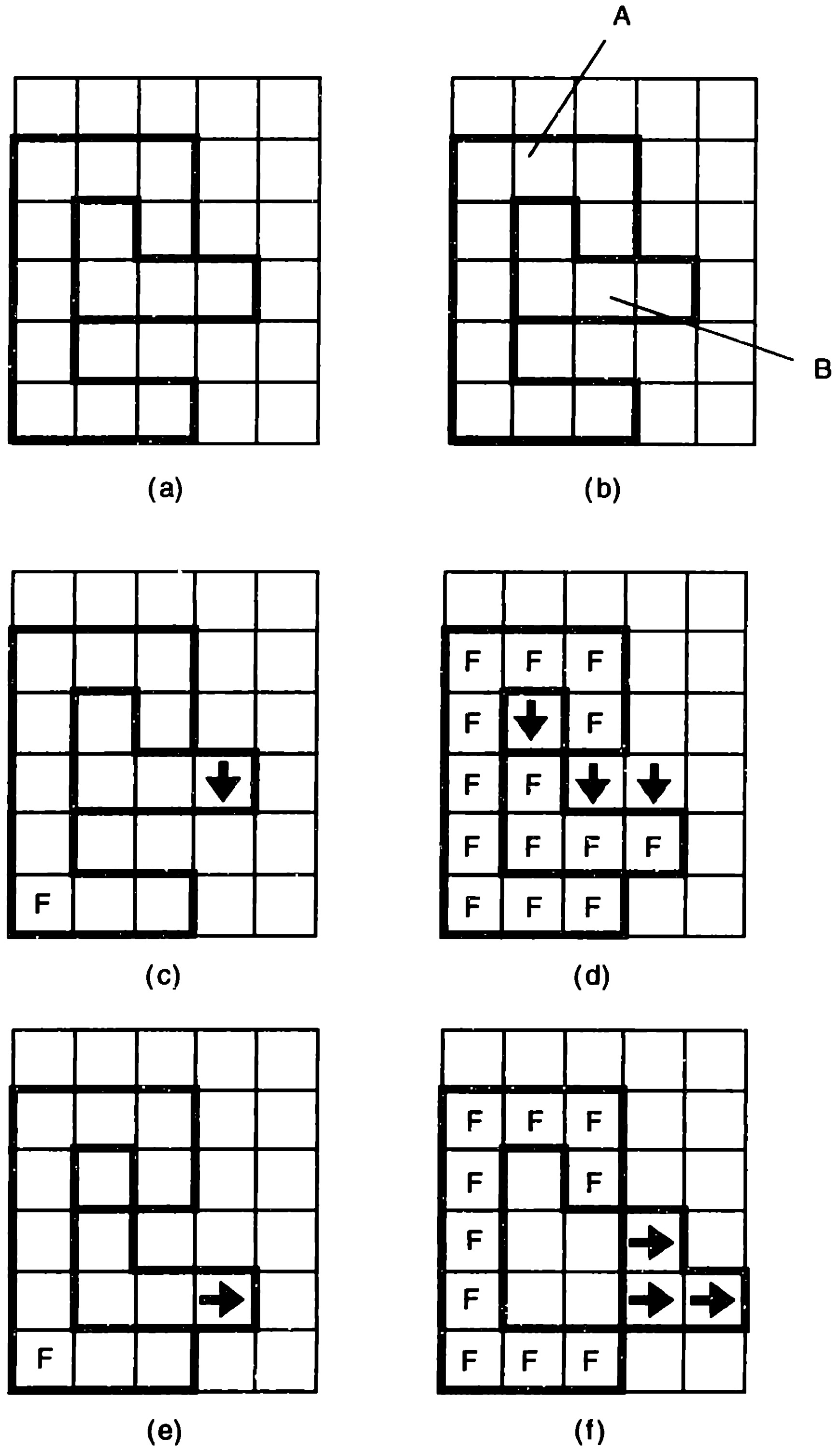


Figure 4.5 Example of a part extraction procedure.

Figure 4.6 shows another example illustrating an infeasible extraction. The only possible motion of Part B is upward while Part A is fixed, as shown in Figure 4.6a. After applying the MovePartUntilStop procedure, Part B collides with Part A as shown in Figure 4.6b. In this situation, Part B can move downward, but moving downward causes the same situation as shown in Figure 4.6a. As a result, it is found that the assembly can't be disassembled.

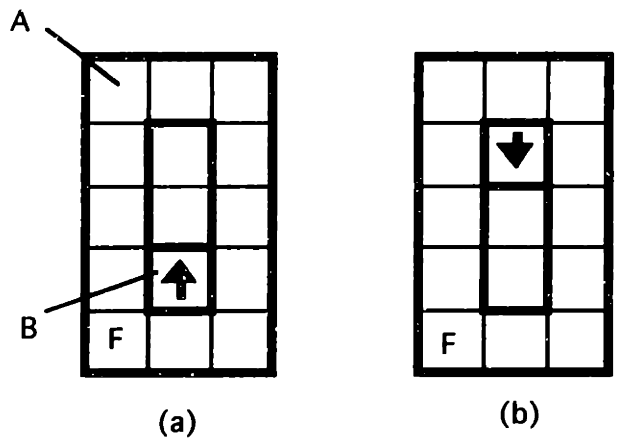


Figure 4.6 Example of no possible part extraction.

In the assembly sequence generation procedure, a series of part extraction processes are simulated to determine the feasibility of decomposing an assembly. When there are feasible extractions of subassemblies (possibly single parts), the same procedure is applied to the subassemblies generated until single-component subassemblies are reached.

4.3 Disassembly Sequence Generation Algorithm

During the disassembly sequence generation, it is necessary to generate a number of alternative ways to decompose a given assembly or subassembly into two subassemblies. These alternative decomposition processes can be referred as *disassembly tasks* or *tasks* (Homem de Mello and Sanderson 1991). A set of disassembly tasks for a given subassembly can be generated by applying the part extraction procedure to the subassembly with different combinations of fixed and moving parts as shown in list 4.6. During this procedure, it is important to note that the extraction of a part may move other parts, in effect creating a subassembly, and that each part is a solid rigid object, meaning, its shape remains unchanged during the disassembly tasks.

List 4.6 Algorithm to generate disassembly tasks.

```
procedure GenerateTasks( subassembly )
begin
    if it is a single-component assembly then return No Task
    repeat
        begin
            select a combination of fixed and moving parts
            apply ExtractSubassembly
            if it is feasible decomposition then store the task
        end
    until there are no more possible combinations
    return disassembly tasks
end
```

Since each task generated by the procedure shown in list 4.6 creates two subassemblies, the same procedure can be applied to the subassemblies in order to decompose them further into smaller subassemblies. Therefore, the complete set of feasible disassembly sequences, or AND/OR graph, for a given assembly can be generated by calling this procedure recursively until all the subassemblies are decomposed to single-part subassemblies.

During the task generation, since the same tasks are often generated from different combinations of fixed and moving parts, it is necessary to identify and eliminate these redundant tasks before decomposing subassemblies further. List 4.7 shows how the AND/OR graph for a given assembly can be generated using the part extraction and task generation procedures.

List 4.7 Algorithm to gcnerate a AND/OR graph.

```
procedure GenerateAND/ORgraph( assembly )
begin
    put assembly into the initial AND/OR graph
    repeat
        begin
            pick an undecomposed subassembly
            tasklist of the subassembly
                ← GenerateTasks( subassembly )
            eliminate redundant disassembly tasks from the tasklist
            for each task from the tasklist
            begin
                add the first subassembly into AND/OR graph
                add the second subassembly into AND/OR graph
            end
        end
    until all the subassemblies in the AND/OR graph are
            decomposed to single-part subassemblies
    return the final AND/OR graph
end
```

Figure 4.7a shows an example assembly composed of four parts in 10x10 design domain. Note that there exists some precedence relationships in the given assembly. For example part 3 cannot be disassembled from the given assembly without removing part 2 first. Figure 4.7b shows one feasible disassembly task where part 1 is fixed and part 2 is subjected to an extraction. Since this task is feasible, two subassemblies, {1, 3, 4} and {2}, will be generated from this task and stored. Figure 4.7c shows a feasible disassembly task for the subassembly, {1, 3, 4}. From the subassembly {1, 3} remaining, a set of feasible tasks will be determined that yield only single-component subassemblies. The AND/OR graph generated by the planner is shown in figure 4.7d.

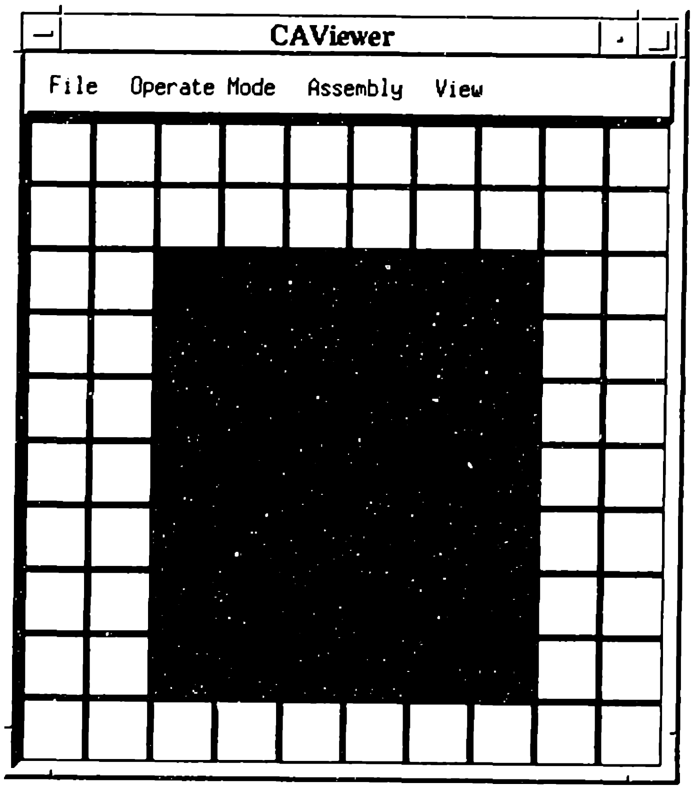


Figure 4.7a Example assembly with four components.

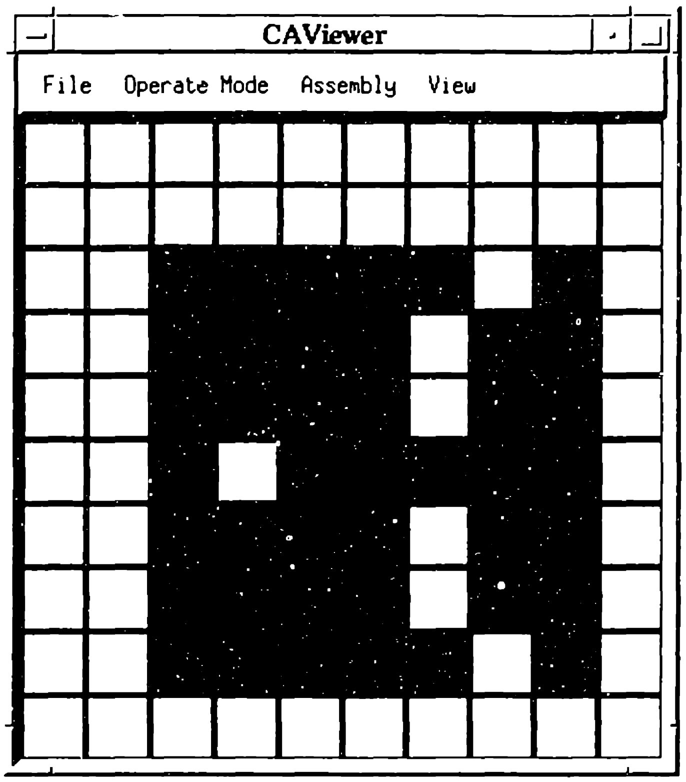


Figure 4.7b Feasible decomposition task for the given assembly.

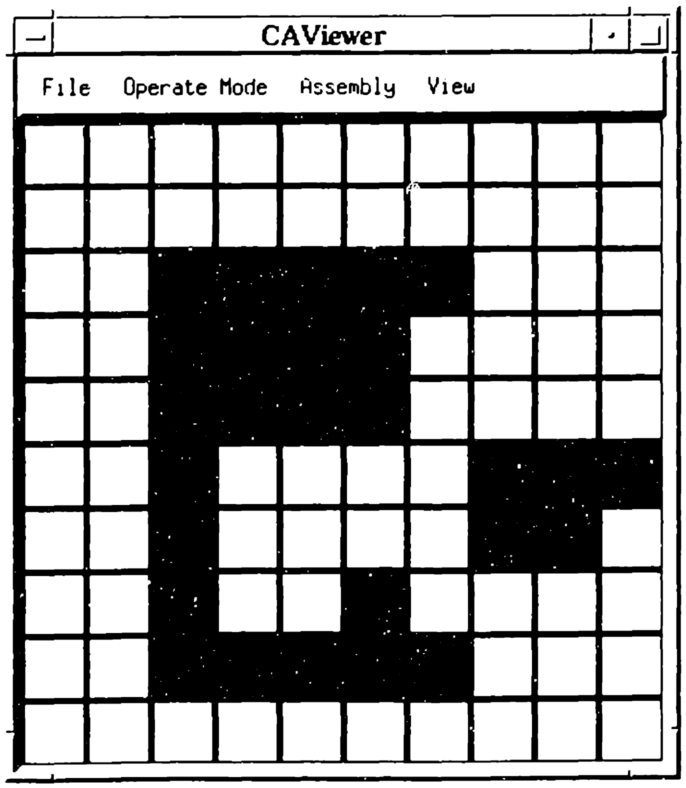


Figure 4.7c Feasible decomposition task for subassembly { 1, 3, 4 }.

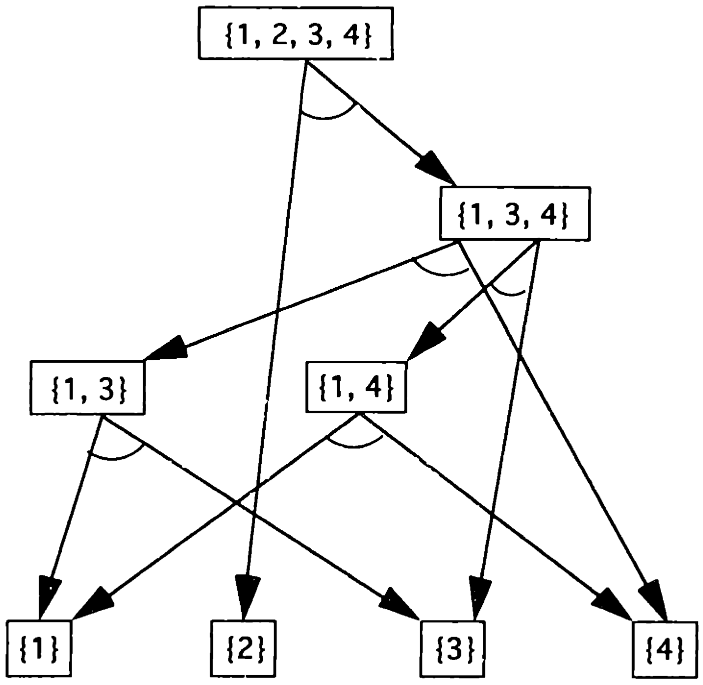


Figure 4.7d AND/OR graph for the example in figure 10a.

4.4 Examples

In this section additional examples are provided in order to yield some initial data on the system's capabilities and runtime performance. At this point it should be noted, however, that our implementation is on a serial processor engineering workstation (Silicon Graphics® Indigo II Extreme®, approximately 85 MIPS). The immediate plan is to develop a parallel processing implementation and an initial parallel implementation is discussed in chapter 6. The data presented below suggest that this is a necessity for practically complex assemblies.

Figure 4.8 gives some indication of the relationship between runtime and assembly complexity, while keeping the discretization size constant. Figures 4.8a, 4.8b, and 4.8c show 4, 6, and 8 part assemblies on a constant array size of 10×10. For the task of generating the AND/OR graph for each assembly, figure 4.8d shows the number of CPU seconds required versus part count. Although many practically interesting subassemblies

Silicon Graphics and Indigo II Extreme are registered trademarks of Silicon Graphics, Inc.

will have 8 or fewer parts, this initial data indicate that even with a parallel implementation, hierarchical subdivision of larger assemblies may be needed.

Figure 4.8a, together with figures 4.9a and 4.9b, depict the same (approximately) assembly modeled on increasing array sizes. Figure 4.9c shows the increase in runtime versus array size. Since a parallel implementation will evaluate an entire array simultaneously, this effect should not be problematic in a parallel implementation.

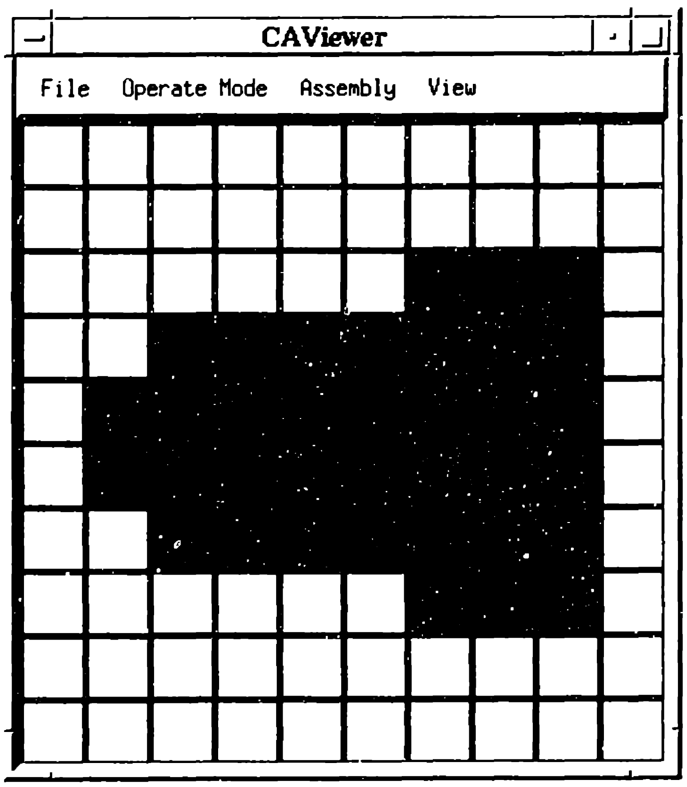


Figure 4.8a 4 part assembly in 10x10 domain.

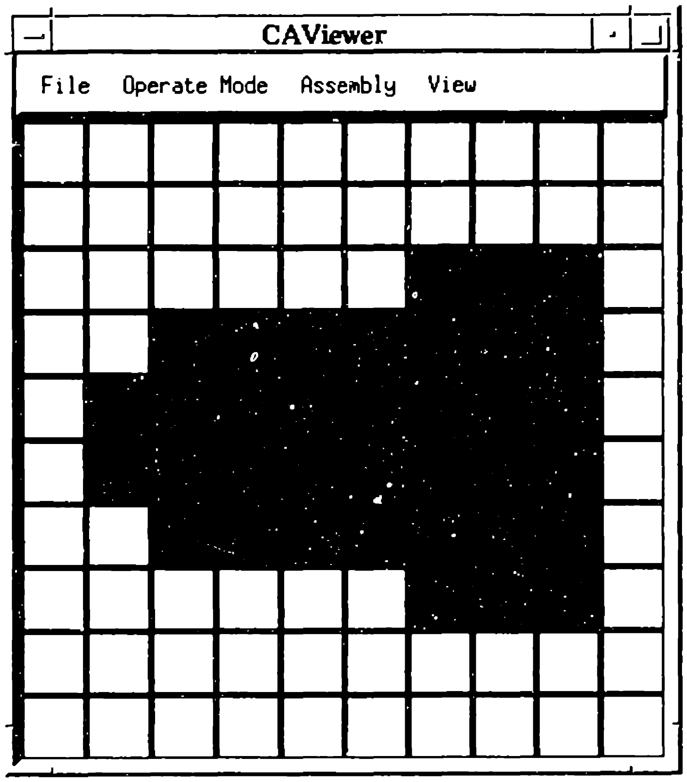


Figure 4.8b 6 part assembly in 10x10 domain.

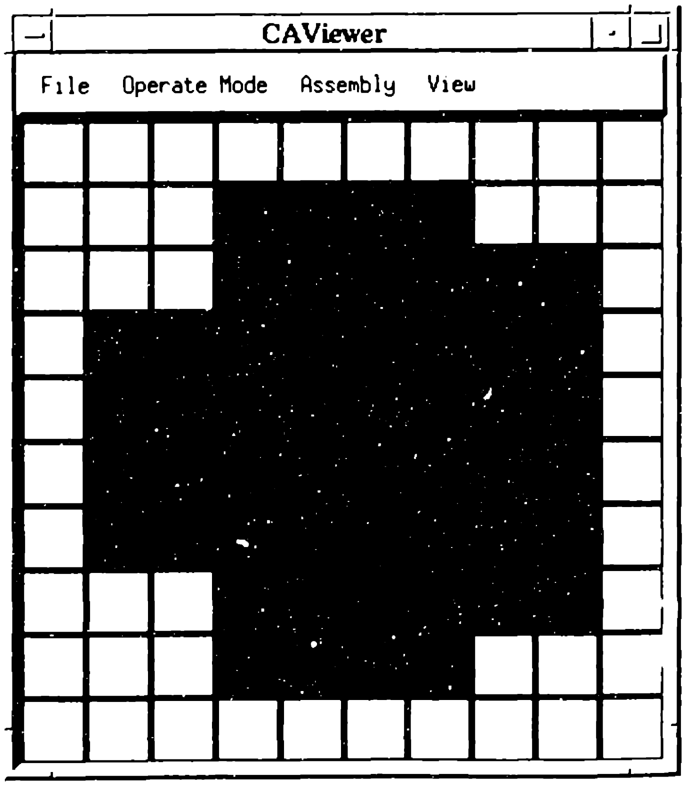


Figure 4.8c 8 part assembly in 10x10 domain.

No. of Parts	4	6	8
CPU (sec)	8.74	51.75	311.20

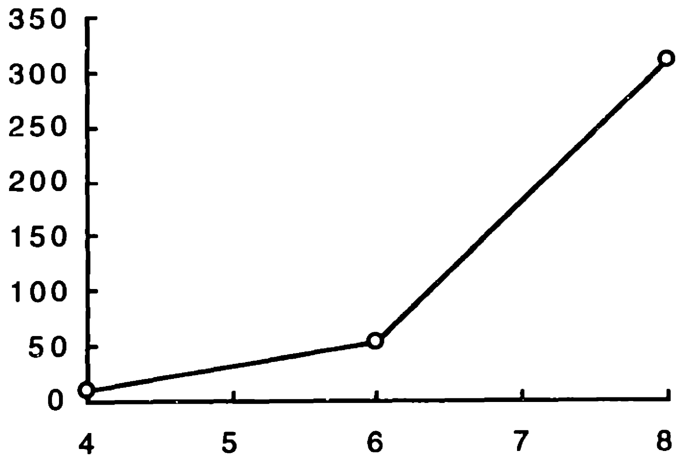


Figure 4.8d CPU seconds versus part count

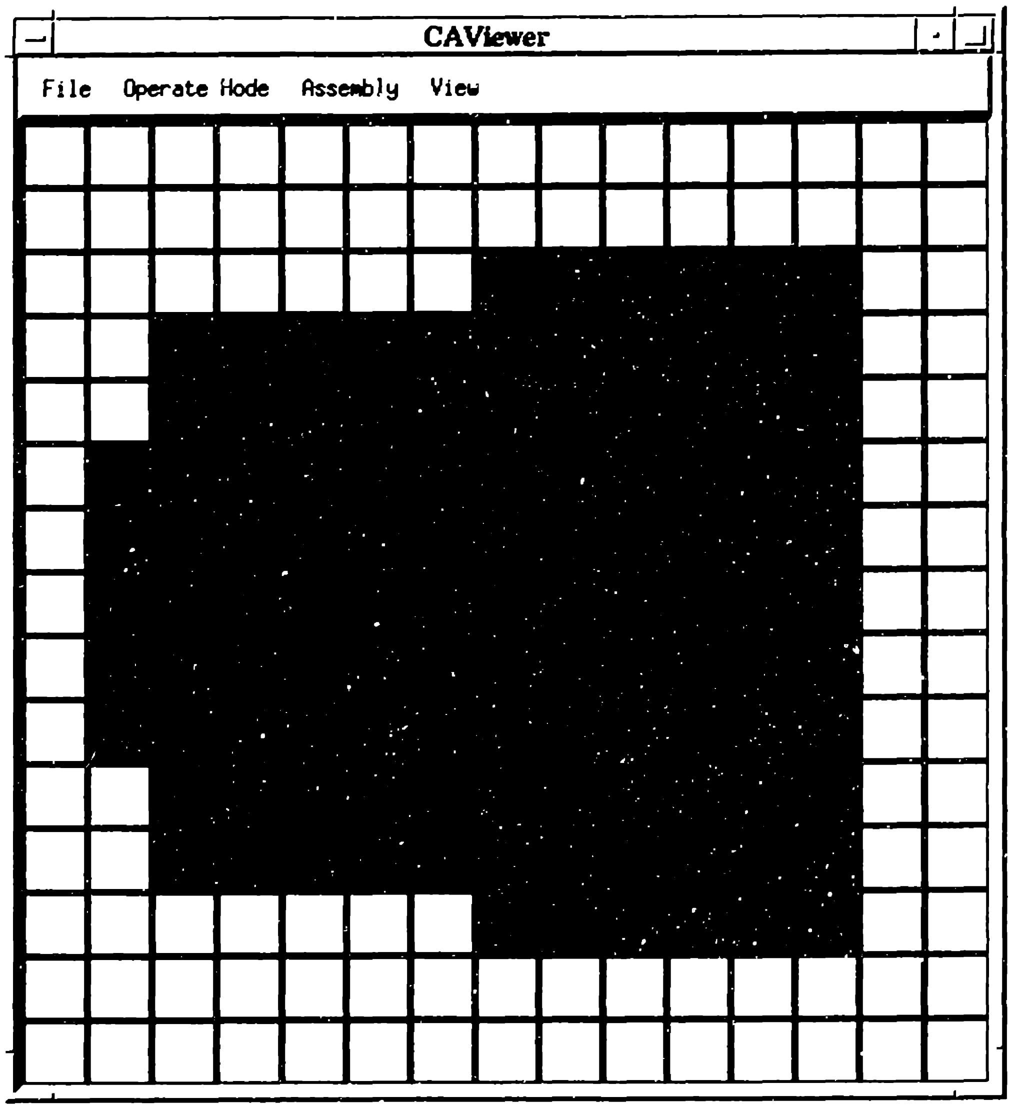


Figure 4.9a 4 part assembly in 15x15 domain.

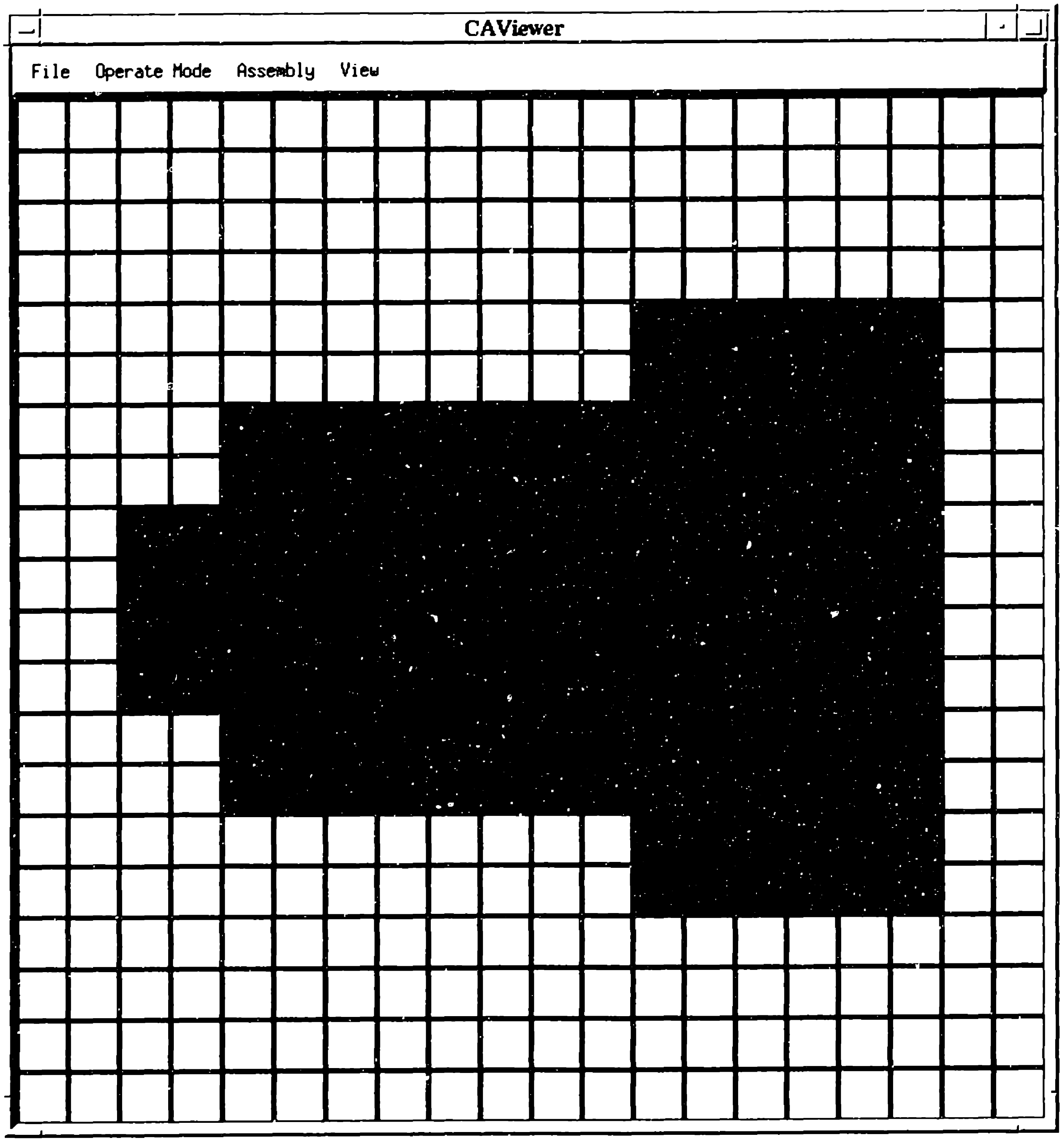


Figure 4.9b 4 part assembly in 20x20 domain.

Size of Domain	10	15	20
CPU (sec)	8.74	29.50	58.12

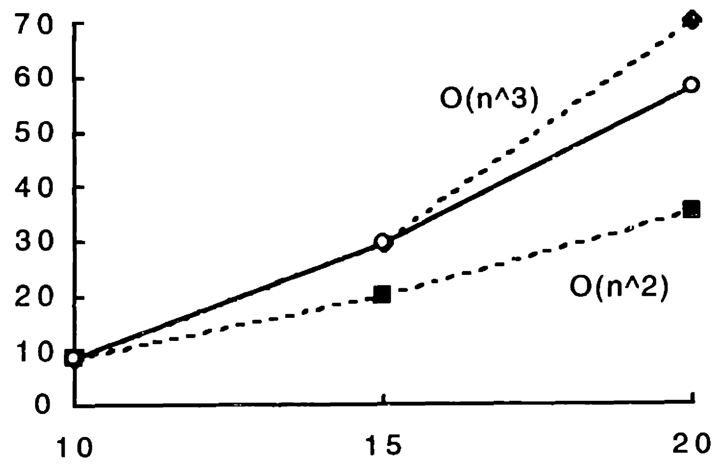


Figure 4.9c CPU seconds versus array size.

5 Assembly Sequence Optimization

5.1 Overview

This chapter first describes the basic scheme for selecting assembly sequences and how each assembly sequence is evaluated during the selection procedure. Then, genetic algorithms and branch and bound search techniques are described. These optimization techniques are used to find good assembly sequences for a given assembly with respect to pre-defined evaluation criteria. Also, the implementation of these techniques is detailed with some examples and the results of both techniques are discussed.

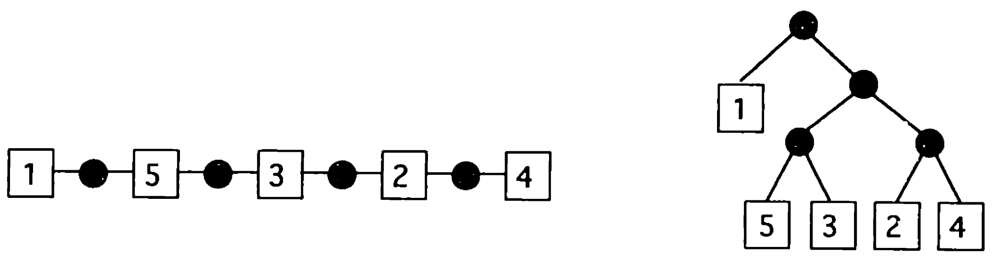
5.2 Selecting Assembly Sequences

5.2.1 Introduction

Once feasible assembly sequences have been generated, the next step is to choose the best or a few good sequences based on important criteria. These criteria must correspond to a general knowledge of assembly processes in production which result in a low cost assembly system. Even if an assembly sequence is feasible in terms of geometric and relational constraints, the sequence is not favorable if it requires difficult assembly tasks, such as frequent turn-overs of a part or subassembly, and placement in unsecured or unstable positions. Furthermore, as the number of parts in an assembly increases, the number of feasible assembly sequences increases exponentially. Therefore, it is desirable to have an automated method that selects only a few candidate sequences with pre-defined criteria, and to allow the designer to finally choose one.

5.2.2 Linear and Parallel Assembly Sequences

Assembly sequences can be either linear or parallel. A *linear* assembly sequence assembles only a single part at a time, which means that multipart subassemblies are not allowed. It is, however, often efficient to divide an assembly into independent subassemblies so that each subassembly can be assembled at a different assembly cell. This type of assembling method is called *parallel* assembly. These subassemblies are then brought together to be assembled into bigger subassemblies or the final assembly. Figure 5.1 illustrates the schematic representation of these assembly sequences.



(a) String representation for linear assembly sequence

(b) Binary tree representation for parallel assembly sequence

Figure 5.1 Schematic representation of assembly sequences types.

The numbers in square boxes and the nodes in Figure 5.1 represent the label of a part to be assembled and an assembly process associated with parts, respectively. Figure 5.1a indicates that parts 5, 3, 2, and 4 are sequentially assembled into part 1. Figure 5.1b shows that parts 3 and 5 are first assembled being independent of the subassembly of part 2 and 4. Then, the subassembly of parts 3 and 5 and the subassembly of parts 2 and 4 are assembled to produce another subassembly of parts 2, 3, 4, and 5, which is assembled with part 1 to make the final assembly. In this investigation, only the linear assembly sequence is considered because it requires a less complicated representation for an implementation, but still allows a general approach for the problem of optimizing the assembly sequence.

5.2.3 Assemblability Evaluation

A qualitative or quantitative method to evaluate an assembly sequence is necessary in order to select the lowest cost assembly sequence from available alternatives. An assembly sequence of many parts is generally evaluated based on its *assemblability*. This assemblability pertains to characteristics or attributes of an assembly sequence that can be either desirable or undesirable from a manufacturing standpoint. There are a number of research approaches related to qualitative and quantitative evaluation methods for selecting a good assembly process ((Boothroyd and Alting 1992), (Miyakawa, Ohashi *et al.* 1988), and (Miyakawa, Ohashi *et al.* 1990)).

To systematically evaluate the assemblability of an assembly, it is necessary to classify the attributes which affect the degree of difficulty associated with assembly operations. These attributes include part locatibility or stability, number of turn-overs required during the assembly process, number of parts to be assembled, direction of assembling action, *etc.*

It is important to note that the evaluation of an assembly sequence is actually determined by evaluating a disassembly sequence, which is obtained by reversing the part labels of the assembly sequence under consideration. Therefore, instead of evaluating each assembly task of putting a part to another part or subassembly, the task of removing a part from a subassembly or assembly is evaluated.

5.2.4 Evaluation Criteria for Assembly Sequence

To evaluate an assembly sequence, assembly tasks that constitute the sequence are evaluated sequentially and the evaluation scores of these tasks is summed as the evaluation score of the assembly sequence. The assembly tasks are evaluated in such an order that they represent a linear disassembly sequence. The criteria used to evaluate each assembly task is based on assemblability.

Three criteria are used to evaluate an assembly task: feasibility, ease of motion and assembly direction, and stability of an assembled component. Equation 5.1 shows how the evaluation score of an assembly sequence is determined based on the evaluation criteria.

$$\text{Evaluation Score}(s) = \sum_{i=1}^{n-1} (E_{feasibility}(s_i) + E_{easiness}(s_i) + E_{stability}(s_i)) \qquad \text{Equation 5.1}$$

where n is the number of parts in assembly and S_i indicates the ith assembly task in an assembly sequence, S.

$$E_{feasibility}(s_i) = \begin{cases} 20\text{: feasible assembly task} \\ 0\text{: infeasible assembly task} \end{cases} \qquad \text{Equation 5.2}$$

$$E_{easiness}(s_i) = \begin{cases} 0\text{: infeasible assembly task} \\ 2\text{: downward disassembly task} \\ 6\text{: sideway disassembly task} \\ 10\text{: upward disassemby task} \end{cases}$$ Equation 5.3

$$E_{stability}(s_i) = \begin{cases} 0\text{: unstable in all directions} \\ 3\text{: stable in one direction} \\ 8\text{: stable in two directions} \\ 10\text{: stable in all directions} \end{cases}$$ Equation 5.4

Equation 5.3 shows that the easiness of an assembly task is defined by its direction. An assembly task in which a part is placed on top of a part or subassembly is the least expensive operation. On the contrary, assembling a part into the bottom of a subassembly is costly because it is in general difficult to hold the part in its position. This operation often leads to an alternative that turns over the subassembly in order to obtain the cheaper operation. However, this turn-over operation consequently increases the final manufacturing cost. The directions of assembly are limited to three orthogonal directions and parts will be moved towards the assembly site in a straight line. This assumption is reasonable for most assembly cases and simplifies the physical reasoning of the system. During the assemblability evaluation, directions in equation 5.3 are determined from how an assembly is defined by the user and displayed on the screen. For example, although an assembly shown in figure 5.7 shows vertically symmetric parts and thus implies that the top part would be part no. 1 or part no. 4, the actual upward direction for this assembly is defined by the direction toward the menu bar of user interface from the assembly.

The stability of a part during the assembly task, shown in Equation 5.4, is defined by the number of supports for the part. If there exist no parts in the subassembly which support and secure the assembled part in its position, the part is defined as ***unstable*** in the assembly task. If a part is sec ıred in its position after the assembly task and thus it is not allowed to move in any direction except the direction in which the part was assembled, the part is said to be stable in all three directions. Similarly, a part can be unstable in one or two directions. An assembly task with an unstable part is costly because an extra part holder is required to hold the part during the assembly process. It should be noted that the assemblability score given in the above equations are assigned by weighing the significance of each aspect of the assembly task.

5.3 Genetic Algorithms Approach

5.3.1 Introduction

The genetic algorithm is used to search for good linear assembly sequences of a multiple part assembly. Assembly sequences that maximize the evaluation score as well as satisfy both geometric and relational constraints among parts are sought. The evaluation score is assigned to each sequence as its fitness value for its corresponding chromosome after evaluating the sequence with respect to the assemblability criteria defined in equation 5.1. For a general introduction to genetic algorithms, please refer chapter 2.

5.3.2 Implementation

A chromosome in this investigation represents a list of decimal numbers, each of which indicates a part in an assembly. Since only linear assembly sequences are considered for the optimization problem, the list of part numbers is sufficient to represent an assembly sequence. For the parallel assembly sequence, a binary tree structure and more complex schemes for the representation of a sequence and GA operators will be required.

List 5.1 describes an algorithm for finding good sequences using the genetic algorithm. In each generation during the search, sequences are evaluated, as shown in List 5.2, based on the criteria described in section 5.2.4. After being evaluated, chromosomes are selected based on their fitness scores and then GA operations are performed on them.

List 5.1 Algorithm for searching sequences using the genetic algorithm

```
procedure ga_main()
begin
    set ga parameters
    create the initial population with randomly generated chromosomes
    while ga not finished
    begin
        foreach chromosome in the population
            EvaluateFitness( chromosome )
        select chromosomes and perform GA operations
        mutate some chromosomes
        evolve the generation
    end
    return the best chromosome from the last generation
end
```

List 5.2 Algorithm for evaluating a sequence

```
procedure EvaluateFitness( chromosome )
begin
    sequence list ← DECODE( chromosome )
    while the sequence list has more than one part
    begin
        part ← pop( sequence list )
        subassembly ← sequence list - part
        evaluate the feasibility of decomposing the part from subassembly
        fitness score = feasibility score
        if the task is feasible
        begin
            evaluate the easiness of task
            fitness score = fitness score + easiness score
            evaluate the stability of task
            fitness score = fitness score + stability score
        end
    end
    return fitness score
end
```

5.3.3 GA Operators for Crossover and Mutation

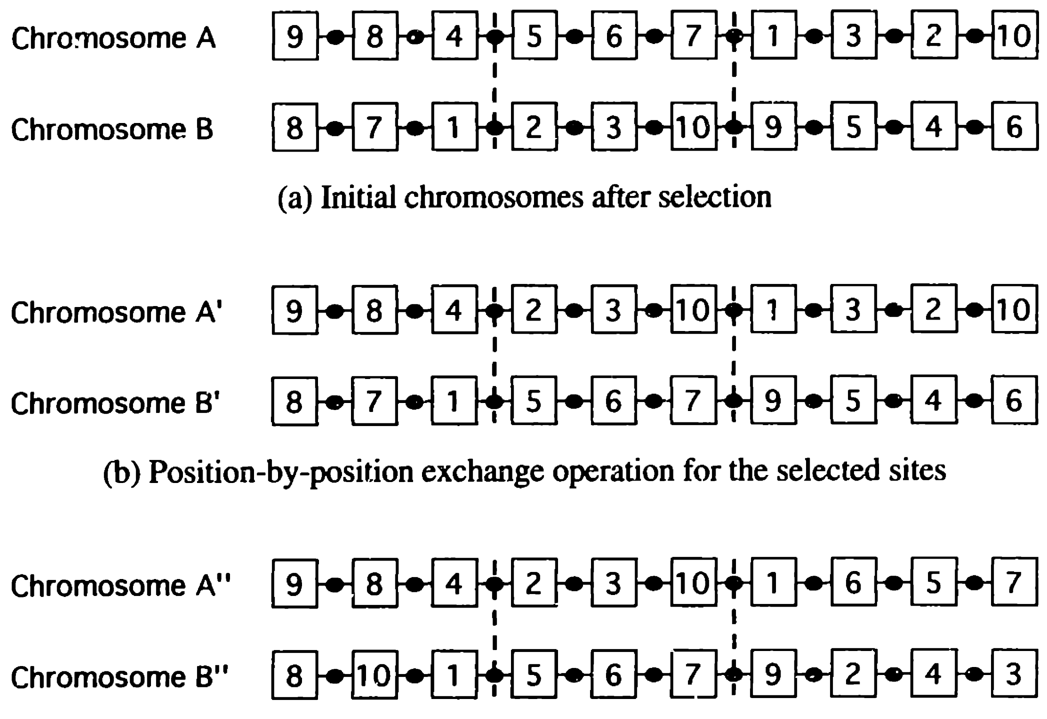
As explained in chapter 2, the genetic algorithm utilizes a series of operations on the population of chromosomes in order to improve the overall fitness of a generation. These operations are selection, crossover, and mutation. Among these operations, in general, crossover and mutation operators have significant impact on the overall performance of genetic algorithms. Two distinct operations for both crossover and mutation were investigated and their effects on the GA performance were examined. For a comprehensive review of these operators, please refer to (Goldberg 1989).

Partial Matched Crossover and Ordered Crossover

In this investigation, since chromosomes that represent linear assembly sequences contain ordered part identification numbers or labels, the genetic algorithm must use a crossover operation that conserves the ordering characteristics of the chromosomes.

Figure 5.2 describes how the partial matched crossover (PMX) operator performs the exchange of selected sites of chromosomes. First, selected mating chromosomes are aligned and crossing sites defined by two points on the chromosomes are determined as shown in figure 5.2a. Then, the PMX operator performs the positionwise exchange of these sites. Since a new set of ordered part labels is replaced in the switching sites for each chromosome, there exist the same or multiple part labels within a chromosome as shown in figure 5.2b. So, the multiple numbers in the unchanged sites must be switched with numbers that have been transferred into the mate. Figure 5.3c shows the resultant offspring chromosomes after PMX.

In PMX, the ordering information of child chromosomes is partially determined by their parents. It should be also noted that PMX tends to conserve the ordering information of part labels with their absolute position.



(a) Initial chromosomes after selection

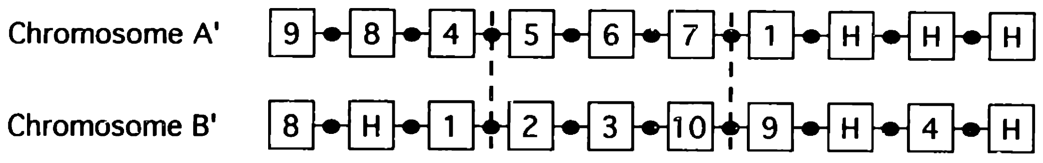
(b) Position-by-position exchange operation for the selected sites

(c) Final chromosomes after determining ordering information

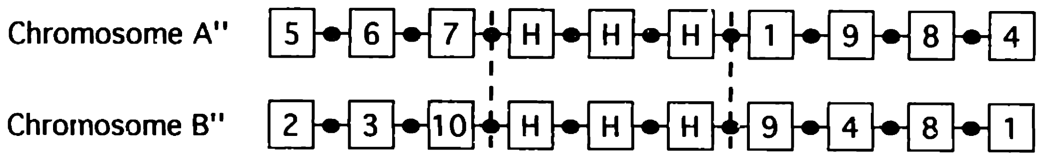
Figure 5.2 Partial matched crossover.

While PMX tends to respect the absolute position of part labels during the crossover by performing point-by-point exchanges to effect the mapping, the ordered crossover (OX) maintains the relative position of part numbers, which are inherited from the parent chromosomes, using sliding operations during the crossover.

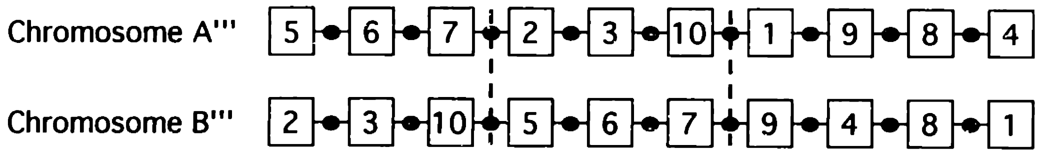
After the switching sites are selected as shown in figure 5.2a, the OX operator creates holes for part labels, which are outside the exchange site and the same as part labels in the exchange site of the mate as shown in figure 5.3a. Figure 5.3b shows these holes are then filled by sliding part labels starting from the second crossing point. After completing the exchange of selected sites of mating chromosomes, OX creates the offspring of chromosomes A and B as shown in figure 5.3c.



(a) Creating holes for nodes of other strings' selected sites



(b) Filling holes with a sliding motion starting after the second crossing site

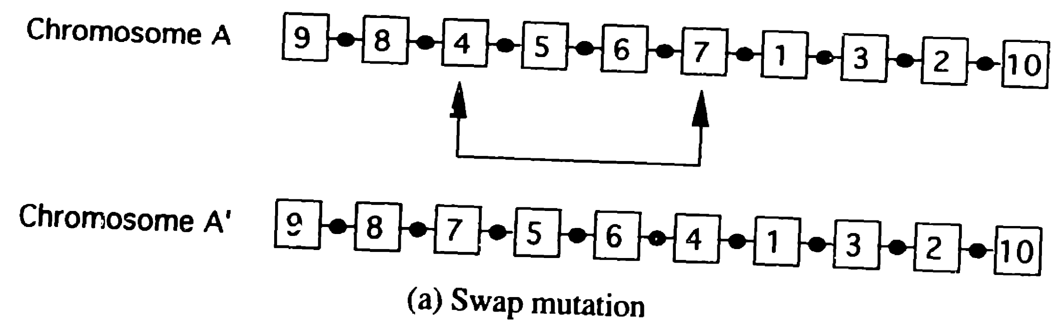


(c) Final chromosomes after copying the mate's the selected sites

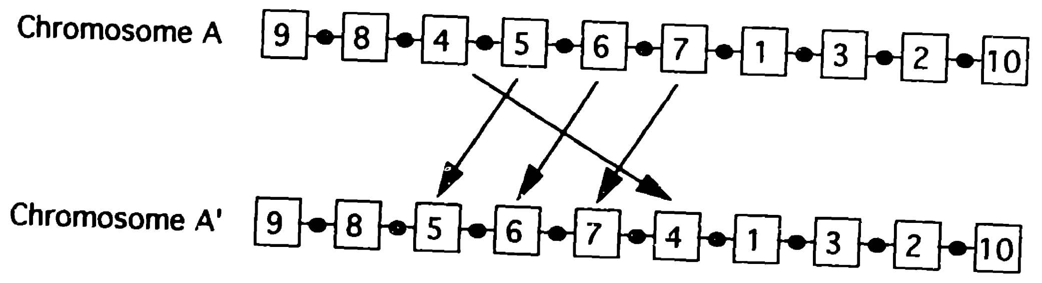
Figure 5.3 Ordered crossover.

<u>Swap Mutation and Shift Mutation</u>

Figure 5.4a shows that the swap mutation operator selects two nodal points and simply swaps the two nodes based on the mutation probability. Similar to PMX, the swap mutation conserves the absolute position of the part labels within a chromosome. On the contrary, as shown in figure 5.4b, the shift mutation conserves the relative position of a part label by selecting a node and inserting it into a different site after shifting other nodes so as to fill up a hole.



(a) Swap mutation



(b) Shift mutation

Figure 5.4 Swap and shift mutation.

5.3.4 Examples

The implementations of the algorithms, described in section 5.3.2, are tested for four example assemblies for the purpose of showing the performance of the system in searching for optimal assembly sequences as well as determining good combinations of parameters for the genetic algorithm. The assemblies are shown in figures 5.5, 5.6, 5.7, and 5.8. The example shown in Figure 5.8 of a twelve part assembly is an example that is similar in shape and part count to one used in (De Fazio and Whitney 1987).

Since the genetic algorithm is a stochastic searching method, it does not always find the same optimal results in every optimization search. Therefore, the results given in this chapter with regards to the genetic algorithm are generated from the average of ten runs for each example.

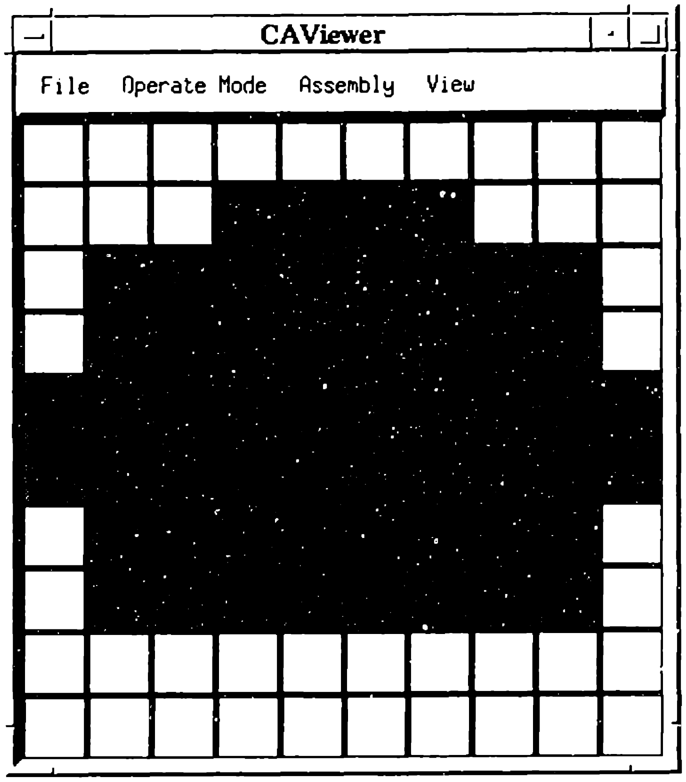


Figure 5.5 Example assembly: Test 1.

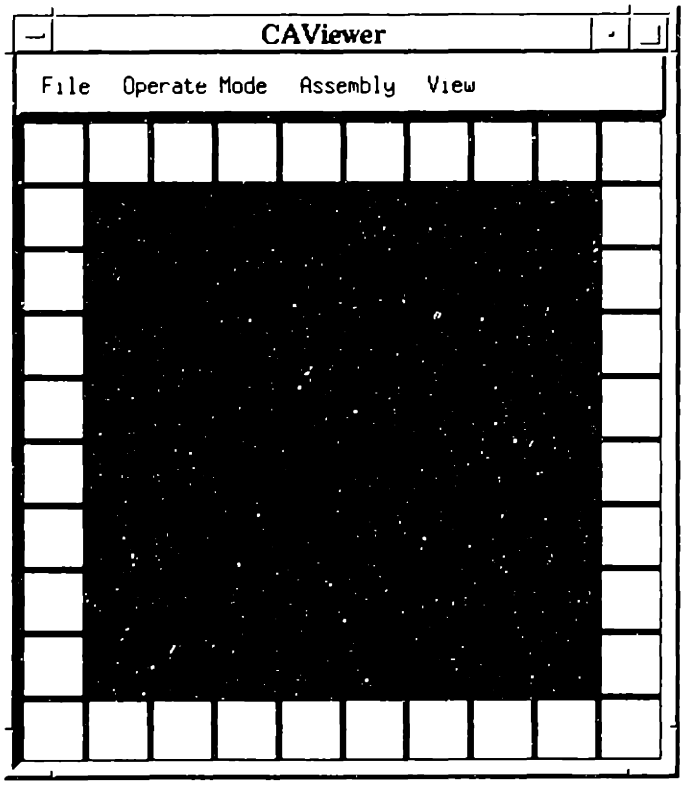


Figure 5.6 Example assembly: Test 2.

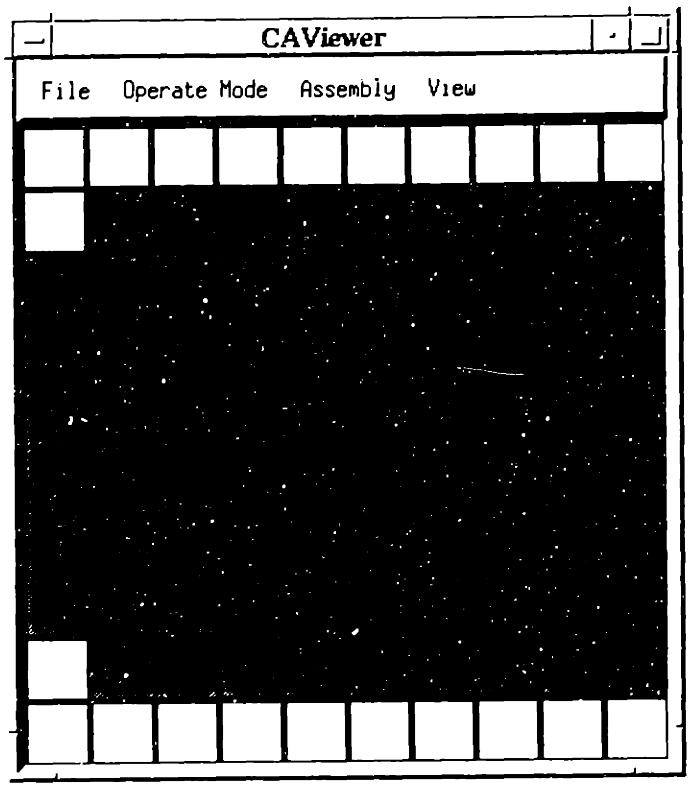


Figure 5.7 Example assembly: Test 3.

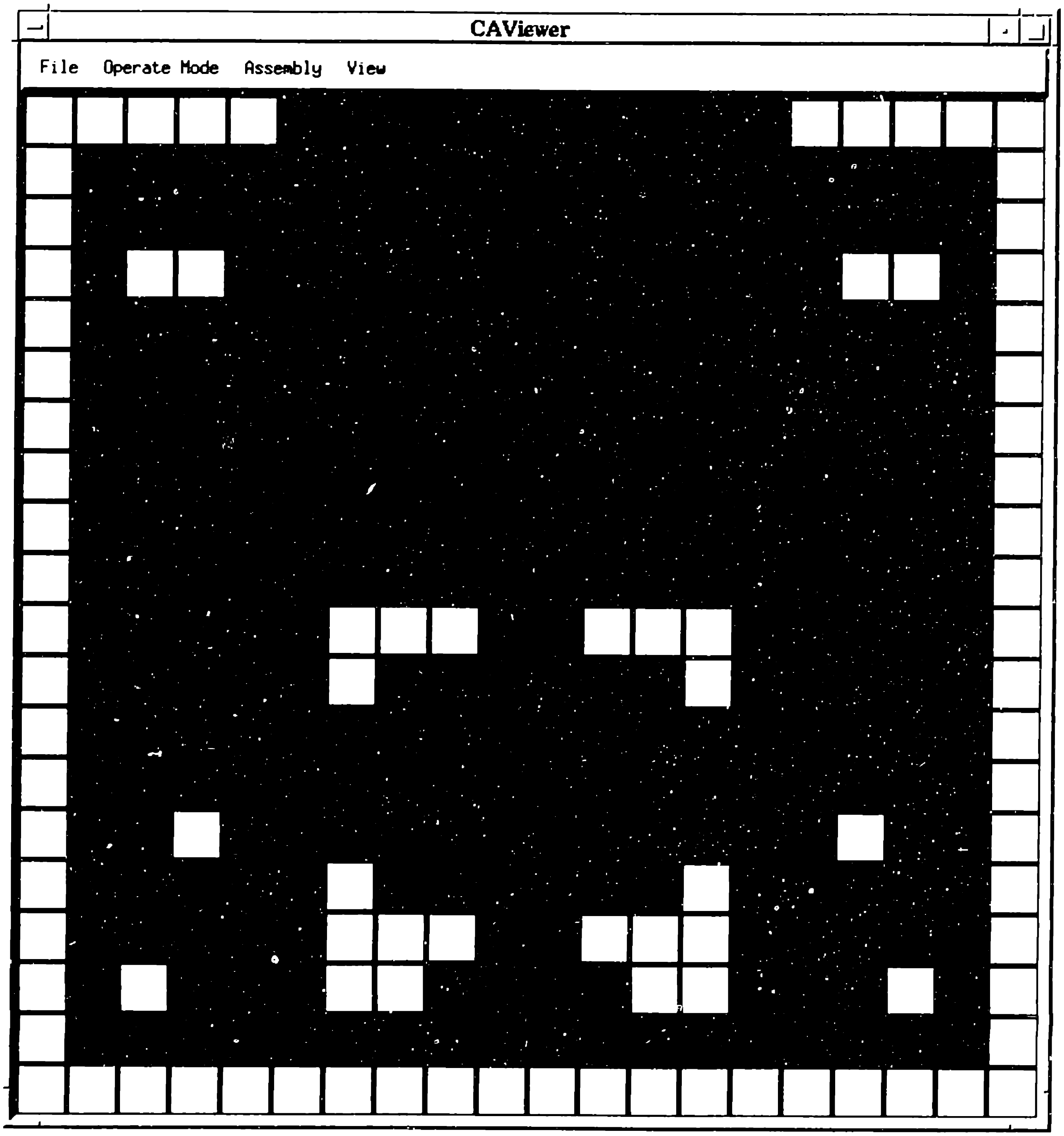


Figure 5.8 Example assembly: Test 4

Figure 5.9 shows the performance of the implementation for finding the optimal linear assembly sequence with respect to different population sizes. This particular example uses partial match crossover, shift mutation, and a steady state genetic algorithm. The number of evaluations axis indicates the total number of function calls to evaluate chromosomes, which is normalized with respect to those of the 50-population-size examples. In general, genetic algorithms perform better with a bigger population size because they then are able to search the larger space at each generation during the evolution. However, since the large population size also increases the computational cost, it is generally necessary to determine the appropriate population size for a certain type of problem. The fitness to optimum ratios in figure 5.9 were determined by normalizing the evaluation scores of assembly sequences with respect to an optimal assembly sequence. The optimal sequence was found by inspection and also validated using the branch and bound method since it always finds the optimal assembly sequence of a given assembly.

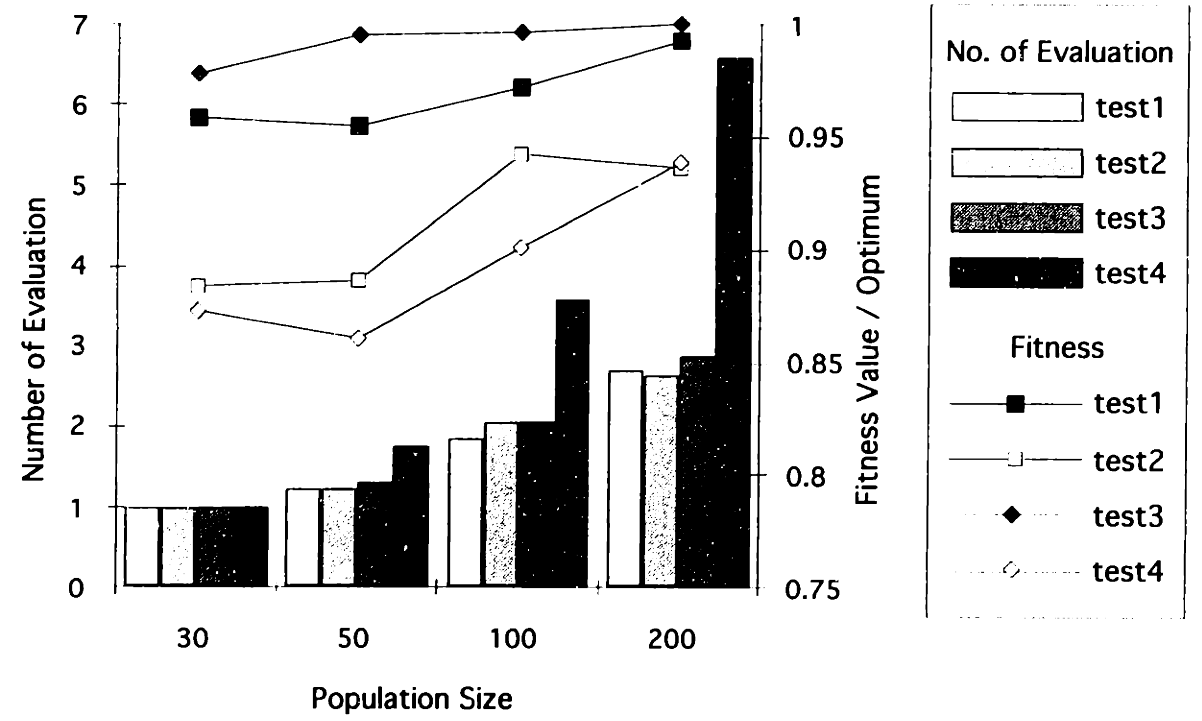


Figure 5.9 Population versus number of evaluation and fitness.

A number of different mutation probabilities were used for each assembly example and their performance measures, number of evaluations and fitness ratio are shown in figure 5.10. For this test, a population size of 100, partial match crossover, the shift mutation operator, and a steady state genetic algorithm were used. In this result, performance measures of different mutation probabilities are normalized with respect to that of the 0.01-mutation-rate example. The result indicates that the optimization runs with the mutation probability of 1 percent generate good result with a relatively small number of evaluation calls.

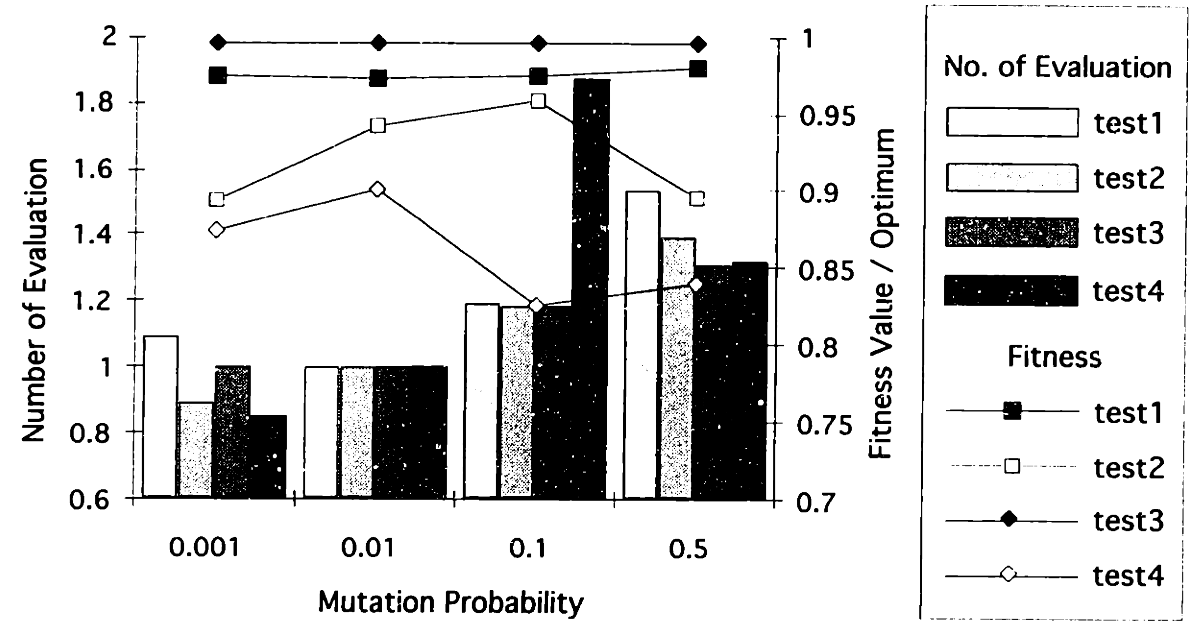


Figure 5.10 Mutation probability vs. no. of eval. and fitness.

In figure 5.11, four combinations of crossover and mutation operators are used to determine the best combination of GA operators. Population size of 100 was used with crossover probability of 1.0 and mutation probability of 0.01. The results are normalized with respect to the performance of the GA optimization with ordered crossover and shift mutation. Overall, the shift mutation with ordered crossover performed well compared to other combinations. This result reflects the physical implication of the ordered crossover and the shift mutation because both operations conserve the relative sequential position of a part or parts in an assembly sequence instead of absolute position.

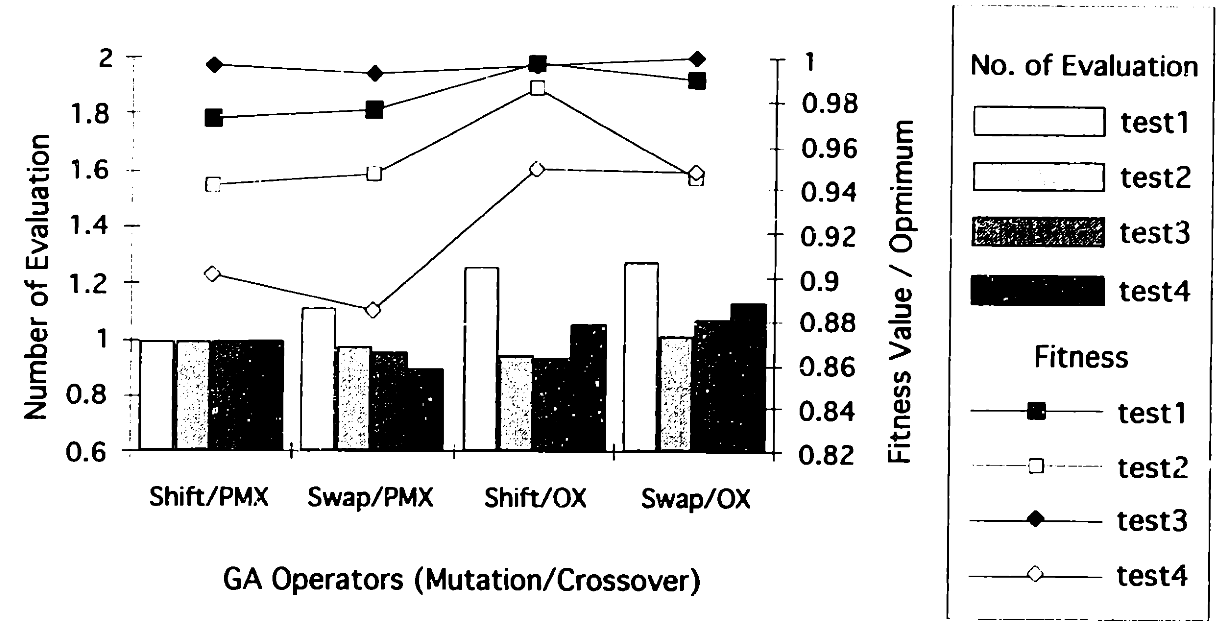


Figure 5.11 GA Operators vs. no. of eval. and fitness.

5.4 Branch And Bound Approach

5.3.1 Introduction

As described in chapter 2, the branch and bound method systematically subdivides the total set of solutions under consideration into smaller sets, in such a way that large subsets of solutions may be pruned from the further examination without evaluating each solution in these subsets. Here, the search space of all possible sequential combinations of the part labels in an assembly is considered. However, since branches with infeasible assembly tasks or relatively low estimation scores are discarded in the early stage of search, the algorithm performed efficiently for most examples tested. Here, the estimation score implies the estimated evaluation score of a branch in which evaluation scores of unknown variables below the branch are assumed to be maximum (i.e. the best possible). An evaluation algorithm for the linear assembly sequence using this approach is described and explained with an example. Some special examples are also provided to show that the branch and bound doesn't perform well for certain types of problems.

5.3.2 Implementation

Lists 5.3 and 5.4 show procedures of the branch and bound technique, which are used to find the best assembly sequence based on the same selection criteria described in the previous section. When an assembly is provided to the Brach and bound procedure shown in list 5.3, branches are generated by first choosing a part from the given assembly in such a way that each part in the assembly is considered as a part to be disassembled. Then, a node for each combination of the part and subassembly is created and its assemblability is evaluated using an evaluation function similar to the one described in list 5.2. After all the nodes are evaluated, they are sorted according to their evaluation scores so that a node with the highest estimation score is evaluated first among the others. By sorting the nodes prior to branching further, the efficiency of the algorithm can be enhanced because it first examines a node that is likely to contain the solution and, thus, obtain the highest bound in the early stage of searching. Therefore, most branches that have lower evaluation scores than the bound can be pruned.

List 5.3 Algorithm to find the best sequence.

```
procedure BNB_main()
begin
    get full assembly from domain
    current score = 0
    bound = 0
    current sequence = NULL
    best sequence = NULL
    BNB( full assembly, current score, bound, current sequence, best sequence )
    return the best sequence
end
```

List 5.4 Branch And Bound Algorithm.

```
procedure BNB( assembly, current score, bound, current sequence, best sequence )
begin
    if ( number of component in the assembly > 1 )
    begin
        tasklist ← NULL
        foreach component in assembly
        begin
            subassembly ← assembly - component
            tasknode ← TaskNode( subassembly, comp)
            evaluate the tasknode using a lookup table
            add tasknode into tasklist
        end
        sort tasknodes with respect to their evaluation score
        foreach tasknode from the tasklist
        while( current_score + evaluation score of the tasknode
                + MaxTaskScore[1] × ( number of subassembly - 1 ) > bound )
```

[1] MaxTaskScore is defined as the best possible evaluation score for an assembly task. Therefore, the assembly task is assumed to be a feasible, downward task, and stable in all directions.

```
            begin
                current sequence ← current sequence + component of the tasknode
                BNB( subassembly of tasknode, current_score + task score, bound,
                    current sequence, best sequence )
                delete current sequence
            end
        end
else
        begin
            if current score > bound
            begin
                bound ← current score
                update current sequence
                best sequence ← current_sequence
            end
        end(else)
        return the best sequence
end
```

Figure 5.12 shows an example assembly composed of five parts. When this assembly is provided as an input to the branch and bound algorithms described above, the algorithm in list 5.3 first generates branches, each of whose nodes contains a part and its corresponding subassembly. For example, the node of branch *a* in figure 5.13 indicates that part 1 is disassembled from the given assembly with parts 1, 2, 3, 4, and 5 generating an subassembly { 2, 3, 4, 5 }. Once these nodes are generated, they are sorted with respect to their current evaluation score (C.S.) so that a node with the highest evaluation score is branched further. Repeating this procedure until a complete disassembly sequence is found, figure 5.13 shows that the first sequence found by the algorithm is 1-2-3-5-4 with an evaluation score of 153. This score is then assigned as a bound of the branch and bound search, which determines whether a branch needs to be examined further for search of a better solution. After obtaining the first bound, the algorithm proceeds the search with comparing the expected evaluation scores of remaining branches to the bound. Figure 5.14 shows how the algorithm determines the expected evaluation score of a branch and update the current bound when it finds a better solution. The expected evaluation score of the branch *f* shown in figure 5.14 is determined by summing the accumulative scores of its incomplete disassembly sequence, its current score, and the best possible score expected below the branch. Since the estimated score is determined to be 158, the branch is further examined and, in this case, a better sequence with evaluation score of 156 is found in branch *g*. Therefore, the bound is updated with the higher score and, thus, the algorithm is able to prune more branches with estimated scores lower than the new bound.

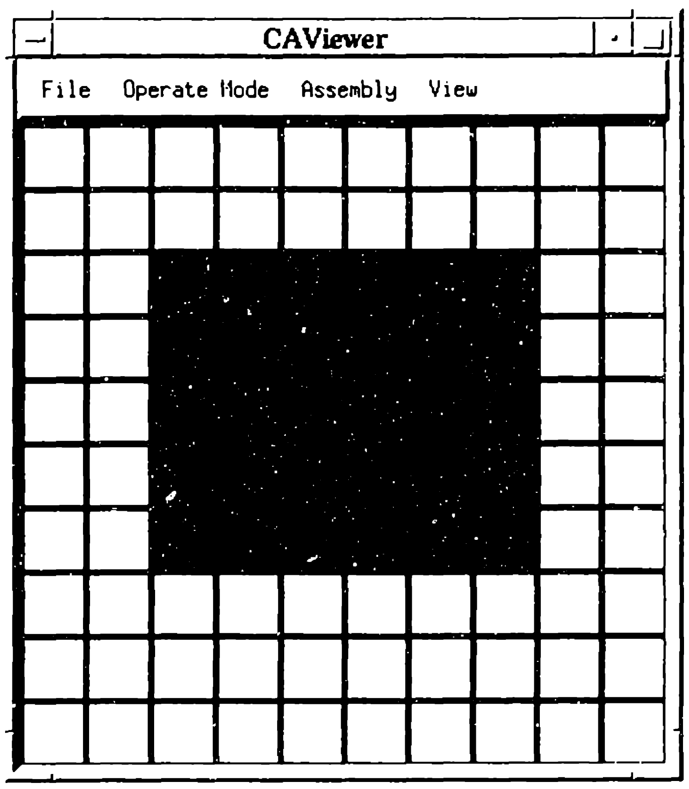


Figure 5.12 Five part assembly.

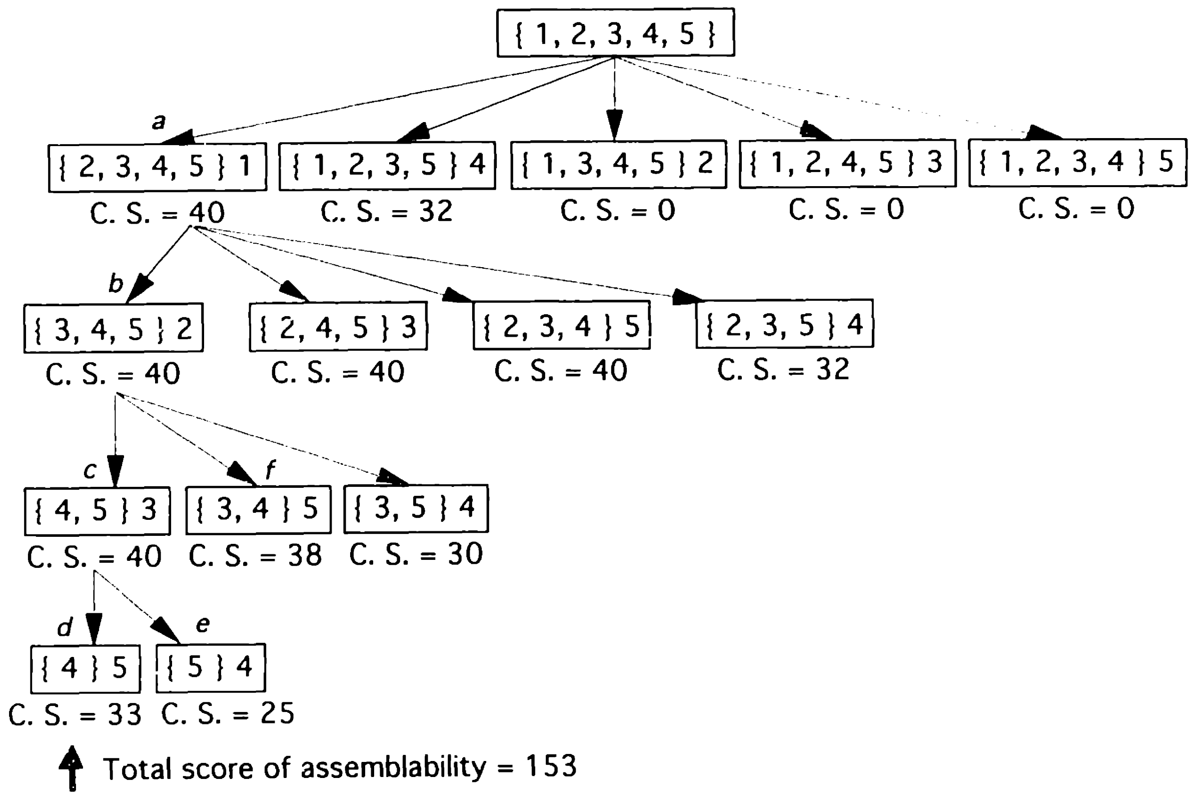


Figure 5.13 Branch and bound tree of the example assembly in figure 5.12.

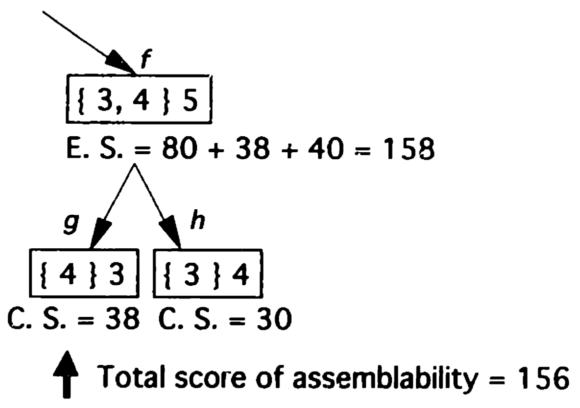

f
{ 3, 4 } 5
E. S. = 80 + 38 + 40 = 158
g
h
{ 4 } 3
{ 3 } 4
C. S. = 38
C. S. = 30
Total score of assemblability = 156

Figure 5.14 Branch and bound tree of the example assembly in figure 5.12.

5.4.3 Examples

The branch and bound technique described in the previous section was tested with the same example assemblies tested with the genetic algorithm. The figure 5.15 provides the comparison between performance measures of the branch and bound method and GA with population sizes of 50 and 100. The performance measures are defined with the number of evaluation function calls and the best solutions found by the methods for each example. For all these examples the branch and bound method performed better than the genetic algorithm as shown in figure 5.15. It should be also important to note that the branch and bound method always finds the true global optima based on the criteria.

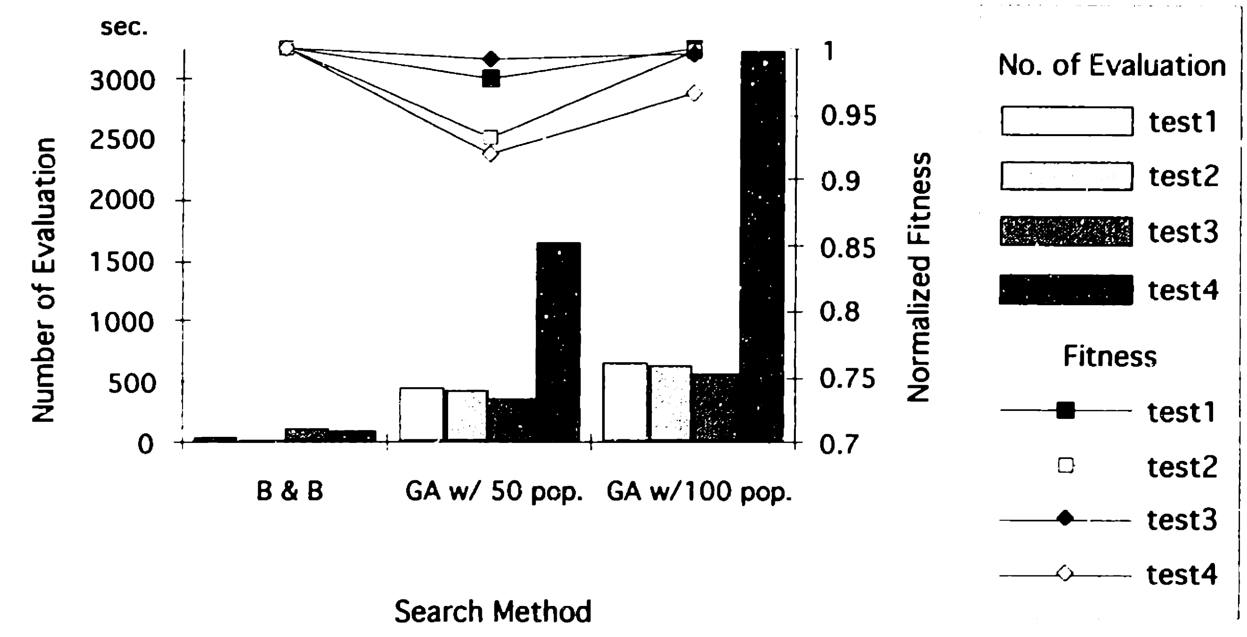


Figure 5.15 Search methods versus number of evaluation and fitness.

For all the previous examples the branch and bound performs better than the genetic algorithm because it can efficiently find the optimal assembly sequence in the early stage of search and, thereby, truncate most branches which have low estimation scores. Most optima were found after two or three updates of the bound. This was possible because the branch and bound method dynamically sorts task nodes, finds a node which is likely to contain the optimal solution, and branches into that node.

Figure 5.16 shows the performances of the branch and bound method and the genetic algorithm for the set of examples given in figures 5.17, 5.18, 5.19, and 5.20. These examples in the form of stacked parts are provided so that they represent several different classes of assemblies. The example in figure 5.17 represents seven parts horizontally placed on top of an large base part. For this type of assembly, there exist not only many alternatives for disassembling a part from the assembly, but also cases in which preceding the branch and bound method with a disassembly task that has the best evaluation score among alternatives can lead to a poor disassembly sequence at the end. On the contrary, the example in figure 5.20 has only a limited number of feasible disassembly sequences. The result in figure 5.16 shows that the branch and bound method performs worse than the genetic algorithm for the example shown in figure 5.17. As discussed above, this is because the branch and bound cannot prune branches effectively in the early stage of search.

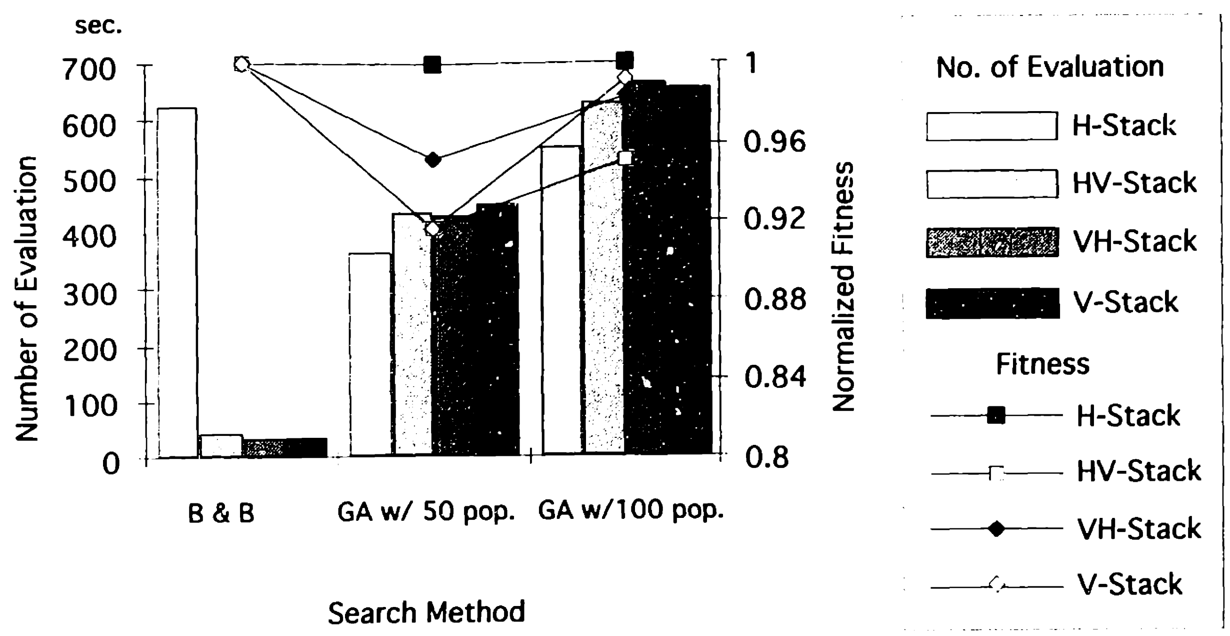


Figure 5.16 Search methods versus number of evaluation and fitness.

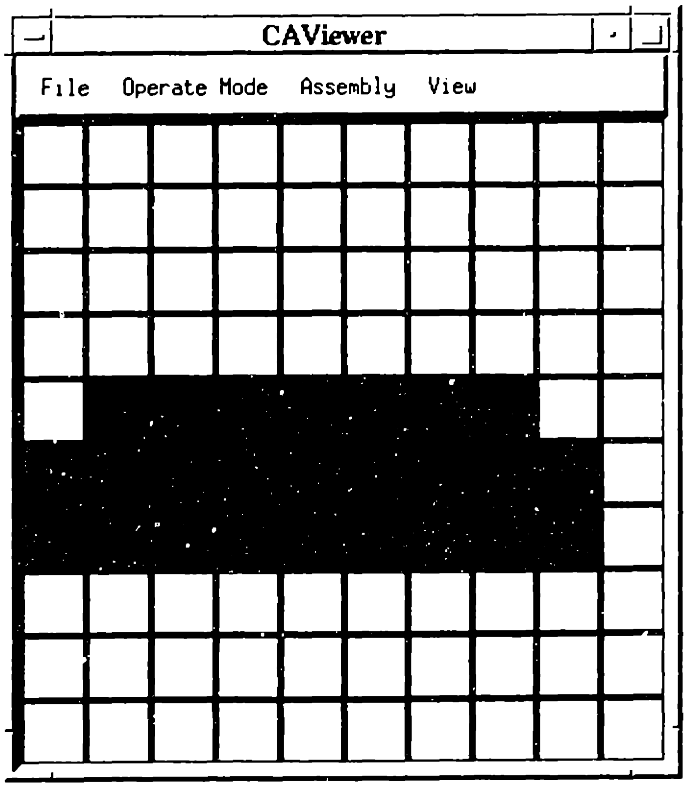


Figure 5.17 Example assembly: Horizontal stack.

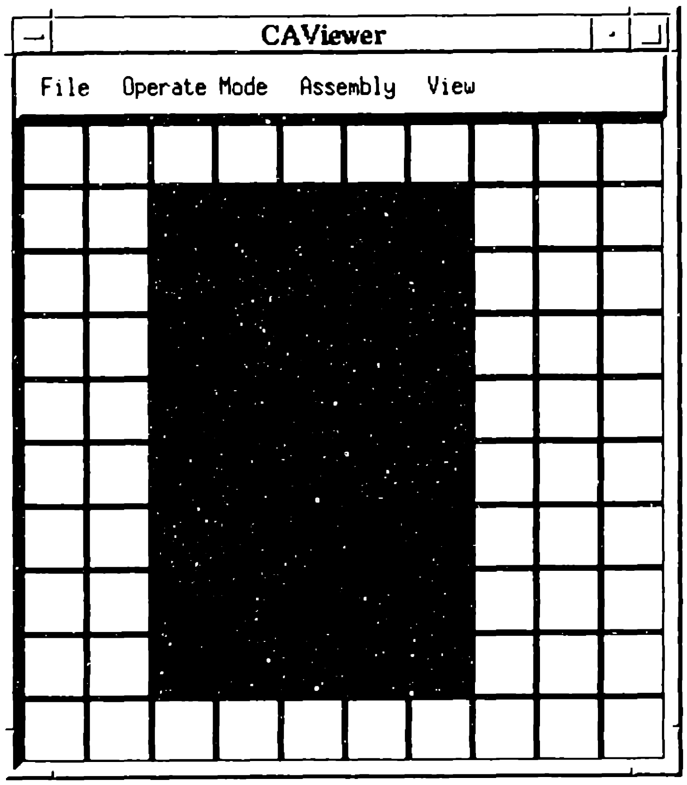


Figure 5.18 Example assembly: Horizontal-vertical stack.

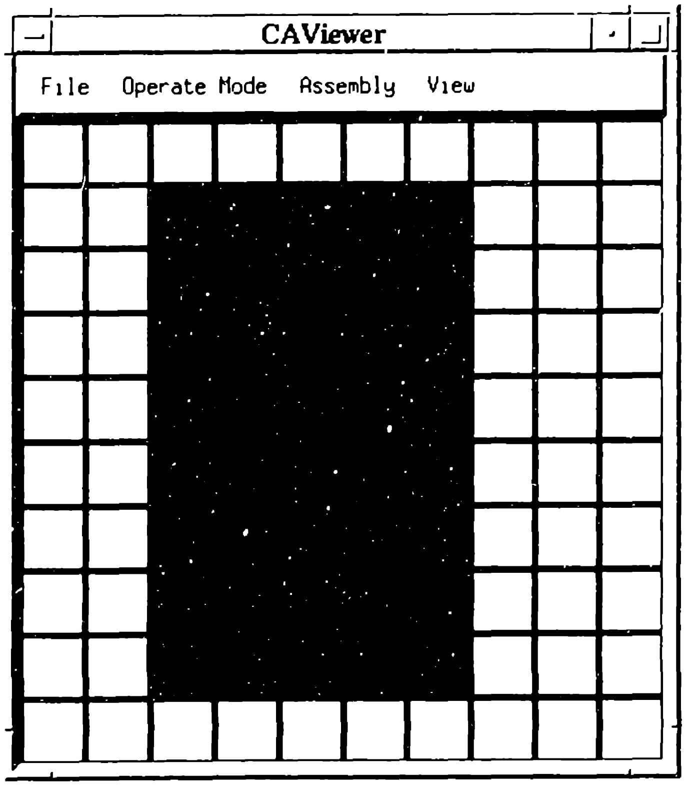


Figure 5.19 Example assembly: Vertical-horizontal stack.

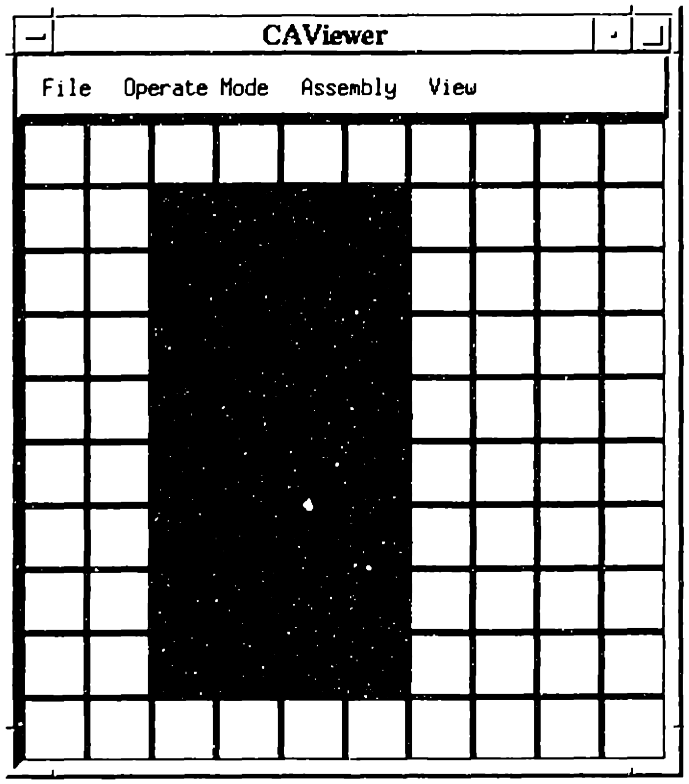


Figure 5.20 Example assembly: Vertical stack.

6 Conclusions

6.1 Overview

This chapter first provides all overview of the capabilities of the system described in this thesis and the optimization of the linear assembly sequence based on assemblability criteria. Then, conclusions regarding the generality of the cellular automata-based assembly planner were provided. Finally, potential areas of future work are discussed.

6.2 Capabilities of the Disassembly Planner

The robustness and generality of the cellular automata physical simulation technique allows an assembly planner to handle special case situations that are difficult with other assembly planning approaches. Wolter (1991, pps. 266 - 267) defines the special situations of "monotone", and "contact coherent" plans.

"Monotone" plans are those that do not allow temporary positions for parts: all parts are assembled into their final goal states. Since parts have temporary positions, finding non-monotone plans is more difficult because there are more possibilities for valid (dis)assembly part moves. In generating a disassembly sequence, for example, possible disassembly motions of a part may need to be examined many times, each time in a new context of other parts having been moved. The cellular automata-based approach does not have any inherent advantage over CAD-based planners in this regard: the approach would check these additional movements. The simplicity and generality of the rule sets, however, would allow a more straightforward implementation. Figure 6.1 (adapted from Wolter, 1991) shows an assembly that requires a non-monotone plan, since part 3 must maintain a temporary position in contact with part 1 as 1 is assembled into part 2. The planner was able to generate a disassembly sequence by not requiring that a part be removed completely from an assembly during the allowable part motion tests.

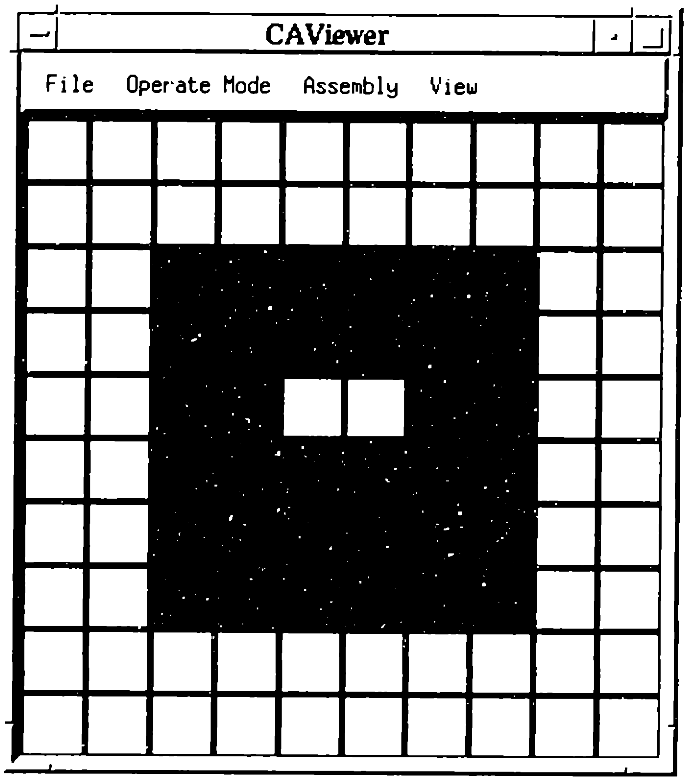


Figure 6.1 A monotone assembly.

Wolter also mentions "contact coherent" assemblies as those that require that parts moved into position are in contact with other parts. In other words, fixtures are not required to hold parts in noncontacted positions until other parts are added to the assembly. An example (figure 6.2 adapted from Wolter, 1991) shows that if part 6 is removed, the remaining two subassemblies will not be in contact. Our disassembly planner could handle the removal of part 6, if it treated one of the other parts as a base part, but the reverse assembly sequence would not be valid because both remaining subassemblies would need to be moved into position without contacting other parts. As the system is based on the detection of part contact, the subassemblies could not be moved into position before assembling part 6, and this *assembly* sequence could not be simulated.

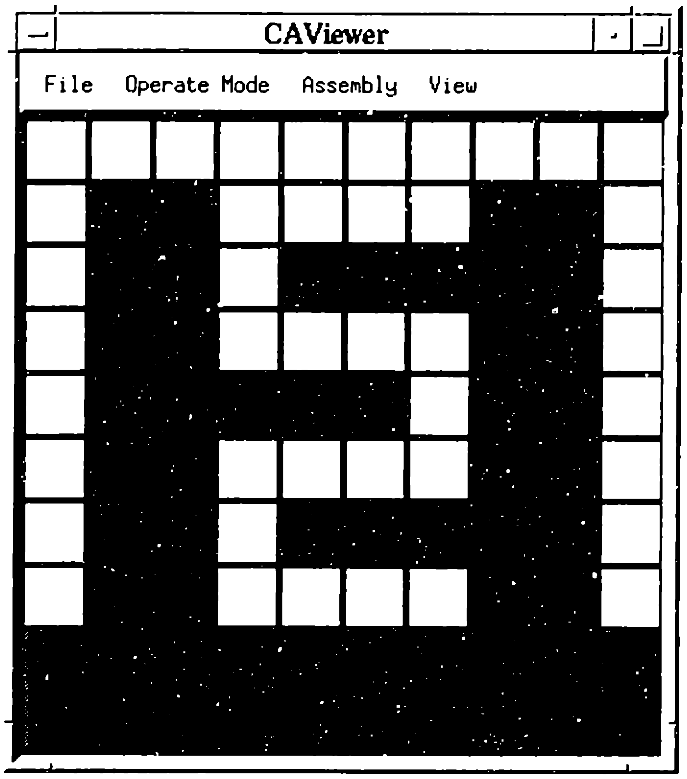


Figure 6.2 A non-contact-coherent assembly.

6.3 Optimization with Cellular Automata

The most important aspect of the cellular automata implementation is that it is extremely general. Within the context of the cellular array, any set of parts can be modeled, and with the disassembly algorithm described in chapter 4, a valid assembly sequence, if one exists, can be discovered. This generality is achieved by employing an "atomic" building block, the cell. The same set of rules applies to each atomic cell, so any geometry that can be built from the cells can be examined with a physical simulation. The price of this generality is a limitation on the detail with which objects can be represented. Only objects whose shapes can be reasonably modeled as arrangements of square cells can be treated. Additionally, only assemblies disassemblable with a series of rectilinear motions can be investigated. For two-dimensional depictions of actual assemblies, these approximation limitations are often reasonable. The cellular automata models will become more accurate, of course, by using a larger number of smaller cells. Computational difficulties, unfortunately, will also increase.

The most important consequence of the atomic building block generality is that it allows automated modification of part shapes and assembly topology. Since any cellular array can be analyzed with the part membership and part motion rules, there is no fear of creating an invalid assembly. As stated earlier, changing the material/void and edge type assignments may change not only the shapes of the parts, but also the number of parts and their connectivities. Since all possible assemblies can be analyzed, this allows the possibility of a very general iterative optimization scheme.

6.4 Future work

There are many avenues for future work and extensions to the system. First, keeping the same basic approach, it will be useful to implement the system on a parallel processing architecture. Cellular automata are inherently parallel and a parallel implementation will allow much faster execution. Currently, a simple assembly sequence planner based on a cellular automata representation is being implemented on a parallel processing architecture Cellular Automata Machine (CAM). CAM-8 is a parallel, uniform scalable architecture which offers superior performance in the fine-grained modeling of spatially-extended systems (Margolus, Toffoli et al. 1994). Also, the same basic approach can be applied to three-dimensional domains. In this case, a parallel processing approach will be even more important.

It will also be useful (and necessary) to devise some techniques that allow the conversion of the cellular representation to a more standard CAD representation. This could perhaps be done by employing intermediate quadtree and octree representations. Transformations between these and standard CAD models have already been investigated (Elber and Shpitalni 1988).

One of the main advantages of a discretized "atomic" domain is that it will allow the use of a combinatorial search technique to perform an optimal design study over the space of possible cellular arrays. Genetic algorithms (Goldberg 1989) or simulated annealing (Kirkpatrick, Gelatt et al. 1983), for example, could be used to search for the optimal assignments of material, void, and edge types to satisfy functional constraints as well as to optimize with respect to assembly considerations. A major challenge will be to model "functionality" and "assemblability." We hope to exploit the discretized binary nature of a cellular array when modeling functionality. Input motions at a subset of cells, for example, could be required to produce output motions at another subset of cells. Assemblability is related to the allowable motion of parts in their assembled goal states as described in chapter 5. We also seek to build on previous approaches to this problem that do not employ cellular representations. (*e.g.* (Miyakawa, Ohashi et al. 1990) and (Lee 1991)).

References

Baldwin, D. F., T. E. Abell, et al. (1991). "An Integrated Computer Aid for Generating and Evaluating Assembly Sequences for Mechanical Products." IEEE Transactions on Robotics and Automation **7**(1): 78-94.

Boothroyd, G. and L. Alting (1992). "Design for Assembly and Disassembly." CIRP Annals **41**: 625-636.

Boothroyd, G. and C. Ho (1976). "Natural Resting Aspects of Parts for Automatic Handling." American Society of Mechanical Engineers (Paper 76-WA/Prod-40).

Boothroyd, G., C. Poli, et al. (1982). Automatic Assembly. New York and Basel, Marcel Dekker Inc.

Bourjault, A. (1984). Contribution to a methodological approach of automated assembly: automatic generaton of assembly sequences, University de Franche-Comte.

Bourjault, A. (1987). "Methodology of Assembly Automation: A New Approach." Robotics and Factories of the Future San Diego, California, 39-45.

Chakrabarty, S. and J. Wolter (1994). "A Hierarchical Approach to Assembly Planning." 1994 IEEE International Conference on Robotics and Automation Los Alamitos, California, 258-263.

Christofides, N., A. Mingozzi, et al. (1979). Combinatorial Optimization, John Wiley & Sons.

De Fazio, T. L., T. E. Abell, et al. (1990). "Computer-Aided Assembly Sequence Editing and Choice: Editing Criteria, Bases, Rules, and Technique." IEEE International Conference on Systems Engineering Pittsburgh, Pennsylvania, 416-422.

De Fazio, T. L. and D. E. Whitney (1987). "Simplified Generation of All Mechanical Assembly Sequences." IEEE Journal of Robotics and Automation **RA-3**(No. 6): 640-658.

De Fazio, T. L. and D. E. Whitney (1989). "Aids for the Design or Choice of Assembly Sequences." 1989 IEEE International Conference on Systems, Man, and Cybernetics **1**: 61-70.

Delchambre, A. (1992). Computer-aided Assembly Planning. London, Chapman & Hall.

Elber, G. and M. Shpitalni (1988). "Octree creation via C.S.G. definition." The Visual Computer **4**: 53-64.

Foley, J. (1993). Computer Graphics: Principles and Practice, Addison Wesley.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization & Machine Learning.

Henrioud, J. and A. Bourjault (1991). LEGA: a compter-aided generator of assembly plans. Computer-Aided Mechanical Assembly Planning. L. S. Homen de Mello and S. Lee. Boston, Kluwer Academic Publishers: 191-215.

Hoffman, R. (1990). "Automated Assembly Planning for B-Rep Products." IEEE International Conference on Systems Engineering Pittsburg, Pennsylvania, 391-394.

Holland, J. (1975). Adaptation in Natural and Artificial Systems. Ann Arbor, Michigan, The University of Michigan Press.

Homem de Mello, L. S. and A. C. Sanderson (1990). "AND/OR Graph Representation of Assembly Plans." IEEE Transactions in Robotics and Automation **6**(2): 188-199.

Homem de Mello, L. S. and A. C. Sanderson (1990). "Evaluation and Selection of Assembly Plans." 1990 IEEE International Conference on Robotics and Automation Cincinnati, Ohio, 1588-1593.

Homem de Mello, L. S. and A. C. Sanderson (1991). A Basic Algorithm for the Generation of Mechanical Assembly Sequences. Computer-Aided Mechanical Assembly Planning. L. S. Hemem de Mello and S. Lee. Boston, Kluwer Academic Publishers: 163-190.

Jakiela, M. J., P. Y. Papalambros, et al. (1985). "Programming Optimal Suggestions in the Design Concept Phase: Applications to the Boothroyd Assembly Charts." ASME Journal of Mechanisms, Transmissions, and Automation in Design **107**(2): 285-291.

Kirkpatrick, S., C. Gelatt and M. Vecchi (1983). "Optimization by Simulated Annealing." Science **220**: 671-680.

Klein, C. J. (1987). Generation and Evaluation of Assembly Sequence Alternatives. Mechanical Engineering. Cambridge, MIT.

Lee, S. (1991). Backward Assembly Planning with DFA Analysis. Computer-Aided Mechanical Assembly Planning. L. Homem de Mello and S. Lee. Boston, Kluwer Academic Publisher: 341-381.

Lee, S. (1994). "Subassembly Identification and Evaluation for Assembly Planning." IEEE Transaction on Sytems, Man, and Cybernetics **24**(3): 493-503.

Lee, S., G. J. Kim, et al. (1993). "Combining Assembly Planning with Redesign: An Approach for More Effective DFA." IEEE International Conference on Robotics and Automation Atlanta, Georgia, **3**: 319-325.

Lee, S. and Y. G. Shin (1990). "Assembly Planning Based On Geometric Reasoning." Computer & Graphics **14**(2): 237-250.

Margolus, N., T. Toffoli, et al. (1994). An Early Sampler of CAM-8 Applications. Cambridge, MA, MIT.

Miller, J. M. and R. L. Hoffman (1989). "Automatic Assembly Planning with Fasteners." IEEE International Conference on Robotics and Automation, 69-74.

Miyakawa, S., T. Ohashi, et al. (1988). "The Hitachi Assemblability Evaluation Method (AEM) and Its Application." Journees De Microtechnique, 99-114.

Miyakawa, S., T. Ohashi, et al. (1990). "The Hitachi New Assemblability Evaluation Method (AEM)." Transactions of NAMRI/SME, 352-359.

Mortenson, M. E. (1985). Geometric Modeling. New York, John Wiley and Sons.

Nevins, J. L. and D. E. Whitney (1978). "Computer-Controlled Assembly." Scientific American **238**(2): 62-74.

Preston, K. J. and M. J. B. Duff (1984). Modern Cellular Automata: Theory and Applications. New York, Plenum Press.

Tsao, J. and J. Wolter (1993). "Assembly Planning with Intermediate States." IEEE International Conference on Robotics and Automation Atlanta, Georgia, **1**: 71-76.

Wilson, R. H. (1990). "Efficiently Partioning an Assembly." IASTED International Symposium on Robotics and Manufacturing.

Wilson, R. H. (1993). "Minimizing User Queries in Interactive Assembly Planning." IEEE International Conference on Robotics and Automation Atlanta, Georgia, 322-327.

Wilson, R. H. and J.-F. Rit (1991). Maintaining Geometric Dependencies in Assembly Planning. Computer-Aided Mechanical Assembly Planning. L. S. Homem de Mello and S. Lee. Boston, Kluwer Academic Publishers: 218-241.

Wolter, J. D. (1989). "On the automatic generation of assembly plans." 1989 IEEE International Conference on Robotics and Automation Scottsdale, Arizona, 62-68.