A COMPUTERIZED SEARCH METHODOLOGY FOR THE DESIGN OF MIXED MODEL ASSEMBLY SYSTEMS

by

Pieter R. Smith

Report submitted to the Faculty of the

Virginia Polytechnic Institute and State University

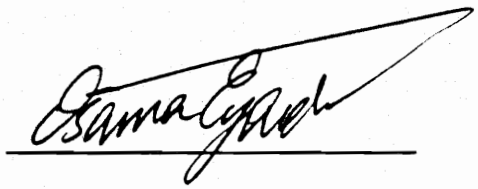
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING

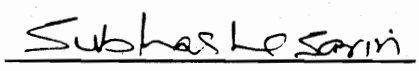
in

Systems Engineering

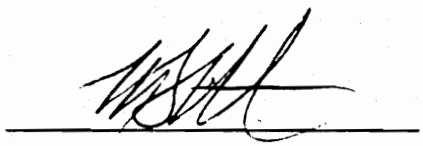
APPROVED:



O. K. Eyada, Chairman



S. C. Sarin



R. T. Sumichrast

December, 1990

Blacksburg, Virginia

A COMPUTERIZED SEARCH METHODOLOGY FOR THE DESIGN OF MIXED MODEL ASSEMBLY SYSTEMS

by

Pieter R. Smith

Committee Chairman: Osama K. Eyada

Systems Engineering

(ABSTRACT)

The majority of the line balancing and model sequencing techniques for the design of mixed model assembly systems are of the sequential search type. Assembly stations are sequentially optimized under some objective function and constraints. This thesis project presents a backtracking exhaustive search algorithm, implemented in Prolog language, for line balancing and model sequencing on mixed model assembly systems. The developed software permits the utilization of different objective functions. For line balancing, either the minimization of difference in work content or the smoothing of stations can be investigated. For model sequencing, either the penalty cost method or the minimization of assembly line length can be employed. A real world case problem was utilized to assess the developed methodology against the sequential search techniques and in all cases, better optimums were found.

Acknowledgements

I wish to acknowledge Dr. Osama Eyada for his commitment and patience during the preparation of this project and report. I also thank the other members of my committee and Dr. Ben Blanchard for his support over the past two years.

Finally, I wish to thank my wife Mariëtte for her love and support. Without her sacrifice this degree would not have been possible.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction.

In 1909, the Ford company decided to concentrate exclusively on the 'Model T'. The chassis was to be the same for all cars, and Henry Ford announced: 'Any customer can have a car painted any color that he wants as long as it is black'. Such were the circumstances in which flow-line technology was born. Sixty years ago, for most people, cars were luxury items and there were comparatively few companies making them. It was a manufacturer's market, and companies were able to make what they wanted, how they wanted, and yet still remain in business - a situation which does not exist today [1]. Unfortunately, for manufacturers of automobiles and most other products nowadays, the novelty of the 'basic model' is past and customers demand their own personal mix of requirements.

The modern tendency in manufacture, therefore, is to offer products with large numbers of options [2]. For example, vehicles may be offered with alternatives for color, type of transmission, engine size, axle ratio, grade of upholstery, type of seat and so on. The purchaser specifies the requirement in detail and is promised delivery within a certain time, say two weeks. Manufacture of the order is then undertaken.

Assembly lines are commonly used to assemble consumer durable items such as cars, radios and domestic appliances. Interchangeable parts are assembled together at a set of sequential work stations at each of which a prespecified part of the total work content is performed. Each work station consists of work elements which have been previously derived from a breakdown of the assembly of a particular product.

These work elements are usually defined in such a way that they are indivisible into smaller elements.

The assembly is usually moved mechanically by belt, conveyor or indexing line. Provided that parts are available and that demand for the product is adequate, these assembly lines can be highly efficient. The well defined sequence of operations minimizes the need for control documentation. Manual handling is reduced. Work in progress is small and as each operator is responsible for only a limited amount of the work, training time can be reduced and semi-skilled labor used.

1.1 Types of flow lines.

In both nonmechanical and moving conveyor lines, it is highly desirable to assign work elements to the work stations so as to equalize the process or assembly times at the work stations. The problem is complicated by the fact that the same production line may be called upon to process more than one type of product. This complication gives rise to the identification of three flow line cases (and therefore three different types of line balancing problems). The three production situations on flow lines are defined according to the product or products to be made on the line. Buxey et al. [3] have characterized the following types of production lines:

- Single model line. This is a specialized line dedicated to the production of a single model or product. The demand rate for the product is great enough that the line is devoted 100% of the time to the production of that product.
- Multi model line. This line is used for the production of two or more models.

Each model is produced in large batches on the line. The models or products are usually similar in the sense of requiring a similar sequence of work elements or assembly operations. The station configuration, however, is unique to each model so that the tasks must be reassigned to stations whenever the production changes over from one model to another.

- Mixed model line. This line is also used for the production of two or more models, but the various models are intermixed on the line so that several different models are being produced simultaneously rather than in batches. Automobile and truck assembly lines are examples of this case.

In the case of the multi model line, if the batch sizes are very large, the multi model line approaches the case of the single model line. If the batch sizes become very small (approaching a batch size of 1), the multi model line approximates to the case of the mixed model line.

In principle, the three cases can be applied in both manual flow lines and automated flow lines. However, in practice, the flexibility of human operators makes the latter two cases more feasible on the manual assembly line. It is anticipated that future automated lines will incorporate quick changeover and programming capabilities within their designs to permit the multi model, and eventually the mixed model, concepts to become practicable [4].

1.2 Mixed model line design.

This research project deals with the design of mixed model lines. The advantage of

this type of production is that, unlike multi model lines, a steady flow of models is produced to meet customer requirements without the need for large inventories of finished products or parts. Another advantage is that mixed model lines eliminates downtime for line changeover, and thus provides greater flexibility in production schedules. Thomopoulos [5] showed that a mixed model line can be designed to be used as a multi model line without major shifts in station assignments, thus providing the best of both worlds. The disadvantages are the uneven flow of work due to the difference in work contents of the models. Another disadvantage is that this type of assembly line undoubtedly presents the most complex design and operating problems [1].

The design of efficient mixed model lines is a problem of considerable complexity. The decisions necessary in the design and operation of a mixed model line are as follows [1] :

- Which models are to be made on each line?
- How are work elements to be allocated to stations?
- Should any station be paralleled or duplicated?
- What will be the method of operation of the line and will buffer stocks be used?
- How, and in what order, will the different models be fed or launched onto the line?

This project looks at both the problem of allocating work elements to stations and the problem of model launching. These two problems are known in the literature as

the line balancing problem and the model sequencing problem.

The line balancing problem is to arrange the individual processing and assembly tasks (work elements) at the work stations according to some objective function without violating any constraints. The objective function can be the minimization of operator idle time or the minimization of the number of stations for example. The constraints are usually a restriction on the sequence of work elements. For example, a threaded hole must be drilled before it can be tapped. A washer must be placed over the bolt before the nut can be turned and tightened. These restrictions are called precedence constraints in the line balancing literature.

The model sequencing problem is to determine the optimum ordering in the flow of models when a variety of models of the same general product is intermixed on one assembly line. The different models typically require different amounts of assembly work, causing an uneven distribution of work along the line and variations in the work load of the individual stations. The objective in finding the optimum can be the minimization of operator idle time, the minimization of work congestion or the minimization of the assembly line length. Operator idle time occurs when an operator is available for work, but no product is available to work on. Work congestion occurs when the product flows through the station with a frequency that prevents the operator from completing the work assignment within the station. In other words, to accomplish the task the operator is forced out of the station.

The efficient design and operation of a mixed model assembly line requires solving two separate but closely related problems, the allocation of work elements

and the sequencing of models. However, since both problems fall into the nonlinear programming class of combinatorial optimization problems, it has consistently defied the development of efficient algorithms for obtaining optimal solutions. The algorithms found in the literature range from a complete enumeration of all possibilities to simulation and heuristic rules. This project is concerned with the development of a computer-efficient approximation or searching algorithm for the design of efficient mixed model lines.

1.3 Work station types.

The following types of work station interfaces are considered [10,21]:

- Closed. All stations are closed, they have fixed boundaries which can not be crossed by the operators (such as spray booth, heat chambers etc.)
- Open. All stations are open, i.e. there are no station boundaries.
- Variable-length station. The station has an upstream boundary and a downstream boundary where the operator can move into.

The open and closed station interfaces are shown in Figure 1.1, which illustrates the movement of adjacent assembly line operators. The distance the operator moves with each product is proportional to the product's service time and is represented in the figure by a solid line. The broken line represents the movement of the operator, after completing a product, to the next product in the sequence. This movement is proportional to the launching interval, the starting time between consecutive units on the assembly line. For Figure 1.1, the launching interval remains constant during the

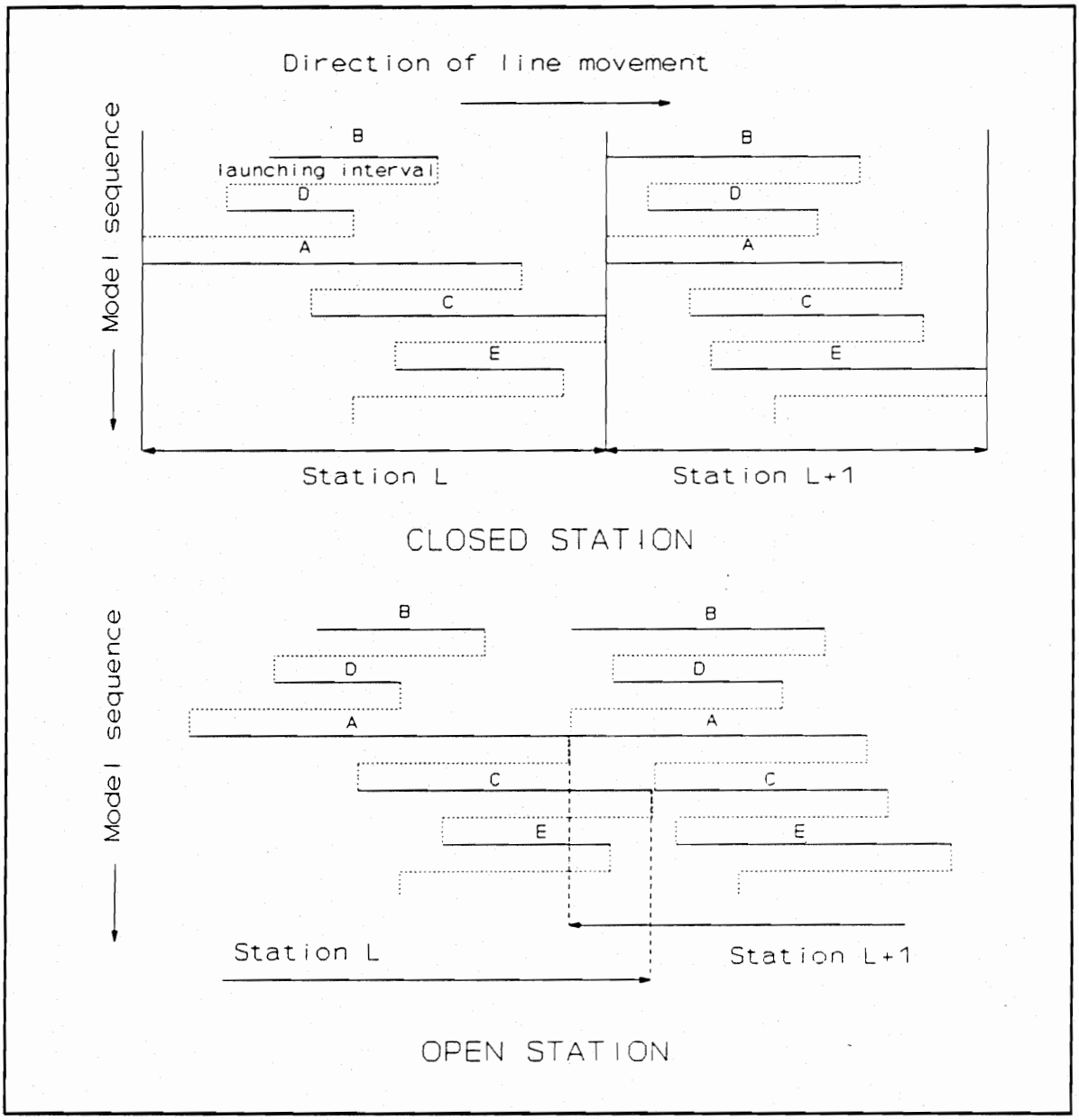


Figure 1.1 : Closed and Open station interface

entire sequence. The variable-length station is illustrated in Figure 1.2. The operator is allowed in the upstream region of the station when no work is available in the desirable station length. Similarly, the operator is allowed to move into the downstream region, if the product could not be completed in the desirable length of the station.

1.4 Prolog.

The computerized search algorithm was developed using Prolog, a fifth-generation language that uses deductive reasoning to solve programming problems. The first official version of Prolog was developed at the University of Marseilles, France by Alain Colmerauer in the early 1970's as a convenient tool for PROgramming in LOGic. It is much more powerful and efficient than most other well-known programming languages, like BASIC and Pascal. A typical Prolog program for a given application will require only one tenth as many program lines as the corresponding Pascal program [6]. Today, Prolog is a very important tool in programming artificial intelligence applications and in the development of expert systems. In contrast to traditional languages, such as BASIC and FORTRAN, which are procedural languages, Prolog is a declarative language. Given the necessary facts and rules, the program will use it's own reasoning mechanism to satisfy a goal or objective. In procedural languages, the programmer must provide step-by-step instructions that tell the computer exactly how to solve a given problem. A Prolog programmer only needs to supply a description of the problem and the ground rules for solving it.

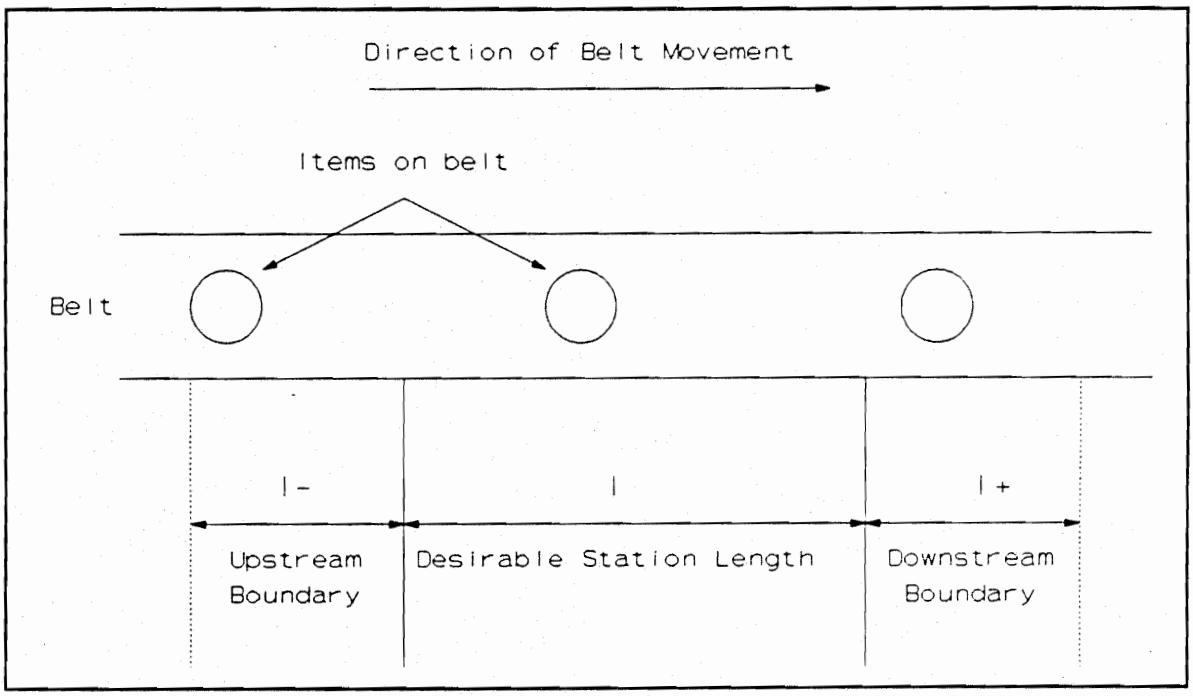


Figure 1.2 : Regions of a variable-length station

For some algorithms discussed in this project, which involves searching methods or the generation of all possible combinations, Prolog was indispensable. In solving these types of problems, a path must be pursued to its logical conclusion. If this conclusion does not give the desired answer, an alternate path must be chosen. For example, a sure way to find the end of a maze is to turn left at every fork in the maze until a dead end is reached. At that point, it is necessary to back up to the last fork, and try the right-hand path, once again turning left at each branch encountered. By methodically trying each alternate path, the right path would eventually be found.

Prolog uses this same backing-up-and-trying-again method, called backtracking, to find a solution to a given problem. As Prolog begins to look for a solution to a problem (or goal), it might have to decide between two possible cases. It sets a marker at the branching spot (known as a backtracking point) and selects the first subgoal to pursue. If that subgoal fails (equivalent to reaching a dead end), Prolog will backtrack to the backtracking point and try an alternate subgoal. This backtracking method simplifies the program code needed for a lot of the searching algorithms examined in this work. Prolog will not only find the first solution to a problem, but is capable of finding all possible solutions. However, the built-in backtracking mechanism can result in unnecessary searching and in turn inefficient programs. Fortunately, Prolog provides techniques to control the backtracking so that these inefficiencies can be eliminated.

Another advantage of Prolog is its implementation of recursion which is very memory efficient. The Prolog version used for this project, Turbo Prolog 2.0, has the

additional benefit that it runs on a personal computer and includes all the features mentioned above.

1.5 Research objective.

The system life cycle and the systems engineering process begins with the identification of a need based on a desire for some item arising out of a perceived deficiency. An individual or organization identifies a need or a function to be performed and a new or modified system is procured to perform that function [7]. In this case the need is for an organization to manufacture several models of the same basic model on the same assembly line. This need may be the result from a bigger need or requirement such as the desire to reduce in process inventory, floor space or labor.

The aim of this research project is to examine methods to find a preferred approach for such an assembly line. Preferred does not imply optimal, as the operational requirements may not permit true optimization, but it does imply the best among a number of alternatives within the given constraints. The selected system configuration must be defined in terms of technical performance characteristics and effectiveness factors.

The operational requirements for the system are determined by the production capacity, the shift duration and the work elements to be accomplished on each model. The system will be subject to certain constraints such as precedence constraints for the work elements and other constraints such as assembly line length, number of operators, etc. Within these constraints, the algorithms or tools examined

in this project can be applied to find a number of design configurations that will satisfy the specified requirements. The problem is to select the best approach according to the evaluation criteria. The evaluation criteria in this case can be the total idle time, overall line length, variation in station times, etc.

The research problem can, therefore, be defined as the assessment of some line balancing and scheduling algorithms. The algorithms will be assessed in terms of their efficiency, suitability for this need and ease of computerization. The algorithms best suited for this need will then be computerized to serve as a tool for designing mixed model assembly lines. During this process, a backtracking exhaustive search algorithm implemented in Prolog language was also developed and assessed against the other techniques.

Chapter 2: Literature survey.

As shown in Figure 2.1, the Assembly Line Balancing (ALB) problem and the accompanying research and literature can be classified into four categories: Single Model Deterministic (SMD), Single Model Stochastic (SMS), Multi/Mixed Model Deterministic (MMD), and Multi/Mixed Model Stochastic (MMS). The SMD version of the line balancing problem assumes dedicated, single model assembly lines where the task times are known deterministically and an efficiency criterion is to be optimized. This is the original and simplest form of the assembly line balancing problem. When other restrictions are introduced (e.g., parallel stations, zoning restrictions) the problem becomes the general case. The SMS problem category introduces the concept of task-time variability. This is more realistic for manual assembly lines, where workers' operation times are seldom constant [8]. With the introduction of stochastic task times many other issues become relevant, such as station times exceeding the cycle time (and the production of defective or unfinished parts), pacing effects on workers' operation times, station lengths, the size and location of inventory buffers, launch rates, and allocation of line imbalances.

The MMD problem formulation assumes deterministic task times, but introduces the concept of an assembly line producing multiple products. Multi model lines assemble two or more products separately in batches. In mixed model lines, single units of different models can be introduced in any order or mix to the line. Since for a batch size of one, the two definitions overlap, it is convenient to consider both

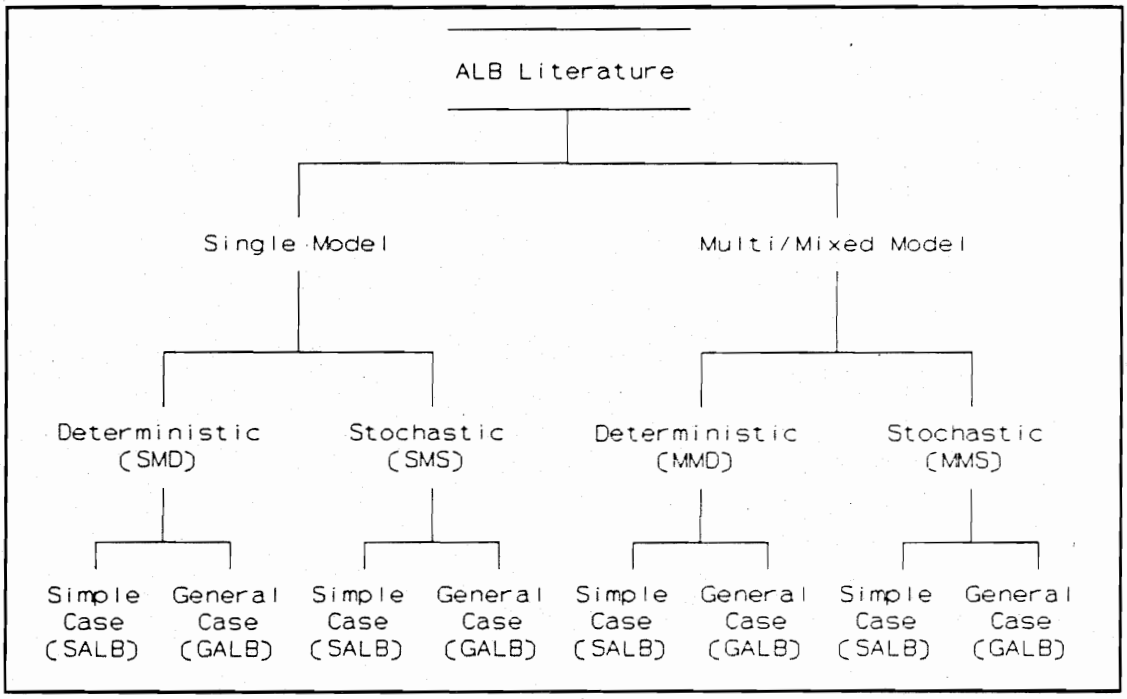


Figure 2.1 : Classification of assembly line balancing literature

types within a single category. Multi/mixed model lines introduce various issues that are not present in the single model case. Model selection, models sequencing and launching rates, and model lot sizes become more critical issues than in the single model case. The MMS problem perspective differs from its MMD counterpart in that stochastic times are allowed. All factors arising from stochasticity that are relevant in the SMS problem also become relevant here. However, these issues become more complex for the MMS problem because factors such as learning effects, worker skill level, job design and worker task-time variability become more difficult to analyze because the line is frequently rebalanced for each model assembled.

The following literature survey will concentrate on the mixed model problem with deterministic times. Although the model sequencing problem is an integral part of the line balancing problem, it is often treated separately in the literature. In turns, this survey will first consider the literature for line balancing and then for model sequencing. Finally the computer programs available for solving line balancing problems will be surveyed.

2.1 Definitions and discussion of terms.

The following is a brief review of single and mixed model line balancing definitions and terms. This is necessary to ensure that a consistent notation and approach are maintained for all the literature reviewed. The single model line notation is first defined because the mixed model notation is a natural extension from the single model line.

2.1.1 Single model line balancing.

Consider a single model assembly line where the production unit consists of K work elements, and where t_k (k = 1, 2, ..., K) represents the elemental process times. The time required to assemble one unit is called the total work content time and is given by:

$$ttwc = \sum_{k=1}^{K} t_k \tag{1}$$

Associated with the K elements are precedence relationships which usually are depicted by a precedence diagram. These relationships define the ordering amongst the elements that are possible. They state, for example, which elemental tasks or work elements must be completed prior to the start of any given elemental task..

If N units are to be assembled in a shift time duration T, the cycle time c becomes:

$$c = \frac{N}{T} \sum_{k=1}^{K} t_k \tag{2}$$

Hence, c represents the desired amount of load time for each station. If n represents the number of stations on the line, and p_i (i = 1, 2, ..., n) the amount of time assigned to the i^{th} station for each unit (called the operation or station time), then:

$$\sum_{i=1}^{n} p_i = \sum_{k=1}^{K} t_k \tag{3}$$

The objective of line balancing algorithms is to assign elements to stations in such a manner as to strictly adhere to all precedence restrictions and to minimize the idle

time associated with a set of station assignments, i.e. minimize:

$$f(n,p_i) = \sum_{i=1}^{n} (c - p_i) \tag{4}$$

where $c \geq p_i$, i = 1, 2, ..., n. Minimizing equation (4) is equivalent to minimizing the number of stations, the cycle time, or the product of the two, depending on what is held constant [5]. In many situations, equation (4) is not used directly. Instead, the operation times are sought such that:

$$c_L \leq p_i \leq c_H \quad (i = 1, 2, ..., n) \tag{5}$$

where the boundary defined by (c_L, c_H) represents an acceptable neighborhood of the operation times. The p_i are never allowed to exceed c_H, but may be less than c_L since it is not always possible to find solutions within the given boundary. Although some authors have obtained procedures which yield optimum solutions [14,15,16], they agree that the methods are frequently not practical from an application standpoint and are primarily of academic interest. The usual practice is to assign elements to stations in a serial fashion [11,12]. This serial approach has been found to yield near optimum solutions on large and complex assembly systems in a relatively small amount of computer time [5].

2.1.2 Mixed model line balancing.

The number of different models to be assembled is designated by J, and the schedule quantity of each to be assembled in time period T is denoted by N_j (j = 1, 2, ..., J).

The elemental times are given by t_{jk} (j = 1, 2, ..., J; k = 1, 2, ..., K), where t_{jk} represents the work time of element k on model j. The total time required to complete all units in the scheduled period for element k becomes:

$$\hat{t}_k = \sum_{j=1}^{J} N_j t_{jk} \tag{6}$$

The station times (consisting of the station work load for time period T for all models) are denoted by T_i. Hence:

$$\sum_{i=1}^{n} T_i = \sum_{k=1}^{K} \hat{t}_k \tag{7}$$

The mixed model line balancing problem can then be stated as follows: elements are assigned to stations in such a manner that all precedence restrictions are enforced and where the T_i (i = 1, 2, ..., n) are sought in order to minimize:

$$f(T_i, n) = \sum_{i=1}^{n} (T - T_i) \tag{8}$$

where $T \geq T_i$ (i = 1, 2, ..., n). As in single model assembly, equation (8) is often replaced by its counterpart:

$$T_L \leq T_i \leq T_H \quad (i = 1, 2, ..., n) \tag{9}$$

where (T_L, T_H) designate the neighborhood of desirable station times.

The precedence restrictions associated with the mixed model situation must again adhere to all models. A combined precedence diagram is usually generated from the precedence diagrams of the respective models. The procedure for this is given by

Macaskill [9].

A precedence diagram G_j (j = 1, 2, ..., J) is associated with each model j. G_j has a set of nodes N(j) and arcs L(j) where:

$$N(j) = \{ n(j)_1, n(j)_2, \ldots \} \tag{10}$$

$$L(j) = \{ l(j)_1, l(j)_2, \ldots \} \tag{11}$$

The nodes represent tasks and the arcs precedence relations. Thus if an arc $l(j)_h$ has initial node $n(j)_i$ and terminal node $n(j)_k$, then the task represented by $n(j)_i$ must be completed before the task $n(j)_k$ may start. Precedence relations for a set of models M = {1, 2, ..., J} are characterized by directed graphs G_1, G_2, ..., G_J. These precedences may be defined by a single graph G_M with nodes N(M) and arcs L(M) where:

$$N(M) = N(1) \cup N(2) \cup \ldots \cup N(J) \tag{12}$$

$$L(M) = L(1) \cup L(2) \cup \ldots \cup L(J) \tag{13}$$

G_M is called the combined precedence diagram. An arc $(n(M)_i, n(M)_j)$ in G_M is redundant if in addition to the arc itself there exists a chain of arcs from $n(M)_i$ to $n(M)_j$. Redundant arcs may be omitted. The combined precedence diagram G_M is feasible only if there are no conflicts in the precedence constraints across models. The procedure is illustrated in Figure 2.2.

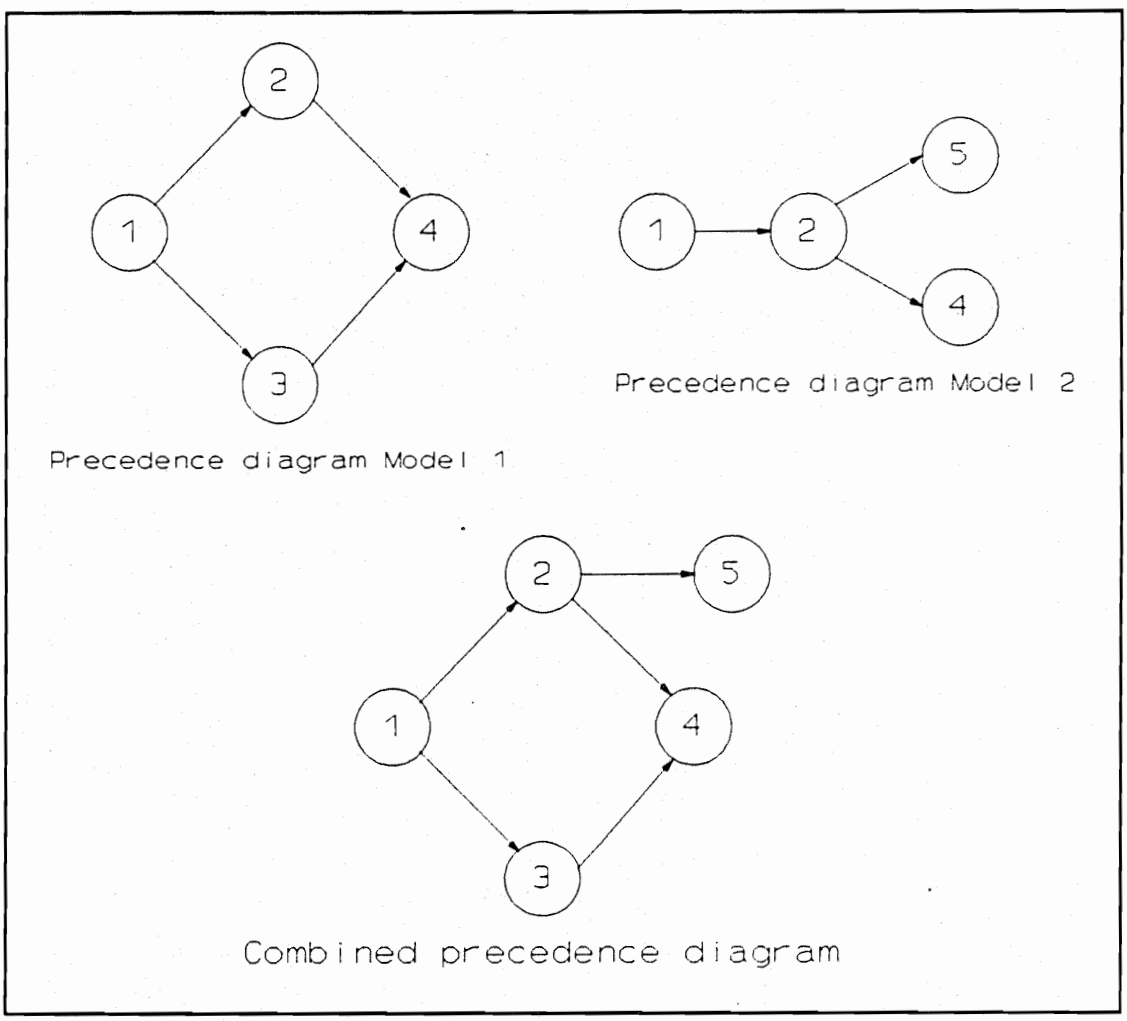


Figure 2.2 : A combined precedence diagram for two models

2.2 Line balancing algorithms.

Line balancing refers to the procedure of assigning work to assembly operators or work stations in such a manner as to allocate the work elements without violating precedence constraints and optimizing some object function (e.g., minimize idle time or line length). In balancing the mixed model line, it would seem possible to consider each model independently, and consequently to balance the work among operators for each separate model. This procedure reduces the larger problem of balancing a mixed model line to a number of smaller single line balancing problems. This was the preferred approach in earlier papers [13]. Unfortunately, this approach leads to serious difficulties [10]. Since an assembler is trained to perform a job which requires some skill, it is desirable, if not imperative, to assign jobs of a specified class to one operator or at most to a small group of operators. This means that when stations are assigned elements, the operator is responsible for those elements on all models entering the station. This consideration is mandatory in most assembly lines. It minimizes parts storage problems, tooling requirements and learning allowances [5].

The work by Thomopoulos [10] was a significant contribution to the mixed model line balancing problem. He introduced a method to solve the balancing problem that considers the total schedule for a whole shift and assigns work elements to operators on a shift basis rather than on a cycle time basis, as is done on single model lines. This approach ensures that one operator will do the same operation on all models. Thus all procedures for single model lines that assign work on a cycle time basis can be adapted to mixed model lines. Popular single line methods that fit this category

are the Kilbridge and Wester method [11] and the ranked positional weight method introduced by Helgeson and Birnie [12]. The objective function of these balancing methods is to assign work elements in such a way as to obtain an even distribution of the work load, where each station on the line should have exactly the same work content per shift. The advantage of this approach is that it satisfies the requirement that each element should be assigned to only one operator. Another advantage is that this approach is computationally very efficient and easy to computerize. The disadvantage is that although the total station times are usually very close to equal, the times that each model spend in the station can be uneven. This can lead to difficulties when determining the model sequence.

The problem of uneven model times was addressed by the same author, Thomopoulos, in a subsequent paper [5] where the concept of smoothed station assignments was introduced. The objective was to show how a modification to the mixed model balancing procedure can lead to smoother (or more consistent) station assignments on a model by model basis. Smoother station assignments are desirable in any mixed model assembly process. Individual operators are able to work at a steadier pace and model sequencing inefficiencies are not as severe [10]. An additional advantage is that line balancing becomes plausible for batched assembly line processes. This allows companies to batch units down the line (at cycle times desirable for individual models) without major shifts, if any, in station assignments as batches are altered from model to model.

Thomopoulos redefines the mixed model balancing problem as follows:

Let p_{ij} (i = 1, 2, ..., n; j = 1, 2, ..., J) represent the amount of time that the operator in station i is assigned on each unit of model j. The total time assigned to the i^{th} station becomes:

$$T_i = \sum_{j=1}^{J} N_j p_{ij} \quad (i=1, 2, ..., n) \tag{14}$$

The total time assigned to station i on model j in period T is:

$$P_{ij} = N_j p_{ij} \quad (i=1, 2, ..., n; j=1, 2, ..., J) \tag{15}$$

Although previous methods had a high degree of success in balancing the T_i values, the methods usually ignored the resulting relationships of the p_{ij} or P_{ij} (i = 1, 2, ..., n; j = 1, 2, ..., J). Ideally, for a specific model j, it is desirable to minimize the fluctuations in p_{ij} or P_{ij} over all i (i = 1, 2, ..., n). However, because of the variation in the work content times of each model, it is not possible to eliminate fluctuations in p_{ij} or P_{ij} for all j in a specific station i. The objective of mixed model line balancing becomes threefold: (1) to adhere to the precedence restrictions of the elements, (2) to seek T_i (i = 1, 2, ..., n) which are desirable with respect to equation (8) or (9), and (3) to obtain operation times which minimize:

$$\Delta = \sum_{i=1}^{n} \sum_{j=1}^{J} |P_j - P_{ij}| \tag{16}$$

where

$$P_j = \frac{N_j}{n} \sum_{k=1}^{K} t_{jk} \quad (j=1, 2, ..., J) \tag{17}$$

P_j represents the average or desired amount from the total work content for model j assigned to each station. Hence, minimizing equation (16) tends to smooth out or equalize the total work load for each model over all the stations.

The disadvantage of this approach is that no exact procedure to obtain optimum solutions that satisfy the above three objectives is known. Thomopoulos then proposed a serial solution which yields near optimum results in a relatively small amount of computer time. This modified line balancing procedure is also directly applicable to batched assembly systems. Station assignments are made so that operators are given the same tasks on all models. This will eliminate the practice, due to the inability to obtain smooth (or consistent) station assignments from model to model, to switch element assignments from one station to another as batches of models are introduced [5].

Macaskill [9] presented balance procedures designed to obtain expeditious computer solutions of large-scale mixed model systems. The performance of a computer program that calculates aggregated task group balances for mixed model lines was investigated. The program used two simple procedures for assigning the work elements to the stations, namely the ranked positional weight (R.P.W.) [12] and the largest candidate technique [17]. Macaskill found in his preliminary investigation that effective measures to smooth the assignments to models within stations without significant degradation of the balance would require considerable computing effort

with an accompanying loss of computational speed. His work was therefore confined to balances obtained without measures to smooth the assignments. Macaskill also recommended the use of special storage techniques in order to achieve high computation speed. A specially formulated matrix was used to store precedence data in the form of bit positions within computer words. He finally concluded that low station/task ratios will often give balances of acceptable efficiency, even with crude balance methods. However, an immediate difficulty of the method is that tasks for a given model will often be shared unevenly between the stations. This effect will become more marked as differences in model work content increase. This unevenness in assignment tends to reduce overall performance of the production line, to increase line length, and to increase sensitivity to the sequence in which products are fed to the line. However, these effects can be limited if the product sequence can be controlled.

None of the previous methods attempted to incorporate in-process inventory costs or line set-up costs. Depending on the environment and the similarity of the models, in-process inventory and set-up costs may be significant if the line is not balanced by taking these costs into account. Chakravarty and Shtub [18] presented algorithms designed to minimize total production costs, composed of labor costs, inventory carrying costs, and setup costs. The algorithms use the concepts of echelon inventory and echelon holding costs. The assembly system is then transformed to a serial system which can then be exploited efficiently for task grouping using a shortest-path algorithm or a heuristic procedure. The advantage of this approach is that operating

costs are accounted for. The method is however difficult to computerize because the algorithm is based on a dynamic programming model. The authors also presented a simpler heuristic procedure but only suboptimal solutions will be generated.

2.3 Model sequencing algorithms.

The mixed model line sequencing problem is to determine the best ordering in the flow of models when a variety of models of the same general product is intermixed on one assembly line. The different models usually require different amounts of assembly work causing an uneven distribution of work along the line and variations in the work load of the individual stations. A sequence of high-work-content models leads to overloading of stations, while a sequence of low-work-content models produces large idle times per stations. A survey by Kilbridge and Wester [13] showed in 1964 that the automotive industry wasted on the average about 25% of the assembler's time through uneven work assignments.

Kilbridge and Wester [13] presented the first general statement of the sequencing problem and proposed two analytical systems for handling it. The objective of their approach was to sequence the models down the line so as to minimize the idle time of operators and the effect of work congestion. Work congestion occurs when an operator is forced out of the station in order to complete work on a product. In other words, the objectives of their approach are the efficient use of assembly manpower and the prevention of bottlenecks on the line. To achieve this, two sequencing criteria are proposed. The primary criterion is that when any operator is available for

work, a unit of product must also be available for him to work on. Operators should not be kept idle waiting for work to enter their stations. On the other hand, the congestion of work in any station is to be avoided. This occurs when products flow through a station faster than the operator can complete the work on them. Since the conveyor moves at a uniform pace, either the operator is forced out of the station to complete the work, or in extreme cases the lines must be stopped until the operator catches up. To avoid this, whenever possible, the sequencing system must satisfy a secondary criterion, namely that each operator's work should be performed within the work station.

The first method proposed to accomplish these objectives is the variable-rate launching, in which the time interval between the starting of successive units down the line is made proportional to the total work content time of the units. The great advantage of variable-rate launching is that units can be put into production in any order and yet cause no operator idle time or work congestion on the line. This launching discipline ensures that stations are kept busy, but only at the expense of high work in progress. There are, however, some serious practical difficulties associated with its introduction in most industries. Considerable replanning of related activities, such as the scheduling of components and integration with other production lines may be required. This sequencing discipline is more appropriate for non-mechanical lines with interstation buffer facilities [1] than for moving-belt lines, since, in the latter, many of the units may be carried some way past a station before being worked on at the station.

The second method is the fixed-rate launching, an algorithmic procedure that can be easily programmed. With this method successive units are carefully sequenced and started down the line at regular intervals. Under certain conditions a cyclic, or carousel pattern can be used, a special case of fixed rate launching. This method assumes that the assembly work of each model can be evenly divided among the n operators so that the amount of time each operator works on a given model is the same. Otherwise stated, the line balancing should have been obtained by an approach similar to Thomopoulos' method of smoothed station assignments. The time for each model is called the model cycle time and designated by c_j. It can be found for each model by:

$$c_j = \frac{\sum_{k=1}^{K} t_{jk}}{n} \quad (j=1, 2, ..., J) \tag{18}$$

The model having the maximum total amount of assembly work will also have the maximum model cycle time for a given n. The maximum model cycle time will be denoted by σ. The launching rate γ can be determined by:

$$\gamma = \frac{\sum_{j=1}^{J} N_j c_j}{\sum_{j=1}^{J} N_j} \tag{19}$$

Thus, the fixed interval between the launching of successive units is the weighted average of the model cycle times. At each step i of the sequencing procedure, the c_i should be chosen such that the following equation is satisfied:

$$0 \leq \sum_{h=1}^{i} c_h - i\gamma \leq \sigma - \gamma \qquad (20)$$

The inequality on the left is to avoid operator idle time and the one on the right is to prevent congestion. Hence, to avoid both operator idle time and work congestion, the double inequality must hold for each step i. In some cases it will not be possible and depending on the environment and other factors, a compromise should be made.

Fixed-rate launching has the advantage of providing a uniform rate of production. It also adapts itself more easily than does variable-rate launching to industrial situations. According to Wild [1], only the fixed-launch rate system appears to be practical. The disadvantage is that it requires very careful sequencing of models. However, the algorithm can be easily implemented on a computer. A much bigger disadvantage of Kilbridge and Wester's approach is the assumption that the work can be divided among the operators so that the amount of time each operator works on a given model is the same. Even with the best of balancing algorithms, this will not be possible even for small problems.

Thomopoulos [10] proposed a penalty cost approach for a mixed model line with variable-length stations. Four types of inefficiencies are identified as follows:

- Idleness. Idleness occurs when an operator is kept idle waiting for work to enter the upstream limit of this allowable work area.
- Work deficiency. Work deficiency occurs when products flow through a station so slowly that the operator is able to complete work on a product before the next product has entered the station, and must leave the station to the left to

start assembly of the next assigned product.

- Utility work. Utility work results when products flow through an operator's downstream limit of the work area faster than he/she can complete work on them. In this situation, a utility worker may be assigned to the station to assist the operator so that the work on the product is completed, or else the unfinished work is completed in a touch-up station further down the line.
- Work congestion. Work congestion results when products flow through a station faster than the operator can complete work on them, forcing him/her to move out of the station to the right.

The four types of inefficiencies are illustrated in Figure 2.3. If the physical dimensions of the station is as follows

l_s = length of the s^{th} station (s = 1, 2, ..., n)

$l_s^{(-)}$ = max distance operator in s^{th} station is allowed to move upstream

$l_s^{(+)}$ = max distance operator in s^{th} station is allowed to move downstream

Let v represent the conveyor belt speed and let d represent the fixed length separating two consecutive units on the conveyor. Since γ given by equation (19) is the fixed time separating two consecutive units, d and γ are proportional and v is determined by:

$$v = \frac{d}{\gamma} \tag{21}$$

The inefficiencies resulting from a given model sequence can now be calculated.

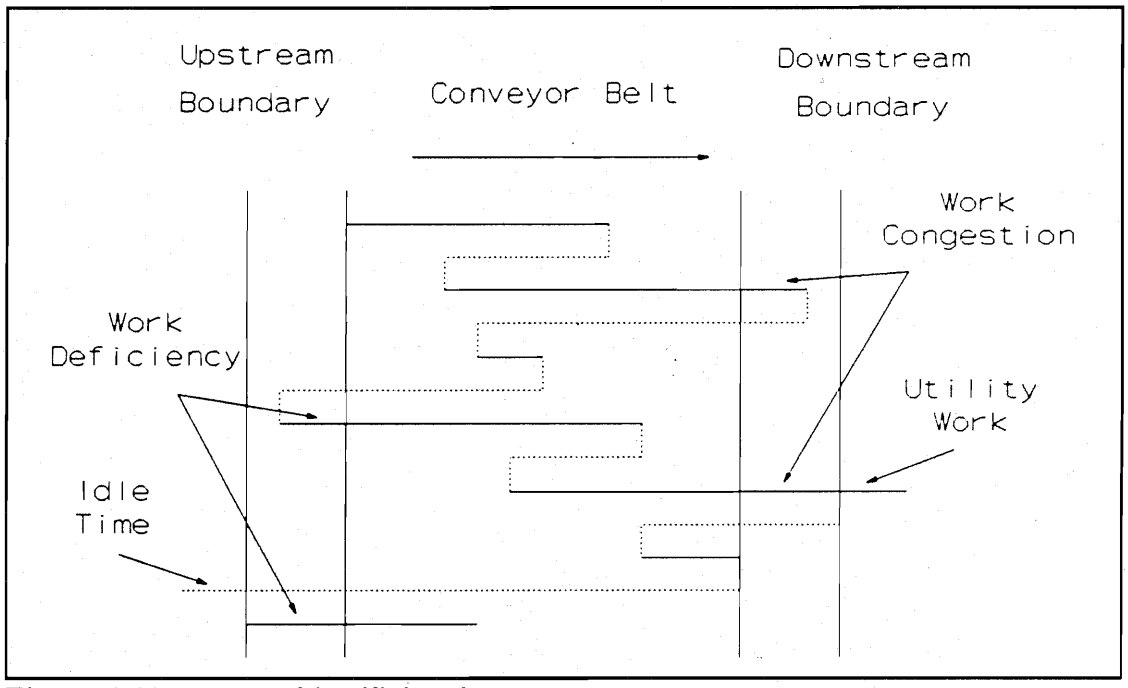


Figure 2.3 : Types of inefficiencies

The various models to be assembled will be sequenced to control the effect of inefficiencies. If a penalty cost is associated with each inefficiency, it would be possible to compute the total cost of inefficiencies resulting from scheduling a unit of a given model in the sequence. A good sequence will then seek to minimize these inefficiency costs over the total day's schedule in all stations. One approach would be to generate all possible combinations of sequences and then select the one with the minimum cost. However, for large production quantities or many different models, this approach might not be feasible. A serial method to determine the model sequence was proposed by Thomopoulos [10]. At each step the total inefficiency cost for each model is computed. The model with the minimum cost is then launched. The procedure must keep track of the number of units launched for each model in order not to exceed the production requirements. Since the sequencing procedure derives sub-optimum rather than optimum solutions, a Monte Carlo analysis was utilized to determine how the solutions would compare with the optimum one. The results were near optimum. However, one of the disadvantages of this method is that very good results are obtained in the beginning of the sequencing process but the inefficiencies increase as the number of units launched increase. The reason for this is that models with uneven station times or high work content will be pushed back until the end and all the 'easy' models will be sequenced first. The net result is a build up of a residue of high penalty models. This problem can be overcome by partitioning the production schedule. Partitioning insures that all units of a model are not clustered in one segment of the sequence, and has the advantage of allowing

different models to be produced more evenly throughout the work day. Another disadvantage of this method is that there is no way to disallow concurrent working. Concurrent working occurs when two operators are working on the same unit. An operator can still be working in his/her station while the next operator has moved into the upstream region and they both work on the same unit.

A slight modification of the Thomopoulos penalty cost approach is presented by Macaskill [19]. A mathematical model of the mixed model assembly line and several algorithms that provide acceptable sequences under various conditions are given. This approach also has an option to either allow or disallow concurrent working. The purpose of this arbitrary assumption is to identify any significant advantage that might accrue through concurrent working. One aim of the author was to examine the effect of product sequence on the line performance. To measure the line performance, the same inefficiencies were adopted as by Thomopoulos [10]. To avoid a build up of high penalty models the sequencing procedure was modified. After a model with minimum cost was selected, the next model chosen was the one with the greatest work content that would not introduce any utility work. If no model was present with zero utility work, the model that gave the minimum utility work was selected. The author also noted that when concurrent work is not allowed, an operator is forced out of the station before the task is finished. The worker in the next station, although free, can't start work because the predecessor worker in the line is still busy. As a result, the second worker starts late and is therefore likely to finish late, perhaps causing a third worker to stand idle and start late, and so on. The

conclusions were that if an assembly is such that concurrent work can be used then it will be advantageous to plan the work to permit concurrency of working to be exploited, and to devise suitable balance methods. In practice there are two main factors that affect concurrency. The first is that normal assembly precedence constraints cannot be violated, and the second that workers must not bother each other. According to the author, it appears that there is considerable scope for introducing concurrency. Another important point made is that unsuitable model sequences seriously degrade assembly performance and it is most important for acceptable sequences to be applied to the line. The paper showed that automatic sequencing algorithms can be developed to do this.

A new formulation of the sequencing problem was proposed by Dar-El and Cother [20]. The objective function was to minimize the overall assembly line-length for no operator interference. According to the authors it is difficult to define the limits to an operator's movements outside the work station - definitions and measurements inherently required in the penalty system [10,19]. Consequently, the arbitrary selection, which directly affects the generated product sequence, is a serious weakness in the solution technique. Furthermore, the literature provides little guidance upon which to choose the physical assembly line length for a given problem. The authors claim that, depending on the choice of line length, any sequence could be made to appear efficient or inefficient. Their formulation of the model sequencing problem is defined as follows: 'Given a production requirement and work load balance, to determine a sequence of products that minimizes the overall assembly

line length required for no operator interference.' This formulation involves no assumptions as to station length, penalties, etc. The overall line assembly is in fact, the objective function whose minimization is sought whilst no idle time, congestion, etc. is designed into the system. The sequencing algorithms described in their paper, are based on the following assumptions:

- The assembly line is considered as a conveyor moving at constant speed with products equi-spaced on the line.
- Models are launched onto the line according to the fixed-launch rate discipline.
- Each station is manned by one operator. Naturally several operators may work at one station providing their respective tasks in no way interfere with each others.
- The total work load for the shifts is divided equally among the operators using some convenient line balancing method.
- The operators perform each operation in the allocated time.
- An operator takes negligible time to move between products.
- Adjacent operators do not service a product simultaneously.
- Steady state conditions prevail, the product-mix does not change between shifts and the line is not empty at the start of the shift.

Two types of work station interfaces are considered. Closed stations which have boundaries that can not be crossed by the operators and open stations that have no station boundaries. An algorithm for each type of station is proposed. The algorithms

basically comprise two heuristics - one for the candidate selection, the other for the assignment to the sequence. The products to be assigned to the sequence are referred to as the pool. Successive products from the pool are added to the sequence until the pool is empty. To select the m^{th} product in the sequence, various products are tried until one is found that satisfies an acceptance heuristic. The order in which the models are tried is determined by a selection heuristic. The selection heuristic is designed to spread the models evenly throughout the sequence. A lower bound limit is placed on every station for the closed station algorithm and on the entire line for the open station algorithm. The acceptance heuristic simulates the service of the selected product. If a station limit is exceeded, the product is rejected and the next model is tried. If the product is accepted, it is assigned to the model sequence and the next unit in the sequence is determined. If at some stage no product satisfies the acceptance heuristic, the station length limits are increased by equal amounts, all products are returned to the pool and the sequencing procedure recommences.

A disadvantage of this approach is that it is best suited for the design of new assembly lines. Once an assembly line is in operation, minor changes in production requirements or in model design might present a problem. If fixed facilities exist such as spray booths, dip tanks, etc. their actual station lengths are used as fixed input design parameters, while the limits for the remaining stations would be determined according to the algorithm. On the other hand, all station line lengths could be retained and the problem reduces to that defined by Thomopoulos [10] and Macaskill [19]. A sensitivity analysis by the authors indicated that three factors mainly influence

the overall assembly line length. These are the number of models, the model cycle-time deviation factor and the operator-time deviation factor. For a given model-mix, it is shown that the assembly line length decreases when better work balances are obtained between operators. The paper also shows that for maximum efficiency in utilization of space, assembly line designers should, wherever possible, use open station interfaces. Such interfaces permit adjacent workers to operate in each others apparent 'work areas' without interfering with one-another's activities. It should be noted that the algorithms in this paper, although not completely serial, will generate only sub-optimum solutions.

A subsequent paper by Dar-El and Cucuy [21] described an algorithm for solving optimally the mixed-model sequencing problem when assembly line stations are balanced for each model. An optimal sequence is obtained with the minimization of the overall assembly line length for zero station idle time. The algorithm incorporates two basic steps. The first involves a search procedure that generates all cycle sequences (sequences having identical 'start' and 'finish' positions and whose work content can be executed within a defined station length). The second step uses integer programming to determine the number and combination of the various cycle sequences such that the production demand is satisfied. The authors define the sequencing problem as follows: 'To sequence the products so that the maximum time the products remain in the station is minimized for zero operator idle time.' A big disadvantage of this method is that it assumes that the total assembly time for each model is equally balanced over all stations. This is very difficult to achieve in practice

even for small problems. The algorithm is also difficult to computerize because of the integer programming problem. However, this paper describes the first successful attempt to optimally solve the mixed model sequencing problem, even though the requirement for equal station balances somewhat restricts its wider application.

Another formulation of the sequencing problem is given by Okamura and Yamashina [22]. The purpose of their paper is to describe a formulation of the sequencing problem in such a way as to minimize the risk of stopping the conveyor under the circumstances of system variability and to develop an efficient heuristic method for large-scale, mixed model assembly lines. Fixed-rate launching is assumed. It is also assumed that a suitable line balance has already been achieved and that the length of each station has been determined. The first step in the algorithm is to obtain the operator movement diagram of the line, which is a diagram with the loci of work starting points for all stations. The locus of the work starting points zigzags within the station and in general consists of several 'mountains' or 'peaks'. The algorithm then presents a number of insertion and interchange methods which attempts to smooth these peaks by changing the sequence order. The disadvantage of this method is that the quality of the final solution depends on the given initial sequence. According to the authors it is desirable to find profitable initial sequences which have a high probability of reaching the optimal sequences with less computation time, but it is extremely complex and difficult to find such initial sequences. Another disadvantage of this method is that it is difficult to computerize.

2.4 Computer software for assembly lines.

At the moment there is no software package dedicated to the design of mixed model lines. Ghosh and Gagnon [8] attributed this to the lack of efficient computational procedures for practical situations. Another problem is that companies may develop solutions that fit their needs but these efforts often go unpublished. These programs are also designed for particular company situations and lack the generality required for the variety of lines in industry. Efficient heuristic single model line packages such as COMSOAL, CALB and NULISP remain the best choice for practitioners.

COMSOAL is an acronym that stands for Computer Method of Sequencing Operations for Assembly Lines. It is a method developed at Chrysler Corporation and reported by Arcus in 1966 [23]. COMSOAL has an option for the mixed model case. For this case the program multiplies the standard task times by the relative frequencies of the different models and uses these average times along with a net cycle time to balance the mixed model line. CALB which stands for Computer Aided Line Balancing [24] can be used for both single model and mixed model lines. To balance the mixed model line, CALB considers the total schedule for a whole shift and assigns tasks to operators on a shift basis rather that on a cycle time basis. The solutions obtained by CALB are described as being nearly optimum [4]. NULISP is a computer program which has been devised at Nottingham University, England after extensive research [25]. The mixed model line balancing problem is handled by specifying a base model and a number of variations for every model in terms of work elements to be added or omitted. NULISP will then balance the line so that on

average each station is not overloaded and that peak loads for certain models can be catered for. The program also provides a variety of algorithms for sequencing the models which give different emphasis between total makespan and balance of work and which enable the production sequencer to determine a sequence to match the current requirements. Unfortunately, as is most often the case with commercial programs, the full extent of these algorithms cannot be discussed due to copyright protection.

Chapter 3: Line balancing.

This chapter describes two line balancing algorithms and their implementation using the Prolog computer language. Different efficiency measurements are used to measure the line performance. The two algorithms are also examined for computational efficiency against the developed backtracking exhaustive search algorithm.

3.1 Approach.

The line balancing problem can be defined as the allocation of work elements to work stations in order to optimize some objective function without violating the precedence constraints. The objective function for single model lines is namely:

$$\min f(n, T_i) = \sum_{i=1}^{n} (T - T_i) \tag{22}$$

This function attempts to load all stations with the same work content but does not take into account the different times for each model. Thomopoulos [5] proposed the following objective function to rectify this:

$$\min f(n, P_{ij}) = \sum_{i=1}^{n} \sum_{j=1}^{J} |P_j - P_{ij}| \tag{23}$$

This function tends to smooth out the total work load for each model over all stations. A third objective function, designed to minimize the sensitivity to the sequencing of models is given by [9]:

$$\min f(n, p_{ij}) = \sum_{i=1}^{n} \left[\frac{1}{n} \sum_{j=1}^{J} p_{ij}^2 - \frac{1}{n^2} \left[\sum_{j=1}^{J} p_{ij} \right]^2 \right] \tag{24}$$

This function minimizes the variance of station times for all models in a given station, but does not take into account the variance of a given model over several stations.

Two computerized algorithms were designed to solve the mixed model line balancing problem by allocating work elements to work stations and optimizing any of the objective functions. The first algorithm is a serial method, while the second is an exhaustive search technique. Both algorithms make use of Prolog's built-in backtracking mechanism. A flow chart for both algorithms is shown in Figure 3.1. The main difference between the two algorithms is that the serial algorithm will not backtrack past the current station. This means that once a station is assigned and the station counter incremented, the work elements assignments for these stations will not be changed. The serial algorithm therefore optimizes one station at a time. In contrary, the exhaustive search technique will explore all possible solutions and will backtrack over past station assignments. The main advantage of the serial approach over the exhaustive search approach is the gain in execution speed. However, the serial method will only produce sub-optimum solutions while the exhaustive search technique will guarantee near optimal solutions. Another important aspect is that the exhaustive search method can find the optimal solution for all the objective functions at the same time, while one specific objective function must be specified in advance

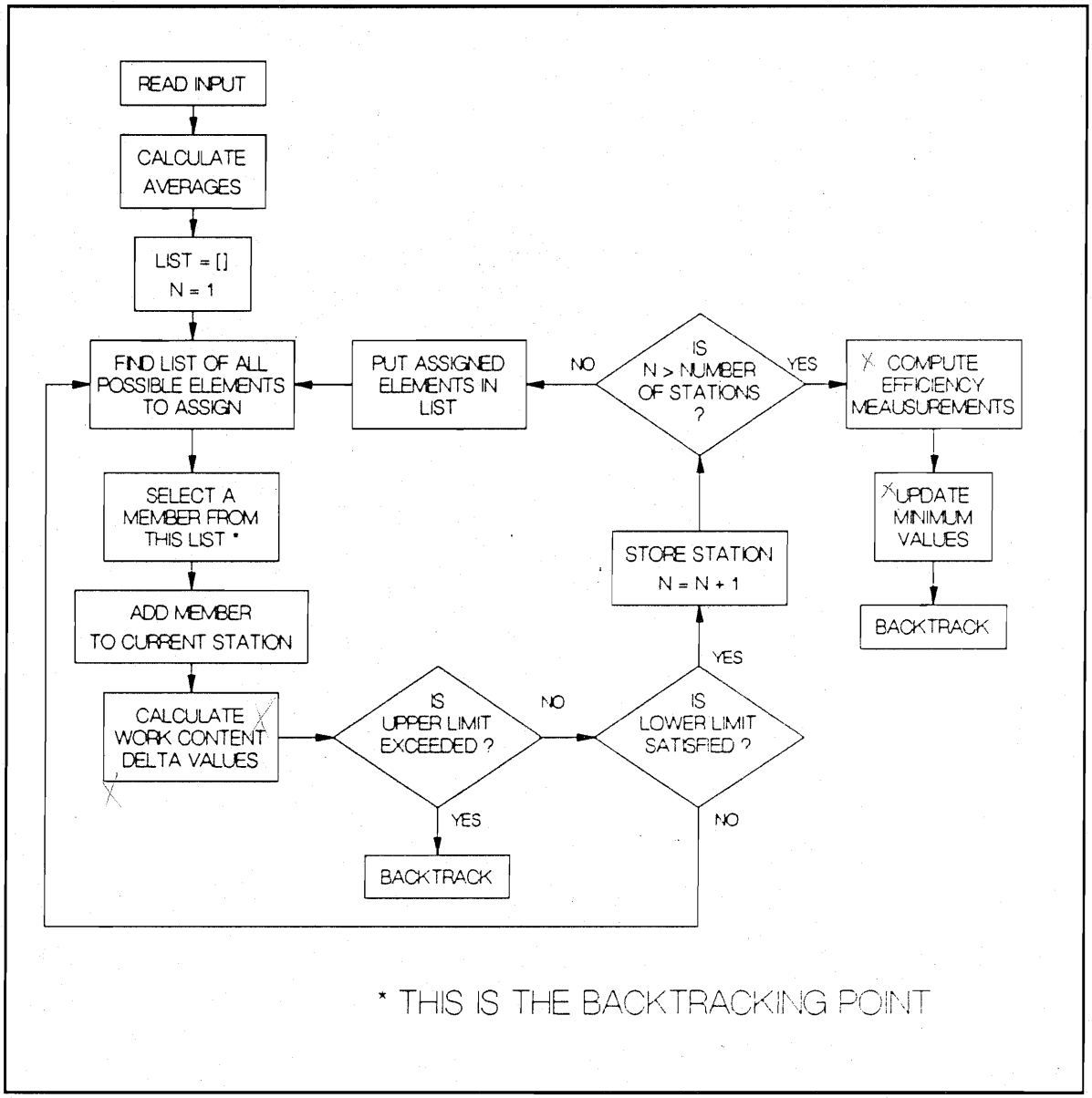


Figure 3.1 : Flow chart for line balancing algorithm

for the serial method. However, depending on the problem size (e.g., number of stations, number of models, etc.) and the computer hardware, the exhaustive search technique may prove too time consuming. Some techniques are proposed later in this report to handle this limitation.

3.2 Data input.

In addition to the combined precedence diagram, the following input data is required for both algorithms:

- n : number of stations
- J : number of different models to assemble
- K : number of work elements
- t_{jk} : duration of work element k for model j
- T : shift duration
- T_L : lower acceptable limit for station work load (%)
- T_H : upper acceptable limit for station work load (%)
- N_j : quantity of model j to be assembled in time T

The first point to notice is that n is specified as an input parameter, while at the same time it appears as one of the variables to optimize in equations (22), (23) and (24). The reason for this is that the value of n must be known to evaluate the objective functions in equations (23) and (24). Therefore, an iterative approach is needed. The smallest possible value for n can be calculated, if the shift time T is known, as follows:

$$n_{min} = \frac{ttwc}{T} = \frac{1}{T}\sum_{j=1}^{J} N_j \sum_{k=1}^{K} t_{jk} \qquad (25)$$

The value of n_{min} is rounded off to the next highest integer. However, due to precedence constraints, this minimum value will rarely be achieved. The algorithm should then be tried with incrementing values of n, starting at n_{min}, until a line balance is found. The acceptable limits for the station work load, T_L and T_H, are expressed as percentage deviations from the shift duration T.

3.3 Data representation in Prolog.

Unlike traditional computer languages like BASIC and FORTRAN, a Prolog program consists of facts and rules. These phrases are known in Prolog as clauses. A fact represents one single instance of either a property of an object or a relation between objects. A fact is self-standing, Prolog doesn't need to look any further for confirmation of the fact, and the fact can be used as a basis for inferences. Rules are dependent relations, and they allow Prolog to infer one piece of information from another. Consequently, while the data would be represented by array variables in traditional languages, facts would be used in Prolog. The work elements can be represented by the following fact:

te(k,j,x)

That indicates that the duration of work element k for model j is x time units. The precedence constraints are represented by the following fact:

node(k,list)

where list is a special Prolog data type containing all the immediate predecessors for work element k. A Prolog list is an object that contains an arbitrary number of other objects within it. Lists correspond roughly to arrays in other languages, but unlike an array, the size of a list does not require to be declared in advance. In this case, the list will contain integers representing the work elements to be completed before the element k can begin. For example, the work elements from the combined precedence diagram in Figure 2.2 can be represented as:

node(1,[])

node(2,[1])

node(3,[1])

node(4,[2,3])

node(5,[2])

Similarly, the production quantities N_j are represented by the following facts:

q(j,N)

Because of Prolog's inflexibility in reading data from a computer file, an intermediate program is used to read the data from a file and to write it in a format acceptable to Prolog. This allows for more flexibility in data input.

3.4 Search methodology.

In order to understand the search methodology, a basic perception of lists and procedural programming is necessary. A Prolog list is really a recursive compound object. It consists of two parts: the head, which is the first element, and the tail,

which is a list comprising all the subsequent elements. Consequently, to process lists, recursive algorithms are needed. The most basic way to process a list is to work through it, doing an operation to each element until the end of the list is reached. An algorithm of this kind usually needs two clauses. One of the clauses specifies an operation on an empty list or a list containing one element, and the other specifies an operation on all other lists not in this category.

Consider the following clause from the Prolog program in Appendix A. This clause investigates if a given integer is part of a specified list.

```
member(X,[X|_]).
member(X,[_|T]) :-
    member(X,T).
```

The first clause investigates the head or first element of the list. If the head of the list is equal to the specified integer (X), it can be concluded that the integer is part of the list and the clause will succeed. Since the remainder or tail of the list is of no interest, it is indicated by an anonymous variable (_). If the head of the list is not equal to X, the remainder of the list or tail is investigated. In English, these two clauses can be translated as:

X is a member of the list if X is the first element of the list, or

X is a member of the list if X is a member of the tail.

This is known as a declarative viewpoint [6]. From a procedural viewpoint, the two clauses can be interpreted as:

To find a member of a list, find its head, otherwise find a member of its tail.

These two points of view correspond to the following goals:

member(2,[1,2,3,4]) and **member(X,[1,2,3,4])**

In effect, the first goal tries to determine if something is true and the second tries to find all members of the list [1,2,3,4]. This is one of the major advantages of a language like Prolog. Clauses constructed from one point of view will also work from the other point of view.

The searching methodology used in this algorithm is a combination of recursion and backtracking. The member clause is the heart of the searching algorithms, because it is the backtracking point so that whenever a dead end is reached, the program returns to this point to take an alternative route. Another clause, the route clause which is called recursively, is the other major element in the algorithm. The pseudocode for the exhaustive search algorithm is shown in Table 3.1. The pseudocode for the serial algorithm is very similar.

The syntax of the route clause is as follows:

route(List1,List2,List3,Station #) where

List1 = list of work elements already assigned in previous stations.

List2 = list of work elements considered for assignment in this station.

List3 = maintenance list to avoid permutations of same list.

Station # = station number.

Prolog tries to satisfy the clauses in the order they appear in a program. Whenever the route clause is called, Prolog will first go to the first clause. If the first clause fails (a dead end is reached or some condition is not met), the second clause is tried. If

Table 3.1 : Pseudocode for exhaustive search algorithm

```
route(List1,_,_,_,Last) :-
    find all remaining work elements not assigned in List1
    calculate efficiency measurements
    check limits on work content
    store station information
    backtrack : fail
route(List1,List2,List3,Station #) :-
    if not final station
    calculate efficiency measurements
    check limits on work content
    store station information
    append List2 to List1
    increment station counter
    start search for next station :
        route(List1+List2,Empty,Empty,Station #+1).
route(List1,List2,List3,Station #) :-
    if not final station
    find list of all possible elements to assign
    get a member of the available list :
        member(X,Available elements),
    check for permutation of same station
    append X to List2 (current station list)
    calculate efficiency measurements
    if work content is bigger than upper limit
    backtrack to member
    else goto second route clause
        route(List1,List2+X,List3,Station #).
```

the third clause fails, Prolog will backtrack to the last point where an alternative existed, in this algorithm it will be the member clause.

The third parameter in the route clause is a maintenance list to avoid investigating permutations of the same station. For example, if the station consisting of work elements [2,4,6] was already investigated, there is no need to investigate the station consisting of work elements [4,2,6]. This safeguard measure makes the algorithm more efficient.

The first route clause will only succeed if it is the last station. All remaining elements are found and put in the last station. The work content of the station is then checked against the acceptable limits and if the station is accepted, the station information is stored. At this stage the efficiency measurements are checked against the minimum values found so far and updated if required. Finally, the fail clause causes Prolog to return to the previous member backtracking point.

The second clause is investigated for all stations except for the last station. The main purpose of this clause is to determine if a feasible station has been found. In that case, the necessary lists are updated and the station counter incremented. The route clause is then called again and will transfer control to the third clause, unless the last station is reached, where it will transfer control to the first clause. The second clause will only call the first or third clause, it will never call itself. This clause fails whenever the work content does not satisfy the acceptable lower limit. In other words, not enough work elements have been allocated for this station to be feasible. Prolog will then investigate the third clause to try and add more work

elements.

The third clause causes the backtracking with the member clause. The big advantage of backtracking is that all variables have exactly the same values as before the alternative was investigated. Prolog saves the state of the clause at each backtracking point. This is not memory efficient whenever a large number of recursions have to be made but for this algorithm the effect is not significant. An important aspect of the third clause is that it checks if the upper acceptable limit has been violated. In that case, the program backtracks to the member clause and tries another possible element. Only the upper limit is examined, when the lower limit is violated, control would be transferred back to this same clause and new work elements would be added.

The logic behind the serial algorithm is very similar. The route clause is modified so that the station counter is never incremented. Consequently, the algorithm will examine all possible solutions for the current station only. Once all solutions are examined, the optimum solution for the station, depending on the objective function, is stored and the next station explored. Another clause is used to control the route clause from station to station.

Although this algorithm can be implemented in any computer language including BASIC and FORTRAN, the main advantage is the fact that Prolog provides a built-in backtracking mechanism. This frees the programmer to concentrate on the algorithm itself. In conventional languages, the programmer would have to code the backtracking mechanism via a memory stack. Most of the programmer's effort would

be spent on the implementation of the backtracking method. A further advantage of Prolog is the use of lists where the size of the lists need not be known in advance.

3.5 Assessment of line balancing algorithms.

An evaluation of the algorithms will be made by using an example from reference [5]. Consider a mixed model assembly process where three models (J=3) are to be assembled and where the production schedule calls for $N_1=120$, $N_2=60$, and $N_3=40$ units during each shift. The work elements for the three models are shown in Table 3.2 for a total of K=19. The third column gives the total elemental times over the shift period for all three models. Also, the total work content of the line is shown to be 1242 minutes. The combined precedence diagram for this problem is shown in Figure 3.2. The total shift time T is 414 minutes and the corresponding minimum number of stations n_{min} is 3. Suppose further that station times within the interval (408, 420) are acceptable. The percentages for (T_L, T_H) are therefore (0.145, 0.145). The data file used as input to the Prolog program is shown in Table 3.3. The results for the serial algorithms are shown in Table 3.4. The first solution is the result of the objective function given by equation (23), while the second solution was obtained with equation (22) as an objective function. The execution times were obtained using Prolog version 2.0 and a 10Mhz 80286 personal computer. Although no numeric coprocessor was used, the gain in speed would not be major. The execution times show that the serial algorithms are very fast. The first point to note is that an acceptable line balance can indeed be found with the minimum number of stations

Table 3.2 : Work elements for three models

Elements k	Element times			Total Times
	t_{1k}	t_{2k}	t_{3k}	
1	0.5	0.0	1.0	100
2	0.4	0.8	1.2	144
3	0.0	0.2	0.4	28
4	0.4	0.0	0.0	48
5	0.2	0.2	0.2	44
6	0.2	0.0	0.0	24
7	0.4	0.5	0.6	102
8	0.0	0.5	0.5	50
9	0.4	0.3	0.2	74
10	0.0	0.0	0.2	8
11	0.3	0.3	0.3	66
12	0.1	0.3	0.5	50
13	0.1	0.0	0.1	16
14	0.2	0.2	0.2	44
15	0.7	1.0	1.5	204
16	0.0	0.1	0.0	6
17	0.5	0.5	0.0	90
18	0.3	0.5	0.3	78
19	0.4	0.3	0.0	66
Total	5.1	5.7	7.2	1242

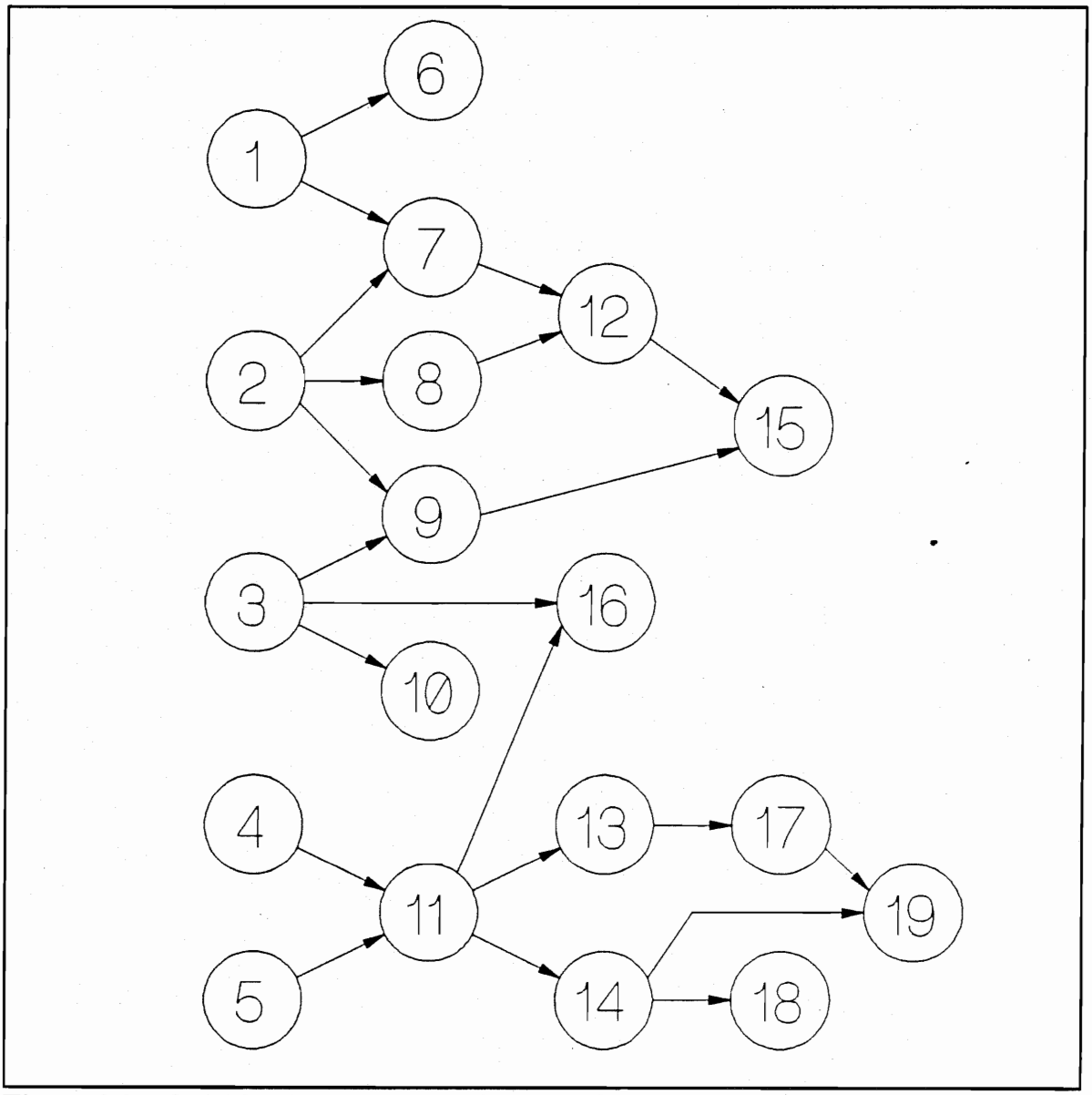


Figure 3.2 : Combined precedence diagram of three models

Table 3.3 : Input data file for the three models example

```
[]
[]
[]
[]
[]
[1]
[1,2]
[2]
[2,3]
[3]
[4,5]
[7,8]
[11]
[11]
[9,12]
[3,11]
[13]
[14]
[14,17]

0.5 0.4 0.0 0.4 0.2 0.2 0.4 0.0 0.4 0.0 0.3 0.1 0.1 0.2 0.7 0.0 0.5 0.3 0.4
0.0 0.8 0.2 0.0 0.2 0.0 0.5 0.5 0.3 0.0 0.3 0.3 0.0 0.2 1.0 0.1 0.5 0.5 0.3
1.0 1.2 0.4 0.0 0.2 0.0 0.6 0.5 0.2 0.2 0.3 0.5 0.1 0.2 1.5 0.0 0.0 0.3 0.0

120
60
40
```

Table 3.4 : Results for the three models example using serial algorithms

Minimize delta : Execution time = 24.33 seconds

Stn	Work Elements	T_i	$T-T_i$	Δ_i	Var_i	P_{i1}	P_{i2}	P_{i3}
1	2 3 4 5 9 10 11 16	418	4	4	0.12	1.7	1.9	2.5
2	1 6 7 8 13 14 18	414	0	24	0.22	1.7	1.7	2.7
3	12 15 17 19	410	4	28	0.03	1.7	2.1	2.0
		1242	8	56	0.37	5.1	5.7	7.2

Minimize difference : Execution time = 25.49 seconds

Stn	Work Elements	T_i	$T-T_i$	Δ_i	Var_i	P_{i1}	P_{i2}	P_{i3}
1	1 3 4 5 11 14 16 18	414	0	48	0.14	1.9	1.5	2.4
2	2 6 9 13 17 19	414	0	72	0.05	2.0	1.9	1.5
3	7 8 10 12 15	414	0	120	0.74	1.2	2.3	3.3
		1242	0	240	0.93	5.1	5.7	7.2

(n_{min}) which is 3 in this case. This can be attributed to the large element/station ratio which results in easier line balances, since the algorithms have a wider range of elements to assign to a given station. In addition, the work elements are comparatively small and the stations can be filled up with small elements until the work content is close to the desired value.

The first two columns in Table 3.4 show the station number and the corresponding work elements assigned to this station. The next column shows the T_i values for each station which is the total work content for that station. Column four shows the absolute value of the difference between the station work content and the shift time T. The next column shows the value Δ_i given by [5]:

$$\Delta_i = \sum_{j=1}^{J} |P_j - P_{ij}| \tag{26}$$

The variance of the station times for each model is shown in column six. Columns seven to nine display the p_{ij} values which represent the amount of time that each unit of model j spends in station i.

The second solution in table Table 3.4 demonstrates that a perfect line balance can be obtained for this problem. However, the uneven model times from station to station become evident when compared with the first solution. The station times for the first model are perfectly distributed in the first solution, where each station does exactly one third of the total elemental time for model one. The times for the remaining two models, although not as well distributed as the first model, are still remarkably better than the second solution. The disadvantage of the first solution is

that the total work content at each station deviates from the shift time although still within the acceptable limits.

A summary of the results from the exhaustive search method is shown in Table 3.5. The algorithm found a total of eight different solutions with a perfect line balance (zero difference), but only the solution with the smallest Δ value is shown in Table 3.5. The algorithm also found the optimal solution to the objective function given by equation (24), but the solution is identical to the first solution in the table. It can be seen from the execution time in Table 3.5 that the exhaustive search requires a lot more computer effort. However, the solutions show a remarkable improvement over the already good solutions from the serial algorithm. In order to gain more perspective, the results are compared with those found by Thomopoulos [5]. He utilized a serial approach but with a finite number of iterations in each station. In turn, not all possible combinations for a station were examined. The comparison, shown in Table 3.6, indicates that both the serial and exhaustive algorithms produced similar or better results than those found by Thomopoulos. However, the superiority of the exhaustive search algorithm is well reflected in determining the two best solutions.

Thomopoulos [5] also mentioned that the line balancing procedure can be applied to batch assembly systems to allow station assignments to be made so that operators are given the same tasks on all models. The frequent practice of switching element assignments from one station to another as batches of models are introduced, can be reduced or eliminated. A modified balance delay index is proposed to assess the

Table 3.5 : Results for the three models example using the exhaustive search algorithm

Minimize delta :

Stn	Work Elements	T_i	$T-T_i$	Δ_i	Var_i	P_{i1}	P_{i2}	P_{i3}
1	2 4 5 8 11 13 14	412	2	22	0.14	1.6	2.0	2.5
2	1 7 12 17 18	420	6	18	0.08	1.8	1.8	2.4
3	3 6 9 10 15 16 19	410	4	4	0.06	1.7	1.9	2.3
		1242	12	44	0.28	5.1	5.7	7.2

Minimize difference :

Stn	Work Elements	T_i	$T-T_i$	Δ_i	Var_i	P_{i1}	P_{i2}	P_{i3}
1	1 2 3 4 5 8	414	0	72	0.65	1.5	1.7	3.3
2	7 9 11 12 14 18	414	0	24	0.04	1.7	2.1	2.1
3	6 10 13 15 16 17 19	414	0	48	0.00	1.9	1.9	1.8
		1242	0	144	0.69	5.1	5.7	7.2

Execution time = 6 minutes 4.10 seconds

Table 3.6 : Comparison of results for the three models example

Line Balancing Procedure	Station i	Time T_i	Δ_i	Total Diff	Total Δ
Thomopoulos (1)	1	414	72	4	156
	2	412	50		
	3	416	34		
Thomopoulos (2)	1	412	22	8	52
	2	412	10		
	3	418	20		
Serial (1)	1	414	48	0	240
	2	414	72		
	3	414	120		
Serial (2)	1	418	4	8	56
	2	414	24		
	3	410	28		
Exhaustive (1)	1	414	72	0	144
	2	414	24		
	3	414	48		
Exhaustive (2)	1	412	22	12	44
	2	420	18		
	3	410	4		

(1) minimize difference
(2) minimize delta

ability of an assembly line to be applied to batch assembly systems. The balance delay index is obtained as follows:

$$c_j = \max\ (p_{1j}, p_{2j}, p_{3j}) \tag{27}$$

If applied to the second solution in Table 3.5, this gives the following:

$c_1 = 1.9$

$c_2 = 2.1$

$c_3 = 3.3$

The percent of idle time occurring on the line for each model or batch is defined as balance delay and is denoted by d_j (j = 1, 2, ..., J). The balance delay is given by:

$$d_j = \frac{nc_j - \sum_{i=1}^{n} p_{ij}}{nc_j} \tag{28}$$

Hence, from the same example:

$d_1 = 10.5\%$

$d_2 = 9.5\%$

$d_3 = 27.3\%$

A balance delay index can be computed as:

$$d = \frac{\sum_{j=1}^{J} N_j d_j}{\sum_{j=1}^{J} Nj} \tag{29}$$

In this situation:

$$d = (140(10.5) + 60(9.5) + 40(27.3))/240$$

$$= 13.05\%$$

Because of the high amount of balance delay, the station assignments would probably be altered for each model. Hence, tasks are varied from model to model, or equivalently from batch to batch in an attempt to reduce the balance delay. This practice increases the cost of storage, tooling and learning. The balance delay index for the different algorithms is shown in Table 3.7. The advantage of smoother stations is apparent. A balance delay of 5% may be an acceptable delay considering the costs involved when moving operators and reassigning work elements.

Another important aspect when evaluating line balances is the sensitivity of the line to the different work contents of the models. To investigate this, the optimal model sequence was obtained for each solution with Thomopoulos' penalty method [10]. This method will be discussed in more detail in the next chapter. A station length of two minutes was assumed for this problem. The operator idle time and utility work as defined in the literature survey chapter was calculated for each solution. The sum of the operator idle time and utility work was taken as the inefficiency of each solution. The results are shown in Table 3.8. and demonstrate that the best line performances are obtained when the delta values are small (when the station times are smoothed out between models). Solutions with perfect line balances did not perform as well as the solutions with small delta values. The conclusion is, therefore, that the smoothing of stations will be beneficial even if no

Table 3.7 : Balance delay index for all solutions

Line Balancing Procedure	Model j	Delay d_j	Delay d
Thomopoulos (1)	1	10.53	14.10
	2	13.64	
	3	27.27	
Thomopoulos (2)	1	5.56	5.16
	2	5.00	
	3	4.00	
Serial (1)	1	15.00	17.64
	2	17.39	
	3	27.27	
Serial (2)	1	0.00	4.65
	2	9.52	
	3	13.64	
Exhaustive (1)	1	10.53	13.07
	2	9.52	
	3	27.27	
Exhaustive (2)	1	5.56	5.16
	2	5.00	
	3	4.00	

(1) minimize difference
(2) minimize delta

Table 3.8 : Operator inefficiencies for all solutions

Line Balancing Procedure	Total Diff	Total Δ	Operator inefficiency
Thomopoulos (1)	4	156	16.14
Thomopoulos (2)	8	52	12.08
Serial (1)	0	240	22.91
Serial (2)	8	56	14.92
Exhaustive (1)	0	144	15.81
Exhaustive (2)	12	44	12.08

(1) minimize difference
(2) minimize delta

batch assembly is planned. Smoother stations will result in less operator idle time and utility work, thereby providing a much more efficient mixed model assembly line.

Chapter 4: Model sequencing.

This chapter deals with the model sequencing problem of mixed model assembly lines. The assumption is made that a suitable line balance has already been achieved with one of the techniques described in the previous chapter. The purpose of the sequencing procedure is to determine the ordering in the flow of models which results in the optimization of a given objective function.

4.1 Penalty cost method.

The assembly line studied is the moving belt type with variable length stations as defined earlier in the literature survey chapter. Each operator is assigned to a specified region called a station, and is permitted to move into the contiguous upstream and downstream stations by amounts called respectively upstream and downstream allowances. Conveyor speed is constant, and the operators move from one product unit to the next without measurable delay. The quantities used to measure assembly performance are operator idle time, congestion, work deficiency and utility work. The penalty cost approach assigns a cost to each of these inefficiencies and then sequences the models in such a way that minimizes the total cost for the entire sequence. The method assumes that a line balance has already been achieved and that the physical dimensions of each station has been determined.

The mathematical model presented here is a slight modification of Macaskill's approach [19]. In this model, concurrent working is not allowed. When concurrent

working is permitted, an operator may start work on a new unit as soon as the work on the previous unit is completed, provided that the unit has reached the upstream allowance region. Thus, there is no need to consider what the worker in the preceding station is doing. When concurrent work is not allowed as in this model, an operator must also wait until the operator in the previous station has completed the work on the unit. Consequently, there will always be only one operator working on a unit at any time. The algorithm can be easily modified, however, to allow concurrent work.

The mathematical model is a set of equations that describe the movement of a given unit through the assembly line. From these set of equations, the inefficiency times are computed and, finally, the total penalty cost. A fixed launching rate is assumed for this model. Kilbridge and Wester [13] have shown that the optimal launching rate for a mixed model line is given by:

$$\gamma = \frac{ttwc}{n\sum_{j=1}^{J} N_j} = \frac{\sum_{i=1}^{n} T_i}{n\sum_{j=1}^{J} N_j} \qquad (30)$$

The times for a product to move through the regions of a station are called station passage times (t_j), upstream allowance times (t^u_j) and downstream allowance times (t^d_j). These quantities can be easily obtained from the station dimensions and belt speed. Let the variables a_{ij} and x_{ij} denote respectively the times of entry and exit for product i in station j. Therefore:

$$x_{ij} = a_{ij} + t_j \tag{31}$$

$$x_{ij} = a_{i,j+1} \tag{32}$$

Given an initial condition, equations (31) and (32) establish all station entry and exit times for every product unit. These times are independent of work done on the product. Let the times of starting and ending work on unit i in station j be s_{ij} and e_{ij} respectively. Equations for operator idle time (I_{ij}), congestion (C_{ij}), work deficiency (D_{ij}) and utility work (U_{ij}) can now be developed.

The upstream equations will be developed first. There are three different conditions to consider. The first case is when the operator has finished a unit and is waiting for the next unit to enter the upstream boundary of this station. Mathematically, this condition is given by:

$$a_{ij} - e_{i-1,j} > t_j^u \tag{33}$$

The next aspect to consider is when the operator in the preceding station has finished work on the current unit. If the operator has finished work before the unit enters the upstream boundary of the current station, in other words:

$$e_{i,j-1} < a_{ij} - t_j^u \tag{34}$$

then

$$I_{ij} = a_{ij} - t_j^u - e_{i-1,j} \tag{35}$$

$$D_{ij} = t_j^u \tag{36}$$

$$s_{ij} = a_{ij} - t_j^u \tag{37}$$

If the operator in the previous station finished the work in the upstream boundary of the current station:

$$a_{ij} - t_j^u \leq e_{i,j-1} < a_{ij} \tag{38}$$

then

$$I_{ij} = e_{i,j-1} - e_{i-1,j} \tag{39}$$

$$D_{ij} = a_{ij} - e_{i,j-1} \tag{40}$$

$$s_{ij} = e_{i,j-1} \tag{41}$$

If the operator in the previous station finished the work inside the current station, in other words in the downstream region, or mathematically:

$$e_{i,j-1} \geq a_{ij} \tag{42}$$

then

$$I_{ij} = e_{i,j-1} - e_{i-1,j} \tag{43}$$

$$D_{ij} = 0 \tag{44}$$

$$s_{ij} = e_{i,j-1} \tag{45}$$

The next case occurs when the operator in the current station, finished work on the previous unit within the upstream boundary of the station, or:

$$0 \leq a_{ij} - e_{i-1,j} \leq t_j^u \tag{46}$$

Again, it is necessary to take the operator in the previous station into account. If this operator finished work on the current unit before the current operator finished work on the previous unit, or:

$$e_{i,j-1} < e_{i-1,j} \tag{47}$$

then

$$I_{ij} = 0 \tag{48}$$

$$D_{ij} = a_{ij} - e_{i-1,j} \tag{49}$$

$$s_{ij} = e_{i-1,j} \tag{50}$$

If the operator in the previous station finished the work after the current operator finished working on the previous unit, but still within the upstream allowance region of the current station, in other words:

$$e_{i-1,j} \leq e_{i,j-1} < a_{ij} \tag{51}$$

then

$$I_{ij} = e_{i,j-1} - e_{i-1,j} \tag{52}$$

$$D_{ij} = a_{ij} - e_{i,j-1} \tag{53}$$

$$s_{ij} = e_{i,j-1} \tag{54}$$

Lastly, if the previous operator finished the work inside the current station, in other words if the operator in the previous station entered the downstream boundary of the current station:

$$e_{i,j-1} \geq a_{ij} \tag{55}$$

then

$$I_{ij} = e_{i,j-1} - e_{i-1,j} \tag{56}$$

$$D_{ij} = 0 \tag{57}$$

$$s_{ij} = e_{i,j-1} \tag{58}$$

The third case for the upstream equations occurs when the operator in the current station finished working on the previous unit after the next unit has entered the station:

$$a_{ij} \leq e_{i-1,j} \tag{59}$$

If the previous operator has not finished working on the current unit then:

$$I_{ij} = e_{i,j-1} - e_{i-1,j} \tag{60}$$

$$D_{ij} = 0 \tag{61}$$

$$s_{ij} = e_{i,j-1} \tag{62}$$

If the operator has finished working on the current unit, or:

$$e_{i,j-1} < e_{i-1,j} \tag{63}$$

then

$$I_{ij} = 0 \tag{64}$$

$$D_{ij} = 0 \tag{65}$$

$$s_{ij} = e_{i-1,j} \tag{66}$$

The downstream equations are much simpler to derive because it is not required to take the preceding operator's movements into account. However, this assumes that the stations overlap is relatively small. For example, the downstream boundary of the station will not extend into the downstream boundary of the next station. Fortunately, in practice that will rarely be the case so the assumption does not limit the applicability of the model.

The first case for the downstream equations occurs when the unit is not completed in the downstream region of the current station. The unit will, therefore, require some utility work to finish the uncompleted work. Let the station time for this model be given by p_{jm}, where j indicates station j and m the model type m (m = 1, 2, ..., J). The first case results whenever:

$$s_{ij} + p_{jm} > x_{ij} + t_j^d \tag{67}$$

then

$$U_{ij} = s_{ij} + p_{jm} - x_{ij} - t_j^d \tag{68}$$

$$C_{ij} = t_j^d \tag{69}$$

$$e_{ij} = x_{ij} + t_j^d \tag{70}$$

The next case occurs when the work on the unit is completed in the downstream region of the current station, or:

$$x_{ij} < s_{ij} + p_{jm} \leq x_{ij} + t_j^d \tag{71}$$

then

$$U_{ij} = 0 \tag{72}$$

$$C_{ij} = s_{ij} + p_{jm} - x_{ij} \tag{73}$$

$$e_{ij} = s_{ij} + p_{jm} \tag{74}$$

The last case results when the work is completed within the normal station bounds, or:

$$s_{ij} + p_{jm} \leq x_{ij} \tag{75}$$

then

$$U_{ij} = 0 \tag{76}$$

$$C_{ij} = 0 \tag{77}$$

$$e_{ij} = s_{ij} + p_{jm} \tag{78}$$

The penalty cost can be computed for each unit at every station by:

$$L_{ij} = aI_{ij} + bD_{ij} + cC_{ij} + dU_{ij} \tag{79}$$

The values a, b, c and d are the penalty costs associated with each type of inefficiency. The total penalty cost is given by:

$$L = \sum_{i=1}^{n} \sum_{j=1}^{N} L_{ij} \tag{80}$$

where

$$N = \sum_{j=1}^{J} N_j \tag{81}$$

A serial method is used to determine the optimal launching sequence. At each step,

the total penalty cost is calculated for all remaining model types to be assigned. The model with the lowest cost is then selected as the next model in the sequence. If there are two or more models with the same penalty cost, the model with the biggest work content is selected. This protects against a build-up of high penalty models near the end of the model sequence. A flow chart for this algorithm is shown in Figure 4.1. The algorithm is very simple to implement on a computer and any of the traditional computer languages can be used. The backtracking capabilities of Prolog are not needed in this case and the program was therefore written in Pascal. The Pascal program is shown in Appendix A.

4.2 Line length minimization method.

One of the disadvantages of the penalty cost method is that the physical dimensions of the stations must be known in advance. Dar-El and Cother [20] mentioned that this is a serious weakness in the penalty cost method. The objective of their minimization method is to determine a sequence of products that minimizes the overall assembly line length required for no operator interference.

Two algorithms are presented, one for closed work station interfaces and one for open work station interfaces. The difference between these two interfaces is illustrated in Figure 1.1. A flow chart for the closed station interface is shown in Figure 4.2 while Figure 4.3 shows the open station interface. The algorithms are of the serial type, where a selection heuristic and an acceptance heuristic are used to determine the model sequence (refer to section 2.3). To determine the optimal

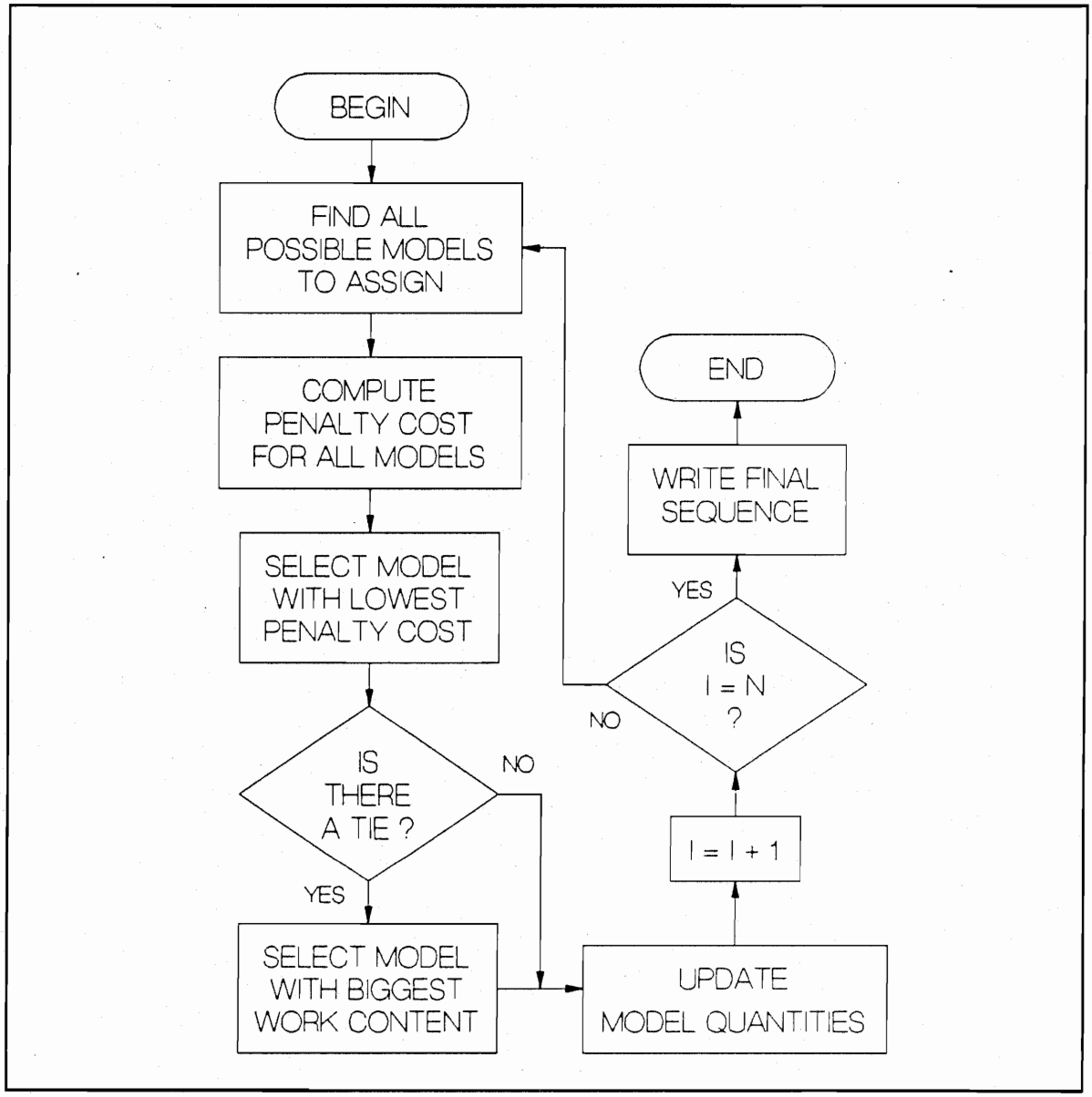


Figure 4.1 : Flow chart for penalty cost approach

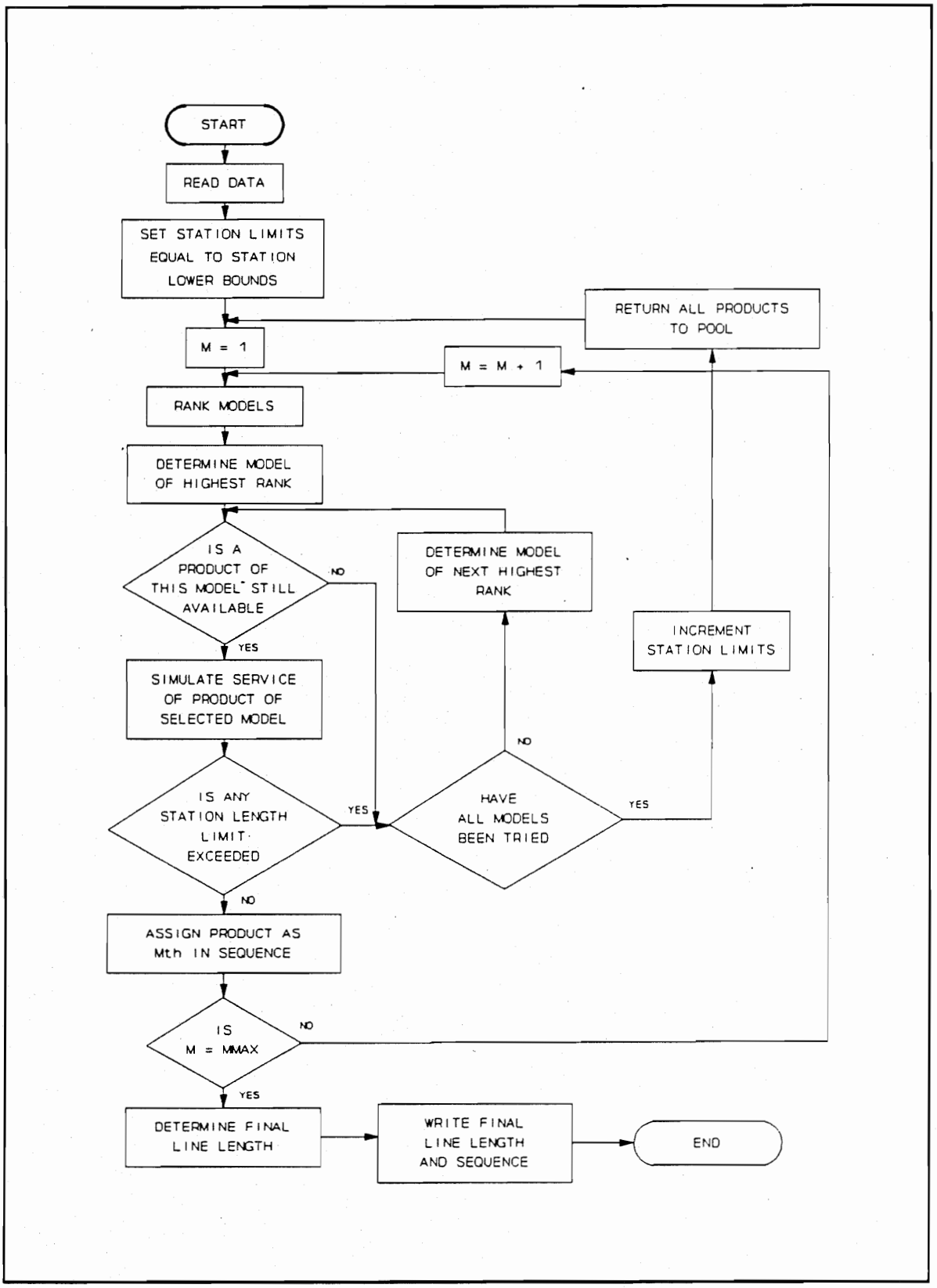


Figure 4.2 : Flow chart for closed station interface

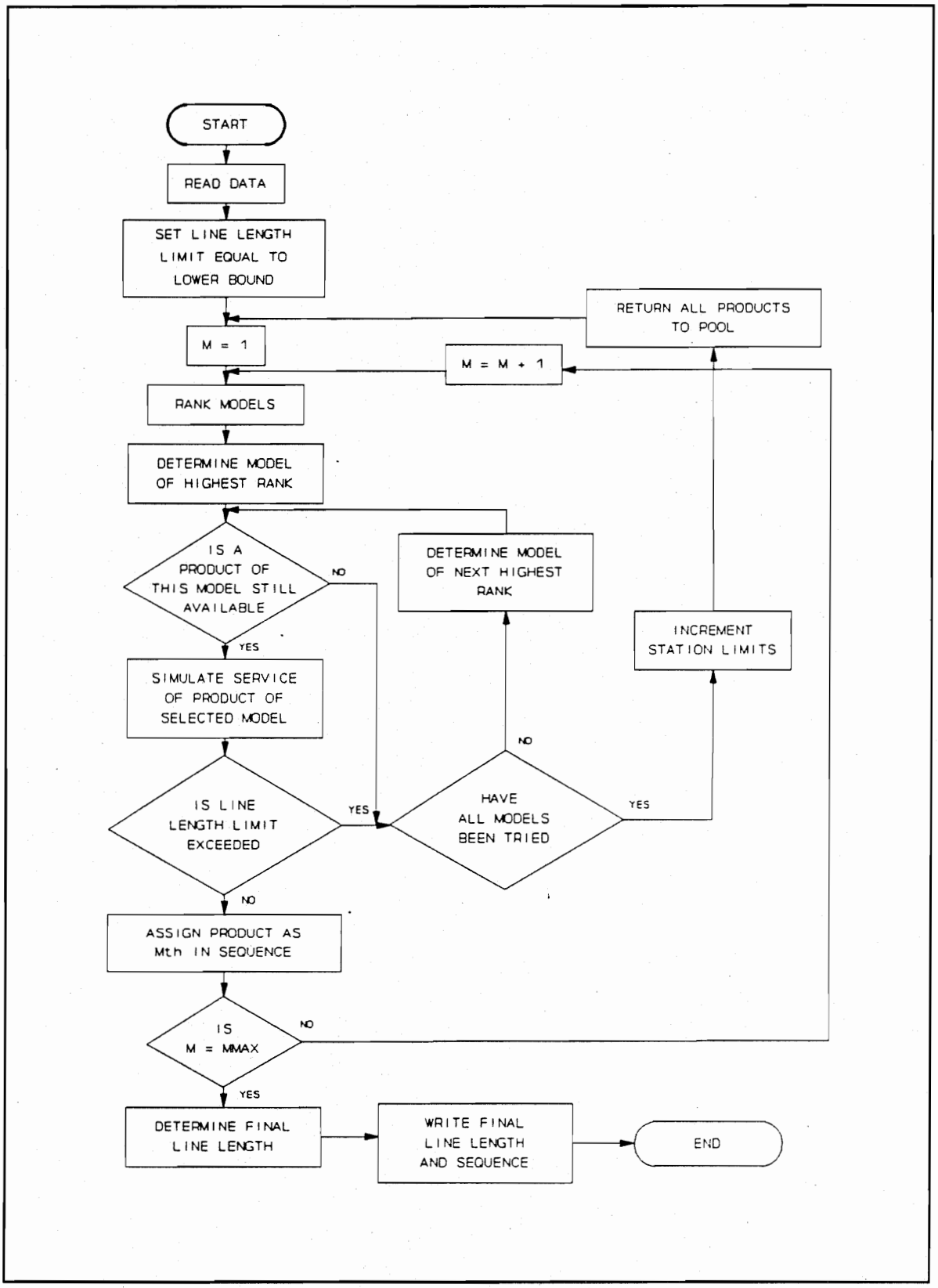


Figure 4.3 : Flow chart for open station interface

model sequence, the service of a given model is simulated and the displacements of the operators are calculated. Let:

dm(l,m)	:	displacement of operator l when starting work on product m
dp(l,m)	:	displacement of operator l when completing work on product m
dmax(l)	:	furthest displacement of operator l downstream
dmin(l)	:	furthest displacement of operator l upstream
j_m	:	model category of the m^{th} product in the sequence
L	:	total line length
n	:	number of stations/operators in the line
N	:	total number of units to be sequenced
p_{lj}	:	operation time in station i on model j
γ	:	fixed launching interval

The displacement of the operators is measured from their respective starting points on the sequence (i.e. for m = 1). Figure 4.4 illustrates the meaning of these terms applied to the first station in the line. The displacement of operator l when starting work on the first product is always zero, i.e.

$$dm(l,1) = 0 \tag{82}$$

The displacement of operator l when completing work on the m^{th} product is given by:

$$dp(l,m) = dm(l,m) + p_{lj_m} \tag{83}$$

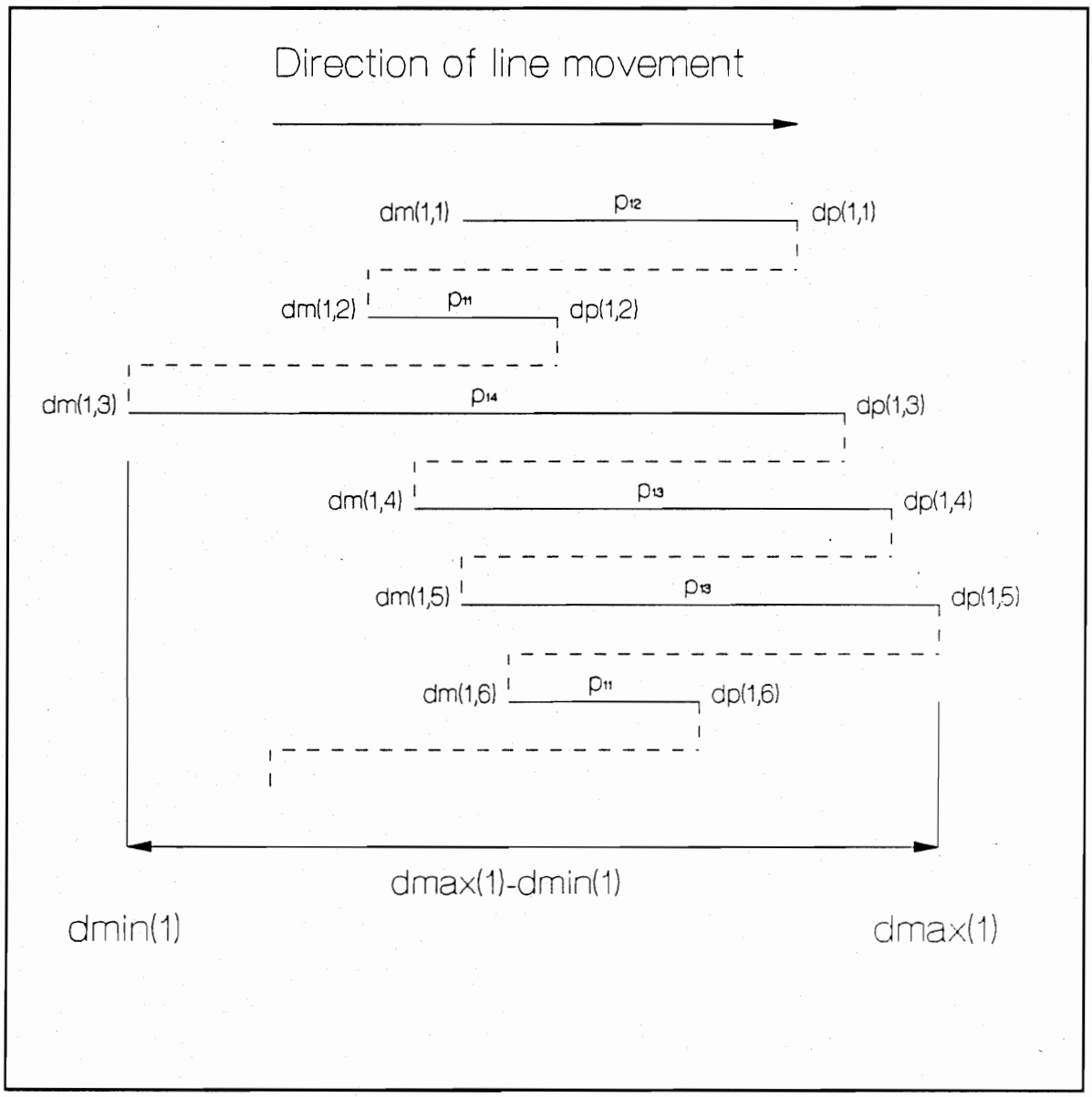


Figure 4.4 : Illustration of various terms used in sequencing approach

The displacement when starting work on the subsequent product (m+1) is:

$$dm(l,m+1) = dp(l,m) - \gamma \tag{84}$$

The maximum displacement of operator l upstream and downstream is given by:

$$dmin(l) = \min|dm(l,m),\ m=1,N| \tag{85}$$

$$dmax(l) = \max|dp(l,m),\ m=1,N| \tag{86}$$

The assembly line length for the closed station interface is given by:

$$L = \sum_{l=1}^{n}[dmax(l) - dmin(l)] \tag{87}$$

If tmax(l) and tmin(l) represent the largest and smallest operation times for station l respectively, the lower bound to the length of station l, B(l), is:

$$B(l) = \max|tmax(l),\ 2\gamma - tmin(l)| \tag{88}$$

For open stations, the approach is a little different. Let $\delta(l)$ be the distance between the starting points where operators l and l+1 begin the sequence. Therefore:

$$\delta(l) = \max|dp(l,m) - dm(l+1,m),\ m=1,N| \tag{89}$$

The assembly line length is then:

$$L = dmax(n) - dmin(1) + \sum_{l=1}^{n-1}\delta(l) \tag{90}$$

For the case of the open station interface, the lower bound is equal to the total assembly time of the product having the greatest work content:

$$B = \max|\sum_{i=1}^{n} p_{ij},\ j=1,J| \tag{91}$$

The algorithms basically consists of two heuristics - one for the candidate selection,

the other for the assignment to the sequence. The products to be assigned to the sequence are referred to as the pool. Successive products from the pool are added to the sequence until the pool is empty. To select the m^{th} product in the sequence, various products are tried until one is found that satisfies an acceptance heuristic. The order in which the models are tried is determined by a selection heuristic. The selection heuristic is designed to spread the models evenly throughout the sequence. To determine the m^{th} product in the sequence, the models are ranked in descending order of Rank(j), where NN_j is the number of products of model j still in the pool:

$$Rank(j) = mN_j - N[N_j - NN_j] \quad (j=1, J) \tag{92}$$

A product is then drawn from the highest ranking model. If this does not satisfy the acceptance heuristic, it is returned to the pool and the next highest ranking model is tried and so on. The acceptance heuristic for the closed station interface operates in the following way. A limit is placed on the length of each station. To determine a product's suitability as the m^{th} product in the sequence, the service of the product is simulated and the length of each station calculated with equations (83) through (87). The simulated operator movements are assigned temporary values at this stage. If the length of any station exceeds the limit, the model is rejected. If the model is accepted, the temporary values become permanent. If at some stage, no model satisfies the acceptance heuristic, the station length limits are increased by equal amounts, all units are returned to the pool and the sequencing procedure recommences. The starting values for the station length is the lower bound as given by

equation (88).

The acceptance heuristic for open stations is as follows. A limit is placed on the overall line length. To determine the m^{th} product in the sequence, the highest ranked model is selected and the service of this model simulated with equations (83) to (86) and (89) to (90). Again, the displacements calculated from these equations are only assigned temporary values. If the line length limit is exceeded, the model is rejected. Otherwise, it is accepted and the temporary values are made permanent. If at some stage no model satisfies the acceptance heuristic, the line length limit is increased, all units are returned to the pool and the sequencing procedure starts over. The starting value for the line length is the lower bound as given by equation (91).

These two algorithms are very straightforward to implement on a computer. Any conventional computer language can be used because no backtracking capabilities are required. The program was therefore written in Pascal. The program is shown in Appendix A.

4.3 Exhaustive search approach.

One of the disadvantages of the serial methods, used in both the penalty cost approach and the line minimization approach, is that once a model is assigned to the line, the algorithm never comes back to this point to reevaluate another alternative. In this section, an exhaustive search approach is used to find the optimal sequence. The objective function can be either the minimization of total penalty cost or the line length minimization. The objective function chosen for this algorithm was the line

length minimization with a closed station interface.

A problem occurs when the number of units or the number of different models are too large to allow the exhaustive search algorithm to examine all possible combinations within a reasonable amount of time. To solve this problem, three remedies are proposed. Firstly, the batch of units can be partitioned into smaller batches. The optimal sequence for the smaller batch can then be found, and this optimal sequence repeated to complete the production requirements. Another approach is to start with a feasible model sequence obtained by a serial method, and let the exhaustive search algorithm attempt to improve this solution. That way, the execution of the algorithm can be interrupted at any time and the resulting solution wouldn't be any worse than that obtained by the serial method. The third approach is to sequence the units in steps of say n units. The optimal sequence for the first n units is determined, then the next n units and so on. For a step size of one, this approach reduces to the serial method.

The search methodology is very similar to the exhaustive search used for line balancing. A flow chart for the algorithm is shown in Figure 4.5, and the pseudocode for the Prolog program is shown in Table 4.1 The advantage of this approach is that all possible combinations for model sequences are generated, therefore the optimal solution is guaranteed. The disadvantage is that the computer time required to reach the solution might be impractical. In such cases, one of the above recommended remedies should be followed.

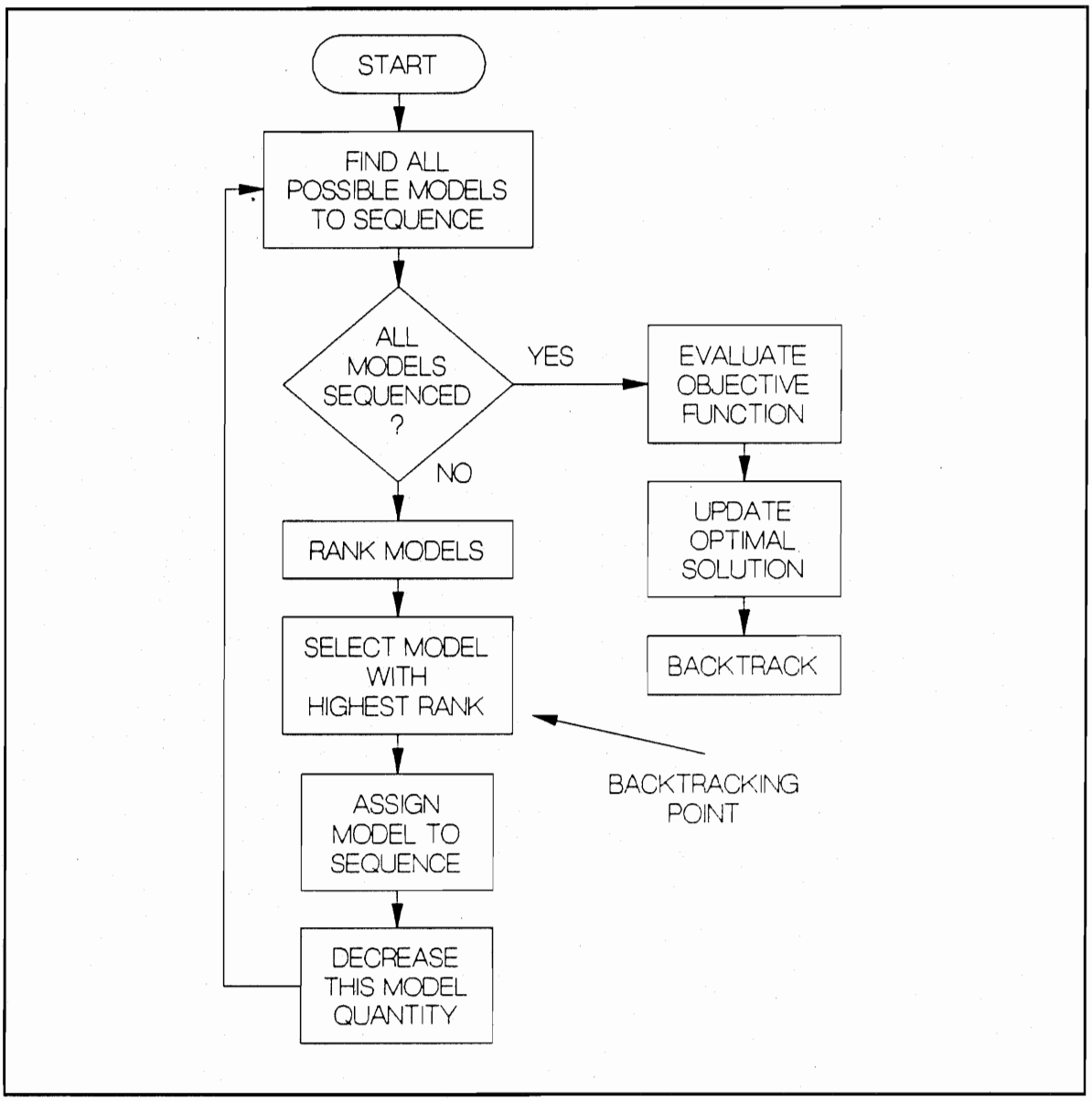


Figure 4.5 : Flow chart for exhaustive search sequencing method

Table 4.1 : Pseudocode for exhaustive search sequencing approach

```
sequence(Models,Sequence) :-
  if no more models then
  compute line length
  write sequence
  update minimum value
  backtrack
sequence(Models,Sequence) :-
  if any models left
  rank all remaining models
  put models in a ranked list
 *take a model from this list : member(X,Ranked_list)
  assign model X to sequence
  decrease the count for selected model X
  sequence(Models,Sequence + X)

 * = backtracking point
```

4.4 Assessment of model sequencing algorithms.

In this section, an assessment of the models sequencing algorithms will be made. Consider the following model sequencing problem from reference [10]. A line balance has already been obtained and the corresponding station times are shown in Table 4.2. The physical station dimensions are shown in Table 4.3 and the required production schedule is shown in Table 4.4. The penalty cost method will first be considered. The penalty cost parameters a, b, c and d were all taken as unity. The solution from the penalty cost approach is shown in Table 4.5. A breakdown of the inefficiencies is given in Table 4.6.

A plot of the penalty cost at each stage of the sequencing process is shown in Figure 4.6. This figure shows that there is no build up of high penalty models towards the end of the sequence. Consequently, it can be concluded that this sequence is an acceptable solution.

The same data, except for the station dimensions, were used with the line length minimization algorithms. The sequence from the closed station algorithm was used as a starting point for the Prolog search algorithm. The resulting station lengths for the open station interface is shown in Table 4.7. The model sequences for both the closed and open station interfaces are shown in Table 4.8. A comparison of all the model sequencing methods is shown in Table 4.9. The advantage of the line length minimization methods is clear from this table. For approximately the same total line length, a sequence can be found that will result in no operator interference whatsoever. The big benefit is that the operator idle time resulting from the penalty

Table 4.2 : Station times for the sequencing example

Station	Station Times for Each Model					
	1	2	3	4	5	6
1	4.37	4.21	4.18	5.21	6.14	4.77
2	4.22	4.01	5.45	5.93	4.12	5.06
3	4.32	4.31	4.53	4.47	4.98	4.93
4	4.67	4.23	4.15	4.36	4.46	4.46
5	4.56	4.51	4.22	3.96	4.22	4.22
6	4.76	4.17	4.47	3.07	3.42	4.61
7	4.67	4.42	4.46	3.57	3.32	4.22
8	5.05	2.89	5.20	3.03	4.72	5.20
9	4.38	3.87	5.01	4.10	4.68	5.30
10	4.72	3.72	4.72	3.72	4.72	4.72
11	4.50	3.85	4.16	3.75	5.77	5.66
12	4.38	4.03	4.89	4.17	5.10	4.80
13	4.55	3.65	5.45	3.87	4.85	4.57
14	4.40	4.23	4.83	4.16	4.65	4.44
15	4.38	4.12	4.65	4.24	4.61	5.12
16	4.52	4.27	4.00	4.22	3.87	4.37
17	4.40	4.04	4.63	4.35	4.14	4.49
18	4.17	4.70	4.39	4.12	3.71	4.79
19	4.65	4.20	4.04	4.46	4.04	4.30
Total	85.67	77.43	87.43	78.76	85.52	90.03

Table 4.3 : Station dimensions for the sequencing example

j	t_j	t^u_j	t^d_j
1	6.4	0.0	2.0
2	6.4	2.0	2.0
3	6.4	1.0	2.0
4	6.4	1.0	2.0
5	6.4	1.0	2.0
6	6.4	1.0	0.0
7	6.4	0.0	0.0
8	6.0	0.0	2.0
9	6.0	1.0	2.0
10	6.0	1.0	2.0
11	6.0	1.0	1.0
12	5.6	1.0	0.0
13	6.4	0.0	2.0
14	6.0	1.0	2.0
15	6.0	1.0	0.0
16	5.0	0.0	0.0
17	5.0	0.0	0.0
18	5.0	0.0	0.0
19	5.0	0.0	0.0

Table 4.4 : Production schedule for the sequencing example

j	N_j
1	7
2	6
3	3
4	1
5	1
6	2
Total	20

Table 4.5 : Solution of the serial penalty cost approach

Unit #	Model #	Total Penalty
1	6	0.00
2	1	0.48
3	2	0.01
4	3	0.89
5	1	0.91
6	2	0.03
7	3	0.43
8	1	1.24
9	2	0.03
10	1	1.12
11	1	0.47
12	2	0.37
13	3	1.98
14	1	1.62
15	2	0.92
16	5	2.03
17	1	1.25
18	2	0.23
19	6	1.61
20	4	1.09
Total		16.68

Table 4.6 : Breakdown of inefficiencies for sequencing problem

Type of inefficiency	Time (minutes)
Idle time	8.24
Work deficiency	7.39
Utility work	0.10
Work congestion	0.95
Total	16.68

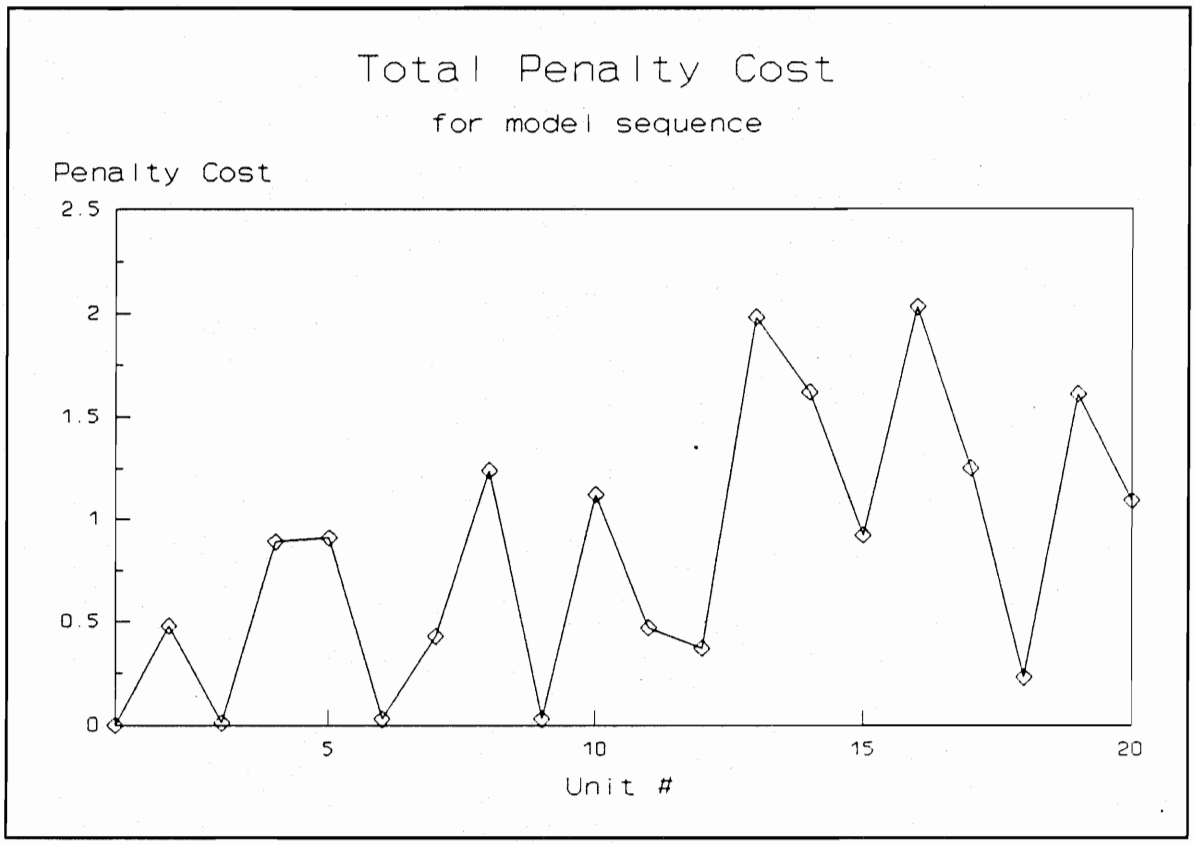


Figure 4.6 : Penalty cost vs unit number

Table 4.7 : Station lengths for closed station algorithms

Station #	Station Length	
	Serial	Prolog
1	7.03	6.38
2	7.39	7.80
3	5.85	5.65
4	5.27	5.21
5	5.05	4.96
6	6.72	5.95
7	5.97	5.51
8	7.37	7.43
9	5.76	5.53
10	5.81	5.85
11	6.63	6.27
12	5.61	5.34
13	6.09	6.09
14	5.09	5.07
15	5.55	5.33
16	6.37	6.37
17	6.18	5.94
18	5.60	5.34
19	5.62	5.62
Total	114.95	111.64

Table 4.8 : Model sequences for line length minimization algorithms

Unit #	Closed Stations Model #	Open Stations Model #	Prolog (Closed) Model #
1	1	1	1
2	2	2	2
3	3	3	3
4	1	1	1
5	2	2	2
6	6	6	6
7	1	1	1
8	2	2	2
9	4	4	4
10	1	1	1
11	3	3	3
12	5	2	2
13	2	5	1
14	1	1	6
15	6	2	1
16	2	1	2
17	1	6	1
18	1	3	3
19	2	2	2
20	3	1	5

Table 4.9 : Comparison of model sequencing algorithms

Method	Type of Algorithm	Station type	Line length	Inefficiency
Penalty Cost	Serial	Variable length	112.8	16.68
Line length minimization	Serial	Closed	115.0	0.00
Prolog	Exhaustive Backtracking	Closed	111.6	0.00
Line length minimization	Serial	Open	103.9	0.00

cost approach can be eliminated. Another important benefit is that no utility work will be required for the line length minimization methods. This means that all the units leaving the line will be fully completed and the need for the floating operator required for the penalty cost approach is eliminated. The slight improvement of the open station interface over the closed station interface stems from the fact that there are no physical boundaries for the open station interface. Whenever possible, this interface should always be preferred over the closed station interface. A comparison of the first three stations for the two scenarios is shown in Figure 4.7 which points out the benefits of the open station interface.

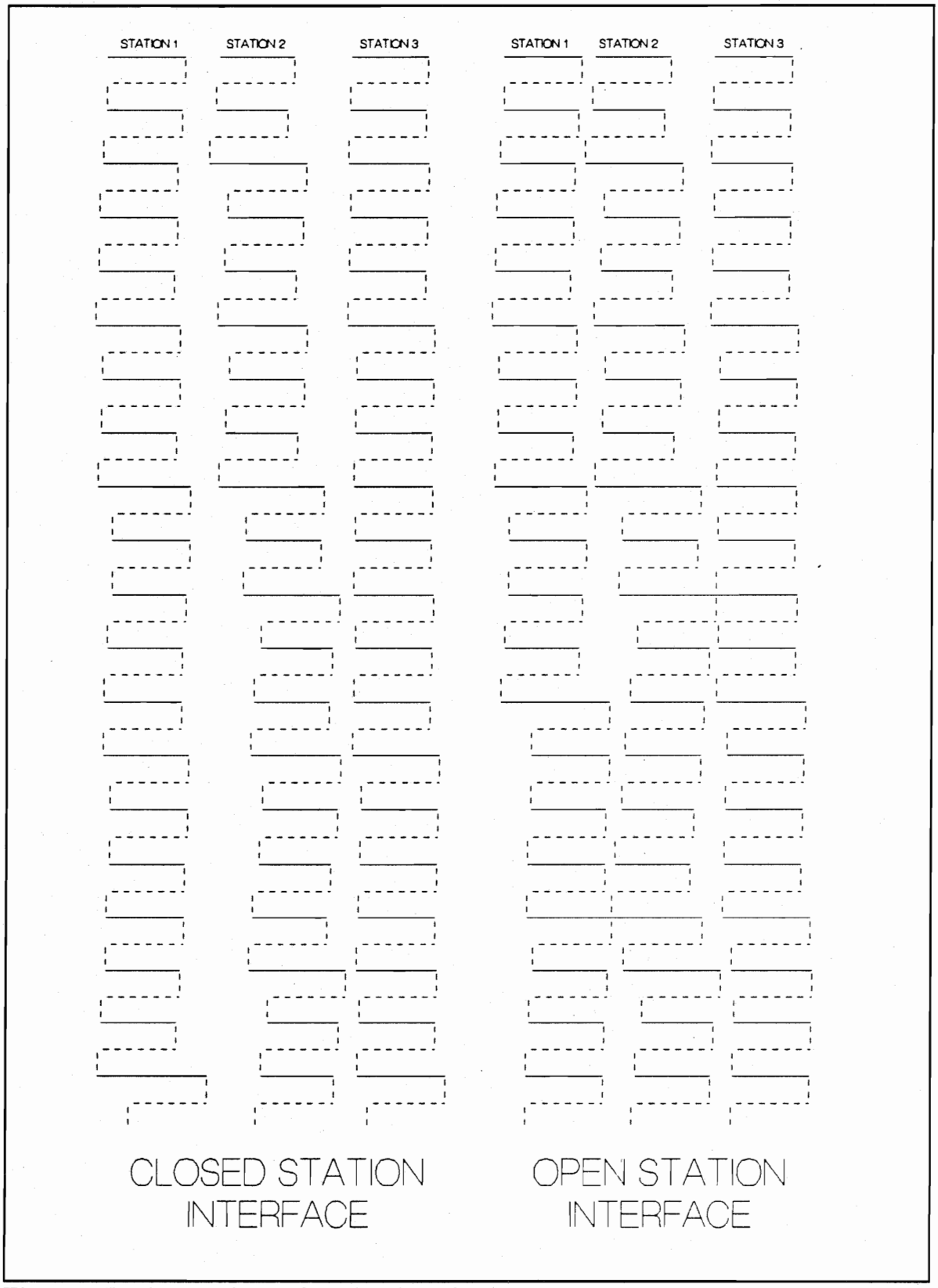


Figure 4.7 : Comparison of station interfaces

Chapter 5: Case study.

In this chapter, data given in reference [26] for a relatively large mixed model assembly line will be utilized to assess the various techniques of line balancing and model sequencing. The objective is to examine the efficiency of the algorithms and corresponding computer programs using a real world problem.

5.1 Problem statement.

The company for this case study is a manufacturer of telecommunications equipment. One of its main product lines includes the assembly of 'frames', several of which are used in the construction of each telephone exchange. The frames not only come in various sizes but also comprise different combinations of 'relays' and 'bridges'. In the current layout, production is divided on a functional basis between several departments, (e.g., winding, relay assembly, mounting frames, etc.), and there are a total of 160 operators involved in these assemblies. Many of the components are expensive and in-process inventory costs were of a great concern to management. The objective of the case study is to design a more economical production system. Since this problem has all the necessary ingredients of a mixed model assembly line, the approach was to design a mixed model line to satisfy the requirements. In-process inventory would be almost zero and less material handling would be required. To test the feasibility of the mixed model line approach, a suitable line balance and model sequence need to be obtained.

There are eight different models to be assembled on the line. The work elements for the models are described in Table 5.1. The production requirements per shift are shown in Table 5.2. The basic data for the problem is shown in Table 5.3. The shift time for this company is 500 minutes. Therefore, dividing the total shift time column by 500 gives the minimum number of workers to operate wholly on each work element. The residual time is the time still needed to be assigned to some worker. The operators do not work simultaneously on the main frame - most work on sub-tasks of the element is done at work stations adjacent to the frame. The line balancing problem will be to balance these residual times for a cycle time of 500 minutes.

5.2 Line balancing.

The goal of this section is to balance the residual times from Table 5.3 to determine the minimum number of stations for a cycle time of 500 minutes. The precedence constraints for the work elements are shown in Figure 5.1. In order to use the line balancing algorithms described in chapter 3, the residual times need to be converted to elemental times for each model. The converted elemental times are shown in Table 5.4. The minimum number of stations needed for a cycle time of 500 minutes is:

$$n_{min} = 11735/500 = 24 \text{ stations.}$$

However, the relatively small element/station ratio for this problem makes it extremely unlikely to achieve a line balance with only 24 stations. Due to the size of

Table 5.1 : Description of work elements

k	Work element description
1	Mounting of details
2	Mounting of yoke of armature
3	First treatment of bobbin
4	Pile up assembly
5	Mounting of selector frame
6	Preparing multiple frame
7	Riveting of blocks
8	Mounting of terminals
9	Inspection of stamping of spring sets
10	Insertion of contact wires
11	Mounting of brushes
12	First winding
13	Adjustment of spring sets
14	Riveting of contact wires
15	Cutting of brushes
16	First wrapping of coils
17	Inspection of spring sets
18	Mounting of selector with spring sets
19	First adjustment of brushes
20	Second winding of coils
21	Pressing of springs
22	Finishing of coil
23	Preparing 'R-C' terminals
24	Soldering of 'R-C' terminals
25	Visual inspection of coils
26	Relay assembly
27	Assembly of electromagnetic sets
28	Final assembly of selector
29	Relay adjustment
30	Selector adjustment
31	Relays testing
32	Mounting of 'S&T' sets on frame
33	Relay oiling for storage
34	Soldering of multiple wires
35	Arranging relays on frames
36	Visual inspection of soldering
37	Assembly of relays sets
38	Testing at 500 volts
39	Stamping of armatures
40	Wrapping of relays sets
41	Soldering of auxiliary components
42	Platine inspection
43	Mounting of platines in frame
44	Mounting of horizontal parts & adjustment
45	Mechanical inspection of frame
46	Electrical inspection of frame
47	Mounting of cable
48	Final electrical test
49	Addition of final assembly items and stamping
50	Packaging

Table 5.2 : Production requirements per shift

Model j	N_j
1	1
2	4
3	2
4	4
5	1
6	2
7	2
8	1
Total	17

Table 5.3 : Basic data for the case study

k	Work elements for model number 1	2	3	4	5	6	7	8	Total Shift Time	Opera-tors per element	Resid-ual times
1	104	102	86	120	125	105	74	60	1707	3	207
2	35	35	10	8	10	44	42	38	447	0	447
3	69	68	34	26	34	59	52	55	824	1	324
4	405	400	255	187	241	479	431	369	5693	11	193
5	38	33	52	33	52	0	0	0	458	0	458
6	6	6	6	6	6	0	0	0	72	0	72
7	14	14	14	14	14	0	0	0	168	0	168
8	88	87	48	49	50	93	64	75	1167	2	167
9	91	90	42	30	42	103	89	84	1165	2	165
10	20	18	28	18	28	0	0	0	248	0	248
11	14	14	28	14	28	0	0	0	210	0	210
12	156	153	93	76	99	336	132	241	2534	5	34
13	108	103	120	102	120	0	0	0	1288	2	288
14	6	5	8	5	8	0	0	0	70	0	70
15	14	14	14	14	14	0	0	0	168	0	168
16	38	37	13	9	15	35	23	45	424	0	424
17	31	30	37	30	37	0	0	0	382	0	382
18	169	148	232	148	232	0	0	0	2049	4	49
19	14	14	14	14	14	0	0	0	168	0	168
20	41	39	30	16	26	79	26	60	617	1	117
21	9	8	12	8	12	0	0	0	109	0	109
22	180	178	92	71	91	171	139	147	2218	4	218
23	11	11	13	11	13	0	0	0	138	0	138
24	70	68	56	46	56	34	27	29	845	1	345
25	34	34	16	14	18	32	28	29	425	0	425
26	58	58	17	13	17	88	65	68	767	1	267
27	6	6	6	6	6	0	0	0	72	0	72
28	47	41	65	41	65	0	0	0	570	1	70
29	265	265	105	71	94	399	352	311	3726	7	226
30	115	100	158	100	158	0	0	0	1389	2	389
31	75	75	34	24	30	120	115	102	1141	2	141
32	37	34	46	34	46	0	0	0	447	0	447
33	21	21	7	6	7	34	25	29	297	0	297
34	106	96	135	96	135	0	0	0	1279	2	279
35	10	10	4	8	4	24	20	18	200	0	200
36	24	21	34	21	34	0	0	0	294	0	294
37	41	41	19	18	19	136	82	71	841	1	341
38	12	12	12	12	12	0	0	0	144	0	144
39	13	13	9	7	8	42	34	36	307	0	307
40	601	601	308	279	323	1172	788	351	9331	18	331
41	10	10	4	4	5	16	17	0	145	0	145
42	24	24	13	10	11	49	44	16	399	0	399
43	24	24	7	7	7	47	33	39	368	0	368
44	133	125	160	125	160	0	0	0	1613	3	113
45	25	25	35	25	35	0	0	0	330	0	330
46	34	30	44	30	44	0	0	0	406	0	406
47	407	401	388	295	405	158	403	247	5741	11	241
48	96	96	114	114	96	120	180	160	2020	4	20
49	87	84	67	57	67	92	63	74	1236	2	236
50	34	34	34	34	34	34	34	34	578	1	78
	4070	3956	3178	2506	3207	4101	3382	2788	57235	91	11735

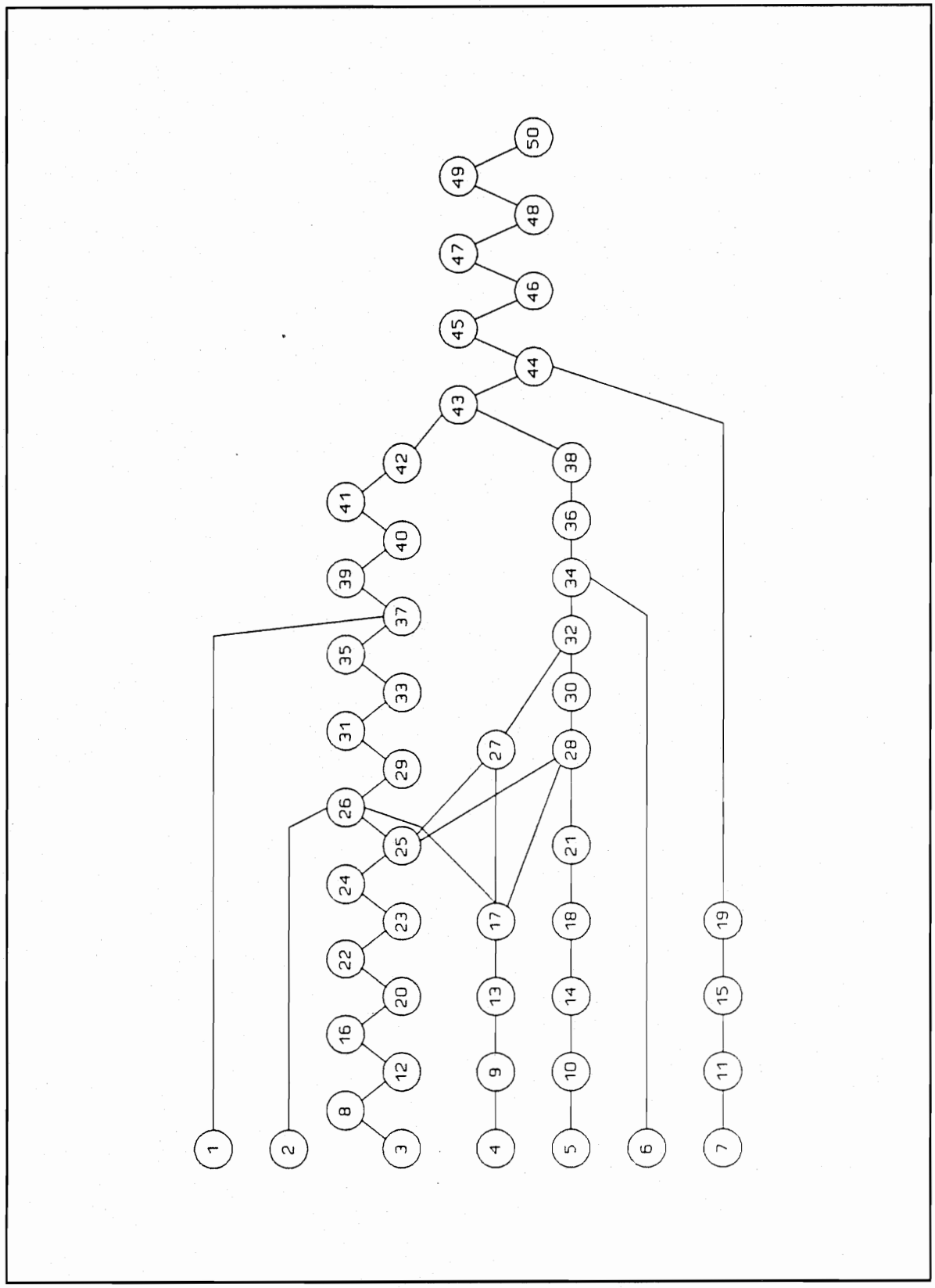


Figure 5.1 : Combined precedence diagram

Table 5.4 : Converted elemental times for case problem

k	1	2	3	t_{4jk}	5	6	7	8	$\hat{t}_k$
1	12.61	12.37	10.43	14.55	15.16	12.73	8.97	7.28	207
2	35.00	35.00	10.00	8.00	10.00	44.00	42.00	38.00	447
3	27.13	26.74	13.37	10.22	13.37	23.20	20.45	21.63	324
4	13.73	13.56	8.64	6.34	8.17	16.24	14.61	12.51	193
5	38.00	33.00	52.00	33.00	52.00	0.00	0.00	0.00	458
6	6.00	6.00	6.00	6.00	6.00	0.00	0.00	0.00	72
7	14.00	14.00	14.00	14.00	14.00	0.00	0.00	0.00	168
8	12.59	12.45	6.87	7.01	7.16	13.31	9.16	10.73	167
9	12.89	12.75	5.95	4.25	5.95	14.59	12.61	11.90	165
10	20.00	18.00	28.00	18.00	28.00	0.00	0.00	0.00	248
11	14.00	14.00	28.00	14.00	28.00	0.00	0.00	0.00	210
12	2.09	2.05	1.25	1.02	1.33	4.51	1.77	3.23	34
13	24.15	23.03	26.83	22.81	26.83	0.00	0.00	0.00	288
14	6.00	5.00	8.00	5.00	8.00	0.00	0.00	0.00	70
15	14.00	14.00	14.00	14.00	14.00	0.00	0.00	0.00	168
16	38.00	37.00	13.00	9.00	15.00	35.00	23.00	45.00	424
17	31.00	30.00	37.00	30.00	37.00	0.00	0.00	0.00	382
18	4.04	3.54	5.55	3.54	5.55	0.00	0.00	0.00	49
19	14.00	14.00	14.00	14.00	14.00	0.00	0.00	0.00	168
20	7.77	7.40	5.69	3.03	4.93	14.98	4.93	11.38	117
21	9.00	8.00	12.00	8.00	12.00	0.00	0.00	0.00	109
22	17.69	17.50	9.04	6.98	8.94	16.81	13.66	14.45	218
23	11.00	11.00	13.00	11.00	13.00	0.00	0.00	0.00	138
24	28.58	27.76	22.86	18.78	22.86	13.88	11.02	11.84	345
25	34.00	34.00	16.00	14.00	18.00	32.00	28.00	29.00	425
26	20.19	20.19	5.92	4.53	5.92	30.63	22.63	23.67	267
27	6.00	6.00	6.00	6.00	6.00	0.00	0.00	0.00	72
28	5.77	5.04	7.98	5.04	7.98	0.00	0.00	0.00	70
29	16.07	16.07	6.37	4.31	5.70	24.20	21.35	18.86	226
30	32.21	28.01	44.25	28.01	44.25	0.00	0.00	0.00	389
31	9.27	9.27	4.20	2.97	3.71	14.83	14.21	12.60	141
32	37.00	34.00	46.00	34.00	46.00	0.00	0.00	0.00	447
33	21.00	21.00	7.00	6.00	7.00	34.00	25.00	29.00	297
34	23.12	20.94	29.45	20.94	29.45	0.00	0.00	0.00	279
35	10.00	10.00	4.00	8.00	4.00	24.00	20.00	18.00	200
36	24.00	21.00	34.00	21.00	34.00	0.00	0.00	0.00	294
37	16.62	16.62	7.70	7.30	7.70	55.14	33.25	28.79	341
38	12.00	12.00	12.00	12.00	12.00	0.00	0.00	0.00	144
39	13.00	13.00	9.00	7.00	8.00	42.00	34.00	36.00	307
40	21.32	21.32	10.93	9.90	11.46	41.57	27.95	12.45	331
41	10.00	10.00	4.00	4.00	5.00	16.00	17.00	0.00	145
42	24.00	24.00	13.00	10.00	11.00	49.00	44.00	16.00	399
43	24.00	24.00	7.00	7.00	7.00	47.00	33.00	39.00	368
44	9.32	8.76	11.21	8.76	11.21	0.00	0.00	0.00	113
45	25.00	25.00	35.00	25.00	35.00	0.00	0.00	0.00	330
46	34.00	30.00	44.00	30.00	44.00	0.00	0.00	0.00	406
47	17.09	16.83	16.29	12.38	17.00	6.63	16.92	10.37	241
48	0.95	0.95	1.13	1.13	0.95	1.19	1.78	1.58	20
49	16.61	16.04	12.79	10.88	12.79	17.57	12.03	14.13	236
50	4.59	4.59	4.59	4.59	4.59	4.59	4.59	4.59	78
Totals	880.41	846.77	765.29	587.25	770.96	649.60	517.89	481.99	11735

the problem the exhaustive search technique would be too slow to implement on a 10Mhz 80286 computer. The prolog program for the exhaustive search was therefore modified so that a step-by-step approach can be taken. The first ten stations were first balanced for example, then the next ten stations and so on. The objective was to provide a good initial solution for the search procedure. It should be noted here that the step-by-step approach was used only to provide the initial sequence for the exhaustive search. The exhaustive search procedure was still used to allow backtracking between stations. Eventually, with a faster computer and more time, all possible solutions could be examined. The advantage of an initial sequence is that the search procedure can be interrupted after some time, and a good solution, although not optimal, would be expected.

There are no acceptable limits for T_i in this case problem since the objective was to load each station as close as possible to the cycle time of 500 minutes (without exceeding it). When the work content of a station is less than 500 minutes, this difference is considered as operator idle time. Consequently, since neither the serial nor the exhaustive algorithms were designed for these conditions, some modifications needed to be made. An iterative approach in which the lower limit was increased at each iteration, was used in this case. The lower limit was assigned a very high value initially. When the algorithm could not go on after a certain number of stations, that meant that the lower limit constraint should be relaxed. The station assignments generated with the higher limit were kept, the lower limit was decreased and the process recommenced until a feasible solution was obtained.

The case problem is first solved with the serial algorithm. The solution for the serial algorithm is shown in Table 5.5. The number of stations or operators is 28 and the total idle time is 2265 minutes. The line balance found using the modified step-by-step exhaustive search algorithm is shown in Table 5.6. The solution with the smoothest stations as objective function was selected in this case. However, the solution that minimizes the difference in work content, resulted in the same number of stations and the same difference in work content or idle time in this particular case. A minimum of 26 operators or stations are required and the idle time is 1265 minutes. The high idle time contents for both solutions could be expected from the low element/station ratio. As expected, the exhaustive search algorithm performed much better than the serial algorithm in this problem. The solution found by the serial algorithm represents an increase of 1000 minutes or nearly 80% in idle time. However, the execution time for the serial algorithm was only 1 minute 8.39 seconds, while the execution time for the exhaustive search was more than 60 minutes. The serial algorithm could be very useful therefore in cases where a 'quick and dirty' approach is needed. A graphical comparison of the two algorithms is shown in Figure 5.2. It is clear from this figure that the exhaustive algorithm is superior to the serial algorithm.

5.3 Model sequencing.

Once a suitable line balance is obtained, the model sequence need to be determined. The type of station used in this company is the closed station interface. Operators

Table 5.5 : Line balance for case problem (serial algorithm)

Station i	Work elements assigned to station i	T_i	$T-T_i$	Δ_i
1	3 7	492	8	73.2
2	1 6 11	489	11	169.7
3	5	458	42	240.1
4	10 14 15	486	14	268.1
5	2 18	496	4	223.5
6	8 12 19 21	478	22	117.2
7	16	424	76	183.1
8	4 9 20	475	25	159.9
9	22 23	356	144	63.1
10	24	345	155	74.1
11	25	425	75	126.1
12	13	288	212	145.8
13	17 27	454	46	236.1
14	26 29	493	7	263.0
15	28 30	459	41	241.1
16	32	447	53	229.1
17	31 33	438	62	215.5
18	34 35	479	21	59.9
19	36 38	438	62	220.1
20	37	341	159	288.0
21	39	307	193	286.9
22	40 41	476	24	195.1
23	42	399	101	225.3
24	43 44	481	19	158.8
25	45	330	170	166.9
26	46	406	94	190.0
27	47 48 49	497	3	77.9
28	50	78	422	341.1
Totals		11735	2265	5238.7

Table 5.6 : Line balance for case problem (exhaustive algorithm)

Station i	Work elements assigned to station i	T_i	$T-T_i$	Δ_i
1	3 8	491	9	137.5
2	5 12	492	8	225.8
3	7 10 14	486	14	251.3
4	16 18	473	27	163.8
5	4 9 20	475	25	153.2
6	1 6 22	497	3	61.5
7	23 24	483	17	125.0
8	11 13	498	2	263.3
9	17 21	491	9	256.3
10	25 28	495	5	101.2
11	27 30	461	39	226.3
12	32	447	53	212.3
13	2	447	53	229.4
14	26 29	493	7	256.4
15	31 34	420	80	65.8
16	15 33	465	35	85.4
17	35 36	494	6	60.2
18	37 38	485	15	160.8
19	19 39	475	25	135.7
20	40 41	476	24	205.8
21	42	399	101	244.7
22	43 44	481	19	153.2
23	45	330	170	173.6
24	46	406	94	191.9
25	47 48 49	497	3	45.7
26	50	78	422	373.4
Total		11735	1265	4559.5

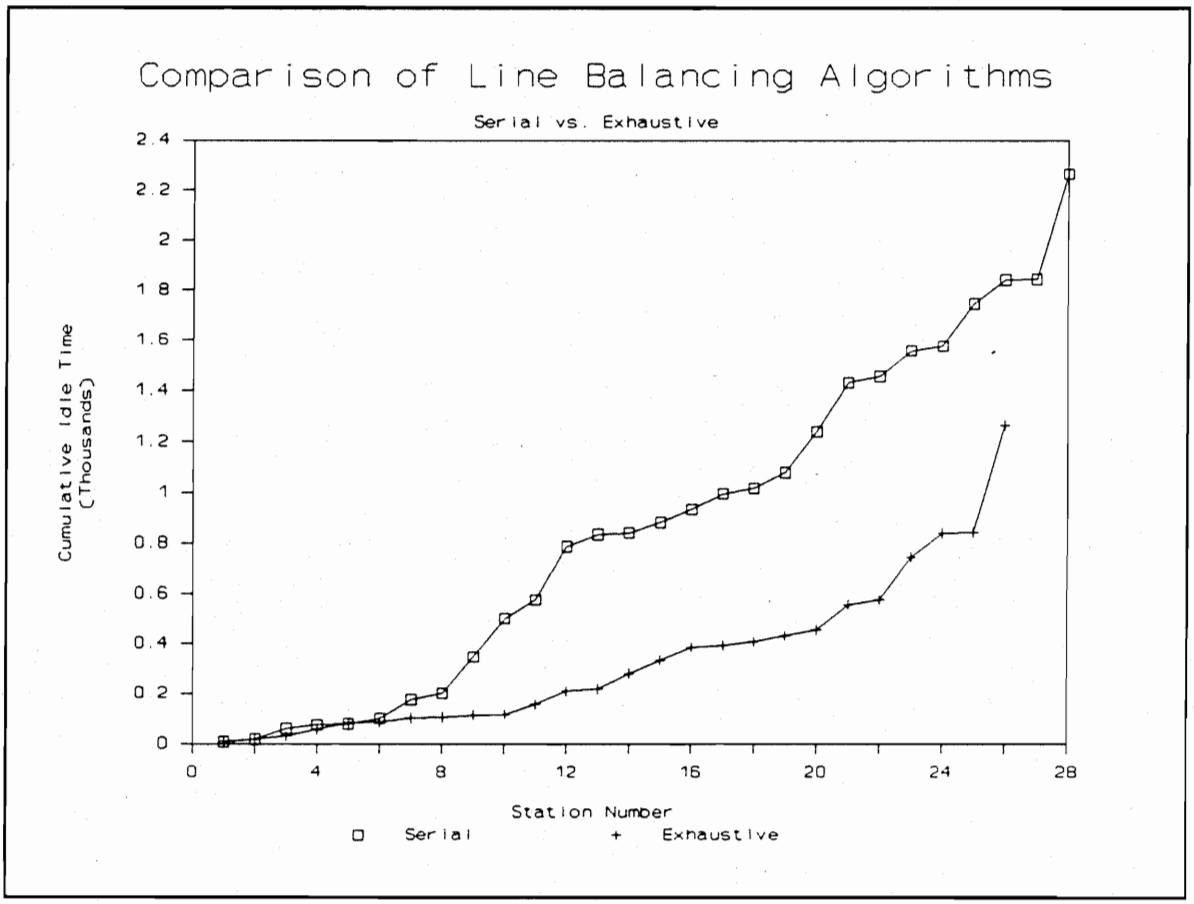


Figure 5.2 : Comparison of serial and exhaustive line balancing algorithms

are confined to their work stations and may not move beyond the station limits.

To determine the optimal model sequence, the original work element times in Table 5.3 are used. Using the line balance in Table 5.6, the total work load in each station is first calculated as shown in Table 5.7. The amount of time assigned to each model j in a given station i, p_{ij}, can then be computed. These values are displayed in Table 5.8. These values are then used as input for the sequencing algorithms. The results from the sequencing algorithms are shown in Table 5.9 and Table 5.10. Again, the prolog search solution is much better than the solution obtained by the serial algorithm from Dar-El and Cother [20]. The reason for this can be seen from Table 5.7. Models 6, 7 and 8 have zero work content for a number of stations, such as stations 8, 9, 11, 12, etc. A station with zero work content means the operator continues to move upstream until the next model arrives. Consequently, if two models with zero content at a specific station follow each other, say models 6 and 7, the operator would move very far upstream and the station would have to be made very large to accommodate these movements. The sequence from the serial method twice selects model 7 just after model 6. The difference in station lengths is especially noticeable at stations 8, 9, 11 and 12 in Table 5.10. The selection heuristic from the serial algorithm was designed to spread the models evenly throughout the sequence, and in turn does not take zero work content stations into account. This type of sequence does not occur in the sequence generated by the exhaustive approach, and the result is a shorter assembly line.

A lot of idle time occurs at the last station where work element 50, which is the

Table 5.7 : Total work load for all stations

i	Work Elements	Work load for each station 1	2	3	4	5	6	7	8	Total	Opera-tors	Resi-dual
1	3 8	157	620	164	300	84	304	232	130	1991	4	9
2	5 12	194	744	290	436	151	672	264	241	2992	6	8
3	7 10 14	40	148	100	148	50	0	0	0	486	1	14
4	16 18	207	740	490	628	247	70	46	45	2473	5	27
5	4 9 20	537	2116	654	932	309	1322	1092	513	7475	15	25
6	1 6 22	290	1144	368	788	222	552	426	207	3997	8	3
7	23 24	81	316	138	228	69	68	54	29	983	2	17
8	11 13	122	468	296	464	148	0	0	0	1498	3	2
9	17 21	40	152	98	152	49	0	0	0	491	1	9
10	25 28	81	300	162	220	83	64	56	29	995	2	5
11	27 30	121	424	328	424	164	0	0	0	1461	3	39
12	32	37	136	92	136	46	0	0	0	447	1	53
13	2	35	140	20	32	10	88	84	38	447	1	53
14	26 29	323	1292	244	336	111	974	834	379	4493	9	7
15	31 34	181	684	338	480	165	240	230	102	2420	5	80
16	15 33	35	140	42	80	21	68	50	29	465	1	35
17	35 36	34	124	76	116	38	48	40	18	494	1	6
18	37 38	53	212	62	120	31	272	164	71	985	2	15
19	19 39	27	108	46	84	22	84	68	36	475	1	25
20	40 41	611	2444	624	1132	328	2376	1610	351	9476	19	24
21	42	24	96	26	40	11	98	88	16	399	1	101
22	43 44	157	596	334	528	167	94	66	39	1981	4	19
23	45	25	100	70	100	35	0	0	0	330	1	170
24	46	34	120	88	120	44	0	0	0	406	1	94
25	47 48 49	590	2324	1138	1864	568	740	1292	481	8997	18	3
26	50	34	136	68	136	34	68	68	34	578	2	422
Total		4070	15824	6356	10024	3207	8202	6764	2788	57235	117	1265

Table 5.8 : Station times for all models

i	p_{i1}	p_{i2}	p_{i3}	p_{i4}	p_{i5}	p_{i6}	p_{i7}	p_{i8}
1	39.25	38.75	20.50	18.75	21.00	38.00	29.00	32.50
2	32.33	31.00	24.17	18.17	25.17	56.00	22.00	40.17
3	40.00	37.00	50.00	37.00	50.00	0.00	0.00	0.00
4	41.40	37.00	49.00	31.40	49.40	7.00	4.60	9.00
5	35.80	35.27	21.80	15.53	20.60	44.07	36.40	34.20
6	36.25	35.75	23.00	24.63	27.75	34.50	26.63	25.88
7	40.50	39.50	34.50	28.50	34.50	17.00	13.50	14.50
8	40.67	39.00	49.33	38.67	49.33	0.00	0.00	0.00
9	40.00	38.00	49.00	38.00	49.00	0.00	0.00	0.00
10	40.50	37.50	40.50	27.50	41.50	16.00	14.00	14.50
11	40.33	35.33	54.67	35.33	54.67	0.00	0.00	0.00
12	37.00	34.00	46.00	34.00	46.00	0.00	0.00	0.00
13	35.00	35.00	10.00	8.00	10.00	44.00	42.00	38.00
14	35.89	35.89	13.56	9.33	12.33	54.11	46.33	42.11
15	36.20	34.20	33.80	24.00	33.00	24.00	23.00	20.40
16	35.00	35.00	21.00	20.00	21.00	34.00	25.00	29.00
17	34.00	31.00	38.00	29.00	38.00	24.00	20.00	18.00
18	26.50	26.50	15.50	15.00	15.50	68.00	41.00	35.50
19	27.00	27.00	23.00	21.00	22.00	42.00	34.00	36.00
20	32.16	32.16	16.42	14.89	17.26	62.53	42.37	18.47
21	24.00	24.00	13.00	10.00	11.00	49.00	44.00	16.00
22	39.25	37.25	41.75	33.00	41.75	11.75	8.25	9.75
23	25.00	25.00	35.00	25.00	35.00	0.00	0.00	0.00
24	34.00	30.00	44.00	30.00	44.00	0.00	0.00	0.00
25	32.78	32.28	31.61	25.89	31.56	20.56	35.89	26.72
26	17.00	17.00	17.00	17.00	17.00	17.00	17.00	17.00

Table 5.9 : Results from model sequencing

Unit #	Model #	
	Serial	Exhaustive
1	2	8
2	4	2
3	3	4
4	6	2
5	7	7
6	2	4
7	4	2
8	1	2
9	5	4
10	8	6
11	2	4
12	4	3
13	3	7
14	6	1
15	7	3
16	2	6
17	4	5

Table 5.10 : Station lengths for model sequence

Station i	Station length		
	Serial	Exhaustive	% Diff
1	55.7	52.6	5.9
2	66.4	60.7	9.4
3	86.6	66.0	31.1
4	84.2	61.6	36.6
5	64.9	51.7	25.4
6	46.3	47.8	-3.1
7	58.6	55.8	4.9
8	93.2	75.9	22.8
9	88.6	69.0	28.3
10	65.6	52.2	25.6
11	89.6	69.9	28.1
12	109.1	75.1	45.3
13	91.1	86.4	5.4
14	85.8	63.6	35.0
15	42.0	43.9	-4.2
16	59.2	59.4	-0.4
17	46.6	48.5	-3.9
18	83.0	76.8	8.0
19	55.0	56.6	-2.9
20	83.9	74.7	12.4
21	127.0	108.6	16.9
22	71.6	57.9	23.6
23	186.6	165.4	12.8
24	132.1	102.1	29.4
25	47.1	46.9	0.3
26	217.2	217.2	0.0
Total	2236.8	1946.4	14.9

packaging of units, is performed. Dar-El and Nadivi [26] decided that the need for the second operator at station 26 could be eliminated, if an operator in a station earlier in the assembly line could help with the additional work required at the last station. Station 23, which also has a lot of idle time, seemed like a good candidate. The modified work load for the stations are shown in Table 5.11 and the corresponding p_{ij} values are calculated in Table 5.12. The same sequence was obtained by the prolog sequence method, but this time with a line length of 1704 minutes. Thus, the reduction in the number of operators, idle time and assembly line length seems like a good tradeoff to make for the extra trouble of duplicating work element 50 at stations 23 and 26.

5.4 Comparison of results.

Finally, the results obtained in this report are compared with the results from the authors of [26]. For line balancing, Dar-El and Nadivi [26] utilized a modified version of Mansoor's [27] backtracking iterative method while a manual version of Dar-El and Cucuy's algorithm [21] was used for model sequencing. A side by side comparison of the results is shown in Table 5.13. A graphical comparison of all the solutions is shown in Figure 5.3. The serial solution performed the worst in this case as could be expected. The backtracking solution by Dar-El and Nadivi performed better and the modified exhaustive search algorithm provided the best solution. The improvement of the exhaustive algorithm over Dar-El and Nadivi's solution can be attributed to the fact that the exhaustive search algorithm did a lot more backtracking, thereby

Table 5.11 : Modified work load for all stations

i	Work Elements	Work load for each station 1	2	3	4	5	6	7	8	Total	Opera-tors	Resi-dual
1	3 8	157	620	164	300	84	304	232	130	1991	4	9
2	5 12	194	744	290	436	151	672	264	241	2992	6	8
3	7 10 14	40	148	100	148	50	0	0	0	486	1	14
4	16 18	207	740	490	628	247	70	46	45	2473	5	27
5	4 9 20	537	2116	654	932	309	1322	1092	513	7475	15	25
6	1 6 22	290	1144	368	788	222	552	426	207	3997	8	3
7	23 24	81	316	138	228	69	68	54	29	983	2	17
8	11 13	122	468	296	464	148	0	0	0	1498	3	2
9	17 21	40	152	98	152	49	0	0	0	491	1	9
10	25 28	81	300	162	220	83	64	56	29	995	2	5
11	27 30	121	424	328	424	164	0	0	0	1461	3	39
12	32	37	136	92	136	46	0	0	0	447	1	53
13	2	35	140	20	32	10	88	84	38	447	1	53
14	26 29	323	1292	244	336	111	974	834	379	4493	9	7
15	31 34	181	684	338	480	165	240	230	102	2420	5	80
16	15 33	35	140	42	80	21	68	50	29	465	1	35
17	35 36	34	124	76	116	38	48	40	18	494	1	6
18	37 38	53	212	62	120	31	272	164	71	985	2	15
19	19 39	27	108	46	84	22	84	68	36	475	1	25
20	40 41	611	2444	624	1132	328	2376	1610	351	9476	19	24
21	42	24	96	26	40	11	98	88	16	399	1	101
22	43 44	157	596	334	528	167	94	66	39	1981	4	19
23	45 50	30	118	79	118	40	9	9	5	408	1	92
24	46	34	120	88	120	44	0	0	0	406	1	94
25	47 48 49	590	2324	1138	1864	568	740	1292	481	8997	18	3
26	50	29	118	59	118	29	59	59	29	500	1	0
Total		4070	15824	6356	10024	3207	8202	6764	2788	57235	116	765

Table 5.12 : Modified station times for all models

i	Work Element	P_{i1}	P_{i2}	P_{i3}	P_{i4}	P_{i5}	P_{i6}	P_{i7}	P_{i8}
1	3 8	39.25	38.75	20.50	18.75	21.00	38.00	29.00	32.50
2	5 12	32.33	31.00	24.17	18.17	25.17	56.00	22.00	40.17
3	7 10 14	40.00	37.00	50.00	37.00	50.00	0.00	0.00	0.00
4	16 18	41.40	37.00	49.00	31.40	49.40	7.00	4.60	9.00
5	4 9 20	35.80	35.27	21.80	15.53	20.60	44.07	36.40	34.20
6	1 6 22	36.25	35.75	23.00	24.63	27.75	34.50	26.63	25.88
7	23 24	40.50	39.50	34.50	28.50	34.50	17.00	13.50	14.50
8	11 13	40.67	39.00	49.33	38.67	49.33	0.00	0.00	0.00
9	17 21	40.00	38.00	49.00	38.00	49.00	0.00	0.00	0.00
10	25 28	40.50	37.50	40.50	27.50	41.50	16.00	14.00	14.50
11	27 30	40.33	35.33	54.67	35.33	54.67	0.00	0.00	0.00
12	32	37.00	34.00	46.00	34.00	46.00	0.00	0.00	0.00
13	2	35.00	35.00	10.00	8.00	10.00	44.00	42.00	38.00
14	26 29	35.89	35.89	13.56	9.33	12.33	54.11	46.33	42.11
15	31 34	36.20	34.20	33.80	24.00	33.00	24.00	23.00	20.40
16	15 33	35.00	35.00	21.00	20.00	21.00	34.00	25.00	29.00
17	35 36	34.00	31.00	38.00	29.00	38.00	24.00	20.00	18.00
18	37 38	26.50	26.50	15.50	15.00	15.50	68.00	41.00	35.50
19	19 39	27.00	27.00	23.00	21.00	22.00	42.00	34.00	36.00
20	40 41	32.16	32.16	16.42	14.89	17.26	62.53	42.37	18.47
21	42	24.00	24.00	13.00	10.00	11.00	49.00	44.00	16.00
22	43 44	39.25	37.25	41.75	33.00	41.75	11.75	8.25	9.75
23	45 50	29.59	29.59	39.59	29.59	39.59	4.59	4.59	4.59
24	46	34.00	30.00	44.00	30.00	44.00	0.00	0.00	0.00
25	47 48 49	32.78	32.28	31.61	25.89	31.56	20.56	35.89	26.72
26	50	29.41	29.41	29.41	29.41	29.41	29.41	29.41	29.41

Table 5.13 : Comparison of case study results for line balancing

Station	Serial		Exhaustive		Dar-El	
i	T_i	$T-T_i$	T_i	$T-T_i$	T_i	$T-T_i$
1	492	8	491	9	472	28
2	489	11	492	8	447	53
3	458	42	486	14	492	8
4	486	14	473	27	458	42
5	496	4	475	25	485	15
6	478	22	497	3	458	42
7	424	76	483	17	456	44
8	475	25	498	2	424	76
9	356	144	491	9	499	1
10	345	155	495	5	495	5
11	425	75	461	39	483	17
12	288	212	447	53	497	3
13	454	46	447	53	493	7
14	493	7	493	7	459	41
15	459	41	420	80	447	53
16	447	53	465	35	438	62
17	438	62	494	6	479	21
18	479	21	485	15	438	62
19	438	62	475	25	341	159
20	341	159	476	24	307	193
21	307	193	399	101	476	24
22	476	24	481	19	399	101
23	399	101	330	170	481	19
24	481	19	406	94	330	170
25	330	170	497	3	406	94
26	406	94	78	422	497	3
27	497	3			78	422
28	78	422				
Total	11735	2265	11735	1265	11735	1765

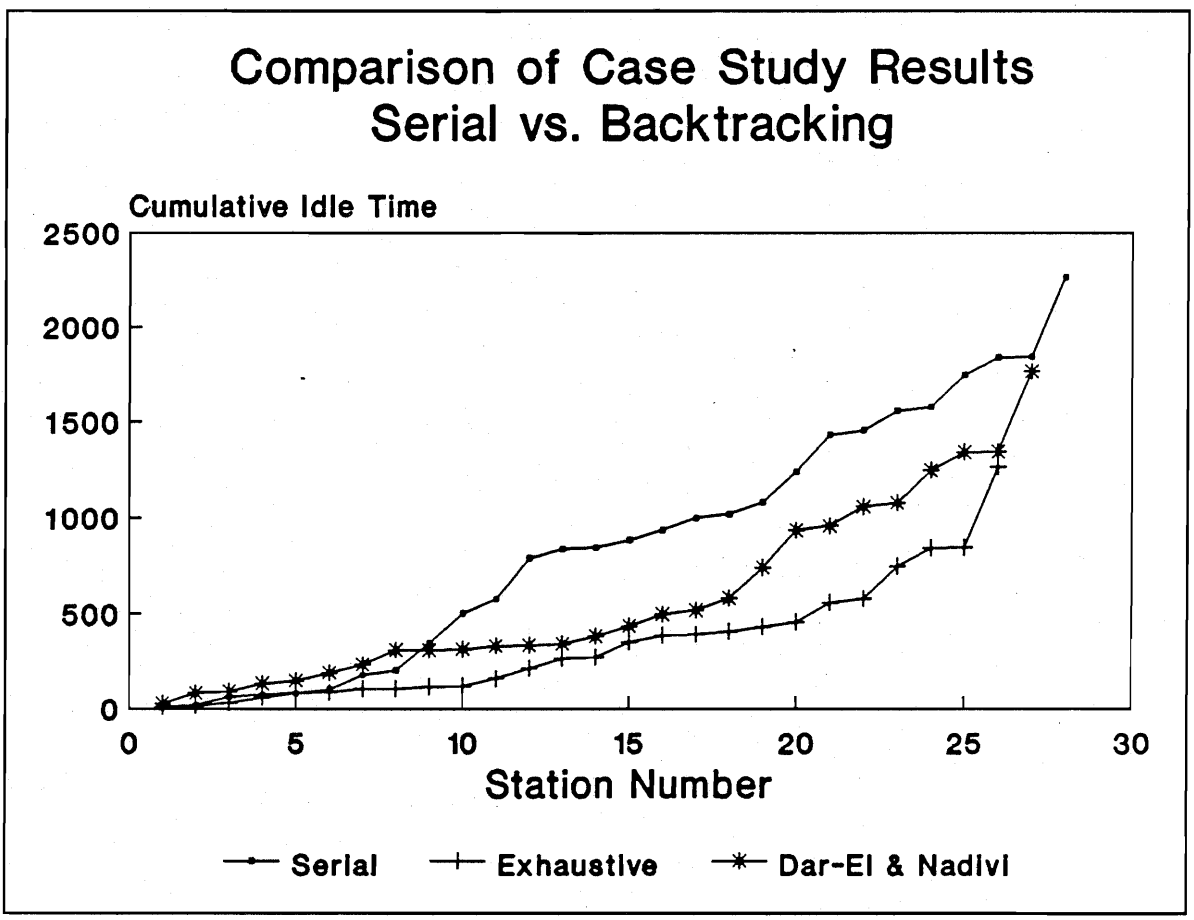


Figure 5.3 : Comparison of case study results

evaluating a larger number of line balance configurations. A full exhaustive search can be expected to provide an even better solution.

A final summary of the results, including the assembly line length for each configuration, is shown in Table 5.14. It can be seen that the results for the exhaustive approach are indeed an improvement over those obtained by the authors of [26]. The use of the exhaustive search algorithm resulted in a reduction of 40% in idle time, 17% in assembly line length and one station less for the assembly line. The reduction in assembly line length over the Dar-El and Nadivi solution can be attributed to the fact that their solution was an adaptation of a single model line balancing technique. No attempt was made to smooth the station times between the different models. As shown in the previous chapter, the ability of the exhaustive algorithm to smooth the station times will result in shorter assembly lines.

Table 5.14 : Summary of results

Solution method	Number of stations	Number of operators	Total Idle time	Assembly line length
Serial	28	119	2265	2689
Dar-El & Nadivi[1]	27	117	1265	2051
Prolog	26	117	1265	1946
Prolog[1]	26	116	765	1704

1) Work element 50 (packaging) is duplicated to reduce the number of operators by one.

Chapter 6: Conclusions.

This research project investigated the mixed model line balancing and model sequencing problem. Since most of the previous work concentrated on serial or sequential techniques, the objective of this project was to develop a backtracking exhaustive search algorithm for line balancing and model sequencing on mixed model lines.

It was demonstrated that a non-sequential or backtracking approach to line balancing and model sequencing can lead to significant improvements in the design and performance of mixed model assembly lines. The backtracking algorithms found better optimal solutions than their serial counterparts in all cases examined in this report. The case study showed that the algorithms could be applied to a real world manufacturing problem.

It was also shown that Prolog is an ideally suited computer language for the computerization of backtracking search algorithms. Prolog's built-in backtracking mechanism resulted in efficient solutions that allowed relatively large problems to be solved on 80286 personal computers.

The use of these algorithms in the design of mixed model assembly systems will result in substantial savings in operator idle time, in-process inventory and labor costs.

A closer integration of line balancing and model sequencing is recommended as future work. In this project, the line was first balanced and then the model sequence

was optimized according to some objective function. A worthwhile extension of this project would be to investigate the possibilities of backtracking between line balancing and model sequencing.

Additional recommendations for future work are:

- To improve the exhaustive search algorithm by adding more heuristic rules.
- Additional objective functions can also be incorporated in the search engine.
- The capability to allow different types of station on the same assembly line.

References.

1 Wild, R., 1972, *Mass-Production Management* (London: John Wiley).

2 Macaskill, J. L. C., 1969, Application of Computers to the Analysis of Mixed-product Assembly Lines. *Proceedings of Fourth Australian Computer Conference*, Adelaide, South Australia.

3 Buxey, G. M., Slack, N. D., and Wild, R., 1973, Production flow line system design - a review. *AIIE Transactions*, **5**, March.

4 Groover, M. P., 1987, *Automation, Production Systems, and Computer Integrated Manufacturing* (New Jersey: Prentice-Hall).

5 Thomopoulos, N. T., 1970, Mixed model line balancing with smoothed station assignments. *Management Science*, **16**, 9.

6 Borland International, 1988, *Turbo Prolog User's Guide Version 2.0* (Scotts Valley).

7 Blanchard, B. S., and Fabrycky, W. J., 1981, *Systems Engineering and Analysis* (New Jersey: Prentice-Hall).

8 Ghosh, S., and Gagnon, R. J., 1989, A comprehensive literature review and analysis of the design, balancing and scheduling of assembly systems. *International Journal of Production Research*, **27**, 4.

9 Macaskill, J. L. C., 1972, Production-line balances for mixed-model lines. *Management Science*, **19**, 4.

10 Thomopoulos, N. T., 1967, Line Balancing-sequencing for Mixed-model Assembly. *Management Science*, **14**, 2.

11 Kilbridge, M. D., and Wester, L., 1961, A Heuristic Method of Assembly Line Balancing. *Journal of Industrial Engineering*, **12**, 4.

12 Helgeson, W. B., and Birnie, D. P., 1961, Assembly Line Balancing Using Ranked Positional Weight Technique, *Journal of Industrial Engineering*, **12**, 6.

13 Kilbridge, M. D., and Wester, L., 1964, The Assembly line model-mix sequencing problem, *Proceedings of the 3d International Conference of Operations Research*, Paris, France.

14 Bowman, E. H., 1960, Assembly Line Balancing by Linear Programming. *Operations Research*, **8**, 3.

15 Jackson, J. R., 1956, A Computing Procedure for a Line Balancing Problem. *Management Science*, **2**, 3.

16 Salveson, M. E., 1955, The Assembly Line Balancing Problem. *Journal of Industrial Engineering*, **6**, 3.

17 Moodie, C. L., and Young, H. H., 1965, A Heuristic Method of Assembly-Line Balancing for Assumptions of Constraint on Variable Work Element Times. *Journal of Industrial Engineering*, **16**, 1.

18 Chakravarty, A. K., and Shtub, A., 1985, Balancing mixed model lines with in-process inventories. *Management Science*, **31**, 9.

19 Macaskill, J. L. C., 1973, Computer Simulation for mixed-model Production Lines. *Management Science*, **20**, 3.

20 Dar-El, E. M., and Cother, R. F., 1975, Assembly line sequencing for model mix. *International Journal of Production Research*, **13**, 5.

21 Dar-El, E. M., and Cucuy, S., 1977, Optimal Mixed-Model Sequencing for Balanced Assembly Lines. *Omega*, **5**, 3.

22 Okamura, K., and Yamashina, H., 1979, A heuristic algorithm for the assembly line model-mix sequencing problem to minimize the risk of stopping the conveyor. *International Journal of Production Research*, **17**, 3.

23 Arcus, A. L., 1966, COMSOAL - A Computer Method of Sequencing Operations for Assembly Lines. *International Journal of Production Research*, **4**, 4.

24 Magad, E. L., 1972, Cooperative Manufacturing Research. *Industrial Engineering*, **4**, 1.

25 Schofield, N. A., 1979, Assembly Line Balancing and the Application of Computer Techniques. *Computer and Industrial Engineering*, **3**, 1.

26 Dar-El, E. M., and Nadivi, A., 1981, A mixed-model sequencing application. *International Journal of Production Research*, **19**, 1.

27 Mansoor, E. M., 1964, Assembly line balancing - an improvement on the ranked positional weight, *Journal of Industrial Engineering*, **15**, 2.

Appendix A: Computer program listing.

```
/*************************************************************************
                              PROGRAM1.PRO
*************************************************************************
                      Exhaustive search Algorithm
                      ---------------------------

This Prolog program finds all the feasible solutions for balancing a
mixed-model production line. The lines with the lowest difference in
work content, the smoothest stations and the lowest variance is saved
and displayed at the end of program execution. When more than one line
has the same value, then both lines will be stored as a minimum. This
version has the modified search algorithm which speeds up search.

To execute the program, type go at the goal prompt. To redisplay the
results at the end of a program, or after a control-break, just type :
write_final after the goal prompt. To redirect output to a file or to
the printer, hit Alt-P.

The format of the input datafile is as follows:

[]          Prededence restrictions, immediate predecessors
[1,2,3]     of a station in square brackets separated by
            spaces. Station with no predecessors have empty
            brackets. Only one station per line.

0.3 4.0 2.3     The work element times in any format,
                separated by at least one space.

120 40 60       The model quantities in any format,
                separated by at least one space.

See SAMPLE1.DAT for an example of an input data file

written by P. R. Smith, November 1990. Tested with Turbo Prolog v2.0
*************************************************************************/

constants
  input_fname = "SAMPLE1.DAT"  % input filename
  n_elements = 19              % number of work elements
  n_models   =  3              % number of different models
  nn = 3                       % number of stations
  middle = 414.0               % average value for station work content
                               % or shift time
  upper = 0.0144928            % upper tolerance for station
                               % work content
  lower = 0.0144928            % lower tolerance for station
                               % work content
/*************************************************************************/

domains
  intlist = integer*
  reallist = real*
  s = s(integer,intlist,real,real,real,real,reallist)
                       % station object
  slist = s*           % list of station objects
/*************************************************************************/

database
  d(real,real,real)         % used for temporary storage
  station(s)                % stores info about station in
                            % internal database
  tts(integer,real)         % average time for model i (Pj)
  node(integer,intlist)     % element j with list of
                            % predecessors from precedence diagram
  q(integer,integer)        % production quantity for model j (Nj)
  te(integer,integer,real) % work time of element k on model j  (tjk)
  min(integer,real)         % minimum value for delta
```

```
  ass_line(integer,slist)  % store a complete assembly line
  t(integer,integer,integer,integer) % used for timing
                                     % elapsed CPU time.
  g(real,real,intlist,reallist)      % used for temporary storage
/*************************************************************************/

predicates
  route(intlist,intlist,intlist,integer)
  member(integer,intlist)
  append(intlist,intlist,intlist)
  subset(intlist,intlist)
  newnode(integer,intlist)
  writelist(intlist)
  writelist2(reallist)
  sumlist(reallist,real)
  var(reallist,real,real,integer,real)
  get(integer,real,intlist)
  get2(intlist)
  get3(real,real,intlist,reallist)
  get0(integer,intlist)
  check_limits(real)
  check_min(integer,real,real)
  save_info(integer,intlist,real,real,reallist)
  write_output
  write_final
  write_final1
  write_final2
  write_final3
  write_elems(slist)
  write_station(slist,real,real,real,real)
  write0
  write1
  write2
  get_tts
  input
  timer
  go

/*************************************************************************/
clauses

/*************************************************************************
go is the main clause in this program. It clears the database, reads
the input file and starts the searching procedure. After the search
procedure, the solutions are displayed and the elapsed CPU time.
*************************************************************************/
go :-
  clearwindow,                % clear the screen
  retractall(_),              % clear internal database
  timer,                      % start timer
  input,                      % get input from file
  get_tts,                    % calculate averages
  route([],[],[0],1).         % start routing sequence
go :-
  write_final,                % write final output
  timer.                      % write cpu time

/*************************************************************************
check_limits is called with TT bound and checks if TT falls between the
limits specified by the user. The upper and lower are specified as
fractions not to exceed the middle value or shift time
*************************************************************************/
check_limits(TT) :-
  TT <= middle*(1+upper),     % check upper limit
  TT >= middle*(1-lower).     % check lower limit

/*************************************************************************
```

```
get_tts calculates the average or "ideal" time for model i for a
station. (Pi)
Pi := (Sum of all element times) x (# of units) / (# of stations)
*************************************************************************/
get_tts :-
  q(I,QQ),                    % get production quantity for model i
  findall(X,te(_,I,X),List),  % find all element times for model i ...
  sumlist(List,Sum),          % ... sum them all
  Temp = Sum*QQ/nn,           % calculate Pi
  assert(tts(I,Temp)),        % put value in internal database
  fail.                       % fail in order to backtrack
                              % through all models
get_tts.                      % returns after all models

/*************************************************************************
input uses the external program INPUT.EXE to write data in the correct
format to load into the internal database. INPUT.EXE Must be present in
default directory or accessible through PATH statement. For the format
of the datafile, see SAMPLE1.DAT
*************************************************************************/
input :-
  format(Outstring,"input.exe  %  temp    %     %",input_fname,
                                                  n_elements,n_models),
  system(Outstring),    % execute INPUT
  consult("temp").      % load facts in internal database

/*************************************************************************
get is called with N and List bound and returns the element time X for
model N for every element in the list.
*************************************************************************/
get(N,X,List) :-
  te(I,N,X),      % get element time X for model N element I
  member(I,List). % check if I is a member of the given list
                  % no backtracking here!

/*************************************************************************
get2 is called with List bound and computes the work content TT, the
"smoothness" indicator Delta and the station times for a MODEL only.
These 3 values are then stored in the internal database to be retrieved
by get3
*************************************************************************/
get2(List) :-
  q(I,QQ),                        % get first model I and quantity QQ
  findall(X,get(I,X,List),List2), % put all element times in List2
  sumlist(List2,Sum),             % sum all element times
  TT = QQ*Sum,                    % calculate total work
                                  % content for model
  tts(I,Temp),                    % get average for this model
  Delta =  abs(Temp-TT),          % calculate difference
  assertz(d(TT,Delta,Sum)),       % insert into database
  fail.                           % fail to next model
get2(_).                          % return when all models done

/*************************************************************************
get3 is called with List bound and returns Total_T and Total_Delta
which is the sum of TT and Delta respectively (from get2) for ALL
models. Times is a list which returns the station times for all models
*************************************************************************/
get3(Total_T,Total_Delta,List,Times) :-
  retractall(d(_,_,_)),             % clear database first
  get2(List),                       % calculate values for all models
  findall(TT,d(TT,_,_),List2),      % find all work
                                    % content times
  findall(Delta,d(_,Delta,_),List3), % find all deltas
  findall(Sum,d(_,_,Sum),Times),    % put station times in a list
  sumlist(List2,Total_T),           % sum the list
  sumlist(List3,Total_Delta).       %     "
```

```
/*************************************************************************
newnode is called with Current (list of nodes) bound and returns all
nodes NOT already in Current for which all precedence requirements have
been satisfied, that is all precedence nodes are in Current.
*************************************************************************/
newnode(I,Current) :-
  node(I,Predlist),         % get node I and it's list of predecessors
  not(member(I,Current)),   % is I already in Current?
  subset(Predlist,Current). % are all predecessors in Current?

/*************************************************************************
subset is called with both lists bound and succeeds if the first list
is a subset of the second list, that is if all elements in the first
list are included in the second list.
*************************************************************************/
subset([],_).             % trivial case
subset([H|T],L) :-
  member(H,L),            % is H a member of list L
                          % this member does not backtrack!
  subset(T,L).            % if yes then test rest of list

/*************************************************************************
get0 is called with List bound and returns all nodes which are not in
List.
*************************************************************************/
get0(N,List) :-
  node(N,_),              % get any node N
  not(member(N,List)).    % is N a member of given List?

/*************************************************************************
save_info stores the information about a station in the internal
database to be retrieved later by the output routine.
syntax : save_info(Station #, List of Elements, Work Content,
                   Difference, Delta,List of model station times)
*************************************************************************/
save_info(N,_,_,_,_) :-
    % first remove the old values if existing
  retract(station(s(N,_,_,_,_,_,_))),
  fail.
save_info(N,Elm_list,Work_content,Delta,Model_list) :-
  Difference = abs(middle-Work_content),            % calc difference
  var(Model_list,0,0,0,Var),            % get variance of station times
  assertz(station(s(N,Elm_list,Work_content,Difference,Delta,Var,Model_list))).
                                        % insert new value in database

/*************************************************************************
write_output is called when the last station is reached and valid, i.e.
station limits are OK. write_output writes the intermediate results and
also updates the minimum values
*************************************************************************/
write_output :-
  write0,write1,write2.

write0 :-                   % writes memory status
  clearwindow,              % clear screen
  storage(S,H,T),           % get memory status
    % write status
  write("\nStacksize = ",S,"  Heapsize = ",H,"  Trailsize = ",T),
  nl.

write1 :-                       % writes the elements in every station
  write("\n Station   Work Elements"), % write heading
  fail.
write1 :-
  station(s(I,List,_,_,_,_,_)),           % get a station
  nl,write("   ",I,"       "),            % write station #
  writelist(List),                        % write work elements
```

```
  fail.                                   % backtracks to next station
write1.                                   % return after all stations

write2 :-                                 % writes rest of information
    % write heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  fail.                                   % backtrack to second clause
write2 :-
  station(s(I,_,Work,Difference,Delta,Var,Times)),   % get a station
    % write station information
  writef("\n %  %7.2  %7.2  %7.2  %7.2",I,Work,Difference,Delta,Var),
  writelist2(Times),                      % write station times
  fail.                                   % backtracks to next station
write2 :-
    % after all stations are written, go to this clause and
    % write totals and update minimum values
    % first find all stations with findall
  findall(Work,station(s(_,_,Work,_,_,_,_)),List1),
  findall(Delta,station(s(_,_,_,_,Delta,_,_)),List2),
  findall(Difference,station(s(_,_,_,Difference,_,_,_)),List3),
  findall(Var,station(s(_,_,_,_,_,Var,_)),List4),
  sumlist(List1,Sum1),sumlist(List2,Sum2),              % sum lists
  sumlist(List3,Sum3),sumlist(List4,Sum4),
  write("\n-------------------------------------------------------------------------------"),
    % write out the totals
  writef("\n    %7.2  %7.2  %7.2  %7.2  \n\n",Sum1,Sum3,Sum2,Sum4),
    % now update minimum values
  check_min(1,Sum2,Min_delta),          % update delta
  check_min(2,Sum3,Min_diff),           % update difference
  check_min(3,Sum4,Min_var),            % update variance
    % write minimum values
  writef("Min values:  %7.2  %7.2  %7.2 \n\n",Min_diff,Min_delta,Min_var).

/*************************************************************************
check_min compares the value with the minimum value found and if it's
less it gets stored as the new minimum value. If the minimum value is
the same, it gets stored as well
*************************************************************************/
check_min(I,Value,Min) :-
  min(I,Min),                     % get value
  Value > Min,                    % compare
  !.                              % if bigger exit, do not backtrack
check_min(I,Value,Value) :-
  min(I,Min),                     % get current minimum
  abs(Min-Value) < 1.0E-10,       % equal value so store also
  !,                              % do not backtrack
  findall(X,station(X),List),     % find all stations for this line
  asserta(ass_line(I,List)).      % insert into database
check_min(I,Value,Value) :-       % new value is smaller than
                                  % current minimum
  retractall(min(I,_)),           % retracts old value
  asserta(min(I,Value)),          % put new value in database
  retractall(ass_line(I,_)),      % retract old assembly line
  findall(X,station(X),List),     % find all stations for new line
  asserta(ass_line(I,List)).      % insert new minimum value in database

/*************************************************************************
write_final writes the station with the minimum values found. It is
called after all combinations have been tried
*************************************************************************/
write_final :- write_final1,
               write_final2,
               write_final3.

write_final1 :-                   % write lines with smoothest stations
  ass_line(1,X),                  % get assm. line with smoothest stations
    % write headings
```

```
  write("\n\nThe assembly with the smoothest stations (minimum delta) : "),
  write("\n\n Station   Work Elements"),
  write_elems(X),              % write work elements
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  write_station(X,0,0,0,0),    % write station information
  fail.                        % fail for next line
write_final1.                  % return after all lines done

write_final2 :-    % write lines with lowest difference in
                   % work content for all stations
  ass_line(2,X),   % get assembly line
    % write headings
  write("\n\nThe assembly with the lowest difference in work content : "),
  write("\n\n Station   Work Elements"),
  write_elems(X),           % write work elements
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  write_station(X,0,0,0,0), % write station information
  fail.                     % fail for next line
write_final2.               % returns after all lines done

write_final3 :-   % write lines with lowest variance
  ass_line(3,X),  % get assembly line
    % write headings
  write("\n\nThe assembly with the lowest variance in station times : "),
  write("\n\n Station   Work Elements"),
  write_elems(X),            % write work elements
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  write_station(X,0,0,0,0),  % write station information
  fail.                      % fail for next line
write_final3.                % returns after all lines done

/*************************************************************************
write_elems writes out work elements of a station
*************************************************************************/
write_elems([]).                                % empty list, so done.
write_elems([s(I,Elements,_,_,_,_,_)|T]) :-     % non empty list
  write("\n   ",I,"       "),                   % write station #
  writelist(Elements),                          % write work elements
  write_elems(T).                               % write rest of list

/*************************************************************************
write_station writes out the information for a line. It is called
recursively until all stations are done. At this point all totals are
written.
*************************************************************************/
write_station([],T_work,T_diff,T_delta,T_var) :-
    % write rest of station info - all stations are done
  write("\n-------------------------------------------------------------------------------"),
    % write station parameters
  writef("\n    %7.2  %7.2  %7.2  %7.2   \n\n",T_Work,T_diff,T_delta,T_var).
  write_station([s(I,_,Work,Difference,Delta,Var,Times)|T],Sum1,Sum2,Sum3,Sum4) :-
    % writes first element in the list
  writef("\n %  %7.2  %7.2  %7.2  %7.2",I,Work,Difference,Delta,Var),
  writelist2(Times),  % write station times
  SSum1 = Sum1 + Work,                % sum the work content
  SSum2 = Sum2 + Difference,          % sum the differences
  SSum3 = Sum3 + Delta,               % sum the delta values
  SSum4 = Sum4 + Var,                 % sum the station variances
    % write remainder of the stations in the line
  write_station(T,SSum1,SSum2,SSum3,SSum4).

/*************************************************************************
The route clause is the search engine which generates all possible
combinations. The syntax is as follows:
```

```
                route(List1,List2,List3,N)
where: List1 = list of elements (nodes) already assigned in PREVIOUS
               stations
       List2 = list of elements considered or assigned for this station
       List3 = maintenance list to avoid permutations of same list
           N = station number (integer)
*************************************************************************/
route(Final,_,_,nn) :-                  % go here for last station
  findall(N,get0(N,Final),Final2),      % find all remaining stations
  get3(TT,Delta,Final2,Times),          % get total time and delta
  check_limits(TT),                     % checks limits
  save_info(nn,Final2,TT,Delta,Times), % store station info
  write_output,                         % write ouput and
                                        % update minimums
  fail.                                 % backtrack
route(Ass,[H|T],_,I) :-
    % this clause checks if TT is within limits, if yes,
    % the station is accepted and the next station is
    % started. If no, the clause fails to the next one
  I < nn,                               % checks if not final station
  g(TT,Delta,[H|T],Times),              % get deltas and total time
  retractall(g(_,_,_,_)),               % remove from database
  check_limits(TT),                     % checks limits
  save_info(I,[H|T],TT,Delta,Times),    % store station info
  append(Ass,[H|T],Ass2),               % append this list to already
                                        % assigned list (List1)
  I2 = I + 1,                           % increment station counter
  route(Ass2,[],[0],I2).                % start search for next station
route(Ass,L,[H|T],N) :-
    % this clause generates the combinations : it either
    % chooses the next element out of the list of possible
    % elements or backtracks to the member clause to find
    % a new combination
  N < nn,                             % checks if not final station
  append(Ass,L,Ass2),                 % append L to already assigned list
  findall(I,newnode(I,Ass2),Avail), % find list of all possible
                                      % elements to assign
  member(X,Avail),                    % get any member of the list
  X>H,                                % checks for permutation of
                                      % same station
  append(L,[X],L2),                   % append X to current station list
  get3(TT,Delta,L2,Times),            % calculate total time
                                      % and delta values
  asserta(g(TT,Delta,L2,Times)),      % store in database
  TT <= middle*(1+upper),             % if work cont. is smaller than
                                      % upper limit goto next station
                                      % else backtrack to member clause
  route(Ass,L2,[X,H|T],N).            % go to second route clause, to
                                      % check if this list can make a
                                      % station, note that station
                                      % counter is not incremented.

/*************************************************************************
The member clause is a very important part of the search algorithm.
There is no cut so all possible elements can be obtained by
backtracking.
*************************************************************************/
member(X,[X|_]).                % check first element in list (head)
member(X,[_|T]) :-
  member(X,T).                  % check rest of the list (tail)

/*************************************************************************
append adds two lists to form a third.
syntax : append(List1,List2,List3). List3 := List1 + List2
*************************************************************************/
append([], List, List).                 % trivial case
append([X|L1], List2, [X|L3]) :-
```

```
  append(L1, List2, L3).            % recursive call

/************************************************************************
sumlist sums the elements of a list of reals
************************************************************************/
sumlist([],0).                      % empty list
sumlist([H|T],S):-                  % non-empty list
  sumlist(T,S2),                    % sum tail of list
  S = S2 + H.

/************************************************************************
writelist writes out a list separated with spaces
************************************************************************/
writelist([]).                      % empty list
writelist([H|T]) :-                 % non-empty list
  write(H," "),                     % write head of list
  writelist(T).                     % write tail of list

/************************************************************************
writelist2 writes out a list formatted in a specific way
************************************************************************/
writelist2([]).                 % empty list
writelist2([H|T]) :-            % non-empty list
  writef("%6.2",H),             % write head of list
  writelist2(T).                % write tail of list

/************************************************************************
var calculates the variance of a list
************************************************************************/
var([],Sx,Sxx,N,Var) :-         % empty list
  Var = Sxx/N - Sx*Sx/N/N.      % get variance
var([H|T],Sx,Sxx,N,Var) :-      % non-empty list
  Sx2 = Sx + H,                 % sum of elements
  Sxx2 = Sxx + H*H,             % sum of squared elements
  N2 = N + 1,                   % increase counter
  var(T,Sx2,Sxx2,N2,Var).       % remainder (tail) of list

/************************************************************************
timer calculates the elapsed CPU time. When called the first time, the
start time is stored in the database. When called the second time, the
difference between the start time and the current time is computed and
displayed.
************************************************************************/
timer :-
    t(H1,M1,S1,D1),             % if it's the second time called then
    !,                          % do not backtrack to second clause
      % calculate start time in seconds
    Time1 = D1/100 + S1 + M1*60 + H1*3600.0,
    time(H2,M2,S2,D2),                          % get ending time
      % calculate end time in seconds
    Time2 = D2/100 + S2 + M2*60 + H2*3600.0,
    Diff = Time2 - Time1,                       % get difference
    Hours = trunc(Diff/3600),                   % calc. hours
    Diff2 = Diff - Hours*3600.0,
    Min = trunc(Diff2/60),                      % calc. minutes
    Diff3 = Diff2 - Min*60,
    Sec = Diff3,                                % calc. seconds
      % write elapsed time
    writef("\n\nElapsed CPU time :  %3 Hours %3 Minutes %5.2 Seconds\n",Hours,Min,Sec).
timer :-                                % if it's first time
    time(H1,M1,S1,D1),                  % get current time
    asserta(t(H1,M1,S1,D1)).            % store in database
```

```
/*****************************************************************************
                                PROGRAM2.PRO
*****************************************************************************
                          Serial search algorithm
                          -----------------------

This Prolog program balances a mixed-model production line by a serial
method. Stations are taken one at a time and all possible combinations
for that station generated. The station with the lowest work difference
or smoothest stations is then saved and the next station is chosen.
Unlike the first program, no backtracking is done once a station is
saved.

The user specifies whether the program should choose the station with
the lowest difference in work content (choice=2) that is the most
perfectly balanced line or the smoothest stations (choice=1) that is the
line with the lowest total delta value. This version has the modified
search algorithm which speeds up search.

To execute the program, type go at the goal prompt. To redisplay the
results at the end of a program, or after a control-break, just type :
write_output after the goal prompt. To redirect output to a file or to
the printer, hit Alt-P.

The format of the input datafile is as follows:

[]          Prededence restrictions, immediate predecessors
[1,2,3]     of a station in square brackets separated by
            spaces. Station with no predecessors have empty
            brackets. Only one station per line.

0.3 4.0 2.3    The work element times in any format,
               separated by at least one space.

120 40 60      The model quantities in any format,
               separated by at least one space.

See SAMPLE1.DAT for an example of an input data file

written by P. R. Smith, November 1990. Tested with Turbo Prolog v2.0
*****************************************************************************/
constants
  input_fname = "SAMPLE1.DAT"  % input filename
  n_elements = 19              % number of work elements
  n_models   =  3              % number of different models
  nn = 3                       % number of stations
  middle = 414.0               % average value for station work content
                               % or shift time
  upper = 0.0144928            % upper tolerance for station
                               % work content
  lower = 0.0144928            % lower tolerance for station
                               % work content
  choice = 1                   % 1 = minimize deltas
                               % 2 = minimize total work
/*****************************************************************************/
domains
  intlist = integer*
  reallist = real*
  s = s(integer,intlist,real,real,real,real,reallist)
                        % station object
/*****************************************************************************/
database
  d(real,real,real)          % used for temporary storage
  station(s)                 % stores info about station in
                             % internal database
  tts(integer,real)          % average time for model i (Pj)
  node(integer,intlist)      % element j with list of predecessors from
```

```
                          % precedence diagram
  q(integer,integer)      % production quantity for model j (Nj)
  te(integer,integer,real) % work time of element k on model j  (tjk)
  min(real,real)          % minimum value for delta and difference
  t(integer,integer,integer,integer)  % used for timing
                                      % elapsed CPU time.
  g(real,real,intlist,reallist)       % used for temporary storage

/*************************************************************************/
predicates
  route(intlist,intlist,intlist,integer)
  serial(integer,intlist)
  member(integer,intlist)
  append(intlist,intlist,intlist)
  makelist(intlist,intlist,integer)
  subset(intlist,intlist)
  newnode(integer,intlist)
  writelist(intlist)
  writelist2(reallist)
  sumlist(reallist,real)
  var(reallist,real,real,integer,real)
  get(integer,real,intlist)
  get2(intlist)
  get3(real,real,intlist,reallist)
  get0(integer,intlist)
  check_limits(real)
  check_min(integer,real,real,integer)
  save_info(integer,intlist,real,real,real,reallist)
  write_output
  write_station(integer,real,real,real,intlist,reallist)
  write0(integer)
  write1
  write2
  get_tts
  input
  go
  timer
/*************************************************************************/

clauses
/*************************************************************************
go is the main clause in this program. It clears the database, reads
the input file and starts the searching procedure. After the search
procedure, the elapsed CPU time is displayed.
*************************************************************************/
go :-
  clearwindow,                % clear the screen
  retractall(_),              % clear internal database
  timer,                      % start timer
  input,                      % get input from file
  get_tts,                    % calculate averages
  serial(1,[]),               % do serial routing
  timer.                      % calculate execution time

/*************************************************************************
check_limits is called with TT bound and checks if TT falls between the
limits specified by the user. The upper and lower are specified as
fractions not to exceed the middle value or shift time
*************************************************************************/
check_limits(TT) :-
  TT <= middle*(1+upper),     % check upper limit
  TT >= middle*(1-lower).     % check lower limit

/*************************************************************************
get_tts calculates the average or "ideal" time for model i for a
station. (Pi)
Pi := (Sum of all element times) x (# of units) / (# of stations)
```

```
*************************************************************************/
get_tts :-
  q(I,QQ),                    % get production quantity for model i
  findall(X,te(_,I,X),List),  % find all element times for model i ...
  sumlist(List,Sum),          % ... sum them all
  Temp = Sum*QQ/nn,           % calculate Pi
  assert(tts(I,Temp)),        % put value in internal database
  fail.                       % fail in order to backtrack
                              % through all models
get_tts.                      % returns after all models

/*************************************************************************
input uses the external program INPUT.EXE to write data in the correct
format to load into the internal database. INPUT.EXE Must be present in
default directory or accessible through PATH statement. For the format
of the datafile, see SAMPLE1.DAT
*************************************************************************/
input :-
  format(Outstring,"input.exe  %  temp    %     %",input_fname,
                                                  n_elements,n_models),
  system(Outstring),    % execute INPUT
  consult("temp").      % load facts in internal database

/*************************************************************************
get is called with N and List bound and returns the element time X for
model N for every element in the list.
*************************************************************************/
get(N,X,List) :-
  te(I,N,X),      % get element time X for model N element I
  member(I,List). % check if I is a member of the given list
                  % no backtracking here!

/*************************************************************************
get2 is called with List bound and computes the work content TT, the
"smoothness" indicator Delta and the station times for a MODEL only.
These 3 values are then stored in the internal database to be retrieved
by get3
*************************************************************************/
get2(List) :-
  q(I,QQ),                        % get first model I and quantity QQ
  findall(X,get(I,X,List),List2), % put all element times in List2
  sumlist(List2,Sum),             % sum all element times
  TT = QQ*Sum,                    % calculate total work
                                  % content for model
  tts(I,Temp),                    % get average for this model
  Delta =  abs(Temp-TT),          % calculate difference
  assertz(d(TT,Delta,Sum)),       % insert into database
  fail.                           % fail to next model
get2(_).                          % return when all models done

/*************************************************************************
get3 is called with List bound and returns Total_T and Total_Delta
which is the sum of TT and Delta respectively (from get2) for ALL
models. Times is a list which returns the station times for all models
*************************************************************************/
get3(Total_T,Total_Delta,List,Times) :-
  retractall(d(_,_,_)),             % clear database first
  get2(List),                       % calculate values for all models
  findall(TT,d(TT,_,_),List2),      % find all work
                                    % content times
  findall(Delta,d(_,Delta,_),List3), % find all deltas
  findall(Sum,d(_,_,Sum),Times),    % put station times in a list
  sumlist(List2,Total_T),           % sum the list
  sumlist(List3,Total_Delta).       %     "

/*************************************************************************
newnode is called with Current (list of nodes) bound and returns all
```

```
nodes NOT already in Current for which all precedence requirements have
been satisfied, that is all precedence nodes are in Current.
*************************************************************************/
newnode(I,Current) :-
  node(I,Predlist),          % get node I and it's list of predecessors
  not(member(I,Current)),    % is I already in Current?
  subset(Predlist,Current). % are all predecessors in Current?

/*************************************************************************
subset is called with both lists bound and succeeds if the first list
is a subset of the second list, that is if all elements in the first
list are included in the second list.
*************************************************************************/
subset([],_).              % trivial case
subset([H|T],L) :-
  member(H,L),             % is H a member of list L
                           % this member does not backtrack!
  subset(T,L).             % if yes then test rest of list

/*************************************************************************
get0 is called with List bound and returns all nodes which are not in
List.
*************************************************************************/
get0(N,List) :-
  node(N,_),               % get any node N
  not(member(N,List)).  % is N a member of given List?

/*************************************************************************
save_info stores the information about a station in the internal
database to be retrieved later by the output routine. syntax :
save_info(Station #, List of Elements, Work Content, Difference,
          Delta,List of model station times )
*************************************************************************/
save_info(N,_,_,_,_,_) :-
    % first remove the old values if existing
  retract(station(s(N,_,_,_,_,_,_))),
  fail.
save_info(N,Elm_list,Work_content,Difference,Delta,Model_list) :-
  var(Model_list,0,0,0,Var),         % get variance of station times
    % insert new value in database
  assertz(station(s(N,Elm_list,Work_content,Difference,Delta,Var,Model_list))).

/*************************************************************************
write_output is called when the last station is reached. It writes the
ouput for all stations
*************************************************************************/
write_output :-
  write0(choice),
  write1,
  write2.

  % writes the correct heading for type of optimization chosen
write0(1) :-
  write("\n\nThe assembly line with the smoothest stations (minimum delta) :").
write0(2) :-
  write("\n\nThe assembly line with the lowest difference in work content :").

write1 :-                        % writes the elements in every station
  write("\n Station   Work Elements"),     % write heading
  fail.
write1 :-
  station(s(I,List,_,_,_,_,_)),            % get a station
  nl,write("    ",I,"        "),           % write station #
  writelist(List),                         % write work elements
  fail.                                    % backtracks to next station
write1.                                    % return after all stations
```

```
write2 :-                                   % writes rest of information
    % write heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  fail.
write2 :-
  station(s(I,_,Work,Difference,Delta,Var,Times)),    % get a station
    % write station information
  writef("\n %  %7.2  %7.2  %7.2  %7.2",I,Work,Difference,Delta,Var),
  writelist2(Times),                        % write station times
  fail.                                     % backtracks to next station
write2 :-
    % after all stations are written,
    % go to this clause and write totals
    % first find all stations with findall
  findall(Work,station(s(_,_,Work,_,_,_,_)),List1),
  findall(Delta,station(s(_,_,_,_,Delta,_,_)),List2),
  findall(Difference,station(s(_,_,_,Difference,_,_,_)),List3),
  findall(Var,station(s(_,_,_,_,_,Var,_)),List4),
  sumlist(List1,Sum1),sumlist(List2,Sum2),                % sum lists
  sumlist(List3,Sum3),sumlist(List4,Sum4),
  write("\n-------------------------------------------------------------------------------"),
    % write out the totals
  writef("\n    %7.2  %7.2  %7.2  %7.2  \n\n",Sum1,Sum3,Sum2,Sum4).

/*********************************************************************
check_min compares the value with the minimum value found and if it's
less it gets stored as the new minimum value. check_min succeeds if the
station currently considered is better than the previous minimum
station. The last parameter is 1 if new minimum found, 0 otherwise.
*********************************************************************/
check_min(1,Delta,Diff,1) :-        % go for lower delta
  min(Min_delta,_),                 % get current minimum value
  Delta < Min_delta,                % if new value is smaller
  !,                                % do not backtrack
  retractall(min(_,_)),             % retract old one
  asserta(min(Delta,Diff)).         % store new one
check_min(1,Delta,Diff,1) :-        % checks if they are equal
  min(Min_delta,Min_diff),          % get current minimum value
  abs(Delta - Min_delta) < 1.0E-8, % if they are equal
  Diff < Min_diff,                  % and the work difference is smaller
  !,                                % do not backtrack
  retractall(min(_,_)),             % retract old one
  asserta(min(Delta,Diff)).         % store new one
check_min(2,Delta,Diff,1) :-        % go for lower work difference
  min(_,Min_diff),                  % get current minimum value
  Diff < Min_diff,                  % if new value is smaller
  !,                                % do not backtrack
  retractall(min(_,_)),             % retract old one
  asserta(min(Delta,Diff)).         % store new one
check_min(2,Delta,Diff,1) :-        % checks if they are equal
  min(Min_delta,Min_diff),          % get current minimum value
  abs(Diff - Min_diff) < 1.0E-8,    % if they are equal
  Delta < Min_delta,                % and the work difference is smaller
  !,                                % do not backtrack
  retractall(min(_,_)),             % retract old one
  asserta(min(Delta,Diff)).         % store new one
check_min(_,_,_,0).                 % no new minimum found!

/*********************************************************************
write_station writes out the information for a given station I
*********************************************************************/
write_station(I,TT,Diff,Delta,List,Times) :-
  clearwindow,                              % clear window
  storage(S,H,T),                           % get memory status
    % write memory status
  write("\nStacksize = ",S,"  Heapsize = ",H,"  Trailsize = ",T),
  nl,
```

```
  write("\n Station   Work Elements"),          % write heading
  nl,write("   ",I,"       "),                 % write station #
  writelist(List),                             % write element list
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Station Times"),
    % write values for station
  writef("\n %  %7.2  %7.2  %7.2  %7.2",I,TT,Diff,Delta),
  writelist2(Times),                           % write station times
  min(X,Y),                                    % get minimums
  writef("\n\nMinimum :    %7.2  %7.2",Y,X).   % write minimums

/*************************************************************************
makelist makes a list of all previous nodes already assigned.
*************************************************************************/
makelist(Temp,X,N) :-
  station(s(N,List,_,_,_,_,_)),      % get station N
  !,                                 % do not backtrack to next clause
  append(Temp,List,List2),           % append list to temporary list
  N2 = N + 1,                        % go for next station
  makelist(List2,X,N2).              % call recursively
makelist(X,X,_).                     % if no more stations then end.

/*************************************************************************
serial controls the route for the serial solution where one station is
searched at a time and no backtracking is done to earlier stations N =
station to search, List = list of elements already assigned
*************************************************************************/
serial(N,_) :-                       % ending clause
  N > nn,                            % if all stations done
  !,                                 % do not backtrack
  write_output.                      % write station information
serial(N,List) :-
  retractall(min(_,_)),              % retract old minimum
  asserta(min(1E5,1E5)),             % insert big minimum for start
  route(List,[],[0],N).              % start routing sequence
serial(N,_) :-                       % comes here if station done
  makelist([],List,1),               % makes list of elements assigned
  N2 = N + 1,                        % increment station counter
  serial(N2,List).                   % go for next station

/*************************************************************************
The route clause is the search engine which generates all possible
combinations. The syntax is as follows:
                  route(List1,List2,List3,N)
where: List1 = list of elements (nodes) already assigned in PREVIOUS
               stations
       List2 = list of elements considered or assigned for this station
       List3 = maintenance list to avoid permutations of same list
           N = station number (integer)
*************************************************************************/
route(Final,_,_,nn) :-               % go here for last station
  !,                                 % do not backtrack for serial case
  findall(N,get0(N,Final),Final2),   % find all remaining stations
  get3(TT,Delta,Final2,Times),       % get total time and delta
  Diff = abs(middle-TT),             % calculate difference
  save_info(nn,Final2,TT,Diff,Delta,Times),  % store station info
  fail.                              % fail to exit route clause
route(_,[H|T],_,I) :-
    % this clause checks if TT is within limits, if yes, the delta
    % and TT values are compared to the minimum values and the
    % minimums are updated.
  g(TT,Delta,_,Times),               % get deltas and total time
  retractall(g(_,_,_,_)),            % remove value if done
  check_limits(TT),                  % checks limits
  Diff = abs(middle-TT),             % calculate difference
  check_min(choice,Delta,Diff,Result),       % compare with minimum
  write_station(I,TT,Diff,Delta,[H|T],Times), % write station
```

```
  Result > 0,                                % if minimum changed then
  save_info(I,[H|T],TT,Diff,Delta,Times),    % store station info
  fail.                                      % backtrack
route(Ass,L,[H|T],N) :-
    % this clause generates the combinations, it either chooses the
    % next element out of the list of possible elements or backtracks
    % to the member clause to find a new combination
  append(Ass,L,Ass2),                % append L to already assigned
                                     % lists
  findall(I,newnode(I,Ass2),Avail), % find list of all possible
                                     % elements to assign
  member(X,Avail),                   % get any member of the list
  X>H,                               % checks for permutation of
                                     % same station
  append(L,[X],L2),                  % append X to current station list
  get3(TT,Delta,L2,Times),           % get delta and total time
  asserta(g(TT,Delta,L2,Times)),     % insert into database
  TT <= middle*(1+upper),            % if station within limits
                                     % then continue else backtrack
                                     % to member clause!
  route(Ass,L2,[X,H|T],N).           % check if this list can make
                                     % a station this goes to second
                                     % route clause, note that station
                                     % counter is not incremented.

/*************************************************************************
The member clause is a very important part of the search algorithm.
There is no cut so all possible elements can be obtained by
backtracking.
*************************************************************************/
member(X,[X|_]).              % check first element in list (head)
member(X,[_|T]) :-
  member(X,T).                % check rest of the list (tail)

/*************************************************************************
append adds two lists to form a third.
syntax : append(List1,List2,List3). List3 := List1 + List2
*************************************************************************/
append([], List, List).              % trivial case
append([X|L1], List2, [X|L3]) :-
  append(L1, List2, L3).             % recursive call

/*************************************************************************
sumlist sums the elements of a list of reals
*************************************************************************/
sumlist([],0).                       % empty list
sumlist([H|T],S):-                   % non-empty list
  sumlist(T,S2),                     % sum tail of list
  S = S2 + H.

/*************************************************************************
writelist writes out a list separated with spaces
*************************************************************************/
writelist([]).                      % empty list
writelist([H|T]) :-                 % non-empty list
  write(H," "),                     % write head of list
  writelist(T).                     % write tail of list

/*************************************************************************
writelist2 writes out a list formatted in a specific way
*************************************************************************/
writelist2([]).              % empty list
writelist2([H|T]) :-         % non-empty list
  writef("%6.2",H),          % write head of list
  writelist2(T).             % write tail of list

/*************************************************************************
```

```
var calculates the variance of a list
*************************************************************************/
var([],Sx,Sxx,N,Var) :-       % empty list
  Var = Sxx/N - Sx*Sx/N/N.    % get variance
var([H|T],Sx,Sxx,N,Var) :-    % non-empty list
  Sx2 = Sx + H,               % sum of elements
  Sxx2 = Sxx + H*H,           % sum of squared elements
  N2 = N + 1,                 % increase counter
  var(T,Sx2,Sxx2,N2,Var).     % remainder (tail) of list

/*************************************************************************
timer calculates the elapsed CPU time. When called the first time, the
start time is stored in the database. When called the second time, the
difference between the start time and the current time is computed and
displayed.
*************************************************************************/
timer :-
    t(H1,M1,S1,D1),               % if it's the second time called then
    !,                            % do not backtrack to second clause
      % calculate start time in seconds
    Time1 = D1/100 + S1 + M1*60 + H1*3600.0,
    time(H2,M2,S2,D2),                          % get ending time
      % calculate end time in seconds
    Time2 = D2/100 + S2 + M2*60 + H2*3600.0,
    Diff = Time2 - Time1,                       % get difference
    Hours = trunc(Diff/3600),                   % calc. hours
    Diff2 = Diff - Hours*3600.0,
    Min = trunc(Diff2/60),                      % calc. minutes
    Diff3 = Diff2 - Min*60,
    Sec = Diff3,                                % calc. seconds
      % write elapsed time
    writef("\n\nElapsed CPU time :  %3 Hours %3 Minutes %5.2 Seconds\n",Hours,Min,Sec).
timer :-                                  % if it's first time
    time(H1,M1,S1,D1),                    % get current time
    asserta(t(H1,M1,S1,D1)).              % store in database
```

```
/*****************************************************************************
                                PROGRAM3.PRO
*****************************************************************************
                    Modified Exhaustive search Algorithm
                    -----------------------------------

This Prolog program finds all the feasible solutions for balancing a
mixed-model production line. The lines with the lowest difference in
work content, the smoothest stations and the lowest variance is saved
and displayed at the end of program execution. When more than one line
has the same value, then both lines will be stored as a minimum. This
version has the modified search algorithm which speeds up search.

This modified exhaustive search algorithm allows the user to do a step
by step solution. For example, the first 10 stations can be balanced,
the optimal solution saved and then the next 10 stations balanced. The
algorithm will still generate all possible combinations, in other words
there still is backtracking to previous stations. To improve the speed
of the search algorithm, the list of elements is sorted from large work
elements to small ones.

To execute the program, type go at the goal prompt. To redisplay the
results at the end of a program, or after a control-break, just type :
write_final after the goal prompt. To redirect output to a file or to
the printer, hit Alt-P. When the program starts, it will ask for a file
to load. This file contains the previous stations found. If no stations
were found, or if it is the first time the line is balanced, choose the
NONE.DBA file. To save the optimal solution after a partial search, use
the lsave(N) clause. The program will then ask for a filename, the
filename must have a DBA extension. The search is controlled by the
start and nn2 constants in the constants definition. Start designates
the station to start and nn2 the station to end.

The format of the input datafile is as follows:

[]          Prededence restrictions, immediate predecessors
[1,2,3]     of a station in square brackets separated by
            spaces. Station with no predecessors have empty
            brackets. Only one station per line.

0.3 4.0 2.3     The work element times in any format,
                separated by at least one space.

120 40 60       The model quantities in any format,
                separated by at least one space.

See SAMPLE1.DAT for an example of an input data file

written by P. R. Smith, November 1990. Tested with Turbo Prolog v2.0
*****************************************************************************/

constants
  input_fname = "SAMPLE2.DAT"       % input filename
  n_elements = 50           % number of work elements
  n_models   =  8           % number of different models
  nn = 26                   % number of stations
  nn2 = 15                  % end of current search
  middle = 500.0            % average value for station work content or
                            % shift time
  upper = 0.00              % upper tolerance for station work content
  lower = 0.40              % lower tolerance for station work content
  start = 1                 % station to start search procedure
/*****************************************************************************/

domains
  intlist = integer*
  reallist = real*
```

```
  s = s(integer,intlist,real,real,real,real,reallist)  % station object
  slist = s*                              % list of station objects
  tree = reference t(val, tree, tree)     % used for sorting lists
  val  = integer                          % used for sorting lists
/*************************************************************************/

database
  d(real,real,real)          % used for temporary storage
  station(s)                 % stores info about station in
                             % internal database
  tts(integer,real)          % average time for model i (Pj)
  node(integer,intlist)      % element j with list of predecessors from
                             % precedence diagram
  n(integer,real)            % work content for node j for all models
  q(integer,integer)         % production quantity for model j (Nj)
  te(integer,integer,real) % work time of element k on model j  (tjk)
  min(integer,real)          % minimum value for delta and TT
  ass_line(integer,slist)  % store a complete assembly line
  t(integer,integer,integer,integer) % used for timing elapsed
                                     % CPU time.
  g(real,real,intlist,reallist)      % used to store deltas
                                     % and work content
/*************************************************************************/

database - initial     % database domain for loading initial stations
  r(integer,intlist)   % station initial values
/*************************************************************************/

predicates
  route(intlist,intlist,intlist,integer)
  member(integer,intlist)
  append(intlist,intlist,intlist)
  subset(intlist,intlist)
  newnode(integer,intlist)
  writelist(intlist)
  writelist2(reallist)
  sumlist(reallist,real)
  var(reallist,real,real,integer,real)
  get(integer,real,intlist)
  get2(intlist)
  get3(real,real,intlist,reallist)
  get0(integer,intlist)
  check_limits(real)
  check_min(integer,real,real)
  save_info(integer,intlist,real,real,reallist)
  retract_all(integer)
  write_output
  write_final
  write_final1
  write_final2
  write_final3
  write_elems(slist)
  write_station(slist,real,real,real,real)
  write0
  write1
  write2
  write3
  get_tts
  input
  timer
  go
  insert(integer, tree)
  instree(intlist, tree)
  sort(intlist, intlist)
  treemembers(integer, tree)
  get_nodes(integer,integer,real)
  get_nodes2
```

```
  check_station(integer,intlist,intlist)
  lsave(integer)
  retrieve(intlist)
  store(slist)
  delete(integer,intlist,intlist)
  buildlist(integer,intlist,intlist)
/*************************************************************************/

clauses

/*************************************************************************
go is the main clause in this program. It clears the database, reads
the input file, loads the file with the initial values for stations and
starts the searching procedure. After the search procedure, the
solutions are displayed and the elapsed CPU time.
*************************************************************************/
go :-
  clearwindow,                          % clear the screen
  retractall(_),                        % clear internal database
  timer,                                % start timer
  write("\nReading Input File ..."),    % write msg
  input,                                % get input from file
  write("\nCalculating Averages..."),   % write msg
  get_tts,                              % calculate averages
  get_nodes2,                           % setup node times
  write("\nRetrieving initial station setup..."), % write msg
  retrieve(List),                      % get initial values if existing
  write("\nStarting Searching Sequence now..."),  % write msg
  route(List,[],[0],start).             % start routing sequence
go :-
  write_final,                          % write final output
  timer.                                % write elapsed CPU time

/*************************************************************************
check_limits is called with TT bound and checks if TT falls between the
limits specified by the user. The upper and lower are specified as
fractions not to exceed the middle value or shift time
*************************************************************************/
check_limits(TT) :-
  TT <= middle*(1+upper),     % check upper limit
  TT >= middle*(1-lower).     % check lower limit

/*************************************************************************
get_tts calculates the average or "ideal" time for model i for a
station. (Pi)
Pi := (Sum of all element times) x (# of units) / (# of stations)
*************************************************************************/
get_tts :-
  q(I,QQ),                    % get production quantity for model i
  findall(X,te(_,I,X),List),  % find all element times for model i ...
  sumlist(List,Sum),          % ... sum them all
  Temp = Sum*QQ/nn,           % calculate Pi
  assert(tts(I,Temp)),        % put value in internal database
  fail.                       % fail in order to backtrack
                              % through all models
get_tts.                      % returns after all models

/*************************************************************************
get_nodes2 calculates the work content for a given node (element) for
all models and inserts the value into the database. This value is then
used when sorting the list of work elements
*************************************************************************/
get_nodes2 :-
  node(I,_),                  % get first node
  get_nodes(I,1,0),           % calculate work content time
  fail.                       % backtrack to next node
get_nodes2.                   % return when all nodes done.
```

```
get_nodes(J,I,X) :-          % recursive clause to get all models
  q(I,QQ),                   % get production quantity
  !,                         % do not backtrack if model found
  te(J,I,Y),                 % get element time
  Sum = X + QQ*Y,            % get sum
  I2 = I + 1,                % get next model
  get_nodes(J,I2,Sum).
get_nodes(J,_,X) :-          % if no model found, then
  assertz(n(J,X)).           % insert value into database

/*************************************************************************
input uses the external program INPUT.EXE to write data in the correct
format to load into the internal database. INPUT.EXE Must be present in
default directory or accessible through PATH statement. For the format
of the datafile, see SAMPLE1.DAT
*************************************************************************/
input :-
  format(Outstring,"input.exe  %  temp    %     %",input_fname,
                                                  n_elements,n_models),
  system(Outstring),     % execute INPUT
  consult("temp").       % load facts in internal database

/*************************************************************************
get is called with N and List bound and returns the element time X for
model N for every element in the list.
*************************************************************************/
get(N,X,List) :-
  te(I,N,X),       % get element time X for model N element I
  member(I,List). % check if I is a member of the given list
                   % no backtracking here!

/*************************************************************************
get2 is called with List bound and computes the work content TT, the
"smoothness" indicator Delta and the station times for a MODEL only.
These 3 values are then stored in the internal database to be retrieved
by get3
*************************************************************************/
get2(List) :-
  q(I,QQ),                          % get first model I and quantity QQ
  findall(X,get(I,X,List),List2), % put all element times in List2
  sumlist(List2,Sum),               % sum all element times
  TT = QQ*Sum,                      % calculate total work
                                    % content for model
  tts(I,Temp),                      % get average for this model
  Delta =  abs(Temp-TT),            % calculate difference
  assertz(d(TT,Delta,Sum)),         % insert into database
  fail.                             % fail to next model
get2(_).                            % return when all models done

/*************************************************************************
get3 is called with List bound and returns Total_T and Total_Delta
which is the sum of TT and Delta respectively (from get2) for ALL
models. Times is a list which returns the station times for all models
*************************************************************************/
get3(Total_T,Total_Delta,List,Times) :-
  retractall(d(_,_,_)),              % clear database first
  get2(List),                        % calculate values for all models
  findall(TT,d(TT,_,_),List2),       % find all work
                                     % content times
  findall(Delta,d(_,Delta,_),List3), % find all deltas
  findall(Sum,d(_,_,Sum),Times),     % put station times in a list
  sumlist(List2,Total_T),            % sum the list
  sumlist(List3,Total_Delta).        %     "

/*************************************************************************
newnode is called with Current (list of nodes) bound and returns all
nodes NOT already in Current for which all precedence requirements have
```

```
been satisfied, that is all precedence nodes are in Current.
*************************************************************************/
newnode(I,Current) :-
  node(I,Predlist),          % get node I and it's list of predecessors
  not(member(I,Current)),    % is I already in Current?
  subset(Predlist,Current). % are all predecessors in Current?

/*************************************************************************
subset is called with both lists bound and succeeds if the first list
is a subset of the second list, that is if all elements in the first
list are included in the second list.
*************************************************************************/
subset([],_).              % trivial case
subset([H|T],L) :-
  member(H,L),             % is H a member of list L,
                           % this member does not backtrack!
  subset(T,L).             % if yes then test rest of list

/*************************************************************************
get0 is called with List bound and returns all nodes which are not in
List.
*************************************************************************/
get0(N,List) :-
  node(N,_),               % get any node N
  not(member(N,List)).     % is N a member of given List?

/*************************************************************************
save_info stores the information about a station in the internal
database to be retrieved later by the output routine.
syntax : save_info(Station #, List of Elements, Work Content,
                   Difference, Delta,List of model station times)
*************************************************************************/
save_info(N,Elm_list,Work_content,Delta,Model_list) :-
  retract_all(N),                      % remove all stations >= N
  Difference = abs(middle-Work_content),
  var(Model_list,0,0,0,Var),           % get variance of station times
  assertz(station(s(N,Elm_list,Work_content,Difference,Delta,Var,Model_list))).
                                       % insert new value into database

/*************************************************************************
retract_all(N) removes all stations >= N from database
allways succeeds
*************************************************************************/
retract_all(N) :-
  retract(station(s(N,_,_,_,_,_,_))),    % retract station N
  !,                                     % do not backtrack
  N2 = N + 1,                            % get next station
  retract_all(N2).                       % retract next station
retract_all(_).                          % return if no more stations

/*************************************************************************
write_output is called when the last station is reached and valid, i.e.
station limits are OK. write_output writes the intermediate results and
also updates the minimum values
*************************************************************************/
write_output :-
  write0,write1,write2.

write0 :-                  % writes memory status
  clearwindow,             % clear screen
  storage(S,H,T),          % get memory status
    % write status
  write("\nStacksize = ",S,"  Heapsize = ",H,"  Trailsize = ",T),
  nl.

write1 :-                      % writes the elements in every station
  write("\n Station   Work Elements"), % write heading
```

```
  fail.
write1 :-
  station(s(I,List,_,_,_,_,_)),          % get a station
  nl,write("   ",I,"       "),           % write station #
  writelist(List),                       % write work elements
  fail.                                  % backtracks to next station
write1.                                  % return after all stations

write2 :-                                % writes rest of information
    % write heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  fail.                                  % backtrack to second clause
write2 :-
  station(s(I,_,Work,Difference,Delta,Var,Times)),   % get a station
    % write station information
  writef("\n %  %7.2  %7.2  %7.2  %7.2",I,Work,Difference,Delta,Var),
  writelist2(Times),                     % write station times
  fail.                                  % backtracks to next station
write2 :-
    % after all stations are written, go to this clause and
    % write totals and update minimum values
    % first find all stations with findall
  findall(Work,station(s(_,_,Work,_,_,_,_)),List1),
  findall(Delta,station(s(_,_,_,_,Delta,_,_)),List2),
  findall(Difference,station(s(_,_,_,Difference,_,_,_)),List3),
  findall(Var,station(s(_,_,_,_,_,Var,_)),List4),
  sumlist(List1,Sum1),sumlist(List2,Sum2),          % sum lists
  sumlist(List3,Sum3),sumlist(List4,Sum4),
  write("\n-------------------------------------------------------------------------------"),
    % write out the totals
  writef("\n    %7.2  %7.2  %7.2  %7.2  \n\n",Sum1,Sum3,Sum2,Sum4),
    % now update minimum values
  check_min(1,Sum2,Min_delta),           % update delta
  check_min(2,Sum3,Min_diff),            % update difference
  check_min(3,Sum4,Min_var),             % update variance
    % write minimum values
  writef("Min values:  %7.2  %7.2  %7.2 \n\n",Min_diff,Min_delta,Min_var).

write3 :-                    % writes the elements in every station
  write("\n Station   TT     Work Elements"),
  fail.
write3 :-
  station(s(I,List,TT,_,_,_,_)),       % get a station
  nl,writef("  %3     %5.1    ",I,TT),
  writelist(List),
  fail.                      % backtracks to next station
write3.

/*************************************************************************
check_min compares the value with the minimum value found and if it's
less it gets stored as the new minimum value. If the minimum value is
the same, it gets stored as well
*************************************************************************/
check_min(I,Value,Min) :-
  min(I,Min),                % get value
  Value > Min,               % compare
  !.                         % if bigger exit, do not backtrack
check_min(I,Value,Value) :-
  min(I,Min),                % get current minimum
  abs(Min-Value) < 1.0E-10,  % equal value so store also
  !,                         % do not backtrack
  findall(X,station(X),List),% find all stations for this line
  asserta(ass_line(I,List)).% insert into database
check_min(I,Value,Value) :-  % new value is smaller than
                             % current minimum
  retractall(min(I,_)),      % retracts old value
  asserta(min(I,Value)),     % put new value in database
```

```
  retractall(ass_line(I,_)),    % retract old assembly line
  findall(X,station(X),List),   % find all stations for new line
  asserta(ass_line(I,List)).    % insert new minimum value in database

/*************************************************************************
write_final writes the station with the minimum values found. It is
called after all combinations have been tried
*************************************************************************/
write_final :- write_final1,
               write_final2,
               write_final3.

write_final1 :-                 % write lines with smoothest stations
  ass_line(1,X),                % get assm. line with smoothest stations
    % write headings
  write("\n\nThe assembly with the smoothest stations (minimum delta) : "),
  write("\n\n Station   Work Elements"),
  write_elems(X),               % write work elements
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  write_station(X,0,0,0,0),     % write station information
  fail.                         % fail for next line
write_final1.                   % return after all lines done

write_final2 :-    % write lines with lowest difference in
                   % work content for all stations
  ass_line(2,X),   % get assembly line
    % write headings
  write("\n\nThe assembly with the lowest difference in work content : "),
  write("\n\n Station   Work Elements"),
  write_elems(X),           % write work elements
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  write_station(X,0,0,0,0), % write station information
  fail.                     % fail for next line
write_final2.               % returns after all lines done

write_final3 :-   % write lines with lowest variance
  ass_line(3,X),  % get assembly line
    % write headings
  write("\n\nThe assembly with the lowest variance in station times : "),
  write("\n\n Station   Work Elements"),
  write_elems(X),           % write work elements
    % write another heading
  write("\n\nSt  Work Cnt    Diff    Delta     Var   Station Times"),
  write_station(X,0,0,0,0),  % write station information
  fail.                     % fail for next line
write_final3.               % returns after all lines done

/*************************************************************************
write_elems writes out work elements of a station
*************************************************************************/
write_elems([]).                                % empty list, so done.
write_elems([s(I,Elements,_,_,_,_,_)|T]) :-     % non empty list
  write("\n   ",I,"       "),                   % write station #
  writelist(Elements),                          % write work elements
  write_elems(T).                               % write rest of list

/*************************************************************************
write_station writes out the information for a line. It is called
recursively until all stations are done. At this point all totals are
written.
*************************************************************************/
write_station([],T_work,T_diff,T_delta,T_var) :-
    % write rest of station info - all stations are done
  write("\n-------------------------------------------------------------------------------"),
    % write station parameters
```

```
  writef("\n    %7.2  %7.2  %7.2  %7.2   \n\n",T_Work,T_diff,T_delta,T_var).
  write_station([s(I,_,Work,Difference,Delta,Var,Times)|T],Sum1,Sum2,Sum3,Sum4) :-
    % writes first element in the list
  writef("\n %  %7.2  %7.2  %7.2  %7.2",I,Work,Difference,Delta,Var),
  writelist2(Times),  % write station times
  SSum1 = Sum1 + Work,                  % sum the work content
  SSum2 = Sum2 + Difference,            % sum the differences
  SSum3 = Sum3 + Delta,                 % sum the delta values
  SSum4 = Sum4 + Var,                   % sum the station variances
    % write remainder of the stations in the line
  write_station(T,SSum1,SSum2,SSum3,SSum4).

/*************************************************************************
The route clause is the search engine which generates all possible
combinations. The syntax is as follows:
                  route(List1,List2,List3,N)
where: List1 = list of elements (nodes) already assigned in PREVIOUS
               stations
       List2 = list of elements considered or assigned for this station
       List3 = maintenance list to avoid permutations of same list
           N = station number (integer)
*************************************************************************/
route(Final,_,_,nn) :-                  % go here for last station
  nn = nn2,
  findall(N,get0(N,Final),Final2),      % find all remaining stations
  get3(TT,Delta,Final2,Times),          % get total time and delta
  check_limits(TT),                     % checks limits
  save_info(nn,Final2,TT,Delta,Times), % store station info
  write_output,                         % write station and compare min's
  fail.                                 % backtrack
route(_,_,_,N) :-                      % go here for intermediate station
  N > nn2,                              % if intermediate station
  write_output,                         % write station and compare min's
  fail.                                 % backtrack
route(Ass,[H|T],_,I) :-
    % this clause checks if TT is within limits, if yes, the station
    % is accepted and the next station is started. If no, the clause
    % fails to the next one
  I < nn,
  I <= nn2,                            % checks if not final station
  not(r(I,_)),                         % are all constraints satisfied
  g(TT,Delta,[H|T],Times),             % get deltas and total time
  check_limits(TT),                       % checks limits
  save_info(I,[H|T],TT,Delta,Times),   % store station info
  write0,write3,           % write the station and it's work content
  append(Ass,[H|T],Ass2),  % append this list to already
                           % assigned list (List1)
  I2 = I + 1,              % increment station counter
  route(Ass2,[],[0],I2).   % start search for next station
route(Ass,L,[H|T],N) :-
   % this clause generates the combinations it either chooses the next
   % element out of the list of possible elements or backtracks to the
   % member clause to find a new combination
  N < nn,
  N <= nn2,                             % checks if not final station
  append(Ass,L,Ass2),                   % append L to already assigned list
  findall(I,newnode(I,Ass2),Avail),     % find list of all possible
                                        % elements to assign
  sort(Avail,Avail2),                   % sort list according to work
                                        % contents
  check_station(N,Avail2,Avail3),       % check station against initial
                                        % values
  member(X,Avail3),                     % get any member of the list
  X>H,                                  % checks for permutation of
                                        % same station
  append(L,[X],L2),                     % append X to current station list
  get3(TT,Delta,L2,Times),              % get total work and delta
```

```
  retractall(g(_,_,_,_)),          % clear database first
  asserta(g(TT,Delta,L2,Times)),   % store in database
  TT <= middle*(1 + upper),        % if element too big,
                                   % backtrack to member
  route(Ass,L2,[X,H|T],N).         % check if this list can
                                   % make a station
                                   % this goes to second route clause,
                                   % note that station counter is not
                                   % incremented.

/*************************************************************************
The member clause is a very important part of the search algorithm.
There is no cut so all possible elements can be obtained by
backtracking.
*************************************************************************/
member(X,[X|_]).                   % check first element in list (head)
member(X,[_|T]) :-
  member(X,T).                     % check rest of the list (tail)

/*************************************************************************
append adds two lists to form a third.
syntax : append(List1,List2,List3). List3 := List1 + List2
*************************************************************************/
append([], List, List).            % trivial case
append([X|L1], List2, [X|L3]) :-
  append(L1, List2, L3).           % recursive call

/*************************************************************************
var calculates the variance of a list
*************************************************************************/
var([],Sx,Sxx,N,Var) :-            % empty list
  Var = Sxx/N - Sx*Sx/N/N.         % get variance
var([H|T],Sx,Sxx,N,Var) :-         % non-empty list
  Sx2 = Sx + H,                    % sum of elements
  Sxx2 = Sxx + H*H,                % sum of squared elements
  N2 = N + 1,                      % increase counter
  var(T,Sx2,Sxx2,N2,Var).          % remainder (tail) of list


/*************************************************************************
The following clauses are all used to sort a list of integers. The
clauses are taken directly from the Turbo Prolog User's Manual. The
sorting method used is tree based sorting and is therefore very fast.
*************************************************************************/
insert(Val, t(Val, _, _)) :- !.
insert(Val, t(Val1, Tree, _)) :-
  n(Val,X),
  n(Val1,Y),
  X>Y, !,
  insert(Val, Tree).
insert(Val, t(_, _, Tree)) :-
  insert(Val, Tree).
instree([], _).
instree([H|T], Tree) :-
  insert(H, Tree),
  instree(T, Tree).

treemembers(_, T) :-
  free(T), !,
  fail.
treemembers(X, t(_, L, _)) :-
  treemembers(X, L).
treemembers(X, t(Refstr, _, _)) :-
  X = Refstr.
treemembers(X, t(_, _, R)) :-
  treemembers(X, R).
```

```
sort(L, L1) :-
  instree(L, Tree),
  findall(X, treemembers(X, Tree), L1).

/*************************************************************************
timer calculates the elapsed CPU time. When called the first time, the
start time is stored in the database. When called the second time, the
difference between the start time and the current time is computed and
displayed.
*************************************************************************/
timer :-
    t(H1,M1,S1,D1),              % if it's the second time called then
    !,                           % do not backtrack to second clause
      % calculate start time in seconds
    Time1 = D1/100 + S1 + M1*60 + H1*3600.0,
    time(H2,M2,S2,D2),                          % get ending time
      % calculate end time in seconds
    Time2 = D2/100 + S2 + M2*60 + H2*3600.0,
    Diff = Time2 - Time1,                       % get difference
    Hours = trunc(Diff/3600),                   % calc. hours
    Diff2 = Diff - Hours*3600.0,
    Min = trunc(Diff2/60),                      % calc. minutes
    Diff3 = Diff2 - Min*60,
    Sec = Diff3,                                % calc. seconds
      % write elapsed time
    writef("\n\nElapsed CPU time :  %3 Hours %3 Minutes %5.2 Seconds\n",Hours,Min,Sec).
timer :-                                 % if it's first time
    time(H1,M1,S1,D1),                   % get current time
    asserta(t(H1,M1,S1,D1)).             % store in database

/*************************************************************************
check_station checks the station I against the value in the initial
database If the station is not in the database, the check_station
succeeds. If the station is in the database and the list of elements
differs, then check_station fails and nothing else is done. If the
lists of elements match then check_station succeeds and the clause is
removed from the internal database. This means that the search
procedure has reached the point at which the setup was saved, and can
now go on and look for more solutions.
*************************************************************************/

check_station(I,X,X) :-                  % no restrictions, just return
  not(r(I,_)),                           % same list
  !.                                     % do not backtrack
check_station(I,List,[H|List2]) :-
  r(I,[H]),!,                            % just one element in initial
                                         % value list
  delete(H,List,List2),                  % delete this value from list
  retractall(r(I,_)).                    % restriction satisfied
                                         % therefore remove restriction
                                         % H is deleted from the middle
                                         % of the list and added to the
                                         % front of the list.
check_station(I,List,[H|List2]) :-
  r(I,[H|T]),                            % more than one element in
                                         % initial value list
  delete(H,List,List2),                  % delete this value from list
  retractall(r(I,_)),                    % remove original restriction
  asserta(r(I,T)).                       % add new one to database
                                         % H is deleted from the middle
                                         % of the list and added to the
                                         % front of the list.

/*************************************************************************
lsave is a command that lets the user save the current setup to a file
that can be retrieved later. The file must have a .dba extension
N is the assembly line to save.
```

```
  N=1 : minimum delta line saved
  N=2 : minimum difference line saved
  N=3 : minimum variance line saved
*************************************************************************/
lsave(N) :-
  ass_line(N,Line),                  % get assembly line N
  store(Line),                       % store station # and element list
  write("\n\nEnter a filename :"),   % get filename
  readln(Filename),
  save(Filename,initial).            % save database to file

/*************************************************************************
store takes a list of stations and breaks it up to insert it into the
initial database domain.
*************************************************************************/
store([]).                         % empty list
store([s(I,List,_,_,_,_,_)|T]) :-
  assertz(r(I,List),initial),      % store station # and element list
  store(T).                        % store rest of line

/*************************************************************************
retrieve loads a set of stations previously saved into the database and
also builds the list of initial stations for the route clause.
If start = 1 then the stations are only restrictions otherwise if
start > 1 the stations already assigned must be taken into account in
order not to be repeated.
*************************************************************************/
retrieve(List) :-
  write("\n\nName of database to retrieve : "),        % get filename
  makewindow(1,7,0,"Database Filename",10,10,10,60),
  dir("","*.dba",Filename,1,1,1),       % get filename
  removewindow,
  consult(Filename,initial),                              % load facts
  write("\nStation with initial values successfully loaded"),
  start > 1,                % if start > 1
  !,                        % do not backtrack
  buildlist(1,[],List).     % get list of already assigned elements
retrieve([]).               % if start = 1 then return empty list

/*************************************************************************
buildlist builds a list for the route clause of elements already
assigned in previous stations. buildlist is called recursively until
the counter = start.
*************************************************************************/
buildlist(start,Answer,Answer) :- !.   % return if counter = start
buildlist(I,List,Answer) :-
  r(I,List1),                      % get station
  !,                               % do not backtrack
  append(List1,List,List3),        % append list
  I2 = I + 1,                      % increment counter
  buildlist(I2,List3,Answer).      % call recursively

/*************************************************************************
sumlist sums the elements of a list of reals
*************************************************************************/
sumlist([],0).                      % empty list
sumlist([H|T],S):-                  % non-empty list
  sumlist(T,S2),                    % sum tail of list
  S = S2 + H.

/*************************************************************************
writelist writes out a list separated with spaces
*************************************************************************/
writelist([]).                     % empty list
writelist([H|T]) :-                % non-empty list
  write(H," "),                    % write head of list
  writelist(T).                    % write tail of list
```

```
/*************************************************************************
writelist2 writes out a list formatted in a specific way
*************************************************************************/
writelist2([]).                 % empty list
writelist2([H|T]) :-            % non-empty list
  writef("%6.2",H),             % write head of list
  writelist2(T).                % write tail of list

/*************************************************************************
delete deletes an element from a list. If the element is not in the
list, the clause fails.
*************************************************************************/
delete(X,[X|T],T) :- !.                 % deletes an element from a list
delete(X,[H|T],[H|Newlist]) :-
    delete(X,T,Newlist).
```

```
{*************************************************************************
                              PROGRAM4.PAS
*************************************************************************
            Macaskill's penalty method : Serial version
            -------------------------------------------

This program finds the optimal sequence for a mixed model line. The
program uses Macaskill's penalty method, which assigns a penalty cost
to each inefficiency. Concurrent work is disallowed.

The format of the datafile is as follows:

1. The model quantities Q1, Q2, ...
2. For each model, the station times for all stations
3. The four penalty cost parameters
4. The station length, upper boundary and lower boundary.

See SAMPLE3.dat for an example of an input file.
No specific formatting of the data is required.
The program calculates the launching rate c.

written by P. R. Smith, November 1990. Tested with Turbo Pascal v5.0
*************************************************************************}
{$N+}

const
  n_station = 19;                 {number of stations}
  n_models = 6;                   {number of models}
  input_fname = 'SAMPLE3.DAT';    {input filename}
{*************************************************************************}

var
  d                              {operation time for unit i in station j}
                                 : array[1..n_models,1..n_station] of double;
  I_cost,C_cost,D_cost,U_cost    {Inefficiency Cost parameters}
                                 : double;
  t,                             {station length}
  tu,                            {upstream allowance}
  td,                            {downstream allowance}
  a,                             {arrival time of unit in station}
  x,                             {exit time of unit in station}
  s                              {start of work on unit}
                                 : array[1..n_station] of double;
  e0,                            {ending time of PREVIOUS model}
  e1,                            {ending time of CURRENT model}
  temp                           {temporary array for holding}
                                 {ending times}
                                 : array[0..n_station] of double;

  ttwc                           {work content for each model}
                                 : array[1..n_models] of double;
  Q                              {prod quantities}
                                 : array[1..n_models] of integer;
  n_units                        {total # of units to produce}
                                 : integer;
  c,                             {launching interval}
  tot_wc                         {total work content for all models}
                                 : double;

{*************************************************************************}
procedure get_input;                          {reads input from datafile}
  var
    i,j : integer;
    input : text;
  begin
    assign(input,input_fname);
    reset(input);
```

```
    for i := 1 to n_models do                {read model quantities}
      read(input,Q[i]);
    for i := 1 to n_models do
      for j := 1 to n_station do
        read(input,d[i,j]);                  {read station times}
    read(input,I_cost,D_cost,C_cost,U_cost); {read penalty costs}
    for i := 1 to n_station do
      read(input,t[i],tu[i],td[i]);          {read station parameters}
    close(input);
  end;

{************************************************************************}
procedure write_input;      {prints out input parameters for verifying}
  var
    i,j : integer;
    sum : double;
  begin
    writeln;
    for i := 1 to n_models do
      begin
        sum := 0.0;
        for j := 1 to n_station do          {calculate work content for}
                                            {each model}
          sum := sum + d[i,j];
        writeln('Total time for model ',i:3,' = ',sum:10:4);
      end;
    writeln;                                {write penalty costs}
    writeln('  I_cost : ',I_cost:6:2,'  D_cost : ',D_cost:6:2,
            '  C_cost : ',C_cost:6:2,'  U_cost : ',U_cost:6:2);
    writeln;                                {write station sizes}
    writeln('     t       tu      td');
    for i := 1 to n_station do
      writeln(t[i]:8:2,tu[i]:8:2,td[i]:8:2);
    writeln;
  end;

{************************************************************************}
procedure init;        {set up initial values for sequencing method}
  var
    j,k : integer;
  begin
    a[1] := 0;                      {arrival time of first unit}
    e0[0] := 0;                     {ending time of previous unit}
    for j := 1 to n_station do
      begin
        x[j] := a[j] + t[j];        {calculate departure times}
        if j < n_station then
          a[j+1] := x[j];           {arrival time of next unit}
        e0[j] := a[j];              {ending times of previous unit}
      end;
    e1 := e0;                       {ending times of current model}
    n_units := 0;
    tot_wc := 0;
    for k := 1 to n_models do       {calculate total work content}
      begin
        n_units := n_units + Q[k];  {calculate total number of units}
        ttwc[k] := 0.0;             {work content at station k}
        for j := 1 to n_station do
          ttwc[k] := ttwc[k] + d[k,j];
        tot_wc := tot_wc + ttwc[k]*Q[k];
      end;
    c :=  tot_wc/n_units/n_station;   {calculate launching interval}
  end;

{************************************************************************}
function max(a,b : double) : double;
  begin                 {returns the maximum value of a and b}
```

```
    if a < b then
      max := b
    else
      max := a
  end;

{************************************************************************}
{ This procedure computes the total penalty cost for a given model}
procedure get_cost (model:integer; var II,DD,UU,CC,Total : double);
  var
    j : integer;
    I,F,U,C : double;  {Idle, deficiency, Utility Congestion}
  begin
    II := 0; DD := 0; UU := 0; CC := 0; Total := 0;
    {write heading}
    {
    writeln('  j    t    tu    td     a      x      s      e0     e1   I    F    U    C');
    }
    for j := 1 to n_station do  {for every station}
      begin
        I := 0; F := 0; U := 0; C := 0;
        { inefficiency TIMES for this station only!}
        { *** Upstream equations *** }
        if (a[j] - e0[j]) > tu[j]  then  {Case 1}
          if (a[j] - tu[j]) > e1[j-1] then
            begin
              I := a[j] - tu[j] - e0[j];
              F := tu[j];
              s[j] := a[j] - tu[j];
            end
          else
            begin
              I := e1[j-1] - e0[j];
              F := max(a[j]-e1[j-1],0);
              s[j] := e1[j-1];
            end
        else if (a[j] - e0[j]) > 0 then  {Case 2}
          if e1[j-1] < e0[j] then
            begin
              F := a[j] - e0[j];
              s[j] := e0[j];
            end
          else
            begin
              I := e1[j-1] - e0[j];
              F := max(a[j] - e1[j-1],0);
              s[j] := e1[j-1];
            end
        else                                {Case 3}
          begin
            s[j] := max(e1[j-1],e0[j]);
            I := max(e1[j-1]-e0[j],0);
          end;

        { *** Downstream equations *** }
          if (s[j] + d[model,j]) > (x[j] + td[j]) then
            begin
              U := s[j] + d[model,j] - (x[j] + td[j]);
              C := td[j];
              e1[j] := x[j] + td[j];
            end
          else
            begin
              C := max(s[j]+d[model,j]-x[j],0);
              e1[j] := s[j] + d[model,j];
            end;
          {display results}
```

```
        {
        writeln(j:3,t[j]:5:1,tu[j]:5:1,td[j]:5:1,a[j]:7:2,x[j]:7:2,
                s[j]:7:2,e0[j]:7:2,e1[j]:7:2,I:5:2,F:5:2,U:5:2,C:5:2);
        }
        {Update Total Inefficiency Times}
        II := II + I;
        DD := DD + F;
        UU := UU + U;
        CC := CC + C;
      end; { end of station loop }
    {compute total penalty cost}
    Total := II*I_cost + DD*D_cost + UU*U_cost + CC*C_cost;
  end;

{**************************************************************************
This procedure computes the total cost for all models of a given
sequence. It also does the sequencing itself, the model with the lowest
penalty cost is chosen. In case of a tie,  the model with the biggest
work content is selected.
**************************************************************************}
procedure get_total_cost;
  var
    {vars to hold times and cost for model}
    Idle,Def,Util,Con,Total_cost,Min,Max,
    {vars to hold total times and cost}
    TTotal,I_Total,D_Total,U_Total,C_Total
                                                    : double;
    {arrays to hold temporary values}
    I_temp,D_temp,U_temp,C_temp : array[1..n_models] of double;

    QQ : {counts number of models produced}
         array[1..n_models] of integer;
    i,j,k,next : integer;
  begin
    init;                                          {initialize a,x,e0,e1}
    TTotal := 0;
    I_Total := 0; D_Total := 0;
    U_Total := 0; C_Total := 0;                    {init total times}
    for i := 1 to n_models do
      QQ[i] := Q[i];
    for i := 1 to n_units do
      begin
        Min := 1e5;                      {initialize min and max}
        Max := 0.0;

        writeln('  Model    Idle      Con     Work Def    Utility     Total  Work Content');

        for k := 1 to n_models do
          if QQ[k] > 0 then              {are all units of model k produced?}
            begin
              get_cost(k,Idle,Def,Util,Con,Total_cost);
              {store in temp arrays}
              I_temp[k] := Idle;    D_temp[k] := Def;
              U_temp[k] := Util;    C_temp[k] := Con;

              writeln(k:5,Idle:10:4,Def:10:4,Util:10:4,Con:10:4,
                      Total_cost:10:4,ttwc[k]:10:4);

              {check if new minimum found}
              if (Total_cost<Min) or ((Total_cost=Min) and (ttwc[k]>Max)) then
                begin
                  Min := Total_cost;
                  Max := ttwc[k];
                  next := k;       {save model number}
                  temp := e1;      {save ending times for this model}
                end;
            end;
```

```
        QQ[next] := QQ[next] - 1;  {decrement counter for this model}
        I_Total := I_Total + I_temp[next];
        D_Total := D_Total + D_temp[next];
        U_Total := U_Total + U_temp[next];
        C_Total := C_Total + C_temp[next];
        {display results}

        writeln('Minimum : ',next:3,'     Total Penalty Cost :',Min:10:4);

        writeln;

        TTotal := TTotal + Min;
        e0 := temp;                    {set up times for next model}
        for j := 1 to n_station do
          begin
            a[j] := a[j] + c;
            x[j] := x[j] + c;
          end;
      end;                              {next unit}
    {all units done, display total output}
    writeln;
    writeln('          Summary Statistics');
    writeln('_____________________________________');
    writeln;
    writeln('Number of units    : ',n_units:10);
    writeln('Total Work Content : ',tot_wc:10:4);
    writeln('Launching Interval : ',c:10:4);
    writeln;
    writeln('Total Inefficiencies for this sequence : ');
    writeln;
    writeln('Idle Time             :  ',I_total:10:4);
    writeln('Work Deficiency Time  :  ',D_total:10:4);
    writeln('Utility Work          :  ',U_total:10:4);
    writeln('Work Congestion Time  :  ',C_total:10:4);
    writeln;
    writeln('Total Penalty Cost    :  ',TTotal :10:4);
  end; {get_total_cost}

{************************************************************************}
{main program}
begin
  get_input;
  write_input;
  get_total_cost;
end.
```

```
{*************************************************************************
                              PROGRAM5.PAS
*************************************************************************
            Macaskill's penalty method : Serial version

This program finds the total penalty cost for a given sequence. The
program uses Macaskill's penalty method, which assigns a penalty cost
to each inefficiency. Concurrent work is disallowed.

The format of the datafile is as follows:

1. The model quantities Q1, Q2, ...
2. For each model, the station times for all stations
3. The four penalty cost parameters
4. The station length, upper boundary and lower boundary.

See SAMPLE3.dat for an example of an input file.
No specific formatting of the data is required.
The program calculates the launching rate c. The sequence for the units
is not given in the data file, but in the constants section of the
program.

written by P. R. Smith, November 1990. Tested with Turbo Pascal v5.0
*************************************************************************}
{$N+}

const
  n_station = 19;                 {number of stations}
  n_models = 6;                   {number of models}
  input_fname = 'SAMPLE3.DAT';    {input filename}
  n_seq = 26;                     {number of units in the sequence}
  sequence                        {array containing model numbers}
                                  : array[1..n_seq] of integer
  = (6,3,2,1,1,2,3,1,2,1,6,5,1,4,6,4,4,4,6,4,4,6,5,6,5,6);
                                  {sequence to compute}

{*************************************************************************}
var
  d                             {operation time for unit i in station j}
                                : array[1..n_models,1..n_station] of double;
  I_cost,C_cost,D_cost,U_cost   {Inefficiency Cost parameters}
                                : double;
  t,                            {station length}
  tu,                           {upstream allowance}
  td,                           {downstream allowance}
  a,                            {arrival time of unit in station}
  x,                            {exit time of unit in station}
  s                             {start of work on unit}
                                : array[1..n_station] of double;
  e0,                           {ending time of PREVIOUS model}
  e1,                           {ending time of CURRENT model}
  temp                          {temporary array for holding ending times}
                                : array[0..n_station] of double;
  ttwc                          {work content for each model}
                                : array[1..n_models] of double;
  Q                             {prod quantities}
                                : array[1..n_models] of integer;
  n_units                       {total # of units to produce}
                                : integer;
  c,                            {launching interval}
  tot_wc                        {total work content for all models}
                                : double;
{*************************************************************************}
procedure get_input;                        {reads input from datafile}
  var
    i,j : integer;
```

```
    input : text;
  begin
    assign(input,input_fname);
    reset(input);
    for i := 1 to n_models do                    {read model quantities}
      read(input,Q[i]);
    for i := 1 to n_models do
      for j := 1 to n_station do
        read(input,d[i,j]);                      {read station times}
    read(input,I_cost,D_cost,C_cost,U_cost);     {read penalty costs}
    for i := 1 to n_station do
      read(input,t[i],tu[i],td[i]);              {read station parameters}
    close(input);
  end;

{************************************************************************}
procedure write_input;      {prints out input parameters for verifying}
  var
    i,j : integer;
    sum : double;
  begin
    writeln;
    for i := 1 to n_models do
      begin
        sum := 0.0;
        for j := 1 to n_station do          {calculate work content for}
                                            {each model}
          sum := sum + d[i,j];
        writeln('Total time for model ',i:3,' = ',sum:10:4);
      end;
    writeln;                                {write penalty costs}
    writeln('  I_cost : ',I_cost:6:2,'  D_cost : ',D_cost:6:2,
            '  C_cost : ',C_cost:6:2,'  U_cost : ',U_cost:6:2);
    writeln;                                {write station sizes}
    writeln('     t       tu      td');
    for i := 1 to n_station do
      writeln(t[i]:8:2,tu[i]:8:2,td[i]:8:2);
    writeln;
  end;

{************************************************************************}
procedure init;         {set up initial values for sequencing method}
  var
    j,k : integer;
  begin
    a[1] := 0;                      {arrival time of first unit}
    e0[0] := 0;                     {ending time of previous unit}
    for j := 1 to n_station do
      begin
        x[j] := a[j] + t[j];        {calculate departure times}
        if j < n_station then
          a[j+1] := x[j];           {arrival time of next unit}
        e0[j] := a[j];              {ending times of previous unit}
      end;
    e1 := e0;                       {ending times of current model}
    n_units := 0;
    tot_wc := 0;
    for k := 1 to n_models do       {calculate total work content}
      begin
        n_units := n_units + Q[k];  {calculate total number of units}
        ttwc[k] := 0.0;             {work content at station k}
        for j := 1 to n_station do
          ttwc[k] := ttwc[k] + d[k,j];
        tot_wc := tot_wc + ttwc[k]*Q[k];
      end;
    c :=  tot_wc/n_units/n_station;   {calculate launching interval}
  end;
```

```
{************************************************************************}
function max(a,b : double) : double;
  begin                 {returns the maximum value of a and b}
    if a < b then
      max := b
    else
      max := a
  end;

{************************************************************************}
{ This procedure computes the total penalty cost for a given model}
procedure get_cost (model:integer; var II,DD,UU,CC,Total : double);
  var
    j : integer;
    I,F,U,C : double;  {Idle, deFiciency, Utility Congestion}
  begin
    II := 0; DD := 0; UU := 0; CC := 0; Total := 0;
    {write heading}
    {
    writeln('  j    t    tu    td     a      x      s      e0     e1   I    F    U    C');
    }
    for j := 1 to n_station do  {for every station}
      begin
        I := 0; F := 0; U := 0; C := 0;
        { inefficiency TIMES for this station only!}
        { *** Upstream equations *** }
        if (a[j] - e0[j]) > tu[j]  then  {Case 1}
          if (a[j] - tu[j]) > e1[j-1] then
            begin
              I := a[j] - tu[j] - e0[j];
              F := tu[j];
              s[j] := a[j] - tu[j];
            end
          else
            begin
              I := e1[j-1] - e0[j];
              F := max(a[j]-e1[j-1],0);
              s[j] := e1[j-1];
            end
        else if (a[j] - e0[j]) > 0 then  {Case 2}
          if e1[j-1] < e0[j] then
            begin
              F := a[j] - e0[j];
              s[j] := e0[j];
            end
          else
            begin
              I := e1[j-1] - e0[j];
              F := max(a[j] - e1[j-1],0);
              s[j] := e1[j-1];
            end
        else                                {Case 3}
          begin
            s[j] := max(e1[j-1],e0[j]);
            I := max(e1[j-1]-e0[j],0);
          end;

      { *** Downstream equations *** }
        if (s[j] + d[model,j]) > (x[j] + td[j]) then
          begin
            U := s[j] + d[model,j] - (x[j] + td[j]);
            C := td[j];
            e1[j] := x[j] + td[j];
          end
        else
          begin
            C := max(s[j]+d[model,j]-x[j],0);
```

```
          e1[j] := s[j] + d[model,j];
        end;
      {display results}
      {
      writeln(j:3,t[j]:5:1,tu[j]:5:1,td[j]:5:1,a[j]:7:2,x[j]:7:2,
              s[j]:7:2,e0[j]:7:2,e1[j]:7:2,I:5:2,F:5:2,U:5:2,C:5:2);
      }
      {Update Total Inefficiency Times}
      II := II + I;
      DD := DD + F;
      UU := UU + U;
      CC := CC + C;
    end; { end of station loop }
  {compute total penalty cost}
  Total := II*I_cost + DD*D_cost + UU*U_cost + CC*C_cost;
end;

{*************************************************************************}
procedure get_total_cost; {get total cost for given sequence}
  var
    {vars to hold times and cost for model}
    Idle,Def,Util,Con,Total_cost,
    {vars to hold total times and cost}
    TTotal,I_total,D_total,C_total,U_total
                                          : double;
    i,j,k,next : integer;
  begin
    init;
    TTotal := 0;
    I_total := 0; D_total := 0; C_total := 0; U_total := 0;
    writeln;    {write a heading}
    writeln('Model   Idle       Def      Util      Con      Total');
    for i := 1 to n_seq do       {for all units}
      begin
        {get penalty cost for selected unit in sequence}
        get_cost(sequence[i],Idle,Def,Util,Con,Total_cost);
        TTotal := TTotal + Total_cost;
        I_total := I_total + Idle;
        D_total := D_total + Def;
        U_total := U_total + Util;
        C_total := C_total + Con;
        {display results}
        writeln(sequence[i]:3,Idle:10:4,Def:10:4,Util:10:4,Con:10:4,Total_cost:10:4);

        e0 := e1;                  {set up times for next model}
        for j := 1 to n_station do
          begin
            a[j] := a[j] + c;
            x[j] := x[j] + c;
          end;
      end;                              {next unit}
    {all units done, write summary output}
    writeln;
    writeln('          Summary Statistics');
    writeln('_____________________________________');
    writeln;
    writeln('Number of units    : ',n_units:10);
    writeln('Total Work Content : ',tot_wc:10:4);
    writeln('Launching Interval : ',c:10:4);
    writeln;
    writeln('Total Inefficiencies for this sequence : ');
    writeln;
    writeln('Idle Time             :  ',I_total:10:4);
    writeln('Work Deficiency Time  :  ',D_total:10:4);
    writeln('Utility Work          :  ',U_total:10:4);
    writeln('Work Congestion Time  :  ',C_total:10:4);
    writeln;
```

```
    writeln('Total Penalty Cost    :  ',TTotal :10:4);
  end; {get_total_cost}

{************************************************************************}
{main program}
begin
  get_input;
  write_input;
  get_total_cost;
end.
```

```
{*************************************************************************
                              PROGRAM6.PAS
*************************************************************************
          Dar-El & Cother method for line length minimization

This program finds the optimal sequence and station lengths for a mixed
model line. The method used is Dar-El & Cother line length minimization
algorithm. This program is for CLOSED workstations.

The format of the datafile is as follows:

1. The model quantities Q1, Q2, ...
2. For each model, the station times for all stations

See SAMPLE4.dat for an example of an input file. No specific formatting
of the data is required. The program calculates the launching rate c.

written by P. R. Smith, November 1990. Tested with Turbo Pascal v5.0
*************************************************************************}
{$N+}

const
  n_station = 19;                       {number of stations}
  n_models = 6;                         {number of models}
  input_fname = 'SAMPLE4.DAT';          {input filename}
  output_fname = 'TEMP.OUT';            {ouput filename}
  large = 5.0;                          {large increment}
  small = 1.0;                          {small increment}
{*************************************************************************}

type
  rank_record = record                  {record used to rank models}
                  model : integer;      {model number}
                  value : double;       {rank value}
                end;

{*************************************************************************}
var
  d             {operation time for unit i in station j}
                : array[1..n_models,1..n_station] of double;
  Q,            {production quantity for model Q}
  QQ            {counter to keep track of models already sequenced}
                : array[1..n_models] of integer;
  n_units,      {total # of units to produce}
  m             {keeps track of # of units already sequenced}
                : integer;
  c             {launching rate for models}
                : double;
  sstn          {station length lower bound}
                : array[1..n_station] of double;
  rank          {array used to rank models}
                : array[1..n_models] of rank_record;

{*************************************************************************}
procedure get_input;                {reads input from datafile}
  var
    i,j : integer;
    input : text;
  begin
    assign(input,input_fname);
    reset(input);
    for i := 1 to n_models do
      read(input,Q[i]);              {read model quantities}
    for i := 1 to n_models do
      for j := 1 to n_station do
        read(input,d[i,j]);          {read model times for each station}
```

```
    close(input);
  end;

{************************************************************************}
procedure write_input; {prints out input parameters for verifying}
  var
    i,j : integer;
    sum : double;
  begin
    writeln;
    for i := 1 to n_models do
      begin
       {calculate work content for each model}
        sum := 0.0;
        for j := 1 to n_station do
          sum := sum + d[i,j];
        writeln('Total time for model ',i:3,' = ',sum:10:4);
      end;
  end;

{************************************************************************}
function max(a,b : double) : double;
  begin                          {returns the maximum of two values}
    if a < b then
      max := b
    else
      max := a
  end;

{************************************************************************}
function min(a,b : double) : double;
  begin                           {returns the minimum of two values}
    if a > b then
      min := b
    else
      min := a
  end;

{************************************************************************}
procedure init; {calculates launch interval c and station lower bounds}
  var
    i,j : integer;
    ttwc,                {Total Work Content for all stations & models}
    min,maks    : double;
  begin
    n_units := 0;
    for i := 1 to n_models do
      n_units := n_units + Q[i];       {get total # of units}
    ttwc := 0;
    for i := 1 to n_models do          {calc total work content}
      for j := 1 to n_station do
        ttwc := ttwc + d[i,j]*Q[i];
    c := ttwc/n_station/n_units;        {calculate launching rate}
    {display results}
    writeln('Total units : ',n_units:3,' Total Work Content : ',ttwc:8:3,
            ' Launching interval : ',c:7:3);
    writeln;
    writeln('Station  Lower Bound');
    for j := 1 to n_station do
      begin
        min := 1e3;
        maks := 0.0;
        for i := 1 to n_models do
          begin
            if d[i,j] < min then
              min := d[i,j];
            if d[i,j] > maks then
```

```
            maks := d[i,j];
          end;
        sstn[j] := max(maks,2*c-min); {calculate station lower bounds}
        writeln(j:5,sstn[j]:12:4);
      end;
  end;

{************************************************************************}
{rank models according to their calculated rank values}
procedure rank_models;
  var
    temp : rank_record;
    i,j  : integer;
  begin {rank_models}
    for i := 1 to n_models do
      begin
        rank[i].model := i;
        {calulate model ranks}
        rank[i].value := Q[i]*m - (Q[i] - QQ[i])*n_units;
      end;
    {sort models using bubble sort algorithm}
    for j := n_models - 1 downto 1 do
      for i := 1 to j do
        if rank[i].value < rank[i+1].value then
          begin              {swap models}
            temp := rank[i];
            rank[i] := rank[i+1];
            rank[i+1] := temp;
          end;
    {dislplay ranked results}
    writeln;
    writeln('Ranked models : m =',m:3,'   Total Units =',n_units:3);
    writeln('  Rank Model    Q     QQ     Value');
    for j := 1 to n_models do
      begin
        i := rank[j].model;
        writeln(j:5,rank[j].model:5,Q[i]:7,QQ[i]:7,rank[j].value:10:3);
      end;
  end;  {rank_models}

{************************************************************************}
procedure sequence;          {main procedure for sequencing models}
  var
    output : text;           {output file}
    i,j,k : integer;
    return_all,rejected,accepted : boolean;
    increment,               {station length increment}
    dp,                      {end position of operator}
    Total_length             {length of all stations combined}
                             : double;
    limit,                   {station limits}
    dmin,dmin_temp,          {minimum displacement for every station}
    dmax,dmax_temp,          {maximum displacement for every station}
    dm,dm_temp               {last position of operator in station}
                             : array[1..n_station] of double;
    seq                      {sequence of models}
                             : array[1..100] of byte;
  begin {sequence}
    init;          {calc launch interval and station lower bounds}
    increment := large;
    for i := 1 to n_station do        {set limits = lower bounds}
      limit[i] := sstn[i];
    return_all := true;
    while return_all do
      begin
        m := 1;                     {first unit to sequence}
        for i := 1 to n_station do  {initialize dmin,dmax,dm}
```

```
          begin
            dmin[i] := 0;
            dmax[i] := 0;
            dm[i] := 0;
          end;
        for i := 1 to n_models do   {return all products to pool}
          QQ[i] := Q[i];
        return_all := false;
        while (m <= n_units) and (not(return_all)) do
          begin
            rank_models;
            k := 1;  {try first ranked model}
            accepted := false;
            {display calculations}
            writeln;
            writeln('Increment = ',increment:10:5);
            writeln;
            writeln('  k  j model  dm[j]      dp     dm_temp  dmax_temp  dmin_temp dmax-dmin
 limit');
            while (k <= n_models) and (not(accepted)) do
              begin
                if QQ[rank[k].model] <= 0 then
                  k := k + 1       {try next ranked model}
                else
                  begin   {simulate service}
                    j := 1;     {first station}
                    rejected := false;
                    while (j <= n_station) and (not(rejected)) do
                      begin {calculate new temporary values}
                        dp := dm[j] + d[rank[k].model,j];
                        dm_temp[j] := dp - c;
                        dmax_temp[j] := max(dmax[j],dp);
                        dmin_temp[j] := min(dmin[j],dm_temp[j]);
                        {display temporary values}
                        writeln(k:3,j:3,rank[k].model:3,dm[j]:10:4,dp:10:4,dm_temp[j]:10:4,
                                  dmax_temp[j]:10:4,dmin_temp[j]:10:4,(dmax_temp[j]      -
 dmin_temp[j]):10:4,
                                  limit[j]:10:4);

                        if (dmax_temp[j] - dmin_temp[j]) > limit[j] then
                          {station length exceeded, reject this model}
                          begin
                            {
                            writeln;
                            }
                            rejected := true;
                            k := k + 1;  {try next model}
                          end
                        else
                          j := j + 1;  {this station ok, goto next station}
                      end;
                    if not(rejected) then
                      begin {this model is assigned to the sequence}
                        accepted := true;
                        seq[m] := rank[k].model;
                        {all temporary values become permanent}
                        dm := dm_temp;
                        dmin := dmin_temp;
                        dmax := dmax_temp;
                        {decrement unit counter for this model}
                        dec(QQ[rank[k].model]);
                        m := m + 1;    {increment overall unit counter}
                      end;
                  end;  {simulate service}
              end; {k < n_models loop}
            if (k > n_models) then {if all models have been tried}
              begin
```

```
                return_all := true; {return all products to the pool}
                for i := 1 to n_station do  {increment station limits}
                  limit[i] := limit[i] + increment;
              end;
          end; {m < n_units loop}
        if (increment = large) and (not(return_all)) then
          {take smaller increments do get more accurate solution}
          begin
            {put stations back to previous size}
            for i := 1 to n_station do
              limit[i] := limit[i] - increment;
            increment := small; {smaller increments = better solution}
            return_all := true;
          end;
      end; {return_all loop}
    {write output to screen}
    Total_length := 0;
    writeln('Station  Length');
    for i := 1 to n_station do
      begin
        Total_length := Total_length + (dmax[i]-dmin[i]);
        writeln(i:5,dmax[i]-dmin[i]:10:4);
      end;
    writeln;
    writeln('Total Line Length = ',Total_length:10:4);
    writeln('Launching Sequence :');
    writeln('  Unit   Model');
    for i := 1 to n_units do
      writeln(i:5,seq[i]:7);
    {write output to file}
    assign(output,output_fname);
    rewrite(output);
    writeln(output,'Station  Length');
    for i := 1 to n_station do
      writeln(output,i:5,dmax[i]-dmin[i]:10:4);
    writeln(output);
    writeln(output,'Total Line Length = ',Total_length:10:4);
    writeln(output,'Launching Sequence :');
    writeln(output,'  Unit   Model');
    for i := 1 to n_units do
      writeln(output,i:5,seq[i]:7);
    close(output)
  end;  {sequence}
{*************************************************************************}
{main program}
begin
  get_input;
  write_input;
  sequence;
end.
```

```
{*************************************************************************
                              PROGRAM7.PAS
*************************************************************************
          Dar-El & Cother method for line length minimization
-------------------------------------------------------------------------

This program finds the optimal sequence and station lengths for a mixed
model line. The method used is Dar-El & Cother line length minimization
algorithm. This program is for OPEN workstations.

The format of the datafile is as follows:

1. The model quantities Q1, Q2, ...
2. For each model, the station times for all stations

See SAMPLE4.dat for an example of an input file. No specific formatting
of the data is required. The program calculates the launching rate c.

written by P. R. Smith, November 1990. Tested with Turbo Pascal v5.0
*************************************************************************}

{$N+}

const
  n_station = 19;                       {number of stations}
  n_models = 6;                         {number of models}
  input_fname = 'SAMPLE4.DAT';          {input filename}
  output_fname = 'TEMP.OUT';            {ouput filename}
  large = 5.0;                          {large increment}
  small = 1.0;                          {small increment}
{*************************************************************************}


type
  rank_record = record                  {record used to rank models}
                  model : integer;      {model number}
                  value : double;       {rank value}
                end;

{*************************************************************************}
var
  d             {operation time for unit i in station j}
                : array[1..n_models,1..n_station] of double;
  Q,            {production quantity for model Q}
  QQ            {counter to keep track of models already sequenced}
                : array[1..n_models] of integer;
  n_units,      {total # of units to produce}
  m             {keeps track of # of units already sequenced}
                : integer;
  c             {launching rate for models}
                : double;
  rank          {array used to rank models}
                : array[1..n_models] of rank_record;

{*************************************************************************}
procedure get_input;                {reads input from datafile}
  var
    i,j : integer;
    input : text;
  begin
    assign(input,input_fname);
    reset(input);
    for i := 1 to n_models do
      read(input,Q[i]);              {read model quantities}
    for i := 1 to n_models do
      for j := 1 to n_station do
        read(input,d[i,j]);          {read model times for each station}
```

```
    close(input);
  end;

{************************************************************************}
procedure write_input; {prints out input parameters for verifying}
  var
    i,j : integer;
    sum : double;
  begin
    writeln;
    for i := 1 to n_models do
      begin
       {calculate work content for each model}
        sum := 0.0;
        for j := 1 to n_station do
          sum := sum + d[i,j];
        writeln('Total time for model ',i:3,' = ',sum:10:4);
      end;
  end;

{************************************************************************}
function max(a,b : double) : double;
  begin                          {returns the maximum of two values}
    if a < b then
      max := b
    else
      max := a
  end;

{************************************************************************}
function min(a,b : double) : double;
  begin                           {returns the minimum of two values}
    if a > b then
      min := b
    else
      min := a
  end;

{************************************************************************}

procedure init; {calculates launch interval c and station lower bounds}
  var
    i,j : integer;
    ttwc : double;  {Total Work Content for all stations & models}
  begin
    n_units := 0;
    for i := 1 to n_models do
      n_units := n_units + Q[i];   {calc total number of units}
    ttwc := 0;
    for i := 1 to n_models do      {calc total work content}
      for j := 1 to n_station do
        ttwc := ttwc + d[i,j]*Q[i];
    c := ttwc/n_station/n_units;   {launching rate}
    writeln('Total units : ',n_units:3,' Total Work Content : ',ttwc:8:3,
            ' Launching interval : ',c:7:3);
  end;

{************************************************************************}
{rank models according to their calculated rank values}
procedure rank_models;
  var
    temp : rank_record;
    i,j  : integer;
  begin {rank_models}
    for i := 1 to n_models do
      begin
        rank[i].model := i;
```

```
      {calulate model ranks}
      rank[i].value := Q[i]*m - (Q[i] - QQ[i])*n_units;
    end;
  {sort models using bubble sort algorithm}
  for j := n_models - 1 downto 1 do
    for i := 1 to j do
      if rank[i].value < rank[i+1].value then
        begin              {swap models}
          temp := rank[i];
          rank[i] := rank[i+1];
          rank[i+1] := temp;
        end;
  {dislplay ranked results}
  writeln;
  writeln('Ranked models : m =',m:3,'   Total Units =',n_units:3);
  writeln('  Rank Model    Q     QQ     Value');
  for j := 1 to n_models do
    begin
      i := rank[j].model;
      writeln(j:5,rank[j].model:5,Q[i]:7,QQ[i]:7,rank[j].value:10:3);
    end;
end;  {rank_models}


{************************************************************************}
procedure sequence;          {main procedure for sequencing models}
  var
    i,j,k : integer;
    return_all,              {binary flags}
    accepted                 : boolean;
    increment,               {station length increment}
    dp,                      {end position of operator}
    Total_length,            {length of all stations combined}
    limit,                   {Limit on total line length}
    temp                     : double;
    dmin,dmin_temp,          {minimum displacement for every station}
    dmax,dmax_temp,          {maximum displacement for every station}
    dm,dm_temp,              {last position of operator in station}
    delta,delta_temp         {distance between stations for closed stations}
                             : array[1..n_station] of double;
    seq                      {sequence of models}
                             : array[1..100] of byte;
    output                   : text; {outputfile}
  begin {sequence}
    init; {calc launch interval and station lower bounds}
    {calc lower bound on station length}
    limit := 0.0;
    for i := 1 to n_models do
      begin
        temp := 0.0;
        for j := 1 to n_station do
          temp := temp + d[i,j];
        if temp > limit then
          limit := temp;  {get model with greatest work content}
      end;
    increment := large;   {set large increment for faster search}
    return_all := true;
    while return_all do
      begin
        m := 1;            {first model}
        for i := 1 to n_station do     {initialize dmin,dmax,dm}
          begin
            dmin[i] := 0;
            dmax[i] := 0;
            dm[i] := 0;
            delta[i] := 0;
          end;
```

```
        delta_temp[n_station] := 0;
        for i := 1 to n_models do   {return all products to pool}
          QQ[i] := Q[i];
        return_all := false;
        while (m <= n_units) and (not(return_all)) do
          begin
            rank_models; {rank models according to selection heuristic}
            k := 1;      {try first ranked model}
            accepted := false;
            while (k <= n_models) and (not(accepted)) do
              begin
                if QQ[rank[k].model] <= 0 then
                  k := k + 1              {try next ranked model}
                else
                  begin   {simulate service}
                    {display results}
                    writeln;
                    writeln('Increment = ',increment:10:5);
                    writeln;
                    writeln('  k   j model  dm[j]       dp      dm_temp  dmax_temp dmin_temp
delta_temp');
                    for j := 1 to n_station do
                      begin {calculate new temporary values}
                        dp := dm[j] + d[rank[k].model,j];
                        dm_temp[j] := dp - c;
                        dmax_temp[j] := max(dmax[j],dp);
                        dmin_temp[j] := min(dmin[j],dm_temp[j]);
                          {calc delta value for open stations}
                        if j < n_station then
                          delta_temp[j] := max(dp-dm[j+1],delta[j]);
                        {display results}
                        writeln(k:3,j:3,rank[k].model:3,dm[j]:10:4,dp:10:4,dm_temp[j]:10:4,
                                dmax_temp[j]:10:4,dmin_temp[j]:10:4,delta_temp[j]:10:4);
                      end;
                    {Calculate Total Length of line}
                    Total_length := 0;
                    for j := 1 to (n_station-1) do
                      Total_length := Total_length + delta_temp[j];
                    Total_length := Total_length + dmax_temp[n_station]-dmin_temp[1];
                    {display results}
                    writeln;
                    writeln('Total  Line  length  =  ',Total_length:10:4,'        Limit  =
',limit:10:4);
                    writeln;
                      {check if line limit exceeded?}
                    if (Total_length > limit) then
                      k := k + 1      {try next model}
                    else
                      begin  {this model is assigned to the sequence}
                        accepted := true;
                        seq[m] := rank[k].model;
                          {all temporary values become permanent}
                        dm := dm_temp;
                        dmin := dmin_temp;
                        dmax := dmax_temp;
                        delta := delta_temp;
                          {decrement counter for this model}
                        dec(QQ[rank[k].model]);
                        m := m + 1;  {increment overall unit counter}
                      end;
                  end;    {simulate service}
              end; {k < n_models loop}
            if (k > n_models) then {if all models have been tried}
              begin
                return_all := true;   {return all products to pool}
                limit := limit + increment; {increase line limit}
              end;
```

```
          end; {m < n_units loop}
        if (increment = large) and (not(return_all)) then
           {set increment to small for more accurate solution}
          begin
             {set station to previous limit}
            limit := limit - increment;
            increment := small;  {set small increment}
            return_all := true;  {return all products to pool}
          end;
      end; {return_all loop}
    {write results to screen}
    writeln;
    writeln('Total Line Length = ',Total_length:10:4);
    writeln('Launching Sequence :');
    writeln('  Unit   Model');
    for i := 1 to n_units do
      writeln(i:5,seq[i]:7);
    {write results to file}
    assign(output,output_fname);
    rewrite(output);
    writeln(output,'Total Line Length = ',Total_length:10:4);
    writeln(output,'Launching Sequence :');
    writeln(output,'  Unit   Model');
    for i := 1 to n_units do
      writeln(output,i:5,seq[i]:7);
    close(output)
  end;  {sequence}
{*************************************************************************}
{main program}
begin
  get_input;
  write_input;
  sequence;
end.
```

```
/*************************************************************************
                              PROGRAM8.PRO
*************************************************************************
                    Exhaustive search sequencing

This program finds the optimal sequence for a mixed model line. The
models are ranked according to the selection heuristic proposed by
Dar-El. The program will therefore start with the solution given by the
Dar-El algorithm, but will do an exhaustive search via backtracking.
This program is written for closed station but can be easily adapted to
open stations.

The station times are read from an inputfile. The inputfile must be in
a Prolog format. Refer to SAMPLE5.DAT for an example of an inputfile.
Other parameters like launching interval, number of models etc. are
specified in the program itself. To run the program, simply type go at
the goal prompt.

written by P. R. Smith, November 1990. Tested with Turbo Prolog v2.0
*************************************************************************/
domains
  intlist = integer*
  reallist = real*
  tree = reference t(val, tree, tree)
  val  = integer

/*************************************************************************/
constants
  c = 4.397105                      % launching interval
  n_units = 20                      % total number of units
  input_file = "SAMPLE5.DAT"        % input filename

/*************************************************************************/
predicates
  member(integer,intlist)
  build(integer,intlist,intlist,intlist,real)
  append(intlist,intlist,intlist)
  decrease(integer,integer,intlist,intlist)
  sequence(intlist,intlist)
  max(real,real,real)
  min(real,real,real)
  calc(intlist,real,real,real,real,real,integer)
  station(intlist,real)
  check_len(intlist,real)
  get_min(intlist,real)
  sumlist(reallist,real)
  sumlist2(intlist,integer)
  q(integer,integer)
  insert(integer, tree)
  instree(intlist, tree)
  sort(intlist, intlist)
  treemembers(integer, tree)
  go

/*************************************************************************/
database
  d(integer,integer,real)       % used for storing station times
  m(intlist,real)               % used for storing minimum
  r(integer,real)               % used for storing list ranks

/*************************************************************************/
clauses

  q(1,7).       % production requirements for each model
  q(2,6).       % must be in sequential order by model number
  q(3,3).
```

```
  q(4,1).
  q(5,1).
  q(6,2).

/*************************************************************************
go is the main clause of this program. It clears the database, loads
the input file and launches the sequencing search procedure
*************************************************************************/
go :-
  clearwindow,                  % clears the window
  retractall(_),                % clear internal database
  consult(input_file),          % load input data
  findall(J,q(_,J),List),       % find all possible model quantities
                                % put it all in a list
  sequence(List,[]).            % launches sequencing procedure

/*************************************************************************
The sequence clause is the search engine of this program. The clause is
called recursively, until all models sequenced, and then fails. It then
backtracks to the member clause.
syntax : sequence(List1,List2) where
  List1 = list of model quantities to keep track of models available
          for assignment.
  List2 = list of model numbers assigned to sequence.
*************************************************************************/
sequence(List1,Answer) :-
    % first clause, go here if all models sequenced.
  sumlist2(List1,0),            % if all models are sequenced
  check_len(Answer,Total),      % compute length and update minimum
  cursor(10,8),
  write(Answer),                % display results
  write("       ",Total),
  fail.                         % backtrack to member clause
sequence(List1,Answer) :-
    % second clause, go here if models are available for assignment
  sumlist2(List1,Sum),          % get total units still available
  Sum > 0,                      % if models are available
  build(1,List1,[],[H|T],Sum),  % rank models
  sort([H|T],List3),            % sort according to rank
  member(X,List3),              % get a member - backtracking point
  decrease(X,1,List1,List2),    % decrease this model counter
  append(Answer,[X],Answer2),    % append model # to sequence list
  sequence(List2,Answer2).      % call sequence recursively

/*************************************************************************
decrease decreases the N th element in the list by 1. In this program
it decreases the model counter of model N by 1.
*************************************************************************/
decrease(N,N,[H|T],[H1|T]) :-   % N'th element found
  H1 = H - 1,                   % decreases head of list
  !.                            % do not backtrack
decrease(N,I,[H|T1],[H|T2]) :-  % N'th element not found
  I2 = I + 1,                   % increase list element counter
  decrease(N,I2,T1,T2).         % call recursively

/*************************************************************************
build makes a list of all model numbers still available for sequencing
and calculates the ranked values according to the selection heuristic
by Dar-El & Cother.
syntax build(I,List1,List2,Answer,Sum) where
  I : model counter
  List1 : list of current model quantities
  List2 : list of model numbers available for assignment being built
  Answer: returned value after all models done
  Sum   : Sum of List1, used to calculate model ranks.
*************************************************************************/
build(_,[],X,X,_).                  % finish building - return answer
```

```
build(I,[H|T],List,Answer,Sum) :-
  H > 0,                              % if models of this type still
                                      % available
  !,                                  % do not backtrack
  M = n_units - Sum + 1,              % M'th unit in sequence
  q(I,Q),                             % get original production quantity
  Rank = Q*M - (Q-H)*n_units,         % calc. model rank
  retractall(r(I,_)),                 % clear database first
  asserta(r(I,Rank)),                 % insert rank value in database
  I2 = I + 1,                         % increment model counter
  append(List,[I],List2),             % add model I to list of available
                                      % models for sequencing
  build(I2,T,List2,Answer,Sum).       % call recursively with new values
build(I,[_|T],List,Answer,Sum) :-     % go here if no models of this
                                      % type
  I2 = I + 1,                         % just increase model counter
  build(I2,T,List,Answer,Sum).        % call recursively with List
                                      % unchanged.

/*************************************************************************
The member clause is a very important part of the search algorithm.
There is no cut so all possible elements can be obtained by
backtracking.
*************************************************************************/
member(X,[X|_]).                  % check first element in list (head)
member(X,[_|T]) :-
  member(X,T).                    % check rest of the list (tail)

/*************************************************************************
append adds two lists to form a third.
syntax : append(List1,List2,List3). List3 := List1 + List2
*************************************************************************/
append([], List, List).           % trivial case
append([X|L1], List2, [X|L3]) :-
  append(L1, List2, L3).          % recursive call


/*************************************************************************
max returns the largest value of X and Y
*************************************************************************/
max(X,Y,X) :-     % return X
  X > Y,!.        % if X > Y
max(_,Y,Y).       % else return Y

/*************************************************************************
min returns the smallest value of X and Y
*************************************************************************/
min(X,Y,X) :-     % return X
  X < Y,!.        % if X < Y
min(_,Y,Y).       % else return Y

/*************************************************************************
calc simulates the service of the units and tracks the movement of the
operator in a given station. It goes through all the models in the list
passed to it and then returns the minimum and maximum displacements of
the operator.
syntax: calc(Modellist,Max1,Min1,Max2,Min2,Current,Station) where
Modellist : list of models to simulate, i.e. sequence so far.
Max1      : temporary value to keep track of current maximum
Min1      : temporary value to keep track of current minimum
Max2      : returned maximum value
Min2      : returned minimum value
Current   : current position of operator
Station   : station number to simulate
*************************************************************************/
calc([],Max,Min,Max,Min,_,_). % all models done, return values found
calc([H|T],Max,Min,Nmax,Nmin,Current,Station) :-
```

```
  d(H,Station,X),                    % get time for model H at Station
  Current2 = Current + X,            % operator works on element
  max(Max,Current2,Max2),            % update maximum
  Current3 = Current2 - c,           % next unit arrives
  min(Min,Current3,Min2),            % update minimum
    % call recursively with remainder of models (tail of list)
  calc(T,Max2,Min2,Nmax,Nmin,Current3,Station).

/*************************************************************************
station(List,Length) is called with List bound to the model list to
simulate and returns the length of the station in Length. All station
lengths can be found via backtracking with this clause
*************************************************************************/
station(Seq,Len) :-
  d(1,Station,_),                    % get any station
  calc(Seq,0,0,Max,Min,0,Station),   % calculate max and min values
  Len = Max - Min.                   % calculate length of station

/*************************************************************************
check_len(List,Total) is called with List bound to the list of models
assigned to a sequence so far and returns the Total length of the
entire assembly line in Total. It also updates the minimum length found
so far.
*************************************************************************/
check_len(Seq,Total) :-
  findall(Len,station(Seq,Len),List),  % get all station lengths
                                       % through backtracking
  sumlist(List,Total),                 % sum the list to get total
  get_min(Seq,Total).                  % update minimum values

/*************************************************************************
get_min(List,X) updates the minimum line length found so far. If X is
smaller than the minimum, X becomes the new minimum and List, the
sequence of models with line length X is stored as well.
*************************************************************************/
get_min(_,Total) :-
  m(_,Min),                  % get current minimum
  Min <= Total,              % if not better solution
  !.                         % return and do not backtrack
get_min(Seq,Total) :-        % else if better solution,
  retractall(m(_,_)),        % remove old one
  asserta(m(Seq,Total)),     % store new one
  cursor(13,1),              % display new optimal solution
  write("Min :  ",Seq,"       ",Total).

/*************************************************************************
sumlist sums all elements in a list of reals.
*************************************************************************/
sumlist([],0).                 % trivial case
sumlist([H|T],S):-
  sumlist(T,S2),               % recursive call
  S = S2 + H.

/*************************************************************************
sumlist2 sums all elements in a list of integers.
*************************************************************************/
sumlist2([],0).                 % trivial case
sumlist2([H|T],S):-
  sumlist2(T,S2),               % recursive call
  S = S2 + H.

/*************************************************************************
The following clauses are all used to sort a list of integers. The
clauses are taken directly from the Turbo Prolog User's Manual. The
sorting method used is tree based sorting and is therefore very fast.
*************************************************************************/
insert(Val, t(Val, _, _)) :- !.
```

```
insert(Val, t(Val1, Tree, _)) :-
  r(Val,X),r(Val1,Y),
  X>Y,
  !,
  insert(Val, Tree).
insert(Val, t(_, _, Tree)) :-
  insert(Val, Tree).

instree([], _).
instree([H|T], Tree) :-
  insert(H, Tree),
  instree(T, Tree).

treemembers(_, T) :- free(T),
  !,
  fail.
treemembers(X, t(_, L, _)) :-
  treemembers(X, L).
treemembers(X, t(Refstr, _, _)) :-
  X = Refstr.
treemembers(X, t(_, _, R)) :-
  treemembers(X, R).

sort(L, L1) :-
  instree(L, Tree),
  findall(X, treemembers(X, Tree), L1).
```

```
/*****************************************************************************
                              PROGRAM9.PRO
*****************************************************************************
          Exhaustive search sequencing with step option
          ---------------------------------------------

This program finds the optimal sequence for a mixed model line. The
models can be sequenced partially. For example, the first ten models
can be sequenced, then the next ten, and so on. Alternatively, a
starting sequence can be given to the program, and the search algorithm
will start with the input sequence and continue from there on. The
initial sequence is just given as a starting solution, a full
exhaustive search via backtracking will still be done. In this program
the models are not ranked and sorted as in the previous program, this
speeds up the algorithm.

The station times are read from an inputfile. The inputfile must be in
a Prolog format. Refer to SAMPLE5.DAT for an example of an inputfile.
Other parameters like launching interval, number of models etc. are
specified in the program itself. The sequence to start with is given in
the constants part of the program. To run the program, simply type go
at the goal prompt.

written by P. R. Smith, November 1990. Tested with Turbo Prolog v2.0
*****************************************************************************/
domains
  intlist = integer*
  reallist = real*

/*****************************************************************************/
constants
  c = 4.397105                      % launching interval
  n_units = 20                      % total number of units
  input_file = "SAMPLE5.DAT"        % input filename
  stop = 10                         % stop sequence at stop units
  list = [1,2,3,4,5]                % initial list to start with

/*****************************************************************************/
predicates
  member(integer,intlist)
  build(integer,intlist,intlist,intlist)
  append(intlist,intlist,intlist)
  decrease(integer,integer,intlist,intlist)
  sequence(intlist,intlist)
  max(real,real,real)
  min(real,real,real)
  calc(intlist,real,real,real,real,real,integer)
  station(intlist,real)
  check_len(intlist,intlist,real)
  check_list(intlist,intlist)
  get_min(intlist,intlist,real)
  sumlist(reallist,real)
  sumlist2(intlist,integer)
  delete(integer,intlist,intlist)
  q(integer,integer)
  go

/*****************************************************************************/
database
  d(integer,integer,real)      % used for storing station times
  m(intlist,intlist,real)      % used to store minimum
  r(intlist)                   % store initial values

/*****************************************************************************/
clauses

  q(1,7).      % production requirements for each model
```

```
  q(2,6).      % must be in sequential order by model number
  q(3,3).
  q(4,1).
  q(5,1).
  q(6,2).

/*************************************************************************
go is the main clause of this program. It clears the database, loads
the input file, sets up the initial values and launches the sequencing
search procedure
*************************************************************************/
go :-
  clearwindow,                  % clear screen
  retractall(_),                % clear internal database
  consult(input_file),          % load station times in database
  findall(J,q(_,J),List),       % find all models to sequence
  asserta(r(list)),             % put initial values in database
  sequence(List,[]).            % start sequencing procedure

/*************************************************************************
The sequence clause is the search engine of this program. The clause is
called recursively, until all models sequenced, and then fails. It then
backtracks to the member clause.
syntax : sequence(List1,List2) where
  List1 = list of model quantities to keep track of models available
          for assignment.
  List2 = list of model numbers assigned to sequence.
*************************************************************************/
sequence(List1,Answer) :-
    % first clause, go here if all models up to stop are sequenced.
  sumlist2(List1,Sum),                % sum list of quantities
  Sum = n_units - stop,               % if all models sequenced
  check_len(List1,Answer,Total),      % compute length and
                                      % update minimum
  cursor(10,8),
  write(Answer),                      % display results
  write("        ",Total),
  fail.                               % backtrack to member clause
sequence(List1,Answer) :-
    % second clause, go here if models are available for assignment
  sumlist2(List1,Sum),          % get total units still available
  Sum > n_units - stop,         % check if models are available
  build(1,List1,[],List3),      % make list of model numbers available
  check_list(List3,List4),      % check against initial values
  member(X,List4),              % get a member - backtracking point
  append(Answer,[X],Answer2),   % append model # to sequence list
  decrease(X,1,List1,List2),    % decrease this model counter
  sequence(List2,Answer2).      % call sequence recursively

/*************************************************************************
check_list checks a list against the initial values required. If no
initial value is specified, the list is returned unchanged. If there
are initial values, the list is modified so that the required model is
at the start (head) of the list. Once the initial value constraint is
satisfied, the constraint is removed from the database to allow full
backtracking.
*************************************************************************/
check_list(X,X) :-              % return list unchanged
  not(r(_)),                    % if no initial values
  !.                            % do not backtrack
check_list(X,X) :-              % return list unchanged
  r([]),                        % if empty list in database
  retractall(r(_)),             % remove restriction
  !.                            % do not backtrack
check_list(List,[H|List2]) :-
    % a list with at least one element exist, the list is modified so
    % that the element is at the head of the list.
```

```
    % restriction is removed at end of this clause
  r([H]),!,                    % if only one element left
  delete(H,List,List2),        % delete from list
  retractall(r(_)).            % remove restriction
check_list(List,[H|List2]) :-
    % a list with more than one initial value element exist, the
    % element is moved to the head of the list and a new restriction
    % inserted in the database
  r([H|T]),                    % get list of initial values
  delete(H,List,List2),        % delete element from list
  retractall(r(_)),            % remove old restriction
  asserta(r(T)).               % insert new restriction

/*************************************************************************
decrease decreases the N th element in the list by 1. In this program
it decreases the model counter of model N by 1.
*************************************************************************/
decrease(N,N,[H|T],[H1|T]) :-  % N'th element found
  H1 = H - 1,                  % decreases head of list
  !.                           % do not backtrack
decrease(N,I,[H|T1],[H|T2]) :- % N'th element not found
  I2 = I + 1,                  % increase list element counter
  decrease(N,I2,T1,T2).        % call recursively

/*************************************************************************
delete deletes an element from a list. If the element is not in the
list, the clause fails.
*************************************************************************/
delete(X,[X|T],T) :- !.              % deletes an element from a list
delete(X,[H|T],[H|Newlist]) :-
    delete(X,T,Newlist).


/*************************************************************************
build is called with the list of model quantities bound and returns a
list of model numbers available to assign. Thus build simply returns
the list of model numbers for which the model quantities are greater
than zero.
*************************************************************************/
build(_,[],X,X).                     % all models done, return
build(I,[H|T],List,Answer) :-
  H > 0,!,                           % if models are available
  I2 = I + 1,                        % increment model counter
  append(List,[I],List2),            % add model to available list
  build(I2,T,List2,Answer).          % call recursively
build(I,[_|T],List,Answer) :-        % no models available
  I2 = I + 1,                        % just increment counter
  build(I2,T,List,Answer).           % call recursively, do not add
                                     % model

/*************************************************************************
The member clause is a very important part of the search algorithm.
There is no cut so all possible elements can be obtained by
backtracking.
*************************************************************************/
member(X,[X|_]).             % check first element in list (head)
member(X,[_|T]) :-
  member(X,T).               % check rest of the list (tail)

/*************************************************************************
append adds two lists to form a third.
syntax : append(List1,List2,List3). List3 := List1 + List2
*************************************************************************/
append([], List, List).              % trivial case
append([X|L1], List2, [X|L3]) :-
  append(L1, List2, L3).             % recursive call
```

```
/*************************************************************************
max returns the largest value of X and Y
*************************************************************************/
max(X,Y,X) :-     % return X
  X > Y,!.        % if X > Y
max(_,Y,Y).       % else return Y

/*************************************************************************
min returns the smallest value of X and Y
*************************************************************************/
min(X,Y,X) :-     % return X
  X < Y,!.        % if X < Y
min(_,Y,Y).       % else return Y

/*************************************************************************
calc simulates the service of the units and tracks the movement of the
operator in a given station. It goes through all the models in the list
passed to it and then returns the minimum and maximum displacements of
the operator.
syntax: calc(Modellist,Max1,Min1,Max2,Min2,Current,Station) where
Modellist : list of models to simulate, i.e. sequence so far.
Max1      : temporary value to keep track of current maximum
Min1      : temporary value to keep track of current minimum
Max2      : returned maximum value
Min2      : returned minimum value
Current   : current position of operator
Station   : station number to simulate
*************************************************************************/
calc([],Max,Min,Max,Min,_,_). % all models done, return values found
calc([H|T],Max,Min,Nmax,Nmin,Current,Station) :-
  d(H,Station,X),                  % get time for model H at Station
  Current2 = Current + X,          % operator works on element
  max(Max,Current2,Max2),          % update maximum
  Current3 = Current2 - c,         % next unit arrives
  min(Min,Current3,Min2),          % update minimum
    % call recursively with remainder of models (tail of list)
  calc(T,Max2,Min2,Nmax,Nmin,Current3,Station).

/*************************************************************************
station(List,Length) is called with List bound to the model list to
simulate and returns the length of the station in Length. All station
lengths can be found via backtracking with this clause
*************************************************************************/
station(Seq,Len) :-
  d(1,Station,_),                  % get any station
  calc(Seq,0,0,Max,Min,0,Station), % calculate max and min values
  Len = Max - Min.                 % calculate length of station

/*************************************************************************
check_len(QList,List,Total) is called with List bound to the list of
models assigned to a sequence so far and returns the Total length of
the entire assembly line in Total. It also updates the minimum length
found so far. The model quantity list is also stored with the minimum
values.
*************************************************************************/
check_len(Qlist,Seq,Total) :-
  findall(Len,station(Seq,Len),List),   % get all station lengths
                                        % through backtracking
  sumlist(List,Total),                  % sum the list to get total
  get_min(Qlist,Seq,Total).             % update minimum values


/*************************************************************************
get_min(List,X) updates the minimum line length found so far. If X is
smaller than the minimum, X becomes the new minimum and List, the
sequence of models with line length X and remaining model quantities
are stored as well.
```

```
*************************************************************************/
get_min(_,_,Total) :-
  m(_,_,Min),                    % get current minimum
  Min < Total,                   % if not better solution
  !.                             % return and do not backtrack
get_min(Qlist,Seq,Total) :-      % else if better solution
  retractall(m(_,_,_)),          % remove old one
  asserta(m(Qlist,Seq,Total)),   % store new one
  cursor(13,1),                  % display new optimal solution
  write("Min :  ",Seq,"        ",Total).

/*************************************************************************
sumlist sums all elements in a list of reals.
*************************************************************************/
sumlist([],0).                    % trivial case
sumlist([H|T],S):-
  sumlist(T,S2),                  % recursive call
  S = S2 + H.

/*************************************************************************
sumlist2 sums all elements in a list of integers.
*************************************************************************/
sumlist2([],0).                   % trivial case
sumlist2([H|T],S):-
  sumlist2(T,S2),                 % recursive call
  S = S2 + H.
```