INQUIRING the MAIN ASSUMPTION of the ASSEMBLY LINE BALANCING PROBLEM: SOLUTION PROCEDURES USING AND/OR GRAPH

A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ali Koç
July 2005

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. İhsan Sabuncuoğlu (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Erdal Erel

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Selim Aktürk

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray

Director of Institute of Engineering and Science

ABSTRACT

Inquiring The Main Assumption Of The Assembly Line Balancing Problem: Solution Procedures Using And/Or Graph

Koç, Ali
M.S. in Industrial Engineering
Supervisor: Prof. İhsan Sabuncuoğlu
July 2005

In this thesis, we consider the assembly/disassembly line balancing (ADLB) problem. The studies in the literature consider assembly and disassembly problems separately and use task precedence diagram (TPD) and AND/OR Graph (AOG) in assembly and disassembly line balancing problems, respectively. In contrast to these studies, we use AOG for both assembly and disassembly line balancing problems, considering these two problems as complementary of each other. Hence, we call the complementary problem as ADLB-AOG. We show theoretically that AOG is a more general version of the TPD. We also develop integer programming (IP) and dynamic programming (DP) formulations to solve the ADLB-AOG problem. Our analysis indicates that the DP formulation performs much better than the IP formulation in terms of the problem sizes that can be optimally solved. We also develop a DP-based heuristic to solve large-size instances of the ADLB-AOG problem. An experimentation of the procedures on some sample problems and the implementation of the heuristic on a sample problem are also given.

Keywords: Assembly, Disassembly, Line Balancing, AND/OR Graph, Task Precedence Diagram, Integer Programming, Dynamic Programming, Heuristic.

ÖZET

Montaj Hattı Dengeleme Problemlerının Temel Varsayımını Sorgulama: And/Or Grafiği Kullanılarak Geliştirilen Çözüm Prosedürleri

Ali Koç
Endüstri Mühendisliği Yüksek Lisans
Tez Yöneticisi: Prof. İhsan Sabuncuoğlu
Temmuz 2005

Bu tezde, montaj/demontaj hattı dengeleme problemini incelemekteyiz. Literatürdeki çalışmalar, montaj ve demontaj problemlerini ayrı ayrı olarak ele alıp montaj hattı dengeleme problemi için iş önceliği diyagramını, demontaj hattı dengeleme problemi için ise AND/OR grafiğini kullanmaktadırlar. Biz ise her iki problemi birbirinin tersi olarak ele aldığımız için, her ikisinde de AND/OR grafiğini kullandık. Binaen aleyh, problemi montaj/ demontaj hattı dengeleme problemi olarak isimlendirdik. Ayrıca, teorik olarak ta gösterdik ki AND/OR grafiği, iş önceliği diyagramından daha genel olduğundan bu grafik kullanılarak çözülen problem diğerinden daha iyi, en azından aynı, neticeler vermektedir. Öne sürülen bu problemi hem tamsayı programlama hem de dinamik programlama yöntemleri ile çözdük. Bu iki yöntemle bazı örnek problemleri çözdüğümüzde, dinamik programlama yöntemi inkar edilemez bir farkla tamsayı programlama yöntemini geride bıraktı. Daha büyük problemlerin çözebilmesi için daha hızlı çalışan bir sezgisel yöntem de geliştirdik. Tüm problem verileri, örnek çözümler ve uygulamaları, gerek metnin içinde gerek ilave bölümlerde verilmiştir.

Anahtar Kelimeler: Montaj, Demontaj, Hat Dengeleme, AND/OR Grafiği, İş Önceliği Diyagramı, Tamsayı Programlama, Dinamik Programlama, Sezgisel Yöntem.

TABLE OF CONTENTS

CHAPTERS

INTRODUCTION **1**

1.1 MOTIVATION 1

1.2 STATEMENT OF THE PROBLEM AND RELATED CONTRIBUTION 3

LITERATURE SURVEY **6**

2.1 BACKGROUND 6

2.2 LITERATURE SURVEY 13

PROPOSED THEORY: QUESTIONING THE FUNDAMENTAL ASSUMPTIONS AND SOLVING THE ACTUAL PROBLEM **16**

3.1 AND/OR GRAPH AND ASSEMBLY/DISASSEMBLY TREE 17

3.1.1 AND/OR Graph (AOG) *17*

3.1.2 Assembly/Disassembly Tree (AT/DT) *19*

3.2 THEOREM OF SUB-OPTIMALITY 24

3.3 THE DERIVATION OF A TPD FROM THE AOG 29

3.4 AN EXAMPLE TO COMPARE TPD AND AOG. 36

THE SOLUTON TO THE ADLB-AOG PROBLEM **40**

4.1 THE PROPOSED DYNAMIC PROGRAMMING (DP) FORMULATION 42

4.1.1 Definitions and Terminology *42*

4.1.1.1 Partial AOG's 42

4.1.1.2 Assembly task sequences 43

4.1.1.3 Relation between partial AOG's and assembly task sequences 45

4.1.2 The Proposed DP Approach *45*

4.1.3 Example *47*

4.2 THE PROPOSED INTEGER PROGRAMMING FORMULATION 53

4.2.1 The Formulation *53*

4.2.2 Example *56*

4.3 SOLVABLE SIZES OF ADLB-AOG PROBLEM BY DP AND IP METHODS 57

4.3.1 The DP formulation *61*

4.3.2 The IP Formulation *70*

4.4 A DP BASED HEURISTIC 73

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS 78

REFERENCES 81

AND/OR GRAPH AND RELATED CONCEPTS IN ASSEMBLY / DISASSEMBLY PROCESS PLANNING 90

A1.1 ASSEMBLY 90

A1.2 ASSEMBLY TASK 92

A1.3 ASSEMBLY SEQUENCE 96

A1.4 DISCUSSION ON AOG 97

FIGURES OF THE EXAMPLE 2 IN SECTION 3.3 106

FIGURES OF THE EXAMPLE 3 IN SECTION 3.3 122

STORING AOG IN A MATRIX 136

SOME EXAMPLES TO PARTIAL AOG (AOG(S)) 138

JAVA CODE FOR THE DP METHOD TO THE ADLB-AOG PROBLEM 142

JAVA CODE FOR THE FORMULATION OF ADLB-AOG PROBLEM AS PURE 0-1 IP PROGRAMMING 154

JAVA CODE TO GENERATE AOG 158

JAVA CODE FOR THE HEURISTIC 165

LİST OF FİGURES

Figure 1 Interaction between government, users, producers and distributors as the driving force of the environmental management (taken from Gungor and Gupta 1999)... 8
Figure 2 Environmental management practices and their environmental contributions....................... 9
Figure 3 Different post-life options for the relinquished products 11
Figure 4 Remanufacturing and demanufacturing as the means of reverse manufacturing 12
Figure 5 A sample product (de Mello and Sanderson (1990)).. 17
Figure 6 AND/OR Graph of the Product in Figure 5 18
Figure 7 All of the 8 DT's obtained from the AOG in Figure 2 22
Figure 8 Eight transformed disassembly trees (TDT) associated with the eight DT's in Figure 4..... 24
Figure 9 Theorem of sub-optimality.. 28
Figure 10 The subassemblies of AOG in Figure 6, their GOC and corresponding tasks 32
Figure 11 The S_{AT^m} obtained from S_{AT} in Figure 8 33
Figure 12 S_{TPD} obtained by combining equitask AT^m's in Figure 11 34
Figure 13 Transformed AND/OR Graph of the AOG in Figure A.14 of Appendix 3 41
Figure 15 The partial AOG's obtained while solving the example problem 51
Figure 16 Dynamic programming solution to the ADLB-AOG problem for the AOG in Figure 14 . 52
Figure 17 Sample AOG's to illustrate the experimentation... 59
Figure 18 Solvable problem sizes when the number of tasks for each artificial node is 1 64
Figure 19 Solvable problem size when the number of tasks for each artificial node is 2 64
Figure 20 Solvable problem size when the number of tasks for each artificial node is 3 65
Figure 21 Solvable problem size when the number of tasks for each artificial node is 5 65
Figure 23 The solution duration vs. difficulty of the problem.. 71
Figure 24 Solution of a sample problem by the DP method... 73
Figure 25 Heuristic solutions for the example problem in Figure 24 75

Figure A.1 Graph of connections (GOC) for the product in Figure 5 of Chapter 2 91
Figure A.2 A feasible assembly sequence (τ_1, τ_2, τ_3) 94
Figure A.3 An infeasible assembly sequence (τ_1, τ_4, τ_5) 95
Figure A.4 A loosely connected product, its graph of connections and AOG 101
Figure A.5 A strongly-connected product, its graph of connections and AOG 102
Figure A.6 Proposed AND/OR graph 105
Figure A.7 An example product and its GOC (Lambert 1999) 106
Figure A.8 AOG of the product in Figure A.7 107
Figure A.9 S_{AT} established from the AOG in Figure A.8 112
Figure A.10 The subassemblies of AOG in Figure A.8, their GOC and corresponding tasks 115
Figure A.11 The S_{AT^m} obtained from S_{AT} in Figure A.9 120
Figure A.12 S_{TPD} obtained by combining equitask AT^m's in Figure A.11 121
Figure A.13 An example product and its GOC (Lambert 1999) 122
Figure A.14 AOG of the product in Figure A.13 123
Figure A.15 S_{AT} established from the AOG in Figure A.14 127
Figure A.16 The subassemblies of AOG in Figure A.14, their GOC and corresponding tasks 130
Figure A.17 The S_{AT^m} obtained from S_{AT} in Figure A.15 134
Figure A.18 S_{TPD} obtained by combining equitask AT^m's in Figure A.17 135
Figure A.19 AOG ({A_{13}}) 138
Figure A.20 AOG ({A_6}) 139
Figure A.21 AOG ({A_7}) 139
Figure A.22 AOG ({A_8}) 140
Figure A.23 AOG ({A_8, A_{13}}) 140
Figure A.24 AOG ({A_8, A_{13}, A_9}) = AOG ({A_8, A_9}) 141

LIST OF TABLES

Table 1 Durations of the SD and SI tasks in the example.. 37

Table 2 The solutions to the three example problems 38

Table 3 Solvable size of the AOG's without parallelism by the DP approach. 62

Table 4 The difficulties of 124 sample problem and solution durations.. 69

Table 5 The results of the heuristic solution to the ADLB-AOG problem compared with the exact results of both ADLB-TPD and ADLB-AOG problem. 76

Chapter 1

INTRODUCTION

1.1 Motivation

Due to the threatening environmental issues, in recent years, demanufacturing and remanufacturing of products gained an increasing attention from both practitioners and researchers (Grenchus et al. 1997, Spicer and Johnson 2004, Thierry et al. 1995, Ayres et al. 1997, Bras and McIntosh 1999, Guide Jr VDR. 2000, Parkinson and Thompson 2003). Both practices rest mostly on the disassembly of the products. Consequently, disassembly process, ranging from factory level, through work system level, to the operation level, gained a considerable attention from the researchers (Brennan et al. 1994, Kochan 1995, Zussman 1995, Wiendahl et al. 1999, Gungor and Gupta 1999, Tang et al. 2000, Lee et al. 2001, Lambert 2003).

Almost all of the researchers sought ways of handling the disassembly issue independent of the assembly process. Even though disassembly and assembly differ to a great extent due to the differences in production planning and inventory control issues, there is not so much difference as far as the shop floor activities are concerned: In the disassembly process, we take apart what are put together while assembling the product. Hence, some of the researchers view the disassembly process as the reverse of the assembly (Homem de Mello and

Sanderson 1990, 1991a and 1991b). The authors define all possible assembly/disassembly tasks for a product and establish the precedence relations among the tasks. Each possible task definition and corresponding precedence relation is represented by an *assembly/disassembly tree (AT/DT)*. They develop a graph called *AND/OR Graph (AOG)* that includes all AT/DT's of a product.

Inspired from the AOG, in this study, we handled the assembly and disassembly processes as complementary of each other. As a result of this, we pinpoint some of the facts that escape from the eyes of many, in assembly studies. For many years, researchers have sought the ways to solve the assembly line balancing problem (ALB). But they did not question the main input to the problem, the task precedence diagram. *Task precedence diagram (TPD)* shows the precedence relations between the assembly tasks. They consider only one the many feasible definitions of the assembly tasks. Two differently defined sets of tasks for the same assembly/disassembly process yield two different TPD's. Consequently, the number of stations in the line to assemble/disassemble the same product for different set of tasks may change. Based on this, when balancing the assembly/disassembly line optimally, we use AOG instead of the TPD to consider all possible task definitions. The objective is to minimize the cycle time while keeping the number of stations constant. We showed that using AOG is better than using TPD.

1.2 Statement of the Problem and Related Contribution

Disassembly activities take place in both remanufacturing and demanufacturing practices. Disassembly is a systematic method for separating a product into its constituent parts, components, subassemblies or other groupings (Gupta and Taleb 1994). Assembly may be defined just as the reverse of the disassembly. Even though due to some characteristics of disassembly process, assembly and disassembly should be treated independently, we suggest that, at least for the operational purposes, they should be considered as the complementary of each other. Hence, all the inferences, discussions and results obtained in this study will apply to both assembly and disassembly.

Assembly/disassembly line is made up of an ordered sequence of stations connected by a material handling system. The line may be *paced* or *unpaced*. In paced assembly lines, the cycle time for all stations is common, whereas in unpaced lines all stations are allowed to operate at their own phase. In assembly line, parts of the product enter the line and move to the downstream stations until (some of) the parts are assembled. Whereas in disassembly line, the whole product enters the line and moves to the downstream stations until the parts of concern are obtained. Both assembly and disassembly may be partial or complete. In partial assembly, parts are assembled not until obtaining the whole product but until some subassemblies are formed. Similarly, in partial disassembly a product is disassembled to obtain subassemblies of interest. Partial assembly/disassembly is motivated by the profit maximization objective including the time and cost components of the tasks and profit component of the subassemblies, while

complete assembly/disassembly is driven by only the time components of the tasks in the process.

Performing assembly and disassembly require certain apparatus and certain operators. Technological and physical conditions define the precedence relations among the tasks. In this study, we will use AND/OR Graphs (AOG) developed by de Mello and Sanderson (1990) instead of the classical task precedence diagrams (Prenting and Battaglin 1964) used in assembly line balancing (ALB) problems. We also show the superiority of AOG over the classical task precedence diagrams (TPD) in line balancing problems.

The problem in this study may be posed as follows: Assign the assembly/disassembly tasks of a product obtained from the AOG to an ordered sequence of stations in order to completely assemble/disassemble the product, such that the precedence relations in AOG are satisfied and number of stations is minimized for a cycle time common to all stations. We call the problem as assembly/disassembly line balancing problem for AOG (ADLB-AOG). The scarce literature related to the problem is given in chapter two.

One of the major contributions of this study is to show that the common practice over fifty years of finding an optimum solution for the ALB problem using a TPD constitutes an upper bound on the actual problem that should be defined on AOG. In the third chapter, we prove that an assembly/disassembly line balancing problem for the classical task precedence diagram yields an inferior solution to the assembly/disassembly line balancing problem for the AOG. By an example we show the superiority of the AOG over the TPD. We also show how to obtain the TPD from the AOG by three examples in the same chapter. Another

contribution is the development of dynamic programming (DP) and integer programming (IP) approach to the proposed problem. In the fourth chapter we develop these two methods. The IP formulation of the problem was previously given by Altekin (2005). But, the IP formulation proposed there considers not only the AOG but also other precedence diagrams (such as part precedence diagrams) and aims partial disassembly. In contrast, the IP formulation proposed here is more effective in the sense that it is tailored to the specific problem that considers only the complete assembly case and uses only AOG as the precedence relations. Hence, for this problem type, development of suitable IP formulation is another contribution. In the same chapter, we compare the IP and DP formulations in terms of the size of the problem they can handle by generating sample AOG's defined by three parameters. We also develop a DP based heuristic in this chapter. Finally, we discuss the conclusions and inferences in chapter five.

Chapter 2

LITERATURE SURVEY

Before the literature review, we will give some background information on why the disassembly process is a vital element of the so-called 'environmentally conscious' industry. Even though this section can be omitted without loosing the integrity of the study, we recommended reading in order to see how the disassembly studies including the disassembly line balancing are developed. Then, we present the scarce literature of disassembly line balancing problem and refer the interested reader to the milestones of the huge world of assembly line balancing problem.

2.1 Background

By the advent of developments in science and technology, the Earth became an industrial world with all of its resources, both energy and minerals. A plethora of desirable products in the market place, may be more than the demand, and heedless consumption have decreased the natural resources, including air quality, water, mines, diversity of flora and fauna, and most importantly the landscape. Furthermore a series of industrial accidents in 1970s and 1980s, such as the accident at Bhopal in India, the Exon Valdez oil spill, etc., have increased the environmental awareness.

People, oblivious to environmental degradation of industrialization before, began to realize the adversarial effects of the industrial world on the environment. Therefore, there appeared the buzzword *green consumerism*, which shows people's reaction as supporting and encouraging the green products and boycotting the others. Government also passed some legislation against the mass production, heedless consumption, indiscriminate disposal habits and polluting hazardous products (Bodily and Gabel, 1982; Bonifant et al. 1995; Klassen and Angell, 1998). What remains to corporations is to obey and comply with the rulings by changing the infrastructural and structural components so that the impacts of the production, usage and disposal of the products on the ecology are appeased (Bodily and Gabel, 1982; Barney, 1991; Corbett and Wassenhove, 1993; Elkington, 1994; Gupta M.C., 1994; Epstein, 1996; Florida, 1996; Azzone et al. 1997; Maxwell et al. 1997; Hart, 1997; Zhang et al. 1997; Hartman and Stafford, 1998; Inmann, 2002).

The endeavor of citizens, government and corporations to alleviate the environmental degradation is called *environmental management (EM)* (Figure 1). It is defined as the total of the efforts lessening and alleviating the adverse environmental affects of industrialization. Industrialization includes the production and service facilities. The term describing management of both of these practices is called *production and operations management (POM)* (Gupta MC 1994). Hence, to attain the goal of environmental management, also dubbed as '*sustainable development*', one should either lessen the hazardous effects of POM activities or decrease the activities itself. To decrease the harmful effects of POM activities is the business of pollution prevention efforts, while to decrease the

POM activities, i.e., to increase the production efficiency by serving the same demand with less production, is the focus of *closed loop manufacturing (CLM)* (Brown and Karagozoglu 1998) (Figure 2). *Pollution prevention* rescues renewable resources of the nature such as air and water quality, and landscape by abating the hazardous waste releases. *Closed loop manufacturing (CLM)* includes product design and process design. Product design is the process of designing for environment (DFE), while process design is to establish and coordinate reverse manufacturing (RM) and reverse logistics (RL) activities to take back the post-use products (cores) and reemploy them either by remanufacturing or demanufacturing.

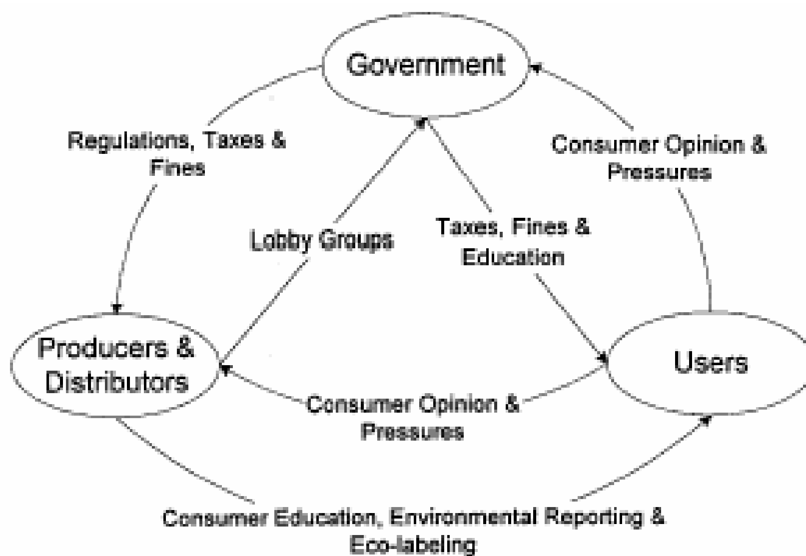


Figure 1 Interaction between government, users, producers and distributors as the driving force of the environmental management (taken from Gungor and Gupta 1999)

Pollution Prevention

Production and usage phases

Liquid and gaseous waste reduction

Solid waste reduction

Core Recovery (Closed Loop Mfg.)

Post-use phase

Decrease in Resource Usage (Energy and Minerals)

New trends in the industry (magical concepts)

Environmental Contributions

Figure 2 Environmental management practices and their environmental contributions

Reverse manufacturing (RM) closes the loop in production processes by reemploying and reutilizing the products, parts and materials that were processed in the past, so that, in the extreme, once spent, natural resources and energy are used forever. Reverse manufacturing is an essential element of closed-loop manufacturing system. It is estimated to be 73,000 firms are engaged in remanufacturing in US, employing 350,000 people (Lund, 1998). Remanufacturing activities amounts to total revenue of $14 billion per year (US Environmental Protection Agency, EPA 2001). As a point of reference, consider the US steel industry has annual sales of $56 billion employing 241,000 people (Lund, 1998).

The automotive industry leads the practice of the reverse manufacturing activities. One example is the Mercedes Benz that decided to implement a recycling program including two elements: design and recycling. What is more, Mercedes Benz started to take back the used cars in 1991 and has been recycling

the material content since then (Gungor and Gupta, 1999). Another example is BMW. That gives credit to the customers for turning the used car back. Also, BMW uses color codes for different plastic materials in order to simplify the recycling, and claims producing 90% recyclable cars. Some other firms making similar efforts are General Motors, Volkswagen, Nissan Motor Company and Volvo Car Corporation (Gungor and Gupta, 1999). Deere and Company entered in to an agreement with Springfield Remanufacturing Company that they will recondition diesel engines and components for Deere and Company. Sales of reconditioned engines in 1996 exceeded $2.5 billion dollars (Guide et al. 2000). Computer and peripheral makers, such as IBM, Hewlett Packard, and Xerox, are applying disassembly. Xerox applies remanufacturing for its photocopiers and toner cartridges in US and abroad and estimates a cost saving of $20 million per year (Guide et al. 2000). In 1998, out of 33 power-tool manufacturers 13 of them agreed to take back their products from the customers in Germany (Klausner and Hendrickson, 2000).

The dichotomy that classifies the reverse manufacturing practices into two parts, demanufacturing and remanufacturing, is mainly decided by the different types of treatments to the cores, which we call *end-of-life (EOL) options*. In the early times before the environmentalism occurred, people were used to get rid of the products without any treatment. Today's attitude is to reemploy the product as much as possible in order to both reduce municipal solid wastes (MSW) and to lessen the usage of energy and resource. But, still, some relinquished products are disposed. Consequently, when a product completes its lifetime and is relinquished by the consumers, there are mainly two types of treatments: disposal and recovery

(Figure 3). Disposal option is to get rid of the product without any further treatment, which is done mainly in the form of landfilling. Recovery options, on the other hand, reclaim some benefit from the used product. Although the aim in recovery processes is to claim a win-win situation that serves both to the environment and to the budget, some of the recovery options, such as composting, incineration, etc, do not aim environmental friendliness.

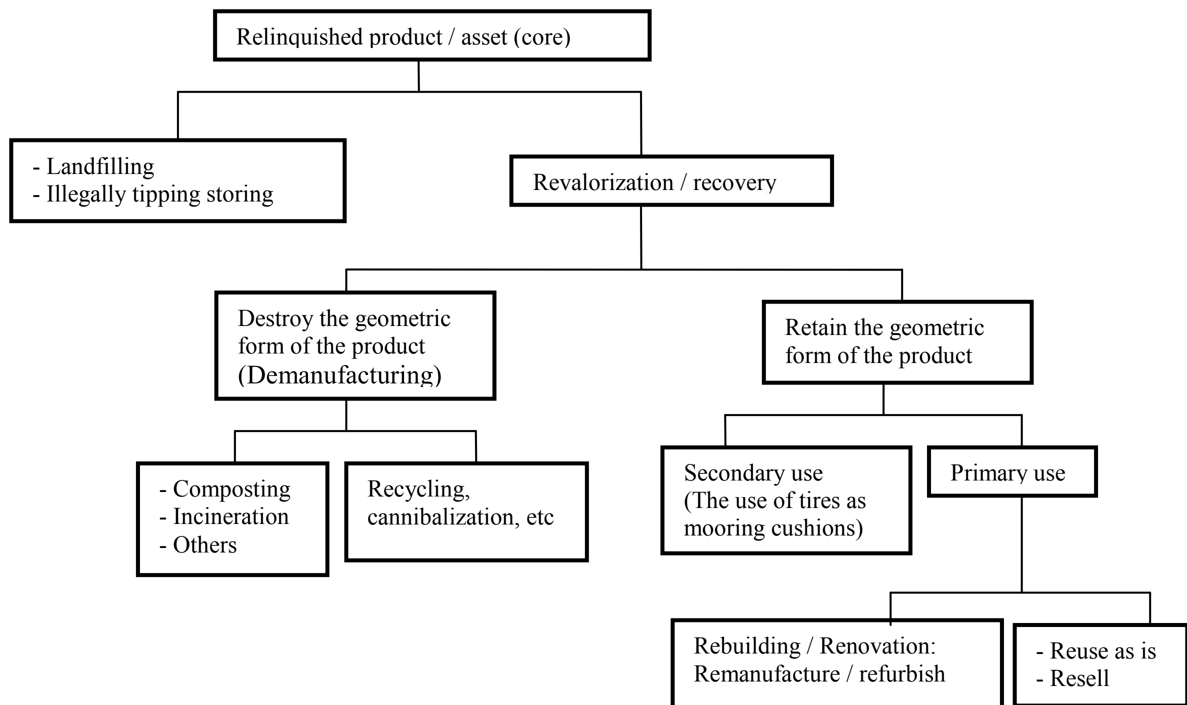


Figure 3 Different post-life options for the relinquished products

Since some of the recovery options require no or little treatments, there are two of them, among all, requiring detailed study and research: remanufacturing and demanufacturing. Remanufacturing is renovating / rebuilding all the

components of the product, while demanufacturing is decomposing the product into components so that most profitable end-of-life (EOL) options for each component can be chosen. Remanufacturing focuses on product recovery, whereas demanufacturing, in the form of recycling and cannibalization, focuses on part and material recovery. Hence, remanufacturing closes both material and energy use cycle, while demanufacturing closes only material use cycle (Figure 4). The following quotation from White et al. (2003) makes the distinction between demanufacturing and remanufacturing clear in the computer industry. “The distinction between demanufacturing and remanufacturing, which has not been emphasized in the literature, is a subtle one. As opposed to remanufacturing’s focus on rebuilding products and reclaiming assets in support of forward remanufacturing, demanufacturing operates mostly to divert wastes from disposal and reuse assets wherever practicable.

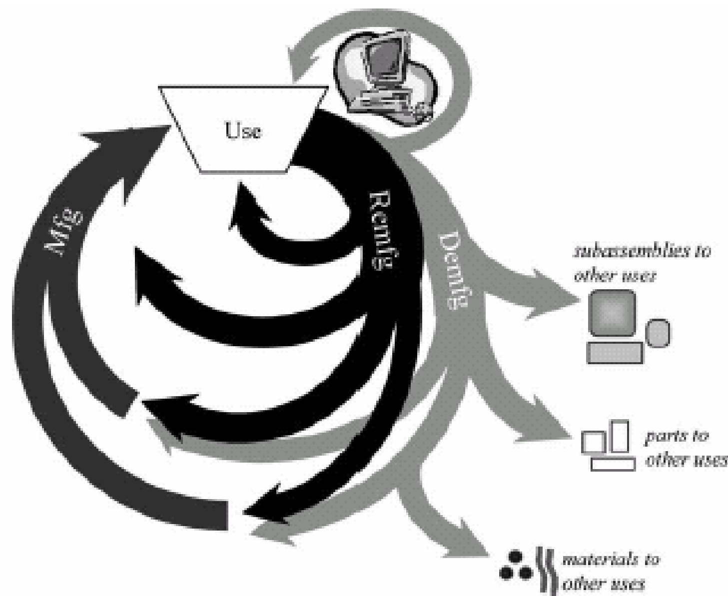


Figure 4 Remanufacturing and demanufacturing as the means of reverse manufacturing

It is seen that there are two main elements of reverse manufacturing; remanufacturing and demanufacturing. Both of the practices are as important as the manufacturing in the in the environmentally conscious world. Since they both rest heavily on the disassembly, disassembly should be as important as the assembly.

2.2 Literature Survey

Assembly line balancing (ALB) problem attracted a great deal of interest for fifty years (Flood, 1956; Talbot et al. 1986; Scholl and Klein, 1999). Researchers considered the problem of assigning a set of tasks to an ordered set of stations such that the precedence relations between the tasks are maintained and the number of stations is minimized. The main inputs to the assembly line balancing (ALB) studies are the task precedence diagram (TPD) and the durations of the tasks (Salveson, 1955; Held et al. 1962). Both of the inputs depend on how the definitions of the tasks are set. Hence, we believe that the questions of "*are there other ways of defining the tasks*?" and "*how does the duration and precedence of the tasks differ with respect to definitions*?" should be of interest. Although there are so many efforts on how to solve the ALB problem, there is little attention on how to define the tasks of the assembly process. The derivation of TPD is typically left to the 'engineering judgment' (Chow 1990). In a study by Prenting and Battaglin (1964), authors consider how to define the tasks and how to form the precedence diagram. They list some of the guidelines in *'element listing'* and *'diagramming'*. But they do not point out the scenario where the solution to

ALB problem changes with respect to the different TPD's formed by listing the elements in a different way.

As the research in ALB literature goes on, there appeared the new concept of disassembly as a result of the environmental concerns (Brennan et al. 1994; Zhang et al. 1997; Wiendahl et al. 1999; Gungor and Gupta, 1999; Lee et al.2001; Lambert, 2003). Firms, under the pressure of both governments and the NGO's, began to consider remanufacturing and demanufacturing as the means of profitability. Remanufacturing aims to recover the after-use products, while demanufacturing reemploys the post-use products by means of cannibalization and recycling. Both of them require disassembly process. When the disassembly studies occurred, researchers realized that there is not only a single way of defining tasks, as it was case in ALB studies. Consequently, De Mello and Sanderson (1990, 1991a and 1991b) established a graph that includes all possible assembly/disassembly task sequences of a product, called AND/OR Graph (AOG). Researchers used AOG in disassembly studies instead of TPD (Lambert, 1997, 1999, 2002; Penev and de Ron, 1996; Pnueli and Zussman, 1997; Rai et al. 2002; Johnson and Wang, 1995, 1998).

Disassembly line balancing (DLB) literature is a scarce one. DLB problem is first defined in the study by Gungor and Gupta (2001b and 2002). However, they use neither the TPD nor the AOG in their studies. They use a *part precedence diagram (PPD)*, which was developed in one of their early studies (Gungor and Gupta, 2001a). PPD involves the parts of the product, rather than the tasks. Since most of the researchers devote the TPD to assembly studies, there is no study that uses the TPD in DLB problems. As it will be clear later, there is no reason to not

using the TPD in DLB studies. The only study in this area that uses AOG is by Altekin (2005). The author considers not only the task durations but also task costs and subassembly profits. Hence, it is a profit oriented disassembly line balancing problem, involving costs revenues and planning horizon. The inputs are task durations and costs, subassembly demands and profits and station opening and operating costs. As a result of the solution, it may turn out that the product is fully or completely disassembled. Also, the decision of disassembly leveling may differ from one period to another. The gigantic model developed cannot be solved to optimality. The author develops some heuristic techniques to solve the integer programming formulation of the problem.

Chapter 3

PROPOSED THEORY: Questioning the Fundamental Assumptions and Solving the Actual Problem

After the disassembly studies began to draw considerable attention, researchers sought ways to compare and contrast assembly and disassembly. As mentioned in the literature, they usually state that disassembly problem is a more general form of the assembly since they used the AOG in disassembly, whereas only the TPD has been used in assembly studies. We discuss in this chapter that disassembly line balancing (DLB) problem is the reverse of the ALB problem. Furthermore, we prove that AOG is a more general version of TPD and should be used for both of the assembly and disassembly studies, as opposed to many that allocate the former to the disassembly and the latter to the assembly. As a result, both assembly and disassembly line balancing problems for AOG constitutes a lower bound on the same problem for TPD.

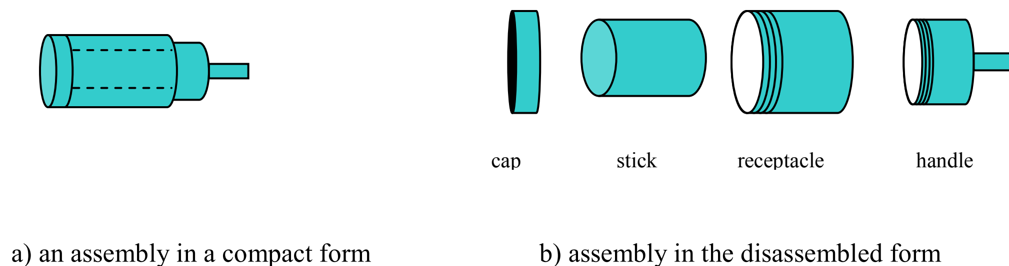
In the first section we introduce the concepts of AOG and AT/DT. We then consider the assembly/disassembly line balancing problem (ADLB) for an AOG, denoted ADLB-AOG, and compare it with the ADLB for a TPD, denoted ADLB-TPD. We prove that the solution to the ADLB-TPD is always an upper bound on the ADLB-AOG problem. Furthermore, in the third section, we show that any

TPD of a product can be obtained from the AOG of the same product. We give two examples on how to obtain a TPD from the AOG. Finally, in the last section, we give an example to illustrate the proposed theory in this chapter.

3.1 AND/OR Graph and Assembly/Disassembly Tree

3.1.1 AND/OR Graph (AOG)

In AOG, each disassembly task is assumed to take apart the product or subassembly into exactly two new subassemblies. Two connected arcs that link the resulting two subassemblies of the disassembly task with the input node is called a hyper-arc (There are fifteen of them in Figure 6). There are nodes and hyper-arcs corresponding to the subassemblies and the disassembly tasks, respectively (Figure 6 is the AOG of the product in Figure 5). To see more about the concepts used here, you may refer to de Mello and Sanderson (1990, 1991 a, 1991b) or the Appendix 1 of this study.



a) an assembly in a compact form   b) assembly in the disassembled form

Figure 5 A sample product (de Mello and Sanderson (1990))

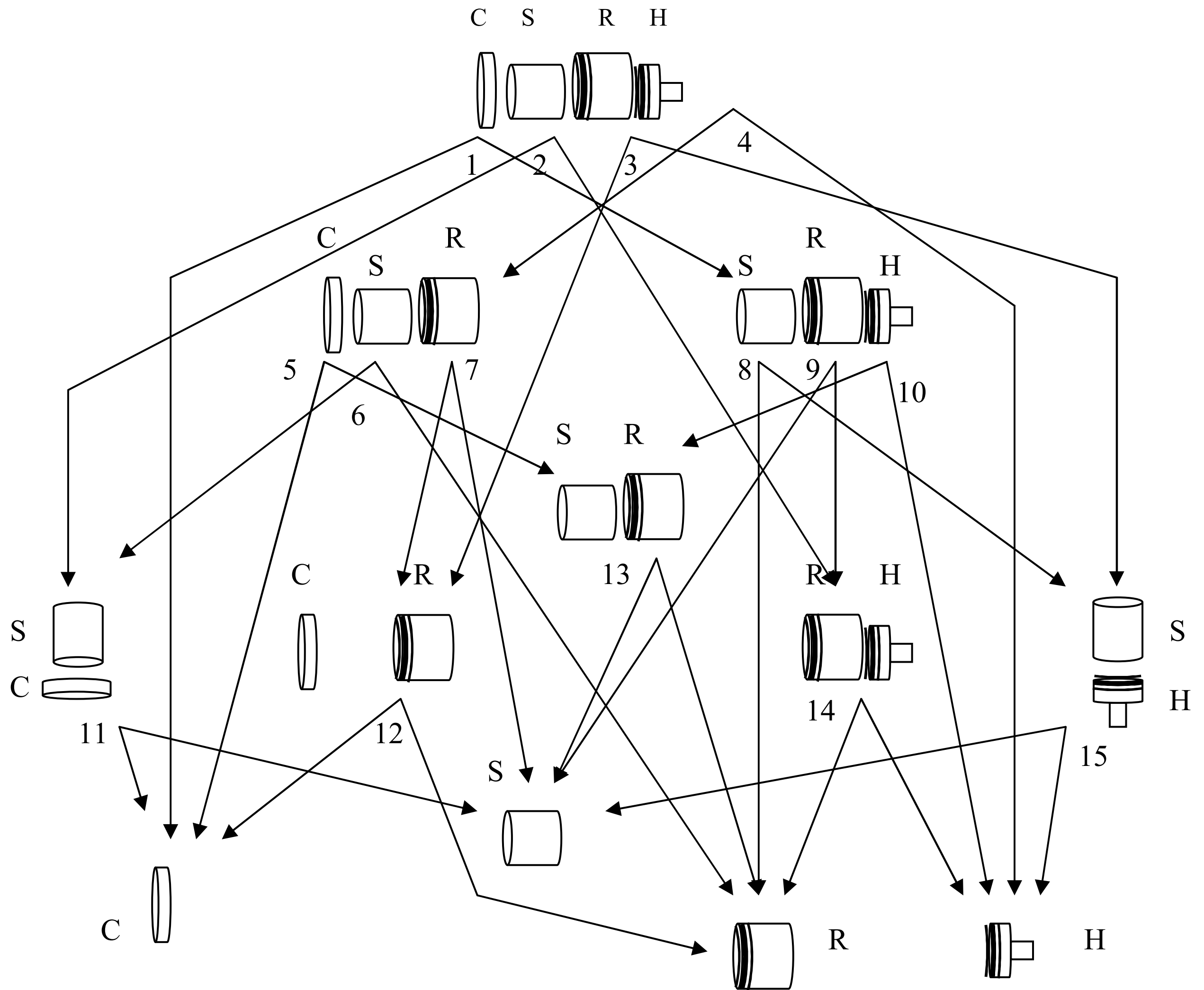


Figure 6 AND/OR Graph of the Product in Figure 5

To make a formal definition, let K be a set of elements and Π (K) be the set of all subsets of K. Consider a product *A* with parts $P_A = \{p_1, p_2, \ldots, p_N\}$. There is a unique *AND/OR graph (AOG)* of assembly/disassembly sequences for *A* defined as $<S_A, D_A>$ such that;

$$S_A = \{\theta \in \Pi (P_A) \mid sa(\theta) = \text{“T”} \wedge st(\theta) = \text{“T”}\} \quad [1]$$

is the set of nodes (subassemblies) in AOG and

$$D_A = \{(\theta_k, \{\theta_i, \theta_j\}) \mid [\theta_k, \theta_i, \theta_j \in S_A] \wedge [\tau(\theta_i, \theta_j) = \theta_k] \wedge gf(\tau) = \text{"T"} \wedge mf(\tau) = \text{"T"}\} \quad [2]$$

is the set of hyper-arcs, where $\wedge$ is the *and* operator. τ is the assembly task, *sa* and *st* are subassembly and stability predicates, *gf* and *mf* are geometrical and mechanical feasibility predicates (Appendix 1). We discuss some properties of AOG's in Section A1.4 of Appendix 1.

3.1.2 Assembly/Disassembly Tree (AT/DT)

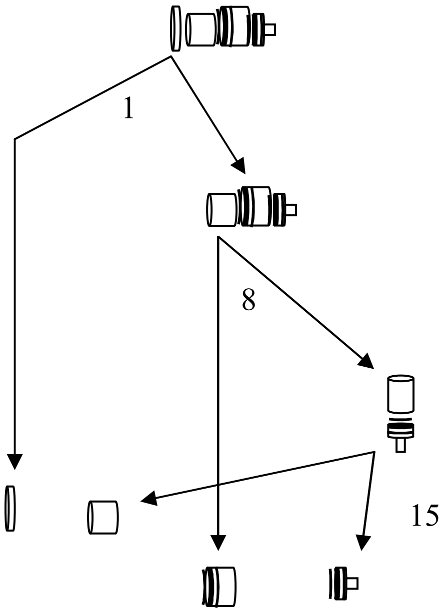
We only need a subset of the tasks in AOG to completely assemble/disassemble the product. The question is how to select these tasks so that, when applied one after another, they achieve this goal. In fact, due to the nature of AOG, the set of these tasks constitutes a tree.

In AOG, an hyper-arc is said to be *adjacent from* a node, if the subassembly associated with the node is the input subassembly of the disassembly task corresponding to the hyper-arc. Similarly, an hyper-arc is said to be *adjacent to a* node, if the subassembly associated with the node is the output subassembly of the disassembly task corresponding to the hyper-arc. Correspondingly, the node **to** which the hyper-arc is adjacent is called the *output node*, and the node **from** which the hyper-arc is adjacent is called the *input node*. Each hyper-arc is adjacent from one input node and adjacent to two output nodes. The node to which none of the hyper-arcs are adjacent is called the *initial node*, and the nodes from which none of the hyper-arcs are adjacent are called the *terminal nodes*.

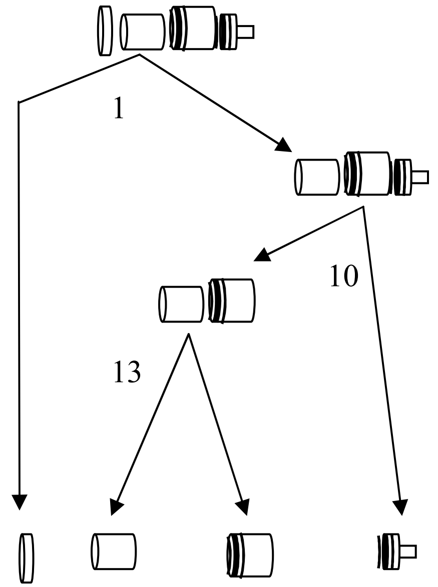
We define an *AND/OR path* in the graph as a set of hyper-arcs with k (>0) elements and their corresponding input and output nodes such that there are no two hyper-arcs that are adjacent from the same node, and there is only one initial node in the path. An AND/OR path that has k = N-1 elements is called *assembly/disassembly tree (AT/DT)* of the product. Note that, a DT (AT) should have $\{p_1, p_2, \ldots, p_N\}$ as its initial (terminal) node and $\{p_1\}, \{p_2\}, \ldots, \{p_N\}$ as its terminal (initial) nodes. Having N-1 hyper-arcs (assembly/disassembly tasks), a AT/DT represents one way of completely assembling/disassembling the product. In Figure 7, there are eight DT's corresponding to the AOG in Figure 6

There are precedence relations among the hyper-arcs in the DT: Hyper-arc h_i is said to immediately precede hyper-arc h_j, if the output node of h_i is the input node of h_j. Using these precedence relations between the hyper-arcs, we transform the AT/DT's to a new form called *transformed AT/DT (TAT/TDT)*. Although TAT/TDT's are like task precedence diagrams (TPD's), we will not call them as TPD to differentiate each other. Figure 8 shows the 8 TDT's corresponding to the 8 DT's in Figure 7. In this study, for the sake of simplicity we will use the terms AT/DT instead of TAT/TDT.

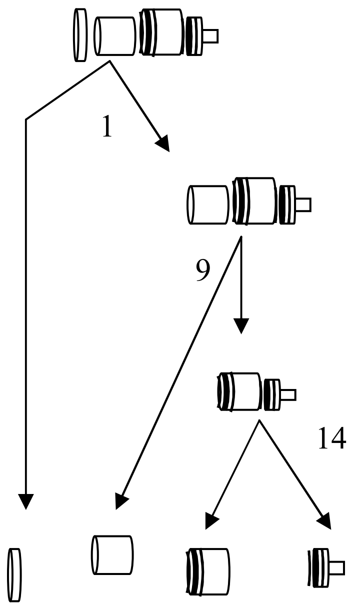
In graph theory, a directed graph is said to be a *tree* if the corresponding undirected graph has no cycles. The number of directed arcs adjacent from a node is called the *branch* of the graph at the corresponding node. Based on this terminology, it is appreciated that AT/DT's are trees with at most two branches. Expressing in the precedence-related words, in AT/DT, there is no task preceded by more than one task and there is no task preceding more than two tasks. Hence, AT/DT's are restricted versions of TPD's.



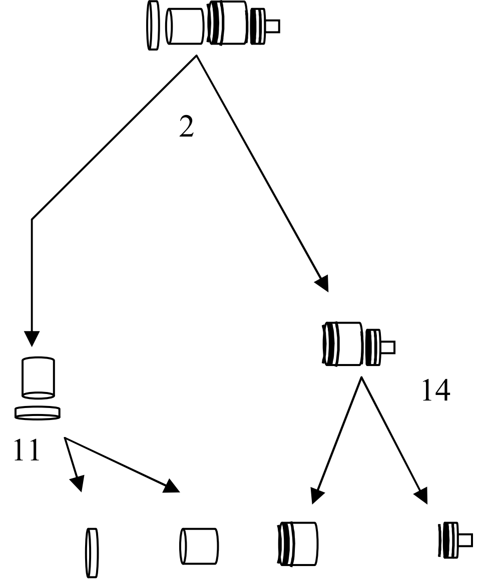
a) DT_1



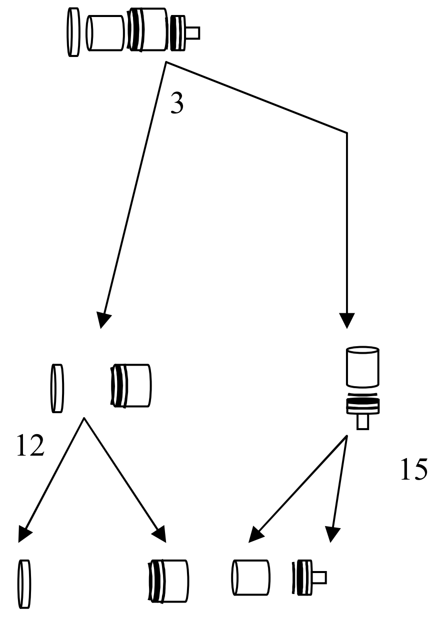
b) DT_2



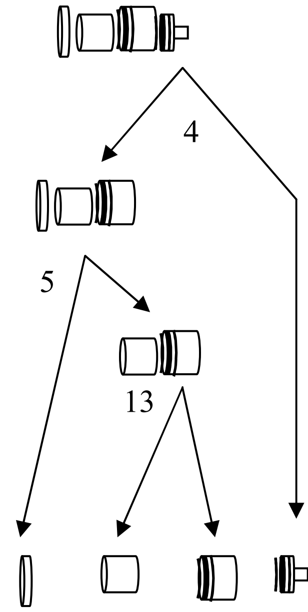
c) DT_3



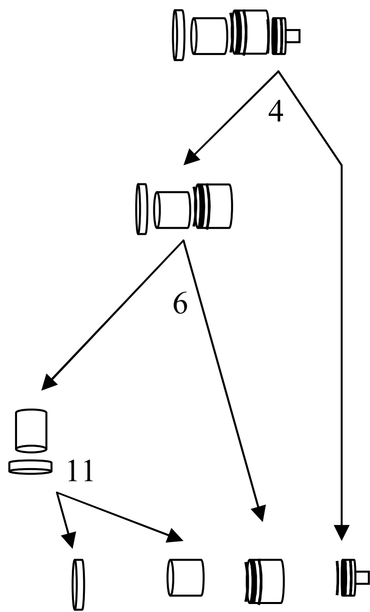
d) DT_4

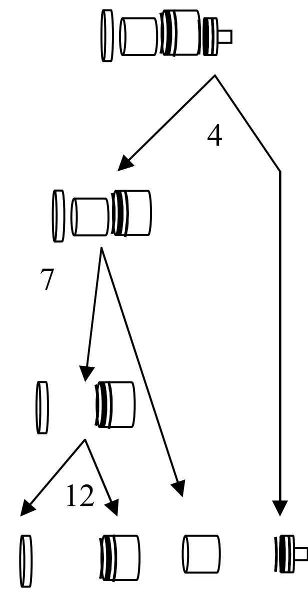


e) DT_5



f) DT_6

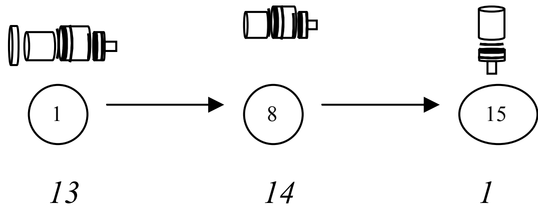




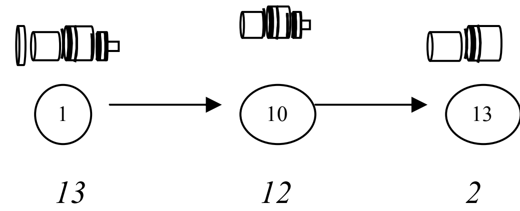
g) DT_7

h) DT_8

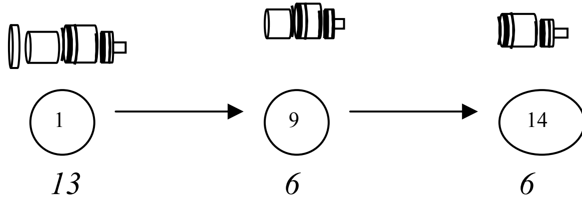
Figure 7 All of the 8 DT's obtained from the AOG in Figure 2



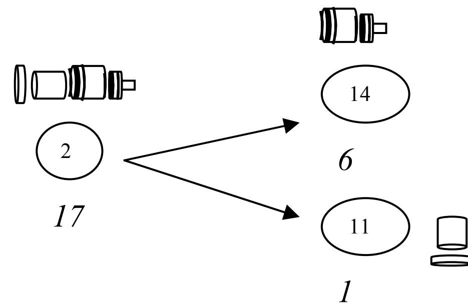
a) Transformed disassembly tree 1 (TDT_1)



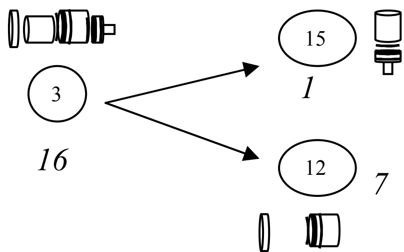
b) Transformed disassembly tree 2 (TDT_2)



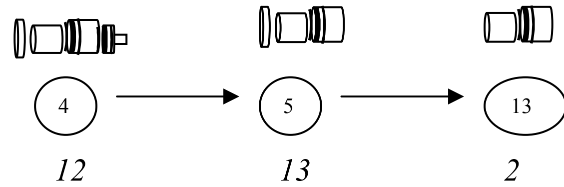
c) Transformed disassembly tree 3 (TDT_3)



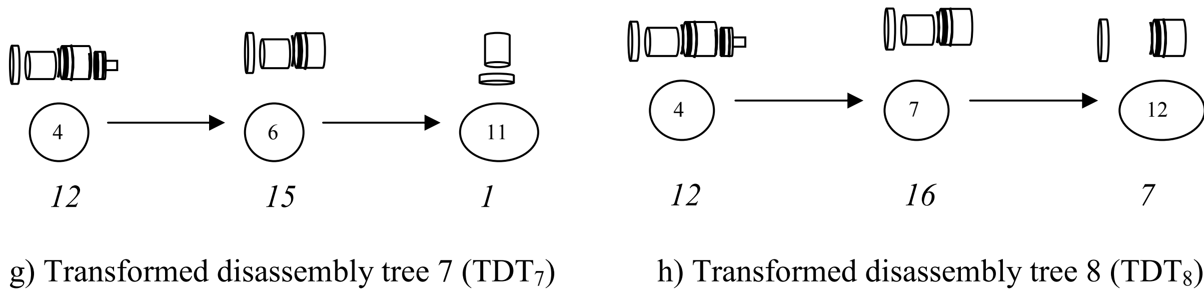
d) Transformed disassembly tree 4 (TDT_4)



e) Transformed disassembly tree 5 (TDT_5)



f) Transformed disassembly tree 6 (TDT_6)



g) Transformed disassembly tree 7 (TDT_7)   h) Transformed disassembly tree 8 (TDT_8)

Figure 8 Eight transformed disassembly trees (TDT) associated with the eight DT's in Figure 4

3.2 Theorem of Sub-optimality

We extend the definition of classical ALB problem to cover disassembly studies as well. Since the disassembly is just the reverse of the assembly, from now on, we call ALB problem as assembly/disassembly line balancing (ADLB) problem. The ADLB problem for a task precedence diagram (TPD) has a fifty years' of history. There are numerous approaches, both exact and heuristic (Erel and Sarin 1998, Baybars 1986). In this chapter, we define ADLB problem for an AOG and compare it with the ADLB problem for a TPD. To be concise, let ADLB-AOG show the latter problem and ADLB-TPD denote the former.

Definition 1 (ADLB-AOG problem): Choose a set of tasks from AOG such that the chosen set of tasks constitutes an AT, and the solution (number of stations) to the ADLB-TPD problem for the chosen AT is the best (minimum) of the ADLB-TPD problems over all possible AT's in AOG.

Both AOG and TPD include the precedence relations between the tasks to be applied to a product in the assembly/disassembly process. The tasks in AOG are defined based on two properties (Appendix A.1.2): (i) The subassembly it is applied to, (ii) and the contacts of the product it disassembles. The first property in the definition causes a set of tasks that disestablishes the same contacts to be labeled differently. For instance in Figure 10, tasks τ_1 and τ_5 break the same contacts (c_1, c_2), i.e., disassembles {cap} from {receptacle}, but are labeled as different tasks. Due to the first property, we call the tasks in AOG as *subassembly-dependent tasks (SD tasks)*. On the other hand, the tasks in TPD are defined based only on the second property above. As a result, the tasks that break the same contacts are labeled as the same task independent of the subassembly they are applied to. Hence, we call the tasks in TPD as *subassembly-independent tasks (SI tasks)*.

> **Theorem 1 (Theorem of sub-optimality)**:
>
> The optimal solution to the ADLB problem for a given TPD of a product (X^*_{TPD}) constitutes an upper bound on the optimal solution to the ADLB problem for the AOG (X^*_{AOG}) of the same product.

Proof: The solution to ADLB-TPD problem is a sequencing problem (Held and Karp, 1962). That is, each SI task sequence obtained from the TPD has its corresponding solution and one of these sequences characterizes the optimal solution. Based on this, we prove the theorem in two steps:

i) Any SI task sequence obtained from the TPD of a product is also obtainable from the AOG of the same product.

ii) The solution to an SI task sequence is an upper bound on the solution to the corresponding SD task sequence.

i) Each AT in AOG of a product includes a number of SD task sequences. Denote the set of AT's as S_{AT}. Re-label the tasks in each AT such that the tasks that break the same contacts are labeled as the same tasks. Denote the resulting trees as AT^m and the set of AT^m's as S_{AT^m}. Note that AT^m's include SI task sequences. We should show that any sequence of SI tasks in a TPD is obtainable from one of the AT^m's in S_{AT^m}.

Let a sequence of SI tasks be $\tau_{SI_1}, \tau_{SI_2}, ..., \tau_{SI_{n-1}}$, where n is the number of parts in the product. Although these tasks are subassembly independent by definition, they are actually performed on specific subassemblies. With the additional consideration of subassemblies these tasks are defined in relation to subassemblies, and are called SD tasks. Let the corresponding sequence of SD tasks be $\tau_{SD_1}, \tau_{SD_2}, ..., \tau_{SD_{n-1}}$. Since all possible ways to assemble/disassemble a product is embedded in AOG, the sequence $\tau_{SD_1}, \tau_{SD_2}, ..., \tau_{SD_{n-1}}$ is obtainable from one of the AT's in S_{AT}. Hence, the sequence $\tau_{SI_1}, \tau_{SI_2}, ..., \tau_{SI_{n-1}}$ is in S_{AT^m}.

ii) Let $X^*_{AT^m}$ be the best solution of the ADLB-TPD problem on S_{AT^m} and

X^*_{AT} be the best solution on S_{AT}.

The argument in the second step holds if and only if $X^*_{AT^m} \geq X^*_{AT}$.

If the duration of the SD tasks and corresponding SI tasks had been the same, the two solutions, $X^*_{AT^m}$ and X^*_{AT}, would be the same. The re-labeling step in the procedure does not only re-label the SD tasks but also changes the durations of the tasks. When one or more SD tasks are re-labeled to form a single SI task, the duration of the resulting SI task should be taken as the maximum of durations of these SD tasks. Otherwise, the cycle time constraint would be violated. To be more specific, (w.l.o.g) suppose that;

τ_{11}, τ_{12}, …, τ_{1k} is the only re-labeled SD tasks in AT_1, AT_2, …, AT_k,

τ_1 is the corresponding SI task in AT_1^m, AT_2^m, …, AT_k^m,

the duration of τ_{11} ($d_{\tau_{11}}$) is the maximum of $d_{\tau_{11}}$, $d_{\tau_{12}}$, …, $d_{\tau_{1k}}$

X_{AT_1} is not the best solution among X_{AT_1}, X_{AT_2}, …, X_{AT_k}

Suppose that the duration of SI task τ_1 (d_{τ_1}) is not set to the maximum duration ($d_{\tau_{11}}$). Then, $X_{AT_1^m}$ will be infeasible since the duration of τ_1 in AT_1^m is considered to be smaller than the original duration, which is the duration of τ_{11} in AT_1. Hence, the job may not be completed within the cycle time. On the other hand, if the duration of the SI task is set to the maximum of the durations of all corresponding SD tasks, there will be no violation of exceeding the cycle time.

Suppose that the duration of the SI task is set to the maximum of the durations of all corresponding SD tasks. The duration of τ_1 in AT_1^m will be same with the original duration, which is the duration of τ_{11} in AT_1. Hence,

$$X_{AT_1^m} = X_{AT_1}. \qquad [3]$$

But the duration of τ_1 in AT_2^m, AT_3^m, …, AT_k^m will be higher than the original durations, which are the durations of τ_{12}, τ_{13}, …, τ_{1k}, in AT_2, AT_3, …, AT_k. Hence,

$$X_{AT_2^m} \geq X_{AT_2},$$
$$X_{AT_3^m} \geq X_{AT_3}$$
$$\vdots$$
$$X_{AT_k^m} \geq X_{AT_k} \qquad [4]$$

by [3] and [4],

$$X^*_{AT^m} = \min\{X^*_{AT_1^m}, X^*_{AT_2^m}, \ldots, X^*_{AT_k^m}\} \geq \min\{X^*_{AT_1}, X^*_{AT_2}, \ldots, X^*_{AT_k}\} = X^*_{AT}$$

Figure 9 depicts theorem of sub-optimality.

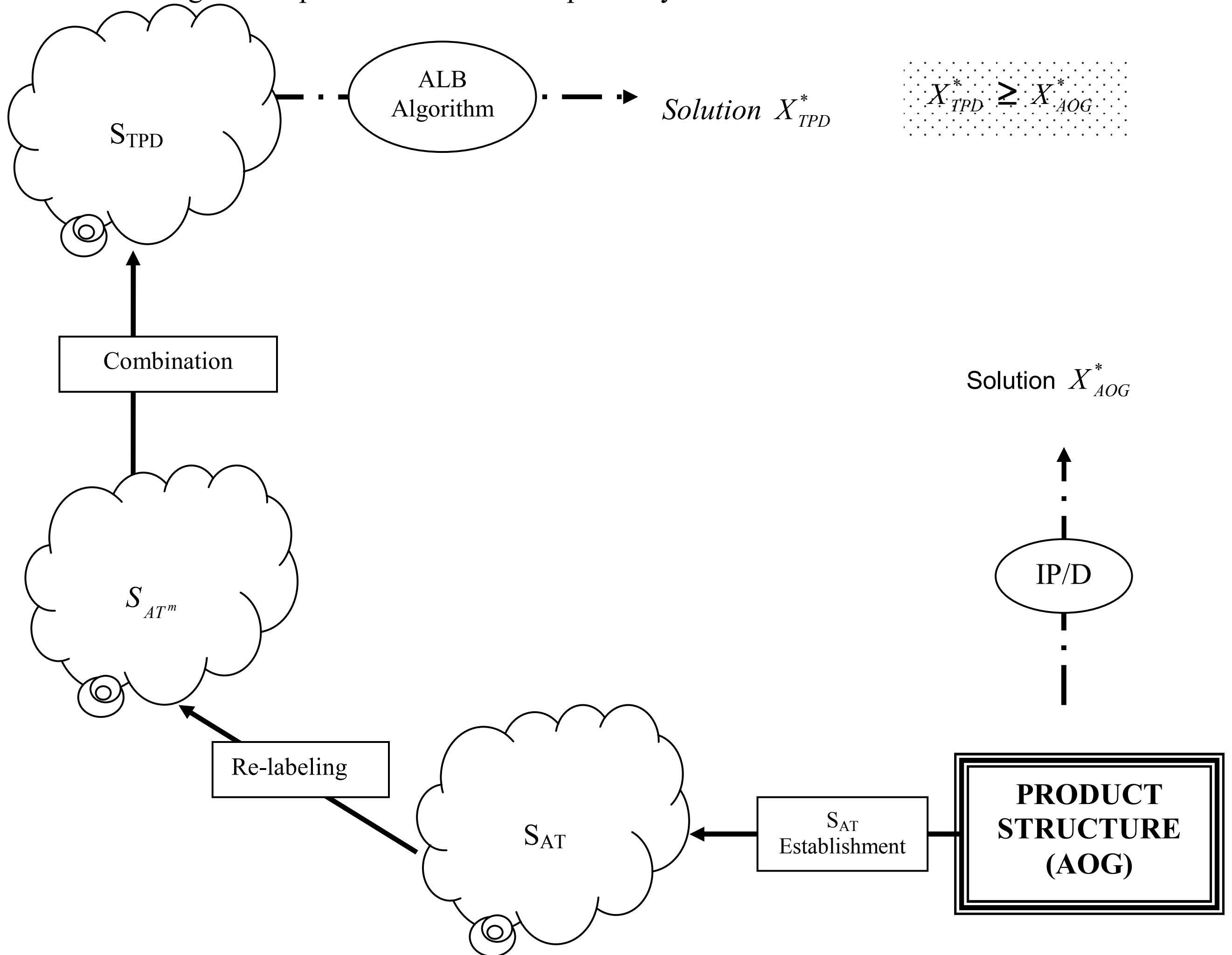


Figure 9 Theorem of sub-optimality

3.3 The derivation of a TPD from the AOG

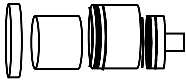
The AT^m's have the precedence relations between the SI tasks. In this respect, although it seems that each of them corresponds to a TPD of the product, they are still restricted version of a TPD, i.e., some of the AT^m's include exactly the same tasks and may be combined to form a TPD. Two AT^m's that have exactly the same tasks are called *equitask AT^m's*. To form a TPD from S_{AT^m}, one establishes the groups of equitask AT^m's and combines them to get one TPD for each group (Let the set of TPD's be S_{TPD}). For instance, suppose that AT_1^m and AT_2^m consist of two tasks τ_1 and τ_2. Also, τ_1 is the predecessor of τ_2 in AT_1^m and successor of τ_2 in AT_2^m. Then, AT_1^m and AT_2^m are *equitask AT^m's.* We combine AT_1^m and AT_2^m so that the resulting TPD does not have precedence relation between τ_1 and τ_2. Note that if there are no equitask AT^m's, each AT^m corresponds to a TPD.

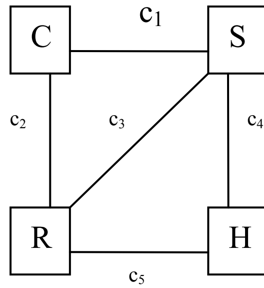
In all of the ALB studies so far, researchers used TPD's that only have AND-type precedence relations. In AND-type precedence relations, to accomplish a task one should perform all of its predecessors. On the other hand, to accomplish a task it may suffice to accomplish only one of its predecessors. We call these types of precedence relations OR-type. Inclusion of OR-type precedence relations in the TPD is more realistic. But in this study, while combining the AT^m's we only look for the AND-precedence relations between the tasks. The inclusion of OR-type precedence relations in establishing S_{TPD} from S_{AT^m} deserves a different study.

We give three examples for the derivation of TPD from the AOG.

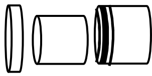
Example 1

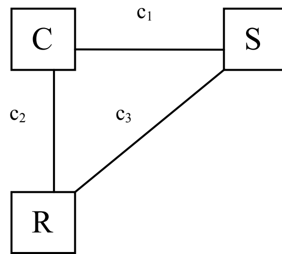
We consider the product in Figure 5. AOG of the product is in Figure 6. We obtain the set of AT's (S_{AT}) from the AOG in Figure 8. To re-label the SD tasks as SI tasks we need to know which contacts each of the SD task break. In Figure 10, each SD task is demonstrated by the corresponding contacts and the subassemblies they are applied. It is seen that tasks τ_1 and τ_5 disestablish the same contacts. Hence, they are re-labeled as E1. Also, the tasks τ_4 and τ_{10} are re-labeled as E2. After re-labeling the tasks the AT's become AT^m's. The resulting set of AT^m's (S_{AT^m}) is in Figure 11. We group the AT^m's that include exactly the same tasks (equitask AT^m's). There is only one group that includes more than one equitask AT^m, which are AT_2^m and AT_6^m. Hence, all of AT^m's except the second and the sixth are accepted as a TPD. AT_2^m and AT_6^m are combined to get TPD_5. We get seven TPD's. The resulting S_{TPD} is in Figure 12.



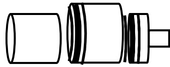


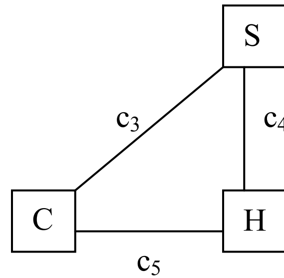
$\tau_1 \rightarrow c_1, c_2 = 13$
$\tau_2 \rightarrow c_2, c_3, c_4 = 17$
$\tau_3 \rightarrow c_1, c_3, c_5 = 16$
$\tau_4 \rightarrow c_4, c_5 = 12$



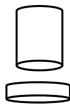


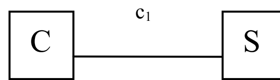
$\tau_5 \rightarrow c_1, c_2 = 11$
$\tau_6 \rightarrow c_2, c_3 = 15$
$\tau_7 \rightarrow c_1, c_3 = 6$



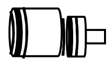


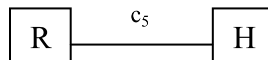
$\tau_8 \rightarrow c_3, c_5 = 14$
$\tau_9 \rightarrow c_3, c_4 = 6$
$\tau_{10} \rightarrow c_4, c_5 = 10$





$\tau_{11} \rightarrow c_1 = 1$





$\tau_{14} \rightarrow c_5 = 6$

a) subassembly

b) GOC

c) tasks with their durations and corresponding contacts

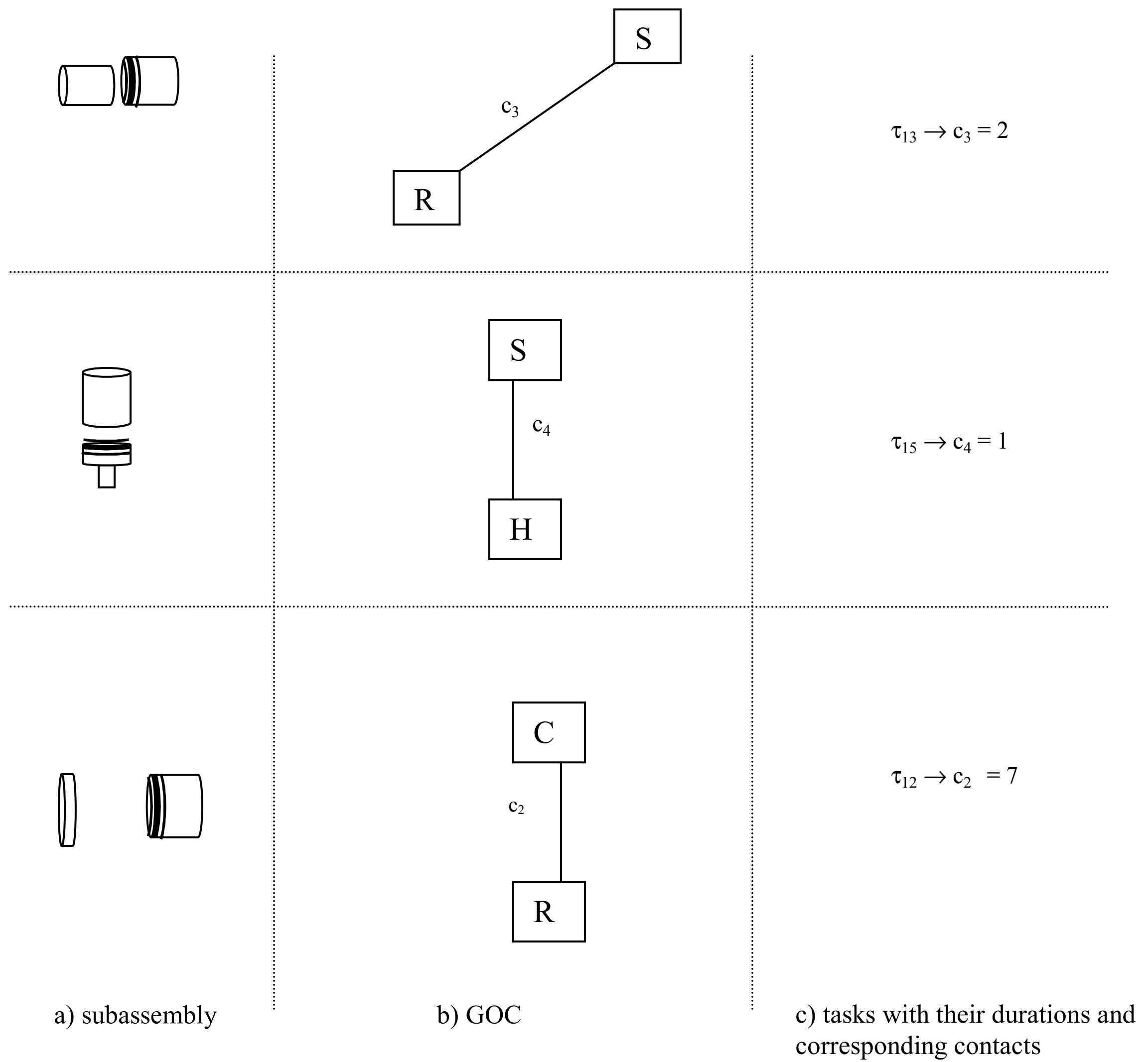


Figure 10 The subassemblies of AOG in Figure 6, their GOC and corresponding tasks

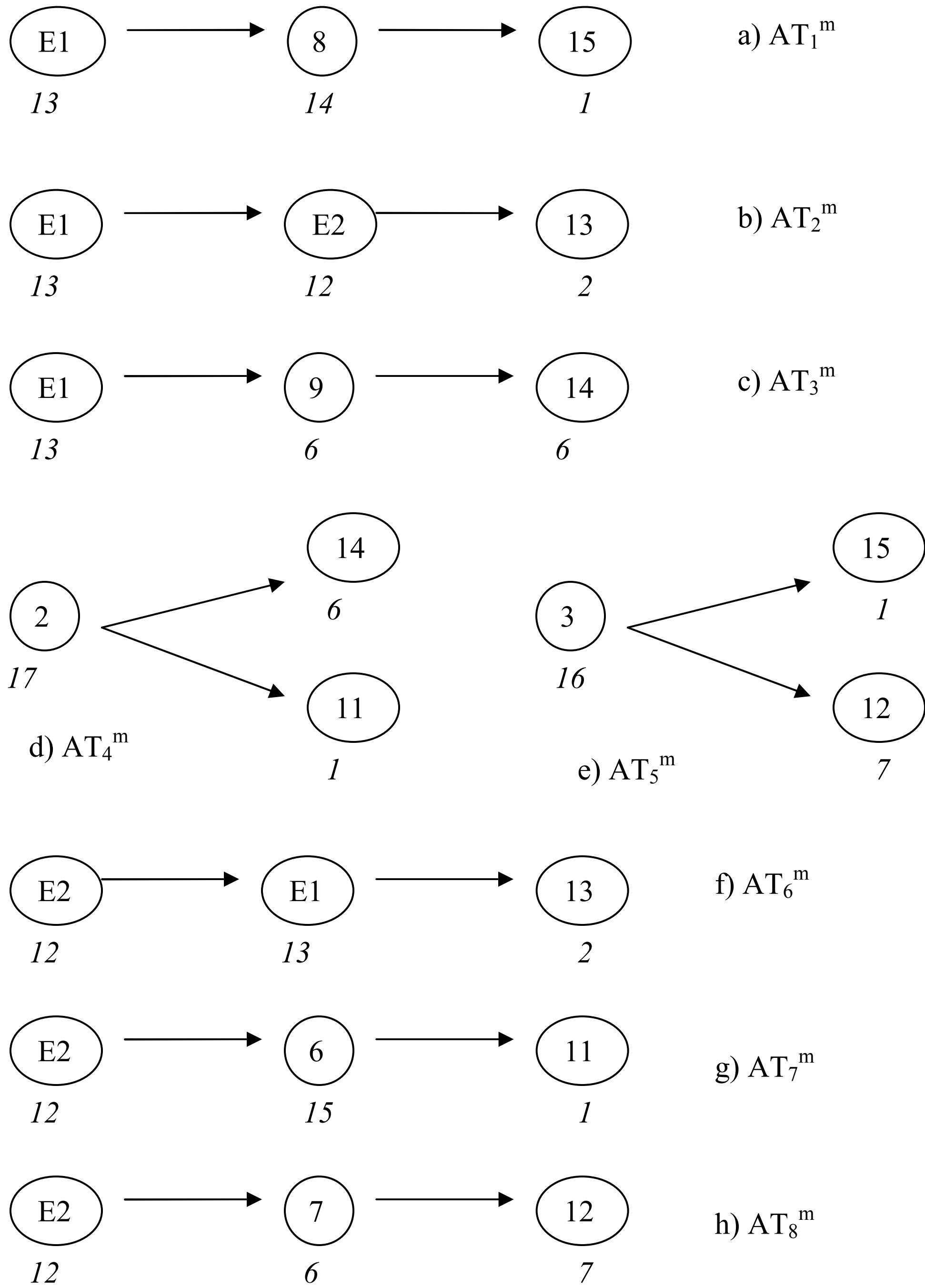


Figure 11 The S_{AT^m} obtained from S_{AT} in Figure 8

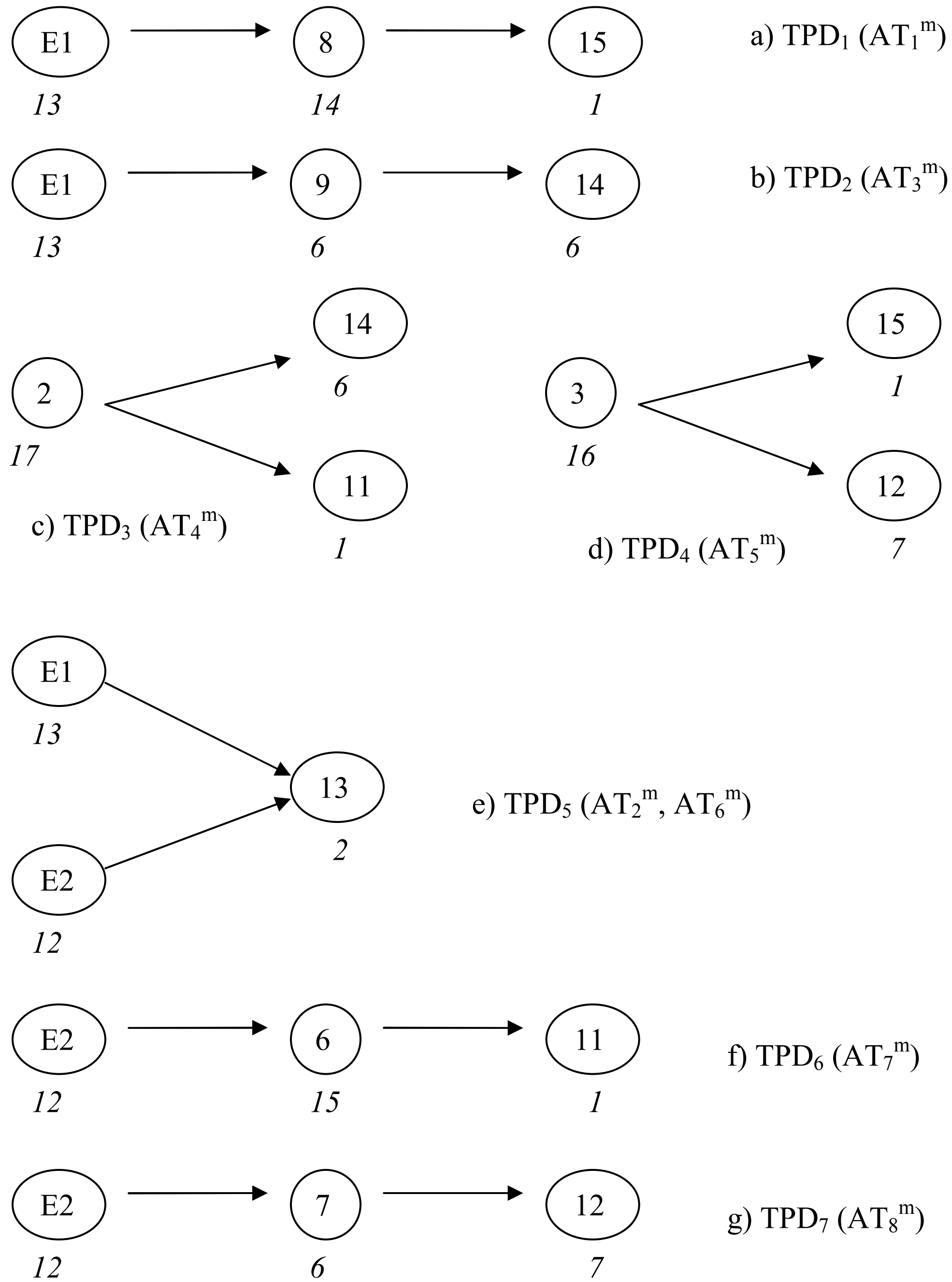


Figure 12 S_{TPD} obtained by combining equitask AT^m's in Figure 11

Example 2

All the related figures of this example are in Appendix 2. We consider the product in Figure A.7. We derive the AOG of the product (Figure A.8). Figure A.9 is the set of AT's (S_{AT}) obtained from the AOG. In Figure A.10, each SD task is demonstrated by the corresponding contacts and the subassemblies they are applied to. By examining the contacts we re-label the tasks as below:

$\tau_j = \tau_b = \tau_s \Rightarrow \tau_1$ (disassemble part 4)

$\tau_a = \tau_c \Rightarrow \tau_2$ (disassemble part 3 from part 9)

$\tau_k = \tau_o \Rightarrow \tau_3$ (disassemble part 2 from part 3)

$\tau_f = \tau_h = \tau_i \Rightarrow \tau_4$ (disassemble part 7 from part 8)

$\tau_p = \tau_d = \tau_m \Rightarrow \tau_5$ (disassemble part 9 from part 10)

$\tau_e = \tau_g = \tau_l = \tau_q \Rightarrow \tau_6$ (disassemble part 8 from part 9)

$\tau_n \Rightarrow \tau_n$ (disassemble part 7 from parts 5 and 6)

$\tau_r \Rightarrow \tau_r$ (disassemble part 5 from part 6)

$\tau_t \Rightarrow \tau_t$ (disassemble part 1 from part 2)

The resulting S_{AT^m} is in Figure A.11. We form the groups of equitask AT^m's. There is only one group. That is, each of the AT^m's has includes the same tasks. Hence, we combine all AT^m's to get a single TPD, which is in Figure A.12.

Example 3

We consider the product in Figure A.13 of Appendix A3. AOG of the product is in Figure A.14. Figure A.15 is the set of AT's (S_{AT}). In Figure A.16,

each SD task is demonstrated by the corresponding contacts and the subassemblies they are applied to. By examining the contacts we re-label the tasks as below:

$\tau_1 = \tau_6 = \tau_8 = \tau_{14} = \tau_{19} \quad \Rightarrow \quad \tau_a$

$\tau_2 = \tau_4 = \tau_{10} = \tau_{12} = \tau_{18} \quad \Rightarrow \quad \tau_b$

$\tau_3 = \tau_5 = \tau_7 = \tau_{11} \quad \Rightarrow \quad \tau_c$

$\tau_9 = \tau_{15} = \tau_{13} = \tau_{17} \quad \Rightarrow \quad \tau_d$

$\tau_{16} \quad \Rightarrow \quad \tau_{16}$

$\tau_{20} \quad \Rightarrow \quad \tau_{20}$

$\tau_{21} \quad \Rightarrow \quad \tau_{21}$

$\tau_{22} \quad \Rightarrow \quad \tau_{22}$

$\tau_{23} \quad \Rightarrow \quad \tau_{23}$

The resulting S_{AT^m} is in Figure A.17. We form the groups of equitask AT^m's. There are two groups of equitask AT^m's. We combine AT_2^m, AT_4^m, AT_5^m, AT_7^m, AT_8^m, AT_{10}^m, AT_{12}^m, AT_{13}^m, AT_{14}^m, AT_{15}^m, AT_{17}^m to get TPD_1 and AT_1^m, AT_3^m, AT_6^m, AT_9^m, AT_{11}^m, AT_{16}^m to get TPD_2. The two TPD's are in Figure A.18.

3.4 An Example to compare TPD and AOG.

To illustrate the whole discussion in this chapter, we take the AOG and two resulting TPD's in the Appendix 3. We assign the durations to the tasks of AOG as in the Table 1 below. Corresponding durations for the SI tasks of TPD_1 and TPD_2 are calculated as the maximum of the durations of the corresponding SD tasks.

Based on these durations, we solve ADLB-AOG and ADLB-TPD for two TPD's. We solve the three problems for each value of cycle time (T) from 22 to 90. Since the task with the minimum duration has the duration of 22, cycle time can not have the value less then 22. We limit the cycle time with 90 because above 90 the results of ADLB-AOG and ADLB-TPD problems are the same. The solutions (number of stations) for each problem are listed in Table 2.

Table 1 Durations of the SD and SI tasks in the example

AOG task	***1***	***6***	***8***	***14***	***19***	***2***	***4***	***10***	***12***	***18***	***3***	***5***	***7***	***11***	***9***	***15***	***13***	***17***	***16***	***20***	***21***	***22***	***23***
SD task duration	22	21	21	20	18	22	21	21	20	18	14	13	13	12	16	15	15	14	14	7	7	7	7
SI task duration	22					22					14				16				14	7	7	7	7
TPD task	***a***					***b***					***c***				***d***				***16***	***20***	***21***	***22***	***23***

In the Table 2, the leftmost column represents the problem when the cycle time is equal to that value. For instance, when the cycle time is 28 the ADLB-AOG problem yields a solution of 3 stations. On the other hand, the ADLB-TPD problem for TPD_1 yields a solution of 5 and the problem for the TPD_2 yields a solution of 4. Hence, when the cycle time is 28, with all the other data being fixed as given, the ADLB-AOG problems yields better solutions than the ADLB-TPD problem no matter which TPD is used.

Table 2 The solutions to the three example problems

T	AOG	TPD_1	TPD_2	T	AOG	TPD_1	TPD_2	T	AOG	TPD_1	TPD_2
22	4	5	5	***45***	2	2	2	***68***	1	2	2
23	4	5	5	***46***	2	2	2	***69***	1	2	2
24	4	5	5	***47***	2	2	2	***70***	1	2	2
25	4	5	5	***48***	2	2	2	***71***	1	2	2
26	4	5	5	***49***	2	2	2	***72***	1	2	2
27	4	5	5	***50***	2	2	2	***73***	1	2	2
28	3	5	4	***51***	2	2	2	***74***	1	2	2
29	3	4	4	***52***	2	2	2	***75***	1	2	2
30	3	3	4	***53***	2	2	2	***76***	1	2	2
31	3	3	4	***54***	2	2	2	***77***	1	2	2
32	3	3	4	***55***	2	2	2	***78***	1	2	2
33	3	3	4	***56***	2	2	2	***79***	1	2	2
34	3	3	4	***57***	2	2	2	***80***	1	2	2
35	2	3	4	***58***	2	2	2	***81***	1	2	2
36	2	3	3	***59***	2	2	2	***82***	1	2	2
37	2	3	3	***60***	2	2	2	***83***	1	2	2
38	2	3	3	***61***	2	2	2	***84***	1	2	2
39	2	3	3	***62***	2	2	2	***85***	1	2	2
40	2	3	3	***63***	2	2	2	***86***	1	2	1
41	2	3	3	***64***	1	2	2	***87***	1	2	1
42	2	3	3	***65***	1	2	2	***88***	1	1	1
43	2	3	3	***66***	1	2	2	***89***	1	1	1
44	2	2	2	***67***	1	2	2	***90***	1	1	1

As can be easily seen from the table that if we pick up the TPD_1 as the precedence diagram and solve the ADLB-TPD problem for the cycle times 22-90, we obtain the actual optimal 28 times and fail at 41 of them. When we solve the ADLB-TPD problem for TPD_2 we obtain the actual optimal 25 times and fail at 44 of them. To be on the optimistic side, if we take both of the TPD's, solve ADLB-TPD problem for each of them and take the best solution, we get the actual optimal 30 times and fail at 39 of them.

It is interesting that using even more than one TPD does not guarantee the optimal solution in ADLB-TPD problem. This is mainly due to the increase in durations of the SI tasks when they are re-labeled, as the theorem of sub-optimality suggests. When we think that researchers or practitioners consider only one TPD, the importance of ADLB-AOG problem stands out. What is more, although some TPD's has OR-type precedence relations, in the literature TPD's that are used has AND-type precedence relations. This deteriorates the solution of the ADLB-TPD problem further.

Chapter 4

THE SOLUTON TO THE ADLB-AOG PROBLEM

In this chapter we construct an integer programming (IP) and dynamic programming (DP), to solve the ADLB-AOG problem. We also compare these two methods in terms of the size of the ADLB-AOG problem solved.

In the formulation process we do not use the AOG since it does not show explicitly the precedence relations between the tasks. Instead, we develop a new graph, called *transformed AOG (TAOG)*. TAOG is formed as follows: Each node in the AOG corresponding to a subassembly is represented by an (artificial) node in TAOG. Each hyper-arc in the AOG associated with a task is represented by a (normal) node in TAOG. In TAOG, an artificial node is preceded by a normal node such that, in AOG, the hyper-arc associated with the node will be adjacent **to** the subassembly corresponding to the artificial node. Similarly, an artificial node precedes a normal node such that, in AOG, the hyper-arc associated with the node will be adjacent **from** the subassembly corresponding to the artificial node. We label the artificial nodes by A_i's and normal nodes by B_i's. In Figure 13 we give an example TAOG of the AOG in Figure A.14 of Appendix 3. From now on, to make the notation more manageable, we use AOG to denote TAOG.

An artificial node may be preceded or succeeded by more than one normal node. But only one of the predecessors and one of the successors should be

processed. Hence, predecessors and successors of the artificial nodes are 1OR-type meaning that exactly one of them must be chosen and it does not matter which one it is. To differentiate between the AND-type and OR-type relations, we put a small curve as indicator of OR-type relations. The fact that there are OR-type relations in AOG reveals that only some percent of the tasks is sufficient to assemble/disassemble the product completely, as opposed to the TPD in which all the tasks should be accomplished. Appendix 4 shows how to store an AOG (TAOG) in a matrix.

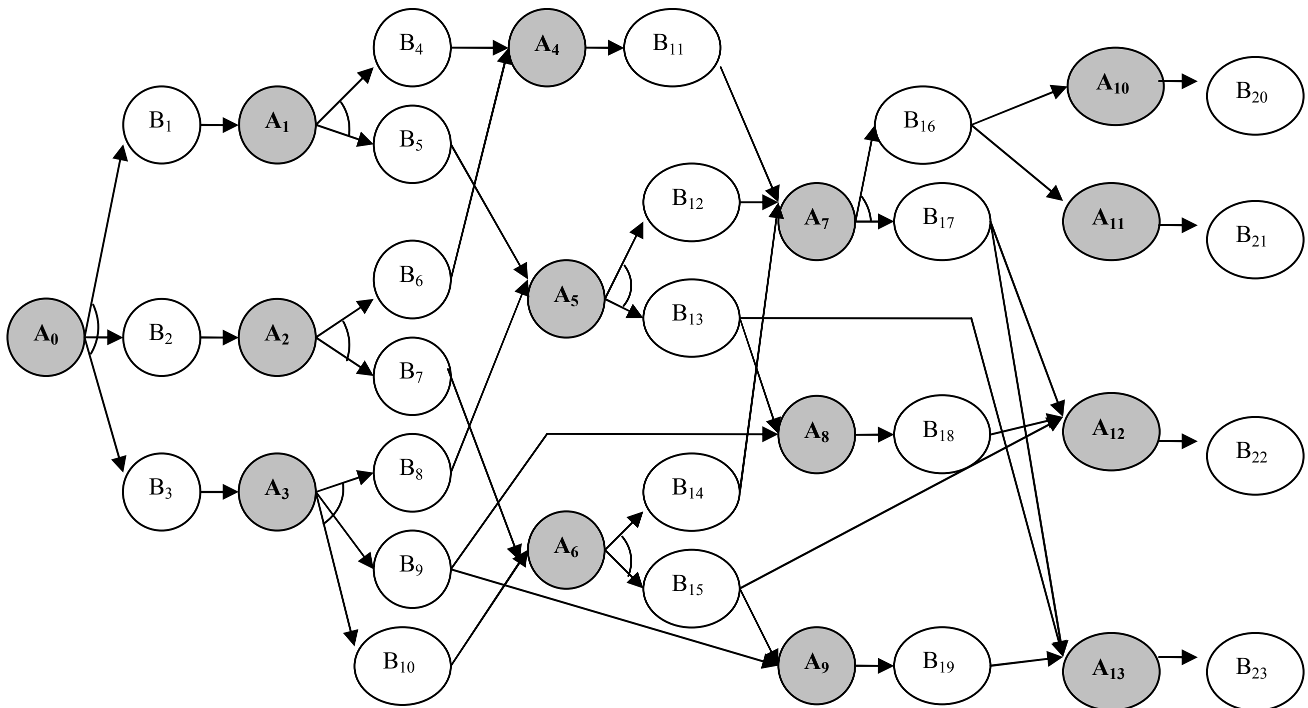


Figure 13 Transformed AND/OR Graph of the AOG in Figure A.14 of Appendix 3

4.1 The Proposed Dynamic Programming (DP) Formulation

The proposed DP approach solves the problem by finding the solutions to the partial problems, which eventually constitute the whole problem. It reduces the permutation-size solution space to combination-size (Held and Karp, 1962).

4.1.1 Definitions and Terminology

4.1.1.1 Partial AOG's

In the formulation of the problem we use some new terminology. A partial AOG, *AOG* $(\{A_i\})$, is defined as a graph obtained from AOG in such a way that all AT's to be obtained from that partial graph should have A_i as one of their final nodes. AOG $(\{A_i\})$ is obtained in two steps: First, delete nodes from AOG such that the AT's including the deleted nodes do not have the node A_i. Then, from the resulting AOG, delete the node A_i together with all of its successors. We then extend the definition of partial AOG to the following: Define *AOG* $(\{A_1, A_2, \ldots, A_k\})$ to be the graph obtained in *k* steps: First, find AOG $(\{A_{i_1}\})$ from AOG, then find AOG $(\{A_{i_1}, A_{i_2}\})$ from AOG $(\{A_{i_1}\})$, and so on until finding the AOG $(\{A_{i_1}, A_{i_2}, \ldots, A_{i_{k-1}}, A_{i_k}\})$ from AOG $(\{A_{i_1}, A_{i_2}, \ldots, A_{i_{k-1}}\})$. Note that the sequence of artificial nodes is arbitrary in this k-step procedure. By convention, AOG $(\{\emptyset\})$ = AOG, and AOG $(\{A_0\}) = \emptyset$. In Appendix 5, some examples to form partial AOG are given.

Let $S = \{A_1, A_2, \ldots, A_k\}$ be a set of artificial nodes. Final nodes of an AOG (S), denoted by *F(AOG(S))*, is defined as the set of normal nodes that do not

precede any other normal node in AOG (S). For instance the nodes B_{11}, B_{12}, B_{14} are the final nodes of the partial AOG in Figure A.21 of Appendix 5.

4.1.1.2 Assembly task sequences

We define assembly task sequences $\sigma = (B_1, B_2, \ldots, B_t)$ obtained from the normal nodes (tasks) of AOG to be *feasible* if;

i. $P(B_i) \neq P(B_j) \qquad \forall i \neq j$

ii. $|\{ B_1, B_2, \ldots, B_{i-1} \} \cap P(P(B_i))| = 1 \qquad \forall\ i = 2, 3, \ldots, t$

where $P(B_i)$ is the artificial predecessor of the normal node B_i, $P(P(B_i))$ is the normal predecessor of the artificial node $P(B_i)$ and | | is the cardinality operator defined on the sets.

The first property above prevents the sequence from having the two OR-successors of an artificial node simultaneously. For instance the sequence $\{B_1, B_4, B_5\}$ is prohibited by this property. Without the second property, the two normal nodes that are not OR-successors of the same artificial node but belong to the different AT's may exist in the sequence. For instance, the sequence $\{B_1, B_4, B_{11}, B_{13}\}$ is not allowed. Furthermore, the second property guarantees the normal nodes to follow the precedence relations dictated by the AT they belong to. For example, the second property does not allow the sequence $\{B_4, B_{11}, B_{16}, B_{20}, B_1, B_{21}\}$ since the nodes (tasks) are not in the correct order, although they belong to the same AT.

Final nodes of a sequence, denoted by $F(\sigma)$, are defined to be nodes of the sequence such that the sequence still remains feasible when they are removed from the sequence. For instance the tasks B_{20} and B_{21} are the final nodes of the sequence $\{ B_1, B_4, B_{11}, B_{16}, B_{20}, B_{21}\}$.

Associated with each feasible sequence σ is a particular assignment of tasks, represented by normal nodes, to the stations, called the induced assignment for σ. This assignment is obtained as follows: Assign as many tasks as possible from the beginning of the sequence to the first station, as many as possible from the beginning of the remaining subsequence to the second station, and so on, while not violating the cycle time (T) constraint. Intuitively, the induced assignment for a sequence is the optimal assignment. If the induced assignment for σ requires r stations and $w^{(r)}$ is the sum of durations of the tasks assigned to the last station, the quantity $c_\sigma = r\text{-}1 + \frac{w^{(r)}}{T}$ is a measure of the 'cost' of executing σ.

If a feasible sequence σ^* is formed by adjoining a task B_{t+1} to the end of σ, then

$c_{\sigma^*} = c_\sigma + \Gamma\,(c_\sigma,\ d_{B_{t+1}})$, where,

$$\Gamma(x, y) = \begin{cases} \lfloor x + y/T \rfloor - x + y/T & \text{if} \quad \lfloor x \rfloor < \lfloor x + y/T \rfloor < x + y/T \\ y/T & \text{if} \quad \lfloor x + y/T \rfloor = \lfloor x \rfloor \; or \; \lfloor x + y/T \rfloor = x + y/T \end{cases} \qquad [5]$$

where $\lfloor x \rfloor$ denotes the highest integer smaller than or equal to x.

The above equation can be interpreted as follows; if the unused idle time in the last station that is used by the induced assignment σ is greater than or equal to $d_{B_{t+1}}$, then $\Gamma = d_{B_{t+1}}$ / T; otherwise, new station is opened, which causes to the term related with the unused idle time to be added to $d_{B_{t+1}}$ / T in the computation of Γ.

4.1.1.3 Relation between partial AOG's and assembly task sequences

There is a natural correspondence between partial AOG's and feasible sequences defined by two mappings; G(σ) = { AOG(S) | F(σ) = F (AOG(S))},

G^{-1}(AOG(S)) = {σ | G (σ) = AOG(S)}.

Note that G is a *one-to-many* mapping. That is, for an AOG (S) the number of feasible sequences is greater than or equal to one, while for each feasible sequence there is only one AOG(S).

We define the cost of each partial AOG (S) as the cost of the sequence that has the minimum cost over all the sequences to be obtained from that partial graph. Hence,

$$C\ (AOG\ (S)) = \min_{\sigma \in G^{-1}(AOG(S))} c_\sigma$$

4.1.2 The Proposed DP Approach

From the discussion it follows that, solving the ADLB-AOG problem is equivalent to find the quantity C (AOG (∅)). Furthermore, the minimum number of stations required for the assembly line to perform the complete assembly/disassembly of the product is $\lceil C(AOG(\varnothing) \rceil$, where $\lceil x \rceil$ denotes the smallest integer greater than or equal to *x*.

Before the formal setting, one can see how the DP method works. In the solution of the problem (i.e., the induced assignment), one of the final nodes of the AOG (∅) will be the last task. This task is chosen among the final nodes of the AOG (∅). The solution of the problem is the minimum over the cost of the

sequences in which the last task is the chosen node. When the last node is chosen, say B_i, a graph that gives the rest of the solution is needed. This graph should be such that any task to be obtained from it should belong to the same AT of the AOG(∅) with the previously chosen task. This graph is AOG(∅∪P(B_i)). Proceeding in this manner until obtaining the AOG{A_0} gives the desired result.

It follows from the terms AOG (S), F (AOG) and the equation [5] that C (AOG(S)) can be calculated by the following recursion:

C (AOG (S)) =

$$\begin{cases} 0 & \text{for } S = \{A_0\} \\ \min\limits_{B_i \in F(AOG(S))} \{C(AOG(S \cup P(B_i))) + \Gamma(C(AOG(S \cup P(B_i))), d_{B_i})\} & \text{for } S \neq \{A_0\} \end{cases} \quad [6]$$

To see how the recurrence relations in [6] hold, it must be realized that, if the solution to ADLB-AOG problem for an AOG(S) yields a sequence of *t* tasks, the solution for the AOG(S ∪ P (B_i)) yields a sequence of *t-1* tasks, where $B_i \in$ F(AOG (S)). That is, the kth stage of the formulation is the set of AOG(S) that are solved to the sequences with *(n-1-k)* tasks, where n is the number of parts in the product. Hence, there are a total of *n* stages, together with stage 0, in the solution of the whole problem. Stage 0 has only one state, which is AOG (∅). Similarly, the final stage (stage *n-1*) has only one state, which is AOG ({A_0}). The number of states in the other stages depends on both the number of parts in the product and the geometry of the parts (Appendix 1 A1.4).

The recurrence relations in [6] enable us to determine C (AOG (∅)) by a computation involving only partial AOG's, which are much less than the number of feasible sequences. When the cost of AOG (∅) is found, the optimal sequence of the tasks can be obtained recursively by the equation below, finding B_t, then B_{t-1}, and so on to B_1:

$$C(AOG(S)) = C(AOG(S \cup P(B_i))) + \Gamma(CAOG((S \cup P(B_i))), d_{B_i})$$
$$where\ B_i \in F(AOG(S)) \qquad [7]$$

The proposed DP is implemented in Java. We give the code In Appendix 6.

4.1.3 Example

To illustrate the methodology explained in this section we solve an example problem for the AOG given in Figure 14. The durations of the tasks are given above the normal nodes. The cycle time (T) is 13.

There are three final nodes of AOG (∅), which are *h, i, j*. The partial AOG's corresponding to them are AOG ({A_3}), AOG ({A_4}) and AOG ({A_5}), respectively (Figure 15 –a, –b, –c). This gives the construction of stage 1 in Figure 16.

From [6], C (AOG (∅)) = min {C (AOG ({A_3})) + Γ (C (AOG ({A_3})), d_h),
C (AOG ({A_4})) + Γ (C (AOG ({A_4})), d_i),
C (AOG ({A_5})) + Γ (C (AOG ({A_5})), d_j)} [8]

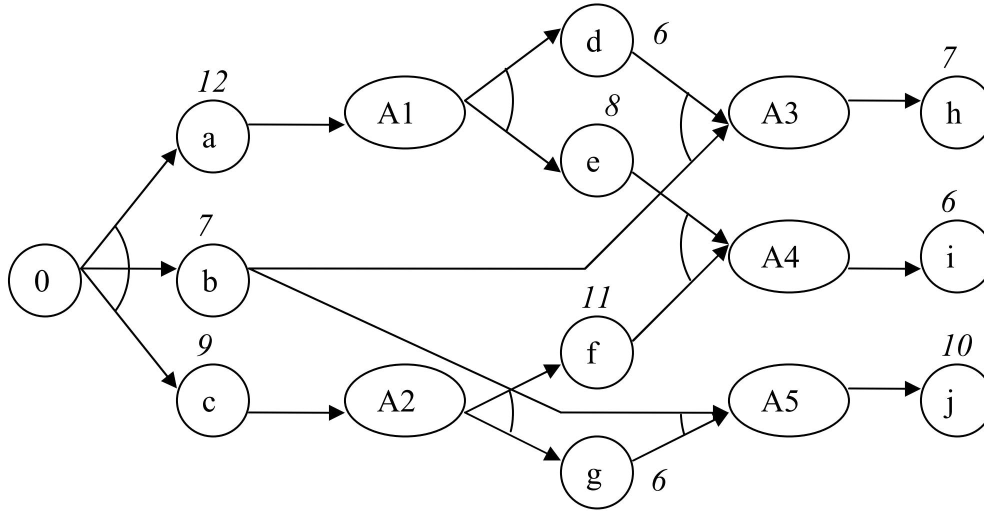


Figure 14 TAOG of the AOG in Figure A.4 of Appendix 1

There are two final nodes of AOG ({A_3}), which are *d, j*. The partial AOG's corresponding to them are AOG ({A_1}) and AOG ({A_3, A_5}), respectively (Figure 15 –d, –e). This is part of the construction of stage 2 in Figure 16.

From [6], C (AOG ({A_3})) = min {C (AOG ({A_1})) + Γ (C (AOG ({A_1})), d_d),

C (AOG ({A_3, A_5})) + Γ (C (AOG ({A_3, A_5})), d_j)} [9]

There are two final nodes of AOG ({A_4}), which are *e, f*. The partial AOG's corresponding to them are AOG ({A_1}) and AOG ({A_2}), respectively (Figure 15 –e, –f). This is part of the construction of stage 2 in Figure 16.

From [6], C (AOG ({A_4})) = min {C (AOG ({A_1})) + Γ (C (AOG ({A_1})), d_e),

C (AOG ({A_2})) + Γ (C (AOG ({A_2 })), d_f)} [10]

There are two final nodes of AOG ({A_5}), which are *g, h*. The partial AOG's corresponding to them are AOG ({A_2}) and AOG ({A_3, A_5}), respectively (Figure 15 –d, –f). This is part of the construction of stage 2 in Figure 16.

From [6], $C(AOG(\{A_5\})) = \min\{C(AOG(\{A_2\})) + \Gamma(C(AOG(\{A_2\})), d_g),$

$C(AOG(\{A_3, A_5\})) + \Gamma(C(AOG(\{A_3, A_5\})), d_h)\}$ [11]

There is only one final node of AOG ({A_1}), which is *a*. The partial AOG corresponding to it is AOG ({A_0}). This is part of the construction of stage 3 in Figure 16.

From [7], $C(AOG(\{A_1\})) = C(AOG(\{A_0\})) + \Gamma(C(AOG(\{A_0\})), d_a)$ [12]

There is only one final node of AOG ({A_3, A_5}), which is *b*. The partial AOG corresponding to it is AOG ({A_0}). This is part of the construction of stage 2 in Figure 16.

From [7], $C(AOG(\{A_3, A_5\})) = C(AOG(\{A_0\})) + \Gamma(C(AOG(\{A_0\})), d_b)$ [13]

There is only one final node of AOG ({A_2}), which is *c*. The partial AOG corresponding to it is AOG ({A_0}). This is part of the construction of stage 2 in Figure 16.

From [7], $C(AOG(\{A_2\})) = C(AOG(\{A_0\})) + \Gamma(C(AOG(\{A_0\})), d_c)$ [14]

By convention AOG ({A_0})) = ∅. Hence, $C(AOG(\{A_0\})) = 0$. [15]

In summary, the DP computations are as follows: (i) Using the equations [8]-[15], we get the solutions as;

C (AOG ({A_1})) = 12/13, C (AOG ({A_3, A_5})) = 7/13,

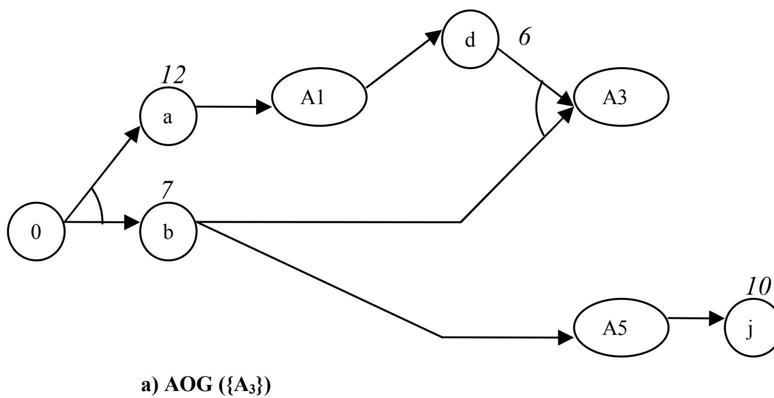
C (AOG ({A_2})) = 9/13, C (AOG ({A_3})) = 19/13,

C (AOG ({A_4})) = 21/13, C (AOG ({A_5})) = 19/13,

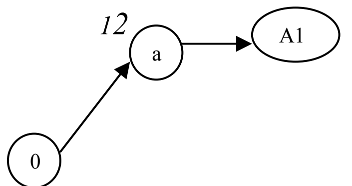
C (AOG ($\varnothing$)) = 2.

The optimal number of stations is $\lceil 2 \rceil = 2$.

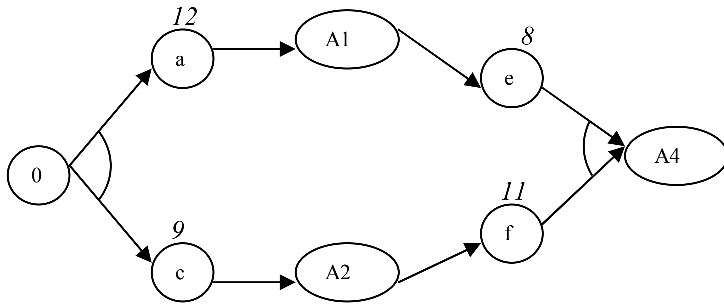
(ii) Using [8], we find two optimal paths: Assign task *a* to Station 1 and tasks *d* and *h* to Station 2, or assign task *a* to Station 1 and tasks *e* and *i* to Station 2.



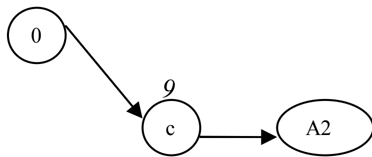
a) AOG ({A_3})



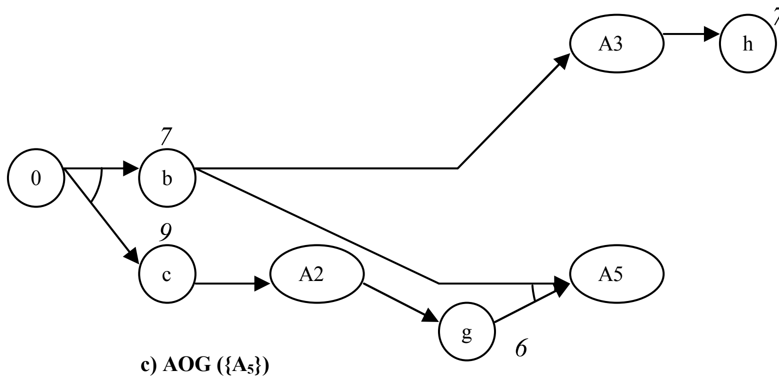
e) AOG ({A_3, A_1}) = AOG ({A_1})



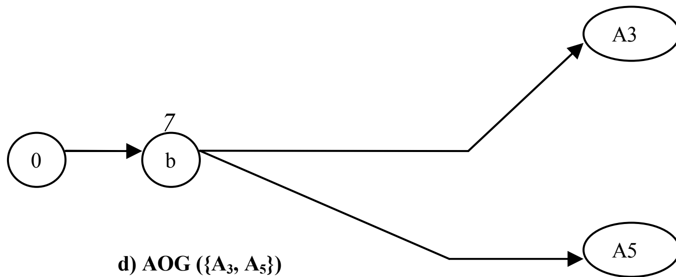
b) AOG ({A_4})



f) AOG ({A_2, A_4}) = AOG ({A_2})



c) AOG ({A_5})



d) AOG ({A_3, A_5})

Figure 15 The partial AOG's obtained while solving the example problem

The solution steps are depicted in Figure 16. Note that since the product has four parts, there are four stages including the stage 0. There is only one state of the stage 0, which is AOG (∅). Similarly, final stage has only one state, which is AOG({$\mathbf{A_0}$}). Consider construction of stage 1.

Note that the AT's composed of the tasks a-d-h and b-h-j have 25 and 24 task durations, respectively. Although the latter AT has less duration, the former yields the optimal induced assignment (optimal solution to the problem).

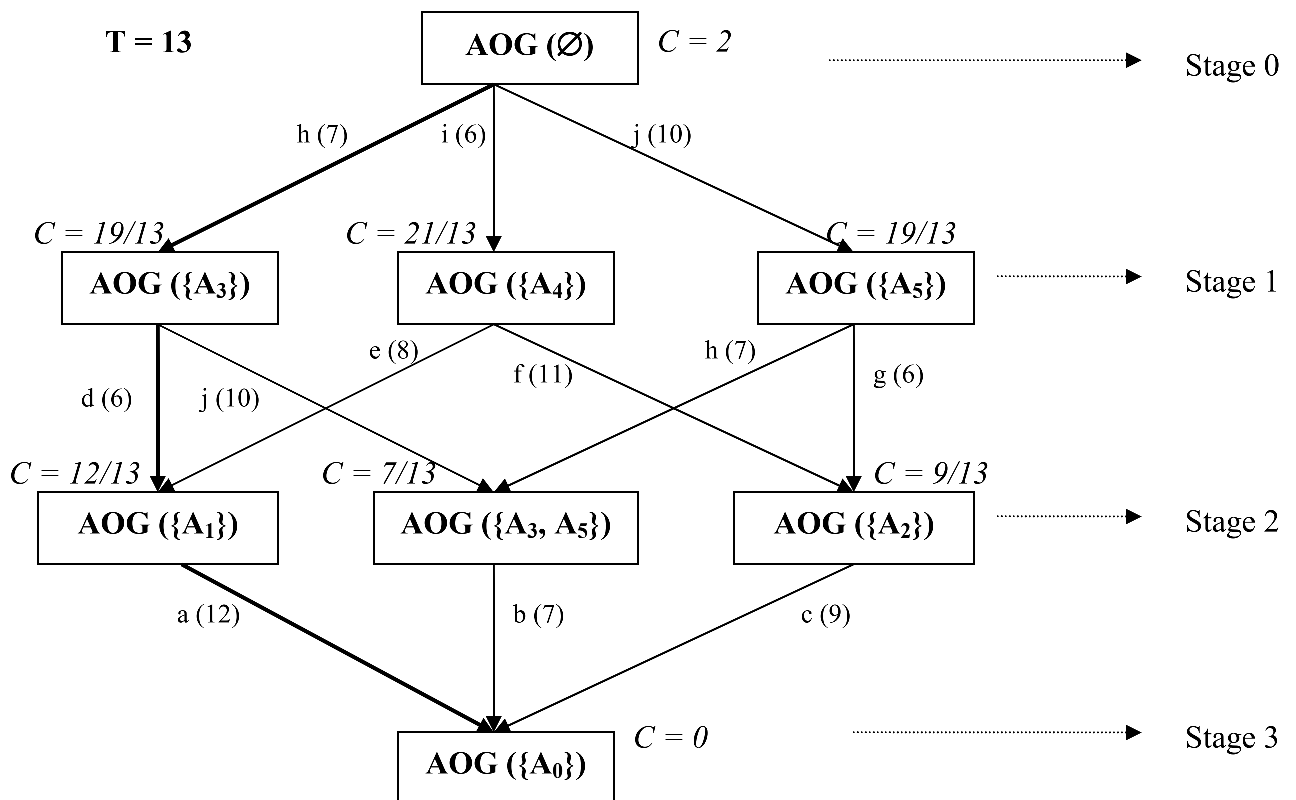


Figure 16 Dynamic programming solution to the ADLB-AOG problem for the AOG in Figure 14

4.2 The Proposed Integer Programming Formulation

4.2.1 The Formulation

In this section we formulate the ADLB-AOG problem as pure 0-1 integer programming problem. The formulation can also be used for the classical ADLB-TPD problems with slight modification.

<u>List of parameters</u>

A_k = artificial nodes in AOG   k = 0, 1, 2, …, *h*,

B_i = normal nodes, in AOG   i = 1, 2, …, *l*,

d_{B_i} = duration of node B_i

P (A_k) = immediate predecessor set of artificial node A_k.

S (A_k) = immediate successor set of artificial node A_k.

P (B_i) = immediate predecessor set of normal node B_i.

S (B_i) = immediate successor set of normal node B_i.

T = cycle time

M_j = station j   j = 1, 2, …, M

Note that *h* and *l* are function of the number of parts (n) in the product. Maximum values for them are calculated in Section A1.4 of Appendix 1. The number M may be taken as equal to the number of tasks (n-1), or an upper bound on the number of stations found by some heuristics. Note also that predecessor/successor of artificial nodes are OR-type, whereas those of normal nodes are AND-type.

Decision variables

Auxiliary variables

$$x_{ij} = \begin{cases} 1 & \text{if task } B_i \text{ is assigned to station } M_j \\ 0 & \text{otherwise} \end{cases}$$

$$f_j = \begin{cases} 1 & \text{if station } M_j \text{ is opened} \\ 0 & \text{otherwise} \end{cases}$$

$$z_i = \begin{cases} 1 & \text{if task } B_i \text{ is performed} \\ 0 & \text{otherwise} \end{cases}$$

The problem is formulated as, Minimize $\sum_{j=1}^{M} j \times f_j$

Subject to

$$\sum_{i:B_i \in S(A_k)} z_i = 1 \qquad \text{for } k = 0 \qquad [16]$$

$$\sum_{i:B_i \in S(A_k)} z_i = \sum_{i:B_i \in P(A_k)} z_i \qquad \text{for } k = 1, 2, \ldots, h \qquad [17]$$

$$\sum_{j=1}^{M} x_{ij} = z_i \qquad \text{for } \forall\ i = 1, 2, \ldots, l \qquad [18]$$

$$\sum_{i:B_i \in P(A_k)} \sum_{j=1}^{M} j \times x_{ij} \leq \sum_{i:B_i \in S(A_k)} \sum_{j=1}^{M} j \times x_{ij} \qquad \text{for } \forall\ k = 1, 2, \ldots, h \qquad [19]$$

$$\sum_{i=1}^{l} x_{ij} \times d_{B_i} \leq T \times f_j \qquad \text{for } j = 1, 2, \ldots, M \qquad [20]$$

$$x_{ij} \in \{0,1\}$$
$$f_j \in \{0,1\} \qquad [21]$$
$$z_i \in \{0,1\}$$

Constraints [16] and [17] assure that exactly one of the OR-successors is selected. Hence, in the course of establishing a solution these two constraints force the solution to be a set of tasks that constitute an AT. Constraint [18] makes sure that if the task is selected it is assigned to one of the stations; if not, it is not assigned. Constraint [19] handles the precedence relations between the normal nodes: Since exactly one of the OR-predecessors and one of the OR-successors of an artificial node will be selected, constraint [19] makes sure that the successor chosen among the OR-successors will be assigned to the higher-indexed station than the predecessor chosen among the OR-predecessors is assigned. Constraint [20] is the cycle time constraint that forces the total workload of a station to be less than the cycle time if the station is opened. If the station is not opened, constraint [20] forces the workload to be zero. Constraint [21] is the 0-1 integrity constraints.

Note that this formulation is a general case of the ALB problem studied in the literature. When the auxiliary variable z_i and the constraints [17], [18], [19] are eliminated, and the constraint [20] is modified as constraint [22] below, the IP formulation of ALB problem is obtained.

$$\sum_{j=1}^{M} j \times x_{rj} \leq \sum_{j=1}^{M} j \times x_{ij} \quad \forall i,\ B_r \in P(B_i) \qquad [22]$$

where $P(B_i)$ = predecessor set of task B_i.

4.2.2 Example

In this example, we formulate the ADLB-AOG problem for the AOG in Figure 14 with the cycle time of 13 as a pure 0-1 integer programming problem. In Appendix 7, we give a java code that takes an AOG matrix, duration array and cycle time constant as input and gives the IP formulation as output.

Minimize 1f1+ 2f2+ 3f3

Subject to

Z1+ Z2+ Z3 = 1 — Constraint for A_0

Constraints for A_1-A_5:

Z4+ Z5- Z1 = 0
Z6+ Z7- Z3 = 0
Z8- Z2- Z4 = 0
Z9- Z5- Z6 = 0
Z10- Z2- Z7 = 0

Precedence constraints:

1X1,1+ 2X1,2+ 3X1,3- 1X4,1- 2X4,2- 3X4,3- 1X5,1- 2X5,2- 3X5,3 <= 0
1X3,1+ 2X3,2+ 3X3,3- 1X6,1- 2X6,2- 3X6,3- 1X7,1- 2X7,2- 3X7,3 <= 0
1X2,1+ 2X2,2+ 3X2,3+ 1X4,1+ 2X4,2+ 3X4,3- 1X8,1- 2X8,2- 3X8,3 <= 0
1X5,1+ 2X5,2+ 3X5,3+ 1X6,1+ 2X6,2+ 3X6,3- 1X9,1- 2X9,2- 3X9,3 <= 0
1X2,1+ 2X2,2+ 3X2,3+ 1X7,1+ 2X7,2+ 3X7,3- 1X10,1- 2X10,2- 3X10,3 <= 0

Constraints for B_1-B_{10}:

X1,1+ X1,2+ X1,3- Z1 = 0
X2,1+ X2,2+ X2,3- Z2 = 0
X3,1+ X3,2+ X3,3- Z3 = 0
X4,1+ X4,2+ X4,3- Z4 = 0
X5,1+ X5,2+ X5,3- Z5 = 0
X6,1+ X6,2+ X6,3- Z6 = 0
X7,1+ X7,2+ X7,3- Z7 = 0
X8,1+ X8,2+ X8,3- Z8 = 0
X9,1+ X9,2+ X9,3- Z9 = 0
X10,1+ X10,2+ X10,3- Z10 = 0

cycle time constraints:

12X1,1+ 7X2,1+ 9X3,1+ 6X4,1+ 8X5,1+ 11X6,1+ 6X7,1+ 7X8,1+ 6X9,1+ 10X10,1- 13f1 <=0
12X1,2+ 7X2,2+ 9X3,2+ 6X4,2+ 8X5,2+ 11X6,2+ 6X7,2+ 7X8,2+ 6X9,2+ 10X10,2- 13f2 <=0
12X1,3+ 7X2,3+ 9X3,3+ 6X4,3+ 8X5,3+ 11X6,3+ 6X7,3+ 7X8,3+ 6X9,3+ 10X10,3- 13f3 <=0

integrity constraints:

$x_{ij} \in \{0,1\}$
$f_j \in \{0,1\}$ i = 1,…, 10 j = 1, …., 3
$z_i \in \{0,1\}$

We formulated the problems by the java code, which is in Appendix 7.

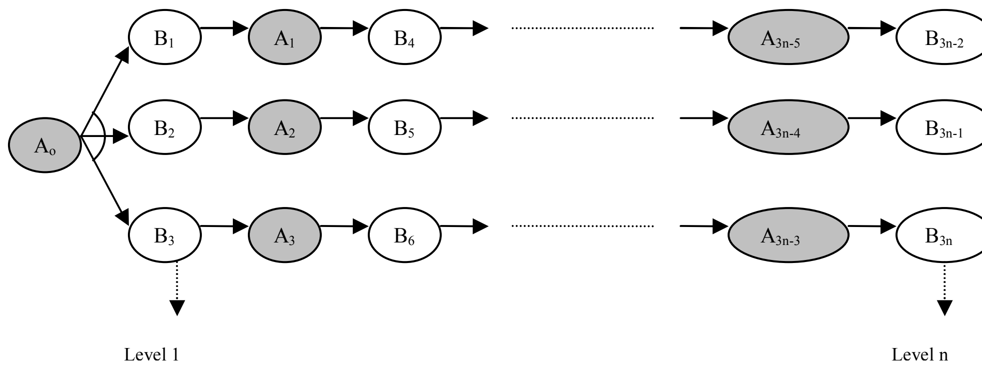
4.3 Solvable sizes of ADLB-AOG problem by DP and IP methods

In this section, we compare the two exact methods in terms of the size of the ADLB-AOG problem. There are three variables that define the size: The first one is the number of the tasks (normal nodes) in the solution, denoted by *n*. Due to the main assumption of AOG that each task disassembles a subassembly into exactly two subassemblies or parts, the number of tasks in the solution is one less than the number of parts in the product. For example, all of the AOG's in Figure 17 belong to the products having *n+1* parts. Second is the number of artificial nodes at each level in the AOG, denoted by *a*. The number of artificial nodes at each level may differ from one level to another. But we took them equal to each other in order to standardize the AOG's so that future studies can easily compare their findings with ours. In Figure 17a, c the parameter *a* is 3, whereas in Figure 17b it is 5.

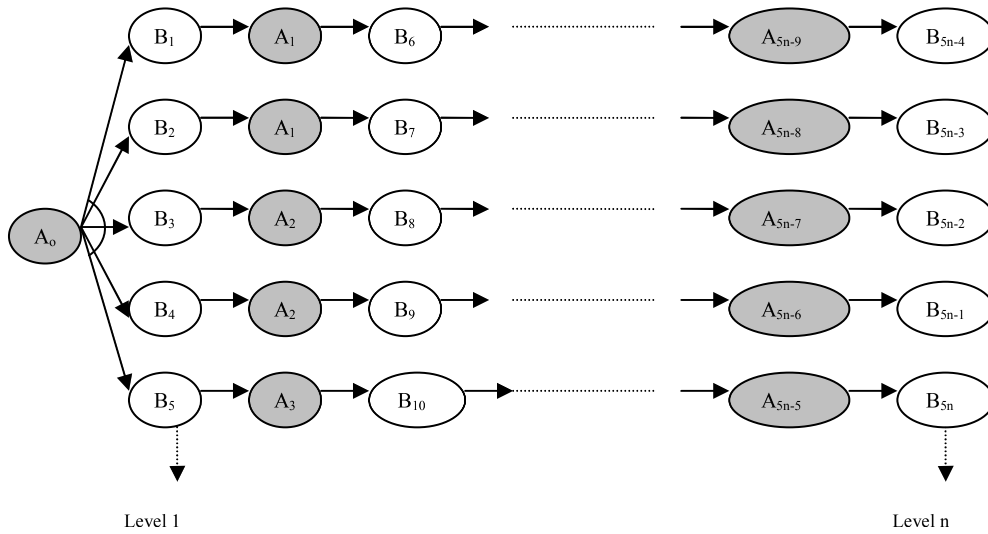
Third determinant of the size is the number of tasks (normal nodes) for each artificial node, except the first artificial node and last *a* artificial node (i.e., A0, $A_{an-(a-1)}$, $A_{an-(a-2)}$, …, A_{an-1}, A_{an}), where *a* is the number of artificial nodes at each level. This parameter is denoted by *t*. In Figure 17a and b the parameter *t* is 1, whereas in Figure 17c it is 2. When the parameter *t* is greater than one, say *x*, we generated the AOG's as follows: Assign the successors of the artificial nodes at level *y* to the artificial nodes at level *y+1* one by one. The first successor of the first artificial node at level *y* precedes the first artificial node at level *y+1*, the second successor precedes the second artificial node, etc. The first successor of the second artificial node precedes the $(x+1)^{st}$ artificial node, the second successor

precedes the *(x+2)*nd artificial node, etc. Whenever no unassigned artificial nodes are remained at level *y+1*, we start from the first artificial node again. Consider the Figure 17c. The parameter *t* is 2. The first successor (B_4) of the first artificial node (A_1) at level one precedes the first artificial node (A_4) at level two; the second successor (B_5) precedes the second artificial node (A_5); the first successor (B_6) of the second artificial node (A_2) precedes the third artificial node. Then, the artificial nodes at level two are finished. Hence, the second successor (B_7) precedes the *first* artificial node (A_4), and so on.

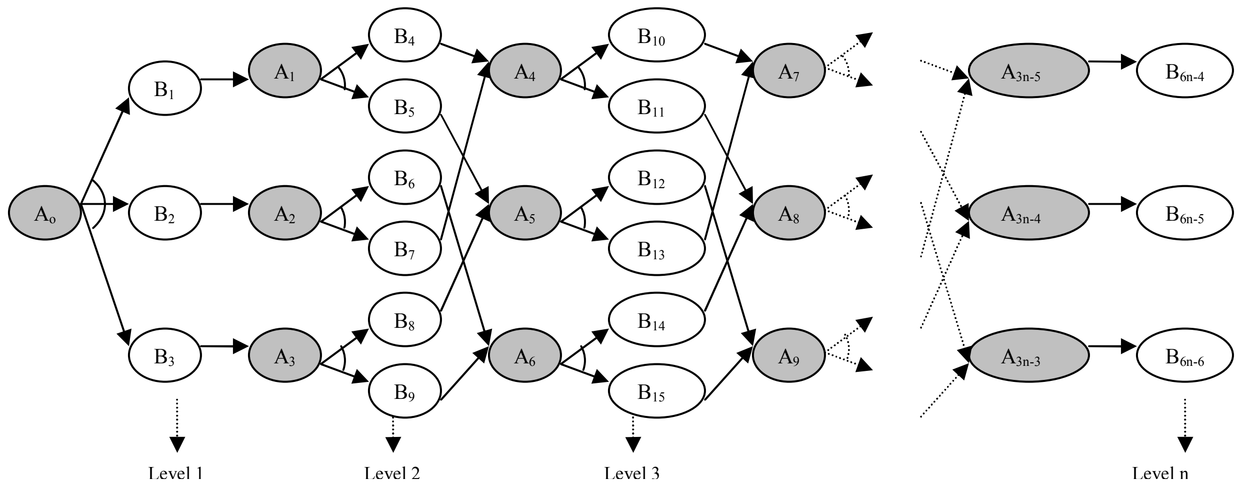
Based on these parameters, total number of artificial nodes in the AOG together with the node A_0 is $a \times (n-2)+1$. Total number of normal nodes is $a \times [t \times (n-3)+2]$.



a) the sample AOG with a = 3, t = 1.



b) the sample AOG with a = 5, t = 1.



c) the sample AOG with a = 3, t = 2.

Figure 17 Sample AOG's to illustrate the experimentation

We did not consider a factor that brings randomness, which is the type of the tasks. There are two types of tasks: the first one is named as *sequential task* and the latter as *parallel task* (Srinivasan et al. 1997). Sequential tasks disassemble only one part from the subassembly, whereas parallel tasks disassemble a subassembly into two subassemblies each of them having at least two parts. In the matrix notation of AOG, the row corresponding to the sequential task has only one entry of 1 and one entry of -1; whereas the row corresponding to the parallel task has two entries of 1 and one entry of -1. Since the output subassemblies of a parallel task may differ from level to level, parallel task bring randomness. On the other hand, the sequential task does not bring randomness since its output subassembly is at one level below its input subassembly. To avoid this randomness we first allow only the sequential tasks to exist in the AOG and find the size of the solvable problem. After, we allow the parallel tasks randomly to exist in the AOG and obtained the results.

In our computational experiments we decided that the problem is solvable if it can be solved without exhausting the memory and within 10 minutes of CPU time in the Pentium 4 Processor with 2.66 GHz using 512 MB RAM. The time limit may seem too strict. Due to the fact that a large number of runs are required to solve the problem is too much, we used strict time limits.

The JAVA code that generates sample AOG's according to the parameters is given Appendix 8.

4.3.1 The DP formulation

Before the experimental results, we will discuss the impact of the parameters *n, a, t* on the DP method. As discussed in Section 4.1.2, the number of stages in the DP formulation is equal to the number of parts in the product (*n*).

We show that the number of states in each stage is equal to the parameter *a*. Denote the set of partial AOG's in the stage *k* by G_k. Let $P(B_i)$ be the predecessor of the normal node B_i, $S(A_i)$ be the successor of the artificial node A_i, and *F(AOG(S))* be the final nodes of the partial AOG. Let H_k = {F(AOG(S)) | AOG(S) $\in$ G_k. }, and A_k = {P(F(AOG(S))) | AOG(S) $\in$ G_k. }. It is easy to see that $A_k = \{A_{a(n-k)-(a-1)}, A_{a(n-k)-(a-2)}, \ldots, A_{a(n-k)-1}, A_{a(n-k)}\}$.

Let $H_k = \bigcup_{t=1}^{a} H^{kt}$, where $H^{kt} = S(A_{a(n-k)-(a-t)})$, and $G_k = \bigcup_{t=1}^{a} G^{kt}$, where G^{kt} = {AOG(S) | AOG(S) $\in G_k \wedge$ F (AOG(S)) $\cap H^t \neq \varnothing$}, where $\wedge$ is the 'and' operator. Let Z^t = {AOG(S $\cup B_i$) | AOG(S) $\in G^t \wedge B_i$ = F (AOG(S)) $\in H^t$}.

Note that $\bigcup_{t=1}^{a} Z^t = G_{k+1}$. Furthermore, due to the definition of the partial AOG's, $Z^i \cap Z^j = \varnothing$, where $0 < i < j < a$; and $| Z^i | = 1$, where $0 < i < a$. Hence, $G_{k+1} = a$, where $0 \leq k < n-2$ and *n* is the number of parts in the product.

It is seen that the parameter *t* does not impact the number of states and number of stages. It affects the number of connections between the states of the successive stages. That is, each state in the stage *k* is connected with *t* stats in the stage *k+1*.

Table 3 Solvable size of the AOG's without parallelism by the DP approach.

Number of artificial nodes at each level (a)	Nuber of tasks for each artificial node (t)	Maximum solvable number of parts	Total number of artificial nodes	Total number of normal nodes	Stopping criteria
3	1	249	742	744	OM
	2	98	289	576	OM
	3	87	256	762	TR
	5	67	196	966	TR
	10	38	109	1056	TR
4	1	225	893	896	OM
	2	74	289	576	OM
	3	64	249	740	TR
	5	48	185	908	TR
	10	27	101	968	TR
5	1	206	1021	1025	OM
	2	60	291	580	OM
	3	53	256	760	OM
	5	35	166	810	TR
	10	21	96	910	TR
10	1	159	1571	1580	OM
	2	32	301	600	OM
	3	24	221	650	TR
	5	17	151	720	TR
	10	12	101	920	TR

Above discussion implies the followings: If one memory space is assigned to each state, a total of $(n-2)\times a+2$ memory space is required. The computations consist of additions and comparisons, which occur in equal amount. The number of additions in the first phase of the DP method, which is finding the optimal cost, is $a\times[(n-3)\times t+2]$. The number of additions in the second phase, finding the optimal path, is $a+(n-3)\times t+1$. It is seen that both the memory space and the

computation time is polynomial with respect to all parameters. This shows the efficiency of the DP method for the ADLB-AOG problem.

In real life applications, it is desired to know how big product can be assembled and disassembled. Hence, we keep *a* and *t* constant and find the solvable size of the problem with varying values of *n*. Table 3 is the results of the hundreds of runs to determine the solvable size of the AOG's without parallelism by the DP approach. We allow the parameter *t* to take the values of 1, 2, 3, 5 and 10. For each of those values we allow the parameter *a* to take the values of 3, 4, 5 and 10. For each combination of these to factors, we run the DP approach many times, increasing the number of parts in the product, i.e. the parameter *n*, each time, until deciding that the AOG can not be solved by the DP approach. The stopping reason for each case is given at the rightmost column in the table. 'OM' stands for the 'out of memory' case and 'TR' denotes the failure of time requirements.

Many inferences can be drawn from Table 3. Figures 18-22 show both the number of parts and the corresponding total number of tasks in AOG vs. the number of artificial nodes at each level.

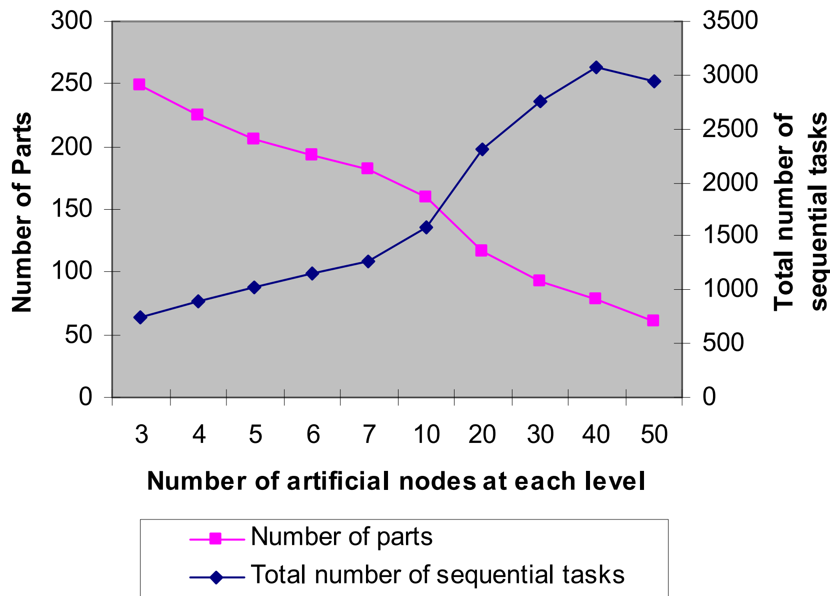


Figure 18 Solvable problem sizes when the number of tasks for each artificial node is 1

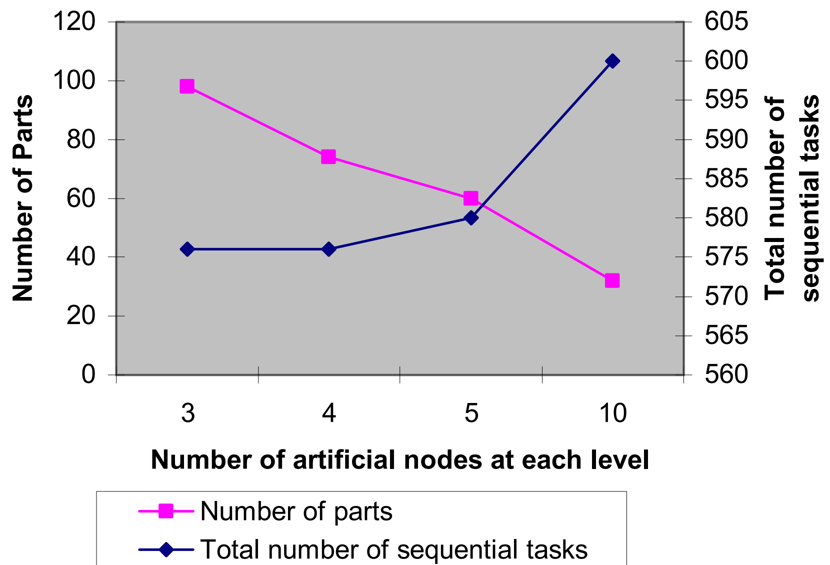


Figure 19 Solvable problem sizes when the number of tasks for each artificial node is 2

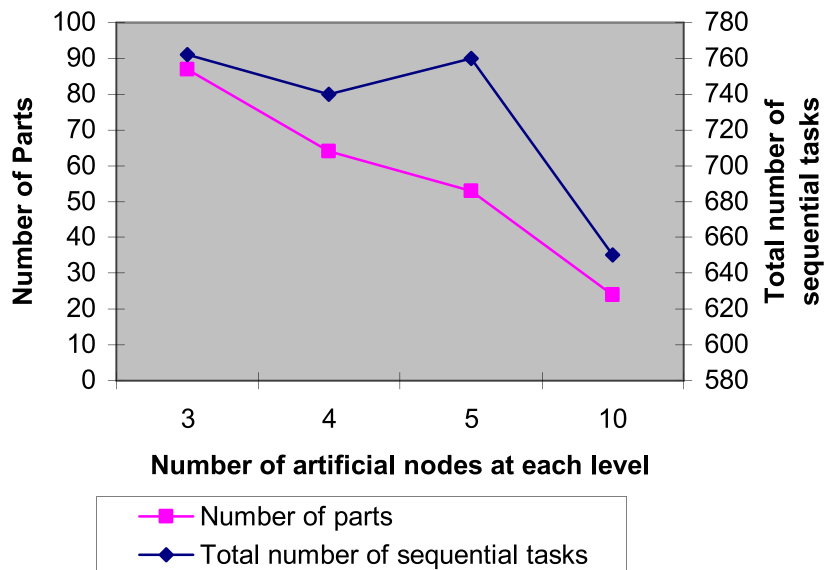


Figure 20 Solvable problem size when the number of tasks for each artificial node is 3

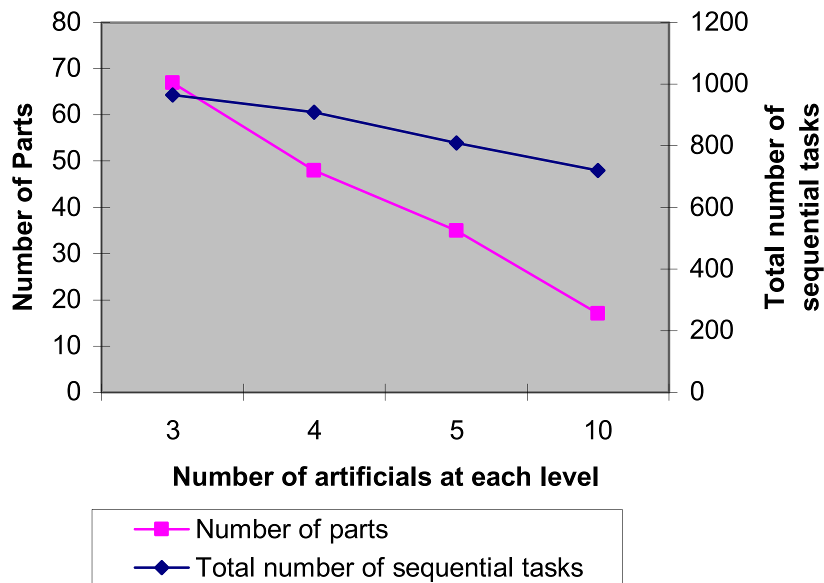


Figure 21 Solvable problem size when the number of tasks for each artificial node is 5

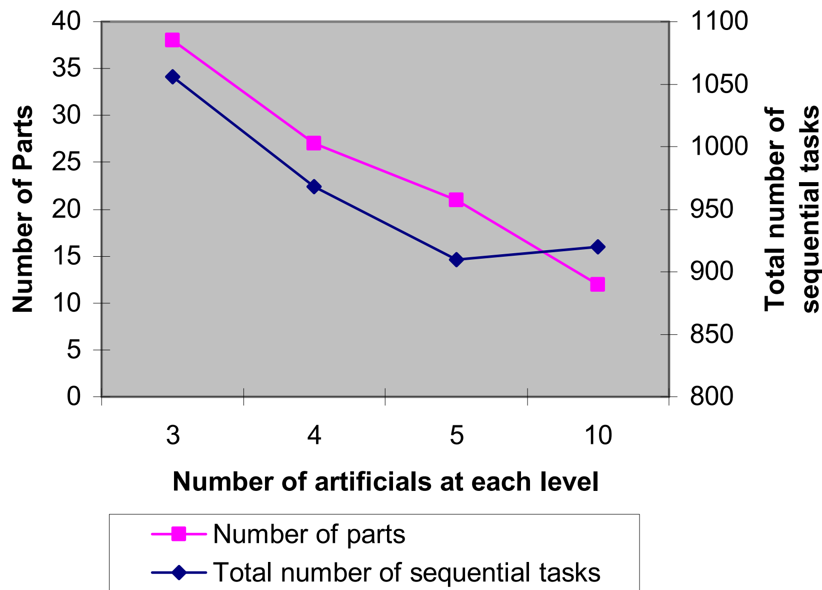


Figure 22 Solvable problem sizes when the number of tasks for each artificial node is 10

From the results displayed in the figures, we make the following observations;

- As the number of tasks for each artificial node increases, the solvable size of the problem in terms of the number of parts behave in parallel with the solvable size in terms of the total number of tasks. This shows that the problems in which the parameter *t* is equal to 1 or 2, while the parameter *n* is too big are not realistic. Hence the Figures 20, 21, 22, represent more realistic scenarios
- When the number of artificial nodes for each level in the graph increases the solvable size of the problem decreases. This result is expected since the parameter *a* impacts the number of the states. This means that, when the

number of feasible subassemblies increases, i.e., when the product tends to be strongly connected, it takes more time for the DP approach to solve the problem.

- When the number of tasks for each artificial node in the graph increases the solvable size of the problem decreases. This result is also expected since the parameter t affects the number of computations. This means that, when the number of feasible tasks increases, i.e., when the product tends to be strongly connected, it takes more time to solve the problem.
- As illustrated in Figure 18, the solvable size of the problem varies from 250 to 50 with varying number of artificial nodes. In the rest of the figures, solvable size does not change significantly. In actual problem instances, since the number of tasks for each artificial usually takes values more than one, we can safely state that the solvable size of the problem do not change with varying values of artificial nodes at each level.
- Usually in the real life, number of tasks for each artificial and the number of artificial nodes at each level is related with the parameter n. When the parameter n is in the range of 30 to 60, the parameters a and t are in the range of 3 to 10. From the figures it is seen that, the solvable size of the problem by the DP approach is 30 to 60.

We then allow three to five parallel tasks at some levels, adding up to a total of ten to twenty parallel tasks. We take 20 experiments for each scenario of the previous case to see how many times the previous problem sizes will be solved. The results gave us two different scenarios. For the cases when the number of

tasks for each artificial is not equal to one, the previous problems size can be solved in 90 percent of the sample problems. Due to the difficulty that parallelism brings, the 10 percent of the problems exceeds the strength of the DP approach. For the case when the number of tasks for each artificial is equal to one, none of the twenty sample problems can be solved. Hence, we reduced the size of the problem gradually until 130, in which 90 percent of the sample problems can be solved. Note that, we only allow ten to twenty parallel tasks in the graph due to the limited time. Inclusion of hundreds of parallel tasks may yield a different outcome.

Final observation about the parallelism is that when the parallel tasks are allowed at the upper levels of the graph it takes more time to solve the problem by the DP approach as compared to when they are allowed at the lower levels. Hence we define the *difficulty* of a parallel task as the multiplication of the levels of its output subassemblies (artificial nodes). The difficulty of the AOG is defined as the summation of the difficulties of all parallel tasks. We compare the solution times of the 124 sample problems with their difficulty in Figure 23.

Since we do not use any state-eliminating techniques in the DP approach, the power of the approach does not depend on the duration or cycle time data. Hence, no matter what the duration of the tasks or the cycle time is, any solvable AOG problem by the DP formulation is always solvable.

Table 4 The difficulties of 124 sample problem and solution durations

	Diff.	Sol. time		Diff.	Sol. time		Diff.	Sol. time		Diff.	Sol. time
1	514	3	**32**	815	60	**63**	904	444	**94**	987	200
2	568	4	**33**	820	100	**64**	905	110	**95**	988	210
3	617	9	**34**	824	311	**65**	908	192	**96**	992	95
4	645	15	**35**	824	143	**66**	909	146	**97**	994	281
5	648	10	**36**	827	71	**67**	910	221	**98**	995	498
6	649	20	**37**	832	182	**68**	913	378	**99**	996	153
7	653	14	**38**	833	121	**69**	917	234	**100**	1004	206
8	654	24	**39**	835	158	**70**	918	652	**101**	1015	577
9	654	91	**40**	838	134	**71**	919	352	**102**	1015	852
10	670	19	**41**	839	84	**72**	919	172	**103**	1028	790
11	682	81	**42**	840	119	**73**	921	50	**104**	1028	351
12	696	17	**43**	841	91	**74**	922	131	**105**	1029	62
13	702	18	**44**	845	237	**75**	926	818	**106**	1037	173
14	716	75	**45**	846	171	**76**	932	308	**107**	1038	549
15	720	18	**46**	853	85	**77**	934	767	**108**	1038	536
16	725	8	**47**	853	23	**78**	943	590	**109**	1040	598
17	736	16	**48**	858	24	**79**	947	617	**110**	1053	514
18	737	115	**49**	858	39	**80**	951	550	**111**	1083	411
19	744	21	**50**	864	128	**81**	952	428	**112**	1084	296
20	745	171	**51**	866	220	**82**	956	760	**113**	1084	472
21	750	494	**52**	872	129	**83**	956	881	**114**	1094	535
22	751	207	**53**	873	34	**84**	960	727	**115**	1101	915
23	753	271	**54**	875	180	**85**	962	410	**116**	1113	219
24	757	41	**55**	875	169	**86**	967	100	**117**	1123	704
25	759	102	**56**	882	350	**87**	968	445	**118**	1131	472
26	762	67	**57**	887	472	**88**	968	142	**119**	1135	635
27	769	337	**58**	893	213	**89**	974	61	**120**	1141	299
28	770	495	**59**	896	179	**90**	975	467	**121**	1143	515
29	778	9	**60**	896	406	**91**	978	151	**122**	1154	363
30	780	120	**61**	897	255	**92**	978	96	**123**	1171	351
31	800	320	**62**	902	150	**93**	986	667	**124**	1183	753

4.3.2 The IP Formulation

As for the IP case, we used CPLEX (Version 8) to solve the formulations. First thing to note is that CPLEX uses some fathoming techniques to expedite the solution process. Since fathoming the nodes depend on the duration and cycle time data, the solvable size of the problem by CPLEX depends on the data. The solution time for the problems with the same input AOG may vary with different data sets. But the variation is not too much. After obtaining the results, we see that the IP formulation for these types of problems is not a suitable one. For instance, while the DP approach solves up to problem size of 249 in the simplest case, i.e., when the parameter a is three and the parameter t is one; the IP approach can solve up to problem size of 20. It can solve 28 instances out of 50.

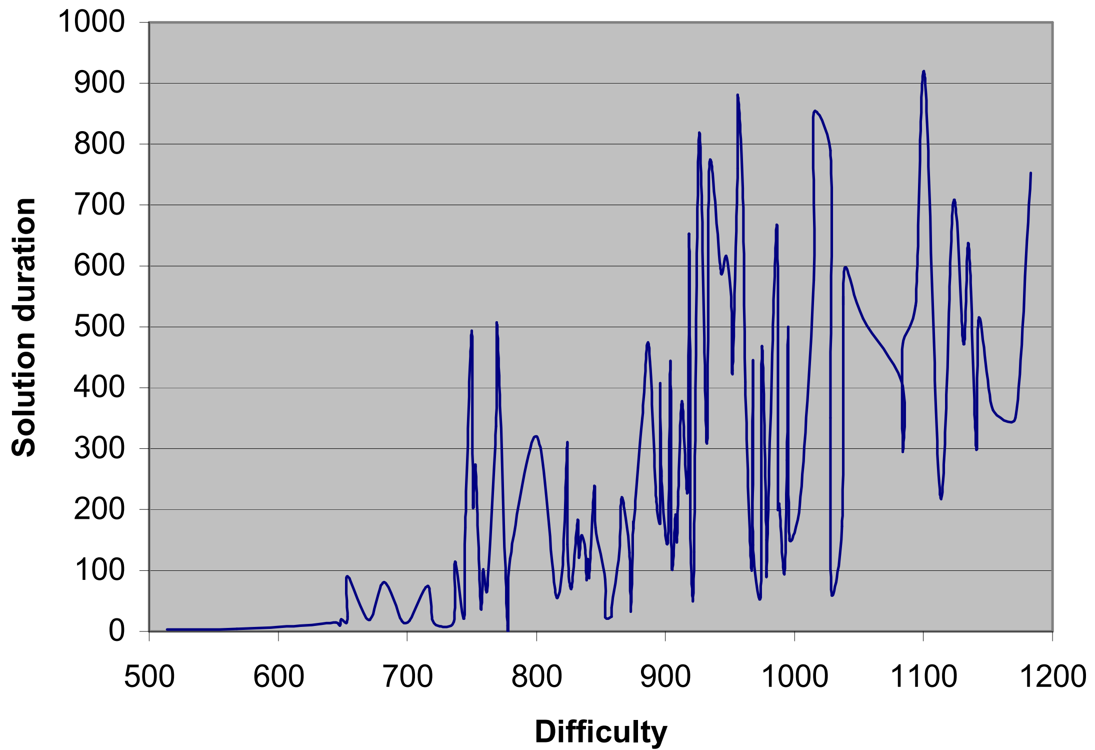


Figure 23 The solution duration vs. difficulty of the problem

The main reason for this big gap between the DP and IP is that the number of variables in the IP formulation increases polynomially with the increase in the parameter *n,* in the order of O(n^2). This can be realized as follows: For the parameter *n*, the number of tasks, which is the upper bound on the index i (*l),* is $a\times[t\times(n-3)+2]$. The upper limit of station index, which is M, is *n*. Furthermore, the solution to the IP formulation increases exponentially with the increase in the number of variables.

On the other hand, the number of stages and the computation time in DP formulation increases in the order of O(n), as discussed above. Since polynomial increase for the DP formulation is not restrictive it can handle large problems compared to the IP method.

As for the other parameters, *t* and *a*, they do not affect the IP formulation as the parameter *n*. For the parameter *n* fixed at 18, IP can solve the problem instances when *a* and *t* are 5 to 10. This is due to the fact that the number of nonzero variables in the IP formulation is $\binom{a \times [t \times (n-3)+2]}{n}$. Hence, the increase in *n* increases the number of nonzero variables more than the other parameters.

Due to exponential increase in the solution time of the IP method, we did not consider improving it, either by modifying the constraints or adding some cuts. As a final word, since the real life problems usually have the parameter *n* greater than twenty, it must be appreciated that in ADLB-AOG problems the DP method is better than the IP method.

4.4 A DP based heuristic

Although the performance of DP is much superior to that of IP, the 'curse of dimensionality' will eventually prevent us solving the problems of realistic sizes. Hence, we develop a DP based heuristic in this section to overcome the limitation of this section. The main characteristic of the heuristic is the circumvention of the rapid growth in the size of the state space.

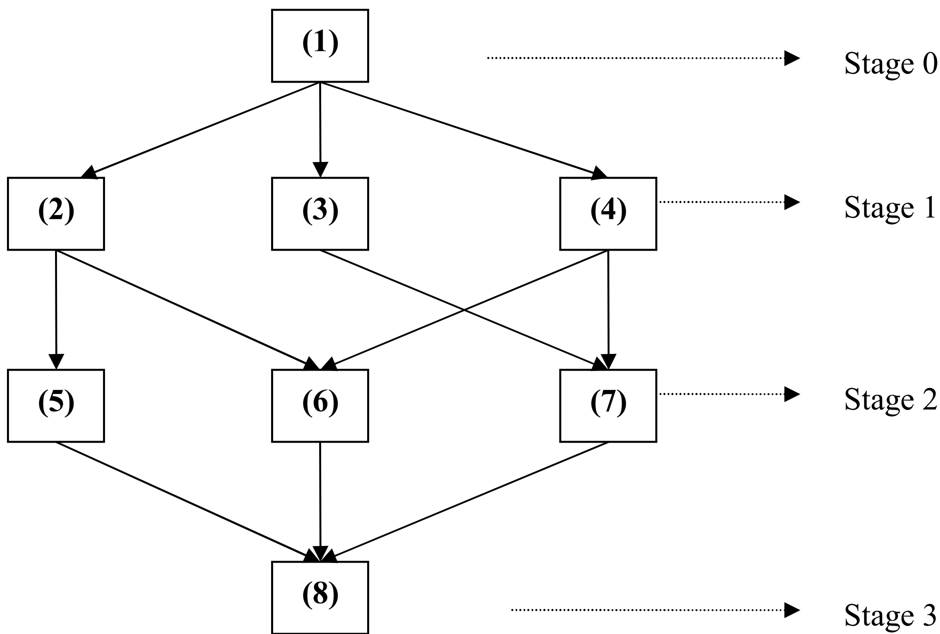


Figure 24 Solution of a sample problem by the DP method

In eliminating the states we chose a simple way. The maximum number of states allowed at any stage is determined by the *reduction parameter*. If the number of states in a stage turns out to be greater than the reduction parameter, we

do not allow those additional states. While eliminating the states in a stage one should be careful of not fathoming the states in the previous stages of the solution. For instance, suppose that the stages and states of the DP solution in the exact method turn out to be as in Figure 24, and we, in the heuristic method, allow the states 2 and 3 to exist in the first stage. In the second stage, if we allow to both of the succeeding states of the state 2 (states 5 and 6), we can not build a solution using the state 3. Hence, we use the following strategy: At any stage k, we make sure that there is at least one succeeding state of each state in stage k-1. To guarantee this, we first allow only one succeeding state for each of the states in the stage k-1 to exist in stage k. If the total number of states is still less than the reduction parameter, we allow the second succeeding states of the states in stage k-1, and so on until either there is no state left or the reduction parameter is reached. Figure 25a is one of the heuristic solutions of the problem in Figure 24 when the reduction parameter is one, and Figure 25b is the solution when it is 2. Note that, when the parameter is three the heuristic and the exact solutions are the same.

Two terms affect the efficiency and the speed of the heuristic. First one is the reduction parameter. As the parameter increases the solution time decreases but the solution quality deteriorates. Second is the distribution of the number of states in each stage of the solution. If the number of states in each state is close to each other, the solution time decreases and quality of the solution deteriorates more. To validate the second argument, we implemented the heuristic to the DP example that we used in Section 4.3 with the parameters of n, a, t being equal to 20, 4 and 2, respectively. The number of states in each stage is equal to the each

other, which is *a*. We generated the durations of each task uniformly between 1 and 20. The cycle time is 30. We generated 100 sample AOG's and obtain the results by both heuristic and the exact method. Average number of stations by the exact method is 6.29, while that result by the heuristic is 1.92. This shows 30 % inferiority of the heuristic. The heuristic is so bad because the reduction parameter is 2 and the number of states in each stage is equal to each other. This means that two states in each stage are eliminated. As for the speed, the exact method can solve up to *n* equal to 74, whereas the heuristic can solve up to 114.

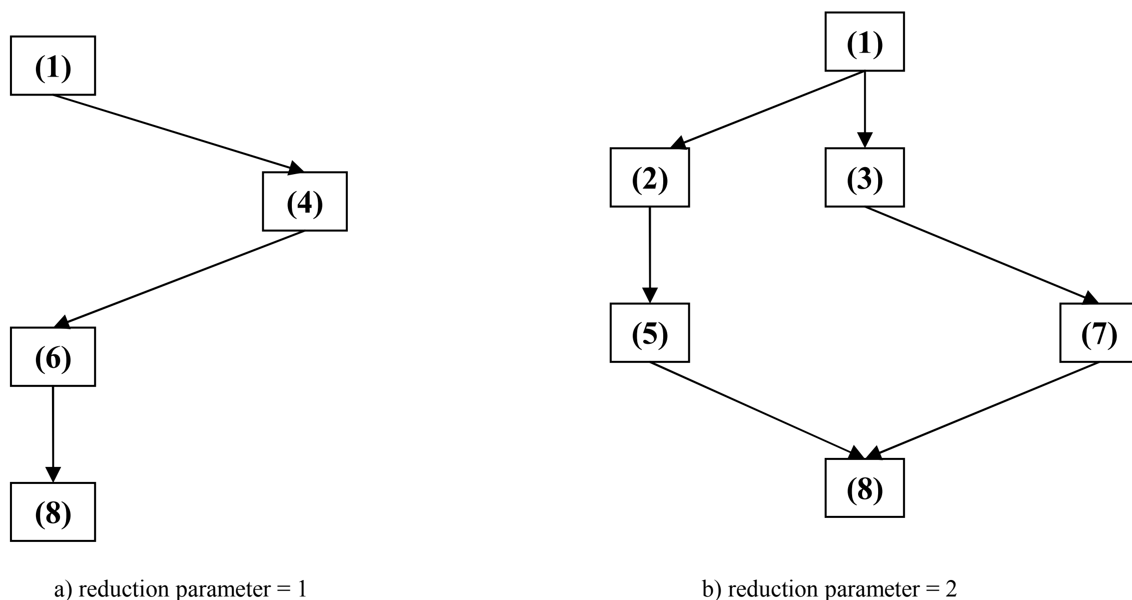


Figure 25 Heuristic solutions for the example problem in Figure 24

To validate the first argument we implemented the heuristic for the example problem in Section 3.4 with varying reduction parameters. The results are reported in Table 5 with the two values of the reduction parameter. Verifying our

argument, the heuristic obtains the exact result 33 times when the reduction parameter is 1, whereas it obtains all the exact results when the parameter is 3.

Table 5 The results of the heuristic solution to the ADLB-AOG problem compared with the exact results of both ADLB-TPD and ADLB-AOG problem.

T	AOG	TPD_1	TPD_2	tp=1	tp=3	T	AOG	TPD_1	TPD_2	tp=1	tp=3	T	AOG	TPD_1	TPD_2	tp=1	tp=3
22	4	5	5	5	4	*45*	2	2	2	2	2	*68*	1	2	2	2	1
23	4	5	5	5	4	*46*	2	2	2	2	2	*69*	1	2	2	2	1
24	4	5	5	5	4	*47*	2	2	2	2	2	*70*	1	2	2	2	1
25	4	5	5	5	4	*48*	2	2	2	2	2	*71*	1	2	2	2	1
26	4	5	5	4	4	*49*	2	2	2	2	2	*72*	1	2	2	2	1
27	4	5	5	4	4	*50*	2	2	2	2	2	*73*	1	2	2	2	1
28	3	5	4	4	3	*51*	2	2	2	2	2	*74*	1	2	2	2	1
29	3	4	4	4	3	*52*	2	2	2	2	2	*75*	1	2	2	2	1
30	3	3	4	4	3	*53*	2	2	2	2	2	*76*	1	2	2	2	1
31	3	3	4	4	3	*54*	2	2	2	2	2	*77*	1	2	2	2	1
32	3	3	4	4	3	*55*	2	2	2	2	2	*78*	1	2	2	2	1
33	3	3	4	3	3	*56*	2	2	2	2	2	*79*	1	2	2	2	1
34	3	3	4	3	3	*57*	2	2	2	2	2	*80*	1	2	2	2	1
35	2	3	4	3	2	*58*	2	2	2	2	2	*81*	1	2	2	2	1
36	2	3	3	3	2	*59*	2	2	2	2	2	*82*	1	2	2	2	1
37	2	3	3	3	2	*60*	2	2	2	2	2	*83*	1	2	2	1	1
38	2	3	3	3	2	*61*	2	2	2	2	2	*84*	1	2	2	1	1
39	2	3	3	3	2	*62*	2	2	2	2	2	*85*	1	2	2	1	1
40	2	3	3	3	2	*63*	2	2	2	2	2	*86*	1	2	1	1	1
41	2	3	3	3	2	*64*	1	2	2	2	1	*87*	1	2	1	1	1
42	2	3	3	3	2	*65*	1	2	2	2	1	*88*	1	1	1	1	1
43	2	3	3	2	2	*66*	1	2	2	2	1	*89*	1	1	1	1	1
44	2	2	2	2	2	*67*	1	2	2	2	1	*90*	1	1	1	1	1

What is more interesting is that, in some cases even the heuristic solutions of the ADLB-AOG problem are better than the exact solutions of the ADLB-TPD problem. Even the worst performance of the heuristic, i.e., when the reduction parameter is 1, beats the exact solution of the ADLB-TPD problem for TPD_2 nine times out of sixty nine instances. Comparing with the TPD1, the heuristic ADLB-AOG outperforms nine times whereas the exact ADLB-TPD outperforms three times.

The JAVA code that implements the heuristic is given in Appendix 9.

Chapter 5

CONCLUSIONS and FUTURE RESEARCH DIRECTIONS

Assembly line balancing problem has been the vital element of the manufacturing practices of the firms. As the environmental issues arise, reverse manufacturing practices, in the form of both remanufacturing and demanufacturing, gained an increasing attention, becoming as vital as the manufacturing. Both of these practices require both disassembly and assembly simultaneously, implying that disassembly and assembly should be considered together. Inspired by this, we investigated the assembly and disassembly line balancing problem

We give and compare the literature in assembly and disassembly line balancing problem. After discussing some properties of the AND/OR Graph, we considered the assembly/disassembly line balancing problem using the AND/OR Graph. The problem consists of assigning a subset of the tasks from the AOG such that the chosen set of tasks completely assembles/disassembles the product and the number of stations required in the line is minimized. We proved and exemplified the theorem of sub-optimality, which states that solving the ADLB-AOG problem gives at least as good solution as the ADLB-TPD problem. In other words, the ADLB-TPD problem solved to optimality may give inferior results than the

ADLB-AOG problem solved by a heuristic. This is because of the fact that the tasks in the AOG are more specifically defined than the tasks of the task precedence diagram. That is, the durations of the task in the AOG is less then or equal to the duration of the corresponding task in TPD. We also give three examples on how to derive the TPD from the AOG.

We constructed dynamic programming and integer programming methods to solve the ADLB-AOG problem. The construction of the DP method is illustrated on sample problem. The 0-1 pure integer programming formulation is implemented on the same problem. We compare the two methods over a set of AOG's defined by some parameters. The merits of DP technique over the IP are remarkable. First is the power to handle the large sized problem. While the IP method can not handle the simple problems, the DP method can bravely cope with the large sized problems up to the problem size of 250. The reason behind this is that the solution time of the DP method grows polynomially by the problem size while that of IP grows exponentially. The second advantage is the rigorousness of the DP method. Since the IP Solvers (CPLEX for instance) uses some fathoming techniques, the solvability of the problem by IP varies with the problem data set. On the other hand, total number of states and stages, and the computation time of the DP method are independent of the data. This in turn implies that the DP method is in its primitive form and can be enhanced.

We implement a DP based heuristic by limiting the number of states in each stage. The heuristic decreases the solution time sacrificing the solution quality. By a numerical example we showed that solving the ADLB-AOG problem

by the heuristic gives better results than the ADLB-TPD problem solved to optimality.

The java code implementing the DP method to a given AOG and a data set is appended. The code that generates the IP formulation of a problem is given. Finally, the code that generates AOG according to the given parameters is included.

Since the research on assembly/disassembly balancing problem is too scarce, the future research directions are numerous. The first one is to modify the AOG. The AOG is fully related with the mechanical properties of the product. A modified graph may include the properties of the shop floor so that the precedence relations include the physical constraint imposed by the shop floor.

We give a procedure to derive TPD's from a given AOG, by combining the AT's. But, we only allowed AND-type precedence relations to exist in the TPD. A study is required to include the OR-type precedence relations as well. Also, the assumption of 'two subassemblies at a time' in the AOG should be relaxed.

The IP and DP methods developed are in crude form. Some improvements on these to methods may further enhance the solution time.

A heuristic that use wise strategies in the solution of the problem should be developed. Also, it is worth to investigate which of the following is better: Solving the ADLB-AOG problem optimally for an AOG, the size of which is reduced by a heuristic, or solving the same problem for the original AOG by a heuristic.

REFERENCES

[1] Alavi Y. 1985, "Graph theory with applications to algorithms and computer science", Wiley, New York.

[2] Altekin FT. 2005, Profit oriented disassembly line balancing, PhD Dissertation, Middle East Technical University, Ankara.

[3] Ayres R, Ferrer G, van Leynseele T. 1997, "Eco-efficiency, asset recovery and remanufacturing", *European Management Journal*, 15 (5): 557-574.

[4] Azzone G, Bertele U, Noci G. 1997, "At least we are creating environmental strategies which work", *Long Range Planning*, 30 (4): 562-571.

[5] Barney J. 1991, "Firm resources and sustained competitive advantage", *Journal of Management* 17 (1): 99-120.

[6] Baybars I. 1986, "A survey of exact algorithms for the simple assembly line balancing problem", Management Science, 32(8): 909-932.

[7] Bodily SE, Gabel HL. 1982, “A new job for businessman: managing the company's environmental resources”, *Sloan Management Review*, Summer 1982, 3-18.

[8] Boneschanscher N. et al. 1988, “Subassembly stability”, In Proc : AAAI-88, Aug 88, 780-785.

[9] Bonifant BC, Arnold MB, Long FJ. 1995, “Gaining competitive advantage through environmental investments”, *Business Horizons*, July/August 1995, 37-47.

[10] Bras B, McIntosh MW. 1999, “Product, process and organizational design for remanufacture – an overview of research”, *Robotics and Computer Integrated Manufacturing*, 15: 167-178.

[11] Brennan L, Gupta SM, Taleb KN. 1994, “Operations Planning Issues in Assembly/Disassembly Environment”, *International Journal of Operations & Production Management*, 14(9): 57-67.

[12] Brown WB, Karagozoglu N. 1998, “Current practices in environmental management”, *Business Horizons*, July/August 1998, 12-18.

[13] Chow W-M, “Assembly line design: methodology and applications”, M. Dekker, New York, 1990.

[14] Corbett CJ, van Wassenhove LN. 1993, “The green fee: Internalizing and operationalizing environmental issues”, *California Management Review*, Fall 1993, 116-135.

[15] De Mello LSH and Sanderson AC. 1990, "And/Or graph representation of assembly plans," *IEEE Transactions on Robotics and Automation*, 6(2): 188-199.

[16] De Mello LSH and Sanderson AC. 1991a, "A correct and complete algorithm for the generation of mechanical assembly sequences," *IEEE Transactions on Robotics and Automation*, 6(2): 228-240.

[17] De Mello LSH and Sanderson AC. 1991b, "Representation of mechanical assembly sequences," *IEEE Transactions on Robotics and Automation*, 7(2): 211-227.

[18] Elkington J. 1994, "Toward the sustainable corporation: win-win-win business strategies for sustainable development", *California Management Review*, Winter 1994, 90-100.

[19] Epstein MJ. 1996, "You have got a great environmental strategy- now what?", *Business Horizons*, Sept/Oct 1996, 53-59.

[20] Erel E. and Sarin S.C. 1998, "A survey of the assembly line balancing procedures", Production Planning & Control, 9(5): 414-434.

[21] Flood M.M. 1956, "The traveling salesman problem", *Operations Research*, 4: 61-75.

[22] Florida R. 1996, "Lean and green: the move to environmentally conscious manufacturing", *California Management Review* 39 (1): 80-105.

[23] Grenchus E, Keene R, and Nobs C. 1997, “Demanufacturing of information technology equipment,” *1997 IEEE Symposium on Electronics and Environment,* 157-160.

[24] Guide Jr VDR. 2000, “Production planning and control for remanufacturing: industry practice and research needs,” *Journal of Operations Management*, 18: 467-483.

[25] Guide Jr VDR, Jayaraman V, Srivastava R, and Benton WC. 2000, “Supply-chain management for recoverable manufacturing systems,” *INTRFACES*, 30(3): 125-142.

[26] Gungor A and Gupta SM. 1999, “Issues in environmentally conscious manufacturing and product recovery: a survey,” *Computers & Industrial Engineering*, 36: 811-853.

[27] Gungor A and Gupta SM. 2001a, “Disassembly sequence plan generation using a branch-and-bound algorithm,” *International Journal of Production Research*, 39(3): 481-509.

[28] Gungor A and Gupta SM. 2001b, “A solution approach to disassembly line balancing problem in the presence of task failures,” *International Journal of Production Research*, 39(7): 1427-1467.

[29] Gungor A and Gupta SM. 2002, “Disassembly line in product recovery,” *International Journal of Production Research*, 40(11): 2569-2589.

[30] Gupta MC. 1994, “Environmental management and its impact on the operation function”, *International Journal of Operations & Production Management*, 15 (8): 34-51.

[31] Gupta SM and Taleb KN. 1994, “Scheduling disassembly,” *International Journal of Production Research*, 8: 1857-1866.

[32] Hart SL. 1997, “Beyond greening: strategies for a sustainable world”, *Harvard Business Review*, Jan/Fab 97, 66-76.

[33] Hartman CL, Stafford ER. 1998, “Crafting environmental value chain strategies through green alliances”, *Business Horizons*, March-April, 62-72.

[34] Held M, Karp R.M. 1962, “A dynamic programming approach to sequencing problems”, *Journal of the Society for Industrial and Applied Mathematics*, 10 (1): 196-210.

[35] Held M, Karp R.M., Shareshian R. 1962, “Assembly line balancing-dynamic programming with precedence constraints”, *International Business Machines Corporation*, 442-459.

[36] Inman RA. 2002, “Implications of environmental management for operations management”, *Production Planning & Control* 13 (1): 47-55.

[37] Johnson MR and Wang MH. 1995, “Planning product disassembly for material recovery opportunities,” *International Journal of Production Research*, 33(11): 3119-3142.

[38] Johnson MR and Wang MH. 1998, "Economical evaluation of disassembly operations for recycling, remanufacturing and reuse," *International Journal of Production Research*, 36(12): 3227-3252.

[39] Klassen RD, Angell LC. 1998, "An international comparison of environmental management in operations: the impact of manufacturing flexibility in the US and Germany", *Journal of Operations Management* 16: 177-194.

[40] Klausner M and Hendrickson CT. 2000, "Reverse logistics strategy for product take-back," *INTERFACES,* 30: 156-165.

[41] Kochan A. 1995, "In search of a disassembly factory", *Assembly Automation*, 15(4): 16-17.

[42] Lambert AJD. 1997, "Optimal disassembly of complex products," *International Journal of Production Research*, 35(9): 2509-2523.

[43] Lambert AJD. 1999, "Linear programming in disassembly/clustering sequence generation," *Computers & Industrial Engineering*, 36: 723-738.

[44] Lambert AJD, 2002. "Determining Optimum Disassembly Sequence in Electronic Equipment," *Computers & Industrial Engineering*, 43: 553-575.

[45] Lambert AJD. 2003, "Disassembly sequencing: a survey", *International Journal of Production Research*, 41 (6): 3721-3759.

[46] Lee D-H, Kang J-G, and Xirouchakis P. 2001, "Disassembly planning and scheduling: review and further research, In Proceedings of the Institution of Mechanical Engineers Part B*, Journal of Engineering Manufacture*, Vol. 215 No. B5 (2001): 695-709.

[47] Lund RT. 1998, The remanufacturing industry: hidden giant, Boston University, Boston, Massachusetts, January 1996.

[48] Maxwell J, Rothenberg S, Briscoe F, Marcus A. 1997, "Green schemes: corporate environmental strategies and their implementation", *California Management Review* 39 (3): 118-134.

[49] Moore KE, Gungor A, and Gupta SM. 2001, "Petri net approach to disassembly process planning for products with complex AND/OR precedence relationships," *European Journal of Operational Research*, 35(1-2): 165-168.

[50] Parkinson HJ, and Thompson G. 2003, "Analysis and taxonomy of remanufacturing industry practice", *Proc. Instn Mech. Engrs*, Vol. 217, Part E, *Journal of Process Mechanical Engineering.*

[51] Penev KD and de Ron AJ. 1996, "Determination of a disassembly strategy," *International Journal of Production Research*, 34(2): 495-506.

[52] Pnueli Y and Zussman E. 1997, "Evaluating the end-of-life value of a product and improving it by redesign," *International Journal of Production Research*, 35(4): 921-942.

[53] Prenting T. and Battaglin R. 1964. "The precedence diagram: A tool for analysis in assembly line balancing", *Journal of Industrial Engineering*, 15 (4): 208-213.

[54] Rai R, Rai V, Tiwari MK, and Allada W. 2002, "Disassembly sequence generation: A petri net based heuristic approach," *International Journal of Production Research*, 40(13): 3183-3198.

[55] Salveson M.E. 1955, "The assembly line balancing problem", *Journal of Industrial Engineering*, 6: 18-25.

[56] Scholl A and Klein R. 1999, "Balancing assembly lines effectively- A computational comparison", European Journal of Operational Research, 114: 50-58.

[57] Spicer A.J., Johnson M.R. 2004, "Third-party demanufacturing as a solution for extended producer responsibility", *Journal of Cleaner Production* 12: 37-45.

[58] Srinivasan H., Shyamsundar N., Gadh, R. 1997, "A framework for virtual disassembly analysis", *Journal of Intelligent Manufacturing*, 8: 277-295.

[59] Thierry M, Salomon M, vans Nunen J, and van Wassenhove L. 1995, "Strategic issues in product recovery management," *California Management Review*, 37 (2): 114-135.

[60] Talbot F.B., Patterson J.H., Gehrlein W.V. 1986, "A comparative evaluation of heuristic line balancing techniques", *Management Science*, 32(4): 430-454.

[61] Tang Y, Zhou MC, Zussman E, Caudill R. 2000, "Disassembly modeling, planning and application: a review" Proceedings of the 2000 IEEE International Conference on Robotics and Automation, April 2000, San Francisco, 2197-2202.

[62] White CD, Masanet E, Rosen CM, Beckman SL. 2003, "Product recovery with some byte: an overview of management challenges and environmental consequences in reverse manufacturing for the computer industry", *Journal of Cleaner Production* 11: 445-458.

[63] Wiendahl HP, Seliger G, Perlewitz H, and Bürkner S. 1999, "A general approach to disassembly planning and control," *Production Planning & Control*, 10(8): 718-726.

[64] Zhang HC, Kuo TC, and Lu H. 1997, "Environmentally conscious design and manufacturing: A state-of-the-art survey," *Journal of Manufacturing Systems*, 16(5): 352-371.

[65] Zussman E. 1995, "Planning of disassembly systems", *Assembly Automation*, 15(4): 20-23.

APPENDIX 1

AND/OR Graph and related concepts in Assembly / Disassembly Process Planning

To introduce the concepts in assembly process planning such as subassembly, assembly task, and assembly sequence, we adopt some of the terminology and assumptions given in De Mello and Sanderson (1990, 1991a and 1991b).

A1.1 Assembly

A *mechanical assembly A* is a composition of interconnected parts forming a stable unit. It can be represented by a simple undirected graph $<P_A, C_A>$ in which $P_A= \{p_1, p_2,\ldots, p_N\}$ is the set of nodes, $C_A=\{c_1, c_2,\ldots, c_l\}$ is the set of edges. $<P_A, C_A>$ is called the *assembly graph of connections (GOC)* (Figure A.1). Each node in P_A corresponds to a part in the subassembly, and there is only one edge in C_A connecting every pair of nodes whose corresponding parts have at least one surface contact. Note that the number of parts in A ($|P_A|$) is N and the number of contacts ($|C_A|$) is l.

A *subassembly* is a nonempty subset of parts that either has only one element or is such that every part has at least one surface contact with at least one another part in the subset. For instance, for the product in Figure 5 of chapter 2,

{cap, stick} is a subassembly but {cap, handle} is not. We will denote the subassembly by its set of parts written in brackets. For instance, {cap} denotes the subassembly composed of cap and {cap, stick, receptacle} denotes the subassembly composed of the parts cap, stick, and receptacle.

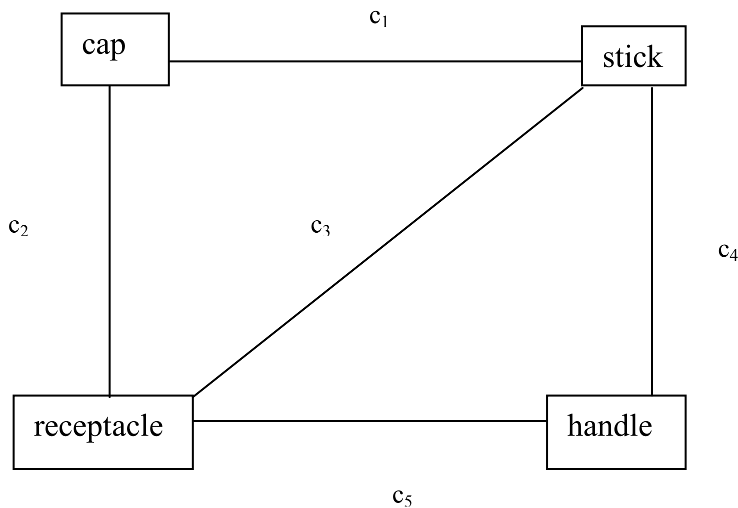


Figure A.1 Graph of connections (GOC) for the product in Figure 5 of Chapter 2

To denote whether or not a subset of parts constitutes a subassembly, we will use the predicate notation *sa(.)*. The argument to this predicate is a subset of parts, and its value is either true or false depending on whether or not that subset corresponds to subassembly. For instance, *sa*(cap, stick) = "T" means that {cap, stick} is a subassembly; while *sa*{cap, handle} = "F" means that {cap, handle} is not a subassembly. The value of this predicate for any subset of parts can be determined from the GOC. If one deletes all the nodes that are not among the argument of the predicate and their corresponding arcs from the GOC, the remaining graph is either connected or not. If it is connected, the predicate is true

and if not, the predicate is false. For instance, to see whether *sa*(cap, stick) is either true or not, we delete the nodes {receptacle} and {handle} from Figure A.1 and corresponding arcs c_2, c_3, c_4, c_5. The remaining graph consisting of {cap} and {stick} is connected with the arc c_1. So, *sa*(cap, stick) = "T".

According to the discussion above, one can claim that {cap, stick, handle} is also a subassembly since its sa(.) is true. A closer examination reveals that it cannot be a subassembly since the parts {cap}, {stick} and {handle} does not constitute a stable unit. A subassembly is said to be *stable* if parts maintain their relative positions and do not break spontaneously. Hence, a subassembly should also satisfy stability predicate *st(.)* that determines whether or not a subassembly described by its set of parts is stable. The determination of *st(.)* is addressed elsewhere (Boneschanscer 1988). A subassembly is said to be *feasible* if both of the *sa* and *st* are true.

When more than one part constitutes a subassembly, their relative positions are assumed to be unique (Assumption 1). For instance, in Figure A.1 {cap, stick} is a subassembly in which the stick has a contact with the open side of the cap and inserted into it slightly. It cannot represent a subassembly in which the stick has a contact with the closed side of the cap.

A1.2 Assembly Task

An assembly task τ takes two subassemblies and joins them. The subassemblies to be joined are called the input subassemblies and the resulting subassembly obtained after the task is applied is called the output subassembly. If

the task is associated with a disassembly process, labeling of subassemblies as input and output is reversed.

Assembly/disassembly tasks have three properties: First property is related with the input and output subassemblies. Given the input subassemblies ϕ and θ we say that joining them is an assembly task if the output, $\{P_\phi, P_\theta\}$, is also a subassembly (i.e., $sa(P_\phi, P_\theta)$ is true). For instance, joining {receptacle} and {handle} is an assembly task, while joining {cap} and {handle} is not. This property can be reversed for the disassembly task. Secondly, an assembly task should be *geometrically feasible.* There should be a collision-free path to join the two subassemblies. For instance, in Figure A.3 joining {cap, receptacle, handle} with {stick} is not an assembly tasks since joining them is not geometrically feasible, i.e., inserting {stick} into {cap, receptacle, handle} is impossible. We use the geometrical-feasibility predicate gf (.) to denote whether or not an assembly task is geometrically feasible. Thirdly, an assembly task should be *mechanically feasible.* It should be feasible to establish the attachments that act on the contacts between the two subassemblies. In our example all disassembly operations are mechanically feasible. We use the mechanical-feasibility predicate mf (.) to denote whether or not an assembly task is geometrically feasible. A task is said to be feasible if it is both geometrically and mechanically feasible.

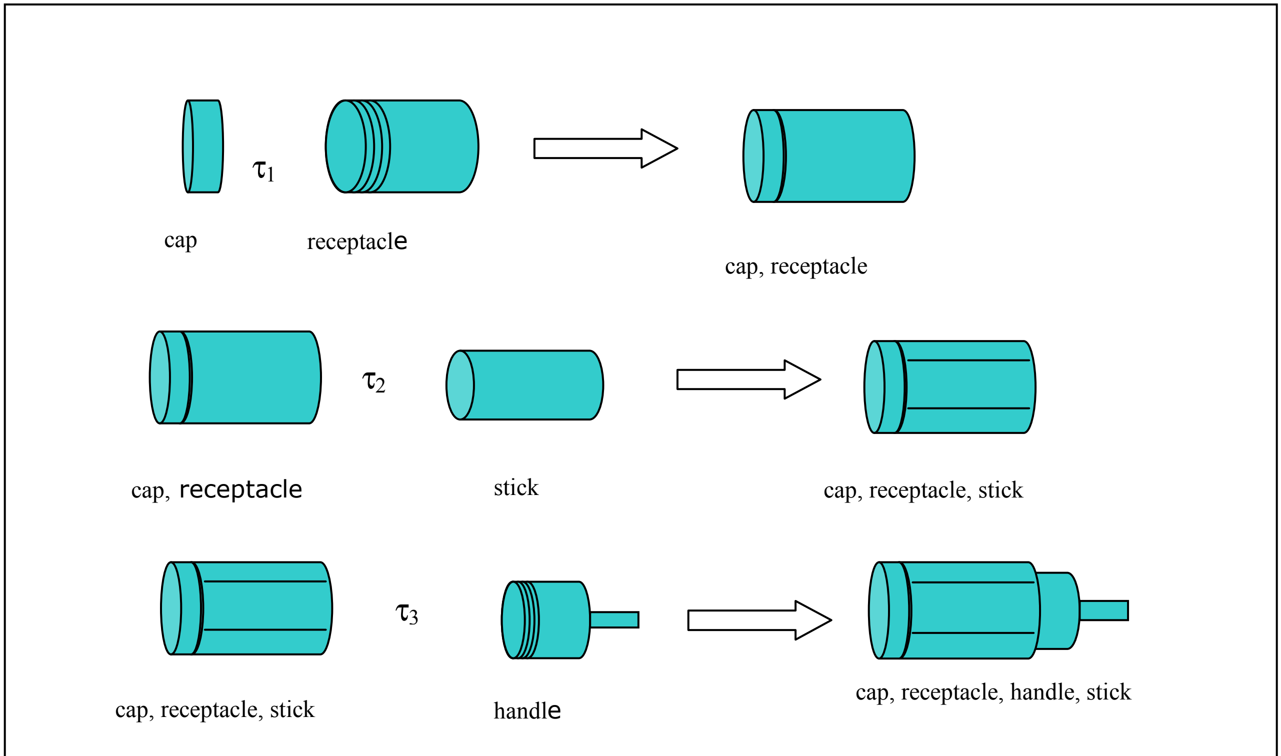


Figure A.2 A feasible assembly sequence (τ_1, τ_2, τ_3)

Assembly/disassembly tasks are assumed have two additional properties: First, whenever two parts or subassemblies are joined all contacts or edges between them should be established (Assumption 2). For instance, when {stick} and {cap, receptacle} need to be assembled, both contacts c1 and c3 should be established. Second, exactly two subassemblies are joined by each assembly task (Assumption 3). In cases where this assumption does not hold, i.e., when more than two parts can be assembled, we can model the situation as sequential assembly operations that assemble those parts two at a time. In practice, one can assemble the three subassemblies {cap, receptacle}, {stick}, {handle} by one

assembly operation. However this assumption implies that this operation is performed by two sequential tasks: First, assemble {cap, receptacle} with {stick}, then assemble {cap, receptacle, stick} with {handle}.

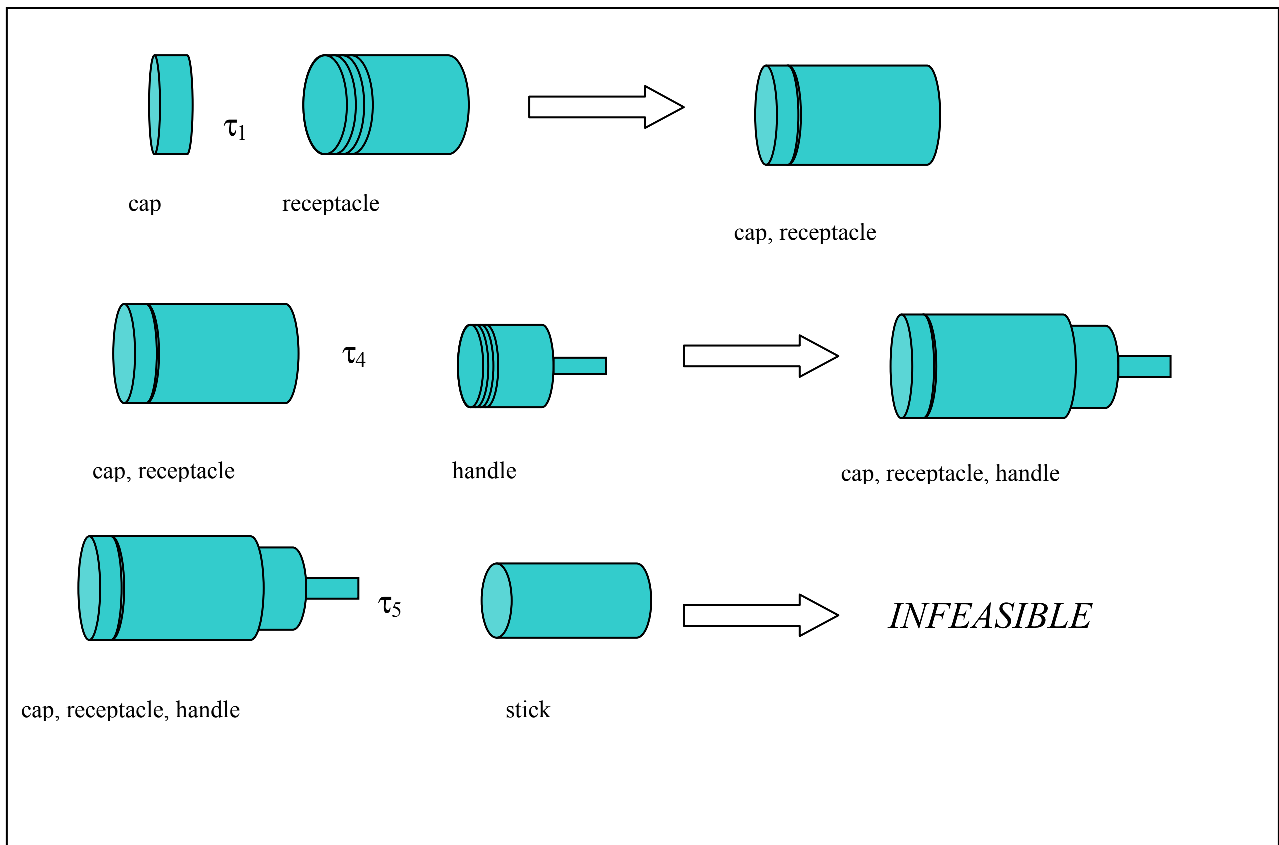


Figure A.3 An infeasible assembly sequence (τ_1, τ_4, τ_5)

We denote an assembly task by the set of input subassemblies. If the task τ_1 joins the {cap} with {receptacle} we denote it by τ_1({cap}, {receptacle}) = {cap, receptacle}. We denote a disassembly task by the input subassembly and the contacts that are taken apart after the disassembly process. If the task ψ_1 disassembles the {cap, receptacle} and obtains {cap} and {receptacle} we denote it by ψ_1({cap, receptacle}, c_2) = {cap}, {receptacle}.

In graph theory, a *cut-set* of a connected graph is the subset of edges in the graph such that the graph becomes more than one piece when those edges are deleted from the graph (Alavi 1985). An assembly/disassembly task corresponds to one of the cut-sets of the GOC that disintegrates it into two pieces. Hence, to find all possible assembly/disassembly tasks that are applied to a subassembly, one should search through all the cut-sets of the subassembly graph of connections that disintegrates the graph into two pieces.

A1.3 Assembly Sequence

Given an assembly A with $|P_A| = N$, an ordered set of N-1 assembly tasks τ_1, τ_2, …. ,τ_{N-1} is an *assembly sequence* if:

- There are no two tasks that have common input subassembly,
- the output subassembly of the last task is the whole subassembly.
- both of the input subassemblies to any task τ_i is either one-part subassembly or the output subassembly of a task that precedes τ_i.

An example assembly sequence (Figure A.2) is τ_1, τ_2, τ_3, where τ_1 is joining the cap to receptacle, τ_2 is joining the stick to the output subassembly of τ_1 and τ_3

is joining the handle to the subassembly made up by τ_2. The sequence τ_1, τ_4, τ_5 (Figure A.3) is not an assembly sequence, where τ_1 is joining the cap to receptacle, τ_4 is joining the handle to the output subassembly of τ_1 and τ_5 is joining the stick to the subassembly made up by τ_4. The reason to that is the infeasibility of τ_5.

A1.4 Discussion on AOG

Although AND/OR graph is a one of the complete representation scheme of assembly/disassembly sequence representation, it has two drawbacks: First one is related to its applicability. In real life, when a disassembly task is applied, the product may yield into more than two subassemblies, whereas in the AND/OR graph a disassembly task results in exactly two subassemblies (See the Assumption 3 in Section A1.2). This restriction can be handled as follows: Suppose that a disassembly task disintegrates a subassembly into more than two pieces. We assume that these subassemblies are sequentially disintegrated from the input node two at a time by two or more tasks. For instance, in Figure 6 of Chapter 3, when receptacle is to be disintegrated from the whole product, the product falls apart into three parts: {cap}, {stick} and {receptacle, handle}. In the AOG representation this is handled by applying two disassembly tasks: Either tasks ψ_{i1} and ψ_{i2} or tasks ψ_{j1} and ψ_{j2}. First, task ψ_{i1} is applied resulting in {Cap, stick} and {receptacle, handle} pairs, then task ψ_{i2} is applied resulting in {Cap}, {stick} and {receptacle, handle}. Similarly first, task ψ_{j1} is applied resulting in {Cap} and {stick, receptacle, handle} nodes, than task ψ_{j2} is applied resulting in {Cap}, {stick} and {receptacle, handle} pairs. This removes the restriction of Assumption

3, but the duration and cost of the new tasks should be properly determined from those of the old task.

We should also point out that although this assumption simplifies the construction of AOG (De Mello and Sanderson 1991a.) by cutting down all possible cut-sets to the ones that disintegrates the graph into two, it increases the size of AOG. For instance, as seen in Figure 6 if the assumption were not made, there would not be subassemblies {C, S, R}, {S, R, H}, {C, S}, {S. R}, {S, H} in the AOG. When cap is disassembled, the product would fall apart into three parts {{C}, {S}, {R, H}} or when handle is disassembled, the product would fall apart into three parts {{C, R}, {S}, {H}}. Thus, the subassemblies that include {stick}, other than the last node, would disappear from the graph.

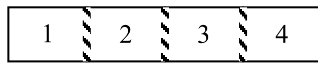
The second drawback is related to storage and computational requirements, i.e., size of AOG. We will show how many nodes, tasks and AT/DT's exist in an AOG, although they are interrelated. Since a node in the AOG shows a feasible subassembly which can be obtained by disassembling an input subassembly, we require two conditions for it to exist in the AOG: All connections between the parts of it should exist and the corresponding task should be feasible, which are related to the number of connections between the parts of the input subassembly and geometry of the parts, respectively. Hence, although every part in a product may be connected to each other, every combination of the parts in the product need not represent a subassembly. There may be some feasible subassemblies of that product that does not exist in the AOG. For instance, there are 14 subassemblies (including the four 1-part subassemblies) in the AOG of the strongly connected product in Figure A.5c instead of 15 (2^4-1=15). This is because

of the nonexistence of the subassembly {2,3,4} in the graph. Although the subassembly {2,3,4} is feasible, it is not included in the graph because of the geometrical infeasibility of the disassembly task that disintegrates {1,2,3,4} into {2,3,4} and {1}. As a result of the above discussion, number of nodes in the AOG depends on the number of parts in the product, on the number of connections of the parts and on the geometry of the parts.

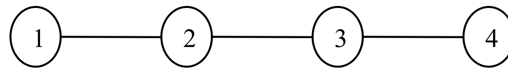
To see the worst case, we will ignore the dependence of the size of AOG on the geometry of the parts of the product. That is, we will assume that any feasible subassembly of the product can exist in the AOG. Two extreme types of products will be considered: Strongly and loosely connected products. In *strongly connected products*, all of the components are connected with each other (Figure A.5b). If the product consists of n components, there are at most $h = \sum_{s=1}^{n} \binom{n}{s} = 2^{n-1}$ feasible subassemblies of the product (i.e., of which subassembly predicate is true). This constitutes the maximum number of possible nodes in the AND/OR graph for an n-part product. The number of nodes in the AOG is, even, less then h, due to infeasibility of some tasks as discussed above. As a future research, one may prove the impossibility of strong connection between the parts of a subassembly when the number of parts exceeds a certain number.

The graph of connections of the *loosely connected product* is a tree such that a node is connected with at most two other nodes (Figure A4b). Thus, the number of feasible subassemblies (h) is at most $h = \sum_{s=1}^{n} (n - s + 1) = \frac{n \times (n+1)}{2} = \binom{n}{2}$.

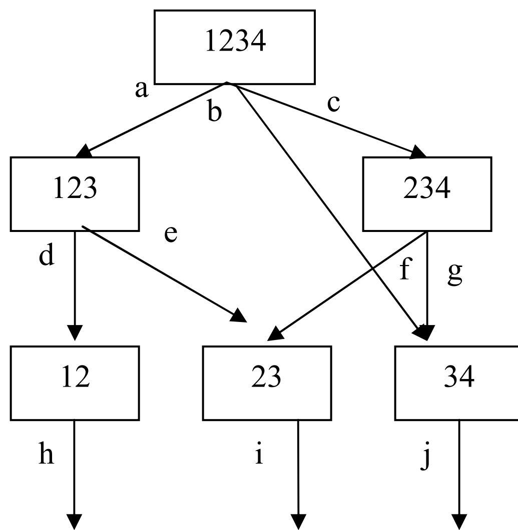
Total number of possible tasks in AOG of a strongly connected product (l) is the total of subassemblies multiplied by the number of tasks applied to them. Hence, number of tasks in the AOG of a strongly connected product with n parts is $l = \sum_{s=2}^{n} \binom{n}{s} \left[2^{s-1} - 1 \right] = \frac{3^n - 2^{n+1} + 1}{2}$. This number for the loosely connected product is $l = \sum_{s=1}^{n} (n - s + 1)(s - 1) = \frac{(n+1) \times n \times (n-1)}{6} = \binom{n+1}{3}$



a) product

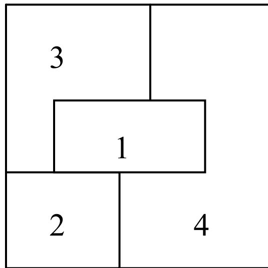


b) graph of connections

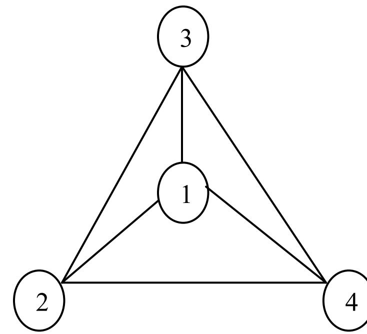


c) AOG

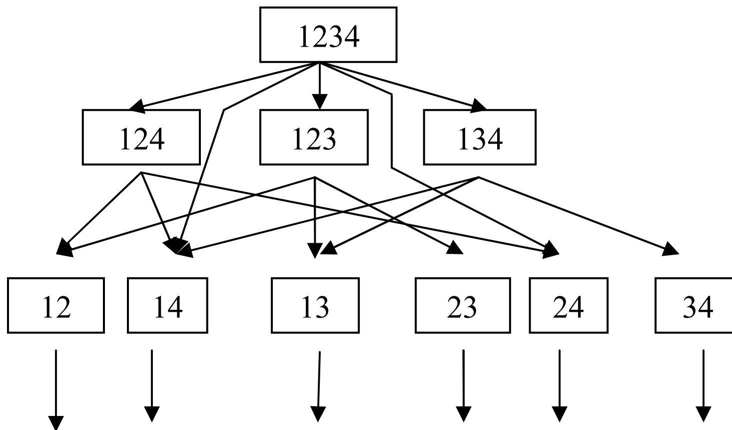
Figure A.4 A loosely connected product, its graph of connections and AOG



a) product



b) graph of connections



c) AND/OR graph of strongly connected product with 20 tasks

Figure A.5 A strongly-connected product, its graph of connections and AOG

Total number feasible assembly sequences in AOG depends on the same variables as well: It can be found by a recursive formula. Let N_{iS} be the total number of assembly sequences for a strongly connected subassembly with the number of parts equal to i.

$N_{2S} = 1$

$N_{3S} = \binom{3}{2} * N_{2S} = 3$

$N_{4S} = \binom{4}{3} * N_{3S} + \frac{1}{2} * \binom{4}{2} * N_{2S} = 15$

$N_{5S} = \binom{5}{4} * N_{4S} + \binom{5}{3} * N_{3S} * N_{2S} = 105$

$$N_{iS} = \frac{1}{2} \sum_{j=1}^{i-1} \binom{i}{j} \times N_{(i-j)S} \times N_{jS} = \prod_{j=2}^{i} (2j-3)$$

where $i \geq 2$ and $N_{1S} = 1$

Let N_{iL} be the total number of assembly sequences for a loosely connected subassembly with the number of parts equal to i.

$N_{2L} = 1$

$N_{3L} = 2 * N_{2L} = 2$

$N_{4L} = 2 * N_{3L} + 1 * (N_{2L})^2 = 5$

$N_{5L} = 2 * N_{4L} + 2 * N_{3L} * N_{2L} = 14$

$$N_{iL} = \sum_{j=1}^{i-1} N_{(i-j)L} \times N_{jL}$$

where i > 2 and $N_{1S} = 1$.

Strongly connected and loosely connected products are two extremes. The typical products are between the two extremes in terms of number of connections.

In the literature there is also a simplified representation of AOG that decreases the number of nodes slightly. For example, Lambert (1999) proposes using normal arcs instead of hyper-arcs that shows the input subassembly (to disassembly task) and only one of the output subassemblies instead of the hyper-arc showing both of the output subassemblies and the input subassembly. In this simplified representation, one does not loose any information since it can be inferred from the complementary nodes in AOG. Also, the subassemblies with a single part can be eliminated as there are no tasks that can be applied to them. To realize how this makes the representation easier, see Figure A.6 as compared to Figure 6 in Chapter 3.

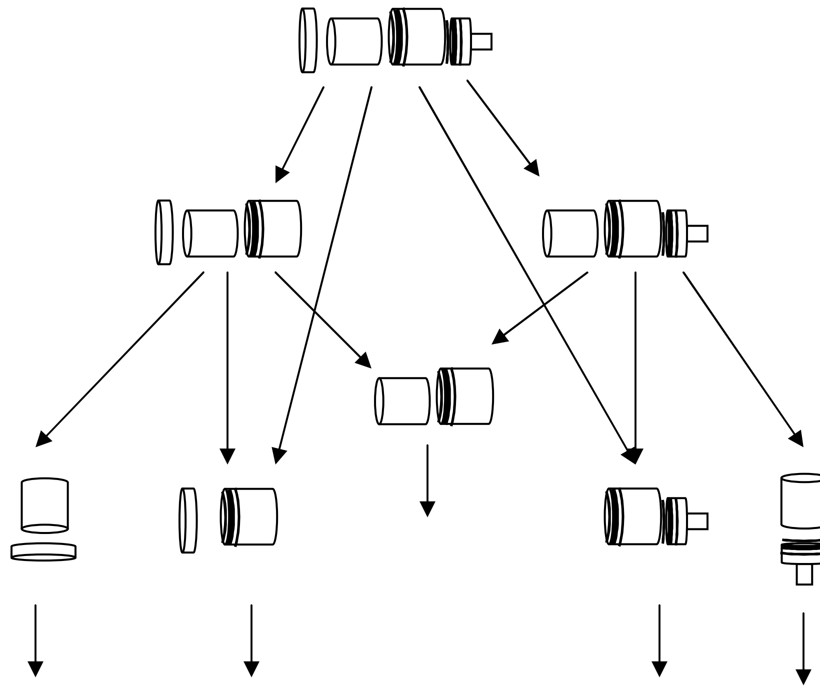
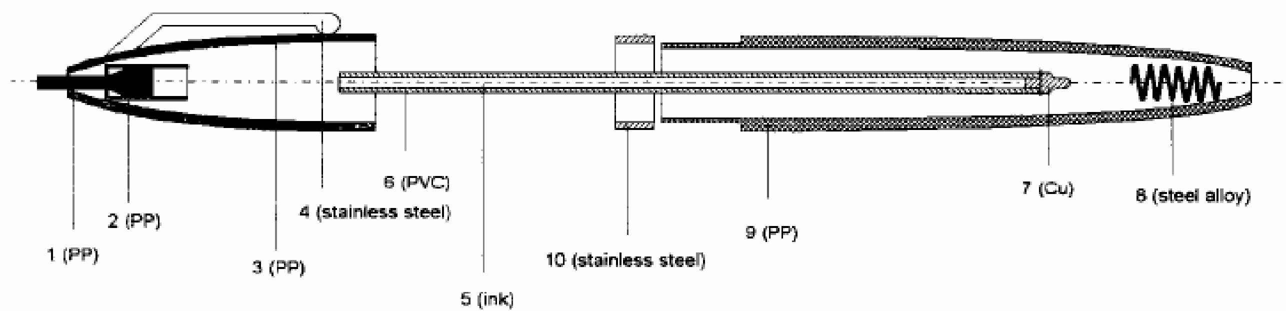


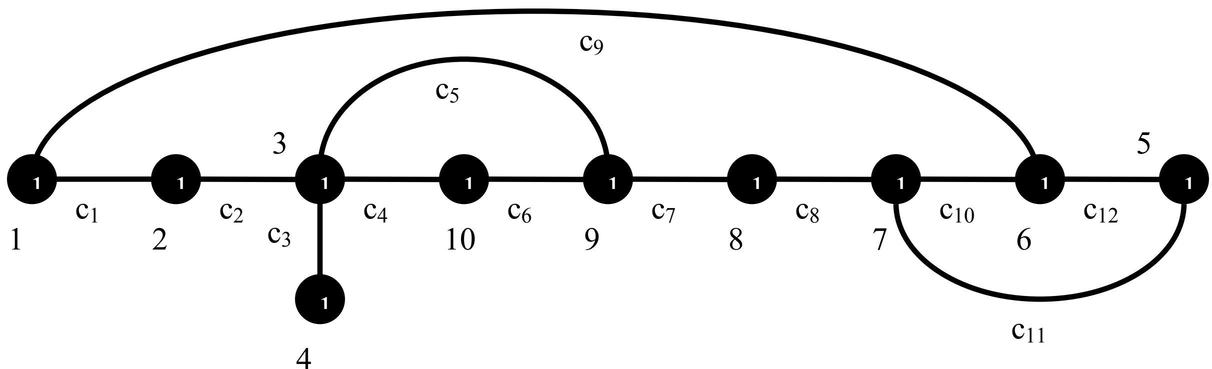
Figure A.6 Proposed AND/OR graph

APPENDIX 2

Figures of the Example 2 in Section 3.3



a) the product



b) its GOC

Figure A.7 An example product and its GOC (Lambert 1999)

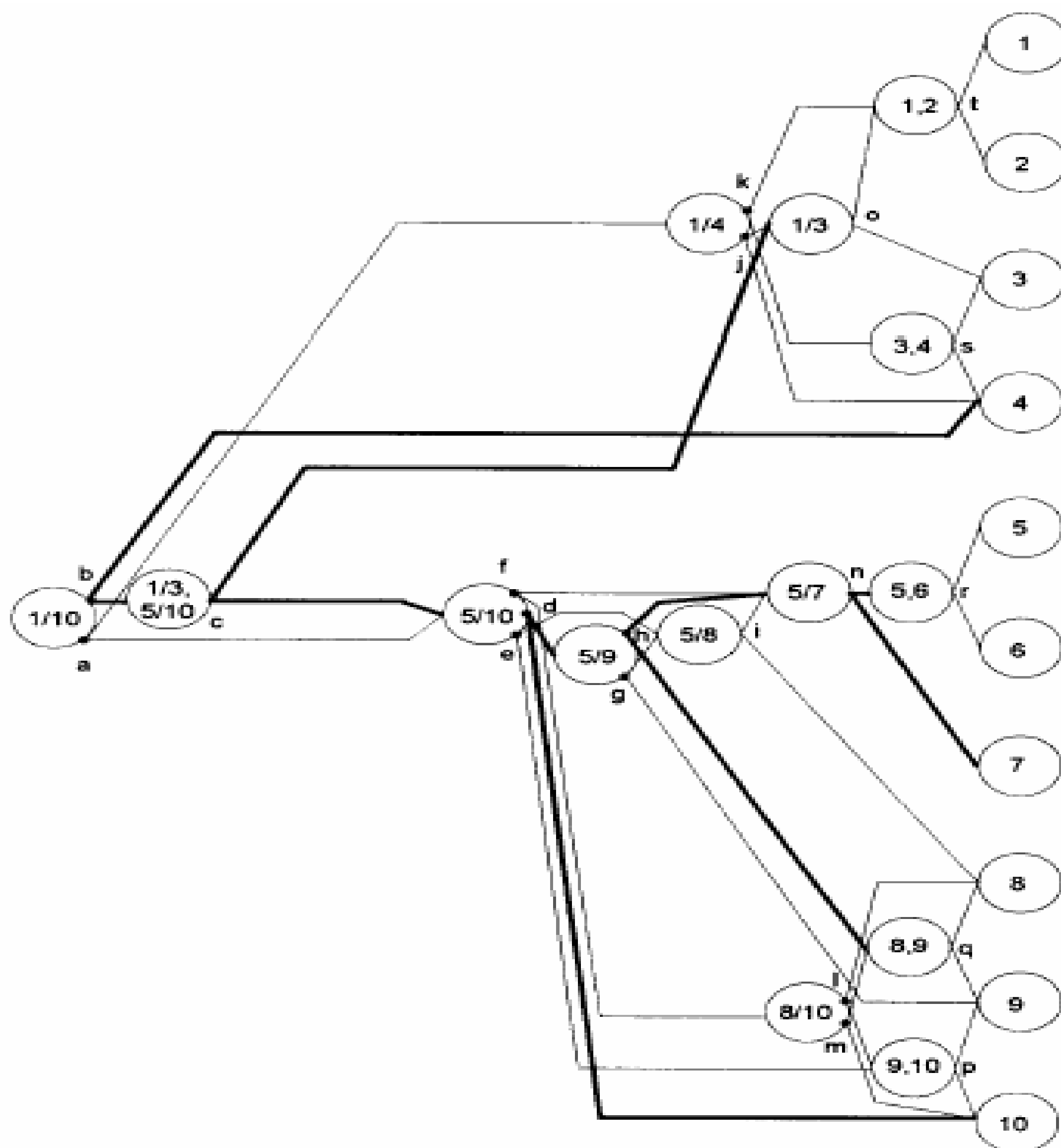
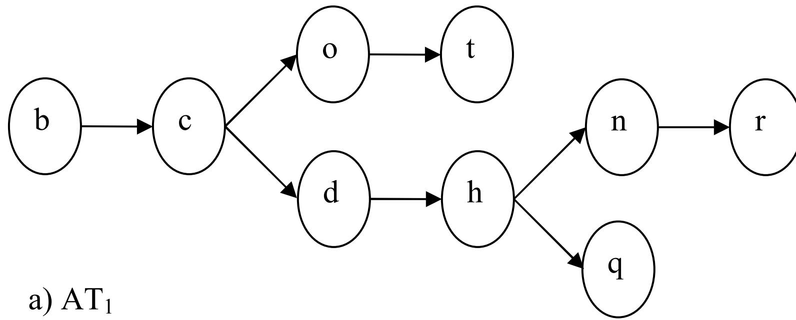
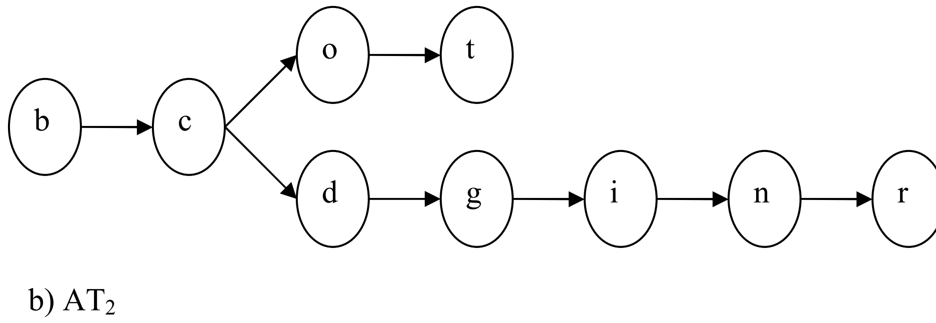


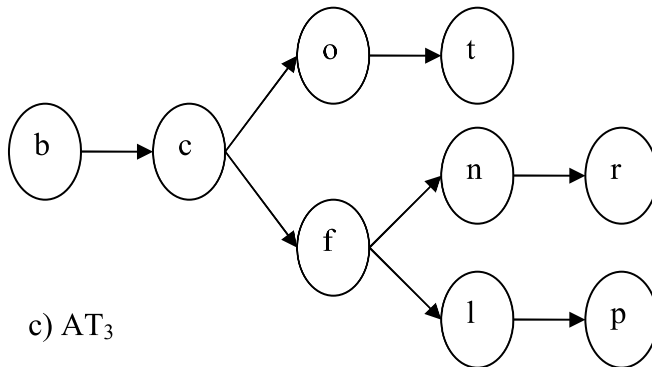
Figure A.8 AOG of the product in Figure A.7



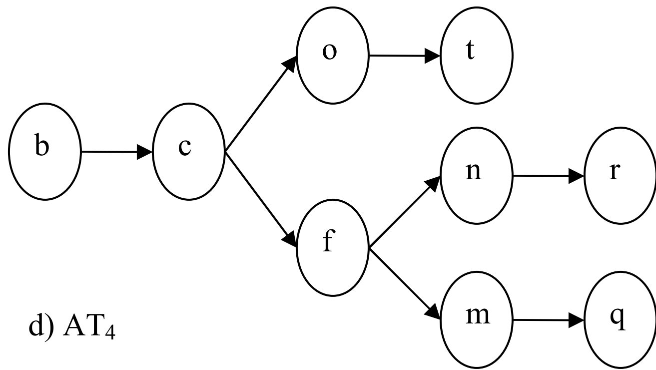
a) AT_1



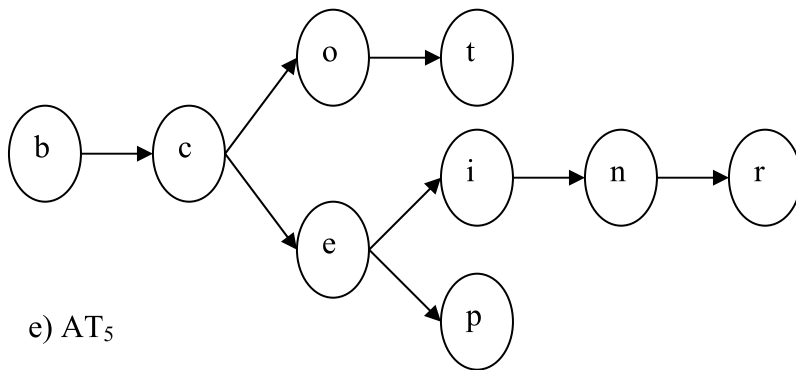
b) AT_2



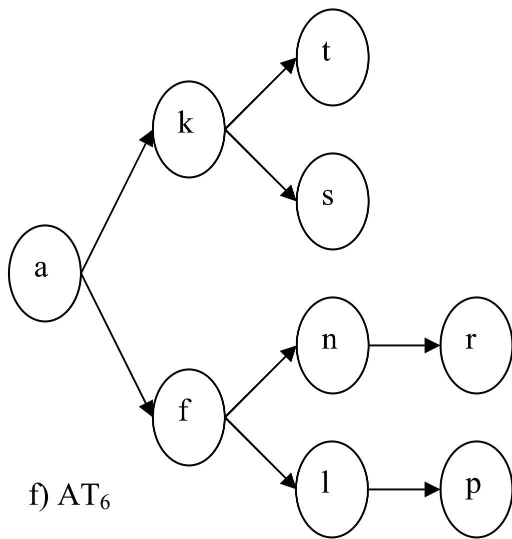
c) AT_3



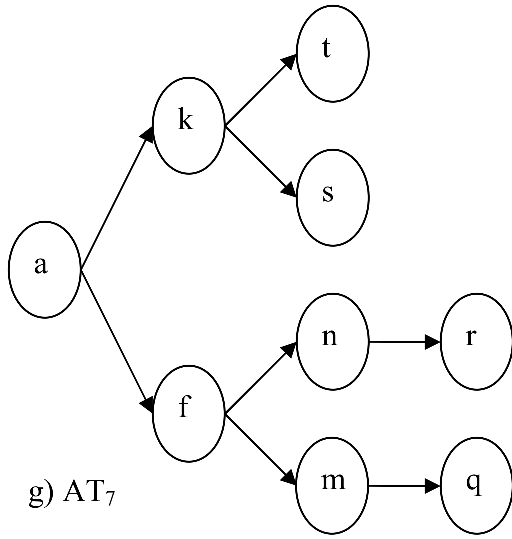
d) AT_4



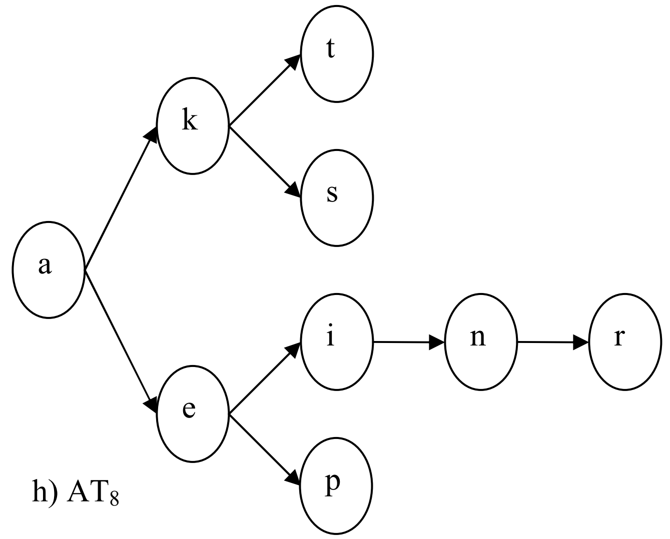
e) AT_5



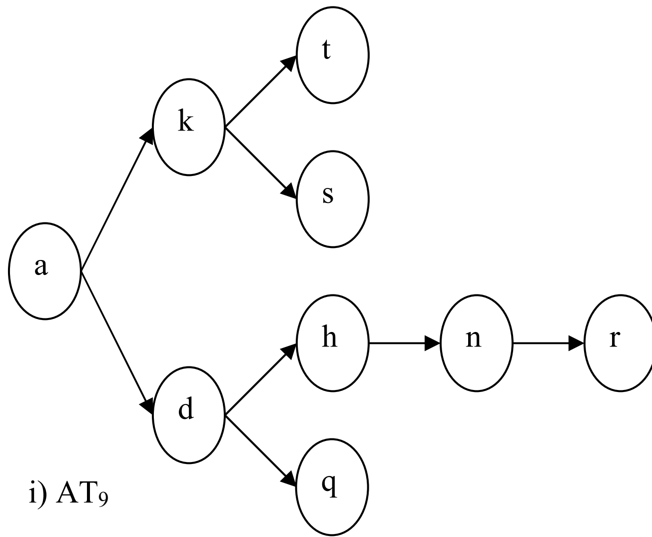
f) AT_6



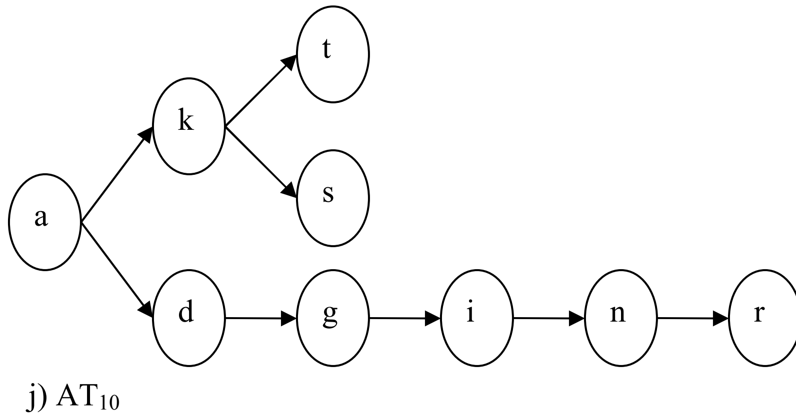
g) AT_7



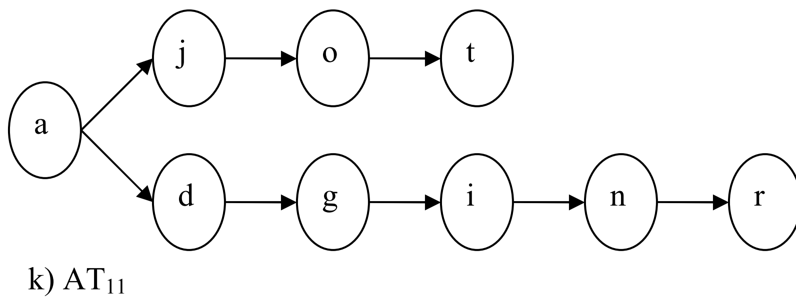
h) AT_8



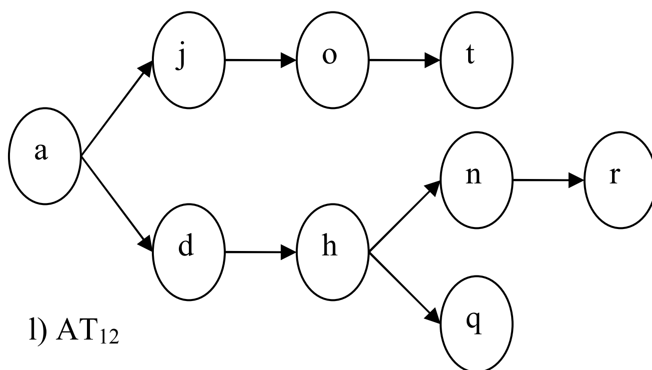
i) AT_9



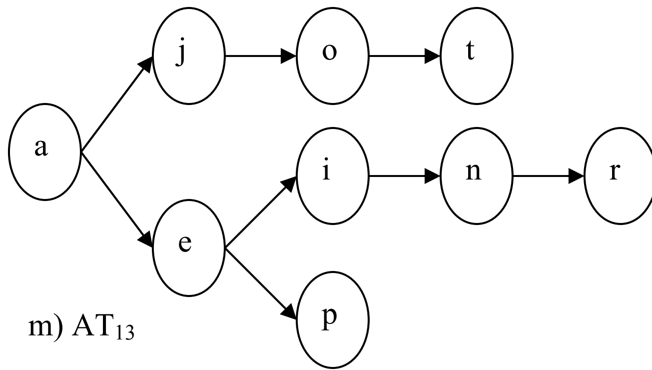
j) AT_{10}

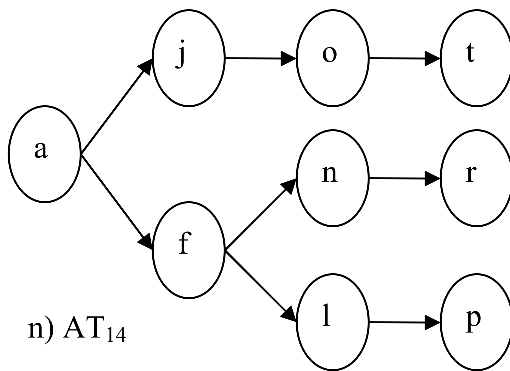


k) AT_{11}



l) AT_{12}





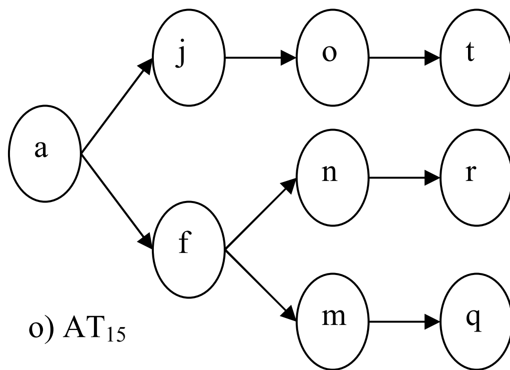
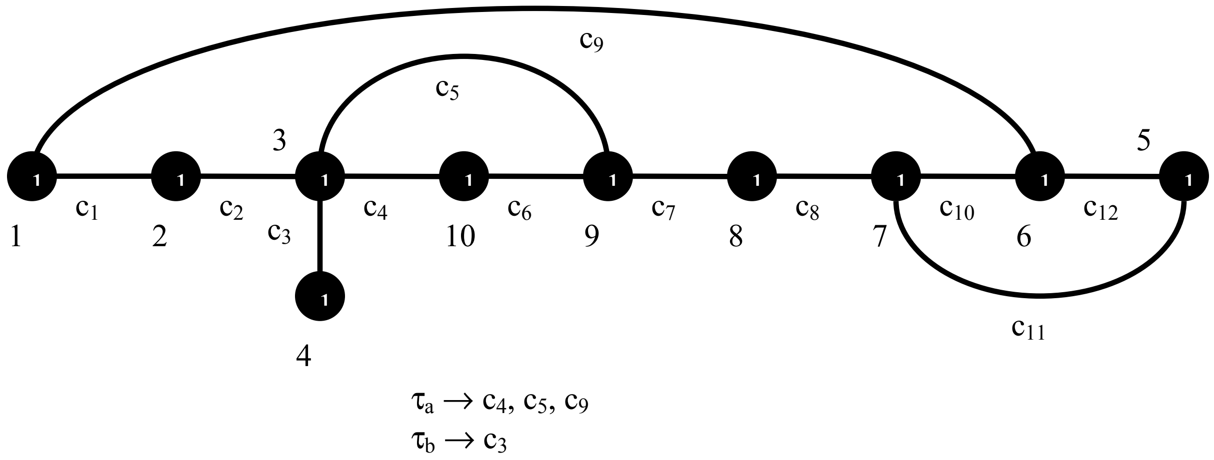
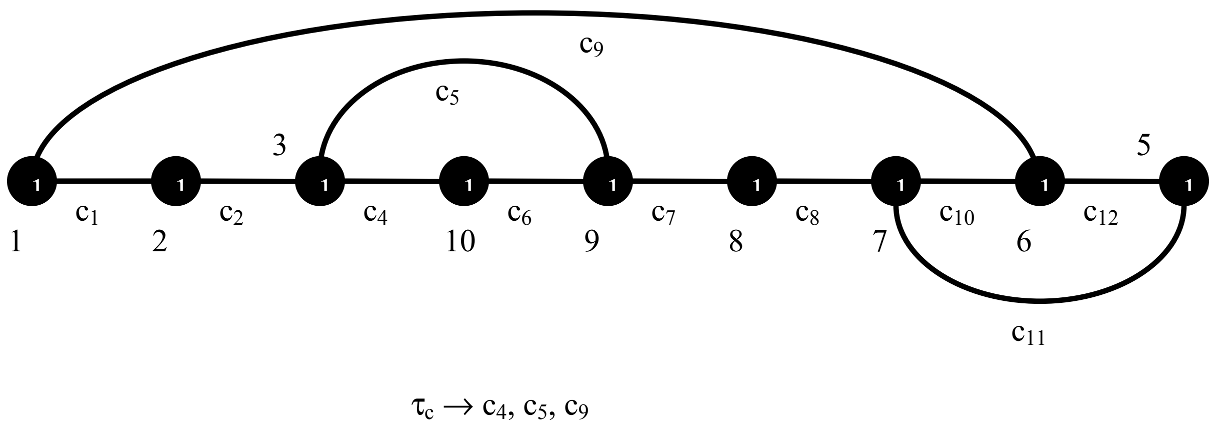
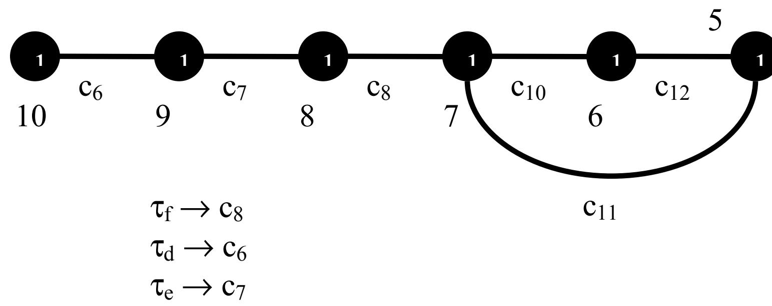
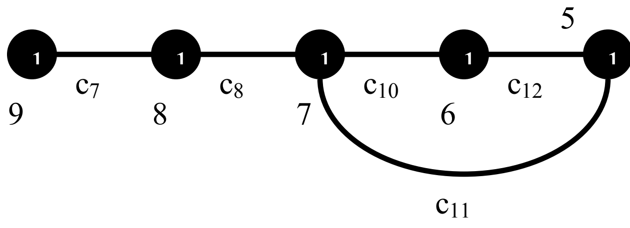


Figure A.9 S_{AT} established from the AOG in Figure A.8

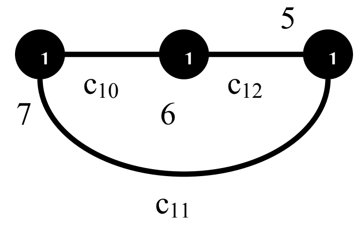

c_9
c_5
3
5
1
2
10
9
8
7
6
4
c_1
c_2
c_3
c_4
c_6
c_7
c_8
c_{10}
c_{12}
c_{11}
$\tau_a \rightarrow c_4, c_5, c_9$
$\tau_b \rightarrow c_3$


c_9
c_5
3
5
1
2
10
9
8
7
6
c_1
c_2
c_4
c_6
c_7
c_8
c_{10}
c_{12}
c_{11}
$\tau_c \rightarrow c_4, c_5, c_9$

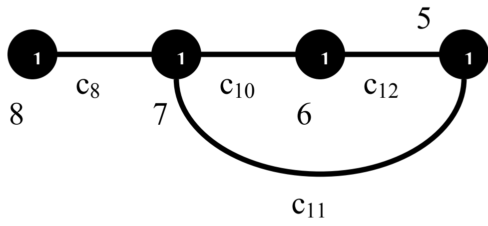

5
c_6
c_7
c_8
c_{10}
c_{12}
10
9
8
7
6
c_{11}
$\tau_f \rightarrow c_8$
$\tau_d \rightarrow c_6$
$\tau_e \rightarrow c_7$



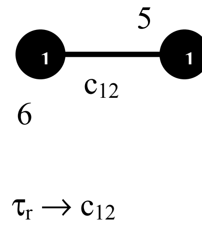
$\tau_h \rightarrow c_8$
$\tau_g \rightarrow c_7$



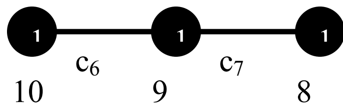
$\tau_n \rightarrow c_{10}, c_{11}$



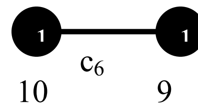
$\tau_i \rightarrow c_8$



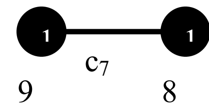
$\tau_r \rightarrow c_{12}$



$\tau_l \rightarrow c_7$
$\tau_m \rightarrow c_6$



$\tau_p \rightarrow c_6$



$\tau_q \rightarrow c_7$

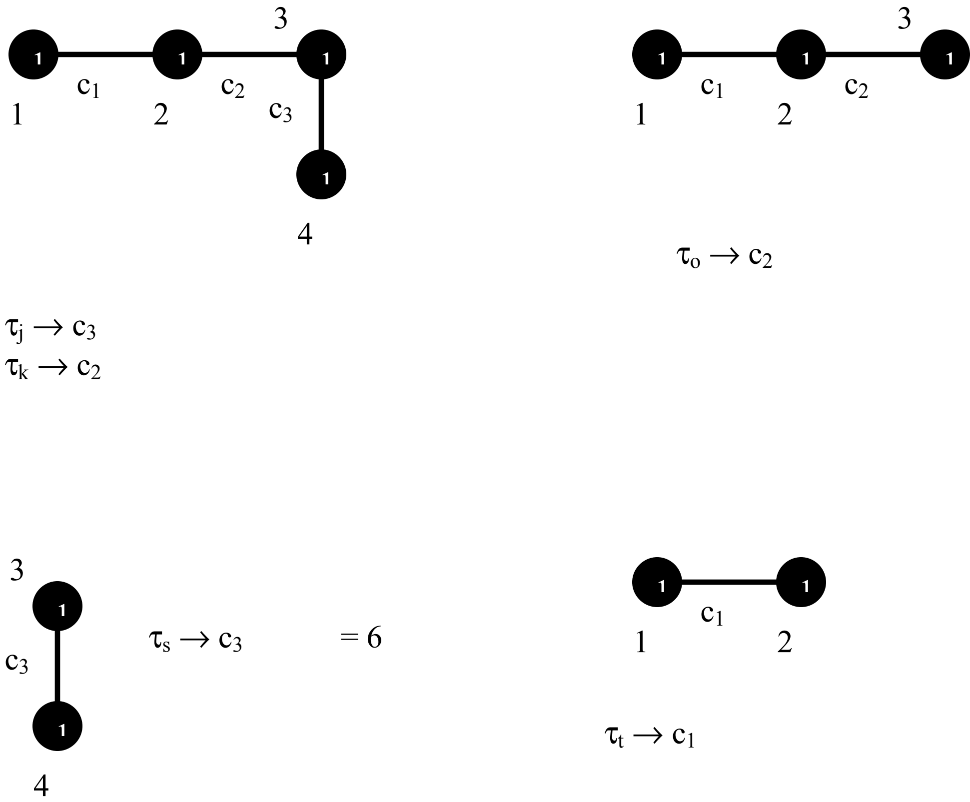
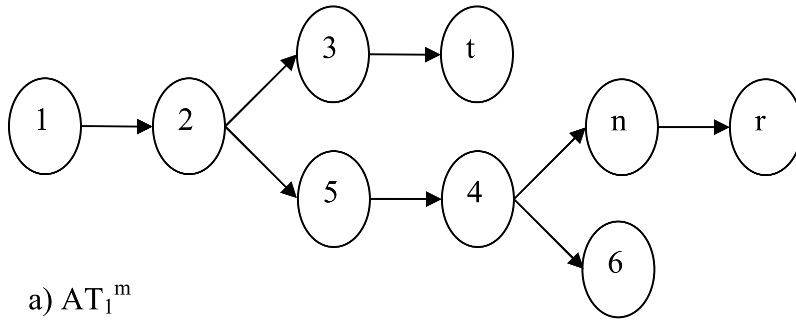
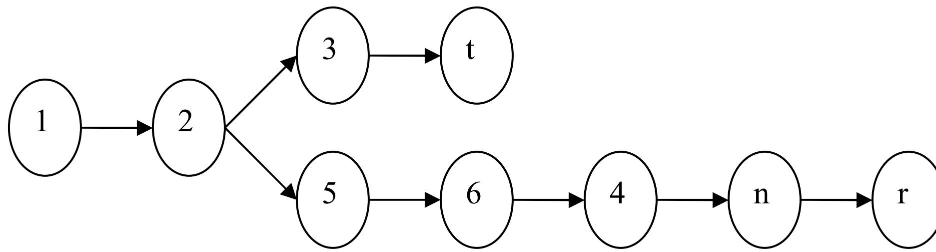


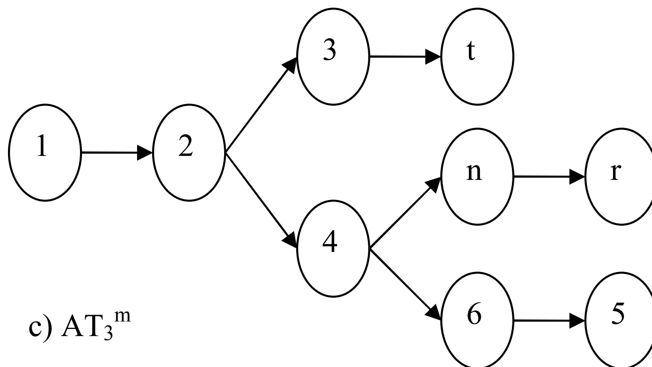
Figure A.10 The subassemblies of AOG in Figure A.8, their GOC and corresponding tasks



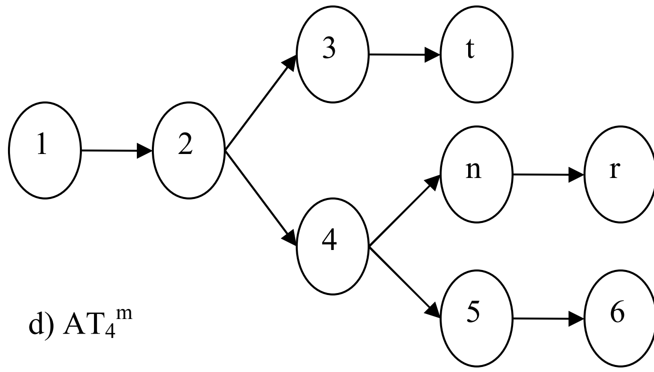
a) AT_1^m



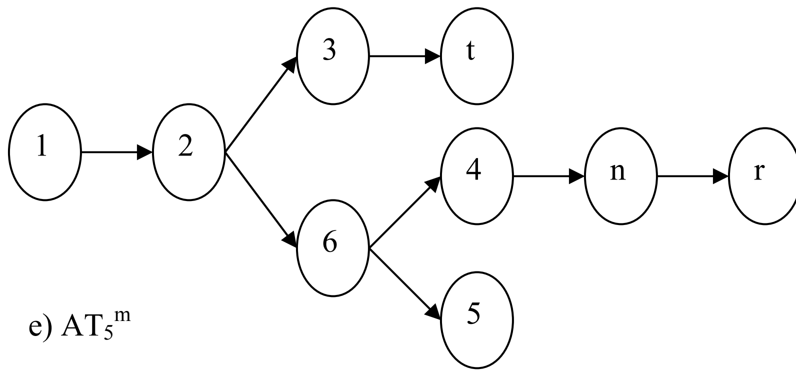
b) AT_2^m



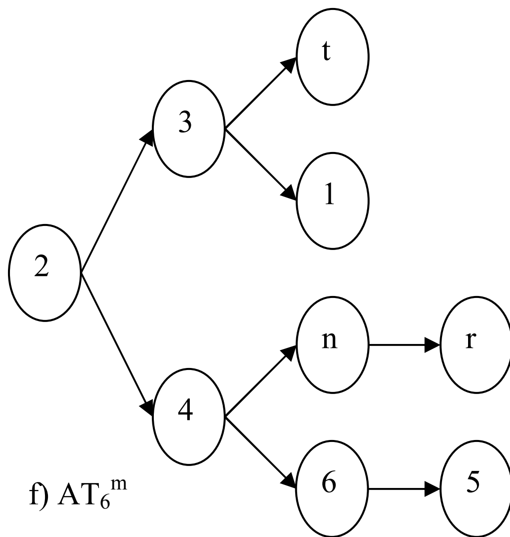
c) AT_3^m



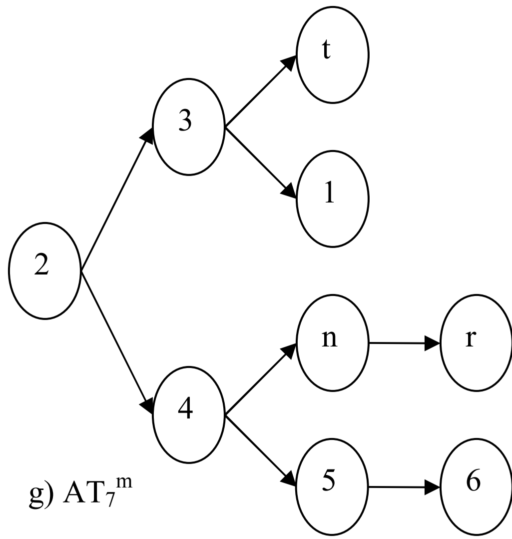
d) AT_4^m



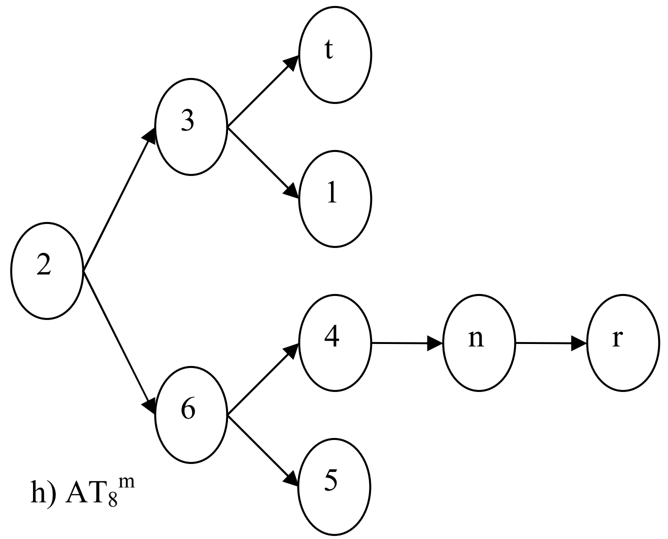
e) AT_5^m



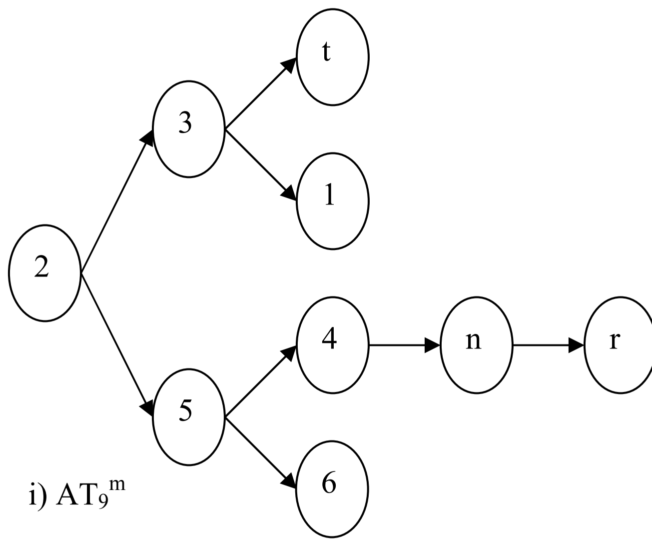
f) AT_6^m



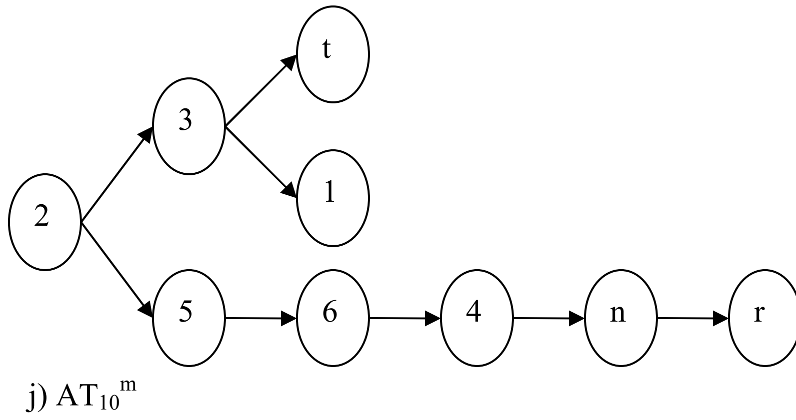
g) AT_7^m



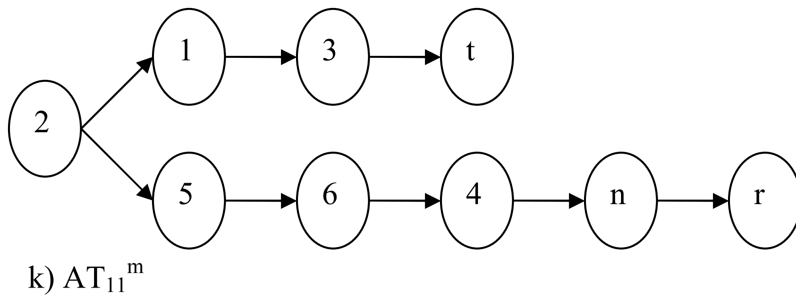
h) AT_8^m



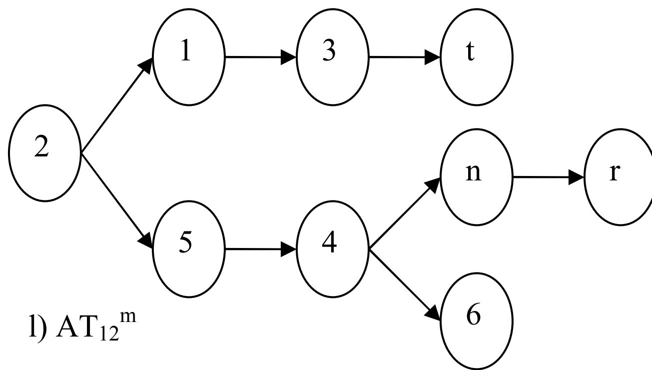
i) AT_9^m



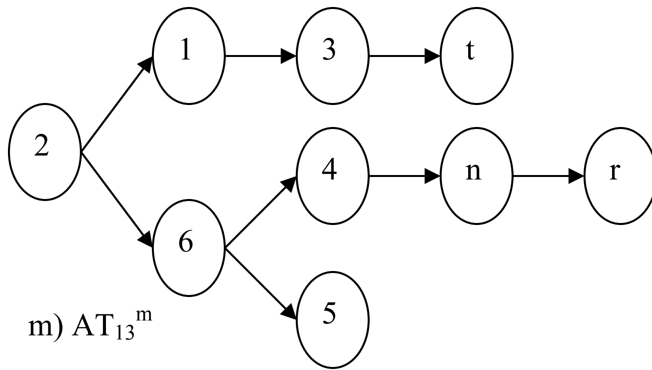
j) AT_{10}^{m}



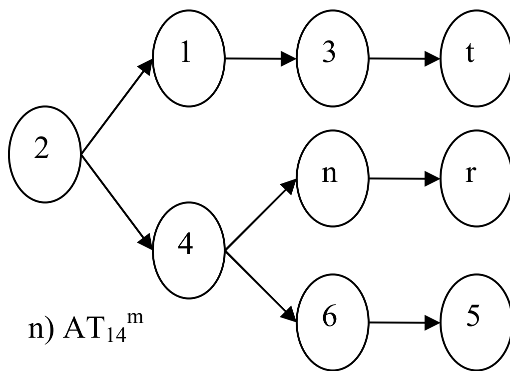
k) AT_{11}^{m}



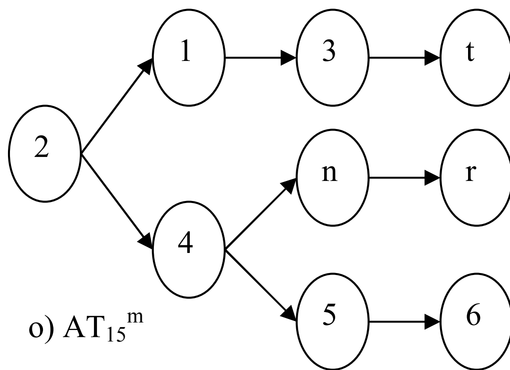
l) AT_{12}^{m}



m) AT_{13}^{m}



n) AT_{14}^{m}



o) AT_{15}^{m}

Figure A.11 The S_{AT^m} obtained from S_{AT} in Figure A.9

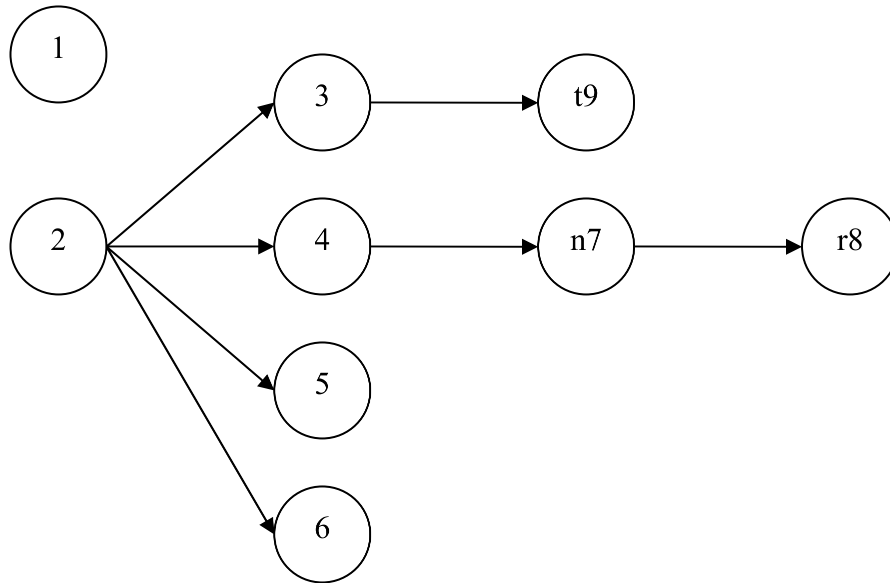
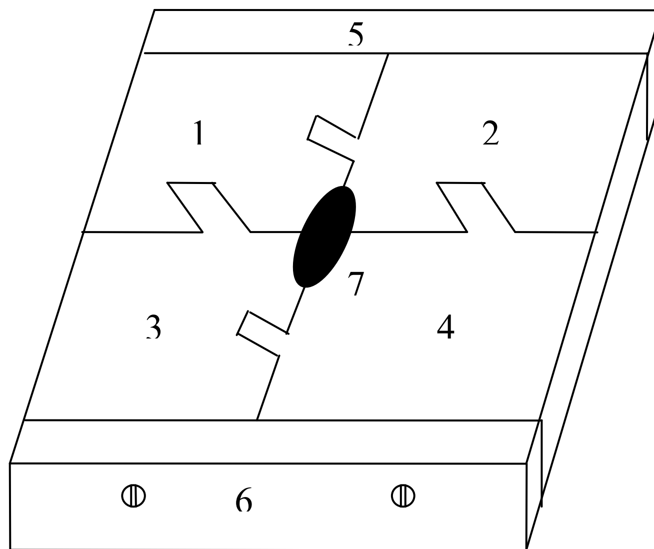


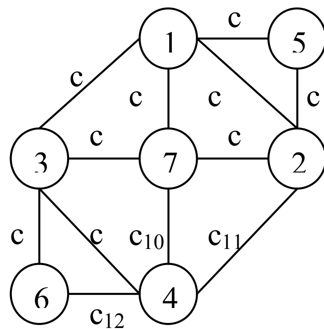
Figure A.12 S_{TPD} obtained by combining equitask AT^m's in Figure A.11

APPENDIX 3

Figures of the Example 3 in Section 3.3



a) the



b) its GOC

Figure A.13 An example product and its GOC (Lambert 1999)

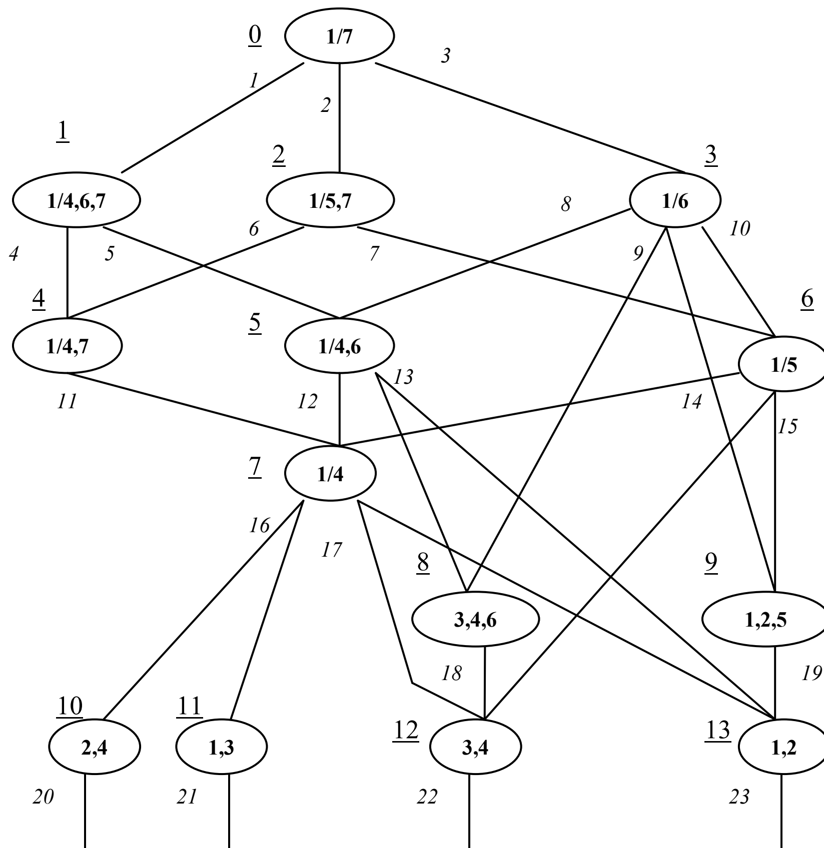
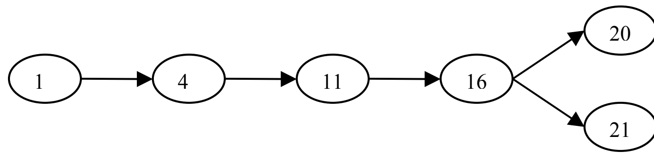
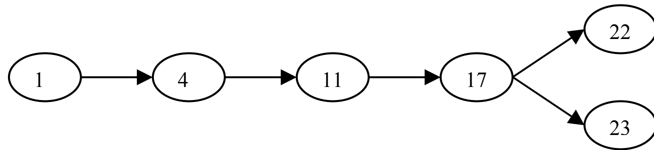


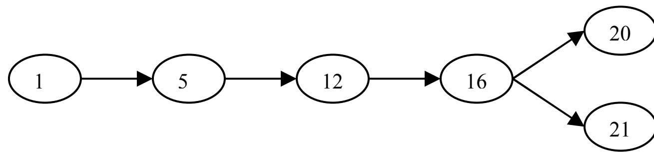
Figure A.14 AOG of the product in Figure A.13



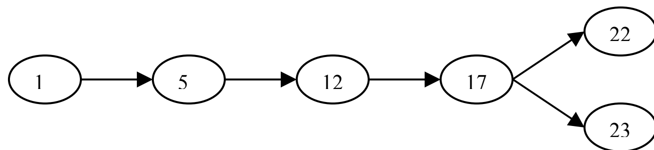
b) AT_1



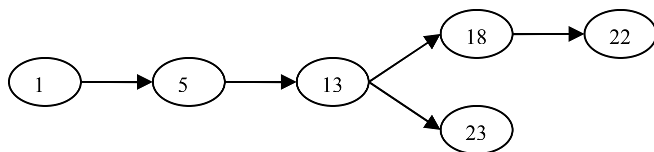
b) AT_2



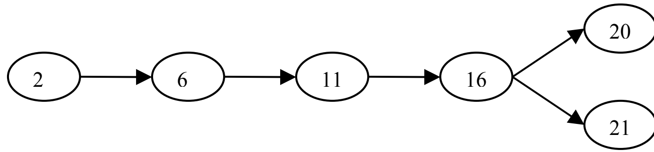
b) AT_3



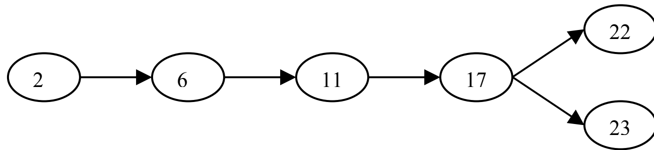
b) AT_4



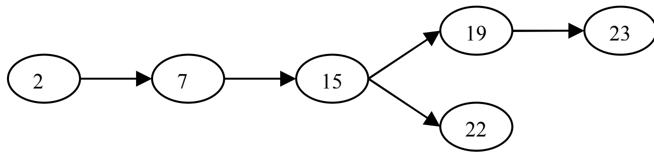
b) AT_5



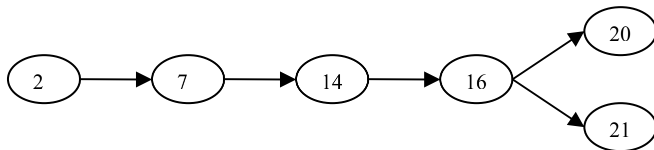
b) AT_6



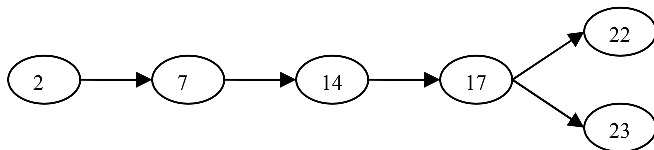
b) AT_7



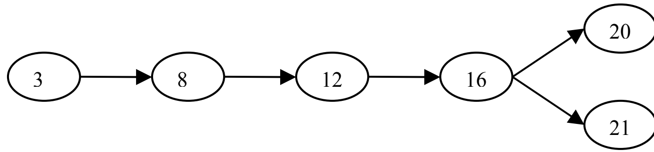
b) AT_8



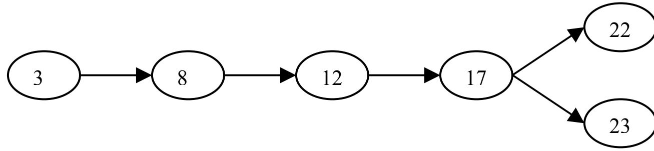
b) AT_9



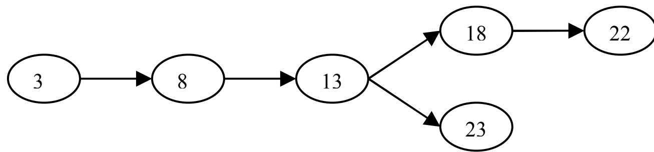
b) AT_{10}



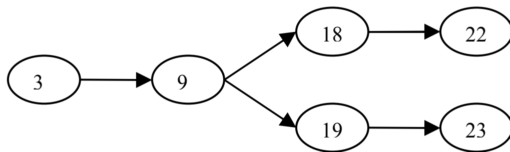
b) AT_{11}



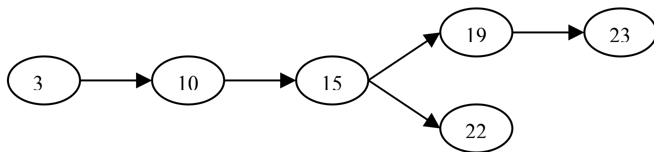
b) AT_{12}



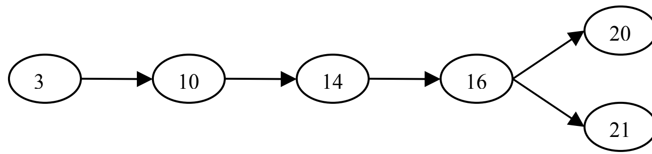
b) AT_{13}



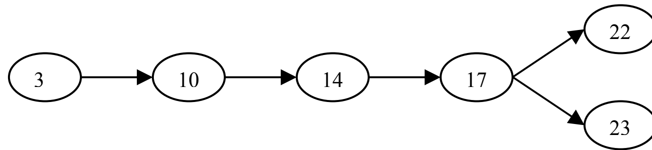
b) AT_{14}



b) AT_{15}

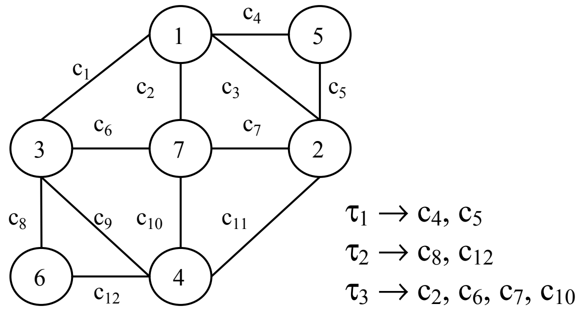


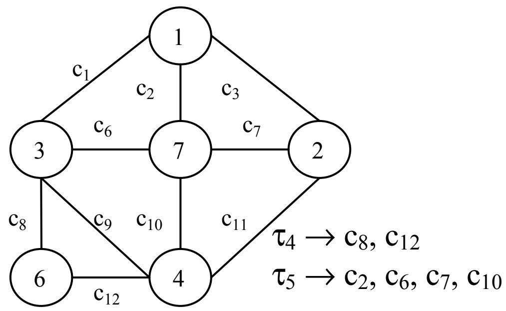
b) AT_{16}

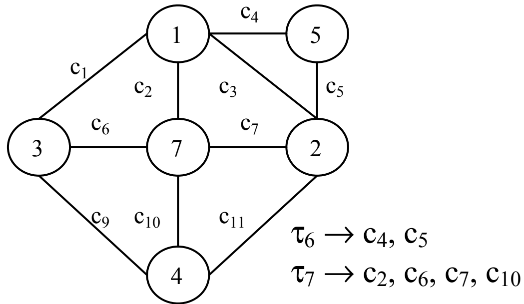


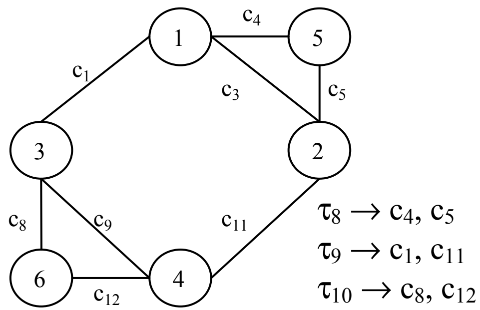
b) AT_{17}

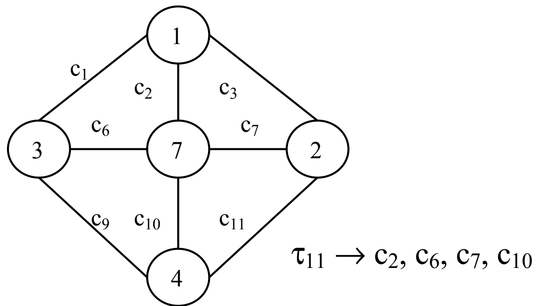
Figure A.15 S_{AT} established from the AOG in Figure A.14

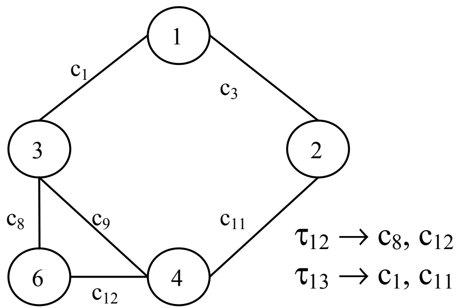

1
5
3
7
2
6
4
c_1
c_2
c_3
c_4
c_5
c_6
c_7
c_8
c_9
c_{10}
c_{11}
c_{12}
$\tau_1 \rightarrow c_4, c_5$
$\tau_2 \rightarrow c_8, c_{12}$
$\tau_3 \rightarrow c_2, c_6, c_7, c_{10}$

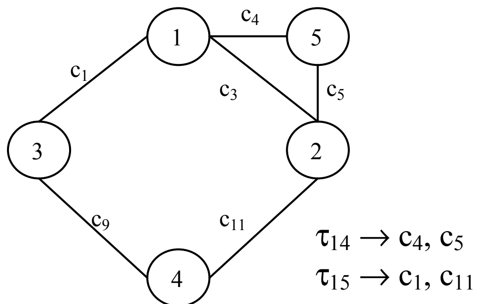

1
3
7
2
6
4
c_1
c_2
c_3
c_6
c_7
c_8
c_9
c_{10}
c_{11}
c_{12}
$\tau_4 \rightarrow c_8, c_{12}$
$\tau_5 \rightarrow c_2, c_6, c_7, c_{10}$

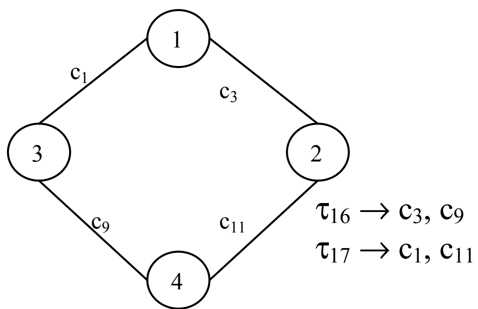

1
5
3
7
2
4
c_1
c_2
c_3
c_4
c_5
c_6
c_7
c_9
c_{10}
c_{11}
$\tau_6 \rightarrow c_4, c_5$
$\tau_7 \rightarrow c_2, c_6, c_7, c_{10}$


1
5
3
2
6
4
c_1
c_3
c_4
c_5
c_8
c_9
c_{11}
c_{12}
$\tau_8 \rightarrow c_4, c_5$
$\tau_9 \rightarrow c_1, c_{11}$
$\tau_{10} \rightarrow c_8, c_{12}$


1
c1
c2
c3
3
c6
7
c7
2
c9
c10
c11
4
τ11 → c2, c6, c7, c10


1
c1
c3
3
2
c8
c9
c11
6
4
c12
τ12 → c8, c12
τ13 → c1, c11


c4
1
5
c1
c3
c5
3
2
c9
c11
4
τ14 → c4, c5
τ15 → c1, c11


1
c1
c3
3
2
c9
c11
4
τ16 → c3, c9
τ17 → c1, c11

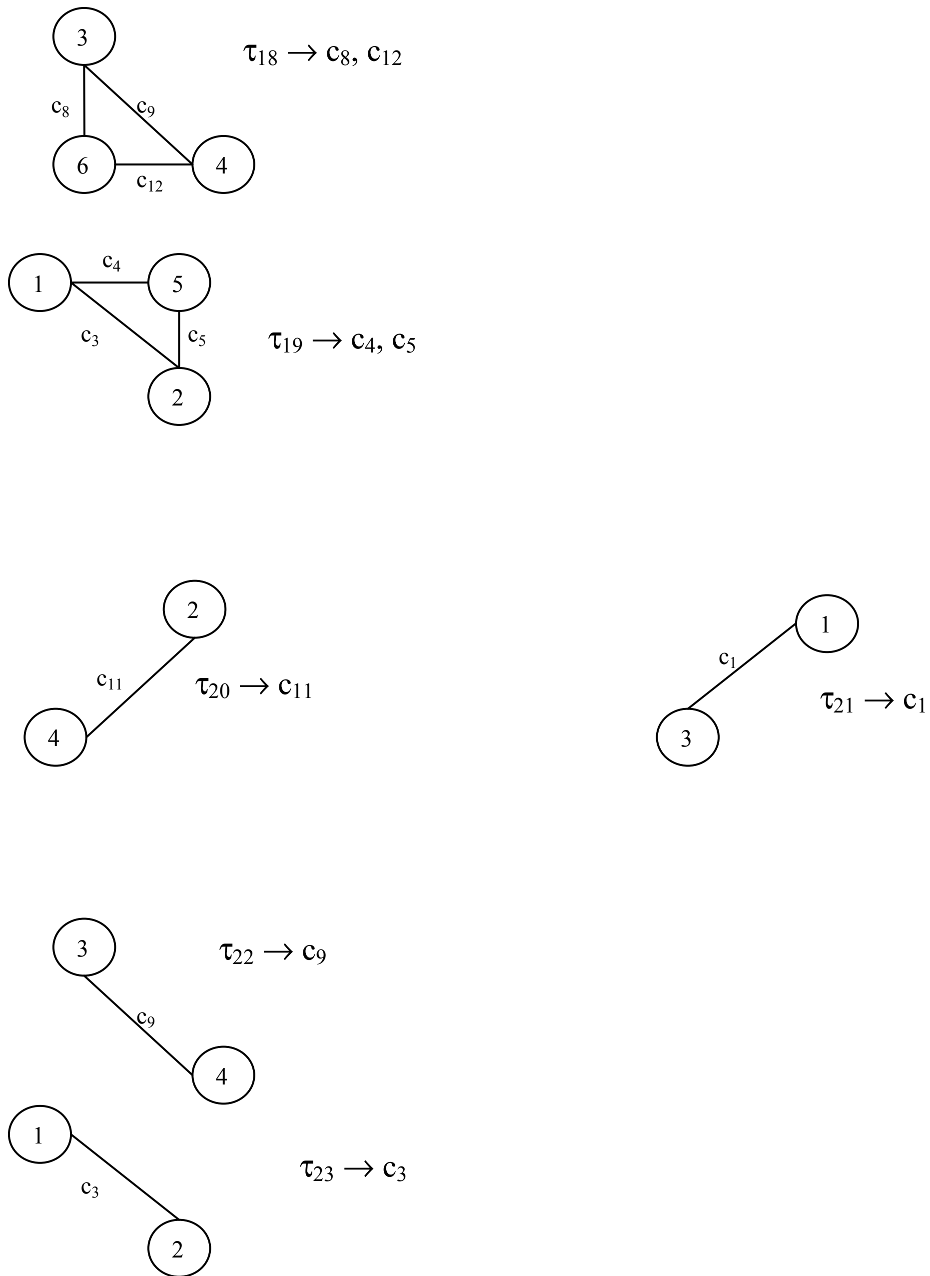
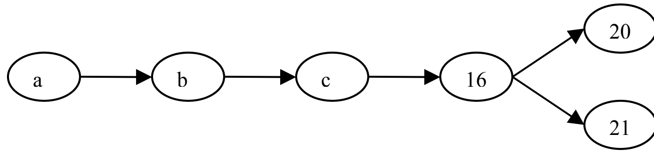
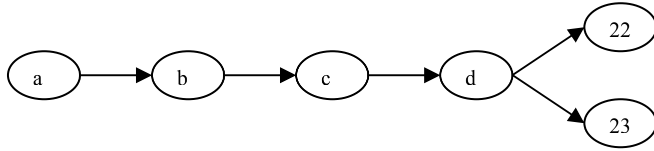


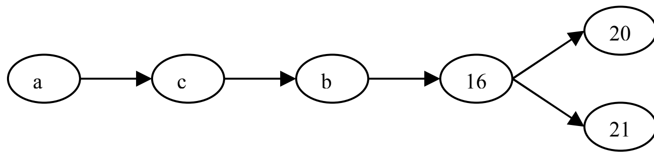
Figure A.16 The subassemblies of AOG in Figure A.14, their GOC and corresponding tasks



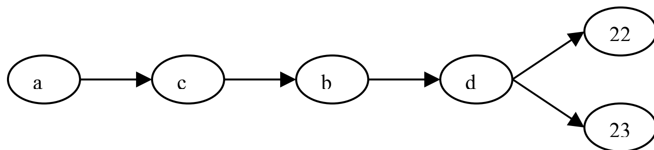
b) AT_1^m



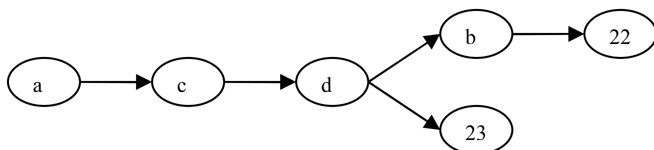
b) AT_2^m



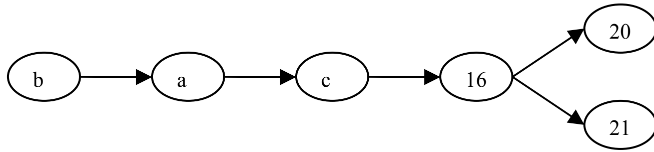
b) AT_3^m



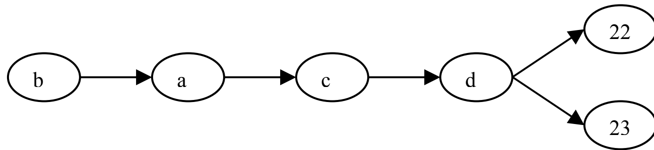
b) AT_4^m



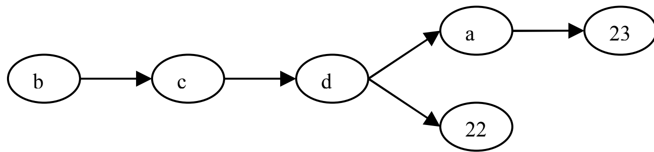
b) AT_5^m



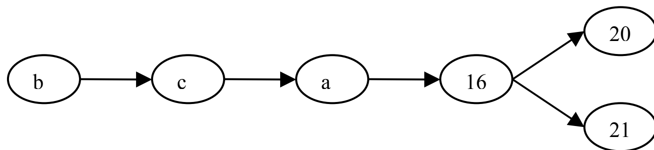
b) AT_6^m



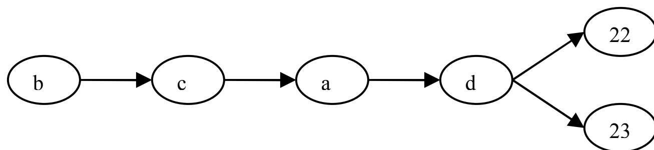
b) AT_7^m



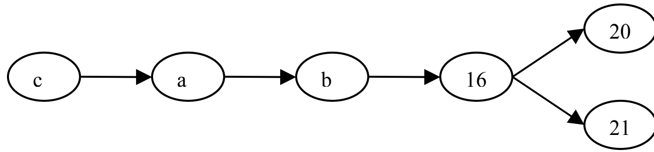
b) AT_8^m



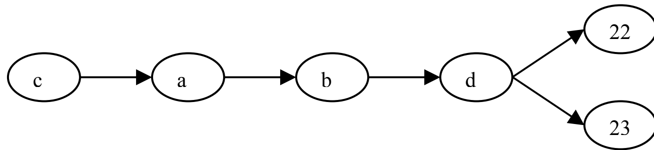
b) AT_9^m



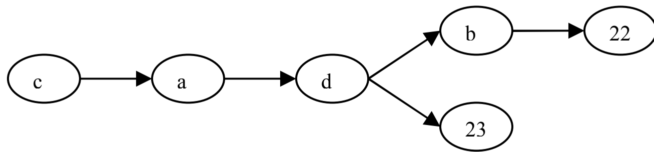
b) AT_{10}^m



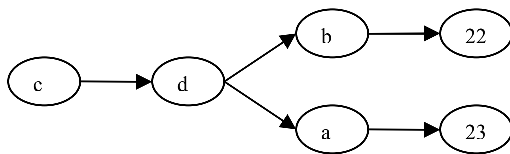
b) AT_{11}^{m}



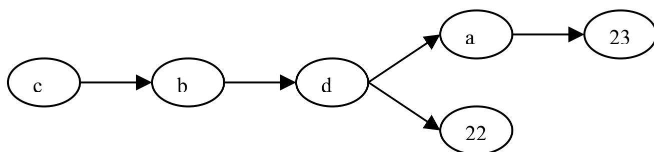
b) AT_{12}^{m}



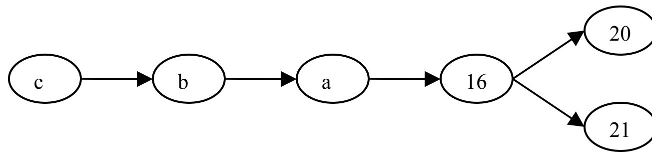
b) AT_{13}^{m}



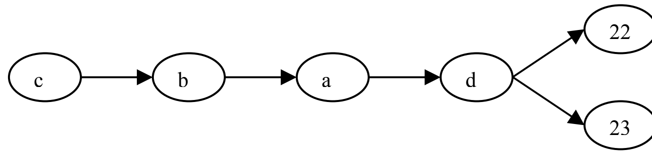
b) AT_{14}^{m}



b) AT_{15}^{m}



b) AT_{16}^{m}



b) AT_{17}^{m}

Figure A.17 The S_{AT^m} obtained from S_{AT} in Figure A.15

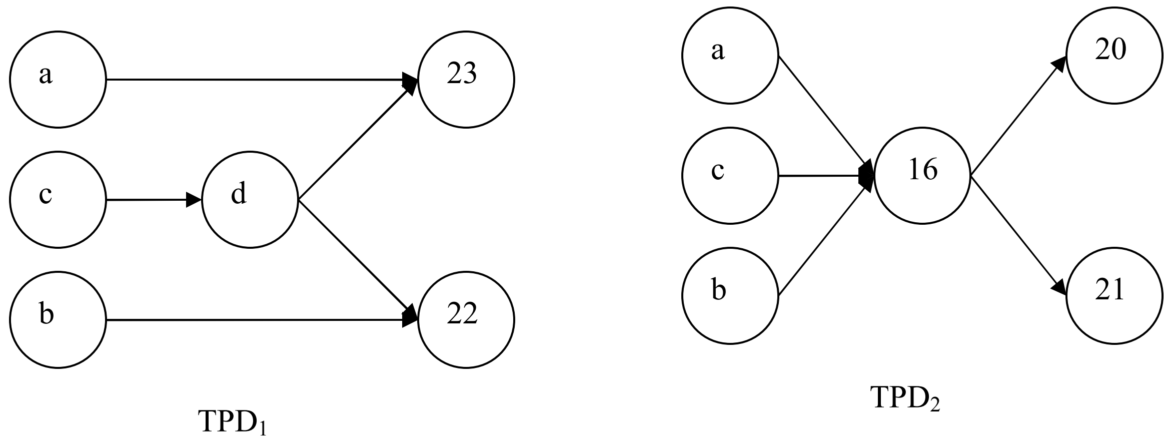


Figure A.18 S_{TPD} obtained by combining equitask AT^m's in Figure A.17

APPENDIX 4

Storing AOG in a Matrix

There are two types of nodes in AOG: artificial nodes and normal nodes. Each normal node is adjacent from and adjacent to one artificial node. The rows of the matrix represent the normal nodes and the columns represent the artificial nodes. The $(ij)^{th}$ entry is 0, if the normal node i is neither adjacent **from** nor adjacent **to** the artificial node j; it is 1 if the normal node i is adjacent **to** the artificial node j; and -1, if it is adjacent **from**. The below matrix represents the AOG in Figure 13 of Chapter 4.

	A0	***A1***	***A2***	***A3***	***A4***	***A5***	***A6***	***A7***	***A8***	***A9***	***A10***	***A11***	***A12***	***A13***
1	-1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	-1	0	1	0	0	0	0	0	0	0	0	0	0	0
3	-1	0	0	1	0	0	0	0	0	0	0	0	0	0
4	0	-1	0	0	1	0	0	0	0	0	0	0	0	0
5	0	-1	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	-1	0	1	0	0	0	0	0	0	0	0	0
7	0	0	-1	0	0	0	1	0	0	0	0	0	0	0
8	0	0	0	-1	0	1	0	0	0	0	0	0	0	0
9	0	0	0	-1	0	0	0	0	1	1	0	0	0	0
10	0	0	0	-1	0	0	1	0	0	0	0	0	0	0
11	0	0	0	0	-1	0	0	1	0	0	0	0	0	0
12	0	0	0	0	0	-1	0	1	0	0	0	0	0	0
13	0	0	0	0	0	-1	0	0	1	0	0	0	0	1
14	0	0	0	0	0	0	-1	1	0	0	0	0	0	0
15	0	0	0	0	0	0	-1	0	0	0	0	0	1	1
16	0	0	0	0	0	0	0	-1	0	0	1	1	0	0
17	0	0	0	0	0	0	0	-1	0	0	0	0	1	1
18	0	0	0	0	0	0	0	0	-1	0	0	0	1	0
19	0	0	0	0	0	0	0	0	0	-1	0	0	0	1
20	0	0	0	0	0	0	0	0	0	0	-1	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	-1	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	-1	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	-1

APPENDIX 5

Some Examples to Partial AOG (AOG(S))

All partial AOG's in this appendix are obtained from the AOG in Figure 13 of Chapter 4.

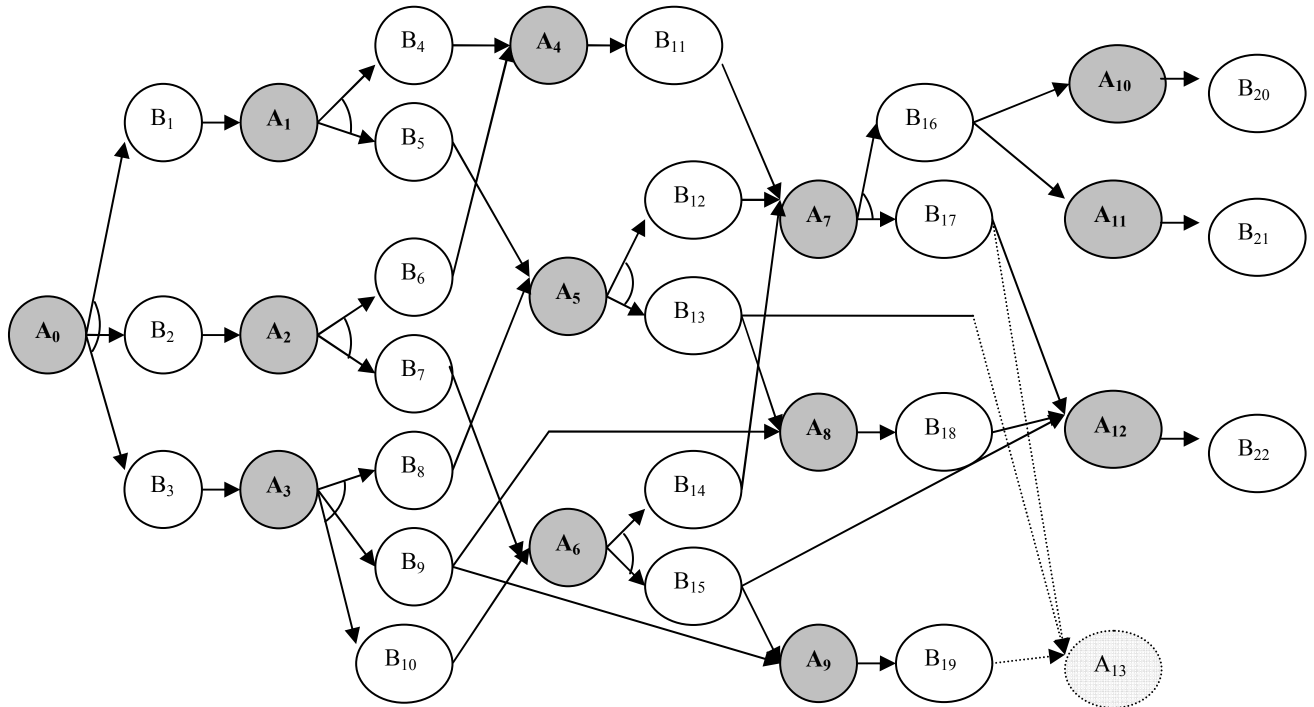


Figure A.19 AOG ({A_{13}})

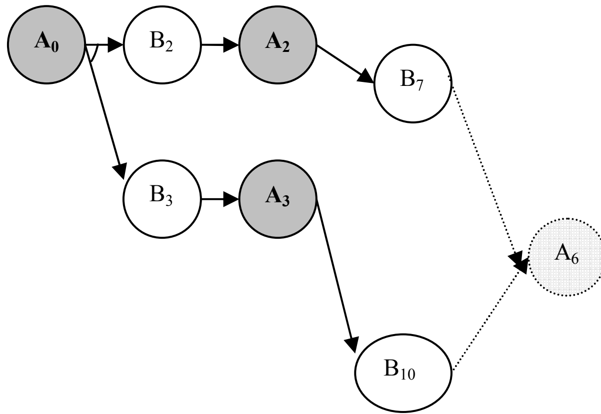


Figure A.20 AOG ({A_6})

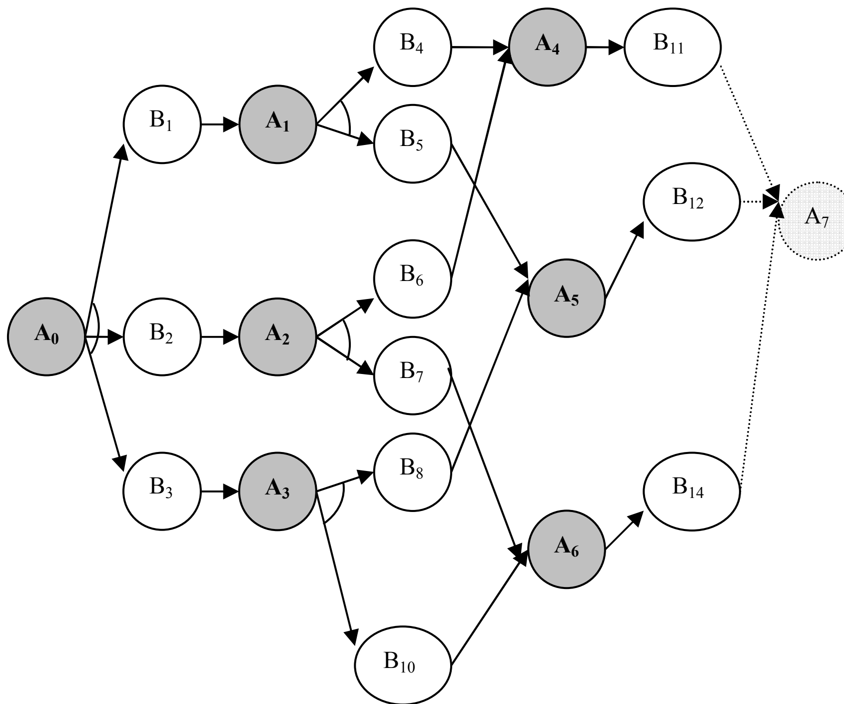


Figure A.21 AOG ({A_7})

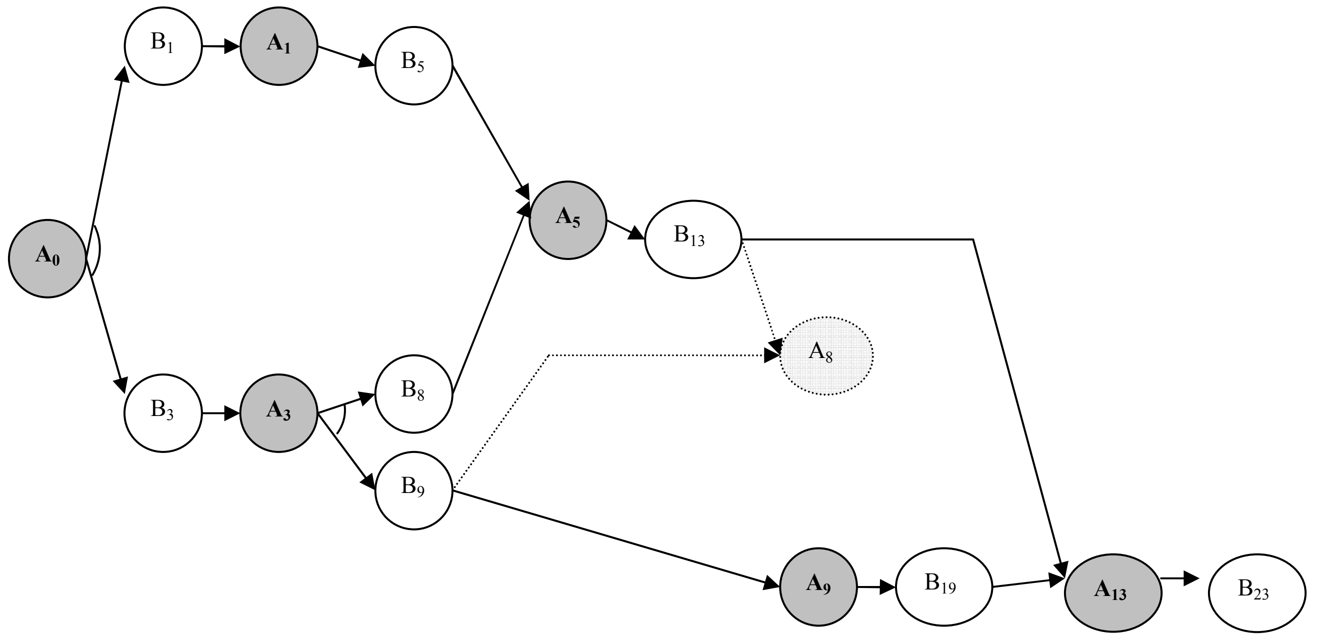


Figure A.22 AOG ({A_8})

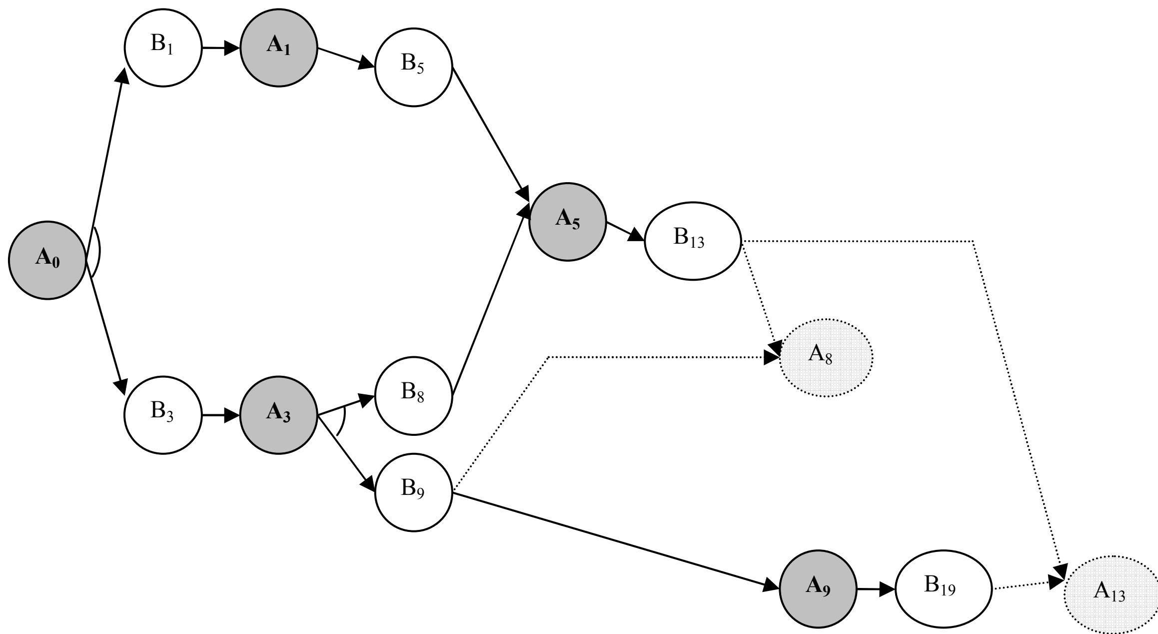


Figure A.23 AOG ({A_8, A_{13}})

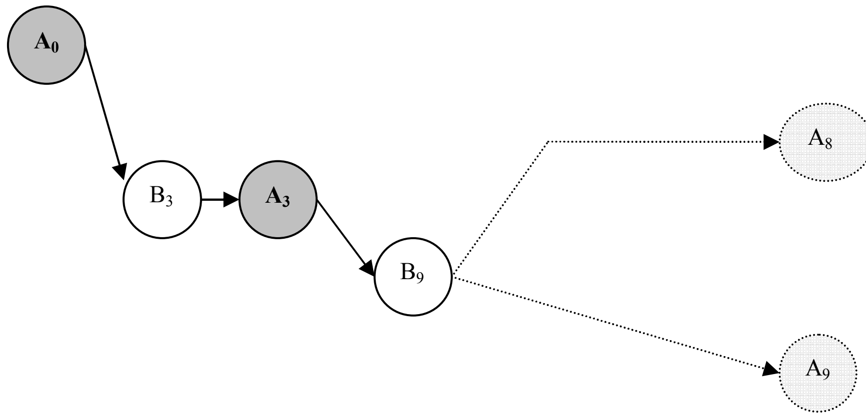


Figure A.24 AOG ({A_8, A_{13}, A_9}) = AOG ({A_8, A_9})

APPENDIX 6

Java Code for the DP method to the ADLB-AOG Problem

Class Result

```
import java.util.ArrayList;
public class Result {

   static Node[ ] nodes=new Node[10000];
   static int NUM=0;
   static int T = 20; //should be greater than the maximum duration




   static int [ ][ ] AOG = {{0,0,1,2,3,4,5,6,7,8,9,10,11,12,13},
            {1,-1,1,0,0,0,0,0,0,0,0,0,0,0,0},{2,-1,0,1,0,0,0,0,0,0,0,0,0,0,0},
            {3,-1,0,0,1,0,0,0,0,0,0,0,0,0,0},{4,0,-1,0,0,1,0,0,0,0,0,0,0,0,0},
            {5,0,-1,0,0,0,1,0,0,0,0,0,0,0,0},{6,0,0,-1,0,1,0,0,0,0,0,0,0,0,0},
            {7,0,0,-1,0,0,0,1,0,0,0,0,0,0,0},{8,0,0,0,-1,0,1,0,0,0,0,0,0,0,0},
            {9,0,0,0,-1,0,0,0,0,1,1,0,0,0,0},{10,0,0,0,-1,0,0,1,0,0,0,0,0,0,0},
            {11,0,0,0,0,-1,0,0,1,0,0,0,0,0,0},{12,0,0,0,0,0,-1,0,1,0,0,0,0,0,0},
            {13,0,0,0,0,0,-1,0,0,1,0,0,0,0,1},{14,0,0,0,0,0,0,-1,1,0,0,0,0,0,0},
            {15,0,0,0,0,0,0,-1,0,0,0,0,0,1,1},{16,0,0,0,0,0,0,0,-1,0,0,1,1,0,0},
            {17,0,0,0,0,0,0,0,-1,0,0,0,0,1,1},{18,0,0,0,0,0,0,0,0,-1,0,0,0,1,0},
            {19,0,0,0,0,0,0,0,0,0,-1,0,0,0,1},{20,0,0,0,0,0,0,0,0,0,0,-1,0,0,0},
            {21,0,0,0,0,0,0,0,0,0,0,0,-1,0,0},{22,0,0,0,0,0,0,0,0,0,0,0,0,-1,0},
            {23,0,0,0,0,0,0,0,0,0,0,0,0,0,-1}};

   static int[ ] durations = {22,22,14,21,13,21,13,21,16,21,12,20,15,20,15,14,14,18,18,7,7,7,7};

   //task numbers and task durations must match.


   public static void main(String[] args) {

            hyerarchy ();

            System.out.println("Optimal cost is "+cost(nodes[0]));
```

```
            System.out.println("Optimal path is "+path());

    }// method main


    public static void hyerarchy (){

            ArrayList set = new ArrayList();
            ArrayList temp_set = new ArrayList();
            boolean bol = true;

            nodes[0]=new Node(NUM,AOG);
            set.add(""+NUM);

            while(set.size()!=0){

                        for(int a=0;a<set.size();a++){

                        if(nodes[Integer.parseInt(""+set.get(a))].matrix.length!=2){
                                    for (int
b=0;b<nodes[Integer.parseInt(""+set.get(a))].lastnodes.size();b++){
                                                if(temp_set!=null){
                                                            for(int c=0;c<temp_set.size();c++){
                                                                        if(check_equivalence
(nodes[Integer.parseInt(""+temp_set.get(c))].matrix,Graph.graph_form
(nodes[Integer.parseInt(""+set.get(a))].matrix,
Integer.parseInt(""+nodes[Integer.parseInt(""+set.get(a))].lastnodes.get(b))))){

    nodes[Integer.parseInt(""+set.get(a))].set_torun(Integer.parseInt(""+temp_set.get(c)),Integer.parse
Int(""+nodes[Integer.parseInt(""+set.get(a))].lastnodes.get(b)));
                                                                                    bol=false;
                                                                        }
                                                            }
                                                }

                                                if (bol){
                                                            NUM++;
                                                            temp_set.add(""+NUM);

    nodes[Integer.parseInt(""+set.get(a))].set_torun(NUM,Integer.parseInt(""+nodes[Integer.parseInt(""
+set.get(a))].lastnodes.get(b)));
                                                            nodes[NUM]=new
Node(NUM,Graph.graph_form(nodes[Integer.parseInt(""+set.get(a))].matrix,
Integer.parseInt(""+nodes[Integer.parseInt(""+set.get(a))].lastnodes.get(b))));

                                                }
                                                bol=true;

                                    }
                        }
```

```
				set.remove(a);
				a--;

			}

			for (int a=0;a<temp_set.size();a++){
				set.add(""+temp_set.get(a));
			}
			while(temp_set.size()!=0){
				temp_set.remove(0);
			}
		}

	}// method hyerarchy



	public static boolean check_equivalence(int[ ][ ] mat1, int[ ][ ] mat2){
		boolean ret_value = true;

		if (mat1.length==mat2.length){
			for (int a=0;a<mat1.length;a++){
				if (mat1[a].length!=mat2[a].length){
					ret_value=false;
				}
			}

			if(ret_value){
				for (int a=0;a<mat1.length;a++){
					for (int b=0;b<mat1[a].length;b++){
						if (mat1[a][b]!=mat2[a][b]) ret_value=false;
					}
				}
			}

		}

		else ret_value=false;


		return ret_value;


	} // method check_equivalence



	public static int cost (Node nd){
		int cost=0, alfa = 0, y = 0, al = 0, b = 100000000;
		if(nd.matrix.length==2){
			nodes[nd.number].set_cost(durations[nd.matrix[1][0]-1]);
```

```
        }

        else {
                for (int a=0;a<nd.torun[0].size();a++){

                        if(nodes[Integer.parseInt(""+nd.torun[0].get(a))].cost!=0){
                                cost=nodes[Integer.parseInt(""+nd.torun[0].get(a))].cost;
                        }

                        else cost=cost(nodes[Integer.parseInt(""+nd.torun[0].get(a))]);

                        y=durations[nd.matrix[Integer.parseInt(""+nd.torun[1].get(a))][0]-1];

                        al=(int)((cost+y)/T);
                        if(al==(float)(cost+y)/T||al==(int)(cost/T)){
                                alfa=y;
                        }
                        else alfa=T*al+y-cost;


                        if (cost+alfa<b){

                                b= cost+alfa;
                        }

                }


                nodes[nd.number].set_cost(b);
                b=100000000;

        }

        return nodes[nd.number].cost;

}//method cost


public static ArrayList path(){
        Node nod=nodes[0];
        ArrayList path = new ArrayList();
        int cost=0, y=0, al=0, alfa =0;


        while(nod.matrix.length>2){
                for(int a=0;a<nod.torun[0].size();a++){

                        cost=nodes[Integer.parseInt(""+nod.torun[0].get(a))].cost;

                        y=durations[nod.matrix[Integer.parseInt(""+nod.torun[1].get(a))][0]-1];
```

```
                                al=(int)((cost+y)/T);
                                if(al==(float)(cost+y)/T||al==(int)(cost/T)){
                                        alfa=y;
                                }
                                else alfa=T*al+y-cost;



                                if(nod.cost==cost+alfa){
  path.add(""+nod.matrix[Integer.parseInt(""+nod.torun[1].get(a))][0]);
                                        nod=nodes[Integer.parseInt(""+nod.torun[0].get(a))];
                                        a=nod.torun[0].size();
                                }
                        }

                }

                path.add(""+nod.matrix[1][0]);

                return path;

  }//method path

} // class Result
```

Class Graph

```
public class Graph {

  public static int[][] graph_form (int[][] matrix, int deleted){

            return new_graph (remaining_nodes(matrix,deleted),matrix);

  }//method graph_form


  private static int[][] new_graph(int[] rem_nodes, int [][] matr){


//...............................row_elimination................//
            int[][] parti1 = new int[rem_nodes.length+1][matr[0].length];

            for (int b=0;b<matr[0].length;b++){
                    parti1[0][b]=matr[0][b];
            }

            for (int a=0;a<rem_nodes.length;a++){
                    for(int b=0;b<parti1[0].length;b++){
                            parti1[a+1][b]=matr[rem_nodes[a]][b];
                    }
            }


//...............................row_elimination................\\

//...............................column_elimination............//
//...............................empty_columns............//
            boolean del=false;
            ArrayList keep_art=new ArrayList();

            for(int b=1;b<parti1[0].length;b++){
                    for (int a=1;a<parti1.length;a++){
                            if (parti1[a][b]==-1){
                                    del=true;
                            }

                    }
                    if(del) {
                            keep_art.add(""+b);
                            del=false;
                    }

            }

            int [ ] columns = new int[keep_art.size()];
```

```
		for(int a=0;a<keep_art.size();a++){
			columns[a]=Integer.parseInt(""+keep_art.get(a));
		}

//................................empty_columns.............\\

		int[ ][ ] parti2 = new int[parti1.length][columns.length+1];

		for (int a=0;a<parti1.length;a++){
			parti2[a][0]=parti1[a][0];
		}

		for (int b=0;b<columns.length;b++){
			for(int a=0;a<parti2.length;a++){
				parti2[a][b+1]=parti1[a][columns[b]];
			}
		}

//................................column_elimination............\\
		return parti2;

   }//method new_graph


   private static int[] remaining_nodes (int[ ][ ] par, int deleted){
		boolean check1 = false, check2 = true;
		boolean check_add1=true, check_add2=true, check_add3=true;

		ArrayList APtC = new ArrayList();
		ArrayList APC = new ArrayList();
		ArrayList NtC = new ArrayList();
		ArrayList NK = new ArrayList();
		ArrayList AStC = new ArrayList();
		ArrayList ASC = new ArrayList();
		ArrayList NStC = new ArrayList();

		NtC.add(""+deleted);

		while(NtC.size()!=0){
//......................................remove from normal to be checked//
//......................................add predecessor artificials//
			for (int a=0;a<NtC.size();a++){
				for(int b=1;b<par[0].length;b++){
					if(par[Integer.parseInt(""+(NtC.get(a)))][b]==-1){
						for(int c=0;c<APtC.size();c++){
							if(Integer.parseInt(""+APtC.get(c))==b){
								check_add2=false;
							}

						}
						if(check_add2){
```

```
                                        APtC.add(""+b);
                                }
                                else check_add2=true;
                        }
                }
//.....................................add predecessor artificials\\

//.....................................add successor artificials//

                for(int b=1;b<par[0].length;b++){
                        if(par[Integer.parseInt(""+(NtC.get(a)))][b]==1){
                                check1 = true;
                                for (int l=0;l<APC.size();l++){
                                        if(b==Integer.parseInt(""+(APC.get(l)))){
                                                check2 = false;
                                        }
                                }
                        }
                        if(check1&check2){
                                for(int c=0;c<AStC.size();c++){
                                        if(Integer.parseInt(""+AStC.get(c))==b){
                                                check_add3=false;
                                        }

                                }
                                if(check_add3){
                                        AStC.add(""+b);
                                }
                                else check_add3=true;
                        }
                        check1=false;
                        check2=true;
                }


//.....................................add successor artificials\\

                NK.add(NtC.get(a));
                NtC.remove(a);
                a--;
        }
//.....................................remove from normal to be checked\\

//................................SUCESSSSSSSSSOOOOOOOOOOOOOOOORRRRRRRRRR
//.....................................remove the previously checked artificial successor//

        while(AStC.size()!=0){
                for (int k=0;k<AStC.size();k++){
                        for (int l=0;l<ASC.size();l++){

  if(Integer.parseInt(""+(AStC.get(k)))==Integer.parseInt(""+(ASC.get(l)))){
```

```
                                        AStC.remove(k);
                                        l=ASC.size();
                                        k--;
                                    }
                                }
                            }

//.....................................remove the previously checked artificial successor\\

//.........................remove from artificial successor to be checked//
//.................................................add successor normals//
                            for (int b=0;b<AStC.size();b++){
                                for(int a=1;a<par.length;a++){
                                    if(par[a][Integer.parseInt(""+(AStC.get(b)))]==-1){
                                        NStC.add(""+a);
                                    }
                                }
//.....................................add successor normals\\

                                ASC.add(AStC.get(b));
                                AStC.remove(b);
                                b--;
                            }

//.....................................remove from artificial suceesor to be checked\\
//.....................................remove from normal successor to be checked//


                            for (int a=0;a<NStC.size();a++){
                                for(int b=1;b<par[0].length;b++){
                                    if(par[Integer.parseInt(""+(NStC.get(a)))][b]==1){
                                        for(int c=0;c<AStC.size();c++){
                                            if(Integer.parseInt(""+AStC.get(c))==b){
                                                check_add1=false;
                                            }

                                        }
                                        if(check_add1){
                                            AStC.add(""+b);
                                        }
                                        else check_add1=true;
                                    }
                                }
                                NK.add(""+Integer.parseInt(""+(NStC.get(a))));
                                NStC.remove(a);
                                a--;
                            }

//.....................................remove from normal successor to be checked\\
```

```
			}//while
//..........................SUCESSSSSSSSSOOOOOOOOOOOOOOOORRRRRRRRRR


//.....................................remove the previously checked artificial predecessors//
			for (int k=0;k<APtC.size();k++){
				for (int l=0;l<APC.size();l++){

   if(Integer.parseInt(""+(APtC.get(k)))==Integer.parseInt(""+(APC.get(l)))){
							APtC.remove(k);
							l=APC.size();
							k--;
					}

				}
			}
//.....................................remove the previously checked artificial\\
//.....................................remove from artificial predeccessor to be checked//


			for (int b=0;b<APtC.size();b++){
				for(int a=1;a<par.length;a++){
					if(par[a][Integer.parseInt(""+(APtC.get(b)))]==1){
						NtC.add(""+a);
					}
				}
				APC.add(""+Integer.parseInt(""+(APtC.get(b))));
				APtC.remove(b);
				b--;
			}

//.....................................remove from artificial to be checked\\
		}//while

//.......................................delete the last node//
		for (int a=0;a<NK.size();a++){
			if(Integer.parseInt(""+NK.get(a))==deleted){
				NK.remove(a);
				break;
			}
		}
//.......................................delete the last node\\

//...........................................rearrange remaining nodes//
		int[ ] rows=new int[NK.size()];
		int s=par.length, t=0;

		while(NK.size()>0){
			s=par.length;
			for (int a=0;a<NK.size();a++){
				if(Integer.parseInt(""+NK.get(a))<s){
```

```
                        s=Integer.parseInt(""+NK.get(a));
                    }
                }
                rows[t]=s;
                t++;

                for (int a=0;a<NK.size();a++){
                    if(Integer.parseInt(""+NK.get(a))==s){
                        NK.remove(a);
                        break;
                    }
                }

            }


//...........................................rearrange remaining nodes//

            return rows;


    }//method remaining_nodes


}// class graph
```

Class Node

```
class Node {
   int number, cost;
   int [ ][ ] matrix;
   ArrayList lastnodes;
   ArrayList [ ] torun;

   public Node(int no, int [ ][ ] matr){
            lastnodes = new ArrayList();
            torun = new ArrayList[2];
            torun[0]=new ArrayList();
            torun[1]=new ArrayList();

            matrix=matr;
            number=no;


            //latnodes are in terms of the row numbers in the matrix
            boolean last_node=false;
            for (int a=1;a<matrix.length;a++){
                     for (int b=1;b<matrix[0].length;b++){
                              if(matrix[a][b]==1) last_node=true;

                     }
                     if(last_node==false){

                              lastnodes.add(""+a);
                     }
                     else last_node=false;
            }

   }//constructor

   public void set_torun(int tor, int last){
            torun[0].add(""+tor);
            torun[1].add(""+last);

   }

   public void set_cost(int cos){
            cost=cos;
   }


}//class Node
```

APPENDIX 7

Java Code for the Formulation of ADLB-AOG problem as pure 0-1 IP Programming

Class AOG_IP_Formulation

```
import java.io.*;
import java.util.ArrayList;
public class AOG_IP_Formulation {


  static int [ ][ ] AOG =      {{0,0,1,2,3,4,5},
                               {1,-1,1,0,0,0,0},{2,-1,0,0,1,0,1},
                               {3,-1,0,1,0,0,0},{4,0,-1,0,1,0,0},
                               {5,0,-1,0,0,1,0},{6,0,0,-1,0,1,0},
                               {7,0,0,-1,0,0,1},{8,0,0,0,-1,0,0},
                               {9,0,0,0,0,-1,0},{10,0,0,0,0,0,-1}};

  static int[ ] durations = {12,7,9,6,8,11,6,7,6,10};

  //task numbers and task durations must match.


  static int numParts = 4;
  static int T=13;


  public static void main(String[] args)throws IOException {

          ArrayList art_eksi1 = new ArrayList();
          ArrayList art_arti1 = new ArrayList();

          String file = "model.txt";
          final int M = numParts-1;


          FileWriter fw = new FileWriter(file);
          BufferedWriter bw = new BufferedWriter(fw);
          PrintWriter outFile = new PrintWriter(bw);
```

```
//Objective Function
		outFile.println("minimize ");
		for (int a= 1; a<=M;a++){
			outFile.print("+ "+a+"f"+a);

		}

		outFile.println();
		outFile.println("subject to");

//constraint for A0

		outFile.println();
		for (int a=1;a<AOG.length;a++){
			if(AOG[a][1]==-1)		art_eksi1.add(""+a);
		}

		for (int a = 0; a<art_eksi1.size();a++){
			outFile.print("+ Z"+art_eksi1.get(a));

		}
			outFile.println(" = 1");



// constraints for Ai

		outFile.println();
		for(int b=2;b<AOG[0].length;b++){
			art_arti1=new ArrayList();
			art_eksi1=new ArrayList();
			for (int a=1;a<AOG.length;a++){
				if(AOG[a][b]==-1)		art_eksi1.add(""+a);
				if(AOG[a][b]==1)		art_arti1.add(""+a);
			}

			for (int a = 0; a<art_eksi1.size();a++){
				outFile.print("+ Z"+art_eksi1.get(a));

			}

			for (int a = 0; a<art_arti1.size();a++){
				outFile.print("- Z"+art_arti1.get(a));


			}

			outFile.println(" = 0");
		}
```

```
//constraints for assuring that the task is assigned to the station

		outFile.println();
		for(int a=1;a<AOG.length;a++){
			for(int j=1;j<=M;j++){
				outFile.print("+ X"+AOG[a][0]+","+j);

			}
			outFile.println("- Z"+AOG[a][0]+" = 0");
		}


//precedence constraints

		outFile.println();
		for(int b=2;b<AOG[0].length;b++){
			art_arti1=new ArrayList();
			art_eksi1=new ArrayList();
			for (int a=1;a<AOG.length;a++){
				if(AOG[a][b]==-1)		art_eksi1.add(""+a);
				if(AOG[a][b]==1)		art_arti1.add(""+a);
			}

			for (int a = 0; a<art_arti1.size();a++){
				for(int j=1;j<=M;j++ ){
					outFile.print("+ "+j+"X"+art_arti1.get(a)+","+j);


				}
			}

			for (int a = 0; a<art_eksi1.size();a++){
				for(int j=1;j<=M;j++ ){

					outFile.print("- "+j+"X"+art_eksi1.get(a)+","+j);

				}
			}

			outFile.println(" <= 0");
		}


//cycle time constraints

		outFile.println();
		for (int j=1;j<=M;j++){
			for(int a=1;a<AOG.length;a++){
				outFile.print("+ "+durations[a-1]+"X"+AOG[a][0]+","+j);
			}
			outFile.println("- "+T+"f"+j+" <=0");
```

```
            }

            System.out.println ("File "+file+" is okey");
            outFile.close();

    }// class main

    private static void PrintToFile(int[ ][ ] matrix, PrintWriter outFile)throws IOException{

            for (int a=0;a<matrix.length;a++){
                    for (int b=0;b<matrix[0].length;b++){

                            if(b==1) outFile.print("\t");
                            outFile.print(matrix[a][b]+"   ");
                    }
                    if(a==0){
                            outFile.println("\n");
                    }
                    else{
                            outFile.println();
                    }
            }

    }// method PrintToFile

}//class AOG_IP_Formulation
```

APPENDIX 8

Java Code to Generate AOG

Class Generate_Graph

```
public class Generate_Graph {

   int numParts;
   int numArts;
   final int numTasks= 2;
   final int numParallel= 0;
   final int duration_ilk=1,duration_son=20;


   static int[ ][ ] AOG;
   static int[ ] durations;

   public Generate_Graph (int nub, int nua){
            numParts=nub;
            numArts=nua;
   }

   public int[][] AOG_generator() {

            Generate_Node[ ][ ] graph = new Generate_Node[numParts-1][ ];
            int count=0,assign=0;
            int toruns, tasks;


// Sequential Disassembly

            for (int a=numParts;a>2;a--){
                    if (a==numParts){
                            graph[numParts-a]=new Generate_Node[1];
                            graph[numParts-a][0]=new Generate_Node(count);
                            count++;

                            toruns = numArts;                   //Rand.triangle(0,4,8,0.75);//
                            graph[numParts-a+1]=new Generate_Node[toruns];

                            for (int b=0;b<toruns;b++){
                                    graph[numParts-a+1][b]=new Generate_Node(count);
```

```
					graph[numParts-a][0].set_tor(""+count);
					count++;
				}

			}

			else {

				int of=1;
				boolean tekrar=true;

				toruns = numArts;			//Rand.triangle(0,4,8,0.75);//
				graph[numParts-a+1]=new Generate_Node[toruns];
				assign=count;
				for (int b=0;b<toruns;b++){
					graph[numParts-a+1][b]=new Generate_Node(count);
					count++;
				}

				while(tekrar&(assign!=count)){

					for (int b=0;b<graph[numParts-a].length;b++){
						tasks=numTasks;
						for(int c=0;c<tasks;c++){
							if((of==1)||tekrar){
								if (assign==count)	{
									assign-=toruns;
									tekrar=false;
								}
								graph[numParts-a][b].set_tor(""+assign);
								assign++;
								if (assign==count)	{
									tekrar=false;
								}
							}
						}
					}
					of++;
				}


			}
		}

/*


//Generate AOG

		int art_count=0,task_count=0;			//count tasks and artificials
```

```
for(int a=0;a<graph.length;a++){
	if(a!=graph.length-1){
		for(int b=0;b<graph[a].length;b++){
			art_count++;

			for(int c=0;c<graph[a][b].tors.size();c++){
				task_count++;
			}
		}
	}
	else {
		art_count+=graph[graph.length-1].length;
		task_count+=graph[graph.length-1].length;
	}
}
System.out.print(task_count+"\t");

for(int a=0;a<graph.length-2;a++){
	for(int b=0;b<graph[a].length;b++){
		for(int c=0;c<graph[a][b].p_tor1.size();c++){
			task_count++;
		}
	}
}


AOG=new int[task_count+1][art_count+1];

for(int b=1;b<AOG[0].length;b++){
	AOG[0][b]=b-1;
}

int cou=1;
for(int a=0;a<graph.length-1;a++){
	for(int b=0;b<graph[a].length;b++){
		for(int c=0;c<graph[a][b].tors.size();c++){
			AOG[cou][0]=cou;
			AOG[cou][graph[a][b].no+1]=-1;
			AOG[cou][Integer.parseInt(""+graph[a][b].tors.get(c))+1]=1;
			cou++;
		}
	}
}

for(int b=0;b<graph[graph.length-1].length;b++){
	AOG[cou][0]=cou;
	AOG[cou][graph[graph.length-1][b].no+1]=-1;
	cou++;
}
```

```
		for(int a=0;a<graph.length-2;a++){
			for(int b=0;b<graph[a].length;b++){
				for(int c=0;c<graph[a][b].p_tor1.size();c++){
					AOG[cou][0]=cou;
					AOG[cou][graph[a][b].no+1]=-1;
					AOG[cou][Integer.parseInt(""+graph[a][b].p_tor1.get(c))+1]=1;
					AOG[cou][Integer.parseInt(""+graph[a][b].p_tor2.get(c))+1]=1;
					cou++;
				}
			}
		}


		return AOG;
	}//method AOG_generator


	public int[] durations_generator() {
		durations=new int[AOG.length-1];

		for (int a=0;a<durations.length;a++){
			durations[a]=Rand.duration(duration_ilk,duration_son);
		}

		return durations;
	}//method duration_generator


}//class Generate_Graph
```

Class Generate_Node

```
import java.util.ArrayList;
public class Generate_Node {

   int no;
   ArrayList tors = new ArrayList();
   ArrayList p_tor1 = new ArrayList();
   ArrayList p_tor2 = new ArrayList();


   public Generate_Node(int number) {
            no=number;
   }// constructor Generate_Node

   public void set_tor(String tor){
            tors.add(tor);
   }// method set_tor

   public void set_p_tors(String tor1,String tor2){
            p_tor1.add(tor1);
            p_tor2.add(tor2);
   }// method set_tor

}//class Generate_Node
```

Class Rand

```
import java.util.ArrayList;
public class Rand {

  public static int triangle(int beg, int mean, int end, double ilk_frequency){
        int ok=0;
        double ran = Math.random();
        double alt, ust;

        for(int a=0;a<(mean-beg);a++){
              alt=(double)a*4*ilk_frequency/(mean-beg);
              ust=(double)(a+1)*4*ilk_frequency/(mean-beg);
              if(ran<ust&ran>=alt)      ok= a+1+beg;
        }

        for(int a=0;a<(end-mean);a++){
              alt=(double)a*(1.0-ilk_frequency)/(end-mean)+ilk_frequency;
              ust=(double)(a+1)*(1.0-ilk_frequency)/(end-mean)+ilk_frequency;
              if(ran<ust&ran>=alt)      ok= mean+a+1;
        }


        return ok;
  }// method triangle

  public static int tombala (ArrayList tops){
        double rastgele=Math.random();
        int sansli=-1;

        if(tops.size()!=0){
              int[]toplar=new int[tops.size()];
              for (int a=0;a<toplar.length;a++){
                    toplar[a]=Integer.parseInt(""+tops.get(a));
              }

              for(int a=0;a<toplar.length;a++){

  if((rastgele>=(double)a/toplar.length)&(rastgele<(double)(a+1)/toplar.length)){
                          sansli=toplar[a];
                    }
              }
        }

        return sansli;
  }//method tombala


  public static int duration(int ilk, int son){

        int duration=ilk+(int)(Math.random()*(son-ilk));
```

```
            return duration;
    }// method duration
}// class Rand
```

APPENDIX 9

JAVA CODE FOR THE HEURISTIC

The classes and methods for the heuristic is the same as the DP codes given in Appendix 6 except the method hierarchy in the class Result. Hence, we give only this method.

Method Hierarchy

```
   public static void hyerarchy (int tp){

            ArrayList set = new ArrayList();
            ArrayList temp_set = new ArrayList();
            boolean bol = true;
            int iter_count,iter_count_temp, tin_par;

            nodes[0]=new Node(NUM,AOG);
            set.add(""+NUM);
            tin_par=tp;
            iter_count_temp=Math.min(nodes[0].lastnodes.size(),tin_par);


            while(set.size()!=0){
                        iter_count=iter_count_temp;
                        iter_count_temp=0;
                        for (int h=0;h<iter_count;h++){
                        for(int a=0;a<set.size();a++){

   if(nodes[Integer.parseInt(""+set.get(a))].matrix.length!=2&nodes[Integer.parseInt(""+set.get(a))].las
tnodes.size()>=h+1){
                                                if(temp_set!=null){
                                                        for(int c=0;c<temp_set.size();c++){

   if(check_equivalence(nodes[Integer.parseInt(""+temp_set.get(c))].matrix,Graph.graph_form
(nodes[Integer.parseInt(""+set.get(a))].matrix,
Integer.parseInt(""+nodes[Integer.parseInt(""+set.get(a))].lastnodes.get(h))))){
```

```
   nodes[Integer.parseInt(""+set.get(a))].set_torun(Integer.parseInt(""+temp_set.get(c)),Integer.parse
Int(""+nodes[Integer.parseInt(""+set.get(a))].lastnodes.get(h)));
                                                            bol=false;
                                                    }
                                            }
                                    }

                                    if (bol){
                                            NUM++;
                                            temp_set.add(""+NUM);

   nodes[Integer.parseInt(""+set.get(a))].set_torun(NUM,Integer.parseInt(""+nodes[Integer.parseInt(""
+set.get(a))].lastnodes.get(h)));
                                            nodes[NUM]=new
Node(NUM,Graph.graph_form(nodes[Integer.parseInt(""+set.get(a))].matrix,
Integer.parseInt(""+nodes[Integer.parseInt(""+set.get(a))].lastnodes.get(h))));

                                            if(nodes[NUM].lastnodes.size()>iter_count_temp)
   iter_count_temp= nodes[NUM].lastnodes.size();
                                    }
                                    bol=true;

                    }
                    if (temp_set.size()>=tin_par)       {
                            a=set.size();
                    }
            }
            if (temp_set.size()>=tin_par)       h=iter_count;
            }
            while(set.size()!=0){
                    set.remove(0);
            }

            for (int a=0;a<temp_set.size();a++){
                    set.add(""+temp_set.get(a));
            }
            while(temp_set.size()!=0){
                    temp_set.remove(0);
            }
    }

   }// method hyerarchy
```