COMBINATORIAL ALGORITHMS FOR STACKING PROBLEMS

by

ANASTASIOS HARALAMPOS ASLIDIS

MS in Operations Research, MIT, 1987

MS in Ocean Systems Management, MIT, 1984

Diploma in Naval Architecture
and Marine Engineering, National
Technical University of Athens, 1983

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

Massachusetts Institute of Technology

January 1989

Signature of Author_____
Department of Ocean Engineering
January 1989

Certified by_____
Ernst G. Frankel
Thesis Supervisor

Accepted by_____
A. Douglas Carmichael
Chairman, Department Committee
on Graduate Studies

COMBINATORIAL ALGORITHMS FOR STACKING PROBLEMS

by
Anastasios Haralampos Aslidis

Submitted to the Department of Ocean Engineering in
January 1989 in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in Ocean Systems Management

ABSTRACT

This Ph.D. thesis deals with the minimization of the overstowage cost in stacking operations. Overstowage cost is the number of necessary reshuffles of items to provide access to those ones that must be delivered at a certain time (port), but are blocked by others with later delivery times. A typical example of a situation where overstowage needs to be managed is containership operations. In that case a set of placement and stability constraints must be satisfied, too.

The one-stack overstowage problem (OSOP) is examined. This problem is solved by a dynamic programming algorithm. The principle of the algorithm is based on the property of the problem to decompose under certain assumptions into problems of smaller size. It is also proven that the rearrangement policy of the items can be expressed as an M-component vector, where M is the number of times when picking up or delivery of items occurs (i.e. the number of ports for the containership problem). The algorithm runs in $O(M^3)$ time.

A number of extensions and generalizations of the OSOP are also pursued, including the consideration of probabilistic storage (shipment) requirements and the incorporation of stability constraints. The former is solved by an approximation scheme, while a polynomial time dynamic programming algorithm is developed for the latter.

The multi-stack overstowage problem (MSOP) is examined next. This problem is significantly more difficult than the OSOP and it is conjectured to be NP-complete. The analysis leads to a set of heuristic algorithms. Furthermore, these heuristics are customized to efficiently solve the containership operations case (without stability or placement constraints). For the latter problem the performance of the heuristics is possible to be evaluated.

The thesis also looks at another kind of stacking operations which also leads to overstowage, the "use-and-restack" case. Certain versions of the one-stack "use-and-restack" problem are formulated and solved efficiently, under certain assumptions.

Thesis Supervisor: E. G. Frankel
Title: Professor of Marine Systems

ACKNOWLEDGEMENTS I

I would like to thank my Thesis Committee, Professors E. G. Frankel, H. N. Psaraftis, and J. Orlin, for their guidance and support during the time this research was performed.

I feel particularly grateful to my Thesis Committee Chairman, Professor E. G. Frankel of the Department of Ocean Engineering, for both what I have learned working with him the past four and a half years and for his financial support. For the latter I also thank the MIT Center for Advanced Engineering Studies, and my family.

Finally, I would like to thank Ms. Sheila McNary for typing this thesis.

ACKNOWLEDGEMENTS II

Completing a Doctor of Philosophy Degree is not only an academic exercise, but also one in learning how to deal with people, ideas, success, failure and, above all, yourself. There is a number of people who have contributed towards that throughout my life in different capacities and times. All those I warmly thank.

First, I would like to express my gratitude and love to my family for their love and support, and for teaching me (in their own way) to set high standards for myself. I cannot think of a better way for a father to love his son than my father's way.

Complementing the support and caring of my family, two of my elementary school teachers, Mr. Kanonidis and Mr. J. Sylleos, provided me the stimulus towards analytical and methodological thinking. The latter was reinforced during my high school years by my mathematics professors, Mr. Holidis and Mr. Trifidis. They, along with my tutor Mr. S. Georgiadis, have given me a solid background in the sciences, which has continued to be an invaluable tool for me. In parallel, another professor of mine, Mr. Toumpanos, introduced me to the ideas of the classical Greek philosophers and the modern Greek literature, providing the required balance in my thought.

I have also greatly benefitted from my studies at the National Metsovean Polytechnic (National Technical University of Athens), where I obtained a fine engineering education. My interaction with people such as Professor T. Loukakis, Prof. J. Ionnides, Dr. P. Kaklis, and Dr. G. Tsiapiras (among others) has been decisive in setting my career goals.

It would be difficult to express my appreciation to the Massachusetts Institute of Technology for giving me the opportunity to be among people of high caliber and distinguished merit. Each of my professors has offered me some unique knowledge and experience. To that extent, I would like to thank Professors E. G. Frankel, H. N. Psaraftis, J. Orlin, H. Marcus, Z. Zannetos, T. L. Magnanti, J. Kildow, A. Drake, J. Sterman, and J. Cox, among many others.

I also feel very lucky to have been - throughout these years - among classmates, colleagues, and friends, whose absence has always made me feel sad, every time I had to switch a school, university, or a city. It is beyond a doubt that much of my character has been influenced by my relations with them.

Finally, I hope I will always be surrounded by and given the love, motivation, and happiness that Maria has shared with me the last year. I further hope my feelings for her are my only violation of R. Kipling's "IF".

TO MAROULAKI

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION TO OVERSTOWAGE PROBLEMS

1.1 Definition of Overstowage

This thesis deals with the management of overstowage. Overstowage is a condition that arises very often in operations which involve stacking of items. "Stacking" is highly correlated with storage of items. There are innumerable occasions in which a group of items (usually boxes or containers) needs to be stored for future use. It is very often the case that the items are stored by being placed one of top of the other. The later may not be always possible. For example, the items may have different sizes (length, width) or their physical characteristics (strength, etc.) may be such that they cannot be stacked. However, when the items are relatively large and have approximately the same size, storing them in stacks is the cheapest alternative because no supporting structure (cells) is required, and also minimum space is occupied, since the stacks can be placed very close to each other. For the latter reason, stacking is very common in practice.

Nevertheless, nothing in this world comes for free. The low cost of storing items in stacks is counterbalanced by the limited accessibility of the items. Suppose that boxes (A, B, C, and D) of equal size are stored in a stack as in Figure 1.1. Box A is on top of the stack and it can be retrieved very easily. That is, there exists direct access to box A. It is not the same with any of the other boxes, though. Retrieval of box B requires first the removal of box A. Similarly, to retrieve box C requires removal of boxes A and B and to retrieve box D it takes

the removal of A, B, and C first.

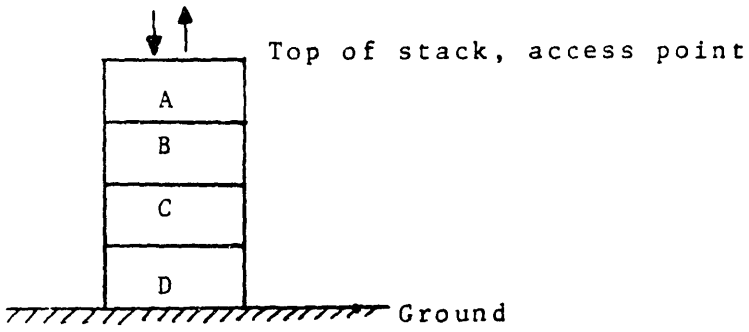


Figure 1.1 - A Stack with Four Items

The reason for the inconvenience in retrieving the items of a stack is the one point access to the item through the top of the stack only.

If item B in Figure 1.1 needs to be retrieved before item A, then item A has to be removed to allow retrieval of B. After B is removed, A can be placed back on the stack. In a situation like this, item A is said to be overstowed. The temporary removal from and placement back onto the stack of item A is called rehandle or rearrangement of item A.

It is natural to ask why an item is overstowed. Several reasons can be identified.

i. If it is not known what item is going to be retrieved first at the moment the items are placed onto the stack, then it is very probably that some items are overstowed and need to be rearranged.

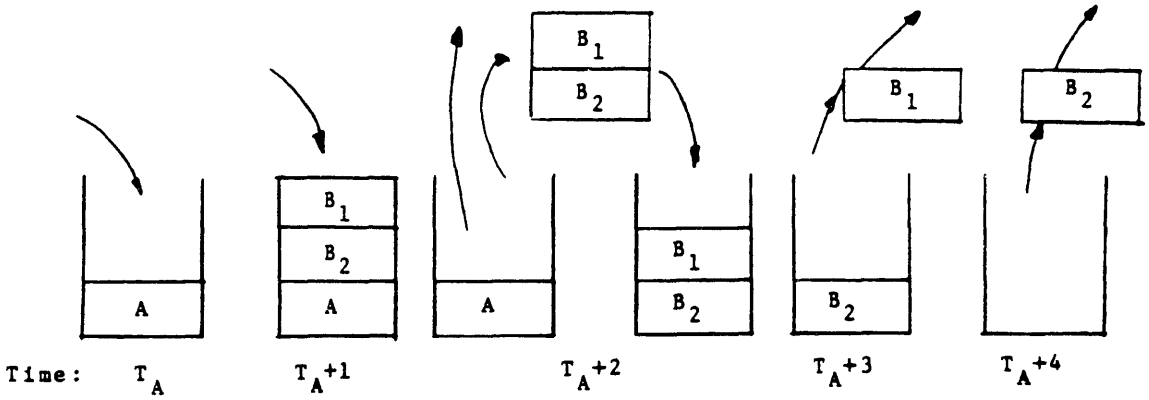
ii. If the items become available for storage at different times, then depending on the retrieval times of them, some of the items may be overstowed.

iii. If items are overstowed simply because of bad planning.

In the first two cases overstowage cannot be avoided.[1] However, it can be minimized. In the third case, any overstowage that is not due to "external" reasons (as in case (ii)) can be eliminated.

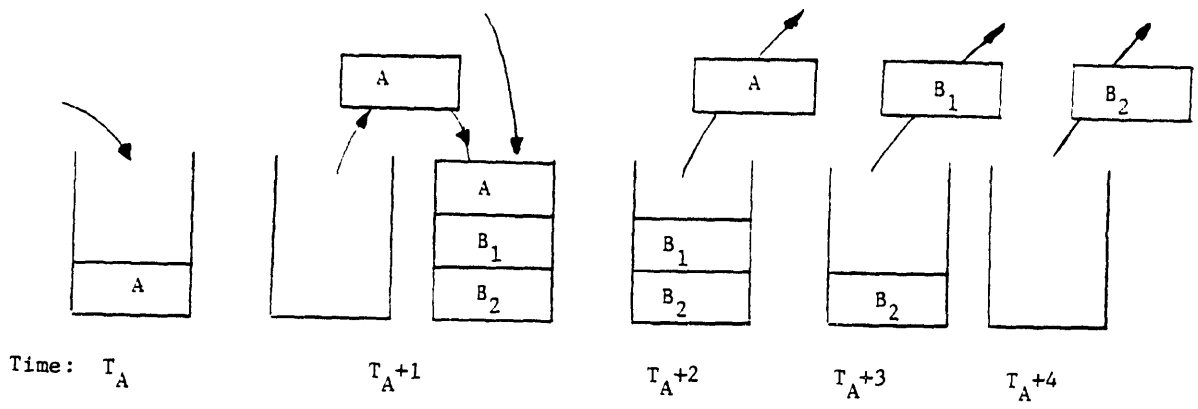
The main theme of this thesis is the minimization of overstowage within the different contexts in which it occurs. The example shown in Figure 1.2 demonstrates the main idea of the analysis of the following chapter. That is: "choose when and what items to rearrange in order to minimize the total number of item rearrangements over a certain period".

Time	Item for Storage	Retrieval Time
T_A	A	$T_A + 2$
$T_A + 1$	B_1	$T_A + 3$
$T_A + 1$	B_2	$T_A + 4$



(a) Total Number of Rearrangements = 2

[1] Of course, the retrieval times may be such that overstowage does not occur.



(b) Total Number of Rearrangements = 1

Figure 1.2 - Choice of Time and Items to Rearrange

It is worth mentioning in the above example it is not rational to place item B_2 above B_1. Were the latter done then the additional overstow that would be created would fall in the third case presented above, that is of bad planning. In terms of minimizing the total number of rearrangements of items it is clear that the choice is between preventively rearranging item A at T_A+1 and being forced to rearrange items B_1 and B_2 at T_A+2. If the objective is to minimize the total number of rearrangements from time T_A to T_A+4, then it is clear that the actions presented in 1.2(b) are the best.

Some definitions follow.

Definition 1.1 - An item of a stack is overstowed when it blocks the retrieval of another item scheduled to be retrieved while the former is still in the stack.

Definition 1.2 - Rearrangement (or rehandle) of an item is called the temporary removal from and placement back onto the stack of that item.

1.2 Overstowage in Containership Operations

It is apparent from Section 1.1 that a usual place in which overstowage situations arise is stacking of items in warehouses. However, this thesis has been motivated by a different application. The latter comes from the field of containership operations. A little background on containerships and containerized cargo is presented below.

Until the early 60's, ocean transport of commodities was done either by bulk (tankers or dry) or general cargo vessels. In those types of vessels the cargo is placed in holds or tanks. Shifting patterns of trade and higher productivity requirements in carrying cargo boosted the introduction of containerized cargo and containerships about 25 years ago, which indeed have revolutionized the shipping industry since then. Under the new idea, the cargo, and in particular shipments smaller than a ship load are placed in containers of the same size. This is done ashore. The containership carries the containers without even knowing what kind of cargo they contain. The advantages are obvious. Not only more efficient handling of the cargo is achieved, but also types of cargo that are totally different can be handled in a uniform way. In addition, shipments less than a ship load (in many cases less than a container load) can be accommodated. Furthermore since the filling of containers with cargo need not be performed necessarily at the port,

door-to-door service can be provided. The latter involves a chain of modes of transport that need to be integrated. Of course, only some types of cargo are appropriate for containerization. Liquid and dry bulk commodities (oil, grain, coal, iron ore, etc.) are still carried by tankers and dry bulk carriers and there is no sign of change in the near future.

The containers on board a containership are placed in stacks. As Figure 1.3 shows there are stacks below and above the deck of the vessel. Access to these stacks is possible only from the top of them. In fact access to stacks below the deck requires clearing the hatch (deck cover) that leads to the compartments below the deck (see Figure 1.3).

The mere existence of stacks on board creates "potential" for overstowage. The nature of containership operations contributes to the latter, as well. Containerships visit (call at) ports from which they pick up and to which they deliver containers. If the ports a vessel calls at are numbered as 0, 1, 2, .. M, M+1, in general, there are c_{ij} containers going from port i to j.

Such shipment requirements clearly fall in case (ii) of the previous section. Overstowage would be unavoidable, had only one stack to be used on board. The existence of more than one stack makes the situation less severe in terms of the number of overstowed containers. Nevertheless, independent of how many stacks exist on board, minimization of overstowage is a desirable outcome.

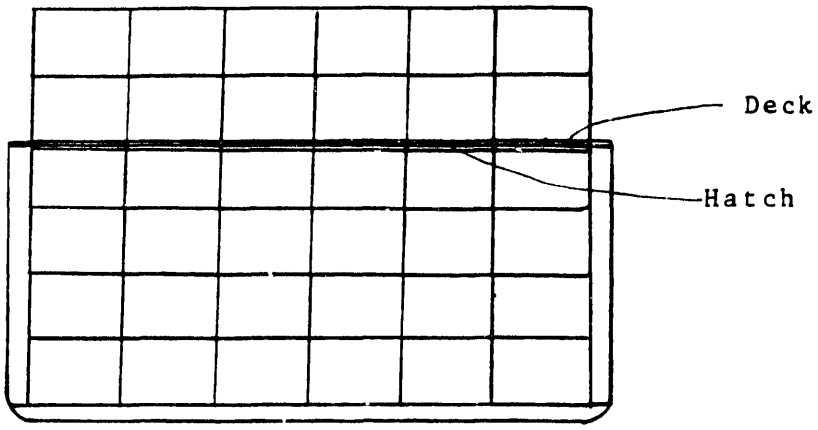


Figure 1.3 - Containership Section

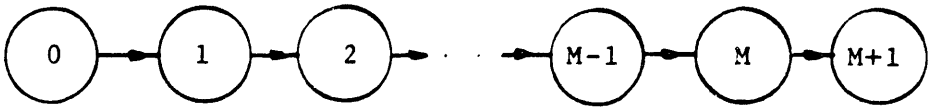


Figure 1.4 - Containership's Itinerary

Particularly in containership stacking operations, stowage of containers is not arbitrary. That is, there exists a set of rules, guidelines, and constraints which must be met. Such are:

i. Stability constraints due to stability requirements of the vessel. These constraints translate in a range (in three dimensions) within which the center of weight of the vessel and the containers must be.

ii. Strength requirements of the vessel's structure. This requirement stems from the limit on the bending and shear stresses that are developed along the vessel. This translates into a requirement for the weight distribution along the

vessel.

iii. Cargo placement constraints. For example, "refrigerated" containers, that is containers that can carry cargoes requiring low temperatures, must be placed to positions supplied with power outlets.

iv. Cargo adjacency constraints due to regulations about hazardous cargo. For example, reacting cargoes cannot be placed in adjacent positions.

v. Container strength constraints due to ability of the bottom container(s) of the stack to carry the weight of the containers above it (them).

vi. Container stability constraints due to the required lashing of the containers above the deck.

The above list is by no means exhaustive. However, it indicates the complexity of the problem, if all constraints are to be considered explicitly. In fact, the containership case is the most intricate among the occasions where overstowage occurs in most other cases, very few or none of the above constraints apply.

1.3 Other Occasions Where Overstowage Occurs

Containership operations is not the only area where overstowage arises. Overstowage is present in all circumstances involving stacking operations. Moreover the word "stacking" should not be interpreted in the strict sense it was used in the previous section but in a much more general sense. This will become apparent later on in this section.

Stacking operations in warehouses is one area in which the phenomenon of overstowage is very common. The operation of warehouses involves the arrival of items for storage and their subsequent retrieval later on. Suppose that the next day's shipments are retrieved overnight and are placed in a "buffer" area awaiting pickup. Similarly, assume that the items that arrive during the day are stored in a buffer area too. The latter are placed in the warehouse main storage area during night time. This kind of operation resembles the one in which a vessel visiting a series of ports. Imagine that the vessel being the warehouse which "travels" in time, and visits a series of "days" (ports). Fortunately, many of the constraints described above do not apply.

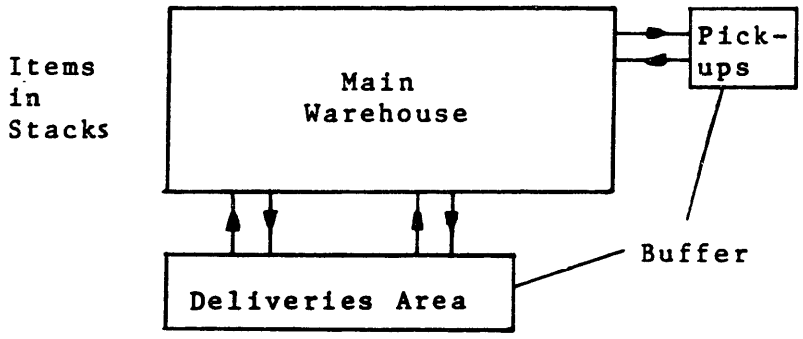


Figure 1.5 - A Model of a Warehouse

A very related situation arises in container terminals in ports. In many ports[2] the containers that are gong to be shipped are stowed in stacks next to the berth,

[2] In other ports, the containers are brought at berth for loading by chassis. That situation may result in another form of overstowage described later on.

awaiting the containerships to pick them up. The containers are stowed as they arrive at the port for shipment. This is clearly a warehouse-type operation and overstowage may occur. It is in the interest of smooth port operations to minimize it. In fact, one may try to build a more sophisticated system which takes into account the overstowage cost incurred to both the vessel(s) and the port or ports of a region in an attempt to minimize the time that vessels spend at port.

A case similar to a containership visiting a series of ports comes up when a truck visits a series of locations where it loads and/or unloads boxes stored in stacks. Of course, the scale of the problem is smaller (fewer stacks) and, again, most of the previous constraints do not apply.

The concept of stacking does not require the stack to be physically vertical. Any linear arrangement with only one access point may serve as a stack. A typical example is parking garage operations as described in Figure 1.6. In this case the stacks are simply horizontal. The same situation may arise in container terminals that use chassis to move containers around.

A somewhat different view of overstowage appears in the following situation. Suppose that there exists a number of tools, say M. These tools are stored in a stack. Every time a tool needs to be used and retrieved, all tools that block it (i.e. are overstowed) need to be removed. After being used, the tool is restacked. Removing a tool costs a certain amount of time and money. If the frequency by which each tool is used is known, a natural question is what is the order of items in the stack that minimizes the expected cost (time or money). In another version,

the sequence by which the tools are used is assumed known. Again, the question is what is the order that minimizes the rearrangement cost? These problems define the class of "use-and-restack" problems. This is to be contrasted with the type of problems introduced earlier (containership, warehouse) which define the class of "pickup-and-deliver" problems since items come on a stack for a certain period of time, and then, go off forever.

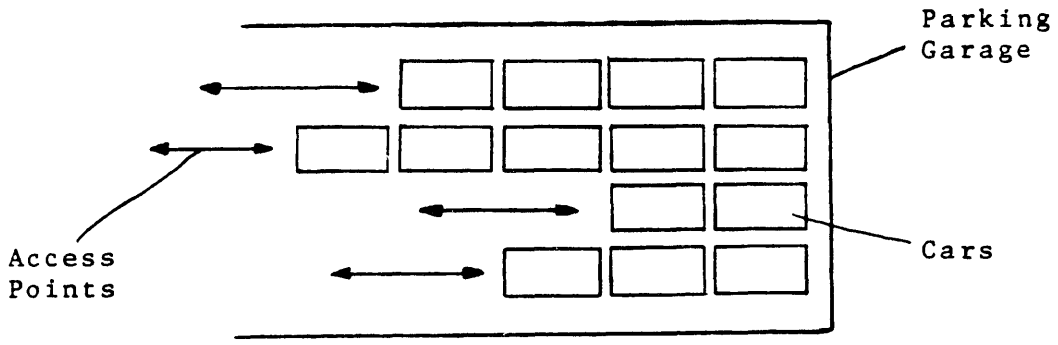


Figure 1.6 - Parking Garages: An Example of Horizontal Stacks

There are many variations of overstowage problems that can be constructed. All of them are of a combinatorial nature. Consequently, their solution requires techniques from the field of combinatorial optimization. Undoubtedly, this class of problems is very interesting from both the theoretical and practical viewpoint. It appears that overstowage problems are hard in the general case. Special or restricted versions of them can be solved efficiently, though.

1.4 Thesis Theme

The thesis is focused on developing analytical methods (algorithms) for solving overstowage problems. The motivation comes from containership operations which is the most difficult version of overstowage. The objective, at least in that field, is to reduce the time at port per vessel visit. With modern containerships carrying several thousands of containers (to the range of 5,000), there are enough margins for improvement. The time it takes to (un)load a container is usually two to four minutes. Then, if 50-100 rehandles at each port can be saved, the amount of time saved in one year is approximately three days:

$$(50\text{-}100)\ \text{containers} \cdot \frac{(2\text{-}4)}{60.24}\ \frac{\text{days}}{\text{containers}} \cdot 20\ \frac{\text{port calls}}{\text{year}} \simeq \text{days/year}$$

that is about 1% increase in the productivity of the vessel (measured in container-miles per year).

A similar calculation can be carried out from the point of view of the port. The same savings in terms of container rearrangements correspond to a greater percentage of port time. If 50 rehandles are saved per ship call, and if the latter involves the pick up and delivery of 1000 to 2000 containers, then the savings may go up to 5% of the port time (=50/1000). Generally, port time goes down proportionally to the number of container rehandles saved.

Solving the problem with analytical methods requires first to have a way of describing the allocation of items to stacks over the horizon under consideration. The most explicit way of doing so is to individually identify each item and then

define variables as x_{ijk} as

$$x_{ijk} = \begin{cases} 1, \text{ if item i is in position j at time k} \\ \\ 0, \text{ otherwise} \end{cases}$$

It does not take much analysis to see that the above approach leads to an enormous number of binary variables. This makes inefficient any computational effort to solve the problem, although it is trivial to express the constraint and other feasibility conditions that arise.

A more promising way to describe the allocation of containers to stacks is to describe the list of items in each stack at times of interest. Items can still be individually identified. The problem can be expressed as a series of assignment problems of items to stack position at the time of interest. The overstowage cost, though, can not be expressed in closed form. Of course, if the assignments of items to stack positions over the horizon are given (known), then the overstowage cost can be easily computed. This is done by simply counting the item rearrangement required for each stack to "move" from the profile it has at one time moment to the profile it assumes at the next one. This approach requires as many pieces of information as the number of available positions multiplied by the number of time moments that deliveries or pick-ups occur. This is almost the minimum amount of information that can be used.

In spite of the short description of the state of the stacks, solving the overstowage problem in the presence of placement constraints (as in containership

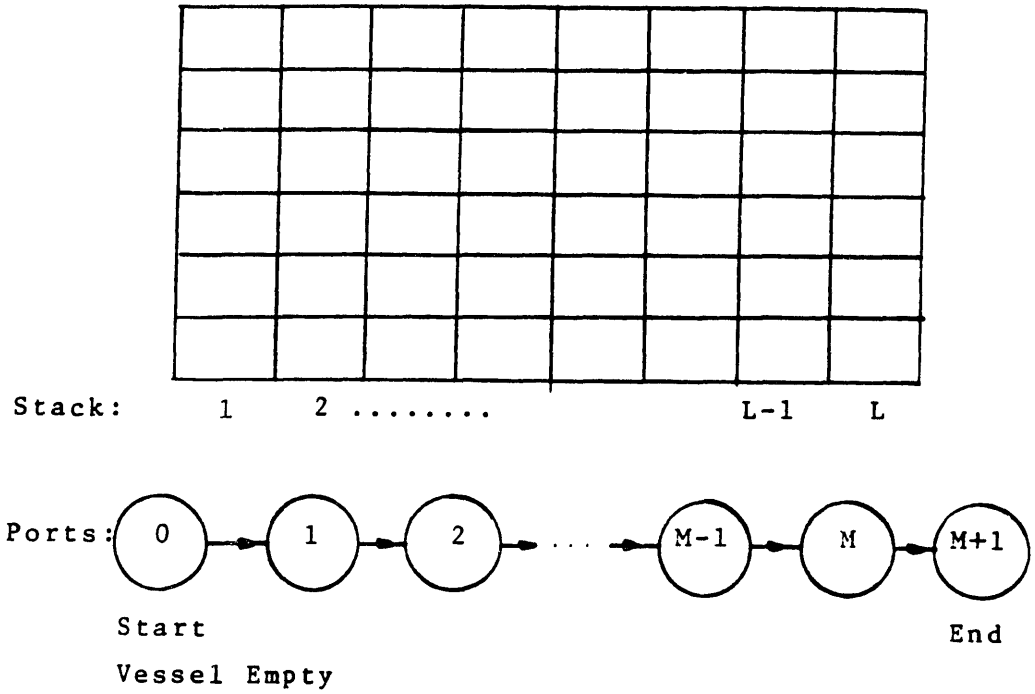
operations) appears to be a hard task. The objective of this thesis is to study the phenomenon of overstowage in stacking operations. The presence of side constraints may obscure aspects of the pure overstowage problem because of the added difficulty. As the literature survey of the following chapter reveals, very little research has been done on the problem of minimizing overstowage costs. Consequently it seems natural to solve first the unconstrained problem except for stack capacity constraint.

Also, this thesis ignores unusual geometric configurations of stacks as that shown in Figure 1.3 with stacks above and below the deck of a containership. Another bold assumption is that the shipments (item movements to and from the stack) are deterministic and known.

Both of the above simplifications are relaxed in the one-stack case. In particular, the effects of probabilistic shipments and of the existence of stability constraints are studied.

Figure 1.7 shows graphically the general version of the problem this thesis attempts to solve. The maritime terminology is used. A final comment relates to the capacity constraint.

Although it is assumed that each stack may have finite capacity, the sum of the capacities of all the stacks is assumed to be always sufficient to accommodate all the items that need storage. That is no selection of items to be stacked (based possibly on a revenue maximization criterion) is necessary.



Shipments: c_{ij}, i=0,1,...M; j=i+1,...M+1 - (known)

Stack capacities: b_i, $1 \leq b_i \leq \infty$, i=1,...2
(usually $b_i = b$, i = 1,...L)

Objective: minimize overstowage

Figure 1.7

Presentation of the Overstowage Problem Considered in This Thesis

1.5 Thesis Outline

This section gives an overview of this dissertation. Numbering of figures and relationships is done separately for each chapter. Numbers in brackets refer to the

references at the end. Superscripts refer to footnotes at the bottom of the page. The content of each chapter is as follows.

Chapter 2 is devoted to a survey of the existing literature. Unfortunately, not much published work is available. There is almost no publications from the warehousing field on this topic. There exist several from the maritime community but they do not deal directly with overstowage. Only one paper was found dealing explicitly with overstowage problems. It does not present any algorithm or provide any analysis, though, other than simply recognizing the existence of the problem.

Chapter 3 deals with the one-stack overstowage problem (OSOP). A vessel visits a series of ports but she is allowed to carry containers only in one stack. This problem is solved in polynomic time in the number of ports in the series. Chapter 4 proceeds to sensitivity analysis of the algorithm developed in Chapter 3 and in other related issues, the most important being a transformation of the shipment schedule (matrix) to one that bears zero overstowage.

Chapter 5 looks at extensions of the OSOP algorithm. Some assumptions are relaxed and most importantly probabilistic shipment matrices are examined. It is proven that the recursive deterministic algorithm is still valid under a probabilistic shipment matrix.

In Chapter 6 stability constraints are introduced in the one-stack overstowage problem. This takes place in two stages: (i) the one destination problem, and (ii) the multi-destination problem. Both problems are solved by polynomial time recursive algorithms.

In Chapter 7 the multi-stack overstowage problem is examined. A series of formulations are presented along with a classification of the problem according to the presence or not of some constraints. Properties of the optimal solution are discussed. Also, the effect of the number of stacks on the overstowage cost is analyzed. The chapter includes a complexity analysis section, but an NP-completeness proof for the MSOP has not been derived. The chapter ends with a model for container assignment to stacks given the rearrangement policies of each stack.

Chapter 8 turns to heuristic algorithms for the MSOP. Certain heuristic approaches are examined and analyzed. Chapter 9 concentrates on heuristics for the containership operations case. A simple heuristic, along with a set of empirical rules for the containers to stacks are proposed.

In Chapter 10, the static overstowage problem is introduced and discussed. Several versions are examined and some are solved by polynomial algorithms.

Chapter 11 reviews the contribution of the thesis and looks to the future. Generalizations of the "stack" concept are suggested and directions for further research are provided. Chapter 11 also looks at containership design issues as they related to container stowage. Port-ship integration concepts are also discussed.

Finally, Chapter 12 contains the references. Appendices A1 and A2 provide some technical support for results in the text and contain the code of the OSOP algorithm.

CHAPTER 2

OVERSTOWAGE PROBLEMS: A LITERATURE SURVEY

This chapter contains a survey of the existing literature on the overstowage problem. It appears that very little has been published on that field. This is surprising, especially because the literature on warehouses and warehouse operations is enormous. The most serious attempts to solve the problem come from the maritime field, but very few are published because such knowledge is proprietary. The latter indicates the importance of the problem to the shipping community. In the following, different approaches to the problem are discussed. However, almost none of those addresses the problem from the viewpoint this thesis intends to do.

2.1 Literature for the Containership Case

The first researchers who dealt with the problem are W. C. Webster and P. Van Dyke, of Hydronautics Inc. [17, 18]. Their work was presented at the Computer-Aided Ship Design Engineering Summer Conference at the University of Michigan in 1970. Their approach is aimed at the loading/unloading process, but it can, as they claim, be extended to include the allocation and storage of containers on the dock. The stability constraints of the vessel are among the primary factors considered. It is also realized how much flexibility one has in allocating the cargo on board because of its unitized form.

The goal is to optimize the operation of the ship and container system. To achieve this the overall system is divided into two components - the "ship" and the "containers". Optimum for the ship consists of: (a) having desirable longitudinal, vertical, and transverse centers of gravity to satisfy stability or trim constraints; (b) carrying the minimum ballast to satisfy the above constraints; (c) utilization of consumables while underway to maintain this desirable condition. Optimum for containers implies: (a) the minimization of handling of any containers between its origin and destination (overstowage); (b) minimum total container handling time by utilization of combined load/unload operations; and, (c) maximum utilization of available cargo space within the limitations of container placement restrictions.

A finite, and periodic, horizon is assumed and optimization is assumed to take place over the entire horizon. Figure 2.1 presents a flow chart behind the logic of such a system. Figure 2.2 graphically shows some of the stability constraints that need to be observed.

As far as overstowage is concerned, containers are kept homogeneous with regard to destination port within cells (position in a transverse group of stack and for the entire group, both below and above deck). It is said in this paper that when mixing is required, containers for further ports are loaded below those for nearer ports in the horizon. It is also claimed in this paper that preliminary results indicated a very shallow optimum curve, with numerous solutions close to the optimum. Because of that the paper follows a trial and error strategy rather than an optimal solution procedure. This strategy is divided into three stages: first, a

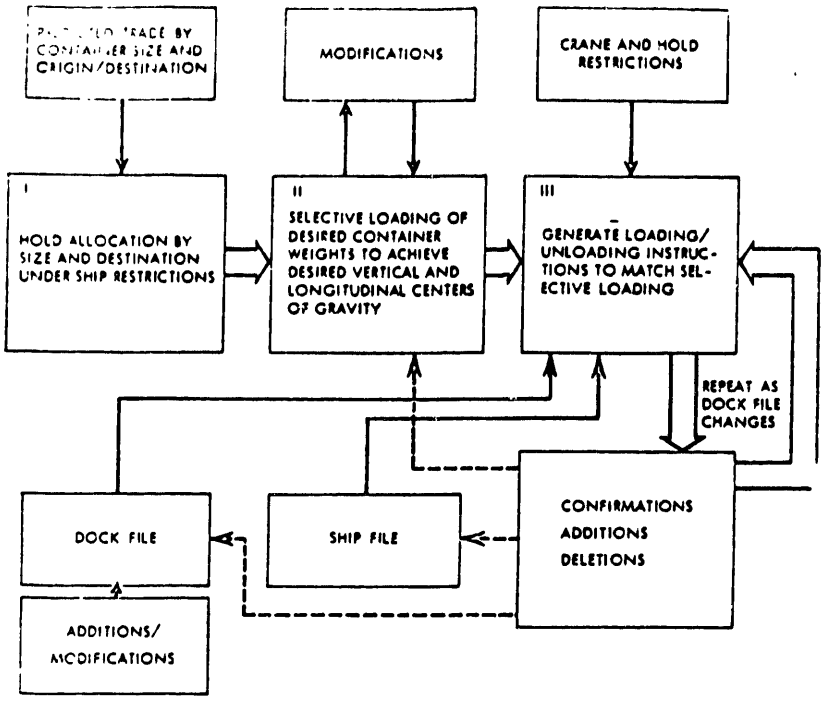
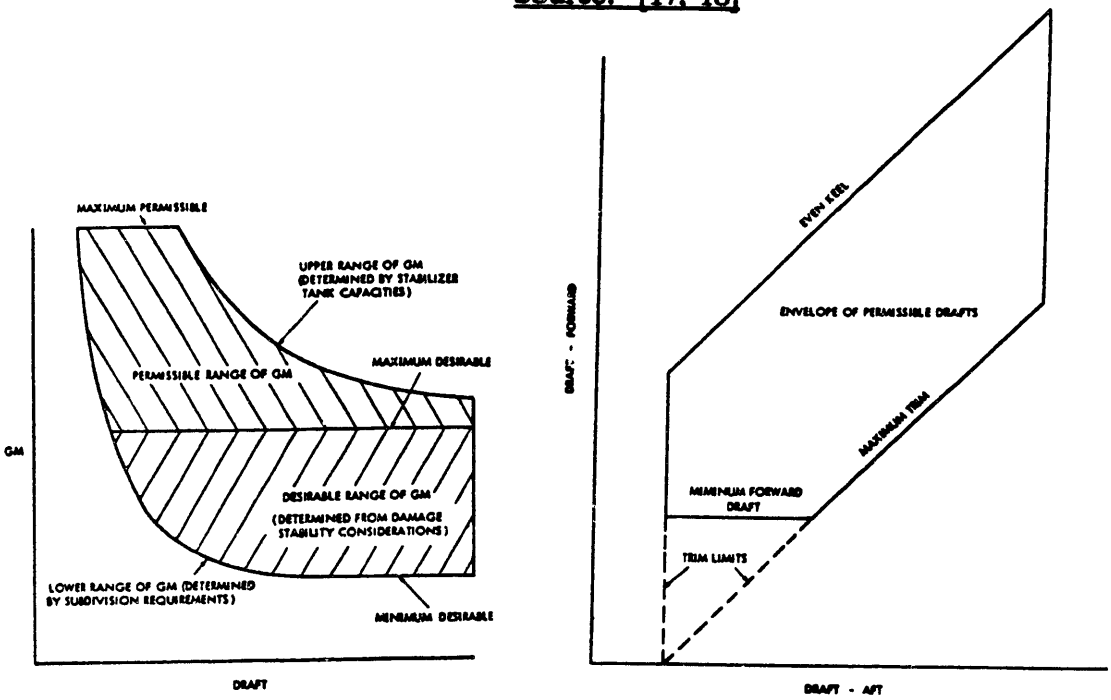


Figure 2.1 - Strategy Flow for Loading/Unloading Instructions
Source: [17, 18]



(a) Metacentric Height Limitations (b) Envelope of Permissible Drafts
Figure 2.2 - Envelope of Permissible Drafts
Source: [17, 18]

general determination of the entire horizon desired characteristics and assignment of containers by destination to the holds for the entire horizon; second, the generation of a selective loading for the next leg of the vessel's itinerary; and third, the determination of the loading and unloading sequence for the next voyage. The procedure is shown in Figure 2.3.

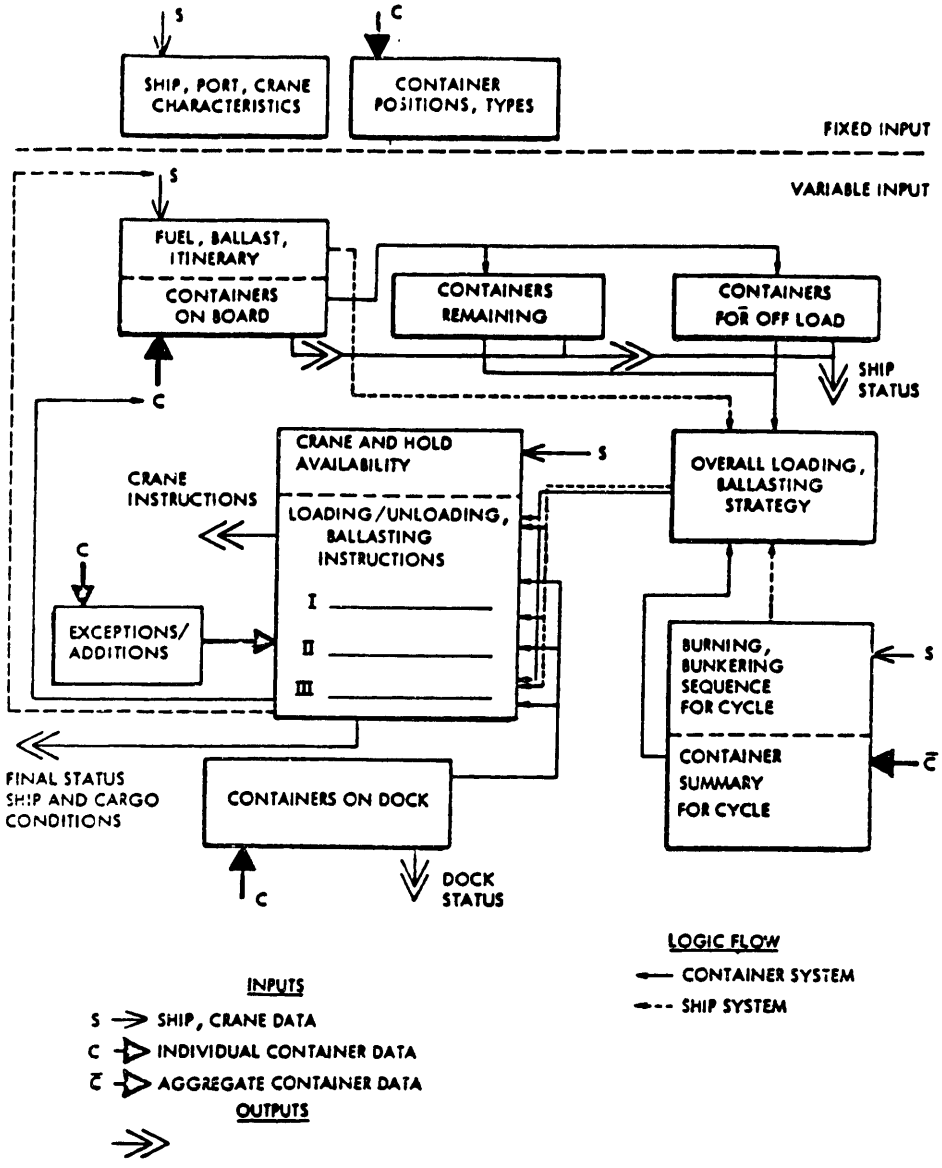


Figure 2.3 - Logic Flow with Inputs

Source: [17, 18]

This is a purely heuristic approach entirely derived from everyday experience that might work well in some cases. In fact, at the time the paper was presented, the method was not fully tested. As it is stated in the paper, the principal value of the procedure lies in the formulation and structuring of the system itself and not in the final implementation which, according to the paper, can be easily customized. The authors report that a small number of trials showed good results ("qualitatively good results") as far as the water ballast and overstowage cost are concerned.

Undoubtedly, this paper is a milestone in analyzing containership operations. The treatment of overstowage, though, is based only on empirical or intuitive "rules of thumb" and it appears to be inadequate. Algorithmically speaking, not much criticism can be applied to a purely practical approach.

Scott and Chen [15] attempted another heuristic approach to the same problem. They adopted three heuristic rules which were used to implicitly satisfy the constraints as follows.

Containers are aggregated into homogeneous groups based on some container characteristics (such as type, length, height, weight, racking strength, and destination) and also on placement restrictions. Ten classes of containers are created. The first nine classes include containers with specific requirements, while the tenth class contains containers suitable to be placed anywhere on the vessel. The containers of the tenth class are stratified into several weight brackets. Dynamic programming is used to determine the ranges of the brackets. The allocation procedure has four stages:

1. The containers of the first nine classes are assigned to individual positions by following three heuristic rules.

2. The containers of the tenth class are distributed to transverse groups of stacks by using an integer planning model. The objective in this stage is to maximize the number of containers to be loaded.

3. The allocation of individual containers within each group of stacks is determined by using integer programming models. The objective is to minimize the transverse moment.

4. The trim, transverse moment and metacentric height (stability) constraints are checked. If violated, container reassignments take place until all constraints are satisfied. If the latter is not possible, the number of containers on board is reduced by one and the process is repeated.

Among the advantages of the above heuristic is the consideration of as many possible constraints as possible. But it does not deal directly with overstowage. In addition, the method uses integer programming models which cannot be solved very efficiently even for small model sizes. A typical integer program is reported in the paper as having 29 constraints and 84 integer (binary) variables. As it is known, solving problems of the above size (in fact, repeatedly) is not a fast process. Also the paper does not explicitly refer to a port sequence but rather to a single port.

In a study sponsored by American President Lines, Shields [14] followed a different path to solving the container stowage problem. The basic idea in this approach was the random generation and evaluation of many different possible

loadings. In order to avoid a large number of loadings, many of which are presumably not optimal, the loading generation process was biased in a manner to produce good results.

The criterion through which a specific loading was generated was selected randomly among a set of criteria. More specifically, each criterion was assigned a weight and a random number generator selected the hierarchy under which the criterion would be applied for the loading of the next group of containers.

The evaluation of the loading was done by imposing penalties each time an increase in the container handling cost occurred or each time a constraint was violated. In the latter case, the related penalty was very large to preclude the selection of that loading as optimal.

Finally, the three top (less costly) solutions were approved. The algorithm commenced the loading of containers of the next port using the three selected loadings as starting points. Again, the three best loadings were chosen. The procedure was repeated at each subsequent port. At the final port, the less costly solution, as well as the intermediate loadings which result, were found and adopted. A diagrammatic representation of the algorithm is shown in Figure 2.4. The shaded ships represent the selected loadings. The doted line indicates the final loading chosen.

This algorithm took into account many parameters and restrictions and satisfied them fairly well. But it did not guarantee optimality even in a relative sense, since there was nothing to secure that the best combinations after, say port #1, would

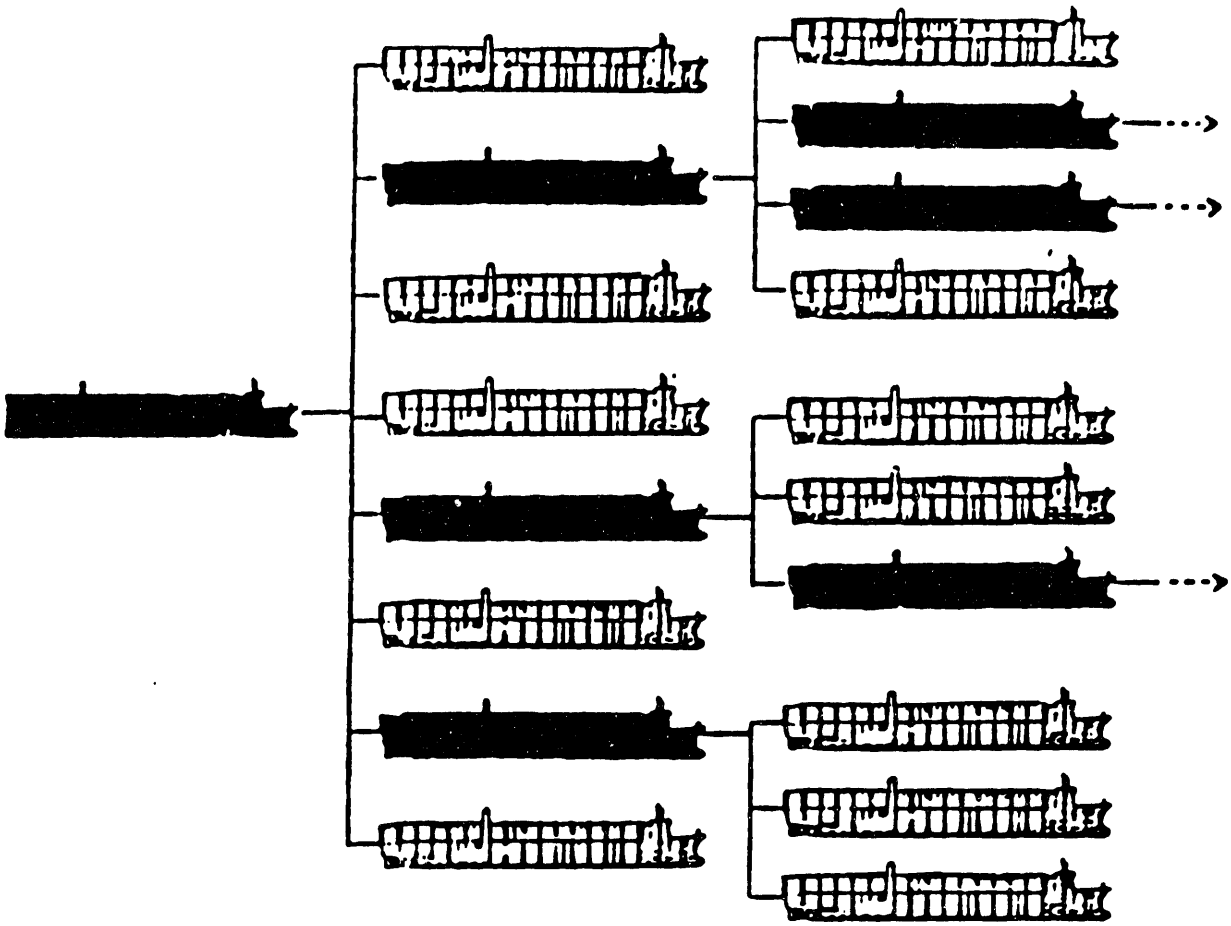


Figure 2.4

Algorithm Proposed by J. Shields: Selection of the Three Top Loadings to Continue the Rest of the Route

Source: [14]

result in an overall optimum at the final port. Moreover, the three selected solutions at each port were not necessarily the best. All the above are true in a probabilistic sense. Of course, if the number of combinations checked as well as the number of loadings approved to continue to the next ports are increased, the probability to find a better solution increases. But this also results in longer computer time.

Finally, Aslidis [2], in his MS thesis, examined a simplified version of containership operations. It was assumed that a vessel visits a series of ports, of which she only picks up containers (one destination problem). Overstowage is not a factor in this case except some constraints are violated, in which case already stored containers need to be rearranged. A heuristic approach is proposed aimed mainly at satisfying the trim and metacentric height requirements with the minimum rearrangement costs.

2.2 Other Literature on Overstowage and Rearrangement Problems

S. P. Ladany and A. Mehrez [11] considered a form of the Traveling Salesman Problem, in which overstowage costs were also included. This has as follows: Suppose that a truck is going to carry boxes from shippers in area A to customers in area B, quite apart from A as shown in Figure 2.5. The truck can carry boxes only in one stack. Suppose also that the total time of the operation is to be minimized. This consists of the traveling time between the pickup and delivery points and of the time spent at each of the above locations.

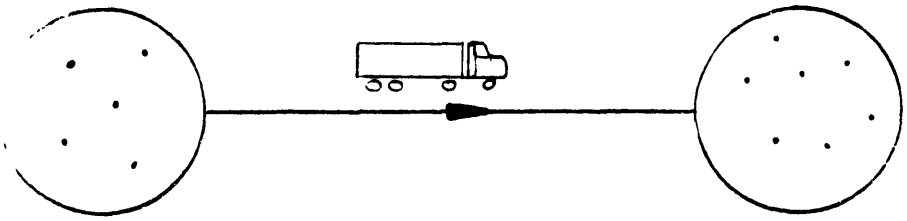


Figure 2.5 - TSP with Overstowage Costs

Source: [11]

The latter is a function of the number of boxes that are unloaded from or loaded onto the truck. Obviously the smaller the number of rehandled boxes at each location the smaller the total operation time. Given the sequence by which the locations are visited, a minimum rearrangement plan may be derived. If the sequence of visits is that specified and is to be decided, then there are two traveling salesman problems to be solved. However, the two TSPs are linked through the requirement of minimization of the rearrangement time. This is generalization of the TSP problem and consequently it is very difficult. In fact, it can be easily proven that the problem belongs to the class of NP-complete problems.

Landany and Mehrez simply recognize the existence of the problem. They also notice the little attention that overstowage problems have received.

N. Christofides and I. Coloff [4] have published a paper concerned with finding the optimal way of rearranging items in a warehouse from their initial positions to their desired final location. Such rearrangements may become necessary because of

changes in the relative demand for each item, with the result that what were once "fast-moving" items at the "front" end of the warehouse are now only slow-moving ones that must be moved towards the rear. The paper gives a two-stage algorithm that produces the sequence of item movements necessary to achieve the desired rearrangement and incur the minimum cost (or time) spent in the rearranging process. This algorithm is optimal in the restricted case where the rearrangements must be done in a number of cycles, each one being of short duration.

In the above algorithm, the item movements is part of the input. These movements could be the solution to an overstowage problem as defined in the previous chapter. That is, the algorithm could serve (after the appropriate modifications) as a follow-up to an overstowage minimization one, and implement the suggested movements at a minimum cost.

CHAPTER 3

THE ONE-STACK OVERSTOWAGE PROBLEM (OSOP)

In this chapter we examine and solve the simplest case of overstowage. This is when all the containers are placed onto a single stack and no other constraints apply. The case represents a pure overstowage problem. This situation is to be contrasted in subsequent chapters with situations in which the overstowage cost (i.e. additional rehandle) also depends on the assignment of containers to stacks. The simplified case treated in this chapter allows for a deep understanding of the problem. An algorithm is developed and several alternative formulations are presented. The algorithm runs in polynomial time ($O(M^3)$, M is the number of ports to be visited), and it will be extensively used as part of the multi-stack case algorithms.

3.1 OSOP: Definitions and Assumptions

Overstowage is a situation arising in all kinds of stacking operations and it is certainly not restricted in applications in the maritime field. So, despite the fact that the terminology to be used emanates from maritime applications, it should be interpreted as representative only; exactly the same concepts apply in all applications of stack management.

Stacks can be defined as one-dimensional storage systems with one access point (see Figure 3.1). Items (boxes, containers, etc.) are stowed one on top of the other. Stowage can be vertical as in Figure 3.1 or horizontal as in Figure 3.2.

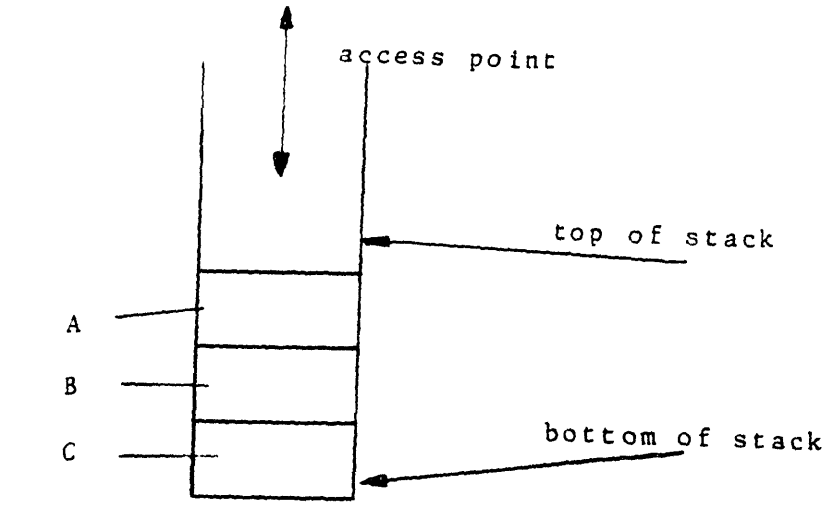


Figure 3.1 - Stacks

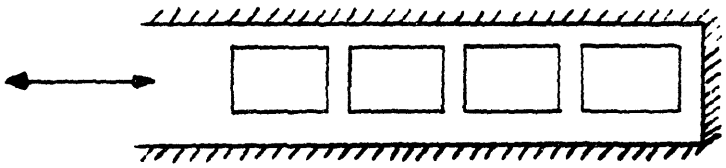


Figure 3.2 - Horizontal Stacks

In this chapter, we employ uncapacitated stacks, that is, stacks with no limit on the items stowed. We call top of the stack the end of it closer to the access point and bottom of the stack the least accessible end (see Figure 3.1). The rule - last item in, first item out - is always valid in stacking operations.

Let us now define the problem we solve in this chapter. Speaking in maritime terms, we assume that a vessel (containership) is scheduled to visit a series of ports 0,1,2,...,M-1, M, M+1 (see Figure 3.3).

She is a special type of ship, though, since she can only carry one stack of containers, all of which are assumed to be of equal size. It is assumed that the

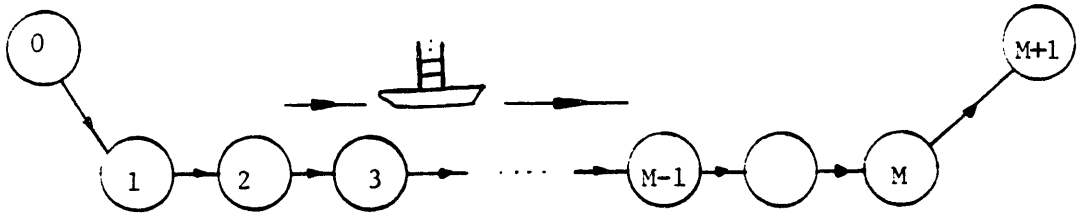


Figure 3.3 - Port Sequence

vessel arrives at port 0 empty. At port 0, and at all subsequent ports up to port M, she picks up containers shipped to the subsequent ports of the series. So, at port 0, she picks up containers going to ports 1, 2,...,M and M+1. If denotes the number of containers going from port i to port j then the shipment requirements can be represented with the following lower triangular matrix, called the shipment matrix.

$$C = \begin{bmatrix} C_{0,1} & & & & \\ C_{0,2} & C_{1,2} & & & \\ C_{0,3} & C_{1,3} & C_{2,3} & & \\ C_{0,M-1} & C_{1,M-1} & C_{2,M-1} \cdots C_{M-2,M-1} & & \\ C_{0,M} & C_{1,M} & C_{2,M+1} \cdots C_{M-2,M} & C_{M-1,M} & \\ C_{0,M+1} & C_{1,M+1} & C_{2,M+1} \cdots C_{M-2,M+1} & C_{M-1,M+1} & C_{M,M+1} \end{bmatrix}$$

Containers with the same origin and destination belong to the same group of containers. There are N=(M+1)(M+2)/2 groups. Containers with the same destination belong to the same type of containers. There are (M+1) types (1,2,...M,M+1). Let d(x) denote the type or destination of container x.

Definition 3.1

Containers with the same origin and destination define a "group" of containers (i.e. group (i,j)). Containers with the same destination are of the same "type". ■

The assumption about the condition of the vessel as she arrives at port 0 - namely, that the vessel/stack is empty - is to be called initial condition assumption. As it will become evident in the next section, the above assumption results in a stack in "in-order" condition as the vessel arrives at port 1.

Definition 3.2

A stack is in "in-order" condition if the containers of the stack are placed in ascending order of destination from top to bottom. ■

Definition 3.3

A stack is in "out-of-order" condition when it is not "in-order". ■

The initial condition assumption of the stack is more formally expressed as:

Assumption 3.1 The stack is "in-order" as the vessel arrives at port 1. ∎

Assumption 3.1 is not critical and will be relaxed in Chapter 5 where extensions of the algorithm of this chapter are discussed. Since it simplifies the analysis of the problem, we will postpone its relaxation for later.

Figure 3.3 along with the shipment matrix C (3.1) and the assumption 3.1 constitute the input for the version of the overstowage problem to be solved in this chapter.

There are two types of stack operations that are performed at every port. First, delivery of containers with destination at the current port, and secondly, placement on board onto the stack of the containers to be shipped out of the current port. Along with the above, there exists some containers that find themselves are quay, although neither are destined to nor originating from the current port. These are containers blocking the delivery of those to be delivered (see Figure 3.4) and consequently had to be taken off the stack temporarily.

But these "rehandled" containers need to be placed back onto the stack along with the "new" ones. In fact, there is no reason to treat them differently from the latter.

Our objective is to minimize the number of rehandled containers during the trip of the vessel. To that extent, we allow ourselves to take initiative and rearrange some containers beyond those required to clear the way for the "for-

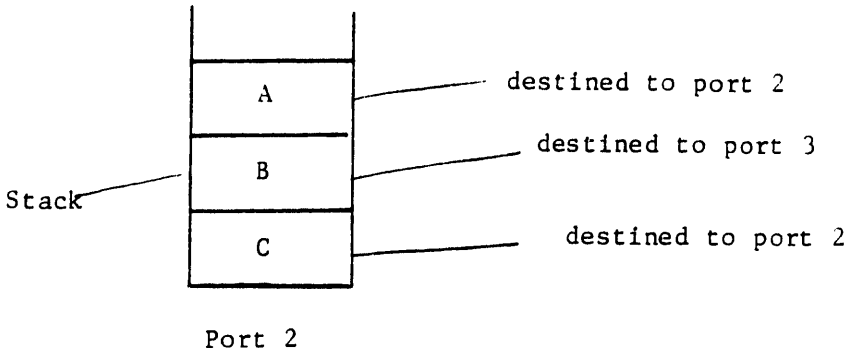


Figure 3.4

Overstowage: Container B is Blocking The Delivery of Container C at Port 2

delivery" ones now, in order to save a larger number of rehandles in later ports. That is, we look for a rearrangement policy (P) which results in the minimum number of additional rehandles (including voluntary ones) of containers.

The above-mentioned rearrangement policy should provide information about what containers of the stack should be rearranged and how they should be placed back onto the stack at every port.

Let $R_P(C)$ be the number of rearrangements that result if we apply policy P on the shipment matrix C, and let R(C) correspond to the optimal policy P*(R(C) = $R_{P^*}(C)$). We will also refer to $R_P(C)$ as the overstowage cost.

3.2 Preliminary Analysis

Although we do not have a measure of "disorder" for a given arrangement of the containers of a stack, we can easily compare two arrangements of the same containers. Suppose that the vessel has called at port i and already delivered the containers destined to that port. Figure 3.5 (a) shows an arrangement in which at least one container is overstowed. Figure 3.5 (b) shows an arrangement which is the same except that container B is not overstowed relative to A. It is true that arrangement (b) is no worse than arrangement (a). To see that, let us take an optimal policy for (a) and apply it to (b). A rearrangement policy for (a) will not be

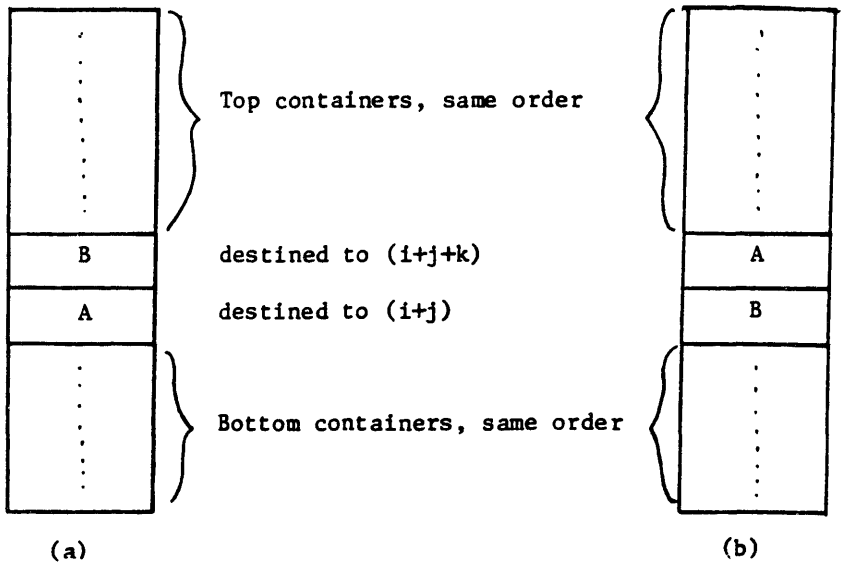


Figure 3.5 - Relative Stack Arrangements

optimal if it asks for the rearrangement of B at any port before (i+j) and not for the rearrangement of A. The term rearrangement is used for both, voluntary and forced rearrangements. The latter is true because since there are no containers going to ports (i+1), (i+2),...,(i+j-1), below A (otherwise A would have also been

rearranged), no such container is blocked by B, too. So, any policy requiring rearrangement of B only at ports (i+1) to (i+j-1) is dominated by one which simply does not rearrange B and is the same everywhere else. But then arrangement (b) does not result in more rearrangements under the same policy, since A and B together get or do not get rearranged at any port between (i+1) and (i+j-1) included. Consequently, the optimal policy for arrangement (b) results in no more rearrangements.

In fact, arrangements (a) and (b) of Figure 3.5 can be thought as initial stack conditions for a vessel starting at port (i+1) and going to (M+1). Then we can prove the following lemma.

Lemma 3.1

Any initial condition (arrangement) with a subset of consecutive containers "in-order" results in no more rearrangements than the same initial condition except from the fact that the previously "in-order" subset of containers is "out-of-order" in the same consecutive positions of the stack, for any port series length M and any shipment matrix $C_{(M+1)x(M+1)}$.

Proof

The proof is based on repeated application of the discussion of the previous page.

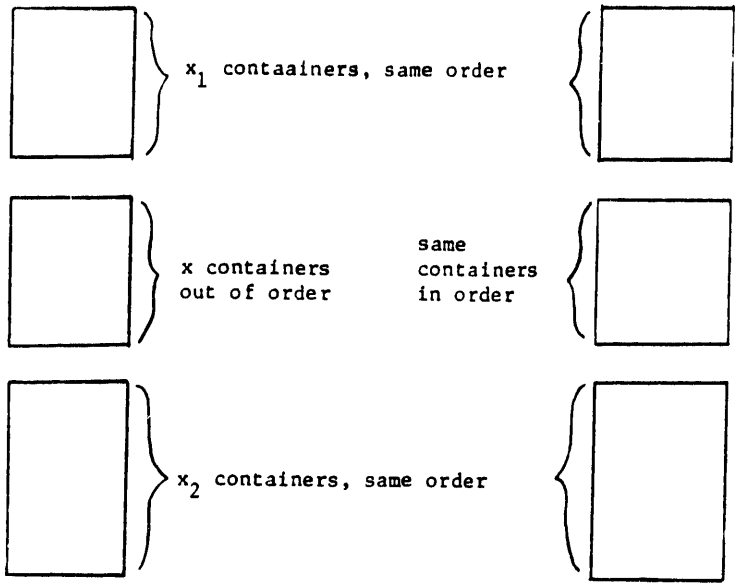


Figure 3.6 - Relative Initial Stack Conditions

The middle x containers can be brought in-order in at most $O(x^2)$ switches of adjacent containers. Each switch results in no increase of the total number of rearrangements, as it has been proven before. ∎

Lemma 3.1 immediately solves part of the problem, namely it answers the question of how to place the containers onto the stack after the containers to be rearranged have been taken off. It is clear from Lemma 3.1 that these containers should be placed in descending order of destination: those going to the further ports should be placed lower onto the stack.

Finally, it must be noted that it is the above lemma that makes assumption 3.1 equivalent to assuming that the stack is empty at port 0.

3.3 Properties of Rearrangement Policies

The way rearrangement policies have been introduced is very abstract. In fact it has not been defined how these policies are expressed. Of course, one can identify each container individually - a total of $\sum_{i=1}^{M} \sum_{j=i+1}^{M+1} c_{ij}$ - and give for each port the profile of the stack along with an indication of whether a container gets rearranged or not. Although detailed, the latter is a wasteful approach. As we will see in this section, the optimal rearrangement policy(s) has a number of properties that make it simply to express. We will exploit these properties later on when we develop an algorithm to solve the OSOP.

At first let us look at containers with common origin (say, i) and destination (say, j); there are c_{ij} containers of this sort. Assume a rearrangement policy that treats each one of them differently. Let t_k, $k=1,2,...,c_{ij}$, be the number of times each one of them gets rearranged, and let $t_{k^*} = \min(t_k)$. Then every container, e, with $t_e > t_{k^*}$ can be "attached" to container k^*, that is it can always be placed next (above or below) to k^* and follow the "fate" of k^*. Since both containers initiate at the same port (i), that is both are off-board at port i, there is no constraint in doing so. The result is a net saving of $t_e - t_{k^*}$. The initial policy remains unchanged except for the fact that container e mimics (follows) the same policy as k^*. But there is no obstacle for any other container with origin i and destination j to do the same. Since k^* gets rearranged the minimum number of times (t_{k^*}) among the containers with origin i and destination j only decreases in the total number of rearrangements can be recorded. So, all the containers of origin i and

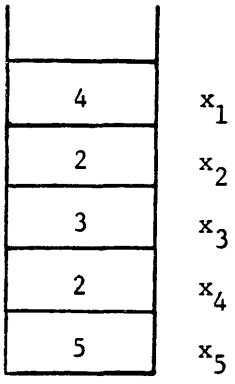
destination j should be grouped together and experience the same treatment in terms of rearrangement policy, otherwise we can always decrease(or not increase) the number of rearrangements by grouping them. Finally, it must be mentioned that there is nothing special about origin i and destination j. That is, the above property holds for all origin-destination pairs, and the following lemma has been proven true.

Lemma 3.2

An optimal rearrangement policy treats containers of the same group the same. ■

An immediate consequence of Lemma 3.2 is that containers of the same group move together and are always placed on adjacent positions of the stack. In fact, we could replace them with a single container with a weighting factor c_{ij}.

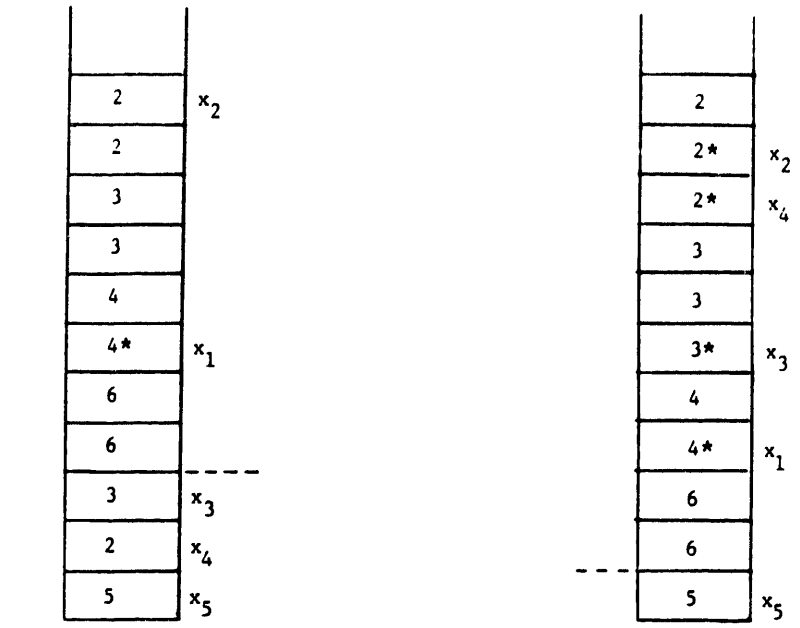
Let us now assume that the stack contains n containers, $(x_1,...x_n)$ while the vessel is at port i and has already delivered the containers of type i. The situation is depicted in the example of Figure 3.7. It is assumed that i=1 and n=5. The numbers indicate the type of each container.



Containers to be placed on board:

2, 3, 3, 4, 6, 6,

Initial Stack Profile (i)



(ii)

Figure 3.7 - Container Rearrangements

Suppose that the optimal policy determines to rearrange the top k containers $x_1, x_2,...,x_k$. Let x_e be a container with $d(x_e) = \min (d(x_q); q = k+1, k+2,...n)$, with $d(x_e) \leq d(x_k)$, if such a container exists. Let (i+r) be the first port. x_e is rearranged (obviously $i+r \leq d(x_e)$ Then, at port i+r, both x_e and x_k must be rearranged. Notice that the rearrangement of x_k at port i does not facilitate the handling of the containers which are above x_k, since it is not overstowed relative to any container of type (i+1),...(i+r), nor does it facilitate the delivery of any of the containers $x_{k+1}, x_{k+2},...x_{e-1}$ because none of them is to be delivered before port $d(x_e)$. So handling exactly k top containers at port i is not optimal; we can do better by rehandling the top k-1. The latter conclusion simply rules out rearranging exactly k

containers; the optimal number can be either greater or smaller than k.

The key notion in the above arguments is that there exists a container of smaller type (i.e. destined to a port closer) than that of the last container to be rearranged. This means that, for example at port i, there are M+1-i choices. Each one of these choices corresponds to rearranging all containers of a certain type, those of smaller types and possibly those of higher types that block the above ones. In terms of the example of Figure 3.7, we have the following choices:

(i) Rearrange up to type 2 --> Rearrange x_1, x_2, x_3, x_4

(ii) Rearrange up to type 3 --> Rearrange x_1, x_2, x_3, x_4

(iii) Rearrange up to type 4 --> Rearrange x_1, x_2, x_3, x_4

(iv) Rearrange up to type 5 --> Rearrange x_1, x_2, x_3, x_4, x_5

If no container x_4 exists, then the choice of the k top containers for rearrangement at port i is acceptable.

The above are summarized in the following lemma.

Lemma 3.3

The optimal rearrangement policy(s) at any port, i, involve rearrangements of all containers up to a certain type of containers (i.e. from (i+1) to (M+1)), and of course of those containers of higher type that block them. ∎

We can express the delivery of containers of type i to port j as rearrangements up to type i. We say that we rearrange up to type i, if we do not perform any "voluntary" rearrangement.

The implications of Lemma 3.3 are noteworthy. The lemma allows for the reduction of number of rearrangement decisions at each port from the number of groups present (see Lemma 3.2) to the number of container types present. For port i, i=1,...,M, the number of choices goes down from i(M+1-i) to (M+1-i), or in terms of big-O notation, from $O(M^2)$ to O(M).

Moreover, it allows the optimal rearrangement policy to be expressed as an M-component vector, $\underline{P}$, each component of which indicates up to what type of containers we rearrange at the corresponding port. For example P(2)=4 means that at port 2 we rearrange containers up to type 4, that is those of type 2 (to be delivered), type 3, type 4, and those of types 5,6,...(M+1) that block any of type 2, 3, or 4. Obviously,

$$P(i) \in \{i, i+1, i+2, \ldots M\}, \quad i=1,2 \ldots M \tag{3.2}$$

Notice that P(i) need not assume the value M+1, because whatever containers of type (M+1) are overstowed get rearranged if P(i) is equal to M, and those (M+1)'s that are not overstowed we do not want to move them since they do not block any container.

What has been achieved with Lemmas 3.1, 3.2, and 3.3 is a concise way to represent a set of policies that contain the optimal policy for any shipment matrix $C_{(M+1, M+1)}$. According to (3.2) this set is the set of M-vectors that satisfy (3.2). We are going to restrict further this set by Lemma 3.4. For the time being let us mention that a policy satisfying (3.2) along with Lemma 3.1 uniquely determine the profile of the stack at every port i, i=0,1,2,...M.

The most important property of an optimal rearrangement policy comes as an extension of the ideas presented in Lemmas 3.1, 3.2, and 3.3, namely to avoid redundant rearrangements. The following lemma presents this property.

Lemma 3.4

The optimal rearrangement policy vector should satisfy the condition:

$$\forall(i,j) : i<j \text{ and } P(i) \geq j \Rightarrow P(i) \geq P(j) \qquad (3.3)$$

Proof

What (3.3) says is that if an optimal policy, P, requires "voluntary" rearrangements of containers up to type P(9) at port i, then at any subsequent port j rearranging containers of type P(i)+1 or greater is non optimal, as long as P(i) $\geq$ j.

We are going to prove the lemma by contradiction. Assume that P is optimal and let i,j be two ports such that (3.3) is not satisfied. We can always choose i and j such that for all ports between i and j (3.3) holds. Since we assume that (3.3) does not hold fir i and j it should be

$$P(i) < P(j) \text{ and } P(i) \geq j \qquad (3.4)$$

Also it is assumed that P(i) > i. Let us now consider the containers of type j to P(i) at port i. These containers are rearranged at port i. However, all intermediate ports between i and j, involve deliveries of containers of types (i+1) to (j-1), and in addition, by our choice of i and j, they do not involve any

rearrangement of type j or greater. So, rearranging container types j to P(i) at port i is of no help at ports (i+1) to (j-1). At port j, the above containers are scheduled for rearrangement again. It is evident then, that their rearrangement at port i is redundant and consequently, a policy not satisfying (3.3) can not be optimal. ∎

By Lemma 3.4 we limited the set of policies containing the optimal one(s) to the M-vectors satisfying (3.2) and (3.3). Notice that nowhere in our proofs of Lemmas 3.1-3.4 have we made any assumption on the characteristic of the shipment matrix C. So, (3.2) and (3.3) should be satisfied independently of C. Two policies P_1 and P_2 satisfying (3.2) and (3.3) can be compared only in reference to a specific C. No policy satisfying these conditions dominates any other which also satisfied them for every C. It can be proven (by construction) that (Lemma 3.5):

Lemma 3.5

For any policy P (with M-components) satisfying (3.2) and (3.3), there exists a shipment matrix $C_{(M+1, M+1)}$ such that $\underline{P}$ is an optimal policy for $C_{(M+1, M+1)}$.

In the next section we define a classification of rearrangement policies. ∎

3.4 Classes of Rearrangement Policies

In the previous section, we have introduced the subset of rearrangement policies that can be represented by M-component vectors, and we have proved that the optimal policy is a member of this set. In this section we formalize the

classification of policies we introduced earlier.

(i) Set of vector-described policies:

$$P_v = \{P : P(i) \in \{i, i+1, \ldots M, M+1\},\ i=1,2,\ldots M\} \qquad (3.5)$$

A special subset of this set is the set defined in (3.2), as

$$P_{vo} = \{P : P(i) \in \{i, i+1, \ldots M\},\ i=1,2,\ldots M\} \qquad (3.6)$$

(ii) Set of efficient policies

$$P_e = \{P : P \in P_{vo} \text{ and } P \text{ satisfies } (3.3)\}$$

For the latter set of policies Lemma 3.5 holds as well. Notice again that is defined for any shipment matrix C. This is not the case with the

(iii) Set of optimal policies

$$P_0(C) = \{\underline{P} : \in \underline{P}_e \text{ and } R_p(C) \leq R_{p'}(C)\ \forall P' \in p_e\} \qquad (3.7)$$

3.5 The Decomposition Property

Let us start this section with some definitions again. Remember that we consider a series of ports 0,1,...M,M+1 that a containership is scheduled to visit. Recall also that c_{ij} is the number of containers which originate at port i and are shipped to port j.

Definition 3.4

The (K+1)-condensed version of the problem is a problem in which the containership is going to visit the first (K+1) ports of the original sequence facing the following shipment matrix:

$$c'_{ij} = c_{ij}\ ,\ i=0,...,k;\ j=1,...,k$$

$$c'_{i,k+1} = \sum_{l=k+1}^{M+1} c_{i,l}\ ,\ i = 0,1,...K \qquad (3.8)$$

Definition 3.5

The L-started version of the problem is a problem in which the containership is going to visit the last M+1-L ports of the original series starting at port L and facing the following shipment matrix:

$$c'_{ij} = c_{ij} \quad i=L+1,...M;\ j=L+2,...M+1$$

$$c'_{L,j} = \sum_{l=0}^{L} c_{l,j}\ ,\ j=L+1,...M+1 \qquad (3.9)$$

If we combine the above definitions, we can define a family of problems whose shipment matrix is a function of the original one and of the particular started and condensed version.

Definition 3.6

An L-started (K+1)-condensed version of the problem is that in which the vessel visits ports L, L+1,...,K,(K+1) and faces the following shipment matrix:

$$c'_{ij} = c_{ij},\ i=L+1,\ldots k;\ j=L+2,\ldots k+1$$

$$C'_{i,k+1} = \sum_{l=k+1}^{M+1} c_{i,l}\ ,\ i=L+1,\ldots k$$

$$C'_{Lj} = \sum_{l=0}^{L} C_{l,j}\ ,\ j=L+1,\ldots,K$$

$$C'_{L,k+1} = \sum_{l=0}^{L} \sum_{m=k+1}^{M+1} c_{lm}$$

A total of (M+1).(M+2)/2 problems can be defined in the above manner. In the following, PR(J,J) will denote the I-started, J-condensed problem ($J \geq I-1$). Then PR(0,M+1) is the original problem, while all the others are similar problems of smaller dimension, that is of shorter series of ports to be visited by the vessel. V(I,J) will stand for the optical overstowage cost of the problem.

The way PR(I,J)'s are defined leads to the following observation. Suppose at port I the stack is rearranged up to type M according to the optimal rearrangement policy (P*(I)=M). This means that as the vessel leaves port I the entire stack is "in-order". But then it satisfies the requirements of definition 3.5 and consequently the remaining overstowage cost is the same as of the problem PR(I,M+1), that is the I-started problem.

Notice that knowing that P*(I)=M is sufficient to compute the remaining overstowage cost. Whatever policy was followed between ports 0 and (I-1) is irrevelant. In fact, even if the policy from port 0 to port (I-1) was not the optimal, knowing that P(I)=M is still sufficient to find the optimal cost and policy from I to (M+1), simply by solving PR(I,M+1). The above is a proof of the following lemma.

Lemma 3.6

If P(i)=M for a policy $P \in P_e$ for the problem PR(0,M+1), then the optimal policy from i to (M+1) given P was followed up to port i, is the solution of the problem PR(i,M+1) and is independent of P along ports 1 to (i-1). ∎

Let us consider again the problem. PR(0,M+1), our original problem. Let P* be an optimal rearrangement policy. Let also i be the first port such that P*(i)=M. (There always exists such an i because $p^*(i) \in \{i, i+1, \ldots, M\}$) Observe that between ports 0 and i, P*(j) < M (j=1,...,i-1). This means that the containers of type M and M+1 are treated the same. Moreover due to Lemmas 3.1 and 3.2 containers of types M and M+1 from the same origin are always placed in adjacent cells (positions). We conclude that there is no need to distinguish between these two container types along ports 0 to i-1.

Now we apply Lemma 3.4. Since P*(j) < M, j=1,...,i-1, and P*(i)=M, it must be $P^*(j) \leq i-1$ for the policy $\underline{P}^*$ to be optimal. This is what Lemma 3.4 dictates. But then by extending our discussion of the previous paragraph to types i, i+1,...,M,M+1, we conclude that there is no need to distinguish among types i,i+1,...,M,M+1 at any of the ports 0 to i-1. In other words we can treat types i and greater as a single type, i. But this exactly meets the requirement of Definition 3.4: It is the i-condensed version. That is the optimal policy is the solution of PR(0,i). The following lemma summarizes the above discussion.

Lemma 3.7 (The decomposition property)

If i-min(j:P*(j)=M) for an optimal policy $\underline{P}^*$, then $\underline{P}^*$ consists of the following three parts:

(i) optimal policy of PR(0,i) for P*(j), j=1,...i-1

(ii) P*(i) = M

(iii) optimal policy of PR(i,M+1) for P*(j), j=i+1,...,M

■

Correspondingly the overstowage cost can be broken down in three components: (i) the cost of PR(0,i), (ii) the cost of rearranging up to type M at port i, and (iii) the cost of PR(i,M+1):

$$R_{p^*}(C) = V(0,M+1) = V(0,i) + r(0,i,M+1) + V(i,M+1) \qquad (3.11)$$

where r(0,i,M+1) represents the rearrangement cost (i.e. number of rearrangements) at port i of types i+1, i+2,...,M, under the assumptions that (i) the vessel visits a series of ports from 0 to M+1, and (ii) no rearrangements (other than the necessary) of types i to M have taken place at ports 0 to (i-1).

3.6 A Recursive Algorithm for the OSOP

We can exploit the decomposition property of the OSOP to develop an algorithm. In (3.11) we have assumed that we know what is i, that is the first part of rearranging up to type M. Since i is one of M possible choices (remember that at least one of P(j)=M, j=1,...,M), it goes without much analysis to conclude that i should be chosen to minimize the overstowage cost, $R_{p^*}(C)$. That is,

$$R_{p^*}(C) = V(0,M+1) = \min_{1 \le i \le M} \{V(0,i) + r(0,i,M+1) + V(i,M+1)\} \qquad (3.12)$$

Equation (3.12) defines a recursive formula to calculate V(0,M+1). The formula is recursive on the length of the series of ports the vessel is scheduled to visit. For example, in (3.12) the original (M+1)-long port series (not counting the starting port, 0) is expressed as a function of two shorter series of length i and M+1-i. Since i varies from 1 to M, we need the solution of 2M problems of smaller dimensions. The latter problems can, in turn, be solved with the same recursion formula. Eventually we must solve all problems PR(i,j), i<j. As we have computed earlier there are (M+1)(M+2)/2 such problems.

To fully define the recursion we must also define the boundary values. In this case it is clear that

$$V(i,i+1) = 0,\ i = 0,1,...M \qquad (3.12a)$$

Figure 3.8 demonstrates that

$$V(i,i+2) = 0,\ i = 0,...,M-1 \qquad (3.13)$$

as well.

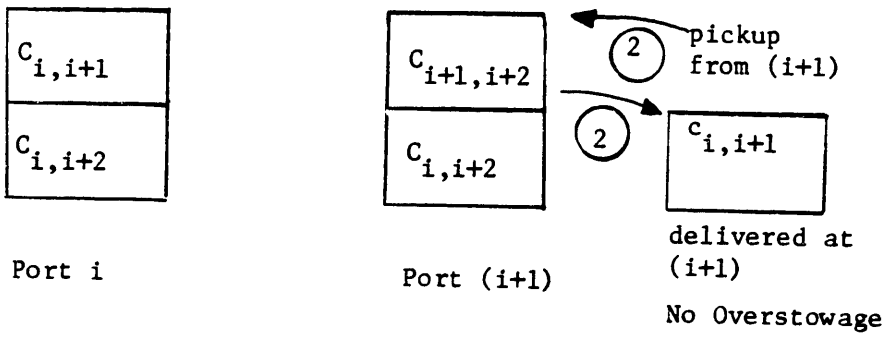


Figure 3.8 - Solution of V(i,i+2)

The general form of the recursive equation with the boundary condition are given in Theorem 3.1 below.

Theorem 3.1

The minimum overstowage cost and the corresponding optimal rearrangement policy of the one-stack overstowage problem, for an initial empty stack visiting a series of ports 0,1,2,...M,M+1 and facing a shipment matrix C is the solution of the following recursive equation.

$$V(i,j) = \min_{i+1 \le k \le j-1} \{V(i,k) + r(i,k,j) + V(k,j)\} \quad i=0,1,...M;\ j=i+1,...,M,M+1 \tag{3.14}$$

where V(i,j), r(i,k,j) as defined in Section 3.5. ∎

The evaluation of r(i,k,j) is done in detail in the next section. There is a total of

$$\sum_{i=0}^{M} \sum_{j=i+1}^{M-1} (j-1-i) \tag{3.15}$$

different values of r(i,k,j) to be computed.

Algorithm REARRANGE shown in Figure 3.9 implements the analysis above.

Algorithm REARRANGE

Step 1 Initialization and boundary conditions

```
For i=0 to M
    Set V(i,i+1) = 0
For i=0 to M-1
    Set V(i,i+2) = 0, h(i,i+2)=i+1;
```

Step 2 Calculation of rearrangement cost functions

```
For i=0 to M
    for j=i+1 to M+1
        for k=i+1 to j-1
            Calculate r(i,k,j);
```

Step 3 Recursive step

```
For d = 3 to M+1
    for i = 0 to M+1-d
        V(i,i+d) = V(i,i+k*) + r(i,i+k*,i+d) + V(i+k*, i+d)
        =   min    {V(i,i+k) + r(i,i+k,i+d) + V(i+k ,i+d)}
          1≤k≤d-1

        h(i,i+d) = k*;
```

Step 4 Rearrangement policy vector

```
S = {V(0,M+1)}
While S not empty to
    pick V(i,j) ϵ S
    erase V(i,j) from S
    add V(i,h(i,j)), V(h(i,j),j) to S
    P(h(i,j)) = j=1;
```

FIGURE 3.9 - REARRANGE: AN ALGORITHM FOR THE OSOP

The algorithm runs in polynomial time in the number of ports in the series as it is proven in the next theorem.

Theorem 3.2

The algorithm of Figure 3.9 correctly solves the one-stack overstowage problem in $O(M^3)$ time.

Proof

It correctly solves the OSOP by construction since it is a direct implementation of (3.14), which by Theorem 3.1, solves the OSOP problem. Step 4 retrieves the optimal policy after the recursion is done (h(i,j) is used to store the optimal choice for V(i,j)). The algorithm takes the following amount of time per step.

(i) Step 1: O(M) operations

(ii) Step 2: $O(M^3)$ operations (this is proven in the next section.

(iii) Step 3: There is a total of $O(M^2)$ problems. It takes O(M) comparisons and O(M) additions to solve each one. That is a total of $O(M^3)$.

(iv) Step 4: A maximum of M problems (V(i,j)) will be scanned. That is, O(M).

As it is evident from the above, the total running time is $O(M^3)$ operations. An operation is defined as an addition or a comparison. ∎

3.7 Evaluation of the Rearrangement Cost Functions

We are going to compute r(i,k,j) directly from their definition. r(i,k,j) has been defined as the number of rearrangements of types K+1, K+2,...,j-1 at port K for the i-started j-condensed version (V(i,j)) under the assumption that no rearrangements (other than the necessary ones) of types K or greater have been performed at ports i+1,...,K-1.

Figure 3.10 shows which groups of containers need to be rearranged. Their sum can be written as:

$$C^{ij} = \left[\begin{array}{ccccc|ccc} C'_{i,i+1} & & & & & & & \\ C'_{i,i+2} & C'_{i+1,i+2} & & & & & & \\ \vdots & \vdots & & & & & & \\ C'_{i,k} & C'_{i+1,k} & C'_{i+2,k} & & & & & \\ \hline C'_{i,j-3} & C'_{i+1,j-3} & C'_{i+2,j-3} \cdots & & & C'_{k,j-1} & & \\ C'_{i,j-2} & C'_{i+1,j-2} & C'_{i+2,j-2} \cdots & & & C'_{k,j-2} & & \\ C'_{i,j-1} & C'_{i+1,j-1} & C'_{i+2,j-1} \cdots & & & C'_{k,j-1} & C'_{j-2,j-1} & \\ \hline C'_{i,j} & C'_{i+1,j} & C'_{i+2,j} \cdots & & & C'_{k,j} & C'_{j-2,j} & C'_{j-1,j} \end{array}\right]$$

FIGURE 3.10

SHIPMENT MATRIX FOR V(i,j), C^{ij} GROUPS CONTRIBUTING TO r(i,k,j)

$$r(i,k,j) = \sum_{m=i+1}^{k-1} \sum_{l=k+1}^{M+1} c_{ml} + \sum_{p=0}^{i} \sum_{n=k+1}^{j-1} c_{pn} \qquad (3.16)$$

or in terms of the shipment matrix faced in problem PR(i,j).

$$r(i,k,j) = \sum_{m=i+1}^{k-1} \sum_{l=k+1}^{j} C'_{ml} + \sum_{n=k+1}^{j-1} C'_{in} \qquad (3.16a)$$

Also observe that if

$$\sum_{m=i+1}^{k} \sum_{l=w+1}^{M+1} C_{ml} = 0, \quad w=k+1,\ldots j-1 \qquad (3.17)$$

then

$$r(i,k,j) = \sum_{m=i+1}^{k-1} \sum_{l=k+1}^{M+1} C_{ml} + \sum_{p=0}^{i} \sum_{n=k+1}^{w_{min}-1} C_{pn} \qquad (3.18)$$

where w_{min} is the minimum value of w satisfying (3.17), if any. In 3.16a we can assume that w=k.

The last two equations take into account discontinuities to avoid redundant rearrangements of containers. For example as Figure 3.10 demonstrates, the group $c'_{i,j-1}$ is going to be rearranged only when there exists at least one container in one of the groups. $c'_{m,j}$, m=i+1,...k-1. Similarly $c_{i,j-2}$ must be rearranged only if there exists at least one container in any of the groups $C'_{m,n}$, m=i+1,...k-1; n=j-1,j.

The same rationale applies on containers of type j (in the j-condensed problem, that is of type j,...,M+1 in the original problem). If

$$\sum_{p=0}^{w} \sum_{n=k+1}^{j-1} C_{pn} = 0, \quad w = 1, i+1, \ldots k-1 \qquad (3.19)$$

then groups $C'_{m,j}$, m=i+1,...,w need not be rearranged and

$$r(i,k,j) = \sum_{m=i+1}^{k-1} \sum_{l=k+1}^{M+1} C_{ml} - \sum_{p=i+1}^{w_{max}+1} \sum_{n=j}^{M+1} C_{pn} \qquad (3.20)$$

where, again, w_{max} is the maximum value of w satisfying (3.19).

In summary, it turns out that the groups of the first column (i) and last row (j) of the matrix C^{ij} of Figure 3.10 give rise to discontinuities and required special treatment, according to (3.18) and (3.20). We are going to call the correction of equations (3.17) and (3.18) as the "row" correction (because it checks the sums of rows) and the correction of (3.19) and (3.20) as the "column" correction (because it checks the sum of columns i to w - see Figure 3.10). In a unified way the

expression for r(i,k,j) can be written as

$$r(i,k,j,) = \sum_{m=i+1}^{k-1} \sum_{l=k+1}^{M+1} C_{ml} + \sum_{p=0}^{i} \sum_{n=k+1}^{j-1} C_{pn} - \sum_{p=0}^{i} \sum_{l=k+1}^{j-1} C_{ml} - \sum_{m=i+1}^{w_{max}+1} \sum_{l=k+1}^{M+1} C_{ml} \qquad (3.21)$$

where w_{min} is the minimum w that satisfies (3.17) if there exists any between k+1 and j-1, else $w_{min} = j$, and, w_{max} is the maximum w that satisfies (3.19), if there exists any between i and k-1, else $w_{max} = i-1$.

The way r(i,k,j) is expressed in (3.21) leads to a straightforward calculation. This approach though even if we do not include the row and column corrections, requires a total of $O(M^2)$ additions for each r(i,k,j). Since there is a total of

$$\sum_{i=0}^{M} \sum_{j=i+1}^{M+1} (j-1-i) = O(M^3)$$

different r(i,k,j) to be computed, it takes $O(M^5)$ time to compute than all. That would determine the running time of algorithm REARRANGE. However, we can do much better. In fact we can compute all r(i,k,j)'s in $O(M^3)$ as described in the next section.

3.8 Recursive Computation of the Rearrangement of Cost Functions

We first define

$$d(i,k) = \sum_{p=0}^{i} C_{pk} \qquad (3.22)$$

and

$$o(i,k) = \sum_{p=k}^{M+1} C_{ip} \qquad (3.23)$$

Now we can write equations (3.17), (3.19) as well as the "row" and "column" corrections in terms of d(i,k) and o(i,k) as follows:

$$\sum_{m=i+1}^{k} \sum_{l=w+1}^{M+1} C_{ml} = \sum_{m=i+1}^{k} o(m,w+1) \qquad (3.24\text{-}i)$$

$$\sum_{p=0}^{w} \sum_{n=k+1}^{j-1} C_{pn} = \sum_{n=k+1}^{j-1} d(w,n) \qquad (3.24.ii)$$

$$\sum_{p=0}^{i} \sum_{n=w_{min}}^{j-1} = \sum_{n=w_{min}}^{j-1} d(i,n) \qquad (3.24iii)$$

$$\sum_{m=i+1}^{w_{max}+1} \sum_{l=k+1}^{M+1} C_{ml} = \sum_{m=i+1}^{w_{max}+1} o(m,k+1) \qquad (3.24iv)$$

From the above we conclude that we basically need to calculate two functions

$$a(i,k,w) = \sum_{m=i+1}^{k} \sum_{l=w+1}^{M+1} C_{ml} = \sum_{m=i+1}^{k} o(m,w+1)$$

$$i=0,\ldots M-1;\ k=i+1,\ldots M;\ w=k+1,\ldots M$$

and

$$b(w,k,j) = \sum_{p=0}^{w} \sum_{n=k+1)}^{j-1} C_{pn} = \sum_{n=k+1}^{j-1} d(w,n)$$

$$w=0,1,\ldots M-1;\ k=w+1,\ldots M;\ j=K+1,\ldots,M+1;$$

The first two terms of (3.21) can be written as:

$$\sum_{m=i+1}^{k-1} \sum_{l=k+1}^{M+1} C_{ml} + \sum_{p=0}^{i} \sum_{n=k+1}^{i=i} C_{pn} = \sum_{m=i+1}^{k-1} o(m,k-1) + \sum_{n=k+1}^{j-1} d(i,n) =$$

$$a(i,k-1,k) + b(i,k,j)$$

Then, we can write r(i,k,j) as

$$r(i,k,j) = a(i,k-1,k) + b(i,k,j) - b(i,w_{min}-1,j) - a(i,w_{max}+1,k) \quad (3.28)$$

We can prove the following.

Lemma 3.8

Functions d(i,k) and o(i,k) can be computed in $O(M^2)$ time.

Proof

By definition

$$d(i,k) = \sum_{p=0}^{j} C_{pk}$$

Then we can write d(i,k) = d(i-1,k) + c_{ik} and the following order of computations calculates all d(i,k)'s in $O(M^2)$ time.

For k = 1 to M do

{d(0,k) = C_{0k}

for i = 1 to (k-1) do

d(i,k) = d(i-1,k) + c_{ik}}

In the above implementation of the recursion for d(i,k), O(M) additions are sufficient to compute all d(i,k)'s. Functions o(i,k) are computed in the same manner.

Lemma 3.9

Functions a(i,k,w) and b(w,k,j) can be computed in $O(M^3)$ time.

Proof

We prove the lemma only for a(i,k,w). The proof for b(w,k,j) is similar. Since

$$a(i,k,w) = \sum_{m=i+1}^{k} o(m,w+1),\ i=0;\ k=i+1,\ldots M;\ w=k+1,\ldots M$$

we can write the following recursions

a(i,k,w) = a(i, k-1, w) + o(k,w+1)

a(i,k,w) = a(i-1,k,w) - o(i-1, w+1)

Also for k=i+1, we get a(i,i+1,w) = o(i+1,w+1) and finally the following implementation achieves the desired computation time. There are three nested loops with at most M iterations each.

```
for i = 0 to M-1 do
    for w = i+2 to M do
        {o(i,i+1,w) = o(i+1, w+1)
        for k=i+2 to w-1 do
            a(i,k,w) = a(i,k-1,w) + o(k,w+1)}
```

For functions b(w,k,j) we have

$$b(w,k,k+1) = d(w,k)$$

$$b(w,k,j) = b(w,k,j-1) + d(w,j-1)$$

So the following scheme computes b(w,k,j) in $O(M^3)$ time

```
for w=0 to (M-1) do
    for k=w+1 to M do
        {b(w,k,k+1) = d(w,k)
        for j=(K+2) to (M+1) do
            b(w,k,j) = b(w,k,j-1) to d(w;j-1)}
```

■

To directly proceed into the computation of r(i,k,j) we further need to know the value of w_{min} and w_{max}. In fact we calculate those for each triple t(i,k,j) from the information contained in a(i,k,j) and b(w,k,j); $w_{min}(i,k,j)$ and $w_{max}(i,k,j)$ hold the values of w_{min} and w_{max}. The following achieves the task in $O(M^3)$ time:

i. w_{min}: for i=0 to M do

for k=i+1 to M do

{if a(i,k,k+1) = 0 then w_{min} (i,k,k+1) = k else w_{min} (i,k,k+1) = k+1

for w = k+2 to M do

if (a(i,k,w) = 0) then

if (w_{min}(i,k,w-1)=0) then

w_{min}(i,k,w) = w_{min}(i,k,w-1)

else w_{min}(i,k,w) = w-1

else w_{min}(i,k,w) = w;}

ii. w_{max}: for k=1 to M do

for j=k+1 to M+1 do

{if (b(k-1,k,j) then w_{max}(k-1,k,j) = k=1 else w_{max}(k-1,k,j) = k-2

for w = k-2 down to 0 do

if (b(w,k,j) = 0), then

if (b(w-1,k,j)=0) then

w_{max}(w,k,j) = w_{max}(w+1,k,j)

else w_{max}(w,k,j) = w

else w_{max}(w,k,j) = w-1;}

Now every term in (3.28) has been computed independently for all possible combinations of i,k,j in $O(M^3)$. Consequently, each r(i,k,j) can be computed in constant time (it takes only three additions). Since there exists $O(M^3)$ r(i,k,j)'s (see page 67) the overall time of calculating the rearrangement cost functions is $O(M^3)$.

Theorem 3.3

The rearrangement cost functions, r(i,k,j), (step 2 of algorithm REARRANGE) can be computed in $O(M^3)$ time.

■

3.9 A Numerical Example

We illustrate algorithm REARRANGE by a numerical example. suppose that M=4, that is a vessel is scheduled to visit ports 0, 1, 2, 3, 4, and 5. The shipment schedule is given below

$$C = \begin{bmatrix} 5 & & & & & 1 \\ 6 & 7 & & & & 2 \\ 2 & 3 & 4 & & & 3 \\ 7 & 1 & 2 & 6 & & 4 \\ 3 & 6 & 11 & 2 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & \end{bmatrix}$$

then we have

(i) $h(0,2)=1,\ h(1,3)=2,\ h(2,4)=3,\ h(3,5)=4$

$0 = V(0,1)=V(0,2)=V(1,2)=V(1,3)=V(2,3)=V(2,4)=V(3,4)=V(3,5)=V(4,5)$

(ii)

$r(0,1,2) = 0$

$r(0,1,3) = 6$

$r(0,2,3) = 3+1+6=10$

$r(0,1,4) = 6+2=8$

$r(0,2,4) = 3+1+6+2=12$

$r(0,3,4) = 1+6+2+11=20$

$r(0,1,5) = 6+2+7=15$

$r(0,2,5) = 3+1+6+2+7=19$

$r(0,3,5) = 1+6+2+11+7=27$

} observe $r(0,i,5)=r(0,i,4) + c_{04}$, $i=1,2,3$

$r(0,4,5) = 6+11+2 = 19$

$r(1,2,3) = 0$

$r(1,2,4) = 2+3=5$

$r(1,3,4) = 2+11=13$

$r(1,2,5) = 2+3+7+1=13$

$r(1,3,5) = 2+11+7+1=21$

$r(1,4,5) = 11+12=13$

$r(2,3,4) = 0$

$r(2,3,5) = 7+1+2 = 10$

$r(2,4,5) = 2$

$r(3,4,5) = 0$

In the above calculations we use (3.21) directly. In fact we do not bother to check the correction terms ("row" and "column") because all c_{ij} are positive. Now we can proceed in the recursion:

(iii)

$$V(0,3) = \min \begin{cases} V(0,1) + r(0,1,3) + V(1,3) = 0+6+0 = 6 \\ V(0,2) + r(0,2,3) + V(2,3) = 0+10+0 = 10 \end{cases}$$

$h(0,3) = 1,\ V(0,3) = 6$

$$V(1,4) = \min \begin{cases} V(1,2) + r(1,2,4) + V(2,4) = 0+5+0 = 5 \\ V(1,3) + r(1,3,4) + V(3,4) = 0+13+0 = 13 \end{cases}$$

$h(1,4) = 2,\ V(1,4) = 5$

$$V(2,5) = \min \begin{cases} V(2,3) + r(2,3,5) + V(3,5) = 0+10+0 = 10 \\ V(2,4) + r(2,4,5) + V(4,5) = 0+2+0 = 2 \end{cases}$$

$h(2,5) = 4,\ V(2,5) = 2$

$$V(0,4) = \min \begin{cases} V(0,1) + r(0,1,4) + V(1,4) = 0+8+5 = 13 \\ V(0,2) + r(0,2,4) + V(2,4) = 0+12+0 = 12 \\ V(0,3) + r(0,3,4) + V(3,4) = 6+2-+0 = 26 \end{cases}$$

$h(0,4) = 2,\ V(0,4) = 12$

$$V(1,5) = \min \begin{cases} V(1,2) + r(1,2,5) + V(2,5) = 0+13+2 = 15 \\ V(1,3) + r(1,3,5) + V(3,5) = 0+21+0 = 21 \\ V(1,4) + r(1,4,5) + V(4,5) = 5+13+0 = 18 \end{cases}$$

$h(1,5) = 2,\ V(1,5) = 15$

$$V(0,5) = \min \begin{cases} V(0,1) + r(0,1,5) + V(1,5) = 0+15+15 = 30 \\ V(0,5) + r(0,3,5) + V(3,5) = 6+27+0 = 33 \\ V(0,4) + r(0,4,5) + V(4,5) = 12+19+0 = 31 \end{cases}$$

$h(0,5) = 2,\ V(0,5) = 21 =$ **optimal (minimum) overstowage cost**

(iv)

The policy vector is

$$P(h(0,5)) = 5-1 \Rightarrow P(2) = 4,\ P(h(0,2)) = 2-1 \Rightarrow P(1) = 1,\ P(h(2,5)) = 5-1 \Rightarrow p(4)$$

and the optimal policy vector is $P^* = (1,4,3,4)$.

3.10 Viewing the Recursion Algorithm as a Generalized Network Flow Problem

As it has become evident from the analysis in the previous sections, that the recursive algorithm that was developed repeatedly uses the decomposition property. Figure 3.11 visualizes this fact by representing the following recursion as a network flow.

In the network each (m,l) node corresponds to the PR(m,l) problem.

$$V(i,j) = \min_{i+1 \le k \le j-1} \{V(i,k) + r(i,k,j) + V(k,j)\} \qquad (3.29)$$

Each (K) node corresponds to the first port at which containers of type (j-i) get rearranged. We also distinguish two types of edges. The A edges are connected to the choice of K for the problem PR(i,j). Transversing these edges carries a cost of r(i,k,j). In addition the flow gets multiplied by a factor of two. So for one unit of flow leaving node (i,j) there are two units arriving at one of the k-nodes. The B edges have no cost or multipliers associated with them. However they must observe a capacity constraint. Specifically they can carry only one unit of flow. So if we sent a unit of flow of node (i,j), it doubles along an A-edge, reaches a k-node, and then it splits in two and uses both of the two available edges (B edges)

emanating from the node. If we associated with the C edges, the overstowage cost of the corresponding problem PR(m,l), then the solution of (3.29), that is the choice of k, corresponds to the minimum cost solution of sending one unit of flow throughput the network of Figure 3.11, entering at node (i,j) and exiting (as two units) at node "sink".

A similar subnetwork can be set up for each of nodes (m,l)-m = i, j-1, l=j or m=i, l=i+j, j. Then the entire network has $O(M^3)$ nodes, because there exists $O(M^2)$ nodes (m,l) and there are at most O(M) K-(or choice)-nodes associated with each one of them. It also has $O(M^3)$ edges because there are only one or two arcs emanating from each node. Also it contains no cycles. Figure 3.12 shows the entire such network for M=3.

The running time of this approach is not better than that of the recursion algorithms. To compute r(i,k,j) takes $O(M^3)$ time as before. The reason we discuss this formulation is to provide a visual representation of the OSOP solution.

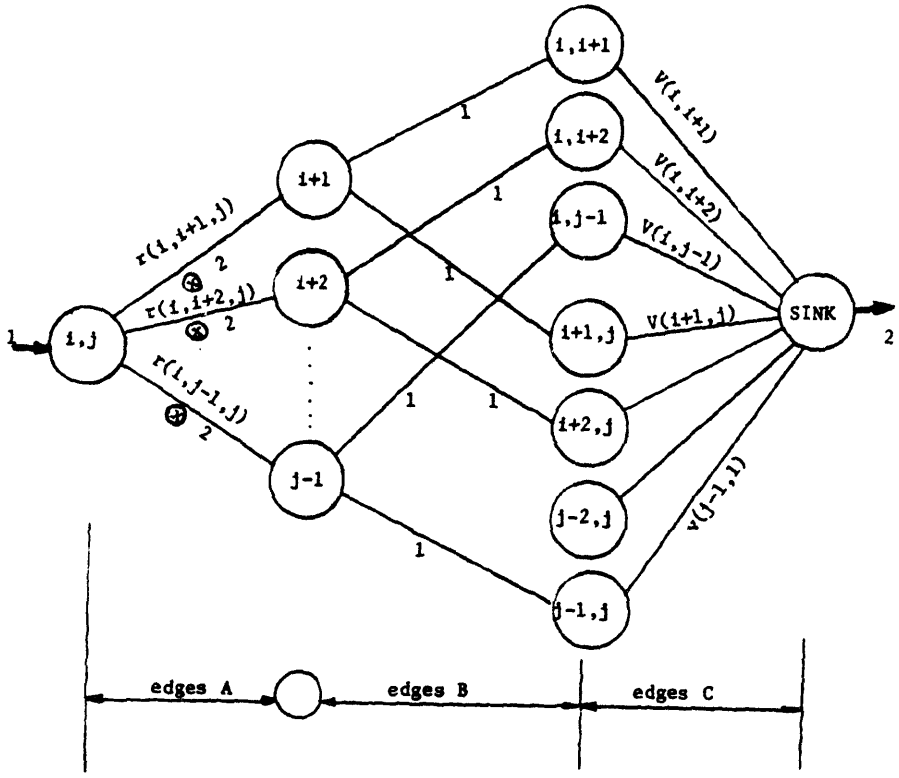


Figure 3.11 - Network Representation of Recursion

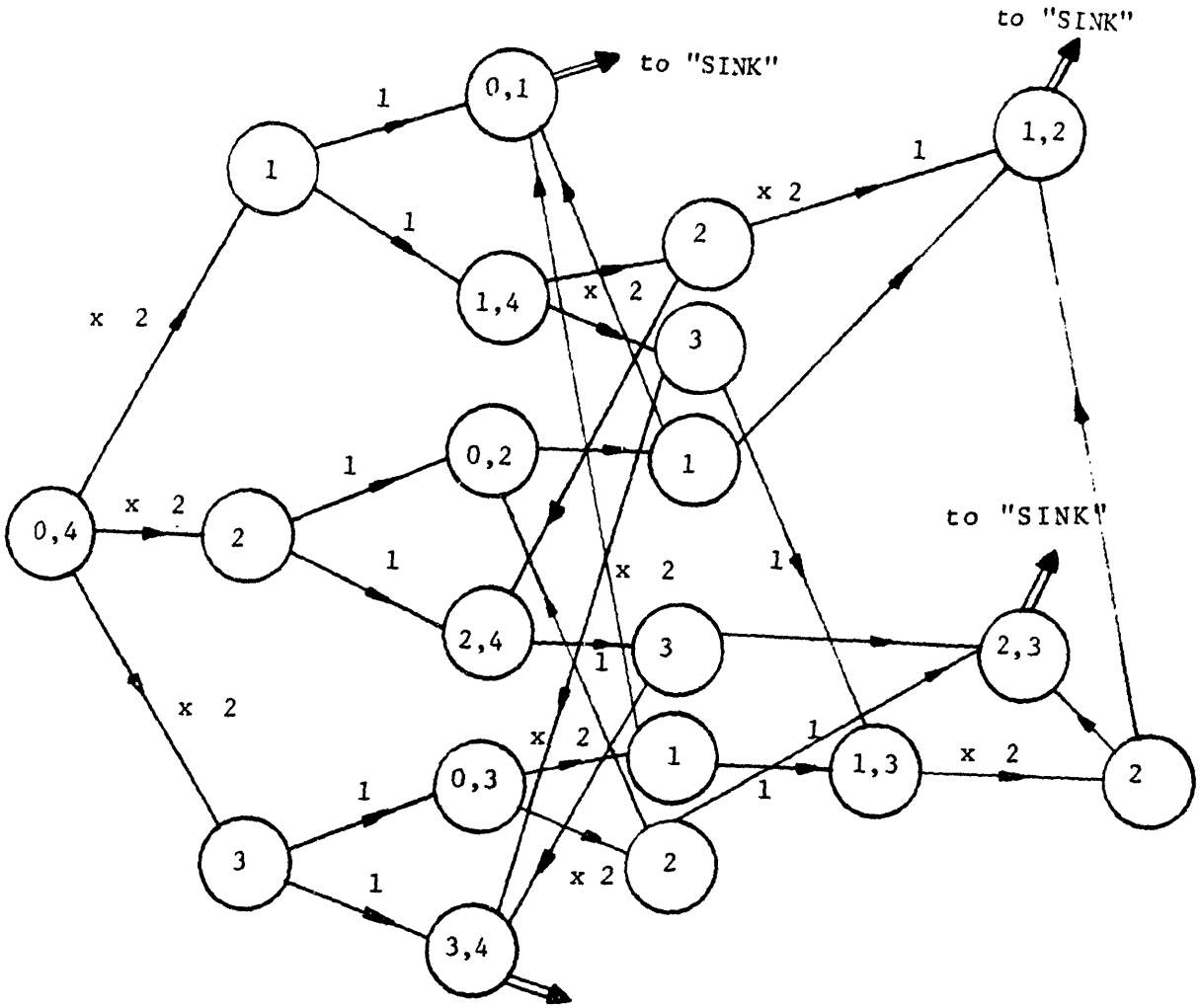


Figure 3.12

Generalized Network Formulation of OSOP
Arcs with Multipliers Bear in Addition Rearrangement Costs

CHAPTER 4

SENSITIVITY ANALYSIS AND RELATED ISSUES OF THE OSOP ALGORITHM

The previous chapter has been devoted to the analysis of the single stack overstowage problem. Rearrangement policies have been expressed in a concise way and a recursive algorithm has been constructed, which runs in polynomial time in terms of the number of ports in the sequence. This chapter deals with sensitivity analysis issues. In particular we look at how the minimum overstowage cost or the optimal rearrangement policy change with changes in the elements of the shipment matrix. We also look at the minimum information required to determine the profile of the stack at any given port. We briefly discuss bounds of the minimum overstowage cost and finally we present a different formulation of overstowage.

4.1 Overstowage Cost as a Function of the Elements of the Shipment Matrix

Let $\underline{P}$ be a vector-described policy that is $\underline{P} \epsilon P_v$. Since containers of the same group are treated the same, then given a policy $\underline{P}$ we can easily compute how many times each group of containers gets rearranged. Let $f_{ij}(P)$ denote the number of times group (i, j) is rearranged under policy $\underline{P}$. Then the resulting rearrangement (or overstowage) cost, R_p can be written as

$$R_p(C) = \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} f_{ij}(P) \cdot c_{ij} \qquad (4.1)$$

The coefficients $f_{ij}(\underline{P})$ in (4.1) depend only on $\underline{P}$, and consequently remain the same for all shipment matrices. This indicates that for any given policy $\underline{P}$, the resulting

overstowage cost is a linear function of the c_{ij}'s.

Obviously the above result holds for efficient and optimal policies as well. Then the minimum overstowage cost, $R_{p^*}(C)$, is:

$$R_{p^*}(C) = \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} f_{ij}(P^*)\ c_{ij} \qquad (4.2)$$

Let $R^*(C)$ indicate also the minimum overstowage cost for the shipment matrix C (we drop $\underline{P}$ from the subscript and we refer to the optimal policy for the given C). Also, since p^* depends on C, we write $f_{ij}^*(C)$, the coefficients that correspond to the optimal policy for shipment matrix C.

Let us now change one c_{ij} by $d(c_{ij}) > 0$. Then we have:

$$R^*(C + d(c_{ij})) \geq R^*(C) \qquad (4.3)$$

Expression (4.3) is obvious. If it were not true, then we could apply $\underline{P}^*$ $(C + c_{ij})$ to the shipment matrix C and get

$$R^*(C) > R^*(C+d(c_{ij})) \geq R_{p^*(C+d(c_{ij}))}(C) \qquad (4.4)$$

which is contradicting the face that $R^*(C)$ is the minimum overstowage cost for shipment matrix C. Then, $R^*(C)$ as a function of c_{ij} looks like in Figure 4.1 below for all other c_{ij} held constant.

For the slight altered shipment matrix $C+d(c_{ij})$ $(d(c_{ij}) > 0)$ also holds that

$$R^*(C+d(c_{ij})) \leq R^*(C) + f^*_{ij}(C).c_{ij} = R_{p^*(C)}(C+d(c_{ij})) \qquad (4.5)$$

The latter expression indicates that $R^*(C+ c_{ij})$ is a concave function of c_{ij}, $(d(c_{ij}))$, and in fact is as drawn in Figure 4.1.

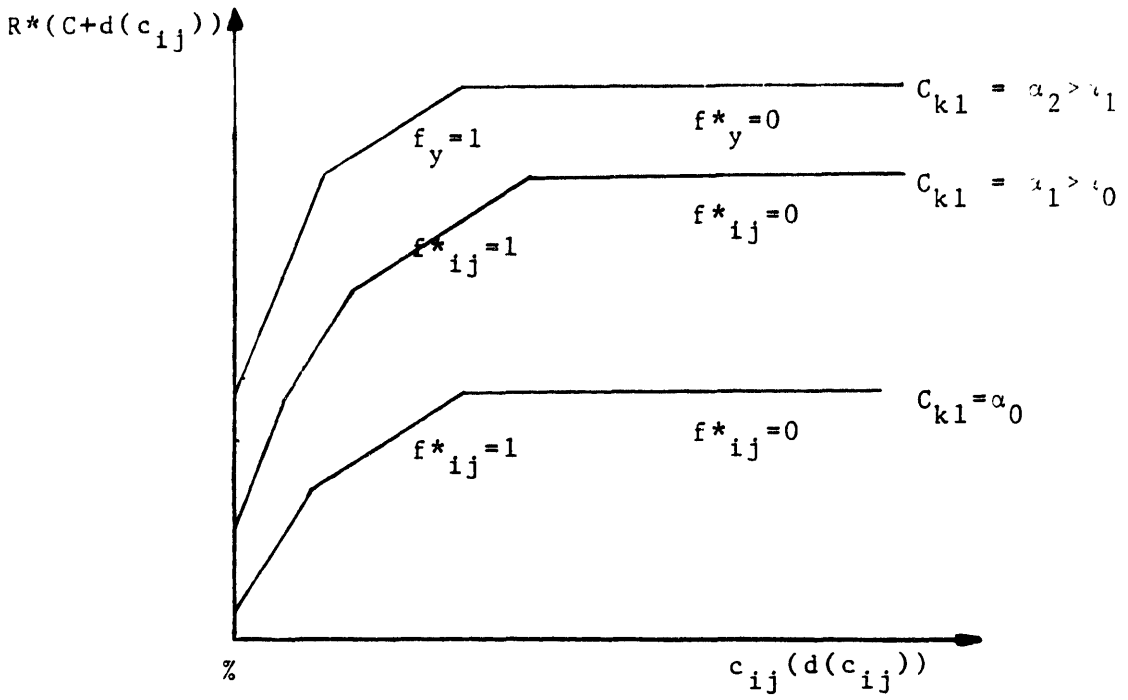


Figure 4.1 - R*(C) as a Function of C_{kl}

It is worth noting that $f^*_{ij}(C'+d(c_{ij}))$ can take a finite number of values, specifically the values

$$f^*_{ij}(C) \in \{0,1,2,\ldots,j-i-1\}, \forall c \qquad (4.6)$$

The result is that the slope of $R^*(C+d(c_{ij}))$ changes according to (4.6) and takes an only integer values. It agrees with our intuition (and it can be proven very easily) that if c_{ij} becomes large enough, no rearrangements of this group will be optimal. This means that it should be $P^*(i) \geq j$.

We can define partial derivatives of an optimal policy, too.

$$d^*_{ij}(C) = \frac{\partial R^*(C)}{\partial C_{ij}} = R^*(C + d(c_{ij}) \; ; \; d(c_{ij}) = 1) - R^*(C) \qquad (4.7)$$

It is clear from Figure 4.1 that $\frac{\partial R^*(C)}{\partial C_{ij}}$ is a non-increasing function of c_{ij}. Also, the following inequality holds:

$$d^*_{ij}(C) = \frac{\partial R^*(C)}{\partial C_{ij}} \leq f^*_{ij}(C) \qquad (4.8)$$

It should become clear that $d_{ij}(C)$ is always a function of the shipment matrix, because it is defined only for an optimal rearrangement policy. On the contrast, $f^*_{ij}(C)$ is the number of times group (i,j) is rearranged under the policy which is optimal for shipment matrix C.

4.2 Minimum Information for the Determination of the Stack Profile

In many cases it is useful to know the minimum information about the rearrangement policy required to determine the profile of the stack at a given port, say port K. Again we restrict ourselves to policies in P_e. Obviously, if we know the policy vector from ports 1 to (K-1), then the profile of the stack is fully determined. Nevertheless, to determine the latter is not always necessary to know all P(1) to P(K-1).

Let us assume that we know that P(i) = M, for some i in {1,2,...,K-2, K-1}. That is we know that the stack is "in-order" as the vessel leaves port i. But then, we do not need to know anything about the rearrangement policy before port i. In fact among those ports, i, with P(i) = M, we want to know only the last one, say i_1.

Now that we know that $P(i_1) = M$ and $P(i) < M$ for $i > i_1$, we turn our attention to ports at which rearrangements up to type (M-1) are performed. Let i_2

be the last such port before port K, $i_2 \in \{i_1, i_1+1,...,K-1\}$. As the vessel leaves port i_2 the stack looks like

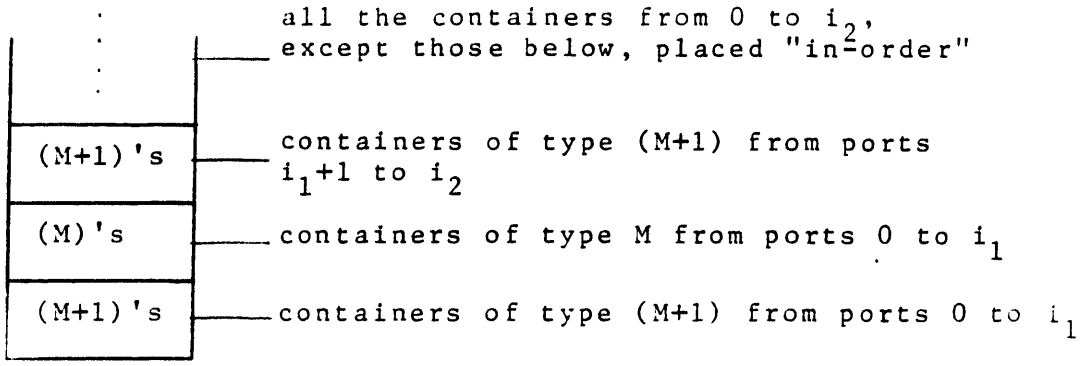


Figure 4.2 - Stack Profile

Continuing in the same line of thought we conclude that in order to determine the profile of the stack at port K, we only need to know where is the port of last rearrangement up to type i, for i = K, K+1,...,M. The following theorem has been proven.

Theorem 4.1

The stack profile as the vessel enters port K is fully determined if the port of last rearrangement of container types K, K+1,...,M+1 is known. ∎

Theorem 4.1 assumes that Lemmas 3.1, 3.2, and 3.3 are observed. Then, as stated, the profile of the stack is uniquely determined. The latter means that a count of the different ways the information required by Theorem 4.1 will be the

same with the number of all possible stack profiles. So there exists a total of S_t stack profiles, where

$$S_K = \sum_{i=0}^{K-1} \binom{M-(k-1)}{i} \binom{K-1}{i} \qquad (4.9)$$

where i indicates how many types among the M-(K-1) have distinct best-rearrangement ports. For example if the last rearrangement port of type M is i_1 and that of type (M-1) is $i_2 = i_1$, then we say that type (M-1) does not have a distinct last rearrangement port. So, the term $\binom{M-(K-1)}{1}$ counts all possible ways of having i types with distinct last-rearrangement ports and the term $\binom{K-1}{i}$ counts the different ways the chosen i types, can be distributed over the ports 1 to K-1. Of course, all possible values of i should be accounted for. Finally, we assume that if a type is not rearranged at any port it does not have a distinct port of last rearrangement (that is, port 0 is not accepted as such for type M).

Theorem 4.2

The number of different stack profiles as the vessel arrives at Port K is

$$S_K = \sum_{i=0}^{K-1} \binom{M-(k-1)}{i} \binom{K-1}{i}$$

■

4.3 An Alternative Formulation of Overstowage

In this section we are going to model overstowage in a different way. The basis of the formulation is the observation that when a group of containers, say (i, j) is rearranged at port K (i<K<j), it gets attached to the group of containers (k, j) (see Lemmas 3.1 and 3.3). So for all practical purpose the containers of group (i, j) become containers of an enlarged group (k, j).

Methodologically, we can substitute c_{ij} containers of group (k, j) for an equal amount of containers of group (i, j) at port k. So, if we know that group (i, j) gets rearranged at port k we can change the shipment schedule to reflect the above substitution. That is to replace c_{ij} containers of group (i, j) by c_{ij} containers of group (i, k) and equal number of containers of group (k, j). Figure 4.3 shows the situation schematically.

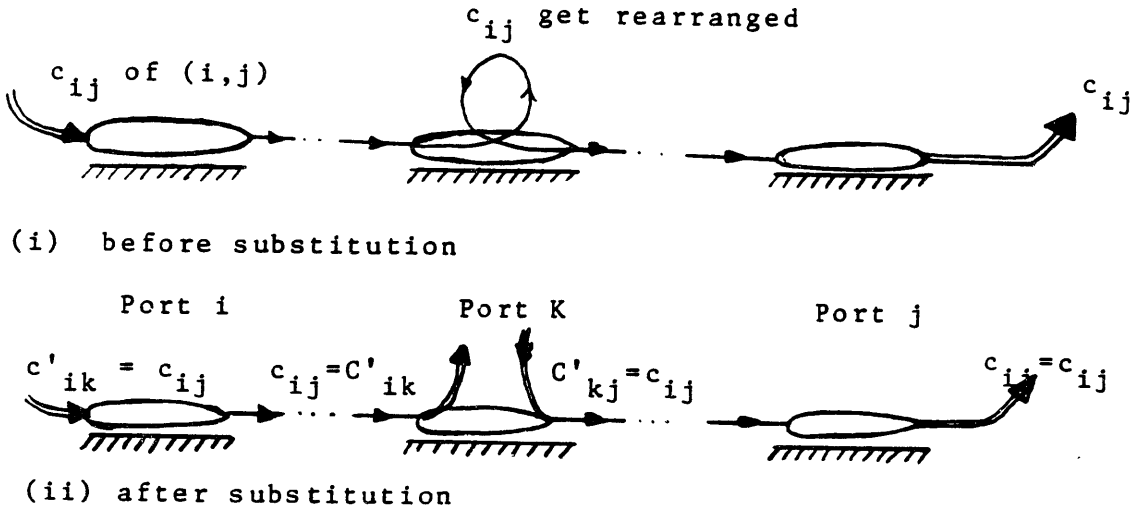


Figure 4.3 - Substitution of Rearranged Containers

This substitution is straightforward as long as we know the port of rearrangement. Assuming that the latter information is known, that is that the rearrangement policy is set, we can perform the above substitution for each group at its first port of rearrangement. This gives a new shipment matrix which does not give rise to any overstowage situation (by construction). However, it employs a larger number of containers because every time we substitute a group of containers we double the number of containers of it. In fact, this is exactly what is achieved by substituting group (i, j) by c_{ij} containers of group (i, k) and c_{ij} containers of group (k, j). The number of rearrangements goes down by c_{ij}, but the number of containers to be shipped increases by the same amount B doing the same substitution for all groups (in a specified order) we drive the overstowage cost to zero and introduce a number of new containers equal to the overstowage cost.

Algorithm FINDSHIPMENT of Figure 4.5 computes the transformed matrix for any given policy $\underline{p} \epsilon p_u$. If n' $(= \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} c'_{ij})$ (is the number of containers in the new matrix and $(= \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} c'_{ij})$ the same number of the original shipment matrix, it holds that

$$n' = n + R_{\underline{p}}(C) = R^*(C') \qquad (4.10)$$

It would be interesting to explore how we could use this trick to design an algorithm to find the optimal rearrangement policy. Since we are interested only in the first part each group gets rearranged we can formulate the problem as shown in Figure 4.6.

In Figure 4.6 for each group of containers there exists a choice for its first port of rearrangement (either "forced" or "voluntary"). Then the containers of the group join the group of containers of the same type originating at that port. There is an externally incoming flow of containers, c_{ij}, at each node (i, j). Of course, there exists a "sink", port of delivery, for each type of containers.

The objective is to minimize the sum of container flows over all arcs except from those leading to the "sink" node (if we include the latter we simply double-count the real containers). As it appears from Figure 4.6, that the solution is easy, and in addition, the network has (M+1) unconnected components. However, this is not the case. Because we have assumed that the overstowage cost of the resulting shipment matrix must be zero,

---

Algorithm FINDSHIPMENT

Input: Shipment matrix C, rearrangement policy P
Output: Shipment matric C' resulting in no overstows under policy P_o, with $P_o(i) = i$, i=1,...M and

$$\sum_{i=0}^{M} \sum_{j=j+1}^{M+1} c_{ij}' = \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} c_{ij} + R_p(C)$$

begin {FINDSHIPMENT}

Initialize $c_{ij}' = 0$ for all i,j;
for i=0 to M do
for j = i+1 to M+1 do

begin
find K = min{l : {(l) $\geq$ P(i)};
$c_{ik}' = c_{ik}' + c_{ij}$;
if (k=j) then $c_{kj}' = c_{kj}' + c_{ij}'$

```
        end;
end.
```

Figure 4.5 - Algorithm FINDSHIPMENT

---

under policy P_o (= policy with no voluntary rearrangements), we must prevent the presence of positive flows on arcs that create overstowage. For example arcs (i, j-k) and (m, l-n) can not both have positive flows in the final solution, if $i<m<K<n$ or $m<i<n<K$.

So along with (M+1)-component network of Figure 4.6 comes a set of "either-or" constraints which make this approach rather inefficient at least to the extent that a polynomial algorithm of the OSOP is sought. The number of "either-or" constraints is equal to the number of groups that are "blocked" by some other groups. We can define:

Definition 4.1: Two groups (i, j) and (k, l) block each other if $i<k<j<l$ or $k<i<l<j$. Groups (i,j) and (k,l) are then called blocking. ∎

Let b_{ij} be the number of groups that block group (i, j). We have

$$b_{ij} = i(j-1-i) + (j-1-i)(M+1-j) \qquad (4.10)$$

$$= (j-1-i)(M-(j-1-i))$$

Figure 4.3 shows the number of blocking groups for each group (i, j) for M = 4. Then we can write the "either-or" constraints in a simple form (x_i denotes the flow on arc i),

$$x_a \cdot x_b = 0 \qquad (4.11)$$

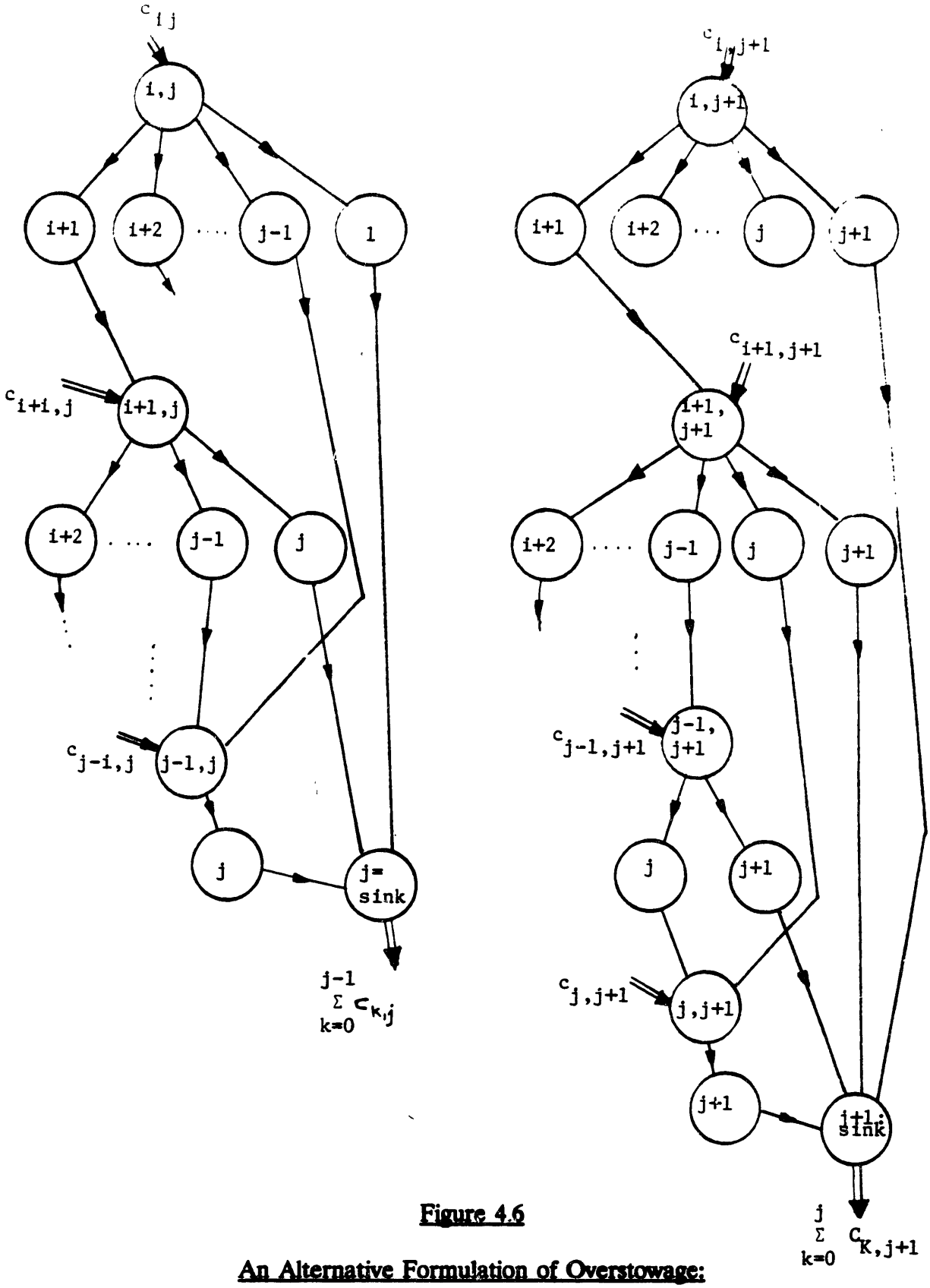


Figure 4.6

An Alternative Formulation of Overstowage: A Fully Developed Network

destination j/i	origin 0	1	2	3	4
1	0				
2	3	0			
3	4	3	0		
4	3	4	3	0	
5	0	3	4	3	0

Figure 4.7 - Number of Blocking Groups for M=4

where a and b are arcs corresponding to blocking groups, or as as,

$$x_a(x_b + x_c + \ldots x_z) = 0 \qquad (4.12)$$

where all the arcs b, c,...z correspond to groups that block the group of arc a. We must impose

$$\sum_{i=0}^{M} \sum_{j=i+1}^{M+1} b_{ij}/2$$

"either-or" constraints of the form of (4.11) or

$$\sum_{i=1}^{M} i - 1 = M(M+1)/2 - 1$$

constraints of the form of (4.12).

The analysis of this section, although it did not lead to an efficient algorithm, revealed a lot of the underlying physical structure of the problem that is going to be useful in the solution of the multi-stack case.

To eliminate some redundance in the representation, we conclude this section by redrawing the acyclic network of Figure 4.6 in a more compact way.

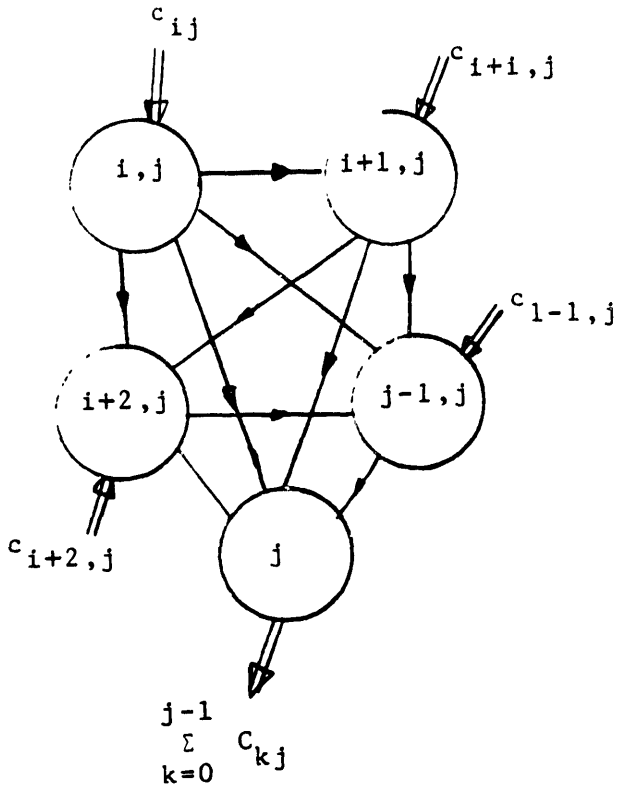


Figure 4.8 - An Alternative Formulation of Overstowage

CHAPTER 5

EXTENSIONS OF THE OSOP ALGORITHM

In this chapter we examine the possibilities of the algorithm developed in Chapter 3 and we try to model a variety of situations. The assumptions about the initial condition of the stack is relaxed and the case of different cost of rearrangement per port is modeled. Finally, we examine optimal policies under probabilistic shipment matrices and we conclude the chapter with the rolling horizon problem.

5.1 Overstowage with Different Port Costs

As the containership visits the different ports, she may meet different operating conditions at each one of them. As a consequence, it may cost more to rearrange a number of containers at one port than at another.

Let us assume that the cost per container rearrangement at port k is w_k dollars. dollars. Then the dollar cost of rearranging containers up to type j-i at port k for the i-started j-condensed version of the problem is:

$$r\$(i,k,j) = w_k \cdot r(i,k,j) \qquad (5.1)$$

where r$(i,k,j) stands for the dollar cost.

It is not hard to see that Theorem 3.1 still applies. Simply we substitute r$(i,k,j) for r(i,k,j) to take into account the different port costs. In fact, Lemmas 3.1 to 3.7 do not assume any particular cost structure other than that the cost of any single rearrangement is positive. That is, as long as there exists a one-to-one

correspondence between the number of containers to be rearranged, x, at a particular port, k, such as the cost of rearranging an additional container (marginal cost) is always positive than all the proofs of Lemmas and Theorems of Chapter 3 still hold.

Let $f_k(x)$ be a monotonically increasing function, like any of those shown in Figure 5.1 (a-d)., and let

$$r\$(i,k,j) = f_k(r(i,k,j)) \qquad (5.2)$$

If we assume also that $f_k(x)$ can be computed in constant time 0(1) for any value of x (x is obviously non-negative), then calculating r$(i,k,j)'s takes again $O(M^3)$ t. The following theorem has been proven.

Theorem 5.1:

The recursive algorithm of Theorem 3.1 for the single-stack overstowage problem holds and runs in the same $O(M^3)$ time, for any rearrangement cost function that is a monotonously increasing function of the number of rearranged containers. These functions need not be the same for every port. ∎

To see why the requirement that f(x) should be increasing[1] is importance, notice that if f(x) were decreasing for some range of x, it would be conceivable to allow rearrangements of that port in violation of Lemmas 3.3 and 3.4 simply because they drive total costs down. In other words, if we are paid to rearrange

[1] Non-decreasing cost functions have at least one optimal solution in P_v. However there may exist optimal solutions not described by vector rearrangement policies.

some containers it is suboptimal not to perform the maximum number of such rearrangements with negative cost. At the expense of computational time the algorithm can be modified to account for negative marginal costs. Since this is not a very realistic possibility, it is not discussed here.

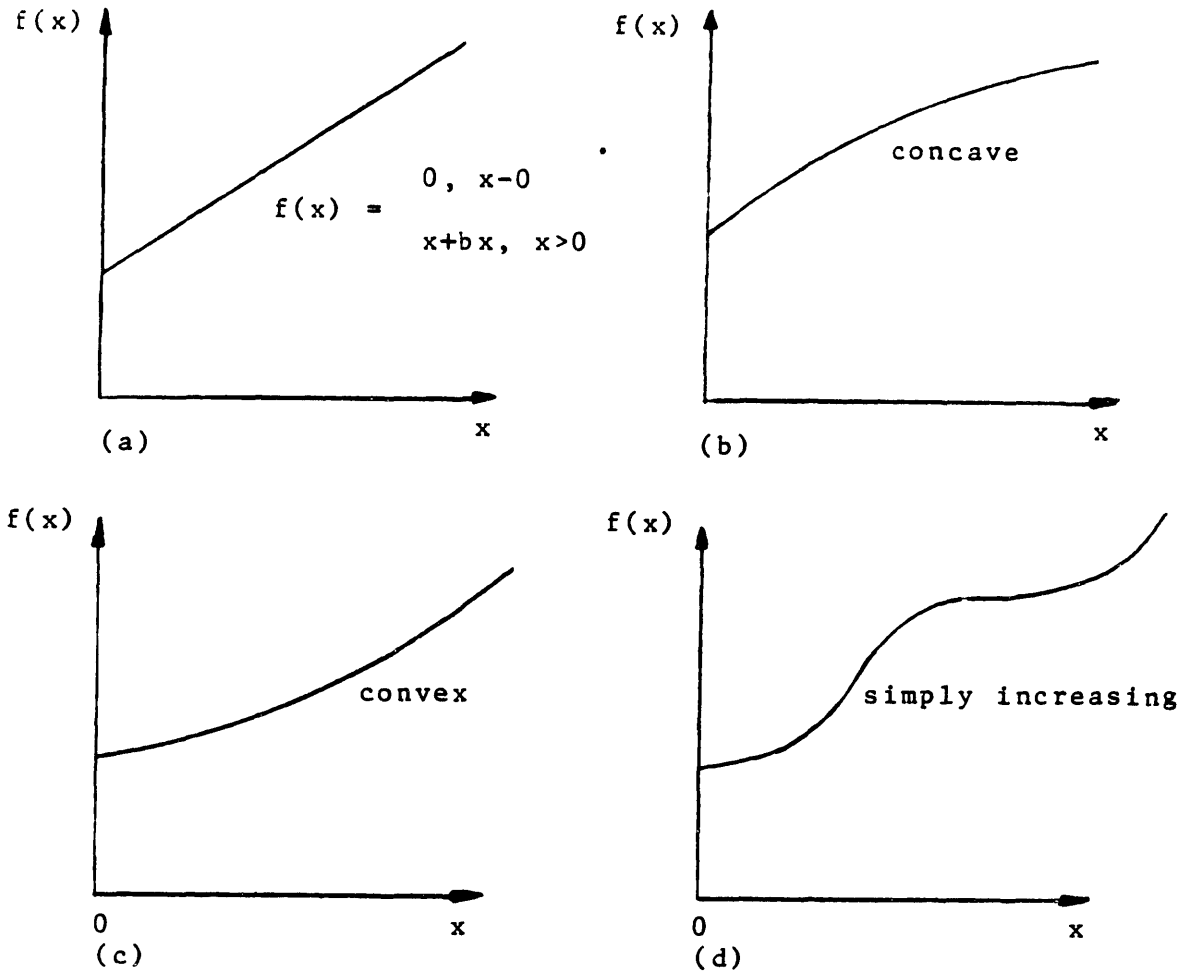


Figure 5.1 - Rearrangement Cost Functions

5.2 Relaxation of the Initial Condition Assumption

Our solution to the single stack overstowage problem has been based so far on the assumption that the stack is empty at port 0, or equivalently, is "in-order" as the vessel arrives at port 1. In this section, we relax this assumption and study how the algorithm should be modified to accommodate situations in which the stack starts her trip in an "out-of-order" condition. An example of such a stack is shown in Figure 5.2. The numbers in the cells denote the destination (i.e. type) of the containers.

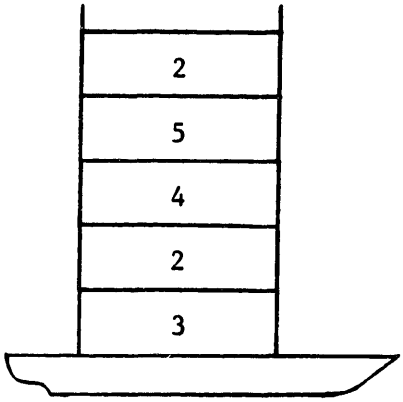


Figure 5.2 - Out-of-Order Stack Arriving at Port 1

It turns out that the method of solving this problem is no different from the method we have already developed. This is because applying constraints on the values that k (the first port of rearrangement up to type j-1) may assume has no effect on the validity (3.14) of Theorem 3.1. Recall that the OSOP was solved by the recursion:

$$V(i,j) = \min_{i+1 \le k \le j-1} \{V(i,k) + r(i,k,j) + V(k,j)\} \qquad (3.14)$$

$$V(i,i+1)=0 \qquad i=0,1,\ldots M;\ j=i+1,\ldots M,M+1$$

If there is some reason such that k cannot take on a particular value, the recursion of (3.14) will still deliver the optimal policy, over the permissible values of K of course. So, even if K can or is forced to take only value, the resulting solution will be optimal.

We can use the latter fact to reproduce a stack profile similar to initial one (as the vessel arrives at port 1). For the stack shown in Figure 5.2, this is done as follows. We create five imaginary ports -5, -4, -3, -2, and -1. At each one of them only one container is ship. Figure 5.3 shows the new expanded series and the new shipment matrix.

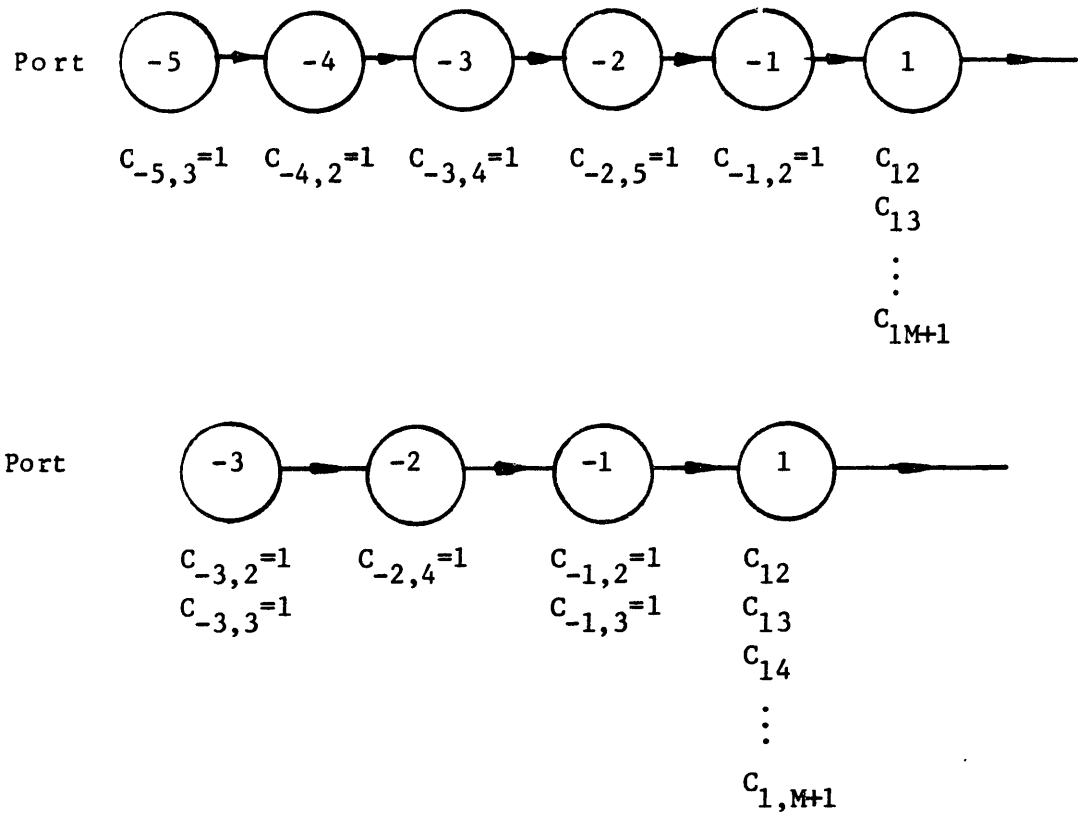


Figure 5.3 - Expanded Port Series

In Figure 5.3 we have introduced as many imaginary ports as the number of containers in the stack. These ports precede port 1 and are given negative numbers. As we have described only one container is picked up at each of the imaginary ports the destination of which corresponds to the destination of the container in the same position in the stack. Let us assume that there are no containers onboard. Then we introduce no imaginary ports. The $(-n_o)^{th}$ port is the origin of the bottom container and so on. Finally, the $(-1)^{st}$ port is the origin of the top container of the initial profile of the stack. Observe that the initial condition corresponds to a "no rearrangement" policy along the imaginary ports. This is feasible since there are not any containers to be delivered at the imaginary ports.

The solution of the problem $PR(-n_o, M+1)$ is the problem that provides the solution to our problem. In fact, we can write

$$V(-n_o,M+1) = \min_{1 \le K \le M} \{V(-r_j,k) + r(-n_o,k,M+1) + V(k,M+1)\} \quad (5.3)$$

In (5.3), K is not allowed to assume any negative value. So there is no need to solve i-started problems for $i = (-n_o+1), (-n_o + 2)....-1$, neither j-condensed problems for $j = (-n_o+1), (-n_o+2)...,-1, 1$. Also observe that all V(-i, -j) are equal to zero since they involved only containers of the same type. That means that we have to solve only O(M) more problems at the form $PR(-n_o,j)$. So, the recursion of (3.14) holds unchanged except from the problem PR(o,j) which is substituted by $PR(-n_o , j)$.

So the total number of PR(i,j) problems remains the same $O(M^2)$. Since each of them requires O(M) comparisons, the recursion runs in $O(M^3)$ time. To calculate the functions r(i,k,j), it takes $O(M^3)$ for all r(i,k,j) with $i \geq 1$ and then it remains to compute those with $i=-n_o$. In particular, we need to calculate those with $i=-n_o$, $k=1$, $j=2,...M+1$. For greater values of K the recursive calculations of section 3.8 can be done in $O(M^2)$ time. Yet $r(-n_o, 1, j)$ are M values and an explicit computation takes $O(n_o.M)$. A more sophisticated computation takes O(M) to compute $a(-n_o,1,j)$ and $b(-n_o,1,j)$, that is $O((n_o+M)M)$ to compute all $r(-n_o, k, j)$. So the overall running time of the algorithm is $O(M^3+(n_o+M)M)$. That is the algorithm slows down only when $n_o = O(M^2)$.

Theorem 5.2:

The OSOP problem with arbitrary initial stack profile can be solved in $O(M^3+(n_o+M).M)$ time by the following recursion:

$$
\begin{aligned}
V(i,j) &= \min_{i+1 \leq k \leq j-1} \{V(i,k) + r(i,k,j) + V(k,j)\} \\
&\qquad i=1,...M;\ j=2,...M+1 \\
V(-n_o,j) &= \min_{1 \leq k \leq j-1} \{V(-n_o,k) + r(-n_o,k,j) + V(k,j)\} \\
V(i,i+1) &= 0,\ i=1,2,...M \\
V(-n_o,1) &= 0
\end{aligned}
\qquad (5.4)
$$

■

Figure 5.3 shows a way to reduce the number of imaginary ports to be introduced, by considering the sequence of containers of the initial profile that are in order. The analysis is the same and since the worst case involves n_o ports, we do not elaborate further.

5.3 Rearrangement Policies with Probabilistic Elements in the Shipment Matrix

Until now we have assumed that the shipment matrix is known with certainty. In this section we make a small (and temporary) deviation and examine optimal policies for cases in which one or some of the c_{ij}'s are known up to a probability distribution function. Of course c_{ij}'s becomes known at port i, at the latest. In many cases also we may revise the probability distribution of c_{ij} at any port before i. With revision we mean either full knowledge of the value of c_{ij}, or simply revision of its probability distribution function (pdf).

For simplicity let us assume that only one element of C is probabilistic, be that c_{ij}, which becomes known with probability 1 at port $K \leq i$. Let $f_{c_{ij}}(\cdot)$ denote the probability distribution function of c_{ij} before port K. Then the best rearrangement policy need have two parts. The first from port 1 to port K-1, and the second, from port K to port M. Clearly the second part is computed at port k after the exact value of c_{ij} is known, and, it also depends on the policy followed up to port K-1, or in other words, it depends on the profile of the stack (state) as the vessel arrives at port K. Then the question is what is the best policy from port 1 to K-1, so that after the calculation of the rest of the policy at port K the overall expected costs to be minimized. We must deal with expected costs since c_{ij} is not known before port K.

Let $P_{1,K-1}$ denote the policy to be followed at ports 1 to K-1 and $R_{P_{1,k-1}}(C)$ the resulting rearrangement cost. Also, let

$$R_{P_{k,M}}(C,\ c_{ij} = c;\ P_{1,k-1})$$

be the rearrangement cost at ports K to M under the rearrangement policy $P_{K,M}$ given that c_{ij} is equal to c and that the policy followed up to port K-1 is $P_{1,K-1}$.

Then, we want to minimize total expected costs, so:

$$R_{P^*_{1,k-1}}(C;\ f_{c_{ij}}(.)) = \min_{P_{1,k-1}} \{R_{P_{1,k-1}}(C) + \int_0^{c_{ij}} R_{P_{k,M}}(C,\ c_{ij} = c;\ P_{1,k-1}) \cdot f_{c_{ij}}(c)dc\} \qquad (5.5)$$

where $R_{P^*_{1,k-1}}(C;\ f_{c_{ij}}(.))$ is the expected cost that corresponds to the optimal policy $P^*_{i,K-1}$, and obviously is a function of C and the probability distribution function of c_{ij}, $f_{c_{ij}}(.)$.

It appears that the solution of (5.5) requires the evaluation of a policy $P_{K,M}$, which depends on $P_{1,K-1}$, and of course c_{ij}. As mentioned above the dependence on $P_{1,K-1}$ is through the profile of the stack right before port K. We have shown in Section 4.2, that there exists an exponential number of states (i.e. profiles) the stack may assume under policies satisfying Lemma 3.4. This is discouraging because it means that there is no efficient algorithm that solves the problem.

However,it is worthwhile to look for approximation schemes that run faster. One such scheme is based on the assumption that the entire policy vector is to specified a priori, and only the part from port 1 to (K-1). Of course as soon as new information comes, at port K, the policy should be reoptimized from then on. But since we look for the full (1 to M) policy the overstowage cost can be written as

$$\begin{aligned} R_p(C;\ f_{c_{ij}}(.)) &= \sum_{m=0}^{M} \sum_{l=m+1}^{M+1} f_{ml}\ c_{mc} \\ &= \sum_{m=0}^{M} \sum_{l=m+1}^{M+1} f_{ml}\ c_{ml} + f_{ij}\ c_{ij} \\ &\quad (m \neq i \text{ and } l \neq j) \\ &= F_{ij} + f_{ij}\ c_{ij} \end{aligned} \qquad (5.6)$$

But now,

$$E[R_p(C;\ f_{c_{ij}}(.))] = \int_{c_{min}}^{c_{max}} (F_{ij} + f_{ij} \cdot c_{ij})\ f_{c_{ij}}(c_{ij})\ dc_{ij}$$

$$= F_{ij} + f_{ij}\ E(c_{ij}) \qquad (5.7)$$

that is c_{ij} can be replaced by its expected value in the calculation of the expected rearrangement cost under any vector-described policy. The same should hold under the optimal policy, which means that we can find the optimal policy if we use the expected value of c_{ij} and run the deterministic OSOP algorithm.

Because of the linearity of any given policy (including the optimal), the above result holds independently of the number of probabilistic elements. Of course as soon as an element becomes known we reoptimize and revise the entire policy. To be exact we should reoptimize every time the probability distribution (expected value) of an element changes. In fact the latter comment is general and refers to the exact case, too. The approximation scheme overestimates the expected cost. This can be seen in Figure 5.4.

The following theorem summarizes our discussion of approximation schemes.

Theorem 5.5

The probabilistic one stack overstowage problem under the assumption that the entire policy vector must be determined a priori, can be solved by substituting the probabilistic elements by their expected values and using the deterministic OSOP algorithm. The policy is reoptimized every time new information arrives. The

running time of this approach is $O(K.M^3)$, where K stands for the number of times of reoptimization. ∎

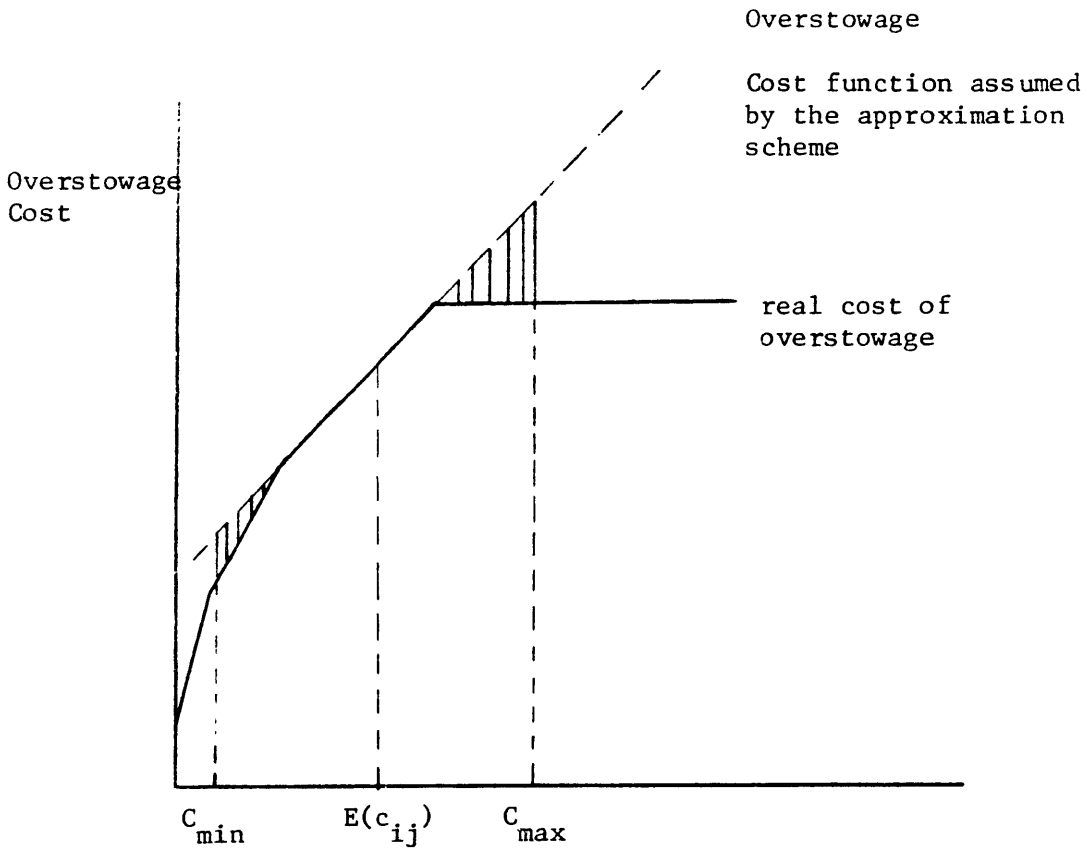


Figure 5.4

Linear Approximation to Overstowage Cost Function

5.4 Rolling Port Horizon Case

A direct application of the results of this chapter is the case in which there is no preset final port, but new destinations come up as the vessel advances. The

basis strategy of our analysis is to explain all available information and develop a policy that minimizes expected cost for the known port horizon. The approximation scheme of the previous section can be also applied to accelerate computational. Every time new information comes in, we reoptimize to take it into account. The current state of the stack serves as initial condition. If n_o is the number of containers on board it takes $O(M^3+(n_o+M)M)$ to reoptimize assuming that there are (M+1) ports yet to be visited.

CHAPTER 6

THE ONE-STACK OVERSTOWAGE PROBLEM WITH PLACEMENT CONSTRAINTS

It is interesting to introduce placement constraints in solving the single stack overstowage problem. This is again a deviation from our theme in this thesis, that is the solution of the overstowage problem without placement constraints. However, it is useful to see how such constraints can be implemented, particularly in the single stack case for which a complete analytical solution is possible.

The only placement constraint which is relevant to the one stack case is the stability constraint. This constraint refers to the center of weight (hence, c.o.w.) of the containers in the stack. In ship operations, this is the well known GM requirement, that is the c.o.w. of the stack should be low enough (and so, GM is high enough) so the vessel does not capsize.

It is clear that we must now treat each container separately, because each one has its own weight. Let n_i be the number of containers to be picked up at port i

$$n_i = o(i) = \sum_{k=i+1}^{M+1} c_{ik} \qquad (6.1)$$

Also let n be the total number of containers handled.

$$n = \sum_{i=0}^{M} n_i = \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} c_{ij} \qquad (6.2)$$

Our analysis will be developed in two stages. In the first stage we assume that there is only one type of container (that is there is only one destination port). This simplified case enhances our understanding of the problem and paves the way for the second stage in which an analysis of the complete single stack overstowage problem with stability constraints is attempted.

6.1 Stability Constraints: The One-Stack One-Destination Case

In this section we deal with the case in which all the containers have the same destination port. This assumption makes the overstowage problem, in the form discussed up to this point, disappear. The only concern now is to satisfy the stability requirement that the c.o.w. of the stack be below a permissible level. Rearrangements of containers may be necessary if these constraints are not met. The topic of this section is to develop a systematic procedure to minimize the number of container rearrangements to satisfy the stability constraints.

Since this is an operational constraint it has to be satisfied at each port. Let r_i be the maximum permissible value for the c.o.w. of the stack upon her departure from port i. The value of r_i depends among others on the displacement of the vessel, her volume distribution under the waterline, her vertical distribution of weight and the minimum required metacentric height (GM). In this section we ignore how the r_i's are calculated and we assume them as given. Let also x_i denote the vertical c.o.w. of the stack at the time of departure from port i. Then the stability constraints can be written as

$$x_i \leq r_i,\ i = 1,2,...,M \qquad (6.3)$$

It is quite obvious that since (6.3) must be observed, it is very rational to place the heavier containers among those picked up at each port, lower positions in the stack. If we follow this policy from the first port, where the stack is assumed empty, then the profile of the vessel in terms of the weight distribution looks as in Figure 6.1.

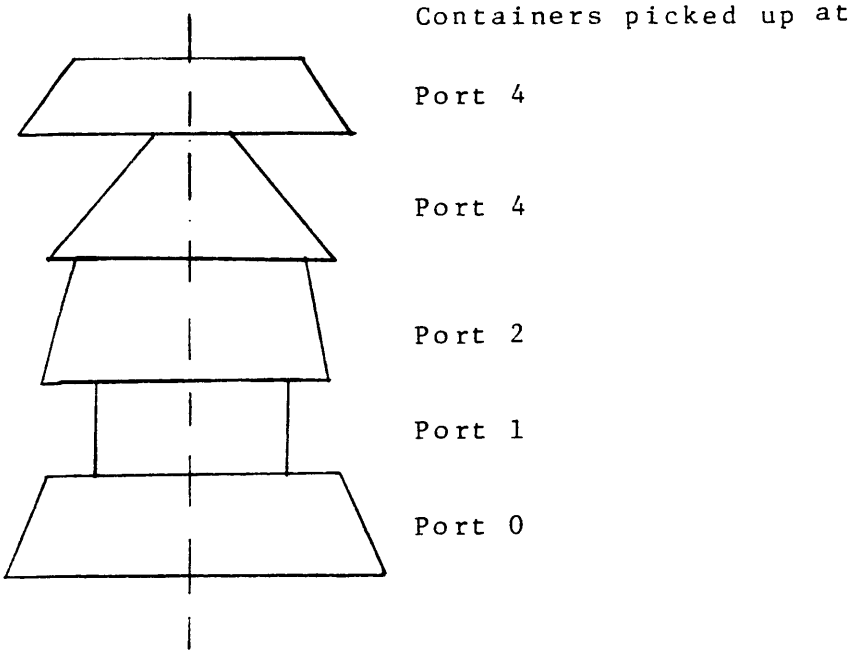


Figure 6.1

Vertical Weight Distribution of Stack After Port 4

The above policy does not result in a single container rearrangement as long as constraints (6.3) are satisfied. If some of (6.3) are not met, then we must move heavy containers to lower positions to satisfy them.

Since all containers of the stack have the same destination, the effect of rearrangements made at early ports of the sequence (that is the decrease in x_i) is observed at later ports, too. For example, it may be optimal to perform rearrangements at port 2 in order to satisfy the stability constraint at port 5. In other words, if x_k is greater than r_k and must be reduced by at least g_k ($=x_k - r_k$), it suffices to reduce any combination of the x_i's ($i=1,...k$) by a total of g_k. Of course

we would like to achieve that by doing the minimum number of container rearrangements.

Let us formally define, g_k, the extent to which (6.3) is violated at port k, if no rearrangement is done at any port.

$$g_k = \max(0, x_k - r_k) \ , \ k=1,2,...M \qquad (6.4)$$

We define $b_k(m)$ as the maximum possible improvement (decrease) of x_k, with at most m container rearrangements performed at ports 0,1,2,...,k-1,k.

Also let $\underline{S}_{k,m}$ denote the stack profile at port k, corresponding to $b_k(m)$. $\underline{S}_{k,m}$ is a vector with as many components as the total number of containers onboard (=n). Each component holds the weight of the container placed in the corresponding positions of the stack (the n^{th} component corresponds to the bottom position).

Since the effects of a rearrangement are permanent then $b_k(m)$ can be calculated as

$$b_k(m) = \max_{l} \ \{b_{k-1}(m-l) + f_{km}(l)\} \ , \ m=1,2,...B_k \qquad (6.5)$$

where,

l may assume values from 0 to $L_k = \min\left(m, \sum_{i=0}^{K-1} n_i\right)$

B_k is equal to $\sum_{i=0}^{K-1} (K-1) \ . \ n_i$

$f_{k,m}(l)$ is the improvement in x_k at port k, with additional rearrangements given that m-l rearrangements have been performed at ports 1 to (k-1)

Also we define

$$b_0(m) = 0, \ \forall m \qquad (6.6)$$

Obviously, $b_k(0) = 0$, $k=1,...M$ (6.7)

Let us now show how $f_{k,m}(l)$ can be calculated. Given that we know that m-l rearrangements have been optimally performed at ports 1 to (k-1) (resulting in a total improvement in the c.o.w. of $b_{k-1}(m-l)$, we also know $\underline{S}_{k-1,\,m-l}$. Then, performing l additional rearrangements simply means that we take off from the stack the top l containers, which we place back along with the group of new containers (that is a total of n_k+l) in decreasing order of weight. The difference between the c.o.w. of the stack as calculated above from the c.o.w. that would result, did we not perform the l rearrangements is the value of $f_{k,m}(l)$.

So, the recursive equation (6.5) along with the boundary condition (6.6) can be used to calculate $b_k(m)$, for $k=1,...m$ and $m=0,1,...B_k$. The following lemma summarizes the above. It can be proven by a simple inductive argument which is not presented here.

Lemma 6.1

Functions $b_k(m)$, $k=1,...m$; $m=0,1,...B_k$ are correctly calculated by using the recursive equation (6.5) and the boundary condition (6.6). ∎

Lemma 6.2

The time it takes to compute $b_k(m)$ is $O(M^2n^3)$.

Proof

Let us first compute how much time it takes to calculate one value $f_{km}(l)$. It is not hard to see that this takes $O((l+n_k)^2)$. The latter also includes the time required to construct $\underline{S}_{k,m}$. For given k and (m-l), $f_{km}(l)$ can be computed in

$$O\left(\left(\sum_{i=0}^{k} n_i\right)^2\right)$$

for all l=0,...,Lk. In more conservative terms this bound can be written as $O(n^2)$. There is a total of $\sum_{k=1}^{M} B_k$ functions to be computed, that is a total of $O(M^2.n)$, again by conservative calculations. The overall time to compute $f_{km}(l)$'s is $O(M^2n^3)$.

Solving (6.3) requires $O(M.(M.n).n) = O(M^2n^2)$ time, where O(M) is the different values of k, O(Mn) the different values of m and O(n) the different values of l. So the total time to compute the $b_k(m)$'s is $O(M^2n^3)$. ∎

So, we have a method to compute $b_k(m)$'s. Let us now examine how we can take advantage of this to satisfy those of (6.3) which are violated. The values of g_k, k=1,...M dictate by how much the c.o.w. of the stack should be decreased up to port k. Then, by solving

$$g_k = b_k(m),\ k=1,...,M \qquad (6.8)$$

we find the minimum number of required rearrangements up to port k to achieve the desired reduction of x_k. Let m_k denote this value of m. From (6.6) we find that

$$m_k = b^{-1}_k(g_k),\ k=1,...,M \qquad (6.9)$$

As long as m_k is known, the only thing that remains to be found is how is distributed over the ports 1 to (k-1). This information can be found by retrieving the value of l that achieves the maximum in the corresponding recursive equation (6.5), and then working backwards in a recursive manner. The information about the above mentioned value of l can be stored at the same time (6.5) is evaluated and consequently retrieved in 0(1) time.

If t^*_k stands for the optimal number of rearrangement at each port, k=1,...,m, that satisfy (after performed) constraints (6.1), it must be

$$\sum_{i=1}^{k} t^*_i \geq m_k \ , \ k = 1,\ldots,M \qquad (6.10)$$

Knowing m_k helps reducing the number of states in the next stages of the recursive algorithm (i.e. calculation of $b_{k+1}(m)$). Specifically if we know that m cannot be less than m_k, and consequently l cannot be larger than m-m* we can reduce both the number of values of $b_{k+1}(m)$ to be computed and the number of terms to be compared.

Figure 6.2 presents an algorithm to solve for the above problem.

Algorithm GM-1

Input: - (M+1) groups of n_i containers, i=1, M with destination port M+1
- The weight of each container
- r_i, i=1,...M, the upper limit for the c.o.w. of the stack after port i

Output: - A rearrangement plan expressed in the number of top containers of the stack to be rearranged of each port, so that r_i's are observed. Containers are always pushed onto the stack in decreasing order of weight.

Step 1 - Initialization-boundary conditions

Set $b_o(m) = 0$, m=1,...n
$m_o = 0$
for i=1 to M do
calculate x_i, $g_i = \max(0, x_i - r_i)$;

Step 2 - Basic recursion

For k=1 to M do
for $m=m_o$ to B_k
for l=0 to $B_k - m_o$

compute $f_{km}(l)$;
compute $b_k(m) = \max_{l} [b_{k-1}(m-l) + f_{km}(l)]$
$l_k(m) = l^*$ such that $b_k(m)=b_{k-1}(m-l^*)+f_{km}(l^*)$
$m_k = b^{-1}_k(g_k)$;

Step 3 - Constraint checking

For k=1 to M do
if $g_k \leq b_k(m_{k-1})$ then $m_k = m_{k-1}$;

Step 4 - Distribution of rearrangements

$t_m = l_m(m_M)$
for k = (M-1) down to 1 do
$t_k = l_k(m_{m+1} - t_{k+1})$;

Figure 6.2 - Algorithm GM-1

The analysis of this section proves the following theorem.

Theorem 6.1

Algorithm GM-1 correctly solves single-stack single-destination overstowage problems with stability constraints in $O(M^2n^3)$ time. ∎

A final interesting point about algorithm GM-1 is its space requirements, because we need sufficient space to store all $\underset{\sim}{S}_{k,m}$ vectors. At each stage m is of the order of O(Mn). There are M stages and a total of n containers, so the space required to store $\underline{S}_{k,m}$ is $O(M^2n^2)$. A slight improvement can be achieved by noticing that only the stack profile of the previous stage (port) need to be maintained, so there is no need to store the stack profiles of all the previous stages.

space (storage) requirements for $\underline{S}_{km}$'s down to $O(Mn^2)$.

Functions $f_{k,m}(l)$ assume $O(M^2n^2)$ different values which means a storage space of $O(M^2n^2)$. However we need to store the values of $f_{k,m}(l)$'s only for the "current" k and m. In fact with a recursive calculation we can compute $f_{km}(l)$ in 0(1) space.

Functions $b_k(m)$ assume a total of $O(M^2n)$ values; we need to maintain the values for the functions of the previous stage only; so we need O(Mn) space.

Vectors $\underline{t}$, $\underline{g}$, $\underline{x}$, and $\underline{r}$ take O(M) space. So the following theorem has been proven.

Theorem 6.2

The space requirements of algorithm GM-1 are $O(Mn^2)$. ∎

6.2 The One-Stack Overstowage Problem with Stability Constraints

In this section we examine how stability constraints can be introduced in the general one-stack overstowage problem. This problem has been solved in Chapter 3. Also in the previous section we have introduced the stability constraint to a simplified version of the OSOP in which there exists only one destination port. A polynomial algorithm to solve the latter problem has also been developed.

Our approach in this section is going to be similar to the one in the previous section. In fact, the formulation of the problem in terms of the satisfaction of the stability constraint is going to resemble the one-destination case. This is achieved

through the following steps.

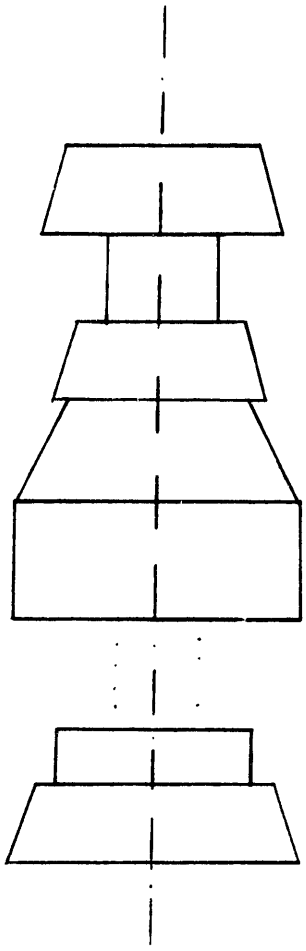
First, we solve the OSOP as if no stability constraint exists. Let $\underline{P}^*$ be the optimal policy and $R^*(C)$ the corresponding minimum rearrangement cost. At this point we perform the transformation of the shipment matrix described in Section 4.3 in reference to policy $\underline{P}^*$. Recall that this transformation is always done in conjunction with a specific rearrangement policy. The result is a shipment matrix C' of which the minimum overstowage cost is zero and the optimal policy is the one with no rearrangements at all ($P^{*\prime}(i)=i$). However, the number of containers shipped under matrix C' is greater than under C by exactly the overstowage cost of the policy P^*.

According to (4.10), if n and n' are the number of containers handled under shipment matrices C and C' correspondingly it holds

$$n' = n + R_{p^*}(C) \tag{6.11}$$

By definition P^*, and consequently, C', correspond to the minimum overstowage cost. Any deviation from what they dictate is going to increase the latter. So we are going to satisfy the stability constraints with the minimum deviation from the above.

Since the transformed shipment matrix results in zero overstowage, the vertical weight distribution of the stack looks as shown in Figure 6.3. Again we assume that containers within each group are placed in decreasing order of weight.



Destination	Origin
K	5 (5 < k)
K+1	5
K+2	4
K+2	3
K+3	3
M	1
M+1	0

Figure 6.3 - Weight Distribution of Stack After Port 5

The definition of r_k, x_k, and g_k is the same as in the one-destination case, that is x_k, k=1,...,M, is the vertical c.o.w. after port k, if no rearrangements are performed under shipment matrix C'. (Remember that from now on we always refer to C'.) The stability constraints can be written again as

$$x_i \leq r_i, \quad i=1,\ldots M \tag{6.12}$$

One of the properties of the one-destination case is that the effect of rearrangements at earlier ports last for the entire sequence. In the multidestination case though, when a group with some intermediate destination is delivered, any improvement in the c.o.w. due to rearrangements involving this group in the previous ports is lost. This is unfortunate because $b_k(m)$ cannot be defined in the same manner as before, since its value depends on what containers of those rearranged are still on board at port k.

As we mentioned a few paragraphs above, there is zero overstowage cost under the shipment matrix C', if the stability constraints are ignored. It can be proven that C' can have a total of (M+1) groups[1] at most. This is so because groups with the same origin have (obviously) different destinations and groups with later origins can have no later destinations than groups from earlier ports. So, the difference "earlier destination-origin" decreases at least by one, as the vessel moves to the next port. The latter is a consequence of the zero overstowage condition (in the absence of the stability constraints).

Figure 6.4 shows how the origin and destinations of the groups may look. What is missing from the figure is groups of the form (i, i+1). We are going to take care of these groups soon; for the time being we ignore them because they are too "temporary", that is they have a very short presence on board (just for one port).

Let us take the example of Figure 6.4 and determine which groups are on

[1] We do not count groups of form (i, i+1).

board along every leg of the sequence. (For simplicity we assume that (M-1) comes directly after 6.)

Leg	Groups
0-1	(0,M+1), (0,M)
1-2	(0,M+1), (0,M), (1,M-1), (1,6)
2-3	(0,M+1), (0,M), (1,M-1), (1,6), (2,6), (2,5)
3-4	(0,M+1), (0,M), (1,M-1), (1,6), (2,6), (2,5), (3,5)
4-5	(0,M+1), (0,M), (1,M-1), (1,6), (2,6), (2,5), (3,5)
5-6	(0,M+1), (0,M), (1,M-1), (1,6), (2,6)
6-(M-1)	(0,M+1), (0,M), (1,M-1)
(M-1)-M	(0,M+1), (0,M)
M-(M+1)	(0,M+1)

As it is evident from the above different groups of containers are present at different legs of the trip, however, if we reorder the legs then it appears as if the vessel only picks up containers having the same destination. For the above example the reordering has as follows.

Leg	Groups On Board
M-(M+1)	(0,M+1)
0-1,(M-1)	(0,M+1), (0,M)
6-(M-1)	(0,M+1), (0,M), (1,M-1)

1-2	(0,M+1), (0,M), (1,M-1), (1,6)
5-6	(0,M+1), (0,M), (1,M-1), (1,6), (2,6)
2-3	(0,M+1), (0,M), (1,M-1), (1,6), (2,6), (2,5)
3-4, 4-5	(0,M+1), (0,M), (1,M-1), (1,6), (2,6), (2,5), (3,5)

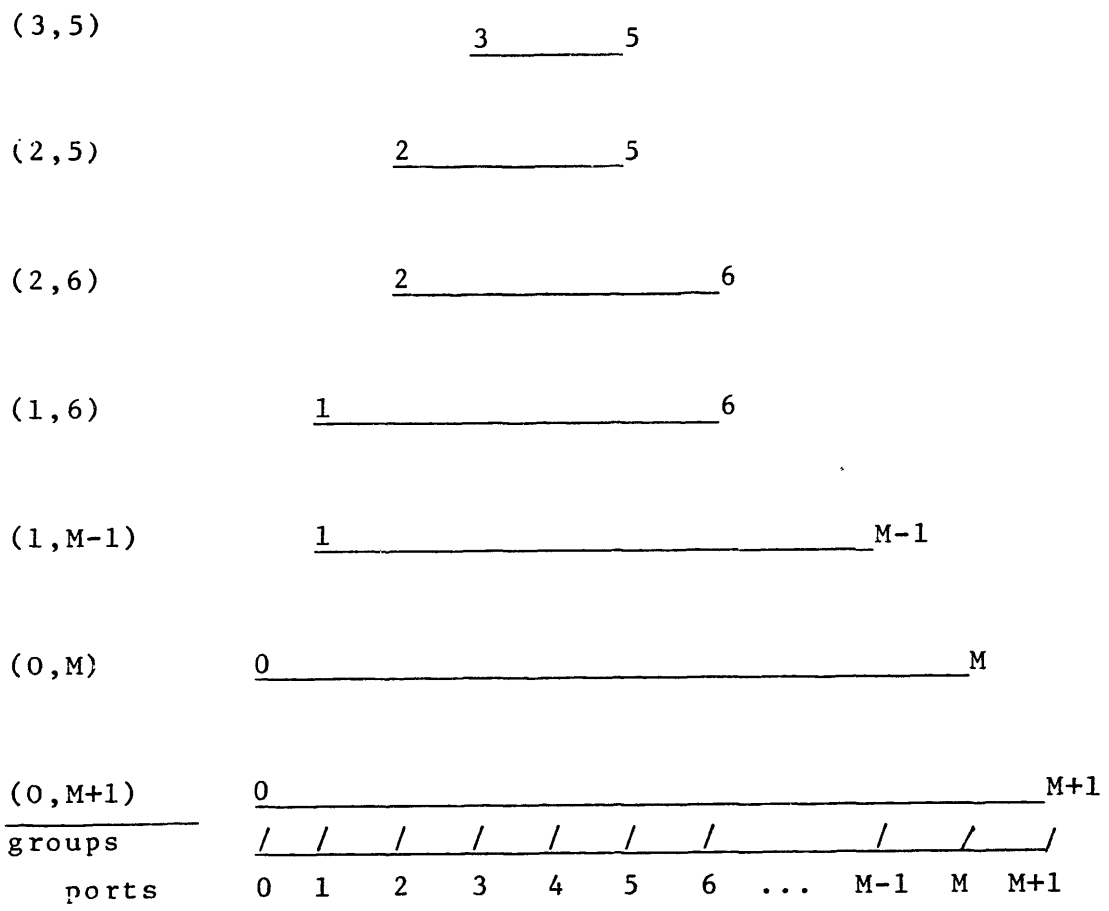


Figure 6.4 - Present Groups in a Zero Overstowage Shipment Matrix

A more careful examination of the above ordering reveals that it is done in an increasing origin-decreasing destination manner of the top group of containers of

the stack in that particular leg. The similarities to the one-destination case become now clear. In principle an equation like (6.5) can be written for this case, too, with the functions $b_t(m)$ similarly defined. There are still some differences though that are not as simple. These are:

(i) The existence of groups of the form (i, i+1), i=0,1,...m, that is groups that stay on board only for one leg.

(ii) The complication in computing functions $f_{t,m}(l)$. This complication is due to the different destinations that some groups of containers may have. For example, in terms of the previous example, if at port 2, we "mix" one - the lightest - container of group (2,6) with group (2,5), it will cost us one rearrangement. If we "mix" one of (1,6)'s with group (2,5), it will cost us two rearrangements - one at port 2 and one at port 5.

From now on we always refer to the reordered[2] trip. In doing so we restore the very important property of the one-destination case that the effects of rearrangements at the early ports last until the end of the trip. The latter is true for rearrangements involving all groups except from groups of (i,i+1) type, the effects of which are short-lasting.

Let us now see how we can account for the two differences from the one-destination case discussed above. We deal first with the second one. In fact, it is straightforward to implement the requirement of different costs in the calculation of functions $f_{t,m}(l)$. We must simply keep track of the origin and destination of each

[2] Except when it is explicitly said otherwise.

container (along with its weight), and, when a container already on board is "intermixed" with the group of new containers (and there is always one such group in the reordered trip), the resulting number of rearrangements is one, if the container and the group have common origin or destination (in terms of the original trip), and, two, otherwise. This becomes obvious by looking at Figure 6.5 below.

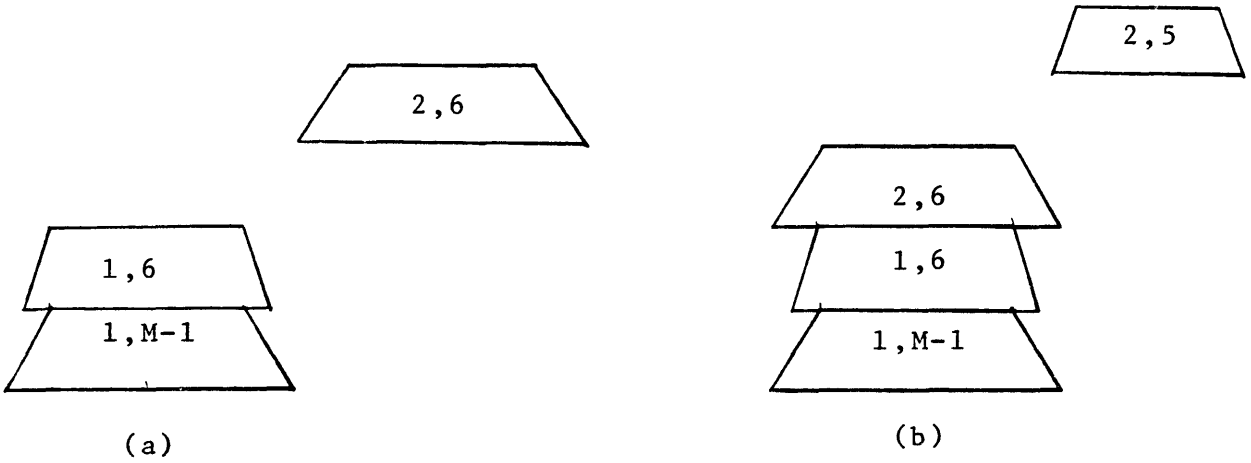


Figure 6.5

Calculation of Functions $f_m(1)$ in the Multi-destination Case

In Figure 6.5a group (2,6) is the newcoming group. If we "mix" one container of group (1,6) with (2,6)'s, we "pay" only for the rearrangement of the (1,6)-container at port 2. If we "mix" a (1,M-1)-container with those of group (2,6) we "pay" for one rearrangement at port 2 and one at port 6 of the (1,M-1)-container.

Even if the (1,M-1)-container is already "mixed" within the (1,6)'s we "pay" for two rearrangements (at port 2 and 6) in addition to the one "paid" at port 1 and which has been accounted for. Similar observations can be made in Figure 6.5b. The above are summarized in Lemmas 6.3 and 6.4 below.

Definition 6.1

The reordered port sequence of an initial one with C' (transformed C under the optimal rearrangement policy) is one in which the vessel picks up the groups of containers in increasing order of origin and, among those with the same origin, decreasing order of destination. It may have up to (M+2) ports. ∎

Lemma 6.3

In the reordered port sequence effects of rearrangements in early ports (i.e. decreases in the c.o.w. of the stack) are maintained throughout the entire initial port sequence. ∎

Lemma 6.4

In calculating the $f_{km}(l)$ functions for the reordering port sequence, we must count twice the rearrangements that do not result in "mixing" (i.e. overstowing) containers of groups with the same origin or destination. ∎

Based on the above results we can define (always in reference to the reordered port series), $b_k(m)$ as the maximum improvement in x_k if m rearrangements are performed, or more accurately charged, along ports 1 to k. Then we can write

$$b_k(m) = \max \{b_{k-1}(m-l) + f_{km}(l)\} , \quad m=1,2,\ldots B_k \qquad (6.13)$$

where now m and l stand for the rearrangement cost (at most twice as much as the number of rearrangements), and

$$l = 0,1,\ldots,L_k = \min(m, 2\sum_{i=0}^{k-1} n_i) =$$
$$B_k = 2 \cdot \sum_{i=0}^{k-1} (k-1)\, n_i, \text{ and,} \qquad (6.14)$$
$$f_{km}(l) = \text{the improvement of } x_k \text{ at port k, with}$$
$$\text{1 additional rearrangement cost given}$$
$$\text{that m-1 has been spent at ports 1 to k}$$

Finally (6.4) and (6.5) still hold; that is

$$b_0(m) = 0, \ \forall m \qquad (6.6)$$
$$b_k(0) = 0, \ k=1,\ldots M \qquad (6.7)$$

So if the groups (i,i+1) are ignored, we can solve the recursive equation (6.13) along with (6.14), (6.6), and (6.7) and get an optimal rearrangement policy which satisfies the stability requirement. The latter process has three possible outcomes.

(a) All stability requirements are satisfied and the solution is optimal even if groups (i,i+1) are considered.

(b) All stability requirements are satisfied but a better (cheaper) solution is

possible if groups (i,i+1) are considered.

(c) Some stability constraints are not satisfied.

In any event, it appears that we should check how much the consideration of groups (i,i+1) affects the solution. This is mandatory in case (c); also, it is worth examining whether and to what extent the optimal solution changes even when the stability constraints are satisfied.

A preliminary intuitive analysis indicates that no significant savings should be expected through rearrangements of the containers of groups (i,i+1), simply because the effect of their rearrangement lasts only for one leg of the trip, while rearrangements of the other groups last for the remaining part of the trip (again, referring to the reordered port sequence). Figure 6.6 shows the g_i's as well as the resulting change in x_i, $d(x_i)$, after rearrangements among containers of groups other than (i,i+1) are performed. The figure refers to the "original" port series.

A methodology to go about using some of groups (i,i+1) to satisfy the stability constraints is desired below. the main idea is to reduce one or some of the g_i's that are positive; let $d(g_i)$ be such a decrease in one g_i. then we solve the problem without groups (i,i+1) but with reduced g'_i ($=g_i-d(g_i)$). Any remaining required decrease in x_i's, if any, is going to be provided by local rearrangements of groups (i,i+1). The following definition is going to make notation simpler.

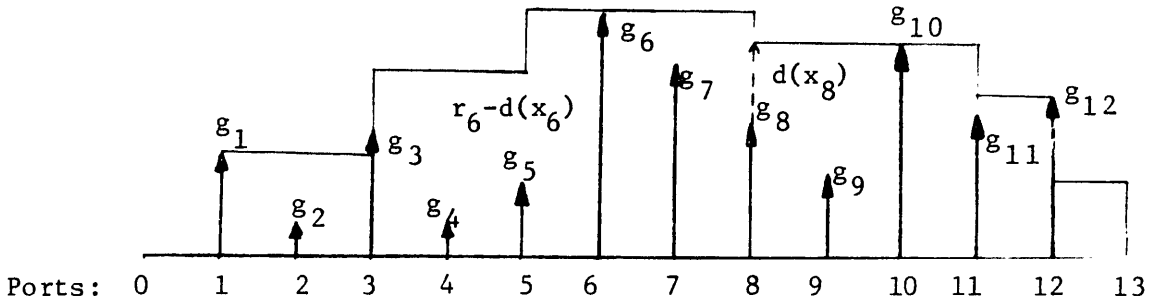


Figure 6.6

r_i's and Resulting dx_i's, After Solution Without

Groups (i,j+1) - Original Port Series

Definition 6.2

Groups (i,i+1), i=0,1,...M are called one-leg groups. ■

Let $C_k(m)$ be the cost of satisfying the stability constraints in ports 1,2,...k with m rearrangements performed on containers of groups other than one-leg groups. That is $C_k(m)$-m is the minimum number of rearrangements involving one-leg groups that are required to satisfy the stability constraints. Then we can write:

$$C_k(m) = \min_{\substack{k-1 \\ 0 \le l \le \sum_{i=0} r_i}} \{C_{k-1}(m-l) + l + h_k(m-l,l)\} \qquad (6.15)$$

where

$h_k(m-l,l)$ is the cost of rearrangements involving group (k,k+1) of port k in order to satisfy the

stability constraint, given that (m-l) rearrangements not involving one-leg groups have been performed at ports 1 to (k-1) and l at port

and $h^*_u(m)$ corresponds to the value of l that achieves the minimum.

The stack profile as the vessel arrives at port k is known, since we have assumed that (6.15) has been solved up to port (k-1). Then, knowing the number of rearrangements of containers of groups other than (k,k+1), we can construct the profile of the stack as leaving port k. Yet we have not considered any rearrangements of containers of the (k,k+1) group. In fact, we are going to perform as many of those as required to satisfy the stability constraint at port k. If the constraint is already satisfied, then $h_k(m\text{-}l,l) = 0$. If the cannot be satisfied then $h_k(m\text{-}l,l) = \infty$. In all other cases, $h_k(m\text{-}l,l)$ is the rearrangement cost corresponding to the rearrangements required to achieve the desired remaining decrease in x_k.

At port 0, where there is only one group on board, there is no way to change x_i by rearrangement; so the boundary condition for $c_k(m)$ is

$$C_1(m) = m+h_1(0,m)\ ,\ m=0,1,...,n_o \qquad (6.16)$$

Equations (6.13) and (6.14) along with the way of calculating $h_k(m\text{-}l,l)$ constitute a recursive algorithm, that evaluates the minimum rearrangement cost to satisfy the stability constraints of the full one-stack overstowage problem. It is interesting to notice that this recursion is the "dual" of the one presented in algorithm GM-1. The former minimizes rearrangement costs, while the latter maximizes

rearrangement benefits.

We must deal now in a more systematic way with the transformation of the original to the "reordered" port sequence. The recursive algorithms developed above refers to the "reordered" port sequence which, as we mentioned earlier, can contain up to (M+2) ports. However, if the shipment matrix contains less than (M+1) groups other than one-leg groups the "reordered" port sequence will consist of less than (M+2) ports. The example on page 117 represents such a case. Two legs of the original series, 0-1 and (M-1)-M correspond to the same leg of the "reordered" port sequence. If, for some reason, r_o and r_{M-1} of the original series are different, then we use the maximum of them in running (6.13), (6.14), (6.6) and (6.7) - that is when the one-leg groups are ignored. This is sufficient, since the profile of the stack is the same in the two legs and we satisfy the stronger requirement. In the case that more than two legs share the same leg of the "reordered" port sequence, the maximum r_i is again used.

The above requirement translates slightly differently in the "dual" recursion, (6.15) and (6.16). Since $h_k(m\text{-}l,l)$ measures the cost to satisfy the stability constraint, we must account for its satisfaction along each leg of the trip. That is, again in terms of the example on page 117, we must account for the cost of satisfying the stability constraints along leg 0-1 and (M-1)-M. These costs may well be different, because (i) r_o and r_{m-1} may be different, and/or (ii) one-leg groups (0,1) and (M-1, M) may contain different number and weight of containers. The above observations are taken into account by replacing (6.13) by

$$C_k(m) = \min_{0 \le l \le L_k} \{ C_{k-1}(m-l) + l + \sum_{j=1}^{J_k} h^j_k(m-l,l) \} \qquad k = 2,3,\ldots M' \tag{6.17}$$

where

J_k is the number of legs of the original trip that correspond to the k^{th} leg of the "reordered" port sequence, and (6.16) by

$$C_1(m) = m + \sum_{j=1}^{J_1} h^j_1(0,m), \quad m=0,1,\ldots n_0 \tag{6.18}$$

If M' is the last port of the "reordered" port sequence, then the optimal solution is

$$C_{M'}(m^*) = \min_m \{C_{M'}(m)\} \tag{6.19}$$

Recursive equations (6.17), (6.18), and (6.19) obviously solve the one-destination problem. This can be seen easily if we consider only groups of type (i, M+1), i=0,1,...M. Then we observe that no reordering of the port sequence is required, and, that all rearrangements contribute one unit to the rearrangement cost (because all the containers have the destination) as it happens with the one-destination problem. Of course, in the latter, there are no one-leg groups. This does not mean that functions $h_k(m-l,l)$ are of no use. In fact, they are used to indicate whether the stability constraints are satisfied. As it was mentioned when $h_k(m-l,l)$ were introduced, $h_k(m-l,l) = 0$ indicates that no rearrangements involving containers of group (k,k+1) are required to satisfy the constraints. Also, $h_k(m-l,l) = \infty$ indicates that the constraints cannot be satisfied. So in the special case of

the one-destination problem function, $h_k(m\text{-}l,l)$ takes on two values, 0 or ∞ depending on whether the stability constraint is satisfied or not.[3] Then it turns out that $c_k(m)$ can be equal either to m or infinity. Nevertheless $c_k(m)$ still depends on l through its dependence on $h_k(m\text{-}l,l)$, which depends on l.

After the above remarks that link our two approaches developed in this chapter we formalize our analysis by presenting an algorithm for solving the OSOP with stability constraints. This algorithm is presented in Figure 6.7, and is called GM-OSOP. Theorem 6.3 has been already proven in the preceding analysis.

Theorem 6.3

Algorithm GM-OSOP correctly solves the one stack overstowage problem with stability constraints in $O(M^2n^3)$ time.

Proof

As mentioned the correctness of the algorithm has been proven. We concentrate on the time bound. We have proven that Step 1 (algorithm REARRANGE) takes $O(M^3)$ time, step 2 (algorithm FINDSHIPMENT) takes $O(M^2)$ time, and step 3 also takes O(M log M) time because it is simply an ordering of at most 2M groups. It remains to examine step 4. There can be at most M ports so M' = 0(M). m varies from 0 to B_k, m = O (M.n) - see (6.12).

[3] We rule out the possibility of not having a feasible solution to the problem by assuming that if sufficient rearrangements are done, the stability constraints can be satisfied.

Algorithm GM-OSOP

Input - Shipment matrix C for a given port sequence 0,1,...M,M+1.
- Weights of the individual containers.
- A vessel that can carry containers in one stack only.
- A requirement (r_i) for the c.o.w. of the containers of the stack at each port.

Output - A rearrangement plan that minimizes the total number of container rearrangements due to overstowage and to the constraint for the c.o.w. of the stack.

Step 1 - For the port sequence 0,1,...M,M+1 and shipment matrix C, run algorithm REARRANGE to get the optimal policy P*(C) , in absence of the c.o.w. constraints. Rearrangement cost R*(C).

Step 2 - Run algorithm FIND-SHIPMENT on shipment matrix C for the optimal policy P*(C). Output a shipment matrix C resulting in zero overstowage under the no-rearrangement policy.

Step 3 - Apply the procedure described in Definition 6.1 and reorder the port sequence based on matrix C', (0,1,...M',n.M'+1)

Step 4 - Run recursion (6.15), (6.16), and (6.17) for the "reordered" port sequence to get rearrangement cost $C_{M'}(m^*)$ and the distribution of rearrangements m^*_k, k=1,...M', of groups other than (k,k+1) and $h^*_k(m^*_k)$ of group (k,k+1), k=1,...M'.

Step 5 - Optimal rearrangement plan: P*(C) , superimposed by m^*_k rearrangements of the top containers of groups other than (k,k+1) and by $h^*_k(m^*_k)$ ones involving group (k,k+1).

Overall rearrangement cost: $R^*(C) + C_{M'}(M^*)$

Figure 6.7 - Algorithm GM-OSOP

Finally l goes from 0 to L_k, that is l = O(n). It also takes O(n) time to compute h_k(m-l,l) for a given k, m, l. So the overall time to perform step 4 is $O(M^2n^3)$. Since M<<<, (otherwise there can be no problem), the time GM-OSOP takes to run is $O(M^2n^3)$. ∎

CHAPTER 7

THE MULTISTACK OVERSTOWAGE PROBLEM (MSOP)

Up to this chapter we have dealt with almost every aspect of the single stack overstowage problem. We have developed the basic algorithm (in Chapter 3), examined alternative formulations (in Chapter 4), discussed probabilistic shipment matrices, and finally, solved the same problem in the presence of placement (stability) constraints. In this chapter, we take the bold step of trying to solve multistack overstowage problems.

As it will become evident in the subsequent sections of this chapter, multistack overstowage problems are much harder to solve than their single-stack counterparts. It appears that exact efficient algorithms cannot be developed even for very simplified versions of multistack problems. There are two possible sources of difficulty for that. First, we must now take into account the problem of assigning containers (or groups of containers) to stacks. Since the optimal overstowage cost of the one-stack overstowage problem is a non-linear function of the c_{ij}'s (i=0,...m; j=i+1,...M+1), the assignment problem alone can make the problem very hard. Secondly, another new feature appears. The latter is the possibility of container switches from stack to stack along the trip of the vessel.

The chapter proceeds with the formal definition and classification of multistack problems. A straightforward formulation is provided. Unfortunately the latter does not lead to an efficient algorithm. Then, complexity issues are reviewed and the stage is set for the development of heuristic algorithms, which take place in

Chapter 8.

7.1 Definition and Classification of Multistack Overstowage Problems

The definition of the problem does not differ much from that of the one-stack case. Simply, we now have the flexibility to stack the containers in more than one stack. In addition, we can assume that the stacks are capacitated.

Let us introduce the necessary terminology to describe the characteristics of the problem.

- c_{ij}, i=1,...M; j=i+1,...M+1 is the number of containers shipped from port i to port j
- $C_{(M+1)\times(M+1)}$ is the shipment matrix (lower triangular)
- M is the number of ports of the series where both deliveries and pickups may occur; the series is shown in Figure 7.1
- L is the number of stacks
- b_k, k=1,...L, is the capacity of stack k. In this thesis we assume that b_k=b, k=1,...L, that is all stacks have the same capacity ($1 \leq b$).

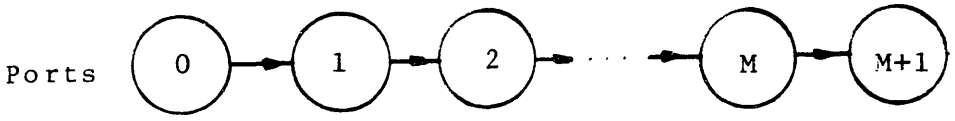


Figure 7.1 - The Port Series

The introduction of stack capacities makes the above model very realistic (warehouse operations, container port terminal operations, doubly-stacked trains, etc.). Furthermore, the existence of finite stack capacities affects the nature of the solution. Thus, it makes perfect sense to distinguish cases depending on whether the stack capacities are finite or not. Also, it is worthwhile examining the relationship between L and M. The following four cases can be identified.

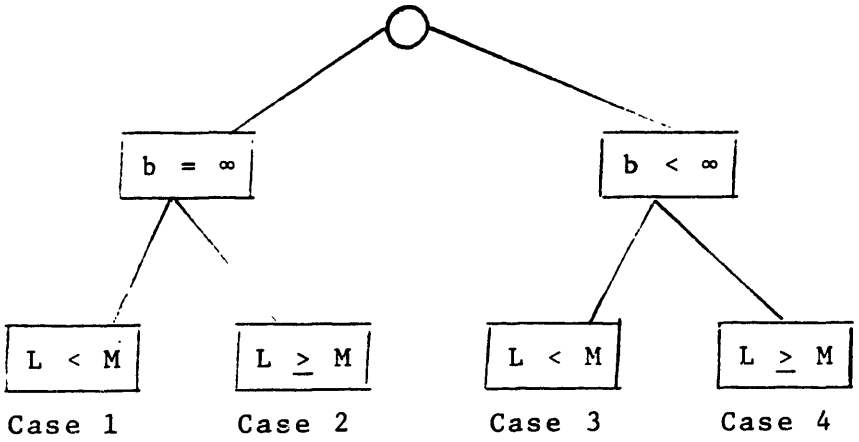


Figure 7.2 - Classification of MSOP

Case 2 is a direct extension of the OSOP to many stacks. It is trivial to see that for $L \geq M$ (Case 2) the problem becomes trivial: one stack can be devoted to each destination and containers with the same destination can always be placed on that stack with zero overstowage.

Cases 3 and 4 do not have any fundamental differences provided that capacity constraint is active. So basically we have two cases to treat: (i) the uncapacitated

case with L<M and (ii) the capacitated case.

The previously mentioned possibility of container switches from stack to stack can be used to provide a finer classification of multistack overstowage problems. We examine versions in which container switches are or are not allowed. In the latter case, a container remains loyal to the stack to which it has initially assigned. This distinction is not unrealistic; it appears when the stacks are physically separated. The latter happens when the containers are to be carried by L one-stack vessels. A more realistic situation arises where there exists a number of vessels to carry the containers. Now, the restriction requirement applies on sets of stacks. One further generalization, as it is discussed in a later section, assumes that the restriction constraint also depends on the port where rearrangements take place. The tabulation of Figure 7.3 classifies the multistack problem and it also introduces a notation to reference the different versions.

	Capacitated	*Uncapacitated*
Unrestricted	*MSOP*	*/MSOP*
Restricted	*R/MSOP*	*R/ /MSOP*

Figure 7.3

Classification of Multistack Overstowage Problems

The generality of the different version is shown in Figure 7.4. The dotted links lead to more restricted versions, while the continuous links lead to special cases

(with b = ∞)

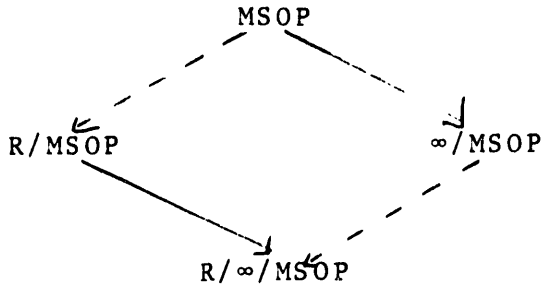


Figure 7.4 - Hierarchy of Multistack Overstowage Problems

7.2 Formulation of the MSOP as a Network Flow Problem with Side Constraints

In this section, we turn our attention to the formulation of the multistack overstowage problem as a mathematical problem. The corresponding formulation for the one-stack case has been described in Chapter 3. One important aspect of such a formulation is how a solution can be described. In the one-stack case the solution, that is the optimal rearrangement policy, has been described as an M-component vector whose components indicate up to what type (destination) of containers we are going to rearrange at each port. That concise presentation is possible due to a series of properties as given in Lemmas 3.1 and 3.3. The situation is quite different in the multistack case though.

First, stacks may have finite capacity. The presence of the latter may result in optimal solutions with containers of the same group assigned to different stacks. Secondly, containers should be assigned to stacks. Even for the restricted version

(R/MSOP) this is an entirely new "feature". In fact, if the assignment part of the problem is resolved, that is if we know what containers - identified by their group - and for what ports are assigned to a stack, then we are able to construct the shipment matrix, to be called effective shipment matrix, which is faced by the stack and for which the one-stack overstowage problem algorithm can be used. Switches of containers from a stack to another are treated as deliveries to that port plus an equal number of pickups from that port with destination the same as those considered delivered. This is the same transformation with the one introduced in Section 4.3. The following example demonstrates that transformation in the current context. Suppose that a container of group (i,j) gets rearranged at port k ($i<k<j$), and in addition is switched from the stack it is on as the vessel enters port k, say stack A, to another one, say stack B. Then the situation is like having a container of group (i,k) in stack A and a container of group (k,j) at stack B.

The above observation reduces the multistack overstowage problem to an assignment problem of containers to stacks at each port. Let us consider a container of group (i,j). It can be assigned to any of the L stacks, at port i. Suppose that k is the first port where it gets rearranged. Obviously up to that port it stays in the stack it has been originally assigned. At port k though it gets off that stack and can be reassigned to any stack, again. Figure 7.5 shows how such a container moves.

It becomes clear from Figure 7.5 that when a container gets rearranged it essentially joins the group of containers of the same type (destination) with that

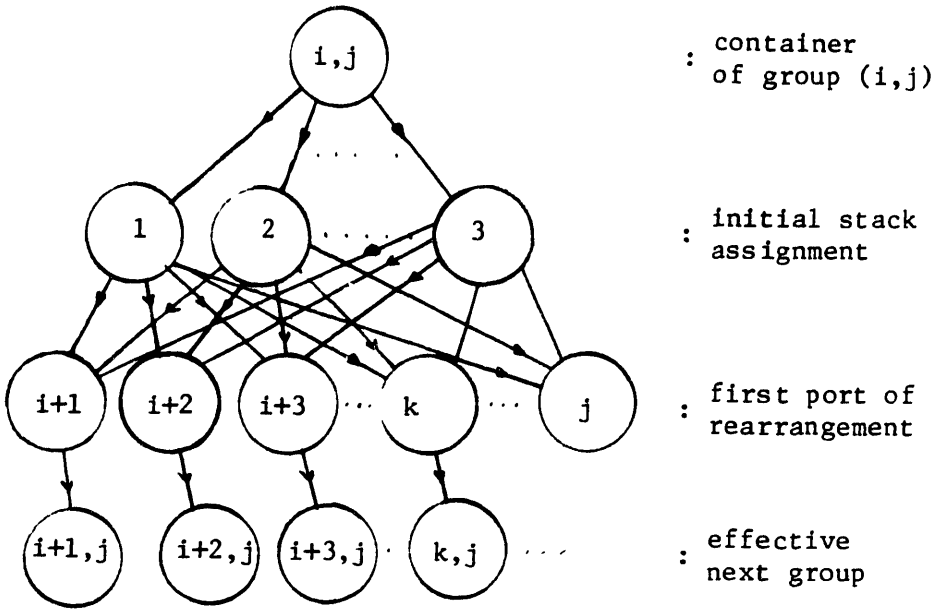


Figure 7.5 - Analysis of Container Assignments

originated at the port of rearrangement. The latter is the "next effective group". Consequently, the "history" of a container onboard can be described by the sequence of ports where it is rearranged plus the stack it gets assigned at each of these ports. This is exactly what Figure 7.5 shows. Every time a rearrangement occurs then the classic, by now, transformation is applied. Reassignments to stacks needs to be made at the port of rearrangement.

Figure 7.6 shows a bigger picture of the model. The network has as many components as destinations, that is M+1. Consider group (i,j) again. c_{ij} is the number of containers of group (k,j) and let c'_{ij} be the number of containers of type

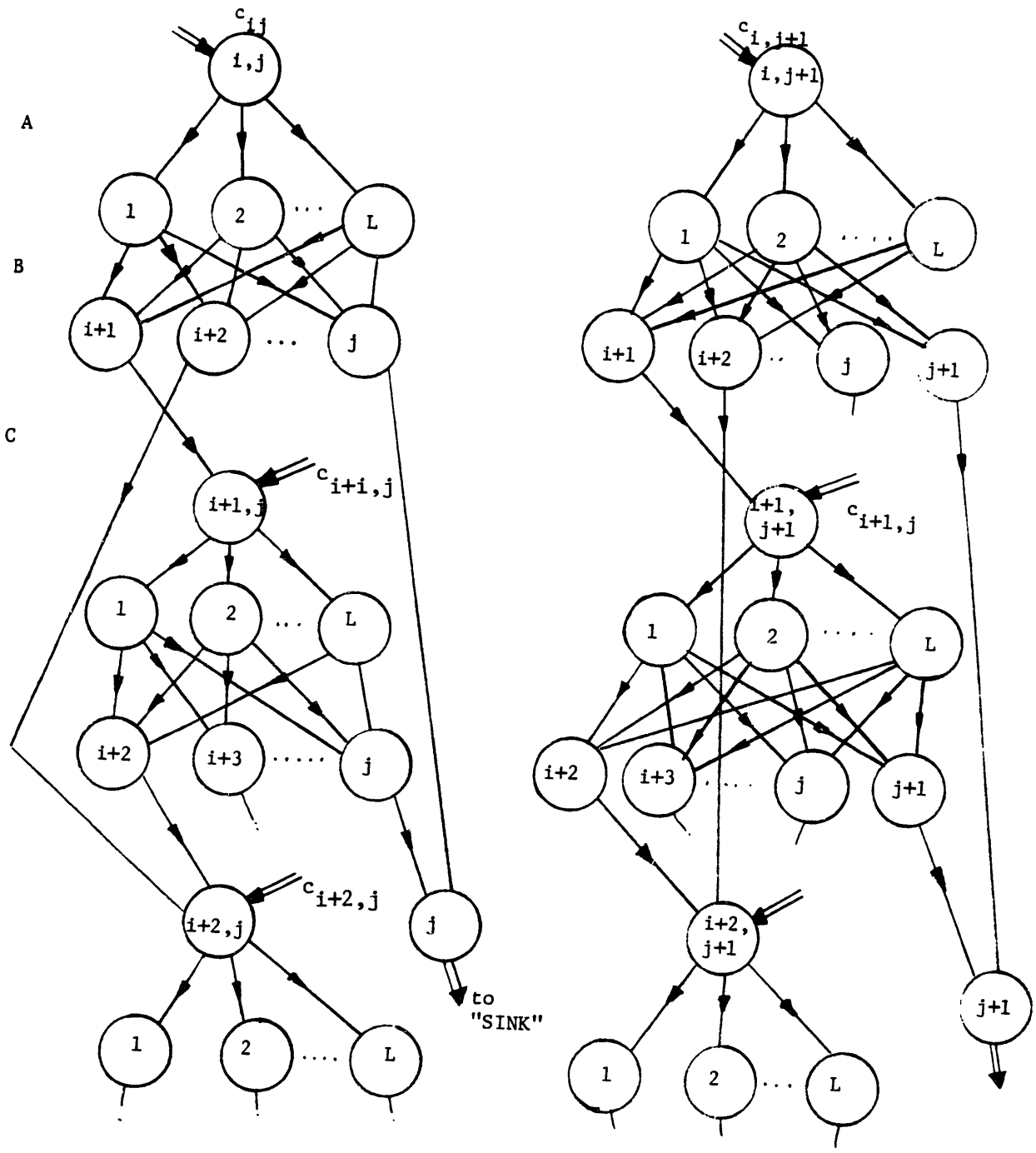


Figure 7.6 - The MSOP Network Formulation

j. With earlier origin which join group (i,j) at port i because they get rearranged. So, a total of (c_{ij} + c'_{ij}) containers need to be assigned to the L stacks. Let the flow of A arcs represent the number of containers of group (i,j) that are assigned to stack k, l=1,...l. Let the flow of B arcs indicate how many containers of group (i,j) from stack l get rearranged at port k (i<k<j). Arcs of type C simply lead the latter flow to the next effective group. Node (j) is the final delivery point and has a demand of ($\sum_{k=0}^{j-1} c_{kj}$) containers. The same happens with all the components of the network.

The objective of the network model is to minimize the flow along the arcs of type C, because the flow of containers along these arcs gives a direct count of the container movements on and off board. Obviously, the flows on all arcs must be integer, otherwise the solution does not correspond to a physical situation. That is, the initial problem has been transformed to a network flow problem. There are two more things to be taken care of. The first is the capacity constraints. The number of containers assigned to a stack at any port cannot be larger than the capacity of the stack. This requirement introduces a number of bundle constraints, L for each port, that is a total of L.(M+1). More importantly, the flow must satisfy another property. This relates to the transformation we perform at ports of rearrangements. As discussed in Section 4.3 (and 6.2) that transformation results in a new shipment matrix which results in zero overstowage. Since the model of Figure 7.6 repeatedly applies this transformation, we must prevent flow on arcs that correspond to groups that give rise to overstowage. For example, if there is

positive from along (i,j)-(1)-(1+2) there cannot be positive flow along (i+i,j)-(1)-(j) because this would result in overstowage. So, we must introduce a set of "either-or" constraints to prevent inappropriate flows. There is a total of $O(M^2)$ "either-or" constraints for each stack.[1] Obviously we need $O(LM^2)$ constraints for all L stacks. Figure 7.6 shows which arcs are grouped for the bundle capacity constraints, and which should satisfy the "either-or" requirement. These two groups of side constraints link the (M+1) components together.

Unfortunately, the above formulation does not lead to an efficient algorithm, although it is the most revealing one. In fact, it appears that this problem is particularly hard in terms of finding the optimal solution, and it is conjectured that no exact efficient (i.e. polynomial-time) algorithm can be found. the latter is the topic of the next section.

7.3 Complexity Analysis of the MSOP

In the previous section we have presented a formulation of the multistack overstowage problem by transforming it into a network flow problem (minimum cost flow) with bundle and "either-or" constraints. The additional requirement of integrality of the solution makes that formulation inappropriate for the development of an algorithm which runs in polynomial time as a function of the size[2] of the

[1] If written in a compact way, that is either $\sum_{i=1}^{\alpha} x_i$ should be zero. α is $O(M^2)$ so the above can be broken down in $O(M^2)$ constraints. Then the total is $O(L.M^4)$.

[2] For a discussion of what is the "size" of the input of a problem see [9], [13].

input. The question now is whether we can formulate the problem in some other way that gives rise to a polynomial time algorithm. In this section we attempt to answer the above question.

Let us consider the simplest[3] version of the MSOP, the R/∞/MSOP. This is the restricted, uncapacitated version. Since no container switches are allowed and no capacity constraints are applied, the only decision to be made is the assignment of containers to stacks. In terms of the formulation of the previous section, we must observe only the "either-or" constraints. Notice that the formulation attempts to solve simultaneously the assignment and overstowage component of the problem. We can separate the two by asking the question slightly differently: "What is the partition of the containers in L sets, that minimizes total overstowage costs?" Then the decision problem (see [9]) can be formally stated as follows.

Definition 7.1: Decision Version of R/∞/MSOP

Input: A series of M+2 ports (0,1,...M,M+1); L stacks with infinite capacity; n containers described by an origin destination port pair; integer number R > 0.

Question: Is there any partition of the n containers into L subsets each corresponding to one of the L stacks, such that each container belongs to exactly one subset and the total overstowage cost of stacking the containers of each subset in a stack is less than R?

■

The overstowage cost of each subset (stack) can be calculated by the one-stack

[3] Not necessarily easiest, too.

overstowage problem algorithm. Then we can easily prove the following.

Theorem 7.1: - The R/∞/MSOP belongs to the class of NP problems.

Proof

If we are given a partition of the n containers into L subsets, then we can compute the minimum overstowage cost corresponding to that partition, by running the OSOP algorithm L times and summing up the results. To run the OSOP algorithm takes polynomial time in terms of the number of destinations, (M+1). So the total running time is $O(LM^3)$. It also takes O(n) steps to construct the effective shipment matrices for all stacks. This makes the running time $O(LM^3+n)$. So, we can use the given partition as a certificate[4], and answer the question of whether the resulting overstowage cost is less than R, in time which is a polynomial function of the input size. Consequently, R/∞/MSOP is in NP. ∎

The decision problem for the R/MSOP is defined in a similar way as for the R/∞/MSOP. Simply the stack (subset) capacity constraint should be satisfied. That is the number of containers among those assigned to a stack, which are simultaneously on board, should be less than or equal to the stack capacity. In terms of proving that R/MSOP we follow the proof of Theorem 7.1. In addition we must show that we can check whether the capacity constraints are satisfied in polynomial time. However, the latter is very easy because we can compute the effective ship matrix for all stacks in O(n) time. Then, it takes $O(M^2)$ to calculate

[4] For a discussion of the theory of NP-completeness, see [9], [13].

how many containers are on board at each port. It takes $O(MM^2)$ to carry out the test for all stacks.

Theorem 7.2: R/MSOP is in class of NP problems. ∎

It is not hard to prove the same results for the general MSOP. Of course now the input should be described in a more explicit way because containers are allowed to switch stacks. That is, we need to know on what stack each container is placed. Consequently, the solution can be described by O(M.n) pieces of information - the stack to which each container is assigned at each port. Then it takes O(Mn) to build the effective shipment matrices for the L stacks and $O(LM^3)$ to run the OSOP algorithm for all the L-stacks and an additional O(L) to carry out the summation to get the total overstowage cost. So, the solution under examination can be used itself as a certificate to check whether the total overstowage cost is less than R. Stack capacity constraints need also be checked. As mentioned above, the latter can be done in $O(L.M^2)$.

Theorem 7.3: MSOP is in the class of NP problems. ∎

We return to the R/∞/MSOP version now. Since it has been proven that it belongs to the class of NP problems, it may belong to one of the three subclasses as shown in Figure 7.7.

Class (P) contains those problems that can be solved by polynomial time algorithms, like the OSOP. The class of NP-complete problems to which all other

problems in NP can be transformed by polynomial time algorithms. For these problems no polynomial time algorithm is known. At the same time, no proof that such an algorithm does not exist is available. If only single NP-complete problems can be solved in polynomial time, then all problems in NP can be so solved. If any problem in NP is intractable, then so are all NP-complete problems. A classic method to prove that a problem is NP-complete is to transform a known NP-complete problem to it by a polynomial transformation.

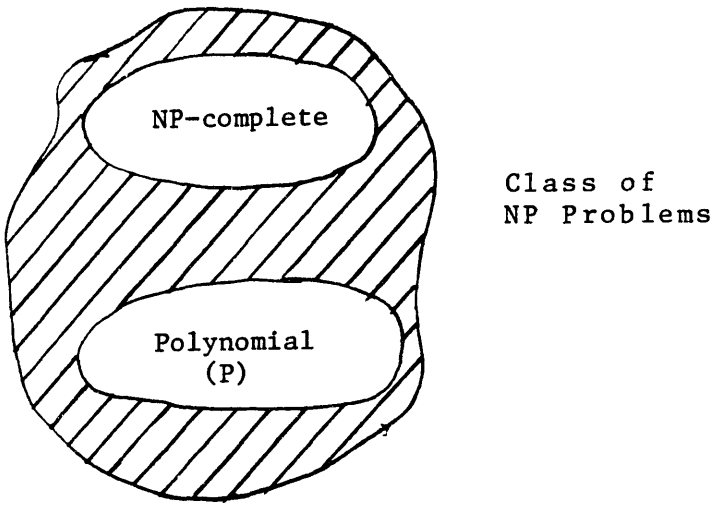


Figure 7.7 - Classification of NP Problems

(Under the assumption P≠NP.)

These subclasses are:

i. the class of polynomially solved problems (P),

ii. the class of NP-complete problems, or

iii. the remaining port of NP (NP-(NP-complete)-P).

Second to showing that **R/∞/MSOP** belongs to the P-class, it would be

interesting to know whether it is an NP-complete problem. This would not answer the question about the existence of a polynomial algorithm for it, but at least it would tie such possibility to the possibility of finding a polynomial algorithm (or proving that no such algorithm exists) for any of the NP-complete problems, for which a vast literature exists.

Although there exists a lot of similarity between R/∞/MSOP and some NP-complete problems, this thesis does not provide a proof for that (nor does it provide an exact polynomial algorithm). We conjecture that the problem is NP-complete. In defining R/∞/MSOP we use an extension of Lemma 3.2, proven again in a latter section of this chapter, which dictates that containers of the same group should all move together in the optimal solution (no group splitting). Alternatively, we can think of the containers of a group as one container with weight (size) equal to the number of containers in the group.

Definition 7.2: R/∞/MSOP

Input: Finite number of groups N, origin (i) and destination (j) of each group, a number $c_{ij} \in Z^+$ for each group, positive integers L and R.

Question: Can the N groups be partitioned in L disjoint sets $N_1, N_2, \ldots N_L$ such that,

$$\sum_{i=1}^{L} \text{(overstowage cost for set } N_i) \leq R \quad ?$$ ■

The complexity of the R/MSOP is associated with that of R/∞/MSOP in the following Lemma 7.1.

Lemma 7.1

If R/∞/MSOP is NP-complete, then R/MSOP is NP-complete, as well.

Proof

The proof is done by restricting the instances of R/MSOP to produce a known NP-complete problem. In this case, this is very simple because by setting B= (infinite stack capacities) we get R/∞/MSOP which has been assumed being NP-complete. Let us now define a generalized version of the multistack overstowage problem. Suppose l_{ij}, i=1,...L; j=1,...M, are binary variables which indicate whether containers of stack i are allowed to switch to other stacks at port j

$$l_{ij} = \begin{cases} 1, \text{ containers of stack i are allowed to switch swack at port j} \\ 0, \text{ otherwise} \end{cases} \qquad (7.1)$$

$$i = 1,...L;\ j=1,...M$$

Then we can define.

Definition 7.4

The multistack overstowage problem (MSOP, ∞/MSOP) along with a set of container stack switching restrictions as defined in (7.1) is called generalized multistack overstowage problem, G/MSOP. ∎

Clearly, if all l_{ij}'s are equal to zero we get the restricted version (R/MSOP, R/∞/MSOP), and if all l_{ij}'s are equal to one we obtain the unrestricted case (MSOP, ∞/MSOP). Now we can prove the following lemma.

Lemma 7.2

If R/∞/MSOP is NP-complete, then G/MSOP is so.

Proof

If we set $b=\infty$ and $l_{ij}=0$ for all, i=1,...L and j=1,...M, then G/MSOP reduces to R/∞/MSOP and the lemma is proven. ∎

The result of Lemma 7.2 does not prove that MSOP is NP-complete if R/∞/MSOP is. However, we conjecture that MSOP is an NP-complete problem.

7.4 Some Results on R/∞/MSOP

In this section we analyze the R/∞/MSOP. We discover certain properties which the optimal solution must have, and we solve a simple case by complete enumeration. Finally, the form of the function of the total overstowage cost as a function of the number of stacks is given.

The first important property that we are going to prove is an extension of Lemma 3.2 and has been mentioned in the previous section too.

Lemma 7.3: An optimal rearrangement policy of the R/∞/MSOP treats containers of the same group the same. ∎

The above lemma can be proven exactly as Lemma 3.2, so the proof is not repeated here. The idea is to attach to the container of the group with the least contribution to the overstowage cost all the other containers of the same group without altering the rearrangement policies of the stacks. Such an action is possible

because the stacks have infinite capacity.

Let us now solve by complete enumeration an instance of the R/ /MSOP in which L=2 and M=3. The shipment matrix is

$$C = \begin{bmatrix} C_{01} & & & \\ C_{02} & C_{12} & & \\ C_{03} & C_{13} & C_{23} & \\ C_{04} & C_{14} & C_{24} & C_{34} \end{bmatrix} \qquad (7.2)$$

In the following we ignore the one-leg groups (i, i+1), i=0,1,2,3, because they do not contribute to the overstowage cost. Also the enumeration below omits cases which are clearly non optimal. So we have the possibilities shown in Figure 7.8.

If we combine the results of all those cases we get for the overstowage cost, the following analytical expression

$$R(R/\infty/L=2,\ M=3) = \min(C_{02},\ C_{03},\ C_{13},\ C_{14},\ C_{24}) \qquad (7.3)$$

Notice that in Figure 7.8, we have examined only 8 out of 32 possible assignments because the suboptimality of the rest was apparent. Our intuition is verified by (7.3) because in no circumstances the overstowage cost could be less than that of (7.3).

If we now solve the one-stack overstowage problem for the same shipment matrix, (7.2), we get (again by enumeration)

$$R/OSOP/M=3) = \min \begin{cases} C_{13} + 2C_{14} + C_{24}, & \text{Policy } (1,2,3) \\ C_{02} + C_{14} + C_{24}, & (2,2,3) \\ C_{02} + C_{03} + C_{24}, & (3,2,3) \\ C_{03} + C_{13} + C_{14}, & (1,3,3) \\ C_{02} + 2C_{03} + C_{13}, & (3,3,3) \end{cases} \qquad (7.4)$$

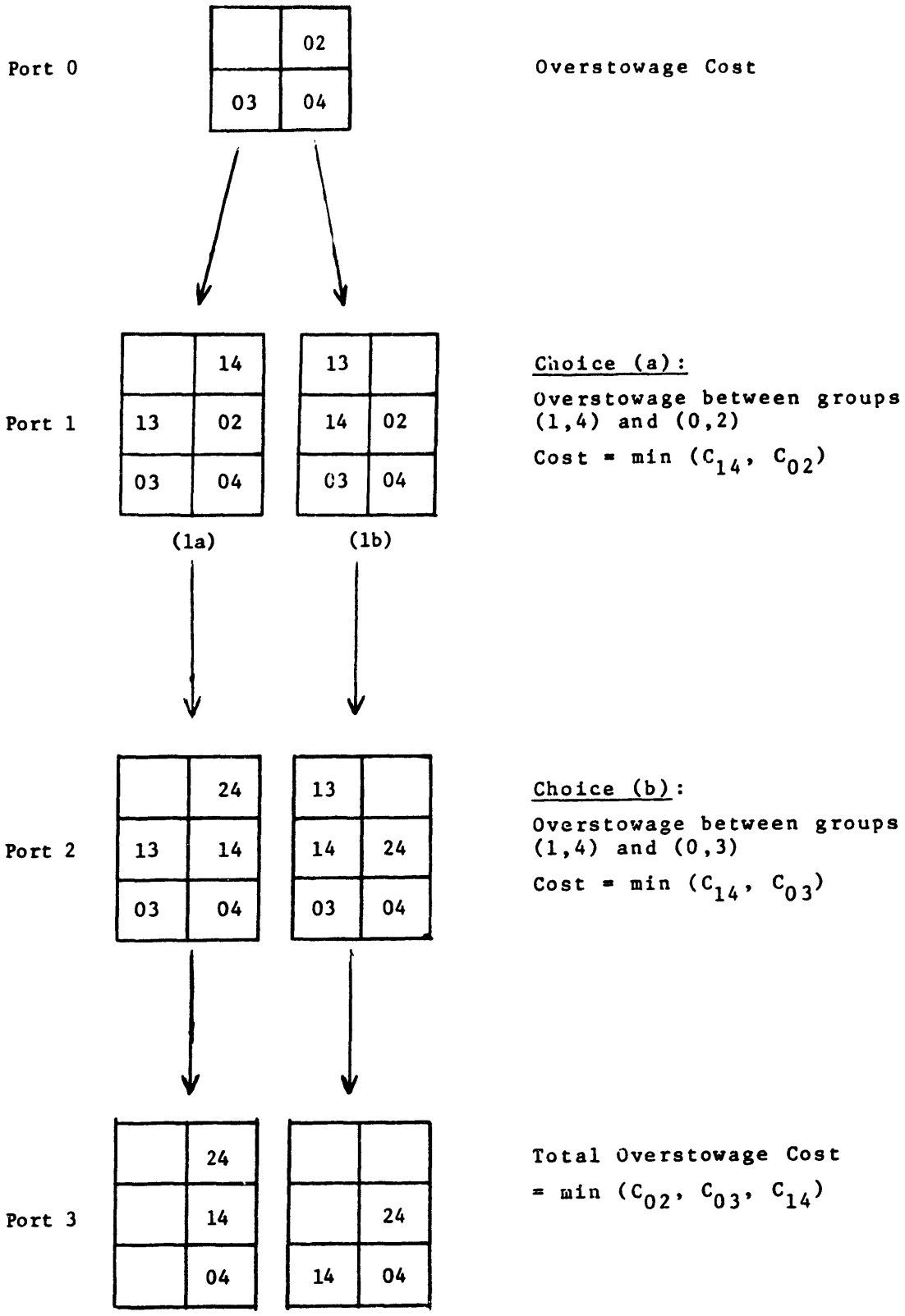


Figure 7.8(i)
R/ /MSOP, L=2, M=3
(Stacks are shown with no rearrangements.)

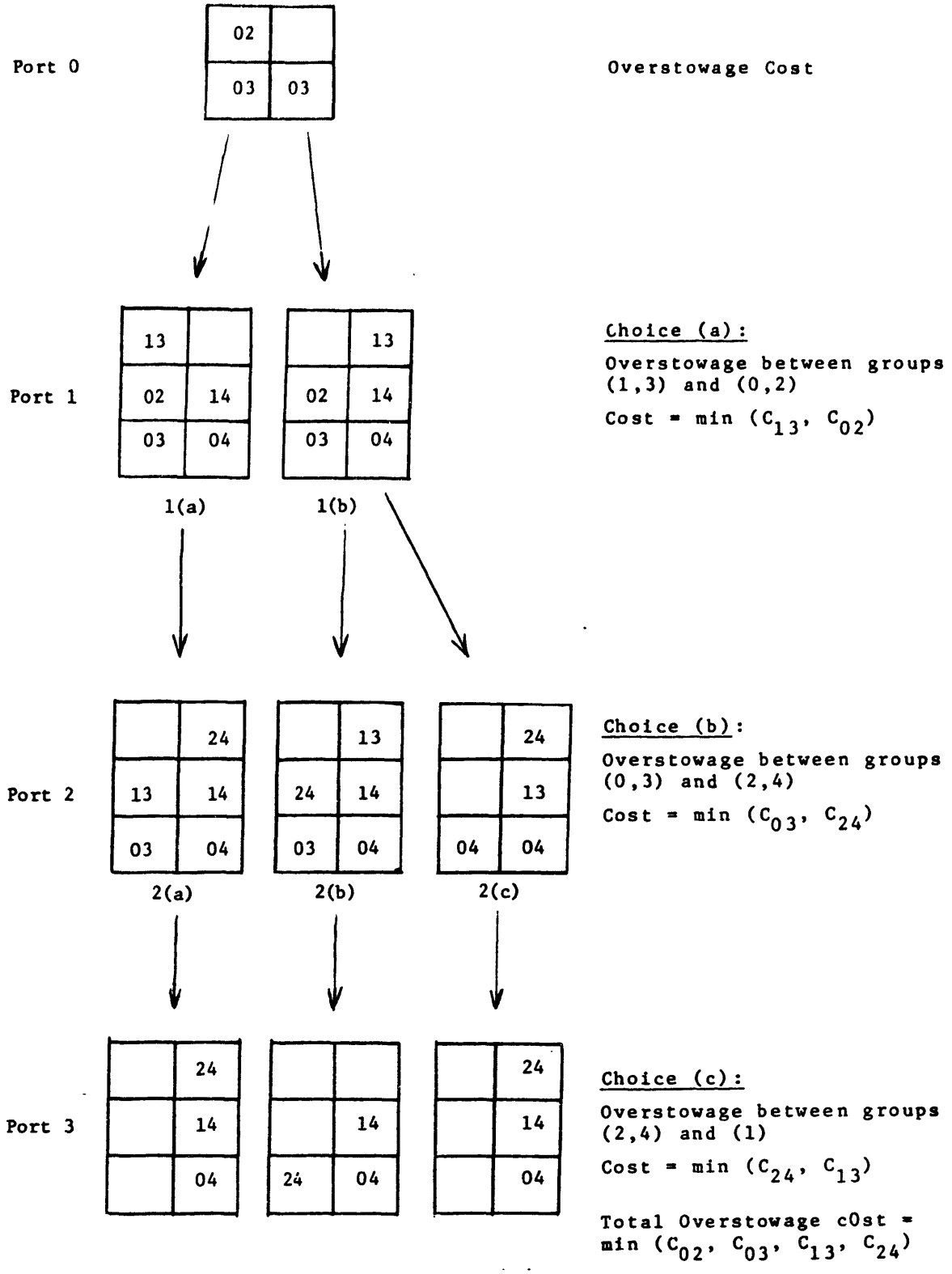


Figure 7.8(ii)
R/ /MSOP, L=2, M=3
(Stacks are shown with no rearrangements.)

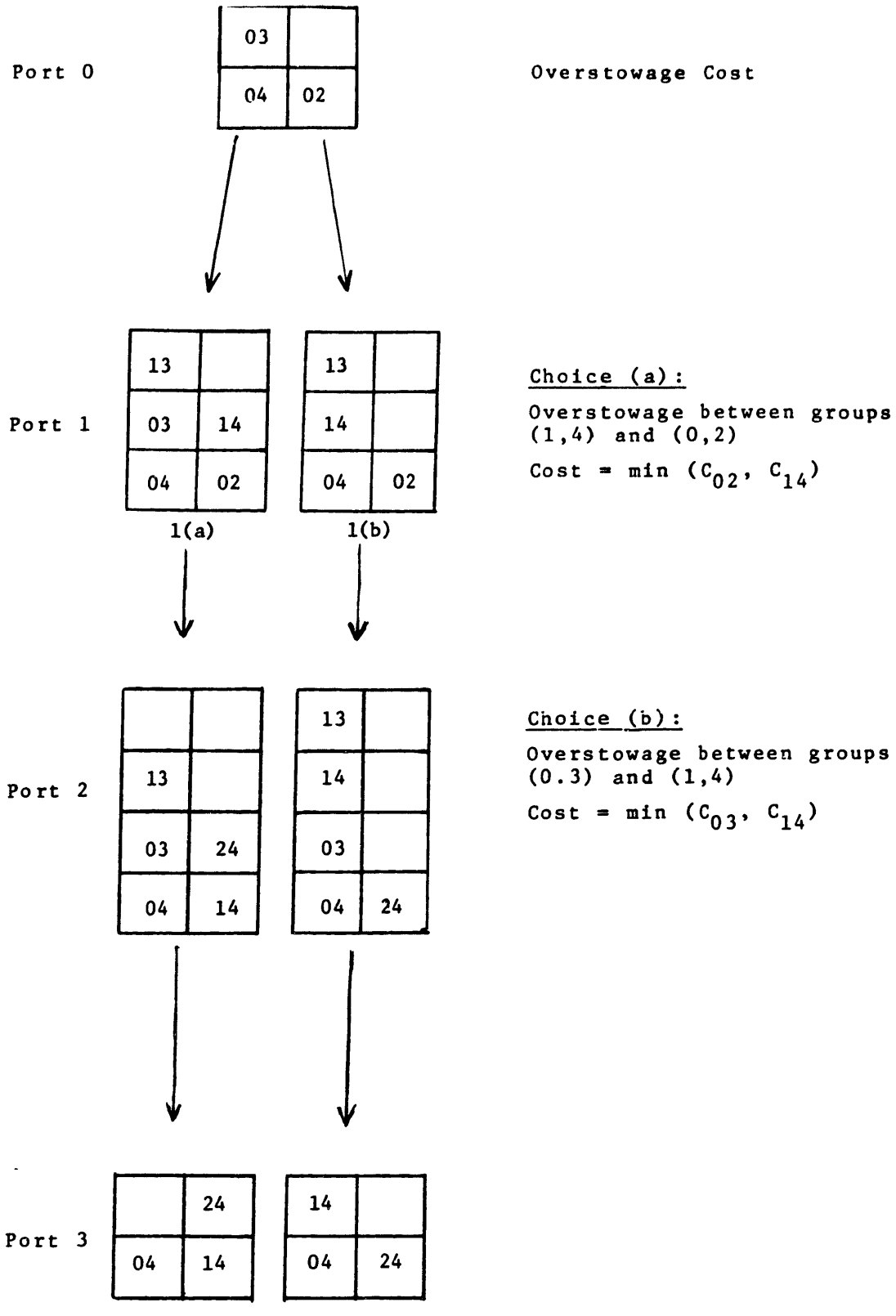


Figure 7.8(iii)
R/ /MSOP, L=2, M=3
(Stacks are shown with no rearrangements.)

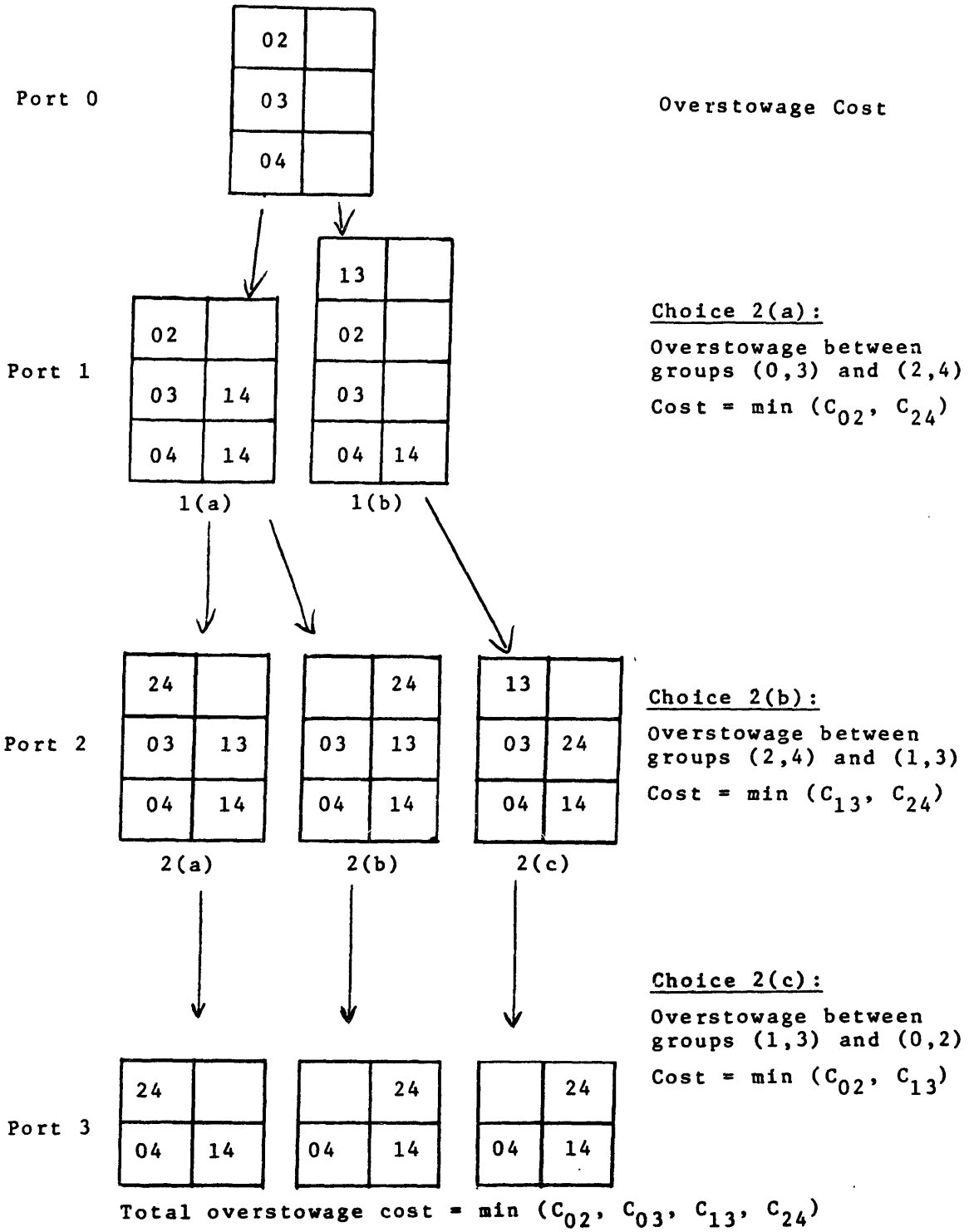


Figure 7.8(iv)
R/ /MSOP, L=2, M=3
(Stacks are shown with no rearrangements.)

From (7.3) and (7.4) we conclude that

$$R(R/\infty/L=2,\ M=3) \leq 1/3\ R(OSOP/M=3) \qquad (7.5)$$

that is, the overstowage cost for the two stack case is less than the one-third of the cost of the one-stack problem.

The solution of the three stack problem with M equal to 3 results in no overstowage. So, the overstowage cost as a function of the number of stacks, for M=3, is as shown in Figure 7.9.

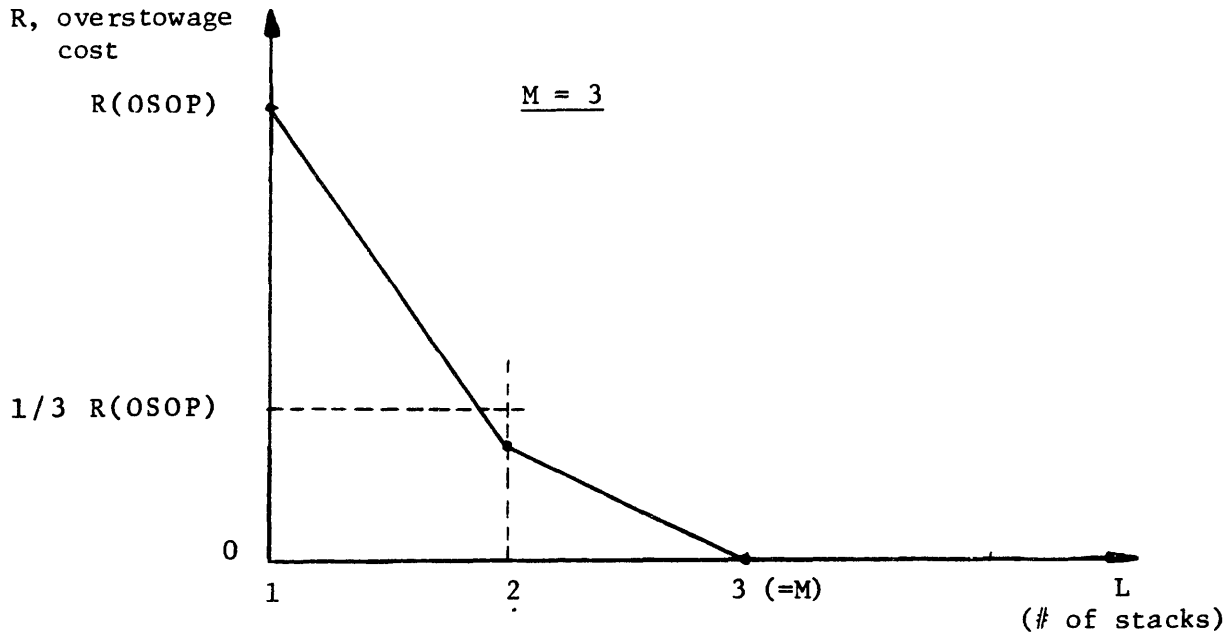


Figure 7.9

Overstowage Cost as a Function of the Number of Stacks M=3

A similar diagram can be drawn for any M. In general, the overstowage cost should be a decreasing function of the number of stacks. This is obvious since the

optimal (minimum) overstowage cost can only decrease if an additional stack becomes available. In fact, it seems intuitive to expect that the marginal gain from the introduction and use of an additional stack decrease as the number of stacks becomes larger. Although formal proof of the latter is not provided in this text, there should be extremely unusual conditions under which this does not hold true (if such conditions exist at all). Figure 7.10 is a generalization of Figure 7.9.

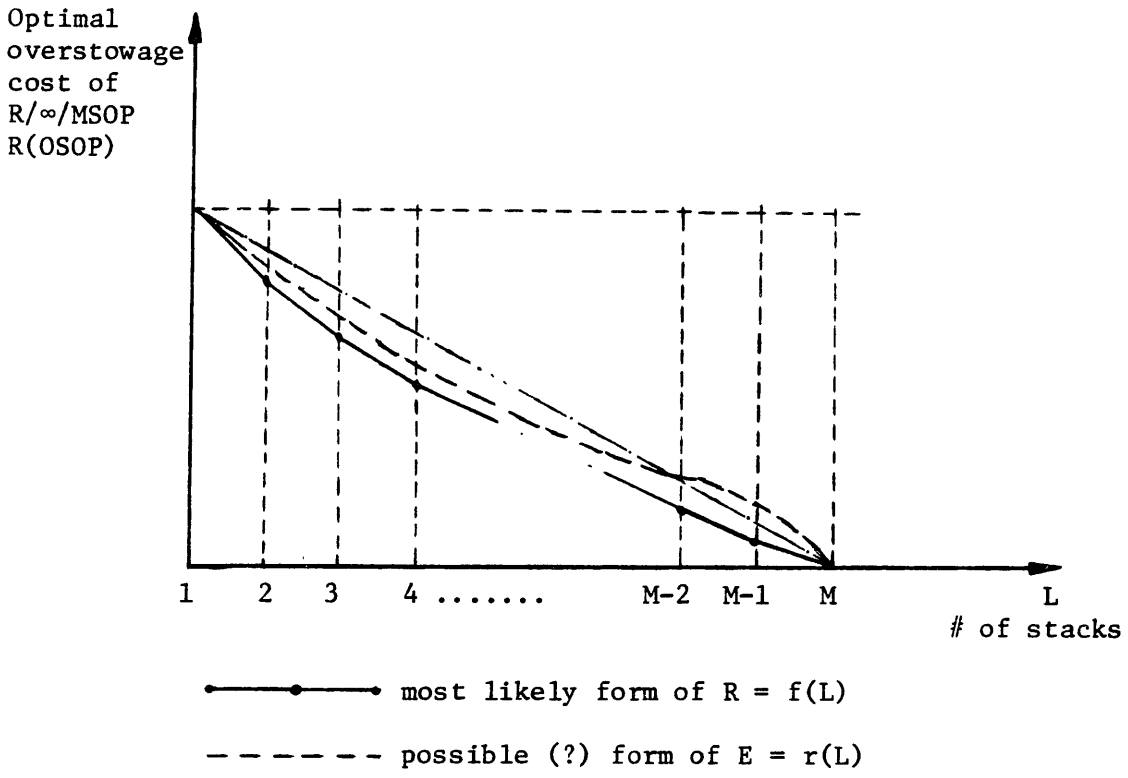


Figure 7.10

R/∞/MSOP: Overstowage Cost as a Function of the Number of Stacks

The above result can be extended to the R/MSOP. As we have assumed all stacks have the same capacity. The capacity of each additional stack is assumed to

be the same with the others. So, it is clear that the optimal overstowage cost should not increase as more stacks become available.

A more interesting case arises when we require the total number of available cells (storage positions) to be the same. That is, we trade number of stacks for stack capacity. Then we have

$$L_1 \cdot b_1 = L_2 \cdot b_2 \qquad (7.6)$$

where L_1, b_1 and L_2, b_2 are the number of stacks and the corresponding capacity of the stacks in two configurations.

In this case, the overstowage cost may go up as the number of stacks increases. This is demonstrated in the following example in Figure 7.11.

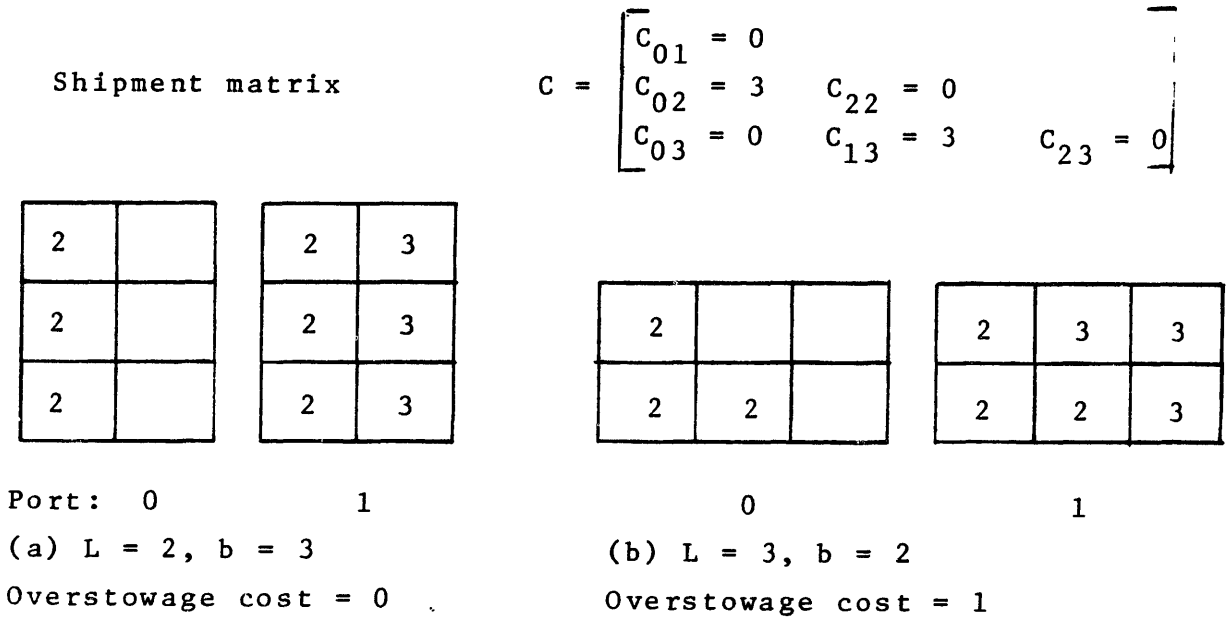


Figure 7.11 - An Example with L.b Constant

However, if L_2 is greater than L_1 and L_2/L_1 is integer then the minimum overstowage cost goes down. This is true because a stack of the old configurations can be broken down into an integer number of stacks in the new configuration. At the very least, the new shorter stacks can be treated as one by filling one before starting to use the next one. This is, of course, suboptimal but it simulates the old configuration. Presumably we can do much better than that with the new configuration. This is demonstrated in Figure 7.12. Filling stack B_2 after stack B_1 (as shown by the arrow) reproduces the operation of stack A.

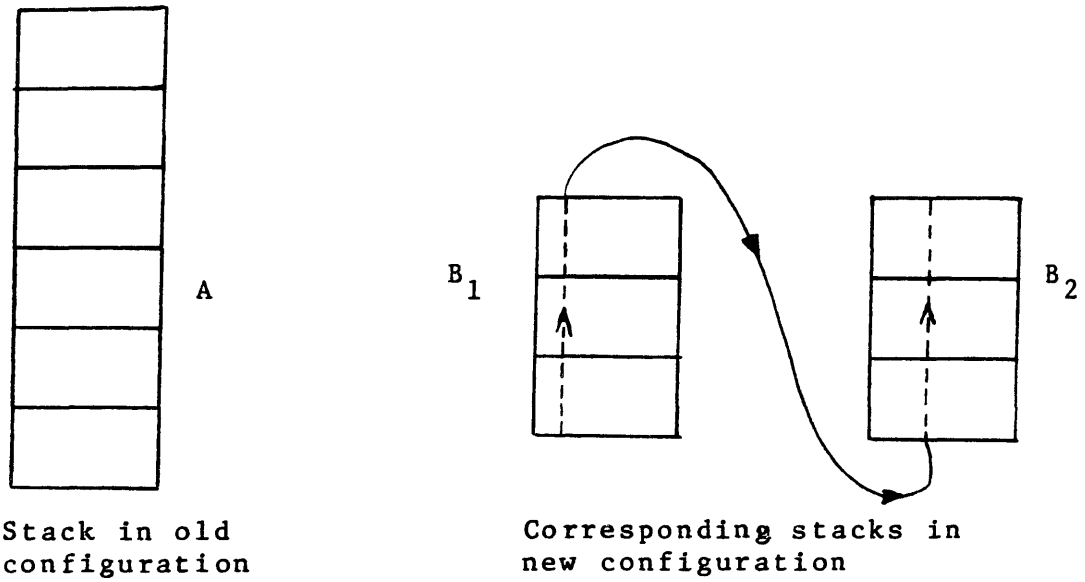


Figure 7.12 - Simulation of Stack A by Stacks B_1, B_2

7.5 Container Assignment for Known Rearrangement Policies

In the previous section we have formulated different versions of the multistack overstowage problem. The objective in those formulations has been to find a set of

optimal rearrangement policies plus the assignment of containers to stacks that minimize total overstowage costs. It has been mentioned that the problem is comprised by two parts: (i) the assignment phase, and (ii) the policy calculation phase. The two phases are naturally interlinked: the assignment of containers to stacks is done in such way so the policies to be found would result in the minimum overstowage cost.

One can separate the two phases discussed above at the expense of optimality. Deciding first about the assignment and then finding the cost minimizing policies is not done differently from the integrated approach. In fact, the early sections of the chapter discuss this case in detail. In this section, we concentrate on the case of assigning the containers to stacks when the rearrangement policies of the stacks are given. The objective of the assignment process is, again, to minimize overstowage costs.

We can use the network formulation of Figure 7.6 to assign the containers to the stacks. The important difference now is that we know the rearrangement policy for each stack. This eliminates the "either-or" constraints. That is, there only one arc of type emanating from each (i) node, and, all such arcs are compatible. A part of such a network is shown in Figure 7.13.

Since the "either-or" constraints have been eliminated, the only constraints that must be satisfied are the stack capacity constraints (bundle constraints). In fact, these constraints are present only in the capacitated versions of the MSOP. This means that the assignment of containers to stacks for the ∞/MSOP can be found by

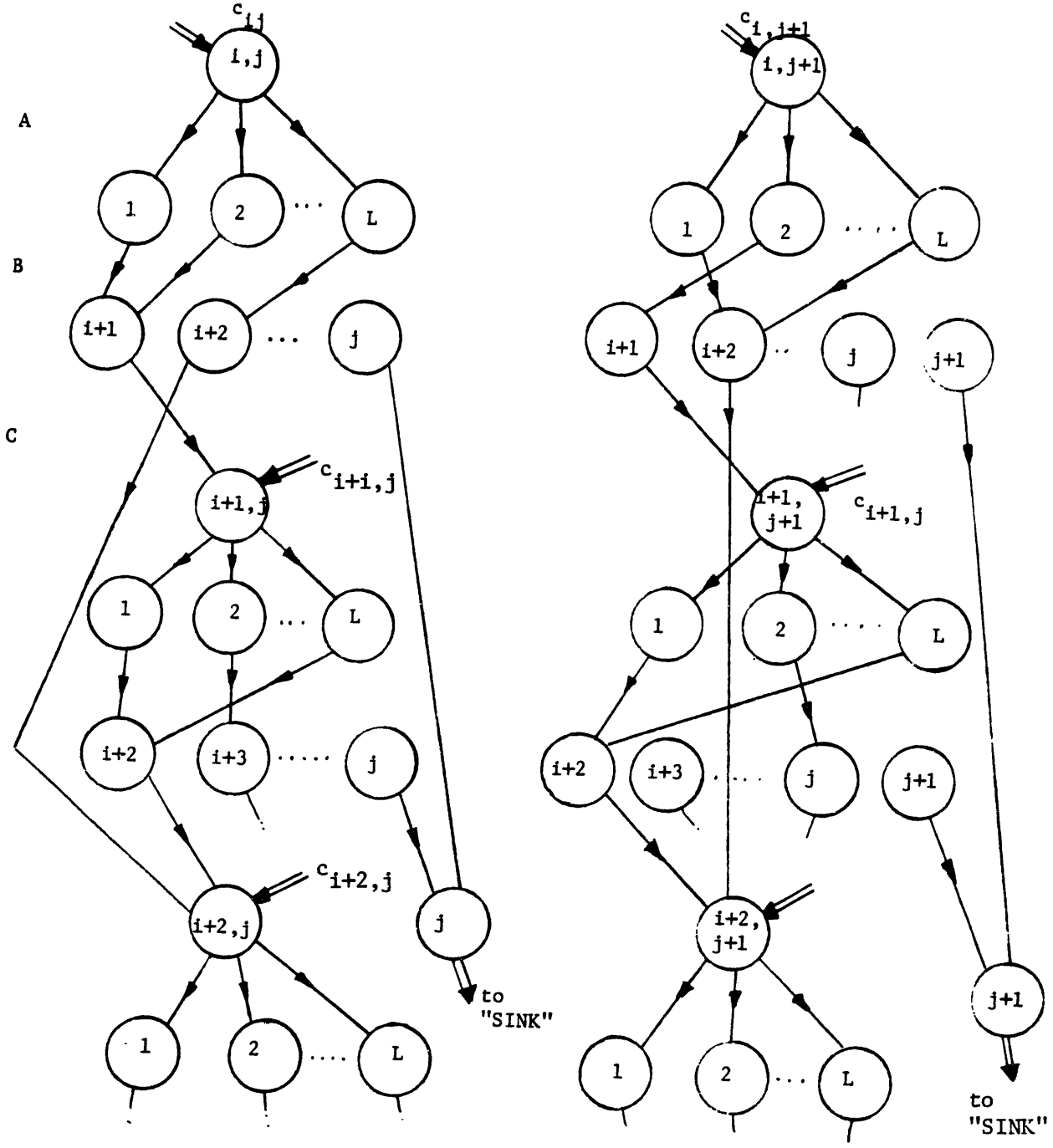


Figure 7.13

Assignment of Containers for Given Set of Rearrangement Policies

solving the minimum cost flow problem on a network similar to that of Figure 7.12. The objective is expressed as a minimum flow requirement on arcs of type C. This arc can be though as having a cost coefficient of unity, while all other arcs have their coefficient equal to zero. A simpler algorithm for solving the assignment problem for the ∞/MSOP is given in Figure 7.14.

It is worth mentioning that the above network solves the unrestricted version of the multistack overstowage problem. To solve the restricted one is much easier, particularly for the uncapacitated version. This is done according to the following rule: "Each group is assigned to the stack whose rearrangement policy rearranges that group the fewest times."

The solution of the capacitated version introduces some problems because of the existence of the bundle capacity constraints (unrestricted case) or side capacity constraints (restricted case). The presence of these constraints results in non integer solutions. It is well known that integer network flow problems with bundle capacity constraints belong to the class of NP-complete problems. In the following we develop a greedy-type heuristic algorithm to circumvent this difficulty.

7.5.1 Assignment of Containers to Stacks Under Given Policies for the ∞/MSOP

In the uncapacitated case, each container (or group of containers) can be assigned independently of the others. Then, the problem reduces to finding the shortest path between the appropriate pair of nodes in the network of Figure 7.12. If we know at what ports a group assigned to a particular stack gets rearranged

(this can be easily computed from the rearrangement policy of the stack), then the shortest path problem mentioned above - in fact for all $O(M^2)$ groups - can be solved as follows.

Observe that if we know the optimal assignment all groups (i,j) with (j-1) less than d, then we can compute any pair (i,j) with (j-1) equal to d. Let y_{ij} be the optimal ararrangement cost for a container of the group (i,j). Then,

$$y_{i,i+d} = \min_{1 \le l \le L} \{f(k_e) + y_{k1,i+d}\} \quad . \quad i=0,\ldots,M+1-d \tag{7.7}$$

where $K_e(i,i+d)$ is the first port that group (i,i+d) of stack l gets rearranged (obviously $i<k_e \le i+d$), and

$$f_{l,i+d}\ (k_l(i,i+d)) = \begin{cases} 1, & \text{if } k_l \neq i+d \\ 0, & \text{if } k_l = i+d \end{cases} \tag{7.8}$$

It is,

$$y_{i,i+1} = 0, \quad i=0,1,\ldots,M \tag{7.9}$$

(7.7, (7.8), and (7.9) define a recursive algorithm which computes the optimal assignment path and cost for each group (i.j). The algorithm (FIXPOL-∞) is shown in Figure 7.14).

The algorithm runs in $O(LM^3)$ time and calculates the assignment sequences a(., ., .) for all groups. In particular it takes $O(LM^3)$ to compute where each group gets rearranged for all the L stacks, $O(LM^2)$ to run the recursion, and $O(M^3)$ to retrieve the assignment sequence. If we are interested only in the assignment of one group, then the running time goes down to O(LM), that is the recursion and

Algorithm FIXPOL-∞

Input: Rearrangement policies of L stacks, M+2 port series (0,1,...M<M+1)

Output: The sequence of stack assignment for each group (i,j) = a(i,j;k), k=i,...,j

BEGIN

Part A: Computation of rearrangement ports of each group for each stack

```
    For (all the stacks) do
      begin
    for k = 1 to M do
      begin
        l: = k;
        repeat l:=l-1 until (P(l) ≥ P,k) or l=0);
        for k=k+1 to P(k) do
            for i: = 0 to K-1 do
                f(i,j,k): = 1;
        for j: = (P(k)+1 to M+1 do
            for i: = l+1 to K-1 do
                f(i,j,k): = 1;
       end;
    end;
end;
```

Part B: Recursion

```
    for i: = 0 to M do
        Y_{i+1}: = 0;
    for d: = 2 to M+1 do
        for i: = 0 to M+1-d do
            begin
                Y_{i,i+d} = min_{i≤l≤L} {f_{i,i+d,l}(k_l(i,i+d)) + Y_{k_l(i,i+d), l+d}}
        q(i,i+d;i) = l_min, l_min is the value of l that minimizes the
                     above expression
        end;
```

Part C: Retrieval of Assignment Sequences

```
for i: = 0 to M do
    for j: i+1 to M+1 do
        for K: = i+1 to (j-1) do
            if ((i,j) of stack a(i,j,k-1) is rearranged at k), then
            a(i,j;k): = a(k,j;k) else a(i,j;k) = a(i,j;k-1);
END
```

Figure 7.14 - Algorithm FIXPOL

the assignment sequence retrieval part are performed for the appropriate values of i, j, and d only, and the ports of rearrangement of that group in each stack can be found in O(LM).

7.5.2 Assignment of Containers to Capacitated Stacks Under Given Policies for MSOP

The difference of the capacitated problem from the uncapacitated is that the assignment of each container cannot be done independently of the others because the number of containers assigned to a stack should always be less than its capacity. We have seen that modeling the problem as a network problem with side (bundle) constraints does not guarantee the integrality of the solution. However, if we cannot assign all the containers at the same time in an optimal way, we can always assign one additional container quite efficiently. This is because now the bundle capacity constraints become conventional capacity constraints, since there is only one container to be assigned. Therefore, we can solve a shortest path problem on the appropriate component of the network of Figure 7.13, in which the capacity

constraints have been incorporated. The latter means that some more arcs have been eliminated, specifically those which diver flow to already full stacks.

A similar algorithm to FIXPOL-∞ can be developed. However, this assigns containers incrementally one by one. The algorithm of Figure 7.15 assigns one container to L stacks partially filled under a given set of policies. Its running time is O( ), the same as for FIXPOL-∞ for only one group (container) assignment.

Algorithm FIXPOL

Input: Rearrangement policies for L stacks; containers already assigned to those stacks; capacity of stacks; container (p,q) to be assigned; (M+2) port series (0,1,...M,M+1)

Output: The assignment sequence of container (p,q)

BEGIN

Part A: Calculation of ports of rearrangment

(identical to FIXPOL-∞)

Part B: Recursion

```
for i: p to (q-1) do
    y_{i,i+1}: = 0;
y_min: = ;
for d: = 0 to (p-q-2) do
    for l=1 to L do
        begin
            if (available capacity of stack l > 1 at all ports from
            (p+d) to q) then if (y_min > f_{p+d,q} + y_{k_e(p+d,p)} then
        begin
            y_min = f_{(p+d),q} (k_l(p+d,q)) + y_{k_e(p+d,p)}
            a(p+d, q; p+d): = l
```

Part C: Retrieval of assignment sequence

(identical to FIXPOL-∞ for i=p, j=q)

Figure 7.15 - Algorithm FIXPOL

CHAPTER 8

DESIGN OF HEURISTICS FOR THE MULTISTACK OVERSTOWAGE PROBLEM

The analysis of Chapter 7 indicates that it is most likely that no efficient (i.e. polynomial time) exact algorithm exists for the multistack overstowage problem. This is stated in the conjecture that the MSOP is an NP-complete problem. A way to cope with the above difficulty is to look for algorithms which do not necessarily find the optimal solution, but guarantee to be within a certain percentage of it and, in addition, run in polynomial time. Unfortunately, neither do approximation algorithms appear to be easy to develop, mainly because it is difficult to come up with the "percent-off" guarantee. Even worse, it is not easy to calculate the worst case performance of such algorithms. It should be stressed that the above statements are not terminal. They are subjected to the time limits for carrying out this research endeavor, and in particular the analysis of the multistack problem. Consequently, further research may lead to stronger results. In the remainder of this study, we concentrate on heuristic algorithms; that is, approaches that do not guarantee anything, but they are expected to perform well in most of the instances most of the time[1], because they are built on a rational basis.

This chapter (8) focuses on the design of heuristics for the MSOP. Specifically we look to how we can decompose the problem so we can solve the components satisfactorily and then integrate them back to get a solution for the original

[1] Some heuristics have incorporated random decisions or depend on an initial starting point, so it is conceivable to get different results every time the heuristic is executed.

problem. The chapter analyzes these ideas, and then proposes a set of heuristics for both the uncapacitated and capacitated problem versions for the restricted and unrestricted case. The following chapter customizes the heuristic designs presented here for minimizing the overstowage cost in container ship operations or other applications with similar characteristics.

Our discussion in this chapter is less rigorous than that of the previous chapters. This is the result of the computational difficulty of solving the multistack overstowage problem, the latter being a function of the highly combinatorial nature of the problem. Our approach is similar to a greedy-type one. As it becomes clear in the following sections, it is difficult to perform a rigorous performance evaluation of the heuristics, even a posteriori. This can be attributed to the lack of meaningful lower and upper bounds for the one stack, and consequently, the multistack problem.

One may wonder how other methods (like branch and bound or lagrangean relaxation methods) perform. Unfortunately, the time limit to finishing this research has not allowed for such considerations. However, we do not expect improvements due to the nature of the problem (bundle constraints, "either-or" constraints). Even for the seemingly simpler R/∞/MSOP there seems to be it no connection between the "either-or" constraints of the different stacks.

The presentation of the heuristics is done at the macroscopic level; that is, we do not bother to deal with the implementation details. It is beyond a doubt though that our understanding of their performance will improve when the algorithms are

coded and tested. This is the first thing to do to extend this research endeavor.

8.1 Introduction to Heuristic Design for the MSOP

The discussion of Chapter 7 has revealed a new aspect which is present in the multistack overstowage problem. That is the assignment of containers to stacks. In fact, it has been realized that the above assignment can be dynamic in the sense that containers may switch stacks along the trip of the vessel. Then for a given container assignment, an effective shipment matrix for each stack can be deduced. The rearrangement policy and the resulting overstowage cost are calculated by running the OSOP algorithm for each stack. Solving the problem optimally requires the container assignment and rearrangement policy calculation processes to be carried out simultaneously in order to find the assignment which minimizes the overstowage cost.

Since it appears to be difficult to find the above assignment in polynomial time, we settle for a possibly suboptimal assignment which is the output of a simpler solution approach. The simplification that we are going to pursue here is the "elimination" of the link between the container assignment process and the policy calculation one. This link is broken in the sense that the two phases are not carried out in parallel, but sequentially. Every time an assignment is made, a set of rearrangement policies is available. The assignment is made based on these policies. Also, whenever rearrangement policies are calculated a set of assignments is given and the policies refer to it (in fact, to the shipment matrices that are

deduced from it).

Then, a heuristic algorithm can be designed as a series of container assignments (or reassignments), alternating with rearrangement policy calculations for each stack and computation of the resulting overstowage cost. The cycle continues for as long as reductions in the overstowage cost are achieved. If no improvement is possible, it means that the heuristic has found a minimum. The suboptimality of the approach lies with the fact that the above minimum may only be local (and not global). This generic approach is presented in Figure 8.1.

The algorithm of Figure 8.1 consists of two phases. In the first phase an initial assignment of containers is obtained (STEP 1 and STEP 2), while in the second phase further improvements are attempted. Along the initial container assignment phase, the algorithm always optimizes the rearrangement policies for a given assignment, and reassigns the containers based on the resulting rearrangement policies (STEP 3). It is clear that the algorithm tries to minimize the increase in R every time a new container is assigned. This strategy is suboptimal because it overlooks assignments that could initially increase the overstowage cost but result in lower costs at the end. As a consequence we observe that the solution that is produced depends also on the order by which containers are considered for assignment. Then, an idea for the solution improvement phase (STEP 6) could be to repeat the entire process with different container assignment order and observe how much the solution changes.

Initialization, Definitions

Step 0

A: set of unassigned containers identified by their origin and destination

B: set of the assigned containers

d_{ij}: partial derivatives of rearrangement policies

Set B = 0, A = {all containers}

Set d^{*}_{ij} = 0, i=0,...M; j=i+1,...M+1; l=1,...L (all stacks and initially empty)

Set R = 0 (overstowage cost)

Initial Container Assignment

Step 1

Pick $\alpha \in A$. Set A = A-{a} and B = BU{a}.
Assign a to a stack(s), given the rearrangement policies in effect, such as to minimize the increase in R.

Step 2

Recalculate the optimal rearrangement policies for those stacks that are affected from the last assignment.

Step 3

Call assignment improvement routine (AIR).

Step 4

If A = 0 to to STEP 1; else END

Solution Improvement
(Assignment Improvement Routine - AIR)

Step 1

Apply solution improvement routine (i.e. systematic container interchanges or policy changes, etc.)

Step 2

End.

Figure 8.1 - Generic Heuristic Algorithm for MSOP

The above approach is to be tailored to the particular version of the multistack overstowage problem we solve. We reserve a more detailed presentation and analysis of the algorithm for the later sections where that tailoring is performed.

The performance of the heuristic algorithm can be evaluated by measuring how much off from the optimal the solution provided by the heuristic is. Unfortunately though, it is very difficult to estimate the latter at this stage of our understanding of the problem. A set of evaluation criteria is provided in the next section.

8.2 Evaluation Criteria of Approximate Solution Methods for the MSOP

Let R_H be the overstowage cost of the solution provided by a heuristic algorithm, H. Let R^* be the optimal overstowage cost. Then, a measure of performance of heuristic H (p_h) is the ratio:

$$p_h = \frac{R_H}{R^*} \tag{8.1}$$

The closer to unity the above ratio is the better the heuristic. Obviously, p_h may be different for different instances of the problem. Then, if we want to judge the performance of the heuristic over all the problem instances we can consider the worst possible performance ratio (p_H), or an average performance ratio ($\bar{p}_h$).

There exist two kinds of performance evaluations: (i) a priori, that is, before the heuristic is run, and (ii) a posteriori, that is after the heuristic is run (R_H is now known). The first one is equivalent to a performance guarantee that the heuristic will not perform worse than p_H. The second one is an evaluation of how the heuristic has performed in a given instance. A disadvantage of the latter is that it

requires the knowledge of the optimal solution, R^*, which is most probably unlikely[2] (otherwise we would not use the heuristic). To overcome such difficulties we use lower bounds of R^* to get approximate performance ratios:

$$p_{h,l} = \frac{R_H}{R_L} \tag{8.2}$$

where R_L is a lower bound of R^*.

Another possible way of evaluating the performance of a heuristic is to see how far from an upper bound of the optimal solution R_H is. In this case the smaller the value of the corresponding performance ratio ($p_{h,u}$) is the better.

$$p_{h,u} = \frac{R_H}{R_U} \tag{8.3}$$

where R_H an upper bound of R^*.

Ideally, we would like to combine an a priori guarantee for the performance of the heuristic with an a posteriori evaluation of the solution. However, it appears that the nature of the problem and of the heuristic approach of Figure 8.1 do not lend themselves to a priori analysis. Consequently, we can use only a posteriori analysis. The following bounds of the optimal (minimum) overstowage cost are defined:

(i) Lower: a lower bound is zero

$$R_L = 0 \tag{8.4}$$

(ii) Upper: An upper bound is the maximum number of rearrangements that can be performed, that is to

[2] The case in which we know R^* but not the optimal assignment has only theoretical value.

rearrange all containers at every port.

$$R_U = \sum_{i=1}^{M} \sum_{j=i+1}^{M+1} (j-i-1) \cdot c_{ij} \qquad (8.5)$$

Since $R_L = 0$, it is not possible to use (8.2) to calculate $p_{h,l}$. For this reason, we redefine the performance ratios (8.1) - (8.3) as follows:

$$r_h = \frac{n+R_h}{n+R^*} \qquad (8.6)$$

$$r_{h,l} = \frac{n+r_H}{n+R_L} \qquad (8.7)$$

$$r_{h,u} = \frac{n+R_H}{n+R_U} \qquad (8.8)$$

where n is the number of containers carried between ports of the port series (0, 1, 3,...,M, M+1).

It must be mentioned that the bounds presented above are relatively loose. The most interesting of the two we can compute is $r_{h,l}$. It is doubtful though whether we can meaningfully use it to evaluate the performance of the heuristic against the optimum when R_L is equal to zero[3]. It can be used more effectively as an indicator of whether the given instance gives rise to extensive overstowage or not.

8.3 Heuristics for the Uncapacitated MSOP

In the uncapacitated MSOP we can take advantage of Lemma 7,1 and design

[3] This is an absolute minimum of R*, in fact the minimum value it can ever asume.

algorithms which deal with "groups" of containers, as defined in Chapter 3, and not individual containers. The above "grouping" results in short running times. In the following, we distinguish between the restricted and unrestricted version of the problem (R/∞/MSOP and ∞/MSOP).

8.3.1 The R/∞/MSOP

First, we examine algorithms for the R/∞/MSOP. The approach which is presented below is derived from Figure 8.1 and it is a greedy-type algorithm. Since the groups of containers are not allowed to switch stacks, the problem can be stated as a partition problem. That is, we want to partition the set of groups into L subsets so that the total overstowage cost is minimized. As mentioned in the beginning of Chapter 7, the problem is meaningful only when L is smaller than M. The steps of the algorithm are shown in Figure 8.2. The algorithm is called MAXSAVER because it always tries to keep the increase in the overstowage cost as low as possible.

Algorithm MAXSAVER

Initialization/Definitions

Step 0

A = set of unassigned groups
B = set of assigned groups

d^l_{ij} = partial derivatives of the rearrangement policies for each stack
A = {all groups}, B = 0, d^l_{ij} = 0, i=0,...M; j=i+1,..., M+1; l=D,...L

Container Assignment

Step 1

Pick a group $g \in A$. Assume g is group (i,j). $A = A-\{g\}$; $B = BU\{g\}$.

$$l_{min} = \min_{1 \le l \le L} \{l: d^l_g \le d^{l'}_g, \ l' = 1, \ldots L\}$$

Assign g to stack l_{min}. Then the increase in R is no more than $d^{l_{min}}_g \cdot C_g$

Step 2

Recalculate the optimal policies and group derivatives for stack l_{min}

Step 3

Call assignment improvement routine (AIR).

Step 4

If $A = \emptyset$ go to STEP 1; else END.

Assignment Improvement Routine (AIR)

Step 1

Apply 1-group reassignments.

For all groups already assigned (set B), check whether their reassignment to another stack reduces R. Implement the reassignment with the greatest reduction in R. If no such reassignment exists to to STEP 3.

Step 2

Recalculate the optimal policy and derivatives for all stacks that change. Go back to step 1.

Step 3

Apply 2-group interchanges.

For all pairs of groups g_1, g_2 has been assigned to stack l_1 and g_2 to stack l_2, check whether any of the following reassignments decreases the rearrangement cost.

(i) $g_1 \to l_2$
$g_2 \to l_j$, $j=1,3,4,\ldots L$;

(ii) $g_1 \to l_j$, $j=3,4,\ldots L$
$g_2 \to l_1$

The change in R is computed by running STEP 2 and STEP 1 of AIR. The best interchange is implemented. All pairs of groups are cyclically tested until no improvement in R is realized during the last cycle.

Step 4
End {AIR}

Figure 8.2 - Algorithm Maxsaver for Solving (R/∞/MSOP)

A number of observations for the algorithm follow. First, STEP 1 and STEP 2 can be combined. The assignment can be done based on the change of R, after the reoptimization (STEP 2). This introduces a bit more work but we save the calculation of the derivatives which is now redundant. Also, the assignment improvement routine is not necessary to be performed. This routine attempts to improve the assignment subject to the effective set of policies. The basic idea of the algorithm is captured by steps 1 and 2 (or, better by a combined step as discussed in the beginning of this paragraph).

STEP 1 and STEP 2 of AIR make sure that a local optimality condition always holds. This condition says that an assignment must be optimal for the given set of policies, and at the same time the policies must be optimal for the given assignment. This process takes care of single group reassignment. STEP 3 of AIR looks at simultaneous reassignments of two groups. The latter makes sense only if one of the groups is reassigned to the stack of the second group and also the second group is moved to another stack; otherwise, the case reduces to two single

group reassignments. Every time a reassignment takes place, the local optimality condition needs to be checked and enforced (steps 1 and 2 of AIR). In fact, the latter need not be called every time a reassignment (or assignment) is made; it can be performed only periodically in order to reduce the computational requirements.

The running time of each step of MAXSAVER is as follows:

Step 0: $O(LM^2)$ - initialization of f_{ij}'s

Step 1: $O(L)$ - find the minimum of L numbers

Step 2: $O(M^3)$ to recalculate policy

$O(M^5)$ to find new derivatives

Combined 1 and 2: $O(LM^3)$ to recalculate policies

$O(L)$ to find the minimum increase in R

AIR - Step 1:

to find what is the best stack for each group of set B

AIR - Step 2: $O(LM^3)$ to find the optimal policies. Steps 1 and 2 can be repeated at most R_u times.

AIR - Step 3: There exist $O(N^2) = O(M^4)$ pairs of groups. For each pair we check $O(L)$ reassignments. For each reassignment we run steps 1 and 2 at most R_u times. The above is run at most R_u times. The total is $O(R_u^2 \cdot L^2 M^9)$.

As we can see from the above computations, the algorithm runs in $O(L.M^5)$ time if no calls to the assignment improvement routine are made. The latter runs in $O(R^2_u \cdot L^2 \cdot M^9)$, where R_u is an upper bound of the overstowage cost. In fact, the first part of the AIR (that is steps 1 and 2) runs in $O(R_u \cdot L \cdot M^5)$ time. The running time of the second part is $O(R^2_u \cdot L^2 \cdot M^9)$ and determines the running time of the routine. Notice that the latter running times depend on the values of the c_{ij}'s. This makes the running time of the AIR a special type exponential function of the size of the input, called pseudopolynomial.

We conclude our discussion on the R/∞/MSOP, by proposing an estimation of R^* through another path. As it has been discussed in Chapter 7, the overstowage cost of R/ /MSOP declines as a function of the number of stacks available for use (Figure 7.10). The cost starts at a value equal to the overstowage cost of the one-stack overstowage problem and goes to zero for L=M. It is conjectured that the marginal benefit of using one additional stack goes down as the number of stacks increase. Then the point that corresponds to linear decline should provide an upper bound of the optimal rearrangement costs:

$$R'_u = (1 - \frac{L}{M}) \cdot R^*(OSOP), \quad 1 \leq L \leq M \tag{8.9}$$

If our assumption about the marginal benefit of using an additional stack is correct, then (8.9) provides not a very a strict upper bound, but at least much better than R_u (see ((8.5)).

8.3.2 The ∞/MSOP

We can now turn to the unrestricted uncapacitated MSOP(∞/MSOP). Lemma 7.1 holds, so our discussion will again be in terms of groups and not individual containers. The new feature of this version is the possibility for groups to move from stack to stack. This problem is meaningful only if L is less than M.

There are many different ways to account for the new possibility. One is to ignore it in the beginning and run the algorithm for the R/∞/MSOP(MAXSAVER). Then we can apply the algorithm of Figure 7.13 (FIXPOL-∞) for the rearrangement policies produced by MAXSAVER. The resulting assignment gives rise to an effective shipment matrix for each stack. The OSOP algorithm is used to obtain the optimal policy for each one of the stacks. The approach is summarized in Figure 8.3.

The running time of STEPS 2-4 is:

Step 2: $O(LM^2)$ - running time of FIXPOL

Step 3: $O(M^2)$ - time to compute all effective shipment matrices

Step 4: $O(LM^3)$ - running time of the OSOP algorithm for L stacks.

The maximum number of times that steps 2-4 are repeated is R_u, where R_u is the solution produced by MAXSAVER. Then, the overall time to run Steps 2-4 is $O(R_u \cdot L \cdot M^3)$. Again, this is a pseudopolynomial function of the size of the input.

Algorithm SWITCH- ∞

Step 1

Run algorithm MAXSAVER (R/ /MSOP) . Get a set of rearrangement policies, a value for overstowage cost, R.

Step 2

Run FIXPOL-∞ on the policies produced in STEP 1 or STEP 4.

Step 3

For the assignment produced by FIXPOL-∞ , calculate the effective shipment matrix for each stack.

Step 4

Run the OSOP algorithm for each stack on the effective shipment matrix. Calculate the resulting overstowage cost. Get the rearrangement policy for each stack and derivation.

Step 5

if R has decreased, go to STEP 2; else, STOP.

Figure 8.3 - Algorithm Switch-∞ for the ∞/MSOP

Another approach to solving the ∞/MSOP is similar to algorithm MAXSAVER, except that when we assign a group, we take advantage of the

possibility that it can switch stacks at later ports. This is achieved by running FIXPOL-∞ on the effective set of policies at the moment of the assignment. This is essentially the same process as running steps 2-4 of algorithm SWITCH-∞ for the group under consideration and the affected stacks. The "assignment improvement routine" can be restricted to the first two steps, because the implementation of step 3 is not possible. the resulting algorithm is presented in Figure 8.4. The running time of this algorithm is as follows:

Step 1: $O(LM^2)$ - running time of FIXPOL-∞

Step 2: $O(M^2)$

Step 3: $O(L.M^3)$

Step 5: $O(M^2)$ - number of groups to be assigned

Step 4: $O(R_u . L . M^3)$ - running time of STEPS 1 and 2 of the Assignment Improvement Routine (AIR) (see page 180).

The total running time without calling routine AIR is $O(LM^3)$. As expected AIR runs in time which is a pseudopolynomial function of the size of the input.

The solution of the ∞/MSOP is always better (results in less overstowage cost) than the solution of the corresponding R/∞/MSOP. So the solution of MAXSWITCH is expected to be better than that of MAXSAVER (SWITCH-∞ gives a better solution than MAXSAVER by construction). Figure 8.5 shows how the different algorithms we have discussed so far are combined.

Algorithm MAXSWITCH

Step 0
Initialization/definitions (see Figure 8.2, Step 0)

Step 1
Pick a group of $g \in A$. Say g = (i,j). Set A = A-{g}, B = BU{g}. Asign group g to a stack (sequence of stacks) by implementing FIXPOL-∞ on the effective rearrangement policies.

Step 2
Calculate the effective shipment matrix for all stacks affected by the above assignment.

Step 3
Run the OSOP algorithm for those stacks affected.

Step 4
Call AIR (assignment improvement routine).

Step 5
If A = 0 go to STEP 1; else END.

Assignment Improvement Routine (AIR)

Step 1
For all assigned groups (set B), check whether their reassignment according to FIXPOL-∞ under the effective rearrangement policies reduces the overstowage cost. Implement the reassignment which decreases the cost the most. If no such reassignment exists, END.

Step 2
Recalculate the effective shipment matrix, optimal rearrangement policy, and group derivatives for all the stacks that are affected. Go to step 1.

Figure 8.4 - Algorithm MAXSWITCH For the ∞/MSOP

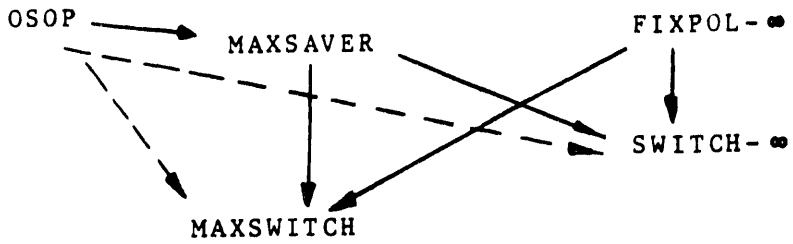


Figure 8.5 - Algorithmic Design Flow of Algorithms for ∞/MSOP

8.4 Heuristics for the Capacitated MSOP

In this section we examine heuristic algorithms for the capacitated case of the multistack overstowage problem. Because of the mere presence of the stack capacity constraints, we cannot work with groups of containers (of the same origin and destination), but rather with individual containers. In addition, the order by which containers are assigned to stacks is now important because lack of capacity will force containers to switch stacks. This can be seen in the following example. Suppose we want to assign two containers with origins and destinations (1,5) and (2,4) correspondingly. Also assume that the available capacity are as in Figure 8.6 below.

Port	1	2	3	4
Stack 1	0	1	1	1
Stack 2	1	1	1	1

Figure 8.6 - Example Available Capacity of Stacks

If we first choose to assign container (2,4), and we assign it to stack 2, then container (1,5) should be assigned to stack 2 for the leg 1-2 of the trip and then moved to stack 1 for the remaining of the trip. To avoid a situation like the above, we can either backtrack and redo some of the assignments of assign the containers in a way that avoids the above problem. The latter is achieved if the assignment process proceeds in a "natural" order, that is, in ascending order of origin and among containers of the same origin in descending order of destination. The former may lead to a long process.

Our assumptions are that (i) all stacks have the same capacity and (ii) there is enough capacity to accommodate all the containers on board (so that we do not have to select which to ship). We are also going to distinguish between the restricted and unrestricted version.

8.4.1 The R/MSOP

The algorithm resemples to MAXSAVER (for R/ /MSOP). It has the following differences from it:

(i) The assignment is done container-by-container (not group-by-group).

(ii) Containers are only assigned to stacks with available capacity.

(iii) The "assignment improvement routine" (AIR) works on a container basis again, and must observe the

capacity constraints.

The rearrangement policies of each stack are calculated by running the OSOP algorithm for the resulting (effective) shipment matrix for each stack. Also, it is expected that the running time of the heuristic will be higher than MAXSAVER's since more operations depend now on the values of c_{ij}'s. It should be mentioned though that the algorithm is to be termed pseudopolynomial or not according to whether n, the total number of containers, is thought as direct input or as a function of input parameters (i.e. $\sum_{i=0}^{M} \sum_{j=i+1}^{M+1} 1 = (M+1)(M+2)/2$ non-negative numbers c_{ij}):

$$n = \sum_{i=0}^{M} \sum_{j=i+1}^{M+1} c_{ij}$$

It seems appropriate in this case to consider n as direct input, from which c_{ij}'s are inferred, given the origin and destination of each container. We should stress the difference between the need to explicitly handle each container individually, and the use of a function of n ($R_n = O(n)$) to determine how many times a loop is executed. For this reason, we will consider running times which are polynomial functions of n as polynomial functions of the input.

The algorithm (MINSLOPE) is shown in Figure 8.7. The running time of it is as follows:

Step 1: O(L) - find minimum of f_{ij}, l=1,...L and check capacity constraints

Step 2: $O(M^3)$ - recalculate policy

Step 3:

Air-Step 1:

containers, L stacks, O(M) to check the capacity constraint for each stack, and $O(M^3)$ to run OSOP algorithm.

Air-Step 2:

STEP 1 and STEP 2 can be repeated at most O(nM) times (R_u = O(nM)).

Air-Step 3: $O(n^2)$ pairs. For each pair we check O(L) assignments. For each assignment it takes O(M) to check the capacity constraints. Again for each assignment AIR-Step 1 and 2 can be performed O(nM) times. Finally, AIR-Step 3 can be repeated at most O(nM) times. The total time for AIR is $O(n^5 M^6 L^2)$.

It should be pointed out that checking for the capacity constraints in AIR is straightforward, but it could be very restrictive, because the required capacity (that is, one container position) should be available for a series of ports (from the origin to the port right before the destination of the container to be moved in). Of

Algorithm MINSLOPE

Step 0 - Initialization

A = {all containers}, set of unassigned containers
B = 0, set of assigned containers

partial derivatives of the rearrangement policies

Container Assignment

For i=0 to M do
For j=i+1 to M+1 do
For x=1 to c_{ij} do
Begin

Step 1

Assign container (i,j) to the stack with the minimum d'_{ij} with available capacity

Step 2

Recalculate rearrangement policy of the stack, compute overstowage cost and derivatives.

Step 3

Call "assignment improvement routine" - AIR;

end;
end;
end;

Assignment Improvement Routine (AIR)

Step 1

Apply single-container reassignments. For each of the containers already assigned (set B), check whether it can be reassigned to another stack, subject to the capacity constraints, with parallel reduction of overstowage cost.

Implement the reassignment with the greatest reduction.
If no such reassignment is found, go to step 3.

Step 2

Recalculate the rearrangement policy of the affected stacks.
Compute derivatives and overstowage cost.

Step 3

Apply two-container interchanges. For all pairs of containers C_i, C_t where C_i has been assigned to stack and l_i to stack C_2, check whether any of the following reassignments decreases the overstowage cost.

(i) $C_i \rightarrow l_2$

$C_2 \rightarrow l_j$, $j \in \{1,3,4,...L\} \cap$ {stacks with available capacity}

(ii) $C_i \rightarrow l_j$. $j \in \{3,4,...L\} \cap$ {stacks with available capacity}

$C_2 \rightarrow l_i$

The change in R is computed by running STEP 2 and STEP 1 of AIR. The best interchange is implemented. All pairs of containers are cyclically tested until no improvement in R is possible.

Step 4

Figure 8.7 - Algorithm Minslope for Solving R/MSOP

---

course, the above depends on how many containers we have to deal with.

It appears that the usefulness of the assignment improvement routine may not be very significant because (i) of the restrictive nature of the capacity constraints, and (ii) of the large running time of the routine. If we exclude the call to this routine from MINSLOPE, the running time of the algorithm is $O(n \cdot (L+M^5))$.

8.4.2 The Unrestricted Capacitated MSOP

The structure of the heuristics for this version of the multistack problem is similar to the algorithms for the ∞/MSOP in Section 8.3.2: algorithm SWITCH-1 processes the output of MINSLOPE, and algorithm SWITCH-2 is an adaptation of MAXSWITCH to the capacitated case. FIXPOL, a capacitated version of FIXPOL-∞, plays a central role in the implementation of the above heuristics. As mentioned in Figure 7.15, FIXPOL is not optimal because the assignments are made sequentially, not simultaneously because the latter (network flow problem of Figure 7.12) may result in non-integer solution of the network flow problem of Figure 7.13 due to the capacity constraints.

The steps of SWITCH-1 and SWITCH-2 are similar to the steps of algorithms SWITCH-∞ and MAXSWITCH, shown in Figures 8.3 and 8.4. There are only two differences: (i) instead of assigning groups in a "random"[4] order, we assign containers in a specified ("natural") order, and (ii) capacity constraints are met through the use of FIXPOL routine (instead of FIXPOL-∞). The running time of the last heuristics is calculated in a similar fashion. There exists n containers to be assigned and in general, $R_u = O(n)$.

The solutions of SWITCH-∞ and MAXSWITCH can serve as estimates to the solutions of SWITCH-1 and SWITCH-2. It is expected that

$$R(\text{SWITCH-}\infty) \leq R(\text{SWITCH-1}) \text{ , and}$$

[4] That is an order that we can choose.

$$R(\text{MAXSWITCH}) \leq R(\text{SWITCH}-2) \qquad (8.10)$$

although there could exist strange instances in which (8.10) may not hold.

CHAPTER 9

APPLICATION OF MSOP HEURISTICS

In this chapter we apply the concepts and results derived in the previous two chapters to a real case where the overstowage problem occurs and, in fact, has been the motivation for this research, the containership operations case. In addition, to apply the heuristics developed in the previous chapter, we also try to take advantage of special characteristics of the problem.

9.1 Parameter Ranges for Various Applications

Solving the ∞/MSOP and R/∞/MSOP has great theoretical significance. Of course, it is rare for someone to find infinite capacity stacks in practice. Capacity constraints are imposed from the mere fact that it is difficult to physically stack very high. In addition there are other considerations that may restrict the capacity of a stack as we like stability or strength requirements. In some cases though one may assume practically infinite stack capacity (i.e. horizontal stacking, computer stacks, etc.)

As it has been discussed in Chapter 7, the complexity of the multi-stack overstowage problem depends on the values (absolute and relative) of the following parameters: (1) number of stacks, L, (ii) number of ports, M and (iii) stack capacities b. The containership operation case is one in which there is a relative large number of stacks ($L \approx 200-500$) with small capacity ($b \approx 2-8$) and a rather small number of ports to be visited ($M \approx 3-10$). Figure 9.1 shows ranges of values

for M and L in various applications. It must be mentioned that these ranges are indicative only, and by no means restrictive. All the above problems are capacitated.

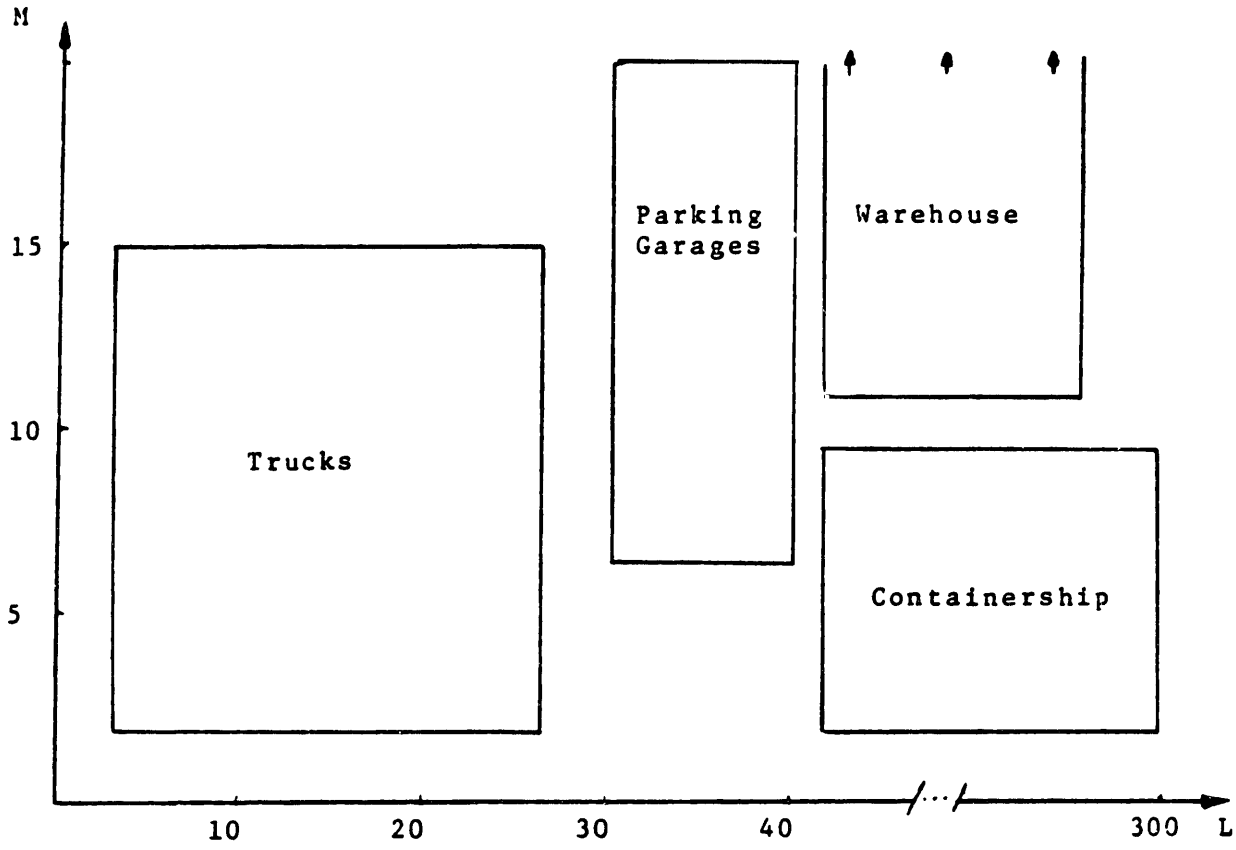


Figure 9.1

Ranges of Parameters in Various Applications of Stacking Problems

In the following, we attempt to take advantage of our knowledge of the ranges of parameters, and customize the heuristic algorithm discussed in the previous chapter or build new ones tailored to the application. Some preliminary thoughts

for the containership operations case follow.

First, we observe that since the capacity of the stacks is small, we can solve any resulting OSOP very quickly. In some cases the faster solution technique may be complete enumeration. Secondly, the overstowage cost should be relatively small (much smaller than the cost of the corresponding one stack overstowage problem) because L is much larger than M. This follows from our discussion[1] in Section 7.4. Thirdly, one would expect that container switches would be very few. The latter is based on the fact that the number of containers per stack is small (because the capacity of the stack is small), and consequently the number of times a container gets rearranged should be small. So, there is relatively little (if at all) to be gained by switching to another stack. Therefore, it appears that we can concentrate on the restricted version of the problem (at least in the beginning).

9.2 Conditions for Zero Overstowage

In Section 7.1, we have first classified the MSOP according to whether it is capacitated or not, and, whether L is greater than M or not. We have also observed that an obvious condition for zero overstowage for the uncapacitated version is L to be greater than M. In this section we extend the above condition to capacitated problems.

Suppose that C is the given shipment matrix and b is the capacity of each

[1] The counter example presented there is an extreme case. Although there may be instances in which the overstowage cost increases, this cannot happen but for a small sequence of stack increases.

stack. We define l_{ij} as

$$\sum_{k=0}^{i} c_{kj} \qquad (9.1)$$

that is l_{ij} is the number of stacks required by containers of type j at port i. Then the following lemma can be easily proven.

Lemma 9.1

A sufficient condition for zero overstowage cost in the capacitated version of the MSOP is $\sum_{j=i+1}^{M+1} \lceil l_{ij} \rceil \leq L, \ \forall \ i=0,1,2,\ldots M$ (9.2)

Proof

$[l_{ij}]$ is the number of required stacks for containers of type j at port i rounded to the nearest integer bigger than l_{ij}. If (9.2) holds throughout the series, it means that we do not have to place the containers of different types on the same stack. Since containers of different destinations are not "intermixed", there is no possibility for overstowage. ■

Condition (9.2) is sufficient, but not necessary. The latter means that even if (9.2) is violated, the overstowage cost can be zero. Furthermore, Lemma 9.1 implies a method to place the containers on board. This method is to stack the containers of the same destination on the same stack as high as possible using the minimum number of stacks for each type of container. If (9.2) holds, then this method leads to an assignment that results in zero overstowage cost.

9.3 Analysis of Overstowage in Containership Operations

If (9.2) does not hold for some i ϵ {0,1,...,M} then the above method stalls, since we must assign containers with different destinations to the same stack. The latter may or may not cause overstowage. Notice, that the following condition

$$\sum_{j=i+1}^{M+1} l_{ij} \leq L \ , \ i=0,1,2,\ldots M \qquad (9.3)$$

should always hold, due to the assumption that there exists enough capacity to accommodate all containers to be shipped on board. Three definitions are next:

Definition 9.1:

A column-wise stacking policy is an assignment policy under which containers are assigned to the stack (and placed on the top of containers already assigned to that stack) with the largest member of containers already assigned to it. ■

Definition 9.2:

A column-wise per destination (origin) stacking policy is a policy under which containers of each destination (origin) are assigned column-wise to a separate set of stacks. ■

Definition 9.3:

A row-wise stacking policy is an assignment policy under which a container is assigned to a stack with the smallest number of containers already assigned to it. ■

Based on the above definitions we can examine how a very simple algorithm behaves. The algorithm (Containership Operations Overstowage Problem, COOP) is presented in Figure 9.2 below:

Algorithm COOP-1

Assign containers column-wise in order of ascending origin and among containers with the same origin in order of descending destination.

Figure 9.2 Algorithm COOP-1

Algorithm COOP-1 is really simple. The resulting assignment is one with a number of stacks filled to capacity with one more partially filled. When containers are delivered and before the new ones are placed on board, a certain number of stacks is partially filled. The latter number gets increasing from the beginning towards the end of the port series. For example, at port 1 we can have up to one partially filled stack before the new containers are loaded (see Figure 9.3(a)). Following the example of Figure 9.3, we observe that at port 2 we have two partially filled stacks. Notice what happened to stack 4. This stack was partially filled at port 1; according to COOP-1, it was the first stack to be used to get containers originating at port 1. So it was assigned 3 containers with destination port 6. The latter containers should be rearranged at port 2 to make possible the delivery of the two containers with destination port 2. There we get the

arrangement of Figure 9.3(b), in which there can be up to 2 stacks partially filled.

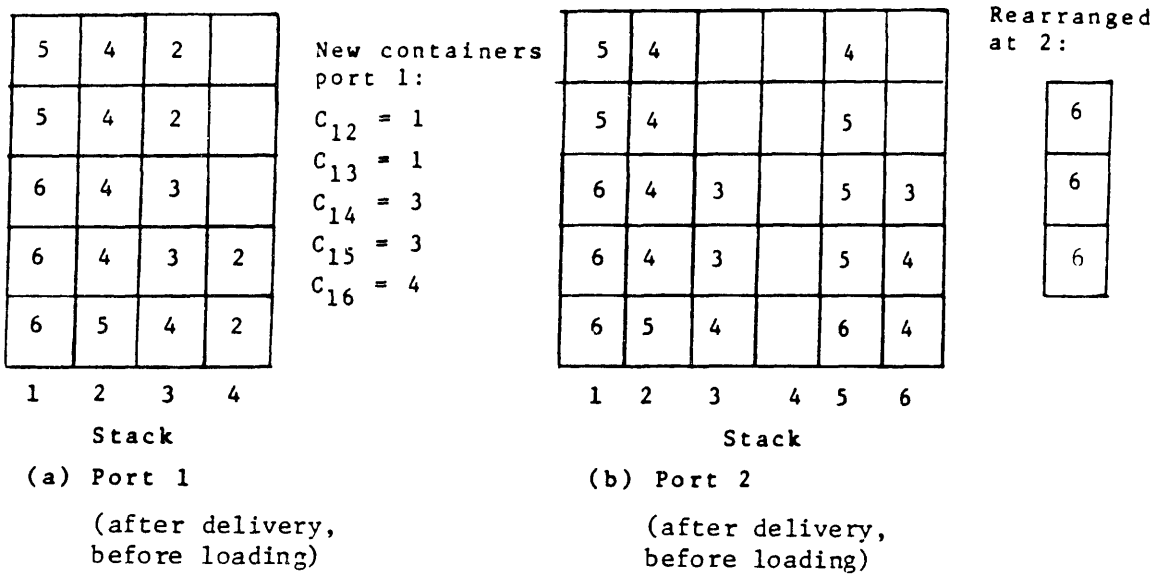


Figure 9.3 - Sample Output of COOP-1

(Numbers indicate the destination of each container.)

Extending the above analysis we conclude that we can have up to i partially filled stacks at port i, i=1,...M. Each of these partially filled stacks can result in an overstowage situation. In the worst case scenario all such stacks give rise to overstowage. There can be (b-1) overstowed containers at most. So the total number of overstowed containers (i.e. container rearrangements) can be no more than

$$R_u(\text{COOP-1}) = \sum_{i=1}^{M} i.(b-1) = \frac{M(M+1)(b-1)}{2} \qquad (9.4)$$

The bound for the overstowage cost of (9.4) is independent of the number of containers to be shipped. In fact, COOP-1 is not a very good algorithm; it does not optimize with respect to overstowage cost (other than abiding to Lemma 3.1).

For example, the algorithm does not check whether there exists enough capacity (stacks) so that overstowage can be avoided.

Let us now examine how the condition (9.2) leads to an algorithm for which we can prove better bounds. Suppose that we relax the constraint on the number of available stacks; that is, we can use more than L stacks (actually we will not need more than L+M stacks). Then, we can apply a column-wise per destination stacking policy. Let

$$L_i = \sum_{j=i+1}^{M+1} \left\lceil l_{ij} \right\rceil \ , \ i = 0,\ldots,M \qquad (9.5)$$

be the number of stacks it is necessary at port i for such an assignment policy to be feasible. Obviously, if some L_i is greater than L, the column-wise per destination policy cannot be used without modifications. Assume now that for some i, $L_i > L$. However, we know that we can combine the fractional (i.e. partially filled) stacks and get a feasible assignment because (9.3) always holds.

At port i we have, at most, M+1-i (=m_i, for simplicity) fractional stacks. Suppose that we are x stacks short. Consequently we must combine the m_i fractional stacks into (m_i-x) ones. If we want to rearrange the minimum number of containers in this merging process, then we can assume that we reassign the container belonging to the x less filled stacks to the remaining m_i-x. Again, this is feasible because of (9.3). Of course, this reassignment destroys the underlying idea of the column-wise per destination assignment policy. To maintain the properties of this policy, we agree to take off the stacks the containers of the x stacks that are reassigned to the m_i-x stacks above, so that a column-wise per destination

assignment policy is still possible at the port. Again, if $L_{i+1} > L$ the same merging process is repeated.

Let us suppose that the average number of containers per stack, for the x less filled stacks, is y. Then xy is the total number of containers to be rearranged and placed on the other (m_i-x) partially filled stacks. The average number of containers per stack, for the latter (m_i-x) stacks, is at least y (by assumption). Then the number of the available positions in these stacks is, at most, $(m_i-x)(b-y)$. The following condition should then be satisfied:

$$x.y \leq (m_i-x) \cdot (b-y) \qquad (9.6)$$

The maximum possible number of rearranged containers can be found by solving the following maximization problem.

maximize: $x.y$

subject to: $x.y \leq (m_i-x)(b-y)$ (9.7)

$x \geq 0$

$y \geq 0$

The solution of the above problem is simple. The maximum is $m_ib/4$, and, is achieved for $x = m_i/2$, $y = b/2$. Then the total number of such rearrangements is

$$2 \cdot \sum_{i=0}^{M} \frac{m_i\, b}{4} = 2 \cdot \sum_{i=0}^{M} \frac{(M+1-i)b}{4} = \frac{(M+1)(M+2)b}{4} \qquad (9.8)$$

The above rearrangements take place only if L_i is greater than L. This implies that there exist, at least, $(L-m_i)$ containers on board at port i. If port K is the last port at which rearrangements are required, then a lower bound for the number of

containers handled (carried) is (L - M - 1 + K)b. The number of rearrangements is

$$\frac{(k+1)\cdot(k+2)}{4} \cdot b$$

Then, an estimate of the ratio (overstowage cost)/(number of containers carried) is:

$$p_h = \frac{(k+1)(k+2)}{4(L-M-1+k)} \qquad (9.9)$$

The maximum of the ratio in (9.9) occurs for K = M. So, we have

$$p_h = \frac{(M+1)(M+2)}{4(L-1)} \qquad (9.10)$$

This ratio gives an upper estimate on the number of container rearrangements in addition to the usual pick up (load) and delivery (unload) operations. The table of Figure 9.4 shows how this ratio varies as a function of M/L and M.

M/L \ M :	3	4	5	6	8	10	15	20
1/100	1.7%	1.9%	2.1%	2.3%	2.8%	3.3%	4.5%	5.8%
1/50	3.4%	3.8%	4.2%	4.7%	5.6%	6.6%	9.1%	11.6%
1/20	8.5%	9.5%	10.6%	11.8%	14.2%	16.6%	22.7%	28.9%
1/10	17.2%	19.2%	21.4%	23.7%	28.5%	33.3%	45.6%	58.0%
1/5	35.7%	39.5%	43.8%	48.3%	57.7%	67.3%	91.9%	116.7%

Figure 9.4

Upper Bound of p_h (percent) as Function of M/L and M

The results of Figure 9.4 are very interesting. It appears that the approach we have discussed in this section performs really well for small values of M/L. This is

exactly the case in the containership problem. Figure 9.1 indicates that M/L is basically with the range 1/10-1/50, and M can vary from 3 to 10 (again, these are only indicative values). So, the above approach, even applies without further refinement, guarantees not bad upper bounds. To appreciate how good the approach is, we must bear in mind that these are worst case bounds, and also that the optimal overstowage cost is most probably greater than zero, too. In addition we most probably undercount the number of containers that the vessel carries between the ports of the port series. We must also realize that these results are good because the problem becomes easier as M/L gets smaller. In that line of thought, and given the simplicity of our approach, the results of Figure 9.4 can be taken as an indication of the difficulty of the problem. Notice that the results are independent of the stack capacity, b.

The above analysis is summarized in the algorithm of Figure 9.5 (COOP-2).

Algorithm COOP-2

for i: = 0 to M do

begin

Step 1: Unload containers from stacks with more than one type of containers. Do not unload the lower placed type;

Step 2: Assign new and rearranged containers column-wise per destination, except from the partially filled stacks if more than L stacks are required; in the latter case continue to Step 3, else jump to Step 4;

Step 3: Reassign the containers of the x less filled stacks to the rest of the partially filled ones. x is the number of stacks in excess of L;

Step 4: Continue;

end;

Figure 9.5 Algorithm COOP-2

The running time of COOP-2 is as follows:

Step 1: O(Mb), because there at most Mb/4 containers reassigned

Step 2: $O(n_i)$, $n_i = \sum_{j=i+1}^{M+1} c_{ij}$ containers picked up at port i

Step 3: O(Mb), the same as STEP 1

In total, the algorithm runs in $O(M^2b+n)$, where n is the total number of containers

$(n = \sum_{i=0}^{M} n_i)$

There is enough room for improving COOP-2. In particular the reassignment of the container of the partially filled stacks can be performed in a more sophisticated way. Also, a set of empirical rules may be helpful. For example, it is always optimal to place at the bottom of the stacks containers of type (M+1). This means that whenever there are empty stacks and containers of type (M+1) to be assigned, it is optimal to follow a row-wise by destination policy for the containers of type (M-1).

CHAPTER 10

OTHER OVERSTOWAGE PROBLEMS

This chapter deals with a different source of overstowage. Let us assume that there exist M tools that are stored in boxes and that the boxes are stored in a stack. The tools are under regular use, so when a tool is needed it is retrieved from the stack. Obviously, if the tool is not in the top box of the stack, other boxes must be retrieved first, incurring some rearrangement cost. Given certain information about the demand for each tool, an interesting question is to find the best rearrangement policy that minimizes overstowage costs. The latter can be a function of the number of rearrangements or of the weight of the tool or of the time it takes to retrieve it, etc. The problem is called a "use-and-restack" overstowage problem because the items (tools, boxes) return to the stack after being used.

Many versions of this problem can be defined. First, the stacks can be more than one. Secondly, the demand for the tools can be deterministic or probabilistic. Thirdly, different assumptions about the operation of the system can be made. For example, it may not be allowed to have two tools operating at the same time. Furthermore, placement constraints may apply too. In this chapter we examine some of the versions of the static overstowage problem. First, we look at the problem where the demand for tools is deterministic and known. They we go to the case in which tools are demanded one at a time with known frequency. Some generalizations follow. All the versions examined here refer to the one stack case.

10.1 The "Use-and-Restack" One-Stack Overstowage Problem with Deterministic Demand

Let us assume that there exist M tools. Suppose that the order by which they are going to be used is known. Also known is the time, di, a tool is used. No assumption about t_i's is made and no restriction on the number of tools simultaneously used is imposed. Let t_i be the time of the i^{th} request. K(i) indicates which tool is requested. Then at time t_i tool K(i) is retrieved and used for d_i time. Following its use, and at time $t_i + d_i$, tool K(i) returns to the stack. We also assume that the demand schedule is feasible.

The objective is to minimize the number of necessary tool rearrangements to satisfy the demand requests. So, we want to find what rearrangements and when should be performed to achieve the above objective.

To solve the static problem, we are going to use the algorithm we have developed for the dynamic case. In fact, the static problem is going to be transformed to a dynamic one, as follows.

As mentioned above, at times t_i there is a request for delivery of one tool of type K(i). Similarly, at t_i+d_i there is a return of one tool of type K(i). The correspondence to the deliveries and pick-ups of the one stack overstowage problem is obvious. We assume that at t_i one tool of type K(i) is delivered; at ti+di a new item of the same type is picked up. The "destination" of this new item is the first next time moment that tool K(i) is scheduled to be used. Figure 10.1 shows the transformation.

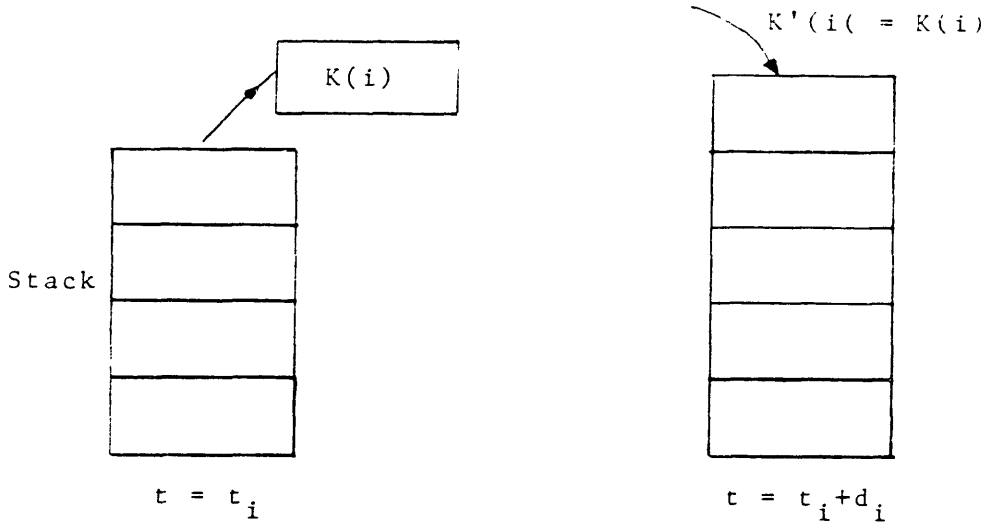
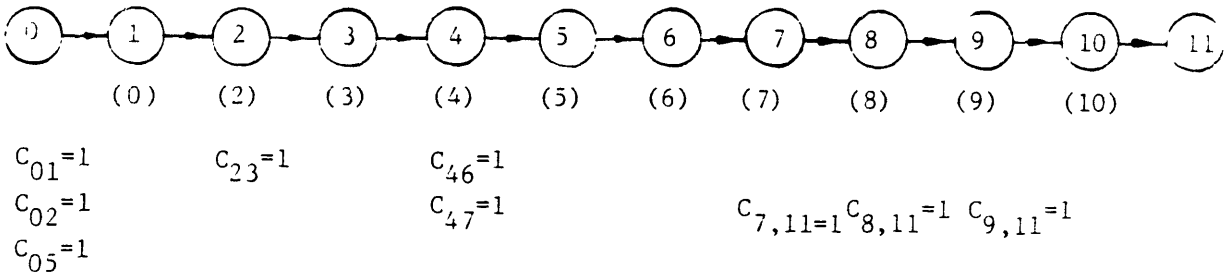


Figure 10.1 - Treatment of Retrieval and Storage of Items

The following example illustrates the process. In the example, let us assume that T=3 (three tools), and there exists six jobs to be done. The tool and time of use requirements are:

Tool	1	2	1	3	2	1	
Start	0	2	3	5	6	7	[time units]
Duration	2	2	1	3	1	2	[time units]
End	2	4	4	8	7	9	[time units)

In the above example there are 9 distinct time moments, 0, 2, 3, 4, 5, 6, 7, 8, 9. At each of these times, either a request or a return of a tool takes place. Let us introduce a series of 11 ports (= 2+9). In parenthesis, the correspondence to the time moments is given



In the above example we see how the six-operation long horizon has been transformed to a very sparse shipment matrix like the matrices encountered in solving the OSOP. By running the OSOP algorithm on the resulting shipment matrix we obtain the rearrangement policy that minimizes the cost of rearrangements. Figure 10.2 contains the algorithm that does the transformation of requests for tools to container shipments, solves the resulting OSOP, and translates the solution back to the "use-and-restack" problem case input.

Algorithm STACK

Input: T tools, N tool requests, times t_i of each request, tool requests k(i), duration of use d_i, i=1,...N.

Output: Port series, 0,1, M,M+1; shipment matrix C.

Step 1: List times of tool requests, t_i, and times of tool returns, t_i+a_i.

Step 2: Merge the two lists and sort the resulting one. Let $l(t_i)$ describe the order of time t_i in the resulting sorted list. Set l(0) = 0.

Step 3: For each tool, j = 1, T construct the sequence of time moment that is requested or returned, t^j_i, i = 1,...,$2N_j$, where N_j is the number of requests for tool j. Obviously

$\sum_{j=1}^{T} N_j = N.$

Let $t_0 = 0$.

Step 4: For i = 0 to M do
for j = i+1 to M+1 do
Cij = 0.

Step 5: For j = 1 to T to
For i = 1 to N_j do

$C_{l(2i-1),l(2i-1)} := C_{l(2i-2),\, l(2i-1)} + 1;$

Step 6: Run algorithm REARRANGE with input M, C; (the one-stack overstowage problem algorithm). Let R* be the optimal rearrangement policy.

Step 7: For the optimal policy found compute what groups are rearranged at each port; y(i,j,k) is one if group h,j is rearranged at port k, zero otherwise.

For k = 1 to M do
l = K;
Repeat (l = l-1) until (P*(l)≥P(k) or l=0);
for j = (k+1) to P*(k) do
for i = 0 to (k-1) do h(i,j,k) = 1;
for j = P*(k)+1 to (M+1) do
for i = (l+1) to (M-1) do y(i,j,k) = 1;

Step 8: For j = 1 to T do
for i = 1 to N_j do
for k = l(2i-2) to l(2i-1) do
if y(l(2i-2), l(2i-1),k = 1 then
tool j is rearranged at time $l^{-1}(k)$;

Step 9: The initial ordering is according to the time of first use of tools, j = 1, T.

Figure 10.2 - Algorithm STATIC

The following theorem can be easily proven.

Theorem 10.1

Algorithm stack correctly solves the static one-stack overstowage problem in $O(N^3)$ time.

Proof

Suppose that the statement of Theorem 10.1 is not true. That is there exists a rearrangement policy that achieves lower rearrangement cost than algorithm STACK. Notice though that the transformation of steps 1 to 5 is uniquely defined. In fact this transformation is nothing more than a renumbering of time moments and a renaming of tools at different time moments. So the transformation of steps 1 to 5 always produces the same shipment matrix C. The rearrangement policy can, in turn, be expressed in terms of the renamed and renumbered problem (OSOP). If the cost of the assumed policy is less than that produced by algorithm STACK, this would mean that algorithm REARRANGE does not produce the optimal cost solution for the OSOP. But this is a contradiction as algorithm REARRANGE correctly solves the OSOP.

The running time of the algorithm is dominated by steps 6, 7, and 8. Each of these steps takes 0(M3), and $O(MN^2)$ time. Since $M \leq 2N$, the algorithm runs in $O(N^3)$ time. ∎

10.2 The "Use-and-Restack" OSOP with Probabilistic Demand: One-Request Look-Ahead Policies

Let us now assume that only one tool can be in use at any moment of time. In addition, the tools are demanded according to a known frequency p_i, i = 1,..T. After a tool is used, it returns to the stack. The next tool request is assumed independent of the previous and follows the same probability distribution.

When a tool is requested it has to be retrieved from the stack. If there exists overstowed tools, they have to be removed to allow access to the one requested. When a tool returns to the stack, it can be placed on the top of the stack with no other rearrangements or in lower positions if the top containers are rearranged.

Let w_i, i = 1,..T be the cost of rearranging (moving) tool i. A legitimate objective in this case is to minimize the expected rearrangement cost due to overstowage. The latter depends on the profile[1] of the stack at the time a tool request is made. So the initial stacking order should be determined along with a policy of what to do when a tool returns to the stack.

There exists a profile that minimizes the expected cost of removing overstowed items if only one tool request is to be made. This profile can be found as follows.

Let us consider two stack profiles that are similar except that two tools, tool i and tool j, are in reverse order. Let, also, m_T denote the bottom tool in the stack, m_{T-1} the tool above it, and so on; m_1 is top tool in the stack. The two profiles we consider look as follows:

[1] Arrangement of tools in the stack.

Profile 1: $m_T, m_{T-1},...,i,j,...,m_2, m_1$

Profile 2: $m_T, m_{T-1},...,j,i,...,m_2, m_1$

Figure 10.3 - Stack Profiles

Then, the expected cost of removing the overstowed tools can be written as

$$RC(m_T, m_1) = \sum_{k=1}^{T} w_{mk} \left(\sum_{l=k+1}^{T} p_{ml} \right) \qquad (10.1)$$

In 10.1 we omit the cost of retrieving tool m_K, because this is going to occur anyway.

If we apply (10.1) for the two profiles of Figure 10.3 and take their difference by subtracting the cost of the second from the cost from the first we have

$$RC(\text{profile 1}) - RC(\text{profile 2}) = p_i \cdot w_j - p_j \cdot w_i \qquad (10.2)$$

Consequently, profile 1 is better (less costly) in terms of the expected cost, if the sign of (10.2) is negative. Otherwise profile 2 is preferred. This means that tool is placed lower in the stack than tool j, if

$$\frac{w_j}{p_j} < \frac{w_i}{p_i} \qquad (10.3)$$

By starting from any stack profile and repeatedly applying the condition of (10.3) we end up with the best one. In fact, we simply have to order the T tools in decreasing order of the ratio w_i/p_i, i=1, T. The tool with the smaller ratio corresponds to the bottom of the stack. This operation takes O (T.log T) time. The above proves the lemma.

Lemma 10.1

The order of stacking T tools that minimizes the rearrangement cost in the case that only one tool is ever used, is the one that corresponds to ordering the tools in decreasing order of w_i/p_i, i=1, T, from the bottom towards the top of the stack, where w_i is the cost of removing tool i, and p_i the probability of using tool i.

■

The above analysis presumably takes place before any tool has been stacked. Then the procedure described in Lemma 10.1 determines the best profile. If, however, some tools have already been placed onto the stack, the cost of removing from the stack should be accounted for. First for the tools off the stack, Lemma 10.1 can be used to determine their relative ordering. Then the question is whether it pays to remove some of the already stacked tools and order them along with the others according to Lemma 10.1. Suppose that we decide to remove the top of the already stacked tools, and suppose that the tool ends up K positions higher in the resorted substack (see Figure 10.3). Let w_i, p_i, k be the removal cost and probability of request for the k bottom of the yet unstacked tools and w_{top}, p_{top} for the top of the already stacked ones.

Then the expected gain (or less) is:

$$g_1 = p_{top} \left(\sum_{i=1}^{k} w_i\right) - \left(\sum_{i=1}^{k} p_i\right) \cdot w_{top} - w_{top} \qquad (10.4)$$

The last term in (10.4) is the cost of removing the top tool in order to resort it along with the others.

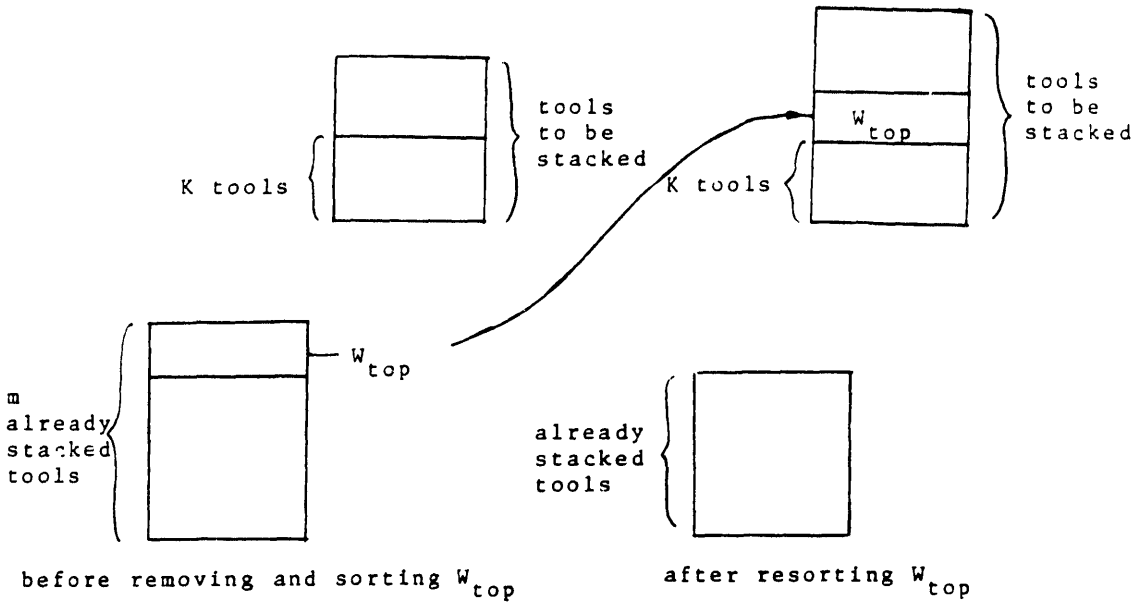


Figure 10.3 - Tool Rearrangements

If the sign of the expression is negative, the expected gain is actually a loss. However, this does not exclude the possibility of having a gain by rearranging more than one tools. The corresponding expected gains (losses) can be computed in an incremental way, by removing and rearranging the already stacked tools one by one. For example, if g_1 corresponds to the expected gain, when the top tool is rearranged, then g_2 can be defined to correspond go the expected gain if a second tool is rearranged. Even if g_1 may be negative, $g_1 + g_2$ can be positive. To get the entire picture, we must examine rearranging all m tools already stacked. Then we choose to rearrange the number of tools that maximizes the expected gains, that is maximize the function

$$g(x) = \sum_{i=1}^{x} g_i, \quad x = 0,1,\ldots,m \tag{10.5}$$

Let us now examine what is the proper action when a tool comes back for restacking (after the use of it has ended). If we only concern with the expected rearrangement cost of the next tool request, that is if we look only one request ahead, the optimal rearrangement policy can be found as follows.

Between the time of a tool request, say T_o, and the time when the tool returns for restacking, say T_1, two restacking operations take place: one at T_o when the tools that were rearranged to deliver the requested one are placed back on to the stack, and one at T_1 when the returning tool is restacked. Figure 10.4 introduces some notations:

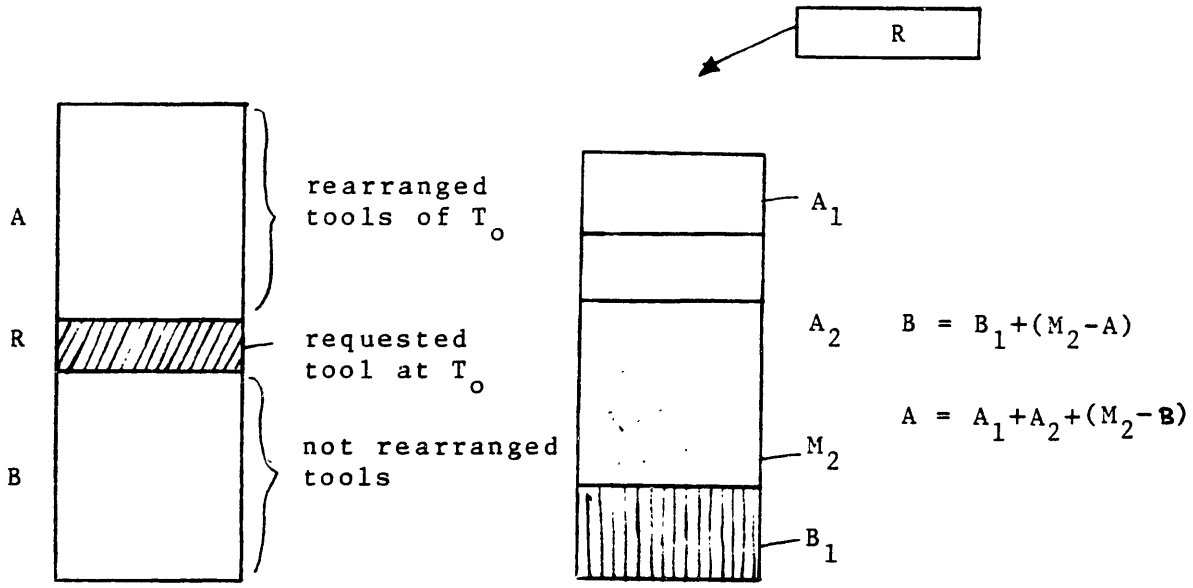


Figure 10.4 - Stack Rearrangements

Let A be the set of tools rearranged at T_o to deliver tool R. Let B be the set of tools that do not block tool R. When tools of set A are placed back onto the stack of T_o, they are placed according to (10.5). This means that all tools of set A are order by decreasing ratio (w_i/p_i). Depending on the value of x that maximizes (10.5), there can be a number of top tools of set B that are taken off the stack and join set A. These tools are represented by the set $(M_2\text{-}A)$[2].

Let $|A_1|$ be the number of tools of set $A+(M_2\text{-}A)$ that lead to maximization of (10.5) at time T_1, the time when tool R returns. Notice, though, that there may be tools that would pay to be switched with tool R, were they placed right below the tools of set A_1, despite that such a placement may contradict the w_i/p_i ratio ordering. If Q is tool to be switched and if Q "jumps" over the set X of tools (X, Q $\subset$ $(A_2 + M_2)$, the above is true as shown in Figure 10.5.

$$g_q = P_o w_2 - P_R w_a - w_q - (P_x w_a - p_q w_x) > 0 \qquad (10.6)$$

that is, when the overall expected gain is positive. Tools of set B_1 involve a similar calculation except for the fact that it needs to be done in an incremental basis (as in (10.5)). Every time a tool from set B_1 joins the tools of set $(A_1 + A_2 + M_2)$ the entire process needs to be recomputed starting from determination set A_1. Then the overall expected gain is computed given that y tools of set B_1 are rearranged.

$$g^B(y) = \sum_{i=0}^{y} g^B_i \;,\; y = 0,1,\ldots,|B_1| \qquad (10.7)$$

[2] According to set theory $(M_2\text{-}A)$ is a set containing all elements of M_2 not belonging to A.

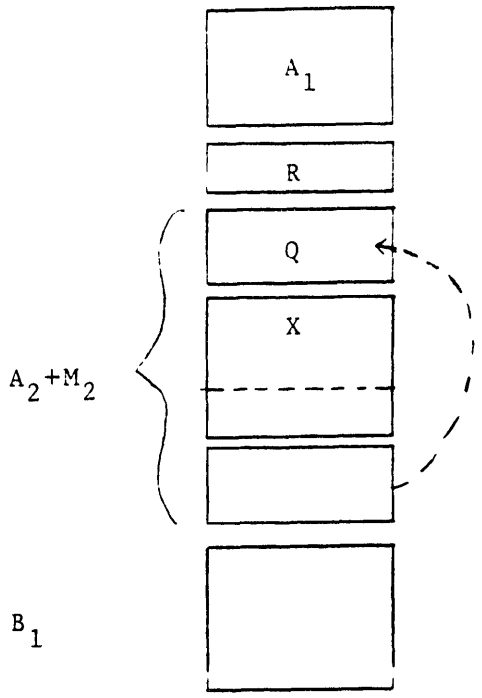


Figure 10.5 - Preventive Tool Switch

where g^R_i the incremental gain (or less) of rearranging the i^{th} top tool of set B_1 (in addition to the first (i-1)). The above analysis is presented in algorithm STATIC-1 in Figure 10.6. The following theorem can also be proven.

Theorem 10.2

Algorithm STACK-1 correctly solves the static one stack overstowage problem with probabilistic tool requests, when only the expected rearrangement costs of the next tool request are considered in $O(T_2)$.

Proof

The correctness is proven as follows. Given the number of the tools (top) among those still onto the stack that are rearranged, the optimal stacking order is that determined by the ratio rule (decreasing order of w_i/p_i), except from those tools that correspond to positive savings (gain) in (10.6). Algorithm STACK-1 checks all possible cases: for each number of tools (among those still onto the stack) it examines whether (10.6) results in positive gain. There are $O(T^2)$ combinations to be checked. Ordering according to the ratio rule takes 0(T log T) the first time and O(T) for each additional insertion. So the running time is $O(T^2)$ and the algorithm correctly solves the problem. ∎

Algorithm STACK-1

Inputs: Stack profile request for tool of the stack (tool R) probability distribution p_i of next request, total rearrangement cost w_i, i=1,...T ; T number of tools.

Output: Rearrangements and stack profile that minimize expected cost of next request.

Step 1: Initialization
- A = set of tools above tool R, in given profile
- B = stack of tools below tool R
  |.| : cordinality of set (i.e. |A| cardinality of set A)
- $w_2(0)$: = 0;

Step 2: For j: = 0 to B do

2.1 - remove top element of stack B and add it to set A
2.2 - sort elements of A according to decreasing w_j/p_j;
2.3 - renumber elements of A so order corresponds to
2.4 - numbering (the bottom element - highest w_j/p_j - is the

2.5 - $g(o):=0$; $w_1(j,0) = 0$;
- for i:=1 to $|A|$ to

$g(i) - g(i-1) + p_i w_r - p_r w_i - w_i$; $w_1(j,1) = w_1(i)+w_i$

2.6 - Let L_{max}: $g(t_{max}) = \max_{0 \le i \le |A|} g(i)$

2.7 - for i: = $i_{max}+1$ to $|A|$ do if

$$(p_i \cdot w_r \quad P_r w_i - w_i - (\sum_{k=i_{max}+1}^{i-1} p_k \cdot w_i - p_i \sum_{k=i_{max}+1}^{i-1} w_k > 0)$$

then {move element i right above $i_{max}+1$;
$w_2(j) = w_2(j)+w_i$; erase i from below $i_{max}+1$;

2.8 - insert element R (tool R) right above $i_{max}+1$;
The resulting profile is the best for the given j

2.9 - compute expected rearrangement cost

$$RC(j) = \sum_{k=1}^{T} w_k \sum_{l=k+1}^{T} p_e + \sum_{k=0}^{j} w_j + w_1(j, i_{max}) + w_2(j)$$

Step 3: Choose j_{max} : $RC(j_{max}) = \min_{0 \le j \le |B|} RC(j)$;

Step 4: End.

Figure 10.6 - Algorithm STACK-1

It is worth mentioning that the algorithm of Figure 10.7 tends to bring the profile of the stack as "closer" to the one corresponding to ordering the tools according to the decreasing (w_i/p_i) ratio rule (from bottom to top of the stack). The existence of already-stacked tools leads to trade-offs between the tool rearrangement cost to achieve the decreasing ratio ordering and the expected costs of the next tool request.

The question, of course, is whether the one-request look-ahead policy analyzed above is optimal in the long run. This is not discussed in this thesis because of the limited time available for its completion. We simply mention here that different assumptions about the problem horizon can be made. If a finite horizon is assumed the objective will be to minimize total expected costs. However, if the horizon is assumed infinite the total expected costs are infinite too. Then we should turn to minimizing average costs per stage.

10.3 The "Use-and-Restack" Overstowage Problem: Generalizations and Comments

The previous two sections have dealt with two versions of the "static" overstowage problem. The nature of the problem is very rich giving rise to a variety of different versions that find applications in different fields from textile production machines to military applications.

It has been our intention to give a brief introduction to this problem, since it represents another aspect of the overstowage phenomenon. In fact very interesting versions of the probabilistic demand case arise in presence of many stacks. In such a case, we may have the option of placing rearranged tools in other stacks. We suspect, however, that this is a difficult problem (as all multistack cases have turned to be). Nevertheless we believe that this deviation of the container overstowage problem has been useful and interesting. In addition, the full spectrum of overstowage situations has been examined.

CHAPTER 11

OVERVIEW OF OVERSTOWAGE PROBLEMS AND SUGGESTIONS FOR FURTHER RESEARCH

Overstowage is a common phenomenon arising in all kinds of stacking operations. Surprisingly, it has not been an area subjected to mathematical analysis, and, the problem of minimizing it has not been solved with analytical but rather empirical tools. To the best of our knowledge this thesis is the first methodological treatment of the problem. The motivation has come from containership operations, but it has been soon realized that the problem comes up in many other applications.

The focus of the thesis has been the "pickup-and-delivery" overstowage problems, much like the containership or warehouse or parking garage operations problems. Other overstowage problems ("use and restack") have been briefly introduced and examined. An example of the latter is the tool usage application discussed in Chapter 10.

The "pickup-and-delivery" stacking problems have been examined in an order of increasing difficulty. We started by examining the one-stack problem which was solved to optimality. Sensitivity analysis has also been performed and certain extensions of the one-stack case have also been solved satisfactorily. The most important of the latter is two incorporation of stability constraints for the stack. Such constraints have not been considered in the analysis of the multi-stack problem. Our analysis of the latter problem has not yielded as nice results as that of the one-stack case. We have conjectured that the problem is NP-complete and

we have turned our attention to developing heuristic algorithms. It has been found out that containership operations are moderately prone to overstowage due to the large value of the ratio (number of stacks to number of destinations). An upper bound on the overstowage cost as a percent of the number of containers "picked up-and-delivered? has been found to be less than 30% in the worse case; this bound is much less than 30 % (around 10%) for most situations arising in containership operations. The heuristics provided for the multi-stack problem are of greedy-type; they result in locally optimal solutions.

The "use-and-restack" overstowage problems are classified as deterministic or probabilistic tool demand problems. The former ones are transformed to "pickup-and-delivery" problems and solved as such. The latter can be solved by using dynamic programming models. In this thesis we have examined only one-step-look-ahead policies with the objective of minimizing the expected rearrangement cost.

This thesis has dealt with a hard combinatorial problem with applications in systems operations. A first big step towards the understanding and solution of such problems has been made. The results are satisfactory and encouraging. However, there is a lot more work yet to be done. Some suggestions and directions are given in the following sections, along with generalizations of the stacking concept and the definition of the overstowage problem.

11.1 Extensions and Suggestions for Further Research

There are certain areas in which further research is an immediate extension of the work presented in this thesis. The resolution of those questions is going to significantly contribute to the enhancement of our understanding of the overstowage problems.

In the one-stack overstowage problem, there are few advancements that can be made. An interesting one is the establishment of lower bounds for the optimal overstowage cost. Such bounds are of great help in the performance assessment of heuristic algorithms for multi-stack problems.

Another extension is the consideration of periodic shipment requirements among a finite set of ports. Usually a time horizon is associated to such a problem expressed in the number of cycles the vessel does. Obviously, one way to solve this problem is to "expand" it over time, and solve the resulting one-stack overstowage problem with the algorithm developed in Chapter 3. Such a transformation is shown in Figure 11.1.

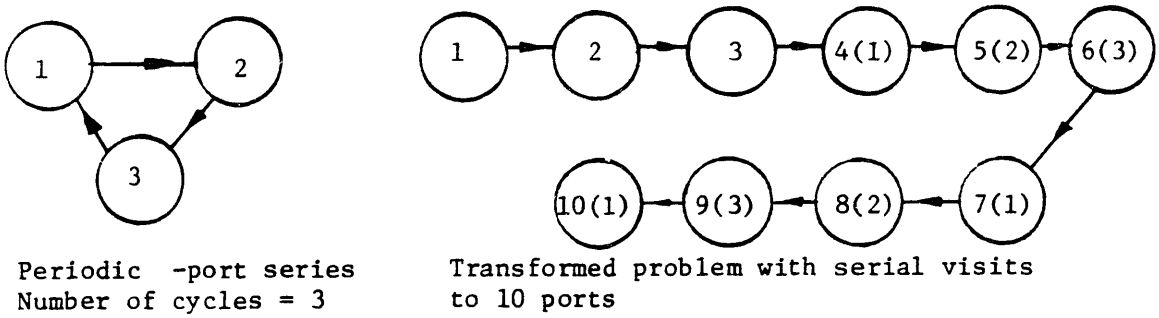


Figure 11.1

Transformation of Periodic Shipment Matrix Problems to Serial Port Series Problems

An analytical solution of the periodic problem not only reduces the computational requirements of the solution, but also is going to be useful in the "use-and-restack" situations as many applications of that sort have periodic demand requirements. Moreover it will help us to better understand the peculiarities of this sort of overstowage problem.

One more area of special interest is problems with probabilistic shipment matrix. In Chapter 5 we have made a first step towards this direction by assuming that one of the elements of the shipment matrix is known only up to its probability distribution. H is useful to extend the analysis to incorporate more general assumptions about when the elements of the shipment matrix become known and how the optimal rearrangement policy is affected.

The above extensions are not restricted in the one-stack overstowage problem only. They apply to the multi-stack case as well. For the latter problem, though, there exists another set of questions that await investigation. The most important of them, from the theoretical perspective, is the question of complexity: "Is the MSOP NP-complete?" The same question can be raised for the problem of assigning the containers to stacks under given rearrangement policies and finite stack capacities.

A point that has not been resolved in this thesis is the proof that the overstowage cost of the $R/\infty/MSOP$ is a concave function of the number of stacks.

In the area of heuristic algorithm design and analysis, there are several research efforts that can be undertaken both from the analytical and computational

viewpoint. Coding and testing of the proposed heuristics is as necessary as further development of the theoretical analysis of the multi-stack overstowage problem and its properties. Coming up with performances guaranteed for the heuristics is related to improving of our understanding of both the one-stack and multi-stack problems.

It is also worth examining alternative approaches such as branch-and-bound algorithms and lagrangean relaxation methods. It is our intuition, for reasons explained in Chapter 8, that at least branch-and-bound algorithms are not going to be very effective. It is necessary though to verify that conjecture. In addition to the above two methods is an alternative way of obtaining lower bounds for the overstowage cost of the MSOP. Of course, a simpler way to get such lower bounds is to drop the integrality of the solution requirement of the network problem shown in Figure 7.6.

In terms of heuristics especially designed for the containership operations case, one can improve the very simple technique proposed in Chapter 9 (algorithm COOP-2). An area of improvement in that algorithm is the reassignment of some of the partially filled stack. Such improvements can be achieved even under the assumption that the reassigned containers are rearranged in the next part of the series. In the latter case the constant multiplying the expression in (9.8) can be sliced down from 2 to a value closer to unity. A proportional reduction in the percentage of Figure 9.4 is then automatically achieved.

The "use-and-restack" problems is another area of possible research. These problems have not received the bulk of our attention in this thesis. There is a variety of problem versions that can be defined and solved. An immediate extension of the work presented in Chapter 10 is the consideration of solutions for the probabilistic demand problem not restricted to the class of one-step-look-ahead policies. Then, the objective can be the minimization of the expected average cost per tool request. The horizon of the problem can be infinite or finite. This problem calls under the theory of dynamic programming with partial information, and we expect that a low of concepts of that field can be applied to solve it. A multi-stack version of the "use-and-restack" problem can also be defined.

The above list of extensions of stack overstowage problems is by no means exhaustive. It refers only to immediate extensions of the concepts, ideas, and algorithms developed in the previous chapters, and it is subjected to the same set of assumptions. In the next section, we go one step further to generalize some of the concepts and define other problems of terest.

11.2 Generalizations and Other Applications

A special version of the MSOP can be constructed by interpreting c_{ij} somehow differently. Notice that in the uncapacitated versions of the MSOP (both restricted and unrestricted) containers of the same origin and destination and treated similarly. Consequently, c_{ij} (the number of containers with origin i and destination j) acts as a weighting factor. Thus, we can interpret c_{ij} as the weight of the single

item ship from i to j. The overstowage cost corresponds to the sum of the weights of the rearranged constraints. Let us now impose capacity constraints on the latter problem. These constraints do not have the same effect as they would in the R/MSOP or MSOP version, because they do not restrict the number of containers per stack, but the number of groups. Of course, if c_{ij} represents some kind of weight for the single container shipped from i to j then the two interpretations of the capacity constraint are equivalent. However, the new interpretation is quite useful. It may come up in a number of applications and so is worth examining.

Definition 11.1

g/MSOP and R/g/MSOP are the unrestricted and restricted versions of the multi-stack overstowage problem with stack capacity constraints on the number of groups of containers per stack. ∎

A very interesting extension of the one stack (but also of the multi-stack) problem is when the sequence of visits to the ports, or in general to the pickup and delivery ports, is also to be determined based on the minimization of overstowage and voyage costs. This is a very difficult problem, since it is a variation of the well-studied Travelling Salesman Problem (TSP) [11].

Another important aspect of overstowage is concerned with the physical implementation of the determined rearrangements. This problem appears in multi-stack operations only, and particularly in warehouse operations. It has been discussed by Christofides [5] and solved to a certain extent. What is extremely

interesting is not a better solution to the problem, Christofides has solved some fifteen years ago, but a solution to the combined problem of minimizing overstowage as defined in this thesis and the cost to implement the resulting rearrangements.

The concept of "stack" can be extended in various directions. First, let us introduce the "tree-stacks". These are stacks which have a main root stack which branches into a number child-stacks as shown in Figure 11.2. In fact, Figure 11.2 shows a physical implementation of a tree-stack.

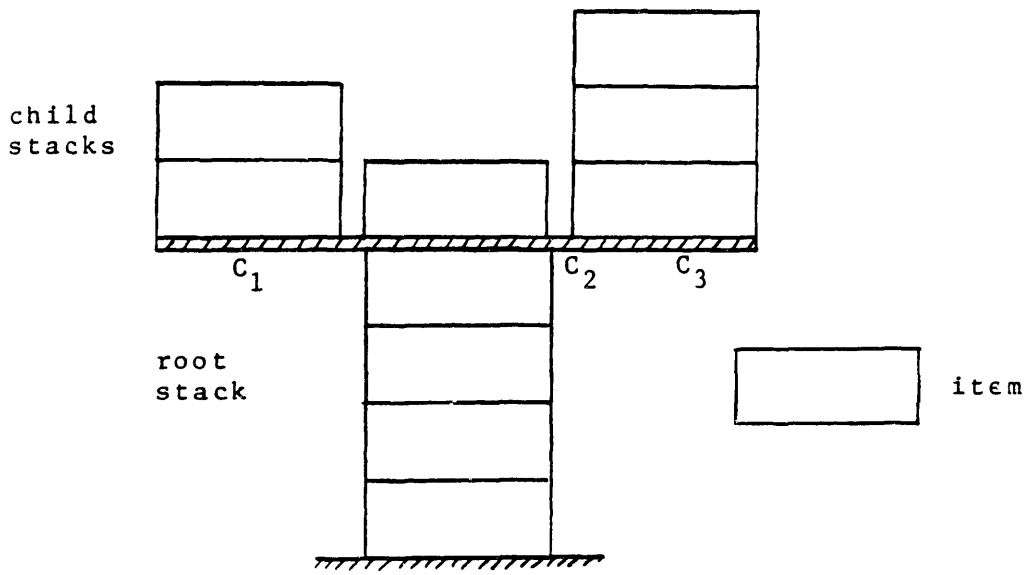


Figure 11.2 - The Concept of Tree-Stack

There is no restriction on the number of root stacks the configuration may have. So we can further generalize the tree-stacks to define multi-level stacks as in Figure

11.3. The idea in both the tree and multi-level stacks is that access to a "parent" stack, or a stack of lower level, can be obtained only if all child or higher level stacks are empty. For example, to acquire access to the root stack in Figure 11.2, we must first empty all child-stacks C_1, C_2, C_3. Such extensions in the definition of stack are useful in modelling stacking operations in containerships. As mentioned in the beginning of our analysis (Chapter 1), containers can be placed both above and below the deck of the vessel; access to stacks below the deck is possibly only when the deck above them is clear of containers. This situation is exactly what the multi-level stacks are built to model.

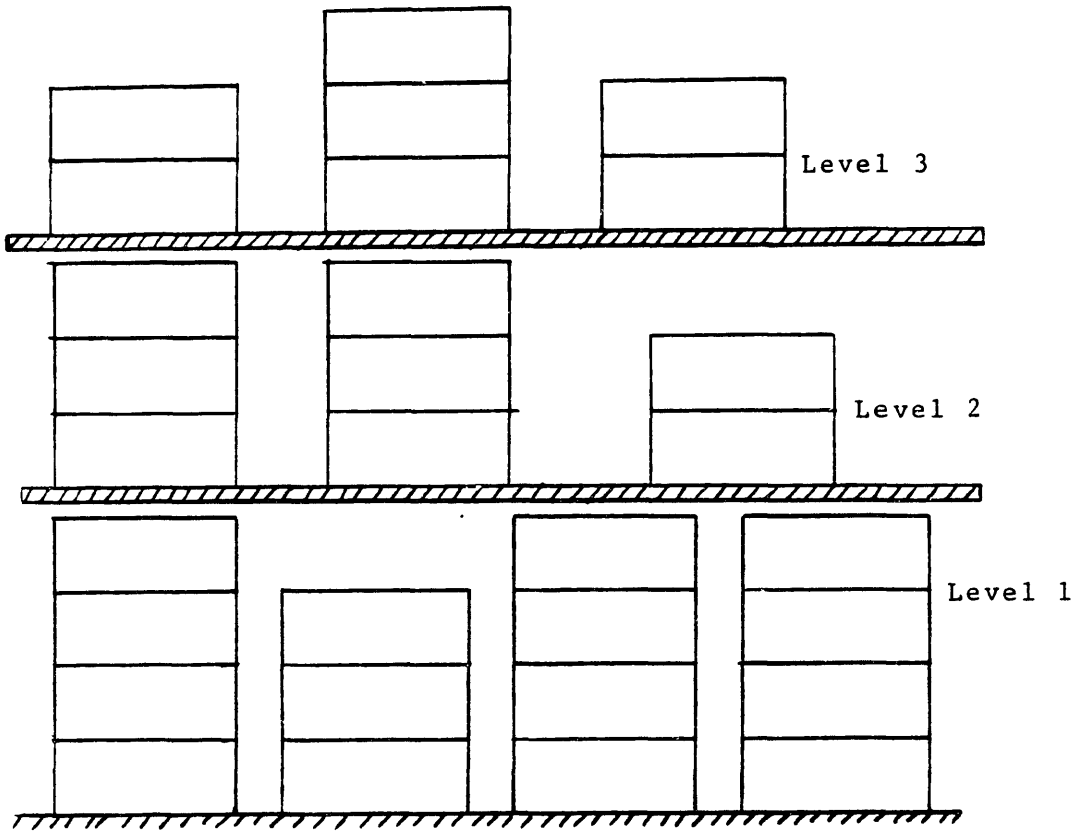


Figure 11.3 - Multi-Level Stacks

Another way of expanding the stack concept is by increasing the number of access points. Let us consider a single stack, and let us assume that items can be removed from the top (as usual), but also from the sides (Figure 11.4).

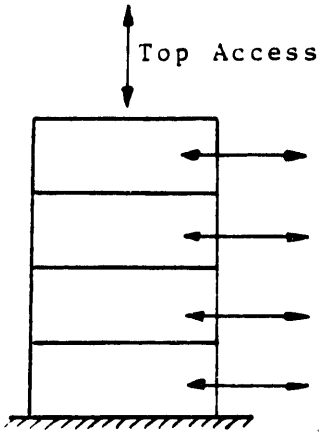


Figure 11.4 - Two-Way Access Stack

Apparently, the stack of Figure 11.4 can never experience overstowage because it behaves as many one-item-high stacks. The situation becomes more interesting though when many such stacks are placed next to each other (Figure 11.5). There we have a two-way-access stacking system. The shaded item in Figure 11.5 can be removed either vertically through the top of vertical stack 3, or horizontally through the left side of horizontal stack 4. In a similar manner, we can define 3-way-access stacking systems by allowing access through the third dimension, too. Such configurations find applications in stacking systems which have a system of physical cells to accommodate the containers.

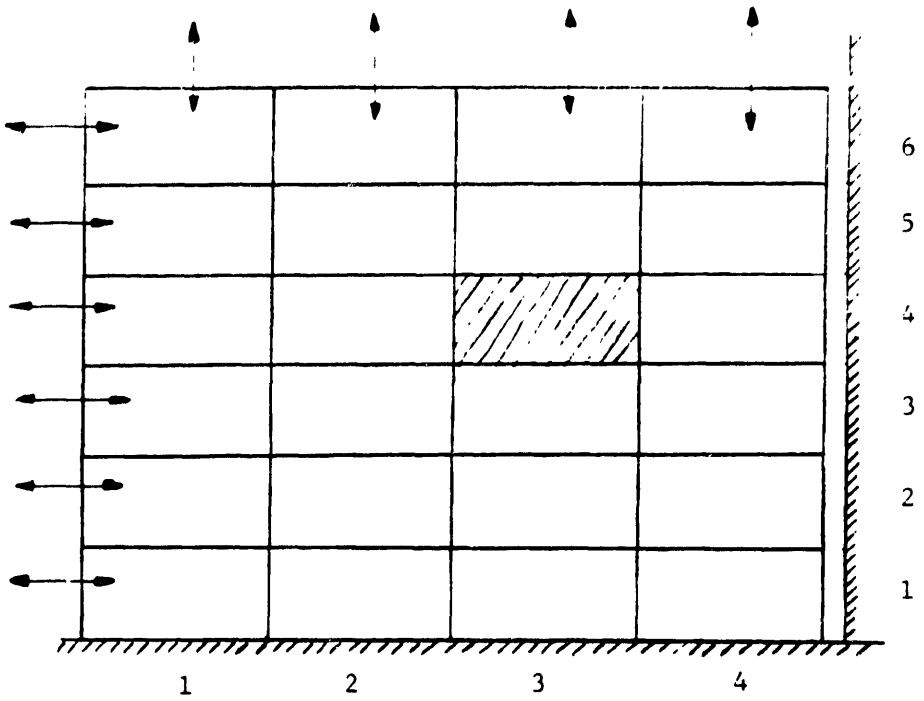


Figure 11.5 - Two-Way Access Stacking System

A compromise that has been done early in this thesis in studying overstowage in containership operations has been the exclusion from consideration of the stability and other hydrostatic requirements of the vessel. We have not considered the two levels of stacks, too. Stability requirements have been extensively discussed in the one-stack case (Chapter 6); their incorporation in the multi-stack problem is a challenge. It must be mentioned that we do not expect all such requirements to be binding, but any realistic implementation of the methods developed in this thesis should be able to make allowances for the satisfaction of the above requirements. It is not to be forgotten though that this is the most difficult version of multi-stack

overstowage problem, and we may have to settle for approximate solutions. What has been common in the past years (see Chapter 2) is to concentrate on the stability requirements and treat overstowage as a secondary problem. For the significant gains in port time that result if overstowage is properly managed we believe that the satisfaction of stability constraints and overstowage problems should be solved in parallel.

Further ambitions include coordination of overstowage handling between vessel and port container terminal. In fact, our discussion of Section 5.1 has been motivated by thoughts of minimizing (average) vessel time at port for the ports of a geographic region, where the relative economic handling costs (particularly, the levels of congestion) are taken into account.

CHAPTER 12

REFERENCES AND BIBLIOGRAPHY

1. Aho, A.V., Hopcroft, J.E., and Ullman, L.D. "Data Structures and Algorithms", Addion-Wesley Publishing Co., 1985.

2. Aslidis, A., "Optimal Container Loading", unpublished Master's Thesis, Massachusetts Institute of Technology, 1984.

3. Bertsekas, D.R., "Dynamic Programming: Deterministic and Stochastic Models", Prentice-Hall, Inc., 1987.

4. Christofides, N. and Colloff, I., "The Rearrangement of Items in a Warehouse", Operations Research, Vol. 20.

5. Christofides, N., Mingozzi, A., and Toth, P., "Dynamic Loading and Unloading of Liquids into Tanks", Operations Research, Vol. 28, No. 3, May-June 1980, pp. 633-649.

6. Drake, A.W., "Fundamentals of Applied Probability Theory", McGraw-Hill Book Co., 1967.

7. Eilon, S. and Christofides, N., "The Loading Problem", Management Science, Vol. 17, No. 5, January 1971, pp. 259-268.

8. Ford, L.K. and Fulkerson, D.K., "Flows in Networks", Princeton University Press, 1974, sixth printing.

9. Gorey, M.R. and Johnson, D.S., "Computers and Interactability: A Guide to the Theory of NP-Completeness", W. H. Freeman and Co., 1979.

10. Gillmer, T.C. and Johnson, B., "Introduction to Naval Architecture", Naval Institute Press, 1987, 3rd edition.

11. Ladany, S.P. and Mehrez, A., "Optimal Routing of a Single Vehicle with Loading and Unloading Constraints", Transportation Planning and Technology, Vol. 8, 1984, pp. 301-306.

12. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., (eds.), "The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization", John Wiley and Sons, 1984.

13. Papadimitriou, C.H. and Steiglitz, K., "Combinatorial Optimization Algorithms and Complexity", Prentice-Hall, Inc., 1982.

14. Shields, J.J., "Container Stowage: A Computer Aided Pre-planning System", Marine Technology, Vol. 21, No. 4, October 1984.

15. Scott, D.K., and Chen, Der-San, "A Loading Model for a Container Ship", November 15, 1978, Los Angeles, California (publication or presentation unknown).

16. Torjan, R.E., "Data Structures and Network Algorithms", SIAM CBMS Publication 44, 1986, fourth printing.

17. Webster, W.C. and Van Dyke, P., "Container Loading: A Container Allocation Model: I - Introduction, Background", presented at Computer-Aided Ship Design Engineering Summer Conference, University of Michigan, June 1970.

18. Webster, W.C. and Van Dyke, P., "Container Loading: A Container Allocation Model: II - Strategy, Conclusions", presented at Computer-Aided Ship Design Engineering Summer Conference, University of Michigan, June 1970.

APPENDIX I

CALCULATION OF FUNCTIONS $f_{km}(l)$

In this appendix, we explain how functions $f_{km}(l)$ are computed. As defined in Chapter 6, $f_{km}(l)$ is the improvement of x_k (vertical center of weight of the stack) at port k, with l additional rearrangements at port k given that m-l ones have been optimally performed at ports 1 to k-1. Since the stack profile is known (with the (m-l) rearrangements already performed), we have to solve the following problem.

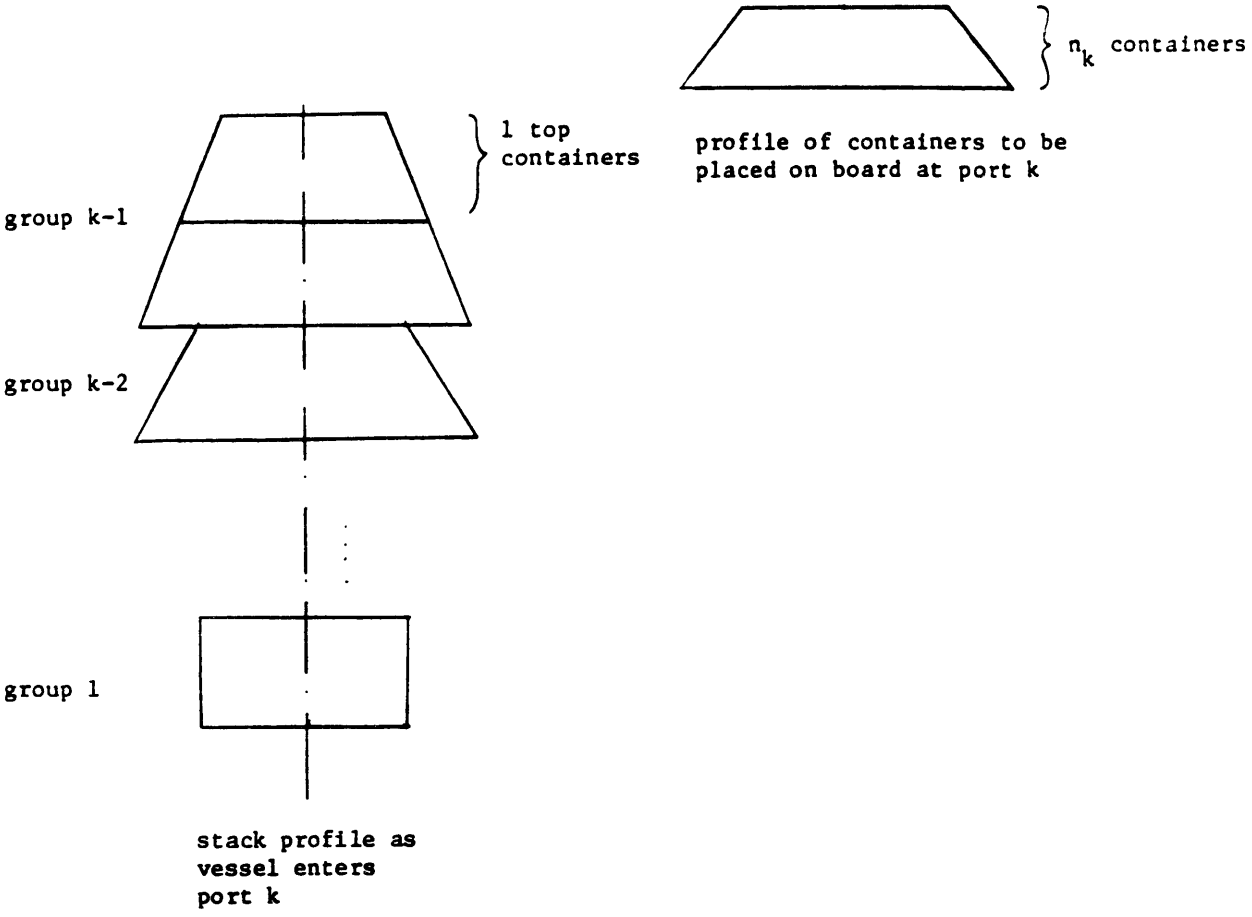


Figure A1.1 - Profile of Stack

The l additional rearrangements can involve no other than the l top container of the stack. The maximum decrease in x_k is achieved when the l containers we rearrange at port k, and the n_k new ones are placed back on the stack in decreasing order of weight. It takes $O(K_l.n_k)$ to sort the top l containers and the n_k ones. This is because when l is large enough that it spreads over more than one group (say k_l groups), we have to restart sorting from the top of the stack with the n_k new containers every time another group begins getting rearranged. Since we have assumed that the containers have the same height, it is straightforward to update the position of the c.o.w. In fact, the l containers to be rearranged can be done so, one by one. When a container is moved p positions higher, the moment of the containers with respect to the bottom of the stack goes down by

$$\{\Sigma(\text{weight of the p containers overpassed}) - p \cdot (\text{weight of container moved up})\} \cdot (\text{height of containers}) \qquad (A1.1)$$

This information can be retrieved in constant time, O(1), if we keep an array with the cumulative sum of the weight of containers from the bottom of the stack, along with the value of the moment before rearrangements start. The c.o.w. can be found as

$$x_k = \frac{(\text{moment with respect to the bottom of the stack}}{\Sigma\ (\text{weight of containers})}$$

or

$$n_k = \frac{\Sigma\ w_i \cdot y_i}{\Sigma\ w_i} \qquad (A1.2)$$

where, w_i is the weight of a container and y_i is the distance from the bottom of the stack.

So, overall, it takes $O(k_i.n_s)$ to sort the l containers, and $O(l.(l+n_k)$ to update the cumulative sum of container weights. Since $k_i \leq l$, the total calculation time is $O(l(l+n_k))$. The update of the stack's profile can be performed in parallel to the update of the cumulative container weights with no increase in the running time.

APPENDIX A2

COMPUTER CODE FOR THE OSOP ALGORITHM

In this appendix, the computer code of the one stack overstowage problem algorithm is presented. The code has been developed in TURBO PASCAL (program "one"). A sample page of output is also provided. In addition, a program that calculates the overstowage cost for any effective rearrangement policy is also included (program "schedule").

```
program one;

type

 problem_type = record
  start  : integer;
  finish : integer;
 end;

 lpointer = ^problem_node;
 problem_node = record
  problem : problem_type;
  next    : lpointer;
 end;

var

 M                : integer;
 i, j, k, l, temp : integer;
 o_port, d_port   : integer;

 c   : array[-1..20, 1..21]        of integer;
 d   : array[-1..20, 0..20]        of integer;
 o   : array[-1..20, 1..21]        of integer;
 B   : array[-1..20, 0..20, 1..21] of integer;
 A   : array[-1..20, 0..20, 1..20] of integer;
 R   : array[ 0..20, 0..20, 1..21] of integer;
 e   : array[ 0..21, 0..21]        of integer;
 cost: array[ 0..20, 1..21]        of integer;
 rhdl: array[ 0..21, 0..21]        of integer;
 f   : array[ 0..20, 1..21]        of integer;
 S   : array[ 0..20]               of integer;

 list : lpointer;
 subproblem : problem_type;

 infile : string[12];

 datafile1  : text;
 outputfile : text;


{******** FUNCTIONS & PROCEDURES *********¶

procedure HeadInsert (       NewData : problem_type;
                        var  Head    : lpointer
                      );
var PNew : lpointer;

begin
 new(PNew);
 PNew^.problem := NewData;
 PNew^.next    := Head;
 Head          := PNew;
end;
```

```
{********** MAIN PROGRAM **********}

begin
 clrscr;
 write(' enter name of input file:');
 readln(infile);

 assign(datafile1,infile );
 assign(outputfile, 'ONE.OUT' );
 reset(datafile1);
 rewrite(outputfile);

 readln(datafile1);
 readln(datafile1);
 readln(datafile1, M);
 readln(datafile1);

 for j := 1 to  ( M + 1 ) do
  begin
   for i := 0 to ( j - 1 ) do
    begin
     read(datafile1, c[i,j]);
     write('  c[',i:1,',',j:1,']=',c[i,j]:2);
    end;
   readln(datafile1);
   writeln;
  end;

 writeln;

 for i := 0 to 20 do
  for j := 0 to 20 do
   for k := 1 to 20 do
    R[i,j,k] := 0;

 for i := 1 to ( M + 1 ) do
  begin
   d[-1,i] := 0;
   for j := 0 to ( i - 1 ) do
    begin
     d[j,i] := d[j-1,i] + c[j,i];
     write('  d[',j:1,',',i:1,']=',d[j,i]:2);
    end;
   writeln;
  end;

 writeln;

 for i := 0 to M do
  begin
   o[i,i] := 0;
   o[i,M+1] := c[i,M+1];
   write('  o[',i:1,',',(M+1):1,']=',o[i,M+1]:2);
```

```
  for j := M downto ( i + 1 ) do
   begin
    o[i,j] := o[i,j+1] + c[i,j];
    write('  o[',i:1,',',j:1,']=',o[i,j]:2);
   end;
  writeln;
 end;


for i := 0 to M do
 for j := i to M do
  begin
   B[i,j,j] := 0;
   B[i,j,j+1] := 0;
   for k := ( j + 2 ) to ( M + 1 ) do
    B[i,j,k] := B[i,j,k-1] + d[i,k-1];
  end;

writeln;

for j := 1 to M do
 for k := ( j + 2 ) to M do
  begin
   i := 0;
   e[j,k] := -1;
   while ( ( B[i,j,k] = 0 ) and ( i <= (j-1) ) )  do
    begin
     e[j,k] := i;
     i := i + 1;
    end;
  end;


for i := 1 to M do
 begin
  A[i,i,i] := 0;
  A[i,i,i+1] := 0;
  for j := ( i + 1 ) to M do
   begin
    A[i,j,j] := A[i,j-1,j] + o[j-1,j+1];
    for k := ( j + 1 ) to M do
     A[i,j,k] := A[i,j,k-1] - d[j-1,k] + d[i-1,k];
   end;
 end;


for i := 1 to M do
 for j := i to M do
  for k := j to M do
   begin
    R[i,j,k] := A[i,j,j] + B[i-1,j,k+1];
    if ( ( A[i,j,k] = 0 ) and ( j > i ) and ( k > j ) ) then
     begin
      R[i,j,k] := R[i,j,k] - d[i-1,k];
      if ( e[j,k] >= (i-1) ) then
```

```
          R[i,j,k] := R[i,j,k] - B[e[j,k],j,k];
       end;
      end;


 for i := 1 to M do
  begin
   cost[i,i] := 0;
   cost[i,i+1] :=0;
  end;

 for l := 2 to M do
  for i := 1 to ( M + 1 - l ) do
   begin
    k := i + l - 1;
    cost[i,k+1] := cost[i,i] + R[i,i,k] + cost[i+1,k+1];
    rhdl[i,k+1] := i;
    for j := k downto ( i + 1 ) do
     begin
      temp := cost[i,j] + R[i,j,k] + cost[j+1,k+1];
      if ( temp < cost[i,k+1] ) then
       begin
        cost[i,k+1] := temp;
        rhdl[i,k+1] := j;
       end;
     end;
    writeln('cost[',i:1,',',(k+1):1,']=',cost[i,k+1]:3);
   end;

 for j := 1 to M do S[j] := 0;

 new(list);
 list^.problem.start := 1;
 list^.problem.finish := M + 1;
 list^.next := nil;

 repeat
  o_port := list^.problem.start;
  d_port := list^.problem.finish;
  list := list^.next;

writeln(o_port:3, d_port:3,rhdl[o_port,d_port]:3);

  S[rhdl[o_port,d_port]] := d_port - 1;

  if ( d_port - o_port >= 2 ) then
   begin
    subproblem.start := o_port;
    subproblem.finish := rhdl[o_port,d_port];
    if ( subproblem.finish - subproblem.start >= 2 ) then
       HeadInsert( subproblem, list );
    subproblem.start := rhdl[o_port,d_port] + 1;
    subproblem.finish := d_port;
    if ( subproblem.finish - subproblem.start >= 2 ) then
       HeadInsert( subproblem, list );
```

```
  end;

until ( list = nil );

for i := 1 to M do if ( S[i] = 0 ) then S[i] := 1;

for i := 0 to M do
 for j := 1 to ( M + 1 ) do
    f[i,j] := 0;

for k := 1 to M do
 begin
  l := k;
  repeat
   l := l - 1;
  until ( ( S[l] >= S[k] ) or ( l = 0 )  );
  for j := ( k + 1 ) to S[k] do
   for i := 0 to ( k - 1 ) do
    f[i,j] := f[i,j] + 1;
  for j := ( S[k] + 1 ) to ( M + 1 ) do
   for i := ( l + 1 ) to ( k - 1 ) do
    f[i,j] := f[i,j] + 1;
 end;

writeln;
writeln('Rearrangement cost = ', cost[1,M+1]:4);
writeln;

write('port    :');
for i := 1 to M do
 write('   ', i:3);
writeln;
write('schedule:');
for i := 1 to M do
 write('   ', S[i]:3);
writeln;

writeln;
writeln('Rearrangment results');
writeln;
for i := 1 to ( M - 1 ) do
 begin
  for j := ( i + 2 ) to ( M + 1 ) do
   write('  rhdl[',i:1,',',j:1,']=',rhdl[i,j]:2);
  writeln;
 end;

writeln;
writeln(' Partial derivatives of S, f(i,j), i=0,..M, j=i+1,...,M+1');
writeln;
for j := 1 to ( M + 1 ) do
 begin
  for i := 0 to ( j - 1 ) do
   write('  f[',i:1,',',j:1,']=', f[i,j]:2);
  writeln;
```

```
    end;
  close(outputfile);
end.
```

NOTE: Character ^ stands for ↑ (pointer).

```
program schedule;

var

 M                 : integer;
 i, j, k, l        : integer;
 cost              : integer;

 check             : boolean;

 c   : array[ 0..20, 1..21]        of integer;
 f   : array[ 0..20, 1..21]        of integer;
 S   : array[ 1..20]               of integer;


 datafile1  : text;
 outputfile : text;


{********** MAIN PROGRAM **********¶

begin

 assign(datafile1, 'ONE.DAT' );
 assign(outputfile, 'SCH.OUT' );
 reset(datafile1);
 rewrite(outputfile);

 readln(datafile1);
 readln(datafile1);
 readln(datafile1, M);
 readln(datafile1);

 for j := 1 to  ( M + 1 ) do
  begin
   for i := 0 to ( j - 1 ) do
    begin
     read(datafile1, c[i,j]);
     write('  c[',i:1,',',j:1,']=',c[i,j]:2);
    end;
   readln(datafile1);
   writeln;
  end;

writeln;

for i := 1 to M do S[i] := 0;

writeln(' enter rearrangement schedule;');
writeln(' (numbers should be greater than or equal to port number)');

repeat
 write( ' PORT    :');
```

```
 for i := 1 to M do write('  ',i:2);
 writeln;
 write( ' SCHEDULE:');
 for i := 1 to M do begin gotoxy(wherex+3,wherey); read( S[i] ) end;
 readln;
 writeln;
 check := true;
 for i := 1 to M do
  begin
  if ( ( S[i] < i ) or ( S[i] > M ) ) then check := false;
  end;
 if ( check = false ) then
        writeln('inconsistent schedule; please enter again');
until ( check );

for i := 0 to M do
 for j := 1 to ( M + 1 ) do
    f[i,j] := 0;

for k := 1 to M do
 begin
  l := k;
  repeat
   l := l - 1;
  until ( ( S[l] >= S[k] ) or ( l = 0 )  );
  for j := ( k + 1 ) to S[k] do
   for i := 0 to ( k - 1 ) do
    f[i,j] := f[i,j] + 1;
  for j := ( S[k] + 1 ) to ( M + 1 ) do
   for i := ( l + 1 ) to ( k - 1 ) do
    f[i,j] := f[i,j] + 1;
 end;

cost := 0;

for i := 0 to M do
 for j := 1 to ( M + 1 ) do
  cost := cost + f[i,j] * c[i,j];

writeln;
writeln('Rearrangement cost = ', cost:4);
writeln;

write('port    :');
for i := 1 to M do
 write('   ', i:3);
writeln;
write('schedule:');
for i := 1 to M do
 write('   ', S[i]:3);
writeln;


writeln;
writeln(' Partial derivatives of S, f(i,j), i=0,..M, j=i+1,...,M+1');
```

```
  writeln;
  for j := 1 to ( M + 1 ) do
   begin
    for i := 0 to ( j - 1 ) do
     write('  f[',i:1,',',j:1,']=', f[i,j]:2);
    writeln;
   end;

 close(outputfile);

end.
```

NOTE: Character ² stands for ↑ (pointer).