National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Optimization Methods in Logic Programming

Applied to Expert Systems for Capital Budgeting

by

Hastings Kyale Muli

Masters of Science thesis

Systems Science programme

University of Ottawa

ISBN 0-315-85832-X

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

This thesis evaluates the benefit of meshing mathematical programming and expert systems for solving capital budgeting problems, using constraint logic programming methods. A review of modelling capabilities of mathematical programs for capital budgeting, and of financial expert systems leads to defining the respective role and potential of each method, and to the proposal of a two-tiered project selection approach: project evaluation and resource allocation.

With emphasis placed on a tight coupling of the two tiers, logic programming is shown to be a language of choice to implement mathematical programming within an expert system shell. Prolog has the requisite properties to deal with both logical considerations and optimization problems. Although Prolog was not primarily designed to solve optimization problems, it is shown that the backtracking mechanism of the Prolog language is powerful enough for that purpose; it liberates the programmer from having to implement *tree-search* programs. A generate and test program is written in Turbo-Prolog, and compared to a more sophisticated test and generate implementation that uses methods of constraint satisfaction programming. Continuous capital budgeting problems are solved in CLP($\Re$), an experimental extension of Prolog that enables the solution of simultaneous algebraic constraints, as required to solve linear programs.

Acknowledgements

Special thanks to my family. Wambua, thanks for encouraging me to rush to school early in the morning. I hope you remember to do the same when you get to University.

I wish to thank my thesis supervisor, Professor Jean-Michel Thizy at the University of Ottawa for all his time and keen interest. His counsel was greatly appreciated. I am also grateful to Professors Dan Lane at the University of Ottawa and Surendra Rawat at Bell Canada / University of Ottawa for their encouragement and advice.

I am thankful to CIDA together with the Kenya government for providing me the financial opportunity to pursue graduate studies at the University of Ottawa. This thesis was partially funded by NSERC grant #OGP 0042197 and Bell Canada University Liaison grant "Expert Systems for resource allocation".

CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

The scope and importance of capital budgeting to top management extends beyond financial analysis. Capital budgeting also involves qualitative considerations needed to ensure that each project approved is consistent with overall corporate strategies. Making capital budgeting decisions requires strategic judgement which includes the balancing of many strategic performance criteria. For example, a proposal to invest in *fibre optic equipment* may have a high strategic importance by positioning a multimedia telecommunication firm at a competitive advantage, allowing it to offer better service. However, such a system may result in lower revenues in the short run.

In large, diversified corporations, project proposals typically originate in individual functional units, such as engineering, research and development, or marketing. Then they move up the organization's executive hierarchy for approval. Authorization of all projects that meet a minimum return on investment is common in companies with strong cash flows. However, it is important for companies with limited capital to approve only those projects that are necessary to further strategic company plans or that have the best economic impact.

Once the project proposals have been submitted, the next task is to use strategic judgement to identify projects that should undergo further technical, financial and strategic evaluation. Capital budgeting decisions based on strategic judgement are typically made by a few selected experts together with the decision-makers in a negotiation process. In related areas of financial planning, similarly expensive and scarce expertise has been captured by expert systems.

Occasionally, project proposals need to be classified into different categories to aid their identification. Each company has its own method of classifying project proposals. One common method is to distinguish between economic projects and noneconomic projects.

- *Noneconomic projects* are typically compulsory projects with little or no direct financial return. They may be required for continuity of operations, or for compliance objectives, such as regulatory requirements. They may not be subjected to strict financial analysis techniques, i.e these projects are not evaluated primarily by their direct economic impact.
- *Economic projects* are those projects whose costs and benefits are primarily economic, and usually involve revenue-enhancing or cost-reducing investments. These projects can usually be compared to one another by using financial analysis techniques.

After the initial identification, the projects that are acceptable are subjected to financial analysis and further strategic evaluation. Individual projects are evaluated in terms of established performance criteria, such as *efficiency*. Efficiency, which may involve both cost efficiency and capital efficiency, is generally measured by return on assets and return on investment. The most attractive projects may be selected at this stage, then the rest of the projects that meet the minimum established criteria are submitted to compete for the remaining resources. The unacceptable proposals are discarded.

In such complex structures, expert systems are useful to manage certain aspects of capital budgeting. They can improve the current practice of allocating capital by ranking the proposals, a simplistic but easily justifiable criterion. In the last two decades, mathematical programming has been a well-known method of solving the budget allocation problem; yet, corporations find it fairly inflexible, as its results tend to antagonise users, either because users unfamiliar with optimization prefer to see gradual improvements that they initiate themselves, or because mathematical programming packages have not been equipped with adequate, user-friendly interfaces. Thirdly and more importantly, many decision makers find that organizational, institutional and tactical constraints are poorly represented by mathematical programs.

Expert systems are expected to better reflect the organizational constraints and also to

support the control of the use of funds; yet, the algebraic aspects of capital rationing are essential. The most popular expert systems shells have limited mathematical capabilities; this calls for expert systems to be more tightly integrated with mathematical programming.

1.1 Focus of the thesis

The focus of this thesis is the applicability of decision-making methods implemented in expert systems to enhance traditional approaches to capital budgeting. First, a definition of capital budgeting reveals that mathematical programming addresses only one aspect of capital budgeting. Even in the narrow context of competitive capital allocation, a critical review of traditional mathematical programming approaches points out phases of project selection that need not be treated by optimization techniques, but rather by interactive, user-driven exploration, where expert systems have had success. A review of current applications of expert systems does not reveal any system tailored to capital allocation, but shows similarities to financial planning applications, a very active field of expert system design. Like capital budgeting, portfolio selection, security trading and risk analysis entail a combination of mundane checks and sophisticated analytic techniques. This analogy leads to refining the objective of this thesis; the actual design of rules for the capital budgeting problem faced by the Network Modernization Group of Bell Canada is now distinguished from the design of an integrated system performing resource allocation. The aim narrows to selecting an appropriate interface between project prescreening, and competition for scarce resources. A first attempt is to implement simple budget allocation, using VP-Expert, the commercial expert system shell originally selected for the target application at the Network Modernization Group. This approach is rejected as impractical, and so is the design of a linear program, using VP-Expert programming facilities, or a recourse to the linear programming utility offered by Exsys, another popular expert system shell. Resorting to logic programming languages, the goal of the thesis is to combine inference techniques with mathematical programming, in order to solve capital budgeting problems. The language Prolog is shown to have properties that are useful for dealing with optimization problems. This study is related to an area of artificial intelligence called constraint satisfaction programming.

Expert systems and traditional logic programming systems focus on entities (such as projects); constraint satisfaction programs operate on the constraints regulating the entities (such as budget limits). In spite of this different emphasis, logic programming languages themselves are appealing for stating constraint satisfaction problems because of their relational form and nondeterminism. Their relational form makes them appropriate for stating constraints. Programming in logic consists of defining relations that should hold between the arguments of logical propositions, and constraints fit perfectly inside this programming style. Nondeterministic algorithms in logic programming contribute to simplifying the design of backtracking programs, as illustrated in Chapter 3. Since backtracking is an important technique for solving constraint satisfaction problems, logic programming has more to offer in this context than functional programming, because it liberates the programmer from having to implement *tree-search* programs.

However, the logic programming literature does not offer many alternatives for solving linear constraint problems. It usually presents *generate and test* procedures equipped with some additional control strategies to improve their efficiency. Generate and test programming style is simple to implement in many cases, and results in a better legibility, extensibility, and modifiability of the programs. It relies on a generator that develops complete candidate solutions and an evaluator that tests each proposed solution by comparing it with the required state. This process of generation, followed by evaluation, continues until a solution is (or solutions are) discovered. This is a style frequently adopted by expert system writers. The generality (and weakness) of the generate and test scheme lies precisely in the fact that the generation process and the test process are completely independent; this scheme is abundantly illustrated in this thesis.

First, a standard generate and test scheme is implemented in Prolog to solve the discrete optimization problem arising from capital budgeting. Simple experimental results confirm the inefficiency of this kind of scheme. The basic reason for the inefficiency of the standard generate and test procedure arises from the fact that constraints are used only to reduce the search space *a posteriori* after discovering a failure. Then, the generate and test scheme is enhanced to obtain

a more efficient *look-ahead* scheme that utilizes the concept of *a priori* pruning, that is, using the constraints to reduce the search space before the discovery of failure. This new procedure attempts to prevent failures and enable both an early detection of failure and a reduction of the backtracking and the constraint checks.

Logic programming implemented in standard Prolog can only handle linear constraints sequentially. On the other hand, CLP($\Re$), a specialized logic programming language, is used to satisfy simultaneous constraints arising in capital budgeting. CLP($\Re$) is an experimental implementation of the constraint logic programming (CLP) paradigm in the domain of real arithmetic, and uses a modified simplex method to solve sets of linear constraints. The ability of CLP($\Re$) (or Prolog) to construct search trees is well suited for classical enumeration of a discrete optimum (branch-and-bound).

As will be seen in Chapter 2, Prolog has been a language of choice to implement financial, and CLP($\Re$), utilized for option trading, takes roots in mathematical programming. Thus, programming languages such as Prolog and especially CLP($\Re$) keep the focus of the thesis directly on expert systems for resource allocation.

1.2 Plan of the thesis

The remainder of the thesis comprises the following chapters:

Chapter 2 presents a review of the capital budgeting problem, and of the classical ranking and mathematical programming techniques for solving this problem. A survey of expert systems and their application to financial decision-making is presented. Limitations of each approach, and potential cross-fertilization are discussed.

Chapter 3 introduces the methodology of inference-making applied to expert systems. Elements of Turbo-Prolog are described in view of implementing a standard generate and test procedure to solve the integer linear program arising in capital budgeting. Constraint satisfaction programming is presented as the method underlying a more efficient generate and test program. An experimental implementation of the constraint logic programming paradigm in the domain of real arithmetic, CLP($\Re$), is also reviewed. Finally, a program is designed in CLP($\Re$) to solve linear programs arising in the continuous version of the capital budgeting problem.

Chapter 4 presents computation results validating the programs written in Prolog and CLP($\Re$), using datasets resembling a base case presented by the Network Optimization Group of Bell Canada.

Chapter 5 presents the conclusions of the thesis, and also discusses possible enhancements to the techniques implemented in the thesis.

Chapter 2

Mathematical Programming & Expert System Approaches to Capital Budgeting

Capital budgeting addresses the selection of capital investment projects in four stages:

(1) identifying projects closely linked to the organization's strategic objectives,
(2) developing project information and cash flow estimates needed for project analysis and selection,
(3) selecting the portfolio of projects to be funded through the use of one or more economic and possibly noneconomic criteria, and
(4) evaluating the performance of approved projects (Pinches, 1982).

This study focuses primarily on the selection phase (3), which decision theorists and management scientists call restrictively Capital Budgeting.

In its simplest form, capital budgeting can be defined as follows (Lorie and Savage, 1955; Weingartner, 1966): given the net present value (NPV)[1] of a set of independent investment alternatives, and given the required outlays for the projects in each of the time periods of the

[1] S_t is net cash flow in period t, r is the discount rate, and K is the initial outlay.

$$NPV = \sum_{t=1}^{n} S_t(1+r)^{-t} - K$$

planning horizon, find the subset of projects that yields a maximum net present value, while simultaneously satisfying a constraint on the capital outlays in each of the periods. For example, the investment decisions may entail the selection of capital equipment, or the choice of several modernization programs, or research and development projects. Capital budgeting decisions may allow partial funding of projects, but in many cases a project cannot be divided.

Capital rationing, arising when there are insufficient funds to finance all attractive projects, is generally the centrepiece of capital budgeting. Typically the funds are limited by budget, capital, organizational or strategic constraints. Other constraints may relate to the consumption of any scarce resource required by all or some of the projects - for instance, labour, materials, machinery, or space. In addition, rationing may need to satisfy requirements such as reporting profit, share price and other financial statistics.

There have been two main approaches to solve the Capital Budgeting Problem: a heuristic approach (ranking), and the more theoretically founded mathematical programming approach.

2.1 RANKING APPROACHES:

Dean (1951) popularized a technique for rationing capital among competing investment proposals based on a ranking of candidate investments using their respective indices of net present value (NPV) relative to their capital requirements. Since that time, ranking has been a widely used method for selection in both the private and public sectors (Farragher, 1986; Guranani, 1984). Other ranking indices are also used, such as a rate-of-return index, a payback index, and many other methods that reduce the costs and benefits of a project to a pure number.

Ranking is an exact method in the simple case of a single constraint restricting availability of capital for divisible projects, i.e. those which can be scaled down with benefits accruing in direct proportion to the amount invested. In more realistic cases, e.g. when several constraint limit resources, or when the projects are indivisible, Lorie and Savage (1955) pointed out some

weaknesses of heuristic ranking by net present value. The net present value (NPV) of the portfolio of projects thus selected may be suboptimal, as the following example shows.

Example 2.1:

Consider the two indivisible projects described in the following table.

Project	NPV	Budget Outlay	Profitability Index (PI)
1	50	50	1.00
2	75	110	0.68
Total budget available		110	

Strict application of the ranking method would call for the selection of Project 1 only, which has a higher profitability index (PI is computed as the NPV/Outlay ratio) with a resulting net present value (NPV) of 50. The unallocated budget funds amount to 60 units. On the other hand, an overall net present value of 75 could be achieved by choosing Project 2 only. Note that if the projects were divisible, the ranking heuristic would indeed generate the largest total net present value of 91 by undertaking all of Project 1 and 54 % of Project 2.

In spite of this shortcoming, it has been suggested (Fogler, 1972) that heuristic techniques such as the net present-value index are often desirable, because they provide reasonably close approximations to solutions attainable by complex models, and yet are very simple to apply. Simulation studies done by Forsyth and Owen (1981) support the above suggestion.

Before ranking can be applied, the firm must pre-screen proposals to obtain a set of independent projects for consideration; it is from the independent projects plus any other projects that may have been sidelined at the pre-screening stage that a subset is chosen to be included in the capital expenditure program. In Section 2.3.5, it will be shown how an expert system can perform a ranking heuristic incorporating some limited logical considerations such as dependencies and synergies.

2.2 Mathematical Programming Models

Unlike the ranking approach, mathematical programming seeks an exact solution by formulating a model in which a single-valued objective function containing the desired criteria is optimized, subject to a series of constraints reflecting various asset limitations.

The first mathematical programming formulation of the Capital Budgeting Problem was proposed in Weingartner (1963):

$$\textit{Maximize} \quad \sum_{j=1}^{n} c_j X_j$$

$$\textit{Subject to} \quad \sum_{j=1}^{n} a_{tj} X_j \leq b_t \qquad t=1,\ldots,T\ ;$$

$$0 \leq X_j \leq 1 \qquad j=1,\ldots,n\ ;$$

the first line describes the objective function, where c_j denotes the net present value of Project j, and the inequalities represent the constraints, where a_{tj} is the amount of resource, such as cash or manpower, required for Project j in period t. The maximum permissible expenditure in period t is given by b_t. The fraction of Project j accepted is given by X_j.

To solve the problems described by Lorie and Savage, selecting indivisible projects, 0-1 integer programming is necessary, where the decision variables are taken to be $X_j = 0$ or 1, indicating whether the j^{th} project is rejected or accepted. Both the linear and integer programs have been studied by Weingartner (1963, 1966). The simplex algorithm is a well recognized method to solve the linear program (Dantzig, 1963), but integer programming problems are substantially harder to solve. There is no equivalent to the simplex algorithm when the variables are required to be integers, and exhaustive enumeration of all solutions is typically impractical. Hence, techniques based on an intelligent enumeration have been devised and are referred to as branch-and-bound (Little, 1963). Balas (1965) developed a different algorithm based in the implicit enumeration of the variables, well suited to Weingartner's 0-1 program. Today, many

commercial mathematical programming systems can solve these problems in reasonable time.

2.2.1 Alternative constraints

The linear formulation has been adapted and extended in various ways. One important special scenario for the capital budgeting problem involves cash flow constraints. In this case, the constraints

$$\sum_{j=1}^{n} a_{ij} X_j \leq b_i$$

reflect the incremental cash balances in each period. The coefficients a_{ij} represent the net cash flow from investment j in period t. If the investment requires additional cash in period t, then $a_{ij} > 0$, while if the investment generates cash in period t, then $a_{ij} < 0$. The right hand-side coefficients b_t represent the incremental exogenous cash flows. If exogenous funds are made available in period t, then $b_t > 0$, while if funds are withdrawn in period t, then $b_t < 0$. These constraints state that funds required for investment must be less than or equal to funds generated from prior investments plus exogenous funds made available (or minus exogenous funds withdrawn).

To reflect the need for other financial, human, physical resources, other constraints have been added, for instance, capacity, liquidity, and manpower. For example, the Network Modernization Group of Bell Canada needs to represent a depreciation constraint:

$$\sum_{j=1}^{n} d_j X_j \leq d_i$$

based on the financial planning of the Corporation, it imposes a limitation on the total allowable depreciation for a given financial year. Other constraints reflect specific requirements. For example, operational feasibility, market needs, strategic requirements may impose bounds of their own, which were partially represented in the Network Modernization Group problem as bounds on the variables:

$$U_j \geq X_j \geq L_j,$$

that prescribe minimum and maximum percentages for each project proposed.

Below is a single period capital budgeting example that illustrates the mathematical programming approach to deciding which capital investments should be undertaken with limited budgetary and manpower resources.

Example 2.2a:

Suppose four proposals are identified as potential capital investments for the upcoming year. Each proposal is further evaluated with respect to capital costs and net annual cash flow during its associated project life. After computing the total capital that would be required to fund all four projects, it is realized that the total amount exceeds the allotted capital available by $68,000. It is also determined that the required man-hours to implement the four projects exceeds those available by 1,290 man-hours. A summary of the individual proposals along with the associated costs and cash flows, including the respective present value and required man-hours is displayed in Table 2.2.4a. S_{jt} represent the net annual cash flow for Project j in year t. K_j is the initial outlay for Project j, and r is the annual discounting rate.

Table 2.2.4a: financial data for Example 2.2a

r = 12%			X_1	X_2	X_3	X_4	Available	Shortfall
t	$(1+r)^{-t}$	$-K_j$	-50	-180	-98	-90	$350	$68
1	.89	S_{jt}	15	35	27	50		
2	.80		17	45	30	40		
3	.71		19	55	33	30		
4	.64		19	65	36	20		
5	.57		19	75	39	10		
NPV			13.3	10.1	18.5	26.3		
Man-hours			1050	890	1450	1400	3,500	1,290

The underlying objective of the problem is to select a set of indivisible capital investment proposals which will yield the highest net present value (NPV). This problem is expressed in the mathematical programming format shown in Table 2.2.4b. The NPV values have been rounded to the nearest integer.

Table 2.2.4b: binary linear program for Example 2.2a

Maximize Z= 13X1 + 10X2 + 19X3 + 26X4

Subject to:

50X1 + 180X2 + 98X3 + 90X4 ≤ 350

1050X1 + 890X2 + 1450X3 + 1400X4 ≤ 3500

X1, X2, X3, X4 = 0 OR 1

Table 2.2.4c: optimal solution of Example 2.2a

The optimal solution selects the following projects:

j	1	2	3	4	Totals
X_j	1	1	0	1	-
Outlay	50	180	-	90	320
Man-hours	1050	890	-	1400	3340
NPV	13.3	10.1	-	26.3	49.7

2.2.2 Multiobjective Models

Experience has shown that NPV maximization is rarely the ultimate corporate criterion for portfolio selection. Among many of its limitations is the choice of the discount rate to use in formulating the objective function. Different objective functions have been proposed, some including uncertainty factors: in addition to NPV, internal rate of return[2] (IRR), payback and dividend criteria, Capital Budgeting criteria may include project risk, borrowing limits, lending guidelines, diversification goals, growth maximization, behavioral considerations, manager's utility function and many more. Moreover, the diversity of pressures (political and economic) exerted on any company leads to complex, changing, and subjectively evaluated objectives. Yet, the specification of an appropriate objective function by capital allocation modellers may critically affect the mix of projects selected.

Many authors (Baumol and Quandt 1965, Naslund 1966) have suggested the goal

[2] IRR is the discount rate that equates economic benefits to costs i.e,

$$\sum_{t=1}^{n} S_t(1+IRR)^{-t}-K = 0$$

programming concept to solve the multicriteria problem. Spronk and Nijcamp (1978), explore the use of interactive goal programming models. Such methods of multicriteria decision making as the *Analytic Hierarchy Process* (Saaty 1980) may prove useful in evaluating qualitative criteria, since they reduce multicriteria problems to a sequence of pairwise comparisons of outcomes and of criteria. The name of the original program implementing the method: *Expert Choice*, the application of expert systems to multicriteria decision making, the presence of dependencies in the projects (as explained in the next section), are early signs that capital allocators are turning to expert systems for support. In 1991 and 1992, the University Liaison Project of Bell Canada supported another master's thesis investigating the application of multicriteria decision-making (MCDM) to resource allocation (Pissarides, 1992). Further analysis of MCDM is purposefully omitted to avoid duplication.

2.2.3 Modelling Project Interdependencies

Mathematical programming can represent a variety of project interdependencies. Suppose for example, that Project 1, (investment in a new line) is contingent upon Project 2 (investment in a new plant). In the case of indivisible projects, this *contingency* can be modeled simply by a constraint

$$X_2 \geq X_1 \qquad (\text{or } X_1 - X_2 \leq 0)$$

which states that if $X_1 = 1$ and Project 1 is accepted, then necessarily $X_2 = 1$, and Project 2 must be accepted. On the other hand, Project 2 may be undertaken independently. Another constraint, concerning incompatible indivisible projects:

$$X_1 + X_2 + X_3 + X_4 \leq 1,$$

states that only one of the four investments can be accepted. Such constraints are commonly called *multiple-choice constraints*. A slightly less common contingency takes the form:

$$\sum_{j \in J} I_j X_j \geq I.$$

It implies that there is a minimum required investment for a set J of interrelated projects. The I_j's represent the investment in Project j. By combining these logical constraints, the model can incorporate many complex interactions between projects.

Example 2.2b:

Revisiting Example 2.2a, we suppose that an additional constraint, $X_3 \geq X_4$ is now included; the program now yields a different optimal solution:

j	1	2	3	4	Totals
X_j	0	0	1	1	-
Outlay	-	-	98	90	188
Man-hours	-	-	1450	1400	2850
NPV	-	-	18.5	26.3	44.8

2.2.4 Limitations of mathematical programming:

In practice, resource allocation decisions are based on many judgmental factors. Some express pragmatic rules or measurements derived from experience, in many cases acquired from previous allocation exercises; for example, decision makers agree that if Project 1 is funded, then the NPV ratio of Project 2 must be increased. Other factors are situational, e.g.: "this year, mandatory projects will be increasingly funded at the expense of those projects which require Research and Development expenditures greater than a tenth of their projected benefits".

Integer programs can model each of these requirements, as well as many logical statements involving the variables of the Capital Budgeting problem. Yet, traditional

mathematical programming approaches have rarely been able to capture such circumstantial requirements. This lack of flexibility may be explained by two reasons:

- mathematical modelling: mathematical programs use equations and inequations involving binary variables to model logical statements (which are called *propositions*). Each variable (e.g. x) holds the truth valuation of a particular proposition (e.g. C), i.e. 1 if the proposition is satisfied, 0 otherwise. Clearly, 1-x represents the negation of C. Logical relations between *two* propositions C_1 and C_2 can also be modeled by linear inequalities. For example, if x_1 represents the valuation of C_1 and x_2 the valuation of C_2, $x_1 + x_2 \geq 1$ expresses the requirement that C_1 or C_2 must be true. However, such modelling remains a mathematical programming specialty, and is prone to errors, and can considerably increase the size of mathematical programs; furthermore, it is difficult to verify the model. For example, expressing project incompatibility is simple, if the projects required minimum funding of 0:

$$Q_1/U_1 - x_1 \leq 0$$
$$Q_2/U_2 - x_2 \leq 0$$
$$x_1 + x_2 \leq 1$$
$$x_1, x_2 = 0 \text{ or } 1,$$

where the variable Q_i designates the level of funding of Project $_i$, the variable x_i indicates whether Project i is funded or not, U_i designates the maximum level of funding of Project i. But the inequations are useless if no upper level of funding is known for a project i and, if an arbitrarily large value is given to U_i, they provide inefficient guidance toward searching for a solution. Given that professional modellers make frequent errors, casual users are very unlikely to find appropriate mathematical expressions to model their requirements. User interfaces can be written to translate user specifications into linear inequations. However, mathematical programmers often don't have the skills to write such interfaces, which they naturally neglect, or design poorly. In fact, the better interfaces have often been predecessors or prototypes of experts systems.

computer programming: mathematical programming solvers are traditionally written, using imperative, compilable languages (e.g. Assembler, Cobol, Fortran, Pascal, C). Numerical or (less often) symbolic input is accepted at run time, but alterations to the execution of the program, e.g. conditional statements, calls to procedures, typically require programs to be recompiled. Therefore, interactive mathematical programs must anticipate the full range and combination of user requirements as parameters, and the programming style becomes cumbersome. For example, suppose a user specifies:

if Project 1 is funded and salary increase is fixed at 7% over two years, apply constraints involving personnel requirements.

This user specification can be satisfied only if it has been anticipated by the program designer, for example by indexing the procedures presiding over the generation of personnel constraints, such as in the following program fragment (where .07 is represented explicitly for brevity):

```
case Project of

    1:    if Salary_Increase > .07 then
          begin
             Personnel1;
             Personnel2;
             Personnel3;
          else
          ...
          end;
    else
    ...
    end;
```

Such conditional sections, anticipating all possible user specifications, become unwieldy. For example, there may be many other conditions triggering the Personnel procedures. Also, a completely different set of tests will be needed to process a variation of the requirement:

If Project 1 is funded or salary increase is fixed at 7% over two years, apply constraints involving personnel requirements.

A dedicated utility is necessary to store, edit, index, access efficiently this set of conditions. Expert systems provide such facilities, plus tailored user interfaces, explanatory capabilities, as explained in the following sections.

2.3 Expert Systems

Feigenbaum (1982, p.1) states:

> "An expert system [knowledge-based system] is an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution".

Compared with conventional software systems, which are heavily procedurally oriented and code intensive, expert software systems are knowledge oriented and use relatively little code. Expert systems have been applied to many *domains*, including financial planning (see Appendix E).

An expert system is comprised of:

(1) a knowledge base:

(2) an inference procedure:

(3) a working memory.

The working memory is sometimes referred to as a *global data base*, and is the one that keeps track of the problem status, the input data for the particular problem status, and the relevant history of what has been done thus far.

The power of an expert system is found in the knowledge base and not in the procedural code. A human *domain expert* usually collaborates to help develop the knowledge base. Once the system has been developed, in addition to solving problems, it can also be used to instruct others in developing their own expertise. Thus, it has been observed that, ideally, there are three different user-modes for an expert system (in contrast to the single mode - "getting answers to problems" characteristic of popular type of computing):

(1) getting answers to problems -- user as client;

(2) improving the system's knowledge - user as tutor;

(3) harvesting the knowledge base for human use - user as pupil.

Users of an expert system in mode (2) are known as *domain specialists*. It is not possible to build an expert system without one. An expert system acts as a systemizing repository over time of the knowledge accumulated by many specialists of diverse experience. Hence, it can and does

ultimately attain a level of consultant expertise exceeding that of any single one of its *tutors*.

An explanation module is typically included, allowing the user to challenge and examine the reasoning process underlying the system's answers. Figure 2.3 is a diagram of an idealized expert system. When the domain knowledge is stored as production rules, the knowledge base is often referred to as *the rule* base and the inference engine as the *rule interpreter*.

An expert system comprises a knowledge base in which an inference engine searches for answers to a particular problem. The user input or query is through the input data block and is matched against relationships in the knowledge base under the control of the search strategy. The reasoning process by which the result is reached may be traced by the explanation capability. It does this, essentially, by tracing the chain of rules that led to the solution and translating it into some form readily intelligible to the user.

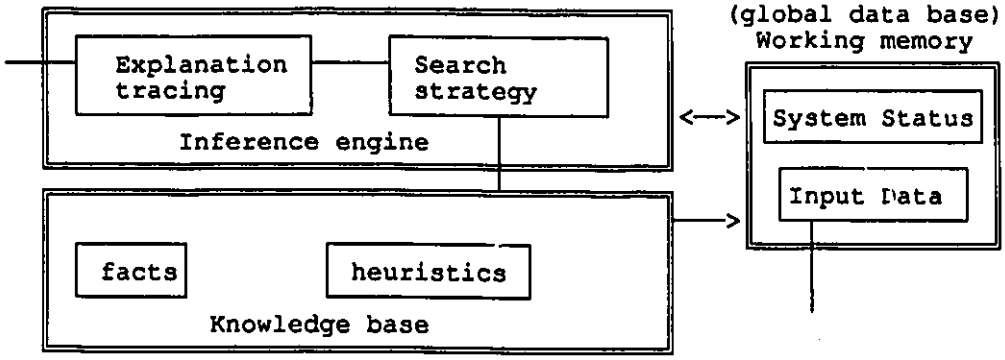


Fig. 2.3 Basic structure of an expert system

An expert system differs from more conventional computer programs in several important respects. Duda (1981, p.242) observes that, in an expert system, "....there is clear separation of general knowledge about the problem (the rules forming a knowledge base) from information about the current problem (the input data) and methods for applying the general knowledge to the problem (the rule interpreter)." In a conventional computer program, knowledge pertinent to the problem and methods for utilizing this knowledge are all intermixed, so that it is difficult to change the program. In an expert system, "....the program itself is only an interpreter (or general

reasoning mechanism) and [ideally] the system can be changed by simply adding or subtracting rules in the knowledge base."

2.3.1 The Knowledge Base

The Knowledge Base of an expert system consists of facts and heuristics. The *facts* constitute a body of information that is widely shared, publicly available, and generally agreed upon by experts in the fields. The *heuristics* are mostly undocumented rules of good judgement (rules of plausible reasoning, rules of good guessing) that characterize expert-level decision making in the field. The performance level of an expert system hinges on the quantity and quality of information that the knowledge base possesses.

A very common approach to representing the domain knowledge needed for an expert system is by *production rules*, which are also referred to as *if-then* rules. In this approach, a knowledge base is made up of rules which are invoked by matching their patterns with some characteristics of the task being processed by the global data base. For example, if a current specification is that data transmission requires digital network services, rules concerning digital network services will be consulted in the knowledge base.

If-then rules in knowledge-based systems are managed differently from similar ones in conventional programming systems. They can be modified much more easily to meet changing needs. Rule-based systems are deductively not as powerful as logical theorem-proving programs because their only rule of inference is *modus ponens* and their syntax only allows a subset of logically well-formed expressions to be clauses in conditional sentences. Their primary distinction from logic-based systems is that rules define facts in the context of how they will be used, while expressions in logic-based systems are intended to define facts independently of their use.

The rules in a knowledge base represent the domain facts and heuristics - rules of good judgement for taking actions when specific situations arise. It is therefore apparent that, for the expert system to be useful, it must contain knowledge specific to the problem domain.

An expert usually has many judgmental or empirical rules, for which there is incomplete support from the available evidence. In such cases, one approach is to attach a level of confidence (certainty factor) to each rule to indicate the degree of certainty associated with each rule. As rules apply, their certainty values are combined with each other and the certainty of the problem data, to calculate the confidence that can be placed in the final solution.

The general syntax of an if-then rule is:

RULE <rule_label>
IF <condition>
THEN <conclusion>,

although there may be slight variations from one expert system shell to another. The <condition> and <conclusion> components may comprise complex logic expressions. An example of *if-then* rules can be seen in the sample program below, Figure 2.3.1b. The premises or conditions in the *if* clauses and conclusions in the *then* clauses are clearly identifiable. The program is written in the *VP-Expert* shell language. In VP-Expert, the conclusion component may include an optional ELSE statement (VP-Expert, 1989). The first *ASK* statement in Figure 2.3.1b, which is stored in the system's knowledge base, causes the system to display the following message:

```
How would you characterize the potential for
technological obsolescence in Proj22 project
[very low, low, average, high, very high]?

>> (A user types in an answer here.)
```

Figure 2.3.1aa Example of knowledge-base user query.

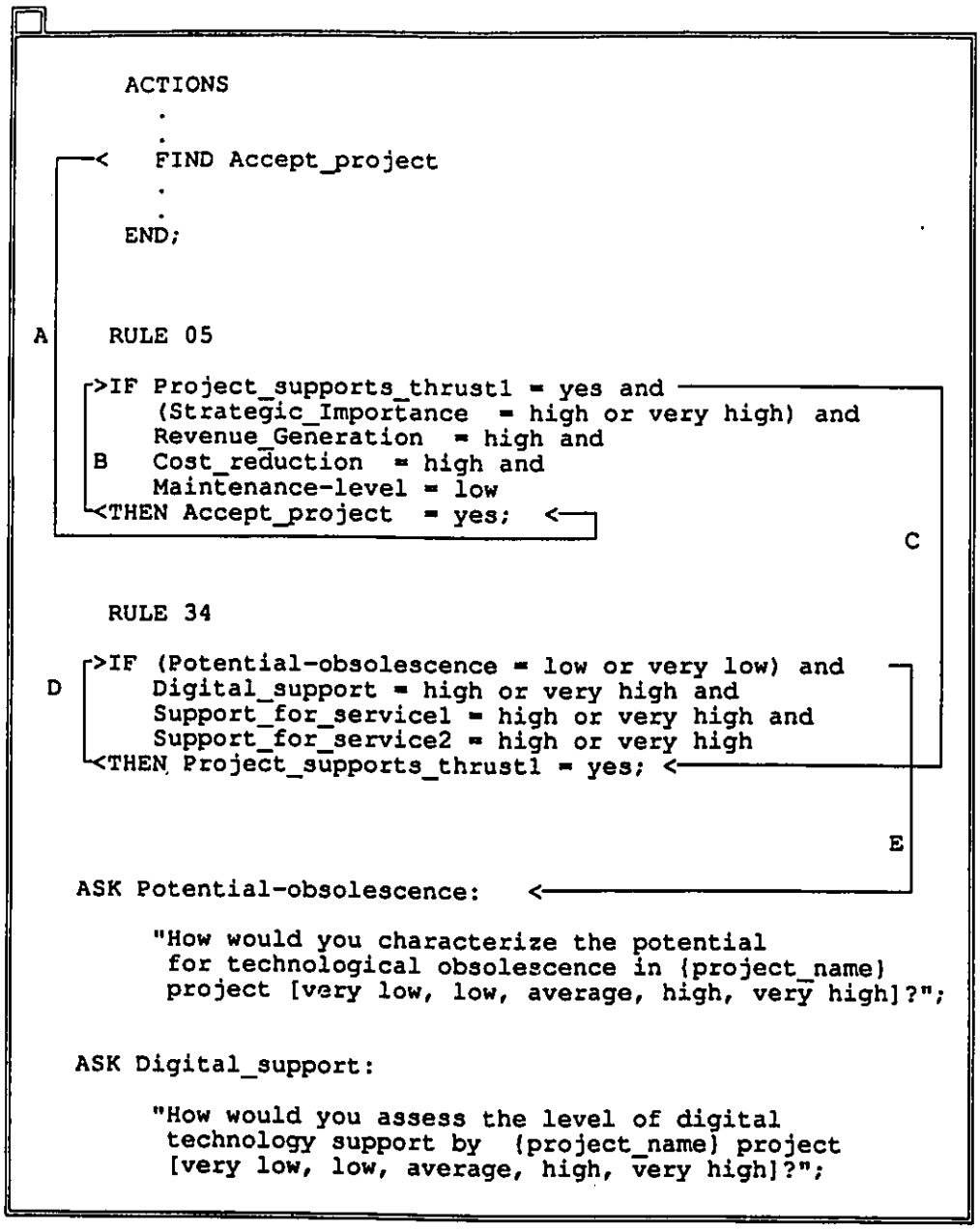


Fig. 2.3.1b **Example** of *IF-THEN* rules and *ASK* statements from a project identification knowledge-base and of the inference engine search process written in *VP-Expert*.

2.2.3 The Inference Engine

The inference engine organizes and controls the steps taken to solve the problem. To that effect, *if-then* rules are chained to form a line of reasoning, a method reviewed in more detail in the following subsections on backward and forward chaining. This type of inference mechanism has been referred to as *inference rules.*

Forward chaining uses facts such as A, and rules such as A→B, to assert further facts, such as B, and possibly reiterate, using further rules, such as B→C, to finally reach the desired conclusion, e.g. C. Backward chaining begins with the conclusion C, and uses rules such as B→C, then A→B, until a fact such as A is encountered.

The simplest arrangement of the production rules is to list them in no particular order. With this arrangement, new rules may be added on, making it easy to enlarge the collection of rules as more is learned about the problem. With a small number of rules, given the high speed of computers, it is practical to search a random list. However, if the number of rules is large, they may be partitioned into sub-lists, or contexts, on some logical basis. The search strategy then uses a higher-level, or a metarule, based on the logic of the partition, to determine which sub-list to run through first, thus reducing the length of the search.

Backward Chaining

Backward Chaining involves starting with one or more possible goals. In a rule-based system for example, the inference engine tests each goal to see whether or not the *if* clauses in the rule containing each possible goal are all answered. If not, it tests each rule in turn until an answer is found (that is, it finds a rule in which all the *if* clauses are true), or until all possible rules are examined and no answer can be found.

The arrows in Fig. 2.3.1b shows how this process works. The program begins its search for a solution with the goal it is seeking, *FIND Accept_project.* It seeks the first rule in the

knowledge base that contains in the conclusion the key word *Accept_project*, as is indicated by the arrow line (A) on the left of Fig. 2.3.1b. The inferenc- finds the key word *Project_supports_thrust1*, as shown by the line (B) in the figure. It then seeks the first rule in the knowledge base that contains in the conclusion the key word *Project_supports_thrust1*, as is indicated by the arrow line (C). The inference finds the key word *Potential-obsolescence*, as shown by the line (D) in the figure. It then invokes the *ASK Potential-obsolescence* statement as indicated by line (E). The answer to this goal is assumed found when:

(1) an answer is found for all of the *if* clauses in a rule containing "Accept_project" in its *then* clause, or

(2) all the rules have been examined and the system determines an answer is not possible.

A possible problem with backward chaining is the handling of conjunctive (joined by logical 'AND') subgoals without appropriate heuristic for guidance. In general, to attack a conjunction, one must find a case where all subgoals are satisfied, a search for which can result in a combinatorial explosion of possibilities. Thus, appropriate domain heuristics must be found to achieve an efficient and effective inference in expert system.

Forward Chaining

Forward Chaining involves working in the other direction. It starts with the data, examines the *if* clauses, and searches for a solution by working from the data toward a goal. Using this reasoning process, the system builds proceeds from the data toward obtaining a solution from the data. When an answer is found to all of the *if* clauses in a rule containing the FIND (goal) word in its *then* clause (for example, "Accept_project"), the system gives the user its recommended solution. It will report that no solution is possible, if after an exhaustive search no answer is found.

One problem with forward chaining, without appropriate heuristics for pruning, is that one could examine a very large number of rules. For this reason, controls are needed to set limits on the search process. To reduce the number of goals examined, hence the time and computer

capacity required for processing, forward chaining and backward chaining have been combined in expert systems such as Molgen in molecular genetics (Stanford University) and Hearsay II in speech understanding (University of Southern California).

2.3.3 Financial applications of expert systems

A search of applications of expert systems to resource allocation was narrowed mostly to a bibliographic survey of financial applications, for which many experts systems have been designed, performing tasks such as claim estimation, credit analysis, tax advising, financial statement analysis, financial planning (Cuena, 1987), loan application, performance evaluation of dealerships, conflict-of-interest consultation, inventory management. A list of current expert systems relevant to capital budgeting is given in Appendix E, comprising in large part systems for credit evaluation and portfolio management. Only a few systems seem to be oriented toward project financial analysis or long-term financial planning (Senicourt, 1987). Among those, CASH VALUE, by Heuros Ltd., cites capital budgeting as an application.

A number of financial expert systems could be adapted to the preliminary phase of capital allocation as follows. An investigation of the practice of project selection at the Network Modernization Group of Bell Canada reveals that an ABC classification may be effective. In Class A, some projects will certainly be funded at a level close to the requested amounts, or proportionately to the overall budget fluctuations. In Class C, some projects will certainly be delayed under current economic and technological conditions. Projects in Class B will compete for allocation after Class A is funded, and form part of the mathematical program presented earlier.

The literature describes financial expert systems from a user requirement perspective, revealing little detail of the knowledge base or decision rules. Sterling (1986, pp. 351-360) presents a fictionalized example of a credit evaluation system, classifying expert rules as client oriented (client and collateral evaluation), and bank oriented (financial performance). An expert system for capital budgeting could comprise a similar grouping: a section for individual project

assessment and another for resource rationing (e.g. mathematical programming), as explained next. In capital budgeting, project evaluation is similar in one respect to customer credit evaluation, project feasibility study, or choice of investments to be considered in a portfolio. In credit evaluation, initial validation depends on a group of rules describing customer specific criteria (e.g. indebtedness, liquidity, revenue, etc.) Another group of rules evaluates whether the overall financial and economic condition of the decision-maker has changed and may accordingly adapt the pre-selection criteria used in the first section. Similarly, if one part of an expert system forms a set of candidate securities for a portfolio, the actual inclusion of the securities in the portfolio may require a part using a different methodology, such as some form of capital asset pricing model. Mathematical programming would play that role for capital budgeting.

The financial importance of resource allocation, the flourishing of financial expert systems, and the scarcity of capital budgeting applications, leads one to conjecture that intrinsic factors work against the rapid adoption of expert systems for capital budgeting. The following list of difficulties is excerpted from (Turban, 1990, p.440). Similar cautions are found in many introductory books (Martin, 1988).

"Here are some factors and problems that slow down the commercial spread of ES:

1- Knowledge is not always readily available. Expertise is hard to extract from humans.

2- The approach of each expert to situation assessment may be different, yet correct.

3- It is hard, even for a highly skilled expert, to abstract good situational assessments when he or she is under time pressure.

4- Users of expert systems have natural cognitive limits. Humans see what they are prepared to see, often only what falls within a narrow attention span.

5- ES work well only in a narrow domain, in some cases in very narrow domains.

6- Most experts have no independent means of checking whether their conclusions are reasonable.

7- The vocabulary that experts use for expressing facts and relations is frequently limited and not understood by others.

8- Help is frequently required from knowledge engineers who are rare and expensive, a fact that could make ES construction rather costly."

Points 1, 2, 3, 5, 6, 8 are particularly relevant to the implementation within the Network Modernization group of Bell Canada. Specific limitations are summarized below.

Encodable rules

The decision process must be captured by an expert system as a set of well recognized, operational rules. After the decision rules are captured, expert systems undergo a training phase whereby experts are asked to assess the quality of the decision, and modify the knowledge base accordingly. Such facts and rules were not available at the Network Modernization Group of Bell Canada at project inception. External demands make expert advice succinct.

Level of decision and accountability

Current expert systems address narrow domain of expertise, applicable in clearly defined situations, where experts concur on the mode of decision and can verify results. In contrast, capital allocation is viewed at the Network Modernization Group of Bell Canada as a high-level, strategic exercise, deriving less from expert diagnosis than from multi-party negotiation.

The suitability of an expert system has also been described from a simpler point of view.

Frequency and mode of use

Introductory books on expert system propose the following acid test of adequacy: an expert system must be designed for repetitive decision-making in casual cases (from the decision maker's point of view). In this view, expert systems are least suited to highly creative decision-making in an uncertain, moving environment. In the Network Modernization Group, capital budgeting is a yearly exercise, barely classifiable as frequent use. More frequent use may occur in advance of yearly negotiations, when each functional unit reviews its bid for resources, in preparation of the group decision. At this stage, the decision criteria may be substantially different from those of the Network Modernization Group of Bell Canada and may need to be tailored to the unit.

In summary, a review of financial expert systems points out applications fairly closely related to capital budgeting. While expert systems are used for individual project evaluation,

portfolio selection and long-range financial planning, can be applied to resource allocation at the Network Modernization Group of Bell Canada, the design of specific selection rules for initial project evaluation remains elusive. Since, on the other hand, resource constraining is a well-recognized problem of the Network Modernization Group, and mathematical programming an acceptable allocation technique to the Group (actually one of the themes of the University Liaison Project), the focus is placed on the role of an expert system in resource constraining by linear integer programming.

2.3.4 Interface between expert systems and mathematical programming

Individual project pre-screening by expert systems fulfils a long recognized mathematical role of pre-processing large problems. In linear programming (Murray, 1988), set covering problem (Garfinkel, 1969), knapsack problems (Martello, 1987), site selection (Mateus, 1991), it is customary to eliminate redundant variables or constraints, performing simple calculations based on the input data. This traditional, data oriented pre-processing adopted by conventional optimization systems can be extended by expert systems, using contextual information. For example, some project contingencies can be revealed by the application of rules, instead of being known explicitly and directly encoded in the mathematical programs. As an illustration, Projects 1 and 2 may well be recognized as incompatible under a given scenario (e.g. high interest rate), but decision makers may hesitate or be unable to embed this rule as a constraint.

The interface between project review and rationing, i.e. between the expert and the mathematical programming systems, predetermines the degree of interactivity that decision-makers will be given between the two phases of project selection under consideration. Several configurations are possible:

- separate systems: a list of fundable projects with their respective coefficients a_{ij} of the mathematical formulation would be written on a file to be processed by a mathematical programming system.

- loosely coupled systems: some expert systems such as Exsys offer a loosely coupled linear programming utility (see also Turban, 1991). Linear programming constraints can be directly defined by rules. The linear program can be solved once all appropriate rules have been selected.

- resident linear programs: whereas linear programming has been viewed as a numerical task, little suited to the symbolic manipulation provided by expert system shells, the merit of numerical treatment of simultaneous constraints has arisen in many independent applications of expert systems, artificial intelligence, logic programming, robotics, computer integrated manufacturing, often captured under the rallying name of Constraint Satisfaction (or Logic) Programming (see Chapter 3).

This thesis explores the third alternative, trying to overcome the limitation that comes immediately to mind: can the efficiency of commercial linear programming packages be matched, using languages devised for designing an expert system shells? Against this difficulty stands the potential benefit of a tightly integrated linear program that could directly influence some pre-selection rules, in the same manner as a bank's financial position influences credit assessment. For example, dual prices could be used as a benchmark for assessing additional candidate projects. Some constraints themselves could be added to pre-selection criteria once some projects had been initially selected; most common should be contingency constraints, requiring a prior project to be performed in support of the candidate project assessed.

In view of a slightly looser interface, the expert system Exsys was reviewed for the University Liaison research contract of Bell Canada, but we did not find the shell attractive for development purposes. Exsys was the only commercial expert system version available that contained a module for linear programming. Upon reflection, our decision may well fit a usual pattern: early combinations of two separate programming technologies are often evaluated for adoption by paying attention to the weakest or least desirable features of each subsystem, in this case linear programming vs. expert systems; users demand separate offerings to perform in-house, tailored development.

As lack of information has hindered the development of a testable system for the Network Modernization group, little methodological benefit would accrue from designing a linear program, using features of commercial expert system such as VP-Expert that resemble (or underachieve) conventional programming languages. Rather, Dean's simple ranking heuristic is implemented in VP-Expert, after which the following chapter analyzes the methods of implementing linear programming in Prolog.

2.3.5 Implementing the ranking heuristic in VP-Expert

The object of the following implementation is to show how logical considerations such as dependencies can be added to the ranking heuristic within VP-Expert (Fig. 2.3.5), a popular expert system shell. Given 4 projects, the ranking heuristic *aims* to maximize the objective function:

$$\sum_{j=1}^{4} c_j X_j$$

where c_j denotes the net present value or any other objective function criterion (see Section 2.2.2), subject to a capital constraint:

$$\sum_{j=1}^{4} a_j X_j \leq b$$

and a logic constraint, $X_3 \geq X_4$, where a_j denotes the capital requirements and b is the total capital available. Each project j is either fully funded or not funded at all, for which X_j takes a value of 1 or 0.

```
ACTIONS
   FIND Total_budget
   Sum = 0
   FOR X = 1 TO 4
      FIND Cost[X]
   END
   FOR X = 1 TO 4
```

```
        FIND Project[X]
        FIND Decision
        RESET Decision
    END
    FOR X = 1 TO 4
       DISPLAY "Project-{X} = {Project[X]}"
    END;

RULE 1
IF   Project[4] = 1  AND
     Total_budget >= (Cost[3]+ Sum)
THEN Project[3] = 1
     Sum = (Sum+ Cost[3])
ELSE Project[3] = 0;

RULE 2
IF   Project[3] <> 1  AND
     Total_budget >= (Cost[4]+ Sum)
THEN Project[4] = 1
     Sum = (Sum+ Cost[4])
ELSE Project[4] = 0;

RULE 3
IF   Project[X] = UNKNOWN  AND
     Total_budget >= (Cost[X]+ Sum)
THEN Decision = do
     Project[X] = 1
     Sum = (Sum+ Cost[X]);

RULE 4
IF   Project[X] = UNKNOWN  AND
     Total_budget < (Cost[X]+ Sum)
THEN Decision = do
     Project[X] = 0;


ASK Total_budget : "What is the budget ceiling?";

ASK Cost[1] : "How much capital does Project 1 require?";

ASK Cost[2] : "How much capital does Project 2 require?";

ASK Cost[3] : "How much capital does Project 3 require?";

ASK Cost[4] : "How much capital does Project 4 require?";
```

Figure 2.3.5: A VP-Expert rule base for a ranking heuristic with project dependencies

This VP-Expert program uses a backward chaining inference mechanism. It first asks the user to input the total capital available, b. It then asks the user to input all the capital requirements, denoted by *Cost[X]*. The net present value indices are already ranked in descending

order (alternatively, the user could input the NPV of each project, and a simple set of rules could order the projects). Therefore, the heuristic simply allocates capital in that order until all the funds are exhausted, while it ensures that the logic constraints are not violated. In the program the allocation is driven by the request *Find Project[X]*; this searches for a conclusion pertaining to Project[X]. If none exists for that Project, the program simply allocates funds (*FIND Decision*) to the Project subject to the remaining funds being sufficient. The syntax of VP-Expert requires that the variable *Decision* be uninstantiated before the next loop (RESET Decision).

Notice that the program can easily be adapted to evaluate more than four projects. The design of this program allows the user to add more logic constraints and maintain them in a rule base. Given the lack of control to completely enumerate all possible portfolios, a realistic optimization model can not be practically implemented in VP-Expert or in other popular expert systems shells. Prolog, a more general programming language that has served to build more sophisticated expert systems, is used in the next chapter to achieve enumeration, by capitalizing on its backtracking mechanism.

Chapter 3

Methodology.

In this chapter, we focus on the implementation of linear programming as the resource allocation mechanism of capital budgeting. The selected computer languages for the implementation are related to Prolog, which has been used to design many expert systems. As explained at the end of Chapter 2, a logic language like Prolog will allow a tighter coupling of the inference engine with a mathematical programming system, thereby enhancing interactive project selection. First, following the approach of Hamidi(1990), an enumeration strategy called *generate and test* in the logic programming literature (Clocksin, 1981) is implemented. The resulting Turbo-Prolog program is quite simple, although the generation of candidate solutions can take a large proportion of the overall computing time. Therefore, a second implementation aims at reducing the number of values to be potentially assigned to the variables. This second method is presented in the context of constraint satisfaction programming. Finally, CLP($\Re$), an extension of Prolog especially designed to handle linear constraints, is used to solve the continuous linear program. CLP($\Re$) has been used in the design of an expert system for portfolio management (Lassez, 1987).

Section 3.1 introduces logic programming, the foundation of the Prolog language. Then in Section 3.2, syntactic particularities of Turbo-Prolog are presented, before discussing the implementation of the integer programs; starting with the simple generation of integer values in a hypercube, there is a gradual implementation of an integer linear program based on the generate and test concept. An introduction to CLP($\Re$) is given in Section 3.4., followed by a linear program in CLP($\Re$) to solve the capital budgeting problem, such as encountered in the Network Modernization Program of Bell Canada. The programs excerpted in the examples (*Prog1 - Prog4*) were written and tested in Turbo-Prolog, and program *Prog5* was written and tested in CLP($\Re$). Complete working versions of all the programs are included in Appendices B, C and D.

The 0-1 integer linear program arising in the capital budgeting problems can be formulated as follows:

$$\text{Maximize} \quad Z = \sum_{j=1}^{n} c_j X_j \ ,$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} X_j \leq b_i \qquad i=1,\ldots,m1 \ ;$$

$$\sum_{j=1}^{n} a_{ij} X_j \geq b_i \qquad i=m1+1,\ldots,m2 \ ;$$

$$\sum_{j=1}^{n} a_{ij} X_j = b_i \qquad i=m2+1,\ldots,m \ ;$$

$$X = 0 \text{ or } 1 \ .$$

The first line describes the objective function, where c_j denotes the criterion for portfolio selection (e.g. NPV), and the inequalities and equalities represent the constraints. In this formulation, a project is either fully rejected or accepted. Correspondingly, the decision variables X_j are restricted to 0 or 1 values. Without loss of generality, the parameters a_{ij}, c_j, and b_i can be restricted to be integer.

The generate and test programs (Prog3 and Prog4) solve the above model, while the model solved by the CLP($\Re$) program is modified as follows:

$$\text{Maximize} \quad Z = \sum_{j=1}^{n} c_j X_j \ ,$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} X_j \leq b_i \qquad i=1,\ldots,m \ ;$$

$$X_j \geq 0 \text{ real} \ .$$

Here, the decision variables, X_j represent the fraction of Project j accepted.

3.1 Logic Programming

This brief review of elements of mathematical logic and logic programming is intended mostly to set the terms of references and the terminology used in this chapter. Mathematical logics are appropriate for representing knowledge in many situations, and well-known techniques of logical inference (automatic theorem-proving) can be applied to such representations. Two logics are commonly used. The propositional calculus considers the implications of logical relationships between propositions. The predicate calculus further utilizes the elements and structures of the propositions themselves. Its additional expressive power makes it an attractive representation mechanism for knowledge representation, which has been adopted in many Artificial Intelligence systems.

As an illustration of the kind of techniques of logical inference used in propositional calculus, we start with some propositions, and then use a rule of inference (say, Modus Ponens, Disjunctive Syllogism, Resolution etc.) to obtain new expressions which logically follow from the starting expressions. For example, suppose a telecommunication company may either offer ISDN services (denoted by X) or ($\vee$) dedicated digital networks (Y). Suppose further that this company may either not offer ISDN services or develop its own data transport technology (Z). We can deduce from these statements that either the telecommunication company will offer dedicated digital networks or it will develop its own data transport technology. We express this logical deduction using the resolution principle as follows:

Assume: $X \vee Y$

and $\neg X \vee Z$

Then: $Y \vee Z$

or $X \vee Y, \neg X \vee Z \Rightarrow Y \vee Z$

Resolution, when applicable, takes two *parent* clauses that share a complementary pair of *literals* and obtain a new *resolvent* clause. In the propositional calculus, a literal is a

propositional symbol (a variable such as X or ¬X) with or without a negation (¬) sign in front of it. A complementary pair of literals is a pair such that one literal is the negation of the other, e.g., X and ¬X. A *clause* is defined (Nilsson, 1980) as a well-formed formula consisting of a disjunction of literals, e.g., $X \vee Y \vee \neg Z$.

Predicate calculus, an extension of propositional calculus, allows one to *break down* the propositions and describe the objects which make up the propositions. Where one might use the symbol X in the propositional calculus to represent the statement "offer ISDN services", in the predicate calculus, one separates the predicate (services offered) from the objects (or subjects, here the sercive medium: ISDN), and writes:

Offered(ISDN).

In applying the resolution principle to clauses of the predicate calculus, detection of complementary pairs of literals is more complicated than in the propositional calculus, because the predicates take arguments, and the arguments in one literal are required to be *unifiable* or compatible with those in the corresponding literal. For example, the pair of clauses

$$P(f(a),x) \vee Q(x)$$
$$\neg P(g(a),x) \vee R(x)$$

cannot be resolved because the first argument of P in the first clause is incompatible with that of P in the second clause. In these expressions f and g are functions, x is a variable, a is a constant. On the other hand, the pair of clauses

$$P(f(a),x) \vee Q(x)$$
$$\neg P(y,g(b)) \vee R(g(b))$$

can be resolved. First, the *unification* operation is performed: a new version of the first clause is obtained by substituting g(b) for x; also a new version of the second is obtained by substituting

f(a) for y. The two resulting clauses, now unified, have a complementary pair of literals and resolve to yield $Q(g(b))\vee R(g(b))$.

A theorem-proving program takes as input a set of axioms and some formula to be proved. The output generally consists of information about whether a proof was found and, if so, what unifications were used to derive it. To prove theorems, logic programming languages, among which Prolog (Kowalski, 1977), rely on a special type of clause: a *Horn clause* in which at most one of the literals in the clauses is unnegated. Thus, the following are Horn clauses (where $P,Q,P_1,P_2,...,P_k$ represent propositions or atomic formulae i.e, of the form $R(c_1,...,c_n)$, where R is a predicate, each c_j is a term and $n\geq 0$):

$$\neg P\vee Q$$
$$\neg P_1\vee \neg P_2\vee...\vee \neg P_k\vee Q$$

these can be rewritten:

$$P \Rightarrow Q$$
$$P_1 \wedge P_2 \wedge...\wedge P_k \Rightarrow Q$$

We can rewrite the Horn clauses in the goal-oriented format as they appear in Prolog programs:

$$Q \Leftarrow P$$
$$Q \Leftarrow P_1 \wedge P_2 \wedge...\wedge P_k$$

Horn clauses have been used in logic programming languages because of their simplicity, and their dual interpretation as logic clauses, or as goals to be satisfied, i.e. specifications of computing procedures. In this thesis, clauses are named after the intended actions that they specify. For example, *update_bounds(..)* is used to denote either a logical relationship between incumbent bounds on variables and their updates, or more intuitively as procedures to update these bounds.

3.2 Capital Budgeting in Prolog.

3.2.1 Turbo-Prolog.

Prolog (an acronym for PROgramming in LOGic), was developed in Europe between 1972 and 1977. Alain Colmerauer and Philippe Roussel of the University of Marseilles are the names associated with the early development (Colmerauer, 1982). A compiler/interpreter was developed by David Warren, Fernando Pereira, and Lawrence Byrd at the University of Edinburgh between 1976 and 1977. Prolog is a programming language that implements a simplified version of predicate calculus relying on Horn clauses (Kowalski, 1977) and is thus a logical language.

Prolog has procedural and nonprocedural aspects. Instead of writing a procedure with an explicit sequence of steps, a Prolog programmer writes a declarative set of rules and facts that state relationships. Because of its declarative style, conventional flowcharts and coding techniques cannot be used with Prolog (Walker, 1987 p.26). Yet, Prolog rules are executed by an underlying *inference engine,* which does impose an order of execution. Because of the implicit ordering, Prolog rules also have a procedural interpretation.

The declarative aspect of Prolog programs facilitates their documentation; well designed programs clearly indicate not only the procedures required, but their logical relationships. Accordingly, extracts of the programs implemented in the following sections are displayed not only for illustration, but as algorithmic (or logic) descriptions of the methods implemented. An elementary understanding of Prolog is assumed. An incremental approach is used, whereby simple programs are developed to emphasize the basic elements of the generate and test and the constraint satisfaction approaches. Turbo-Prolog, one of the numerous implementations of the Prolog language, was chosen for the targeted application at Bell Canada because of its availability and ease of installation on popular micro computers under DOS. Special features of Turbo-Prolog used in the implementation are summarized next.

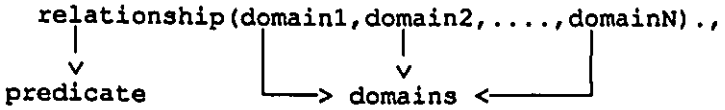
Syntactic Features of Turbo-Prolog.

The syntax of Turbo-Prolog is illustrated in Figure 3.2.1a, describing a program for the selection of four competing projects. Turbo-Prolog programs are composed of four major sections: *Domains, Predicates, Clauses, and Goal.* The beginning of each section is identified by the corresponding keyword. Below is a description of the purpose and function of each section, illustrated with excerpts from the program displayed in Figure 3.2.1a.

(1) **Domains:** defines the type of objects manipulated (integer, real, symbol for an alphanumeric type), e.g.

```
domains        /* required keyword */
m=integer.
```

(2) **Predicates:** A predicate represents a relationship among several objects called the arguments of the predicate. This section contains and defines the structure of all the predicates used throughout the program, by listing their argument types called domains:



e.g

```
predicates              /* required keyword */
budget(m,m)
plan(m,m)
rule(m,m).
```

(3) **Clauses:** this section constitutes the real program, it contains all the facts and rules. In Prolog, a fact is a statement that some entity has a particular property or that some relationship holds between two or more entities. Facts are expressed by applying a predicate to a list of arguments. Predicates may be defined in two ways: by a complete list of all the facts in which they occur; or by general rules that determine their truth or falsity. The rules are structured in the format of Horn clauses (see Section 3.1). In Turbo-Prolog, the facts corresponding to the same predicate, and the rules having this same predicate as a conclusion must be grouped

together; for instance, one must separate the list of predicates *plan* from the list of predicates *allocate*:

```
clauses                    /* required keyword */

plan(B,S1,S2,S3,S4)  :- S1+S2+S3+S4<=B,
                        S4>=S3.

allocate(X,S):-requirement(X,S).
allocate(_,0).
```

In a list of given predicates, Prolog syntax allows the facts and rules to be given in any order, but it is important to design a program, bearing in mind that the unification uses the predicates from top to bottom, and backtracking is chronological. Chronological backtracking is a search procedure that makes guesses at various points during problem-solving and returns to a previous point to make another choice when a guess leads to an unacceptable result (Hayes-Roth, Waterman, and Lenat 1983).

(4) **Goal:** defines the problem, or group of problems to solve; the goal is composed of subgoals, for example,

```
goal
total_budget(B),
allocate(1,S1), allocate(2,S2), allocate(3,S3), allocate(4,S4),
plan(B,S1,S2,S3,S4).
```

Turbo-Prolog can either contain the goal within the program (internal goal), or the user can supply the goal interactively at runtime (external goal). If an external goal is a fact to be verified, the answer returned is either "**true**" if the goal succeeds, or "**false**" if the goal fails. If, to satisfy an external goal, some of its variables need to be instantiated, then all the combinations of possible instantiations are returned. The goal of a program behaves otherwise as a *headless clause* similar to a Horn clause without the left side (the head).

In practice, the keyword *goal* preceding the internal goal specifies that running the

program will automatically attempt to satisfy its corresponding subgoals, using the standard computational mechanism of Prolog. For example, if a goal has only a single internal subgoal, e.g.

```
Goal
total_budget(B);
```

the process will seek to satisfy the subgoal. Also, when we use an internal goal, no indication is given about the satisfying values of the parameters, so one has to request that they be printed explicitly e.g.

```
Goal
total_budget(B), write("B=",B).
```

Execution Model.

The logic procedure of resolution is implemented in Prolog by:

- Unification and
- Backtracking.

It was indicated in Section 2.3.5 that popular expert system shells lacked proper control to enumerate feasible solutions of a mathematical program. In Prolog, backtracking and control thereof are essential mechanisms for the implementation of integer programming.

To illustrate the affinity of expert systems and mathematical programming made possible by logic programming, the capital budgeting resource allocation is presented under its two aspects. In Fig. 3.2.1a, it is introduced as an extension of the Ranking Heuristic done in VP-Expert (Section 2.3.5) that can enumerate all feasible portfolios of projects satisfying the constraints and logical conditions. In this program, mathematical programming notation is avoided, and the predicate *plan* can be viewed as a production rule. As a side feature, a repetition of the predicate *total_budget* is Prolog's natural way of requesting elementary what-if analysis, based on several budget levels.

```
domains
  file=output
  m=integer

predicates    /*  <----- list of functions with their arguments-domains */
  total_budget(m)
  plan(m,m,m,m,m)
  requirement(m,m)
  allocate(m,m)
  rule(m,m)

goal                        /*     <---------------------- internal goal */
  total_budget(B),
  allocate(1,S1), allocate(2,S2), allocate(3,S3), allocate(4,S4),
  plan(B,S1,S2,S3,S4),
  fail.                     /*       <-------------- forces backtracking  */

clauses
  plan(B,S1,S2,S3,S4) :- S1+S2+S3+S4<=B, S4>=S3,
                         write("S=[",S1,",",S2,",",S3,",",S4,"] B=",B),nl.

  requirement(1,36).
  requirement(2,24).
  requirement(3,45).
  requirement(4,68).
  requirement(4,34).    /* <-- it is possible to fund half of Project 4*/

  allocate(X,S):-requirement(X,S).
  allocate(_,0).

  total_budget(95).
  total_budget(135).
  total_budget(150).
```

Fig. 3.2.1a: A simple capital budgeting expert system program in Turbo-Prolog.

```
S=[36,24,0,34] B=95
S=[36,24,0,34] B=95
        .
S=[36,24,34,34] B=135
S=[36,24,0,68] B=135
        .
S=[36,24,34,34] B=150
S=[36,24,0,68] B=150
        .
```

Figure 3.2.1b: Partial output from the program of Fig. 3.2.1a.

77Fig. 3.2.1c. represents a similar problem in mathematical programming notation; it displays a dedicated Prolog program to enumerate capital projects satisfying a budget constraint and a logical constraint, represented by the following integer program:

enumerate X1 , X2
subject to
$$4X1 + 5X2 \leq 8$$
$$X2 \geq X1$$
$$X1, X2 = 0 \text{ or } 1$$

By providing an external goal, *plan(X1,X2)*, this program is able to generate all feasible solutions to the above integer program.

```
predicates
 plan(integer, integer)
 budget(integer)

clauses
 plan(X1,X2) :-
    budget(X1),
    budget(X2),
    4*X + 5*X2 <= 8,
    X1<=X2.

 budget(0).
 budget(1).
```

Fig. 3.2.1c
A Prolog program to enumerate feasible projects.

This example illustrates the compactness of Prolog in dealing with the declarative style of mathematical programming and acts also as an introduction to the more generalized programs described in the following sections.

3.2.2 Generate and Test.

We first implement an elementary program to generate all integer values of X_1 and X_2 satisfying $0 \leq X_1 \leq 10$, $0 \leq X_2 \leq 9$. Presenting this elementary example of the generate and test strategy serves two purposes. It removes much complexity typically associated with *test* while focusing on the elements of generate and test. It also serves to describe how candidate solutions are enumerated in Prog2, which conversely will emphasize the role of *testing* linear inequalities. The clauses section and an internal goal in Turbo-Prolog is shown in Figure 3.2.2a. In the predicate *enumerate_coordinate(X,L,U)*, the parameters X, L, and U are lists of integers representing the variables and their lower and upper bounds specified as input data. The recursive predicate *enumerate_coordinate* drives a predicate *integer_generator*.

```
[prog1]

  goal
  enumerate_coordinate([X1,X2],[0,0],[10,9]),
  write("X1=",X1,"X2=",X2),nl,
  fail.

  clauses

/*processes one coordinate at a time  */

  enumerate_coordinate([],[],[]).
  enumerate_coordinate([X_j|X],[L_j|L],[U_j|U]) :-
        integer_generator(X_j,L_j,U_j),
        enumerate_coordinate(X,L,U).

/*generates a new integer value for X_j*/

  integer_generator(X_j,X_j,U_j) :- X_j <= U_j.
  integer_generator(X_j,L_j,U_j) :- New_L_j=L_j+1,
        tester(New_L_j,U_j),
        integer_generator(X_j,New_L_j,U_j).

  tester(New_L_j,U_j) :- New_L_j ≤ U_j.
```

Figure 3.2.2a
A Generate and Test program for enumerating the vertices of a hypercube.

The *integer_generator* predicate plays the role of the guessing module while the predicate *tester* plays the checking module. In this program, the *integer_generator* and *tester* predicates were separated only for illustrative purposes; the checking module could have been intertwined with the generation module as in the next programs Prog2, Prog3 and Prog4. The predicate

enumerate_coordinate enumerates the sets in ascending order ([0,0], [0,1], [0,2] ,..., [0,9], [1,0],..., [10,9]), one at a time. The *integer_generator* predicate generates a candidate value of X and then the *tester* predicate checks the candidate's feasibility. The built-in backtracking capability of Prolog ensures that *integer_generator* remembers the candidates previously generated. In Turbo-Prolog, an expression of the form: New_L_j=L_j+1, assigns a value to New_L_j.

Note that the features of *Prog1* are general enough to accommodate a search in a hypercube of any dimension, by extending the goal to include more variables, as in goal2 (external goal); output2 is obtained.

goal2

```
enumerate_coordinate(X,[0,0,0],[1,1,1]),
```

Output2

```
X= [0,0,0]
X= [0,0,1]
X= [0,1,0]
X= [0,1,1]
X= [1,0,0]
X= [1,0,1]
X= [1,1,0]
X= [1,1,1]
```

This can loosely be interpreted as the enumeration of all combinations of projects with no budget constraints. In the next section, we see how to introduce constraints in the enumeration to achieve a more realistic capital budgeting problem.

3.2.3 Linear constraints

Prog2 is an elementary code for an integer program containing one or more inequality constraints of the type ≤ (less than or equal to). The parameter A is defined as the list of lists of integers a_{ij} of the integer program (page 36). The parameter B is defined as a list of integers b_i in the same model.

For example:

$$X1 + 3X2 \leq 5,\ \ 0 \leq X_1 \leq 10, \text{ and } \ 0 \leq X_2 \leq 9,$$

would be input as:

integer_program(X,[0,0],[10,9],[[1,3]],[5])

where L=[0,0], U=[10,9], A=[[1,3]] and B=[5].

```
{prog2}
    clauses

    integer_program(X,L,U,A,B) :-
          enumerate_coordinate(X,L,U),
          test_linear_constraints(X,A,B).
    integer_program([],[],[],_,[]).

 /*extracts one constraint at a time*/

    test_linear_constraints(_,[],[]).
    test_linear_constraints(X,[A_i|A],[B_i|B]) :-
              sum_product(X,A_i,Ax),
              Ax <= B_i,
              test_linear_constraints(X,A,B).

/* summation of the terms (A_ij.X_j) over j */

    sum_product([],[],0).
    sum_product([X_j|X],[A_j|A],New_Sum) :-
              sum_product(X,A,Sum).
              New_Sum=Sum+X_j*A_j,
```

Fig 3.2.3a
Partial generate and test program for integer programming with only ≤ constraints .

Prog2 differs from Prog1 by the addition of the predicate *test_linear_constraints (X,[A_i/A],[B_i/B])*. The second parameter denotes the matrix of coefficents of a linear system, the third parameter denotes its right hand-side, and the first parameter represents the integer variables satisfying the linear system with ≤ inequalities. The predicate *test_linear_constraints* invokes a predicate *sum_product(X,A_i,Ax)*, that computes:

$$Ax = \sum_{j=1}^{n} a_{ij}X_j .$$

where A_i contains the coefficients of a given constraint for a fixed index i.

Figure 3.2.3b shows how the predicates of Prog2 have been modified to accommodate equality and inequality constraints (=, ≤ and ≥). The predicate *test_linear_constraints(X,A,R,B)*

defined in Prog3 is different from that defined in Prog2; it includes an extra parameter R that indicates the type of constraint being processed. The parameter R is defined as a list of symbols; the only possible values of R are, "<", ">", and "=", to indicate the direction of the inequality ≤, ≥ and =.

```
{prog3}

  clauses
  integer_program(X,L,U,A,R,B,C,Z) :-
      enumerate_coordinate(X,L,U),
      test_linear_constraints(X,A,R,B),
      sum_product(X,C,Z),
      write("Z=",Z,"  X=",X),nl.
  integer_program([],[],[],[],[],[],[],_).

/* extracts and tests one constraint at a time */

  test_linear_constraints(_,[],[],[]).
  test_linear_constraints(X,[A_i|A],[R_i|R],[B_i|B]) :-
      sum_product(X,A_i,Ax),
      right_hand_side_feasibility(Ax,R_i,B_i),
      test_linear_constraints(X,A,R,B).

/* checks if the left hand-side is in proper relation
   with the right hand-side B */

  right_hand_side_feasibility(Ax,"<",B):- Ax <= B.
  right_hand_side_feasibility(Ax,">",B):- Ax >= B.
  right_hand_side_feasibility(Ax,"=",B):- Ax = B.
```

Fig 3.2.3b
Partial generate and test program for solving an integer program.

•

test_linear_constraints(X,A,R,B).

Once a complete set of values has been generated, the predicate *test_linear_constraints* verifies that the generated values satisfy all the constraints simultaneously. The predicate *test_linear_constraints* invokes two other predicates, *sum_product(X,A_i,Ax)* and *right_hand_side_feasibility(Ax,R_i,B_i)*. The predicate *sum_product* computes the sum of the product A_i*X and returns an answer Ax. Then the predicate *right_hand_side_feasibility* compares Ax with B_i depending on the symbol R_i. The predicate *sum_product(X,C,Z)* computes the objective function value, after a feasible set of values of X has been found.

3.2.4 Elementary analysis of complexity

In a generate and test approach, as illustrated in the previous programs (Figures 3.2.2a, 3.2.3a, and 3.2.3b), a generator enumerates all the possible values of some variables to be constrained. In capital budgeting problems with n indivisible projects and m constraints, the generator *enumerate_coordinate* will enumerate 2^n values. In the worst case, the testing predicate *test_linear_constraints* will perform a number of operations proportional to $m.2^n$, for instance when all constraints are feasible. It is difficult to evaluate an average computation time required by the predicate *test_linear_constraints*. Since all points must be tested by at least the first constraint, testing time will always comprise a term proportional to 2^n. Although it will typically increase with n and m, one can easily design some classes of data for which testing does not depend on the parameter m, for example, the extreme case when the first constraint eliminates all feasible points. At the other extreme, there are classes of data sets for which the term m can dominate the number of feasibility checks: consider for example a problem with m constraints, the first of which eliminates in 2^n steps all vertices of the unit hypercube, except its origin, which itself satisfies all other constraints; computing time to verify these m-1 constraints is proportional to m-1, which in contrived cases could be larger than 2^n.

The previous argument pointed to a factor intrinsic to the degree by which the constraints filter the vertices of the hypercube. When setting out to design a computational experiment, it was felt that computing an average ratio of right hand-side values to left hand-sides (evaluated at the center of the hypercube) might capture the tightness of the constraints. Many factors hamper this scheme, in particular the presence of equations. Instead, this constraining factor will simply be evaluated by the total number f of feasible integer points. To measure f on a simple scale, a constraining factor (CF) is defined as follows:

$$CF=1-(f/2^n)$$

The constraining factor is a real number between 0 and 1. A constraining factor CF=1 means that there are no feasible solutions in the problem. A constraining factor CF=0 means that all the constraints are satisfiable by all possible vertices of the hypercube. Obviously, datasets for

problems of given values of n and m (the number of constraints) may have different constraining factors. Thus, the constraining factor is used as a pragmatic measure to control experiments that is treated as independent from m and n. The computation time required by generate and test depends not only on the parameters summarizing the integer program, but also on the fashion that the problem is represented by the data. In the previous example, suppose now that the only tight constraint is at the end of the data set. Testing feasibility feasibility will then require a time proportional to $m2^n$, rather than $2^n + m-1$ if the constraint was at the beginning of the dataset. For given values m and n, the average time taken to test feasibility and therefore the overall computing time is expected to increase with f, and therefore to decrease with the constraining factor.

Linear inequalities have so far been used in a passive way, for testing feasibility. The next sections implement a method of constraint logic programming that uses the constraints to structure the domain of feasible solutions.

3.3 Constraint Satisfaction Programs

Our next program will attempt to reduce the computing time required by the generator, by avoiding to generate all the vertices of the unit hypercube. Rather than presenting the method as an ad-hoc remedy to solve capital budgeting problems, the next implementation is cast in the general framework of constraint satisfaction programming that attempts to reduce the combinatorial explosion faced by many search algorithms, especially in discrete optimization.

Logic programming languages are generally based on a depth-first search with chronological backtracking, therefore most formulations, when executed on a logic programming system (Prolog) tend to lead to a generate and test approach or *standard backtracking* (i.e depth-first search with chronological backtracking) (Van Hentenryck, 1989). It has been argued that this kind of search procedure is not very efficient. The basic reason for this inefficiency comes from the way constraints are used, only to reduce the search space *a posteriori* after discovering a failure. According to Naish (1985), these search procedures are oriented to recovering from failures and do not try to avoid failures. For example, in the selection of 7 indivisible projects with constraint

$$\sum_{j=1}^{7} x_j \leq 1,$$

generate and test would enumerate 128 binary vectors, and eliminate 120 of them.

The inefficiency of generate and test has been addressed through a variety of techniques (Haralick, 1980; Freuder, 1978) based on the idea of *a priori* pruning, that is, using the constraints to reduce the search space before the discovery of failure. Thus, the techniques to address the inefficiency of generate and test are oriented toward the prevention of failures and enable both an early detection of failure and a reduction of the backtracking and the constraint checks. A simple remedy to the unnecessary generation of 128 binary vectors would be to set all (but one) variables zero as soon as one variable is set at the value one.

In cases where pruning can be applied, it should be possible to provide information to the generator for recognizing that a class or range of proposed solutions will not meet some specified *elimination constraint.* That class or range of elements is dropped from further consideration. This process has been called *hierarchical generate and test* (HGT). Hierarchical generate and test is effective only when it is possible to factor the solution space and evaluate partial solutions. A variant of generate and test where only a portion of the solution is generated and tested is also classified as HGT; subsequent generate and test steps are required to achieve a complete solution. For example, given the inequality $X_1 + X_2 + X_3 \leq 2$, and the assignment $X_1 = 1$, there remains to generate binary values for X_2 and X_3 only within the triangle constrained by $X_2 + X_3 \leq 1$.

To factor a solution space, it must be possible to divide the space into classes that can be considered independently. The value of hierarchical generate and test derives from the quick reduction gained from early pruning of partial solutions. Therefore, it must be possible to predict that the final version of a solution would be acceptable by comparing a partial version against some specified constraint. We will call *test and generate*, a method where the pruning is enough to ensure that a candidate solution generated later will always be acceptable. However, most real life problems do not have enough parameters to enable a test and generate; we must be content with *test-generate-test.*

Hierarchical generate and test is frequently used for data interpretation problems that attempt to make identifications from a large volume of data. It is especially effective when the set of acceptable solutions is a small subset of all the possibilities. The most frequent condition preventing the use of HGT is the lack of an effective evaluator for partial solutions.

A formalization of the constraint satisfaction problem follows (Van Hentenryck, 1989):

> Assume the existence of a finite set of variables, $I=\{X_1,X_2,...,X_n\}$, which take values from their corresponding finite domains $D_1,D_2,...,D_n$ and a set of constraints. A constraint, $c(X_{i1}, X_{i2},...,X_{ik})$ among k variables from I is a subset of the cartesian product $D_{i1} \times .... \times D_{ik}$, which specifies which variables are compatible with each other.

The Constraint Satisfaction Problem can in general be expressed abstractly as the task of finding all sequences $(X^*_1,...,X^*_n)$ that satisfy some property $P_n(X^*_1,...,X^*_n)$, where $X^*_1,...X^*_n$ are the values given to $X_1,...,X_n$, and P_n holds if all the constraints are satisfied for this assignment. The task is to find all possible solutions. Since each variable ranges over a finite domain, in theory, we are able to decide on the feasibility of some of the sequences or the infeasibility of all sequences.

As inferred by the short examples of this section, linear integer programming for capital budgeting belongs to the above class of constraint satisfaction programs, because of the importance of the linear programming constraints which must be handled efficiently. In the next section, we design a constraint satisfaction progr
am in Turbo-Prolog to solve the integer program for capital budgeting, applying a specialization of the pruning concepts outlined above.

3.3.1 Domains of Variables.

The preceding section illustrated algebraic tests to reduce the number of solutions generated. Linear equations and inequalities produce substantial reduction of the search space by narrowing the domains of variables. Thus in general, it is not appropriate to delay their consideration until only one value is left uninstantiated. Next, we show how these constraints can be used systematically.

Given a linear constraint:

$$a_1X_1 + ... + a_nX_n \leq b_1,$$

if a_i is positive, an additional constraint can easily be derived as follows:

$$X_i \leq \frac{b_1 - \sum_{k=1, k\neq i}^{n} a_k \cdot {}^{-}X_k}{a_i} \qquad [1a].$$

If the coefficient a_k is positive, then ${}^{-}X_k$ is taken as the lower bound value of variable X_k. If a_k is negative, then ${}^{-}X_k$ is taken as the upper bound value of variable X_k. When the divisor a_i is negative, the equality sign in [1a] changes as follows:

$$X_i \geq \frac{b_1 - \sum_{k=1, k\neq i}^{n} a_k \cdot {}^{-}X_k}{a_i} \qquad [1b].$$

Similarly, $a_1X_1 + \ldots\ldots + a_nX_n \geq b_2$, for a_i positive yields:

$$X_i \geq \frac{b_2 - \sum_{k=1, k\neq i}^{n} a_k \cdot {}^{+}X_k}{a_i} \qquad [2a].$$

and for a_i negative,

$$X_i \leq \frac{b_2 - \sum_{k=1, k\neq i}^{n} a_k \cdot {}^{+}X_k}{a_i} \qquad [2b].$$

If the coefficient a_k is positive, then ${}^{+}X_k$ is taken as the upper bound value of variable X_k. If a_k is negative, then ${}^{+}X_k$ is taken as the lower bound value of variable X_k.

Finally, the equality constraint $a_1X_1 + \ldots\ldots + a_nX_n = b_3$ yields two additional constraints as follows:

The directions of inequalities [3a] and [3b] are reversed when the divisor a_i is negative.

$$X_i \geq \frac{b_3 - \sum_{k=1,k\neq i}^{n} a_k .^{+} X_k}{a_i} \qquad [3a]$$

$$X_i \leq \frac{b_3 - \sum_{k=1,k\neq i}^{n} a_k .^{-} X_k}{a_i} \qquad [3b].$$

3.3.2 Dynamic Domain Restriction.

In practice, values for the variables X_j will be generated sequentially. Once some of the variables have been instantiated, the formulae of the preceding section may be applied to provided tighter bounds on some of the remaining variables. Suppose that values have already been generated for variables X_k, k=1,..,i-1. We can rewrite [1a] as follows:

$$X_i \leq \frac{b_1' - \sum_{k=i+1}^{n} a_k .^{-} X_k}{a_i}$$

where an expression for b'_1 is given below, in which X'_k represents the instantiated value of X_k:

$$b_1' = b_1 - \sum_{k=1}^{i-1} a_k X_k' \qquad [4]$$

Similarly, [2a] can be rewritten as follows:

$$X_i \geq \frac{b_2' - \sum_{k=i+1}^{n} a_k .^{+} X_k}{a_i}$$

where

$$b_2^l = b_2 - \sum_{k=1}^{i-1} a_k X_k^l \qquad [5]$$

Finally, when a_i is positive, [3a] and [3b] can be rewritten respectively as follows:

$$X_i \geq \frac{b_3^l - \sum_{k=i+1}^{n} a_k \cdot^{+} X_k}{a_i}$$

$$X_i \leq \frac{b_3^l - \sum_{k=i+1}^{n} a_k \cdot^{-} X_k}{a_i}$$

where,

$$b_3^l = b_3 - \sum_{k=1}^{i-1} a_k X_k^l \qquad [6].$$

If a_i is negative, the directions of the inequalities are reversed.

The partial enumeration on which the inequalities of this section are based can be easily implemented in the framework of Prolog's chronological backtracking mechanism. As applications of a look-ahead scheme (test and generate), the expressions partially avoid failures by reducing the search space *a priori*, accelerating enumeration substantially.

The algorithm implemented in the program Prog4 displayed in Fig 3.3.3a dynamically incorporates the preceding inequalities as additional constraints; it extends beyond a simple generate and test scheme, to a look-ahead scheme that partially avoids failures. The following sections discusses the major predicates used in Prog4.

3.3.3 Implementation of test and generate.

A benefit of logic programming is its ease of documentation. The implementation of test and generate can be followed directly by consulting Fig. 3.3.3a. The general structure is described in the clause *integer_program* simplified below:

```
integer_program :-
        enumerate_coordinate,
        test_linear_constraints,
        sum_product                    (evaluate the objective function).
```

The details of the coordinate-wise enumeration can in turn be found in its corresponding clause, *enumerate_coordinate* which shows its recursive aspect explicitly. Thus, the methodological description of test and generate will unfold with the analysis of program Prog4. Well-known mathematical methods have been reported in a similar format, explaining the role of variables only locally within a recursive framework (Karmarkar, 1984).

The predicate *enumerate_coordinate (X,L,U,A,R,B)* holds the major innovations introduced by the program Prog4. The predicate *integer_program also refers to the auxiliary predicates test_linear_constraints (X,A,R,B)* and *sum_product(X,C,Z)* which are identical to those of Prog3. The following paragraphs explain in some detail the predicates that constitute the body of the *enumerate_coordinate* predicate. **For brevity, Figure 3.3.3a displays only one clause for each predicate. The other clauses can be found in Appendix C.**

```
{prog4}

clauses
integer_program(X,L,U,A,R,B,C,Z) :-
    enumerate_coordinate(X,L,U,A,R,B),
    test_linear_constraints(X,A,R,B),
    sum_product(X,C,Z),
    write("Z=  ",Z,"  X= ",X),nl.

/*----------processes one coordinate at a time-----------------*/

enumerate_coordinate([],[],[],_,_,_).
enumerate_coordinate([X_j|X],[L_j|L],[U_j|U],A,R,B) :-
    reduce_interval([L_j|L],[U_j|U],A,R,B,New_L_j,New_U_j),!,
    integer_generator(X_j,New_L_j,New_U_j),
    update_B(X_j,New_B,B,A),
    reduce_A(A,New_A),
    enumerate_coordinate(X,L,U,New_A,R,New_B).

/*----------extracts one constraint to reduce domain-----------*/

reduce_interval([L_j|_],[U_j|_],[],[],[],L_j,U_j).
reduce_interval([L_j|L],[U_j|U],[A_i|A],[R_i|R],[B_i|B],New_L_j,New_U_j):-
    update_bounds([L_j|L],[U_j|U],A_i,R_i,B_i,Better_L_j,Better_U_j),!,
    reduce_interval([Better_L_j|L],[Better_U_j|U],A,R,B,New_L_j,New_U_j).

/*-----uses the instantiated value X_j to update right hand sides---*/

update_B(_,[],[],[]).
update_B(X_j,[[A_j|_]|A],[B_j|B],[New_B_j|New_B]) :-
    New_B_j = -A_j*X_j + B_j,
    update_B(X_j,A,B,New_B).

/*-----shrinks the A-matrix by the column corresponding to X_j----*/

reduce_A([],[]).
reduce_A([[_|New_A_i]|A],[New_A_i|New_A]) :-
    reduce_A(A,New_A).

/*-----updates bounds of variable X_j using a single constraint----*/

update_bounds([L_j|L],[U_j|U],[A_j|A],"<",B_i,Better_L_j,Better_U_j) :-
    A_j<>0,
    left_hand_side_bound(L,U,A,A_times_bound),
   candidate_bound(L_j,U_j,A_j,B_i,A_times_bound,Better_L_j,Better_U_j).

/*------updates the bound using the left hand side-----------------*/

left_hand_side_bound([L_j|L],[_|U],[A_j|A],New_A_times_bound):-A_j >= 0,
    left_hand_side_bound(L,U,A,A_times_bound),
    New_A_times_bound = A_times_bound - L_j*A_j.

 candidate_bound(L_j,U_j,A_j,B_i,A_times_bound,L_j,Better_U_j):- A_j>0,
    Candidate_U_j=(B_i+A_times_bound)/A_j,
    min_and_max(Better_U_j,Candidate_U_j,U_j,_).

 min_and_max(X,X,Y,Y) :-  X <= Y.
 min_and_max(Y,X,Y,X) :-  X > Y.
```

Fig 3.3.3a
A portion of the look-ahead Turbo-Prolog program to solve the integer program.

reduce_interval(L,U,A,R,B,New_L_j,New_U_j)

The predicate *reduce_interval* constitutes the novelty of the program Prog4 over the previous one Prog3. It pre-screens the feasibility space before generation takes place. *Reduce_interval* is a direct implementation of inequalities [1a], [2a], [3a], and [3b] in the dynamic and sequential format shown in section 3.3.2. The *reduce_interval* predicate uses the available lower and upper bounds of each uninstantiated variable together with the given constraints to improve on the bounds of the variable awaiting instantiation. Consider the following constraints,

$$4X_1 + 5X_2 \leq 30,$$
$$2X_1 + X_2 \geq 10,$$
$$X_1 + X_2 = 14,$$
$$0 \leq X_2 \leq 9,$$
$$X_1 \geq 0.$$

Before invoking the *integer_generator* predicate to generate a value for X_1, the *enumerate_coordinate* predicate invokes *reduce_interval*. The predicate *reduce_interval* aims at tightening the lower and upper bounds of the variable X_1. It extracts the constraints sequentially and passes them to the predicate *update_bounds*. The predicate *update_bounds* processes each constraint and updates the bounds of X_1. The predicate *update_bounds* has four clauses corresponding to the three types of constraints and another one common to all types of constraints where A_j = 0 (Appendix C). Fig 3.3.3a shows only one case of the clauses that corresponds to the ≤ inequality and A_j ≠ 0 (in this case A_j is the coefficient of X_1).

To update the bounds of X_1, first of all, the predicate *update_bounds* invokes another predicate, *left_hand_side_bound* that applies [1a] to $4X_1 + 5X_2 \leq 30$. New bounds for X_1 are consequently obtained as follows:

$$0 \leq X_1 \leq 7.$$

Next we consider the system,

$$2X_1 + X_2 \geq 10,$$
$$X_1 + X_2 = 14,$$
$$0 \leq X_2 \leq 9,$$
$$0 \leq X_1 \leq 7,$$

which is handled by the predicate *update_bounds*, applying [2a] to $2X_1 + X_2 \geq 10$, to obtain stricter bounds for X_1 as follows:

$$1 \leq X_1 \leq 7.$$

Finally the predicate *update_bounds*, applies [3a] and [3b] to the following system of (in)equalities,

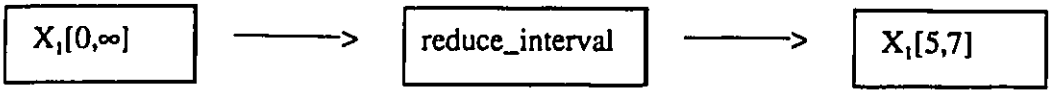
$$X_1 + X_2 = 14,$$
$$0 \leq X_2 \leq 9$$
$$1 \leq X_1 \leq 7$$

new bounds of X_1 are returned as follows:

$$5 \leq X_1 \leq 7.$$

Unlike functional programs, Prolog uses separate variables to keep updated values, hence the schema:

where New_L_j ≥ L_j and New_U_j ≤ U_j. Applying this schema to the above example we obtain,



The predicate *reduce_interval* module can drastically reduce the search space by pruning, and therefore improving considerably the time required for the search. However, improvement depends on the nature of the problem. Consistent with other applications of constraint satisfaction programming, problems with a small feasible region are shown to have very high improvement factors; some were 100 times faster in enumerating feasible solutions (see computation results in Chapter 4).

update_B(X_j,New_B,B,A)

The predicate *update_B* follows immediately after a given variable X_j has been instantiated; it uses the value of X_j to update the right hand-side of all the inequalities and equations, before attempting to generate a value for X_{j+1}. This predicate computes b'_1, b'_2, and b'_3 described in Section 5.1.4 by [1a], [2a], and [3c] respectively. In the program, the parameter New_B corresponds to the list of the updated right hand sides.

In the previous example, assume that a value of 5 has been generated for X_1 at one point in the enumeration; the original constraint, $4X_1 + 5X_2 \leq 30$ then becomes,

$$5X_2 \leq 10$$

Simply, the predicate *update_B* updates the vector B of right hand-side values by replacing the original value 30 by the new value 10. Then the restricted problem with variables $X_{j+1}...X_n$ can be treated recursively as the original one.

reduce_A(A,New_A).

As variables are set, the problem size is reduced, and so is the data. The predicate *reduce_A* ensures that the constraint matrix A is appropriately reduced at each iteration; once a variable X_j is instantiated, we need to recursively process the next variable X_{j+1}. This predicate eliminates the j-th column of data of matrix A corresponding to X_j, whose value has recently been

instantiated.

Within the scope of constraint satisfaction programming, the continuous version of capital budgeting cannot be handled by explicit domain enumeration, because the domain of the decision variables is infinite, although easily represented algebraically. To address the continuous capital budgeting problem, we turn to constraint logic programming.

3.4 Constraint Logic Programming (CLP)

Constraint Logic Programming allows one to define domains implicitly. Implicit representations are particularly important when one cannot obtain finite explicit representations. For example, the set of points (x,y) defined by $x \neq y$, or $x \geq y$, cannot be represented explicitly.

Constraint Logic Programming aims at extending the pattern matching mechanism of unification, as used in Prolog, by the more general mechanism of constraint satisfaction. The unification mechanism of logic programming is a particular case of constraint solving; it tells us whether two terms, such as f(x,a) and f(b,y), can be made identical by a particular instantiation of the variables x and y (here, x = b and y = a). In other words, it tells us that the equation f(x,a) = f(b,y) is solvable by explicitly representing the set of all solutions. For many domains of interest, such finite explicit representations do not exist; constraint satisfaction determines if the equation is solvable by comparing it with the constraints established on the variables: for example, f(x,a) = f(b,y) can be satisfied under the constraint $x \leq b$ but not under the constraint $x < b$.

Like Prolog, many constraint logic programming schemes use a syntax based on Horn clauses, which have the form

$$p(...) :- t_1(...),t_2(...),.....,t_n(...).$$

where p is a predicate symbol and the t terms are either predicates or constraints. The constraints consist of terms using operations in a particular domain of application. Some examples of domains include rationals, infinite trees, reals, and boolean.

3.4.1 CLP($\Re$)

CLP($\Re$) is an experimental implementation of the constraint logic programming paradigm in the domain of real arithmetic $\Re$. The language was developed by J. Jaffar and S. Michaylov at Monash University in Australia (Jaffar, 1990). One of the reasons for using this language is that

many capital budgeting problems, such as the one proposed by the Network Modernization Group, are in fact linear programs, which CLP($\Re$) is well equipped to solve, as shown in this chapter. Even solving budgeting problems with indivisible projects could be enhanced by CLP($\Re$), since most algorithms to solve integer programs also use linear programs repeatedly (Little, 1963).

Although most logic programming languages allow some arithmetic, this capability is an add-on feature, not integrated in the original design of the language. In CLP($\Re$), the semantics are defined directly in the domain $\Re$, and are therefore well-founded. Certain privileged predicates, such as equality, and various forms of inequalities ($\leq$, $\neq$, $\geq$) define *constraints*, and may comprise arithmetic expressions over real numbers as the following examples show.

Example 1 Consider the predicate:

```
double(X,Y) :- X=2*Y.
```

the goal,

```
?- double(X,2)
```

would succeed in both Prolog and CLP($\Re$), however, the goal,

```
?- double(4,Y).
```

succeeds in the CLP($\Re$) whereas it would fail in Prolog. Unlike standard Prolog, CLP($\Re$) is also able to respond correctly when queried about the following simultaneous equations:

Example 2.

```
?- 2X+Y=6, X+2Y=6.
```

The CLP($\Re$) interpreter, illustrated in Figure 3.4.1, contains a Prolog-like *engine*, a constraint *solver*, and a module that provides an *interface* between the two.

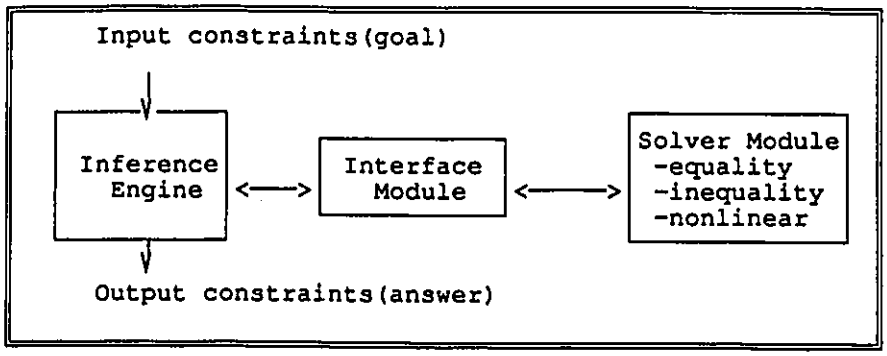


Figure 3.4.1:
A diagram of the components of the CLP($\Re$) interpreter.

The engine at the centre of the system is an inference mechanism that recognizes the constraints and hands over to the interface the constraints that it cannot solve. The major step performed by the engine is the creation of bindings of certain variables, plus the usual Prolog engine functions (unification, backtracking).

The interface evaluates complex arithmetic expressions and transforms constraints into a small number of standard forms. In this transformation, the input constraints are dispatched to different modules of the solver. The interface assists the engine as illustrated in the following example.

The constraint solver solves constraints that cannot be handled by the engine or the interface. This solver is the main novelty in the implementation of the CLP($\Re$) system. It is divided into three sub-modules, an equation solver, an inequality solver, and a nonlinear constraint solver. The nonlinear constraint solver provides a delay mechanism for nonlinear constraints by storing nonlinear equations, and sends equations to the linear equation solver whenever they become linear. Given a set of constraints, the solver first determines the solvability of that set and, if solvable, computes the solution. CLP($\Re$) uses a modified simplex method [Shambin, 1974] to solve sets of linear constraints.

A variant of Example 2 is now used to illustrate the role of the engine and of the interface in preparing the work for the solver:

```
eq1(S,T) :- 2S+T=6.
eq2(U,V) :- U+2V=6.

?- eq1(X,Y), eq2(X,Y).
```

The engine produces the following sequence of equations:

```
X=S
Y=T
2S+T=6
X=U
Y=V
U+2V=6
```

The interface simplifies this set and only sends to the solver the following two equations:

```
2X+Y=6
X+2Y=6
```

The solver then returns the solution

```
X=2, Y=2.
```

CLP($\Re$) also has the potential to produce implicit output. As constraints are defined syntactically, not only do they operate as tests, but they can define results. For example the following goal:

```
?- X1>1, X1>4, X1<5.
```

returns the answer below,

X1>4, X1<5.

The ability to return implicit answers is a powerful feature of CLP($\Re$). Answering a goal is defined in terms of satisfiability of the corresponding set of constraints and not in terms of finding a value. At each step of the process, the interpreter has to deal with only a single question: are the constraints solvable? This property is exploited in *Prog5* apreseneted in the next section.

3.4.2 Linear Programming in CLP($\Re$).

CLP($\Re$), a new experimental extension of Prolog for constraint logic programming, is chosen because it handles linear constraints, which form the core of the capital budgeting problem. In this section, a linear programming formulation incorporating *less than* inequalities ($\leq$) is represented and solved in CLP($\Re$), and yields an optimal solution.

3.4.3 Implementation of Linear Programming.

Figure 3.4.3a shows part of Prog5 (the complete program is given in Appendix D), the goal and the *constraints* predicates. The first predicate invoked by the goal is *readdata* ; this predicate reads the data from a file. The predicate *constraints* establishes linear inequalities, e.g. capital restriction, and the predicate *inbounds* sets the bounds on the variables.

{Prog5}

```
g( Z ) :-
    readdata( M, N, C , L, U, A, B, ),
    upperbound(N, C, L, U, Z_l, Z_u),
    constraints( M, N, X, A, B ),
    inbounds( N, X, L, U ), Epsilon = 1e-8,
    bisect( Z_l, Z_u, Z, Epsilon, N, X, C ),
    write (X).

.....

.....


constraints( 0, _, _, _, _ ).
constraints( M, N, X, [A_i|A], [B_i|B] ) :-
    constraint( Sum, N, X, A_i ),
    Sum <= B_i,
    constraints( M-1, N, X, A, B ).
```

Fig 3.4.3a
Part of a CLP($\Re$) program for a linear program with constraints $\leq$.

Below is a description of data variables read from the data file and subsequently used by the program.

Definition of the variables:

M: Total number of constraints.

N: The number of decision variables in the problem.

C: A vector of size N, comprising the objective function coefficients.

L: A vector of size N, comprising the lower bound values of the variables.

U: Same as L, for the upper bound values of the variables.

A: The M x N matrix of constraint coefficients.

B: A vector of size M comprising the right hand-sides of the constraints.

Bisection

A bisection algorithm is introduced to optimize the objective function; a standard method to transform an optimization problem into a series of satisfaction problems, which CLP($\Re$) is designed to solve. Initial brackets Z_l and Z_u for the bisection are computed by the *upperbound* predicate. This predicate works on a principle similar to that developed in Section 3.3.2, i.e.

$$Z_l = \sum_{k=1}^{n} (c_k^- X_k),$$

$$Z_u = \sum_{k=1}^{n} (c_k^+ X_k).$$

The bisection algorithm is accomplished by three clauses of the predicate *bisect* (see Fig 3.4.3b) as follows: the first clause checks if the accuracy required, *epsilon* (a constant specified before compilation) has been achieved.

```
/*** A Bisection of the objective value  ***/

    bisect( L, U, U, Epsilon, _, _, _ ):-
          U - L <= Epsilon.

    bisect( L, U, Z, Epsilon, N, X, C ):-
          U - L >  Epsilon,
          Mid = (U+L)/2,
          obj( Mid, N,  X, C),
          bisect( Mid, U, Z, Epsilon, N, X, C ).

    bisect( L, U, Z, Epsilon, N, X, C ):-
          U - L >  Epsilon,
          Mid = (U+L)/2,
          bisect( L, Mid, Z, Epsilon, N, X, C ).

/******** Objective Function *****************/

    obj( Thresh, N, X, C ):-
         objlist( C_X, N, X, C ),
         Thresh <= C_X.

    objlist( 0, 0, _, _ ).

    objlist( Sum, N, [X_j|X], [C_j|C] ) :-
         objlist( OldSum, N-1, X, C ),
         Sum = OldSum + X_j * C_j.
```

Fig 3.4.3b
Part program Prog5 performing bisection for linear programming.

For a given value of Z_l and Z_u, the second clause bisects the range of Z_l and Z_u into two halves; shifts the lower limit to the mid point $(Z_l+Z_u)/2$, then checks whether any values in the upper range are feasible. To check for feasibility, this clause *bisect* invokes a predicate *obj(Mid, N, X, C)*, where *Mid* is the mid point of the established objective value brackets.

If the previous clause *bisect* fails (no feasible values are found that satisfy an objective value equal to or higher than *Mid*, the last clause *bisect* applies. This clause shifts the upper limit to the mid point. The process is repeated until the range between the upper limit and the lower limit is within the error supplied by the user, thus, ensuring an optimum value to within the accuracy required. When *epsilon* is within the range of precision 10^{-8} set by CLP($\Re$) to verify equality, the solver treats the inequality $U - L \leq epsilon$ as an equality, thereby fixing the objective function at the common value of the variables U=L, and using this additional equality to determine values for the decision variables X, if the optimal solution is unique.

Chapter 4.

Computational Results.

The computational experiments of this chapter have two goals:

(i) to validate the methods described in Chapter 3.

(ii) to test the sensitivity of the computing time required by the 3 programs Prog3, Prog4 and Prog5 to variations of input data. These experiments do not follow an experimental design that would control of each of the 3 factors selected: the number of variables n, the number of constraints n and the constraining factor CF. More simply, datasets are constructed as variations of a base case, by adding variables or constraints. Resulting limitations are briefly discussed after each experiment.

4.1 Zero-one programming for capital budgeting in Turbo-Prolog

The first set of computational experiments is designed to compare the performance of the 2 programs to solve the integer program described in Chapter 3 (Sections 3.2.3 and 3.3.3). Prog3 implements the generate and test strategy (*G & T*) and Prog4 is based on a test and generate strategy (*T & G* also called look-ahead G & T).

The Turbo-Prolog program for the generate and test (*Prog3*) is included in Appendix B, and the program for the test and generate (*Prog4*) program is included in Appendix C. The programs were run on an IBM compatible PC (386-33) under Microsoft DOS 5.0 .

The data sets generated for the experiments, included in Appendix F, extend a base case provided by the Network Modernization Group of Bell Canada, which contains 10 variables and 5 constraints. A formulation of the problem follows.

$$\text{Maximize} \quad Z = \sum_{j=1}^{n} c_j X_j \ ,$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} X_j \leq b_i \qquad i=1,\ldots,m1 \ ;$$

$$\sum_{j=1}^{n} a_{ij} X_j \geq b_i \qquad i=m1+1,\ldots,m2 \ ;$$

$$\sum_{j=1}^{n} a_{ij} X_j = b_i \qquad i=m2+1,\ldots,m \ ;$$

$$X_j \text{ integer} \ .$$

The inequality constraints $\leq$ represent resource availability. In the datasets with 12 variables and 12 constraints, the first two inequalities represent capital and manpower availability. Each project requires these resources. The next two constraints indicate that a subset of projects require similar resources. Such a structure will occur if some projects are planned over a longer horizon than others, or in different administrative regions. The inequality constraints $\geq$ represent logical requirements. A first constraint $\geq$ specifies a minimum number of projects to be undertaken. The remaining inequality constraints $\geq$ model the dependency of two of three projects: $X_i \geq X_j$. Equality constraints specify that a balance b_i between the number of funded projects between two sets of projects must be maintained.

$$\sum_{j \in N1} X_j - \sum_{j \in N2} X_j = b_i \qquad .$$

Although no such requirement was included in the base case, the intent is to model functional or regional balance requirements, while validating the algorithm in the presence of equations. As a special case, the set N2 can be empty, and the equation forces a prespecified number b_i of projects of the set N1 to be performed. In other data sets with 12 variables, the types of the 12 constraints differ slightly. Similarly, smaller data sets feature the same combination of constraints, whenever possible; for example, one capital allocation, and one depreciation constraints are maintained, while the number of contingencies constraints is reduced.

To measure the central processing time elapsed, the Turbo-Prolog time counter is zeroed *(time(0,0,0,0)* just before enumeration starts with the predicate *integer_program.* The computation time, given by a predicate for elapsed time *time(H,M,S,T),* is measured immediately after enumeration. The timing is requested in the *goal* section of the programs (Fig. 4.1.5a and Appendices B and C). The computation times are reported in seconds throughout this chapter.

```
goal
time(0,0,0,0),
integer_program(...),
fail;
time(H,M,S,T)
write("***Mn=,M," Sec=",S," Cent=",T, "***"), nl.
```

Fig. 4.1.5a Part of the goal section containing the predicate for timing the program

Sensitivity to the number of variables

The elementary analysis of Section 3.2.4 predicts that the computation time of generate and test (Prog3) will increase exponentially with the number of variables. No analysis was performed for the more complex case of test and generate (Prog4), but the theory of computational complexity predicts that the computation time required to solve an integer program may increase exponentially with the number of its variables. For datasets involving from 3 to 30 variables, the first experiment (Fig 4.1a) exemplifies this, but shows that the coefficient of increase is much more moderate in the case of T & G than in the case of G & T. This result typifies the contrasting performance of two enumerative algorithms for integer programming: although both require exponentially increasing computing time as a function of the number of variables, test and generate is clearly more efficient than generate and test. This experiment uses data sets yielding a relatively stable constraining factor CF, as further analyzed in the next experiment.

Figure 4.1a

Computation time vs # variables(n)

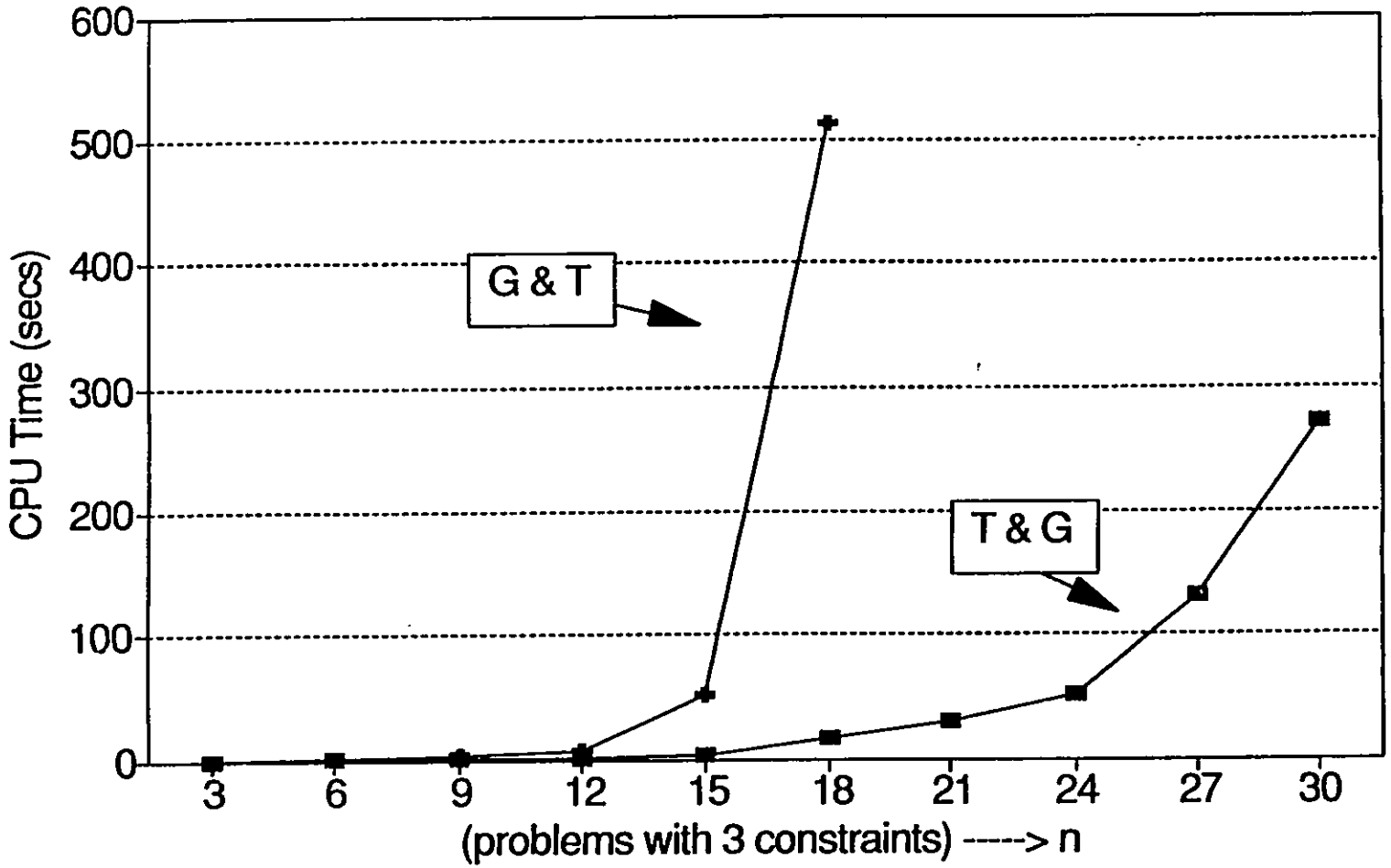
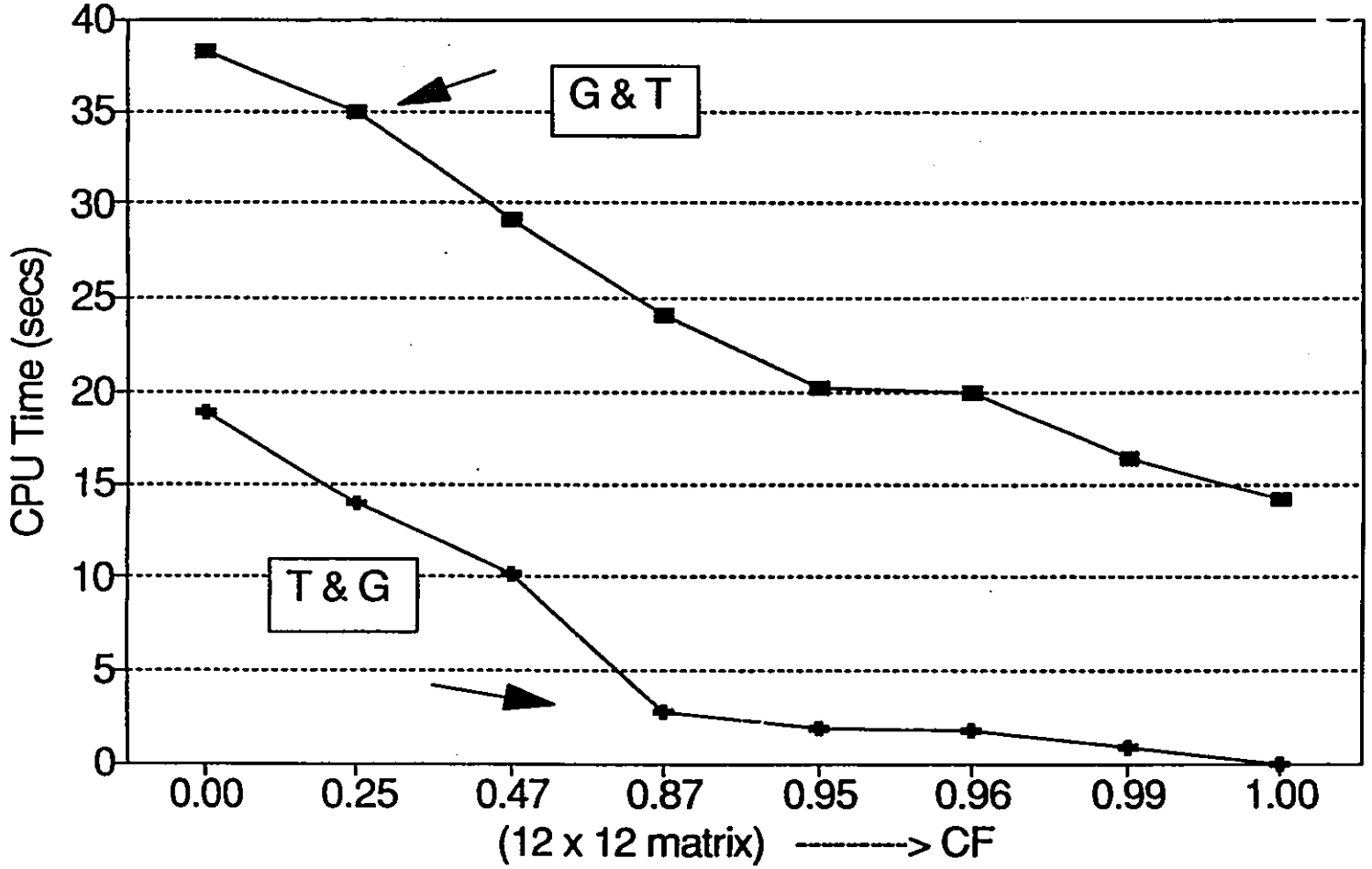


Figure 4.1b

Comp. time vs constraining factor (CF)



Sensitivity to the Constraining Factor:

The next experiment, reported in Figure 4.1b, evaluates the relative sensitivity in computation time of Prog3 (Generate and Test) and Prog4 (Test and Generate) to the constraining factor, CF, using datasets with 12 variables and 12 constraints,.

Most conspicuous is the different performance of G&T and T&G when the constraining factor is close to 1, i.e. when few feasible solutions exist. In Prog3 (G&T), generation remains as an overhead, whereas computing time of Prog4 (T&G) decreases, pointing out the effectiveness of using constraints early, to decrease the domain over which candidate solutions are generated. In that respect, the experiment confirms that Constraint Satisfaction Programming is mostly effective when few solutions exist. The experiment does not control the effect of the data configuration, e.g. the order in which constraints are processed: for example, all equations are processed last. As observed in Section 3.2.4, the computation time taken to test feasibility could vary markedly with this configuration. For instance, if the last constraints were the ones to determine the constraining factor, the computing time of Prog3 would be quite insensitive to CF.

Sensitivity to the number of constraints

Table 4.1 below shows the results of a preliminary test to display the sensitivity of program performance to the number of constraints for a 12 project problem.

Computation time (Seconds)		
T & G (prog4)	G & T (prog3)	# constraints (m)
15.3	110.4	6
10.4	115.2	9
8.3	135.4	12

Table 4.1 Computation times with increasing m.

In this typical case, the performance of T & G improves with additional constraints, while G &

T degenerates moderately. This occurs because additional constraints do not affect the number of integer values that need to be generated by G & T, but with these data sets, they are adding extra feasibility testing steps; thus, CPU is increasing with *m*. On the other hand, the additional constraints are decreasing the generation time of T & G, because they are dynamically reducing the generation space. In brief, the performance of each of the programs, and particularly that of T & G, depends substantially on the effect that the additional constraints have on the feasible space of solutions.

The next experiment attempts to control the effect of additional constraints on the overall tightness of the program, by selecting some variations of the base case that yield similar values of the constraining factor CF. Figure 4.1c reports the results of this experiment for a CF value of 0.94.

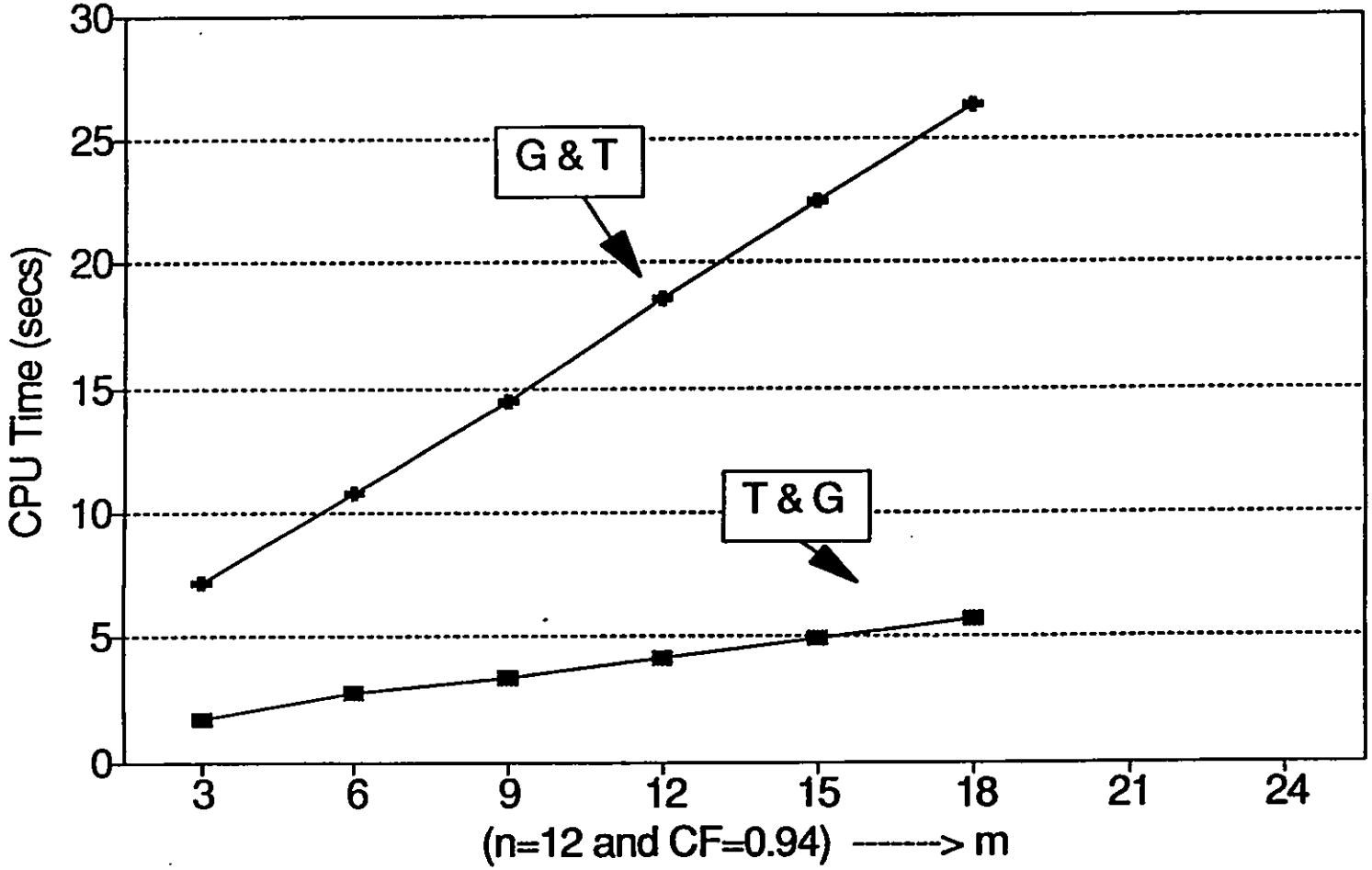
Maintaining a constant constraining factor introduces a quasi-linear relationship between the number of constraints and the computational time of each of the programs. In this experiment, additional constraints maintained a given constraining factor because they were redundant. Therefore their only role was to increase the CPU time to test feasibility, or to attempt unsuccessfully to update the bounds of the variables. Generally, the increase need not be linear because some vertices may be diagnosed infeasible by one of the additional constraints, thereby avoiding the test of all subsequent constraints. On the other hand, if these redundant constraints do not contribute to discard any vertex of the hypercube, the relationship between m and CPU time will indeed be linear.

Conclusion

For any realistic problem ($n \geq 4$), the test and generate (T & G) program is always faster than the generate and test program. The generate and test program reaches combinatorial explosion very quickly with n. On the other hand, the test and generate program performs significantly well under very large constraining factor ($CF \approx 1$). In particular, it takes very little time (Fig. 4.1b) to detect that a problem has no feasible solution.

Figure 4.1c

Computation time vs m



4.2 Linear programming for capital budgeting in CLP($\Re$)

The objective of this second set of computational experiments is to evaluate the performance of the CLP($\Re$) program presented in Section 3.4 in solving capital budgeting linear programs. The experiments test the sensitivity of CPU time to problem difficulty, measured by the number n of variables, and the number m of constraints.

Data sets

The datasets used in these tests reflect also the model implemented for Bell Canada's modernization program (with ten variables). The CLP($\Re$) program was designed to accept only the less or equal ($\leq$) type of constraints. Therefore, all inequalities are adapted to be of that same type.

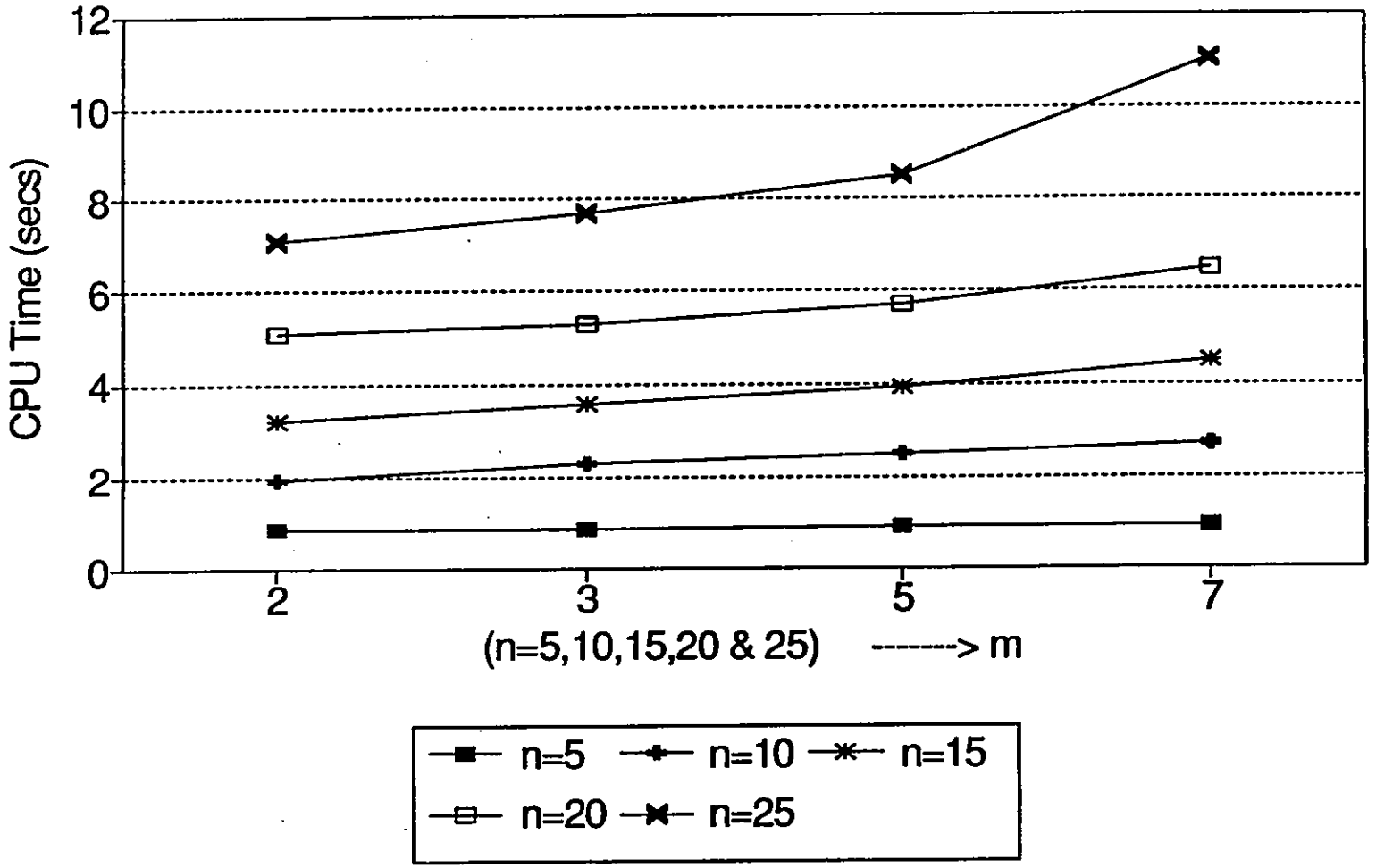
The complete linear program solver Prog5 written in CLP($\Re$) version 1.1, is included in Appendix D. The program was tested on SUN OS Release 4.1 (generic), an operating system of the Unix family. CLP($\Re$) uses a symbolic simplex algorithm to solve systems of linear inequalities. The size of the problems are limited to avoid tests with more than 20 seconds of CPU time.

Sensitivity to the number of constraints

The simplex algorithm is known to possess a theoretical exponential complexity with respect to the size of the problem. Empirical results reported in the literature display linear relationships between the number of constraints and CPU time (Dantzig, 1963, p.160; Hoffman, 1953; Wolfe, 1963; Goldfarb, 1977; Avis, 1978; Ho, 1983). The results presented in Figure 4.2a show a moderate increase in computational time with respect to m (number of constraints). There is a near linear relationship between performance and m, and the slope slightly increases with n.

Figure 4.2a

Comp. time vs # constraints (n fixed)



Sensitivity to the number of variables

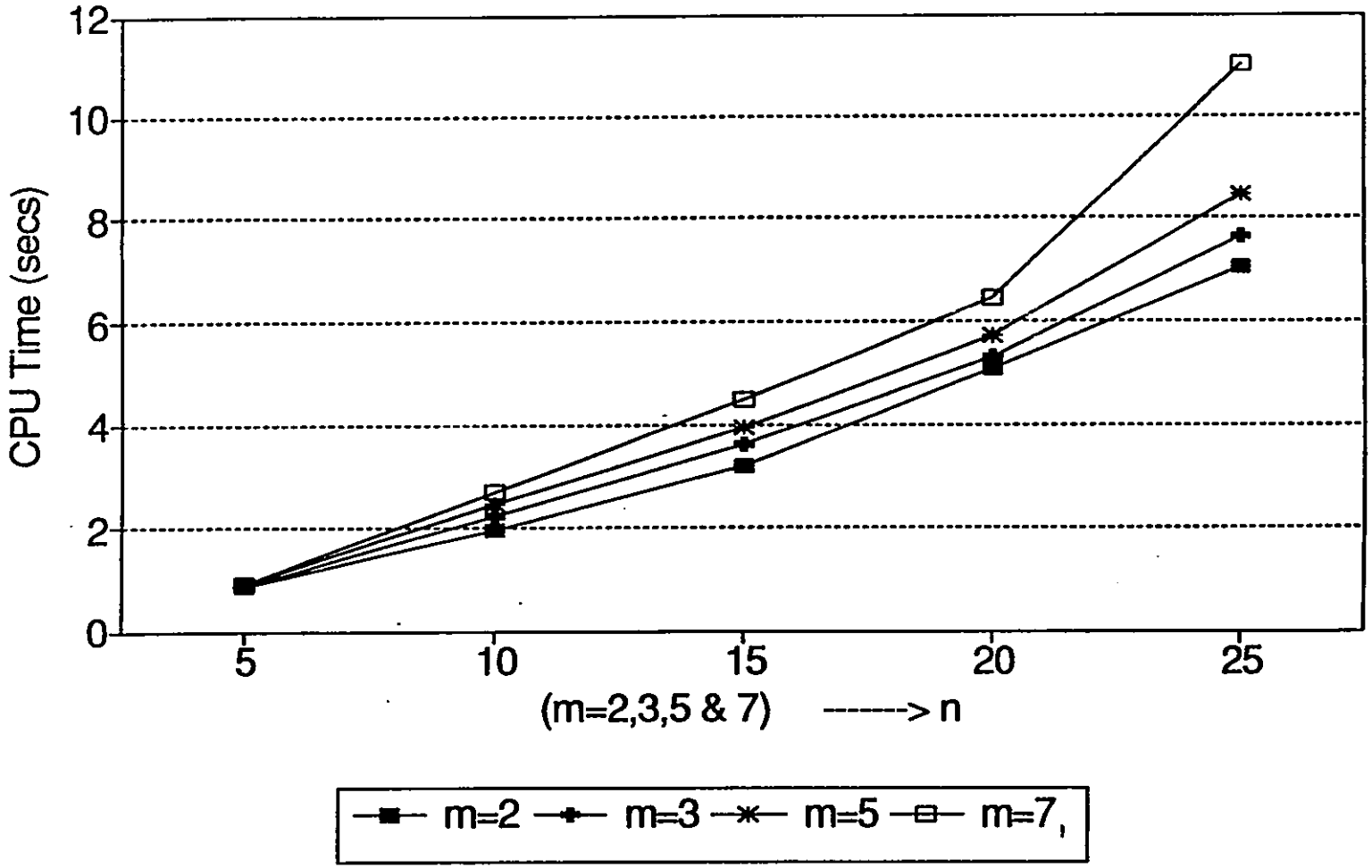
The computational time as shown in Figure 4.2b increases superlinearly as a function of n, although much less sharply than in the experiments with test and generate. The increase is attributed to the overhead required by CLP($\Re$) to carry out symbolic manipulation, i.e., find an explicit basis for the system, hand it to the interface (see Section 3.4), and finally report it in a format that the user can exploit.

Conclusion

For problems with at most 25 variables, the CLP($\Re$) program performed quite well. The increase in number of variables does not depict a clear combinatorial explosion as early as it did with the generate and test programs. It is possible to extend the linear program to solve integer programs in CLP($\Re$): for example, branch-and-bound can be implemented by recursive calls to the linear programs, extending the partial 0-1 candidate solution by one component in which the incumbent linear programming solution is fractional or not ground.

Figure 4.2b

Comp. time vs n (m fixed)



Chapter 5

Conclusion

In this thesis, some of the difficulties associated with the implementation of a mathematical programming approach have been analyzed, and expert systems suggested to alleviate these problems. A review of financial expert systems has revealed little evidence that current products address capital budgeting. It has shown what the state-of-the-art expert systems could offer toward solving capital budgeting problems, i.e. via a decomposition of the selection procedure: pre-scanning versus competition for scarce capital. This approach was shown to be compatible with both mathematical programming and expert system methods.

A simple application of the ranking heuristic in VP-Expert showed the limitation of some expert system shells, and the potential benefit of logic programming to implement mathematical programming within an expert system shell.

Prolog possesses the necessary properties for dealing with both logical and optimization aspects of capital budgeting. Although Prolog was not primarily designed to solve optimization problems, the backtracking facilities of the Prolog language suit that purpose; they liberate the programmer from having to design explicit backtracking procedures. The simplest use of enumeration in Prolog was illustrated by a short generate and test program, a well-known method used in expert systems, but newly applied to mathematical programming. In our specific generate and test strategy, the generate component is a small part of the program that accounts for the largest computing time. The enhanced version, test and generate, requires a more complex version of the generator. One contribution of the thesis is to apply the methods of constraint logic programming to the capital budgeting problem, and to show experimentally that (as customary in constraint logic programming) test and generate is most effective when the capital budgeting

problem has few feasible solutions, i.e. when the linear constraints drive the search and 'shape' the solution.

For continuous capital budgeting problems, CLP($\Re$) is an ideal extension of Prolog because it can satisfy simultaneous algebraic constraints as required to solve linear programs. Here, the contribution of the thesis is simply the use of an undocumented, experimental system, and its novel application to linear programming. In this case, as well as in test and generate for integer programming, the search for feasible solutions relies on the use of algebraic constraints.

Open problems

The most conspicuous problem is the actual collection of knowledge and the design of rules for capital budgeting by the Network Modernization Group. A review of existing rule base systems in the area of financial analysis expert systems may determine if these expert systems may be useful in addressing the Group's requirements.

Many refinements of Prog4, the program implementing test and generate, are possible. First is the determination of the order in which constraints should be processed to update the bounds. Another extension expected to tighten the domains of integers is to process the variables cyclically, i.e. using updated bounds of a variable X_1 to update the bounds of X_2, X_3 etc., and then using the new bounds to attempt again to tighten the bounds of X_1, X_2,.... The number of times this process is repeated may be predetermined or can be left to be determined dynamically.

Concurrent research for a master's thesis aims at replacing the bisection method implemented in CLP($\Re$) by a primal-dual approach, to be embedded in a branch-and-bound search. In practice, the enumeration is fairly delicate, because of the mode of presentation chosen in CLP($\Re$). For example, the value of a linear program can be found, while its solution remains undefined, which hampers branching.

A more immediate, and yet more far-reaching direction of research opened by this thesis,

is the design of an expert system relying more on allocation constraints than on using individual Project information. To simplify, if a constraint capital allocation is found to be critical in an exercise or over the years, the project selection rules could incorporate the linear constraint defining the ranking heuristic. This is not a step backward, since the heuristic would not stand alone, as it did in the VP-Expert implementation, but its underlying constraint would be the first of several domain-restricting constraints in a logic programming approach to mathematical programming for capital budgeting.

Appendix A

Linear Constraints.

This appendix presents a methodology of manipulating linear constraints to facilitate pruning of the feasible space. The results are adapted from a general calculus of intervals of variations. Variations intervals of the form $l \leq X \leq u$ can be viewed as special cases of the inequalities; then the results of (Van Hentenryck, 1989) can be obtained as a specialization of Fourier's elimination procedure (Fourier, 1826), the central role of which has been recently emphasized in (Lassez, 1989).

Equality.

Suppose the linear equality constraint has been *normalized* into an expression of the following form:

$$a_1X_1 + ... + a_nX_n + a = b_1Y_1 + ... + b_mY_m + b$$

Assume that

$$term1 = a_1X_1 + ... + a_nX_n + a$$

takes values in $[Min_1, Max_1]$, and that

$$term2 = b_1Y_1 + ... + b_mY_m + b$$

takes values in $[Min_2, Max_2]$.

To be equal, these two terms must necessarily range over the intersection of the 2 original ranges, [MN,MX], where MN is the maximum of Min_1 and Min_2 ($MN = Max[Min_1,Min_2]$), and MX is the minimum of Max_1 and Max_2 ($MX = Min[Max_1,Max_2]$).

The following equations show how to compute the variation intervals, Min_1 and Max_1 respectively.

$$Min_1 = \sum_{k=1}^{n} a_k \cdot {}^{-}X_k$$

$$Max_1 = \sum_{k=1}^{n} a_k \cdot {}^{+}X_k .$$

If the coefficient a_k is positive, then ${}^{+}X_k$ is taken as the upper bound value of that variable, if a_k is negative, then ${}^{+}X_k$ is taken as the lower bound value of that variable. Conversely, if the coefficient a_k is positive, then ${}^{-}X_k$ is taken as the lower bound value of that variable, if a_k is negative, then ${}^{-}X_k$ is taken as the upper bound value of that variable. Min_2 and Max_2 are computed in the same way.

From this, new constraints are derived for term1:

$$\sum_{k=1}^{i-1} a_k X_k + a_i X_i + \sum_{k=i+1}^{n} a_k X_k + a \geq MN \quad if \quad Min_1 < MN,$$

$$\sum_{k=1}^{i-1} a_k X_k + a_i X_i + \sum_{k=i+1}^{n} a_k X_k + a \leq MX \quad if \quad Max_1 > MX.$$

Similarly, for term2:

$$\sum_{k=1}^{j-1} b_k X_k + b_j X_j + \sum_{k=j+1}^{m} b_k X_k + b \geq MN \quad if \quad Min_2 < MN,$$

$$\sum_{k=1}^{j-1} b_k X_k + b_j X_j + \sum_{k=j+1}^{m} b_k X_k + b \leq MX \quad if \quad Max_2 > MX.$$

But these constraints directly imply other constraints on the values of the variables. Using term1, each variable X_i should satisfy,

$$a_i X_i \geq MN - \sum_{k=1,k\neq i}^{n} a_k \cdot {}^{+}X_k - a,$$

$$a_i X_i \leq MX - \sum_{k=1,k\neq i}^{n} a_k \cdot {}^{-}X_k - a,$$

A special case of this linear equality is where $Y_j = 0$, for j=1,..,m, giving

$$a_1X_1 + ... + a_nX_n = b_3,$$

where b_3 =(b - a). Then, the constraints on the values of variables become:

$$a_i X_i \geq Max[Min_1 - a\ ,\ b_3] - \sum_{k=1,k\neq i}^{n} a_k \cdot {}^{+}X_k\ ,$$

$$a_i X_i \leq Min[Max_1 - a\ ,\ b_3] - \sum_{k=1,k\neq i}^{n} a_k \cdot {}^{-}X_k\ .$$

Similar handling is achieved for linear inequalities as shown below.

Inequality (≥).

A linear inequality constraint of the type ≥ may be normalized into an expression of the following form:

$$a_1X_1 + ... + a_nX_n + a \geq b_1Y_1 + ... + b_mY_m + b$$

To satisfy the constraint, the two terms (term1 and term2), must take values in $[Min_2,Max_1]$. Consequently, the constraints on the values of the variables are now given by:

$$a_i X_i \geq Max[Min_1, Min_2] - \sum_{k=1, k\neq i}^{n} a_k . {}^{+}X_k - a,$$

$$a_i X_i \leq Max_1 - \sum_{k=1, k\neq i}^{n} a_k . {}^{-}X_k - a,$$

Again, the familiar case where $Y_j = 0$, for j=1,..,m gives,

$$a_1X_1 + ... + a_nX_n \geq b_2,$$

where $b_2 = (b - a)$, the constraints are now given by:

$$a_i X_i \geq Max[Min_1 - a, b_2] - \sum_{k=1, k\neq i}^{n} a_k . {}^{+}X_k ,$$

$$a_i X_i \leq Max_1 - \sum_{k=1, k\neq i}^{n} a_k . {}^{-}X_k - a,$$

Inequality ($\leq$).

A linear inequality constraint of the type $\leq$ may be normalized into an expression of the following form:

$$a_1X_1 + ... + a_nX_n + a \leq b_1Y_1 + ... + b_mY_m + b$$

To satisfy the constraint the two terms takes values in $[Min_1, Max_2]$. Hence, the constraints on the values of the variables are now given by:

$$a_i X_i \geq Min_1 - \sum_{k=1,k\neq i}^{n} a_k \cdot^{+} X_k - a,$$

$$a_i X_i \leq MX - \sum_{k=1,k\neq i}^{n} a_k \cdot^{-} X_k - a,$$

Again, $Y_j = 0$, for j=1,..,m giving,

$$a_1 X_1 + ... + a_n X_n \leq b_1$$

where $b_1 = (b - a)$, the constraints become:

$$a_i X_i \geq Min_1 - \sum_{k=1,k\neq i}^{n} a_k \cdot^{+} X_k - a$$

$$a_i X_i \leq Min[Max_1 - a , b_1] - \sum_{k=1,k\neq i}^{n} a_k \cdot^{-} X_k$$

Appendix B

Generate and Test Programs

This appendix presents three generate and test programs written in Turbo Prolog.

(i) Prog1 generates integer values in a hypercube,

(ii) Prog2 is a one type of constraint linear(integer) program, and

(iii) Prog3 is a program implementing a standard generate and test to solve the integer linear program. All variables in Prog3 can be real (or integer), with the exception of the decision variables upper and lower bounds that have to be integer.

```
/* Prog1 */

domains
list=i*
i=integer

predicates
enumerate_coordinate(list,list,list)
integer_generator(i,i,i)


goal

enumerate_coordinate([X1,X2],[0,0],[10,9]),
write("X1=",X1,"X2=",X2),nl,
fail.


clauses

/*------------- process one coordinate at a time ---------------*/


enumerate_coordinate([],[],[]).
enumerate_coordinate([X_j|X],[L_j|L],[U_j|U]) :-
      integer_generator(X_j,L_j,U_j),
      enumerate_coordinate(X,L,U).

/*---------- generate a new integer value for X_j ---------------*/

integer_generator(X_j,X_j,U_j) :- X_j <= U_j.
integer_generator(X_j,L_j,U_j) :- New_L_j=L_j+1,
      New_L_j <= U_j,
      integer_generator(X_j,New_L_j,U_j).
```

```
                                /* Prog2 */

/* Solve the integer program  max cx
                              s.t.
                              Ax '<=' b
                              Lower <= x <= Upper

Method:    generate all values between Lower and Upper, then test linear
           inequalities.

Notation:  the Turbo-Prolog variables A,B,C,X refer to the integer programming
           variables and parameters described above,
           ..._i refer to the i-th row, ..._j to the j-th column, "times" denotes
           the scalar product;

Timing:    feasible values are written to a file, this feature is eliminated
           during computation experiments to remove I/O overhead.          */

domains
file=output
i=integer
li=i*
lli=li*
r=real
lr=r*
llr=lr*
s=symbol
ls=s*

predicates
enumerate_coordinate(li,li,li)
integer_generator(i,i,i)
integer_program(li,li,li,llr,lr)
test_linear_constraints(li,llr,lr)
sum_product(li,lr,r)


goal
clearwindow,
openwrite(output, "c:prog2.out"),
writedevice(output),
time(0,0,0,0),

/*------------------------------ Input data -------------------------------*/

integer_program(X,
[0,0,0,0],
[1,1,1,1],
[
[50,180,98,90],
[1050,890,1450,1400]],
[350,3500]),
fail;
time (H,M,S,T),
write("***Mn=",M," Sec=",S," Cent=",T, "***"),
closefile(output),
write("***Mn=",M," Sec=",S," Cent=",T, "***"),nl.
```

```
clauses

  integer_program(X,L,U,A,B) :-
       enumerate_coordinate(X,L,U),
       test_linear_constraints(X,A,B)
       write("X= ",X),nl.
  integer_program([],[],[],_,[]).

/*------------------------ enumerate each variable ----------------------*/

  enumerate_coordinate([],[],[]).
  enumerate_coordinate([X_j|X],[L_j|L],[U_j|U]) :-
       integer_generator(X_j,L_j,U_j),
       enumerate_coordinate(X,L,U).


/*-------------- generate a new integer value for X_j ---------------*/

integer_generator(Lower_Bound,Lower_Bound,Upper_bound) :-
      Lower_Bound<=Upper_bound.

integer_generator(X,Lower_Bound,Upper_bound) :-
      Updated_Lower_Bound=Lower_Bound+1,
      Updated_Lower_Bound<=Upper_bound,
      integer_generator(X,Updated_Lower_Bound,Upper_bound).


/*------------------ test one constraint at a time ------------------*/

  test_linear_constraints(_,[],[]).
  test_linear_constraints(X,[A_i|A],[B_i|B]) :-
       sum_product(X,A_i,A_times_x),
       A_times_x <= B_i,
       test_linear_constraints(X,A,B).


/*------------ summation of the terms (A_ij.X_j) over j -------------*/

  sum_product([],[],0).
  sum_product([X_j|X],[A_j|A],New_Sum) :-
       sum_product(X,A,Sum),
       New_Sum=Sum+X_j*A_j.
```

```
                                /* Prog3 */

/* Solve the integer program  max cx
                              s.t.
                              Ax 'restriction' b
                              Lower <= x <= Upper

Method:    generate all values between Lower and Upper, then test linear
           inequalities.

Notation:  the Turbo-Prolog variables A,B,C,X refer to the integer programming
           variables and parameters described above,
           ..._i refer to the i-th row, ..._j to the j-th column.

Timing:  feasible values are written to a file, this feature is eliminated during
         computation experiments to remove I/O overhead.                        */

domains
file=output
i=integer
li=i*
lli=li*
r=real
lr=r*
llr=lr*
s=symbol
ls=s*

predicates
enumerate_coordinate(li,li,li)
integer_generator(i,i,i)
integer_program(li,li,li,llr,ls,lr,lr,r)
right_handside_feasibility(r,s,r)
test_linear_constraints(li,llr,ls,lr)
sum_product(li,lr,r)

goal
clearwindow,
openwrite(output, "c:prog3.out"),
writedevice(output),
time(0,0,0,0),


/*------------------------------ Input data ------------------------------*/

integer_program(X,
[0,0,0,0],
[1,1,1,1],
[
[     50,180,98,90],
[1050,890,1450,1400],
[1,1,1,1],
[0,1,0,0]],
["<","<",">","="],
[350,3500,3,1],
[133,101,185,263],Z),
fail;
time (H,M,S,T),
write("***Mn=",M," Sec=",S," Cent=",T, "****"),
closefile(output),
write("***Mn=",M," Sec=",S," Cent=",T, "****"),nl.
```

```
clauses

integer_program(X,Lower_Bound,Upper_bound,A,Restriction,B,C,Z) :-
      enumerate_coordinate(X,Lower_Bound,Upper_bound),
      test_linear_constraints(X,A,Restriction,B),
      sum_product(X,C,Z),
      write("Z= ",Z,"  X=",X),nl.


/********************************** GENERATE **********************************/

/*-------------------------- enumerate each variable --------------------------*/

enumerate_coordinate([],[],[]).
enumerate_coordinate([X_j|X],[L_j|L],[U_j|U]) :-
      integer_generator(X_j,L_j,U_j),
      enumerate_coordinate(X,L,U).


/*------------------- generates a new integer value for X_j ------------------*/


integer_generator(Lower_Bound,Lower_Bound,Upper_bound) :-
      Lower_Bound<=Upper_bound.

integer_generator(X,Lower_Bound,Upper_bound) :-
      Updated_Lower_Bound=Lower_Bound+1,
      Updated_Lower_Bound<=Upper_bound,
      integer_generator(X,Updated_Lower_Bound,Upper_bound).


/************************************ TEST ************************************/

/*------------------- use each constraint to test feasibility ----------------*/

test_linear_constraints(_,[],[],[]).
test_linear_constraints(X,[A_i|A],[Restriction_i|Restriction],[B_i|B]) :-
      sum_product(X,A_i,A_times_X),
      right_handside_feasibility(A_times_X,Restriction_i,B_i),
      test_linear_constraints(X,A,Restriction,B).


/*-------------------- general purpose inner vector product -------------------*/

sum_product([],[],0).
sum_product([X_j|X],[A_j|A],New_Sum) :-
      sum_product(X,A,Sum),
      New_Sum=X_j*A_j+Sum.


/*---------------- compare left and right side of each constraint ------------*/

right_handside_feasibility(Left_hand_side,"<",B) :-Left_hand_side<=B.
right_handside_feasibility(Left_hand_side,">",B) :-Left_hand_side>=B.
right_handside_feasibility(Left_hand_side,"=",B) :- Left_hand_side=B.
```

Appendix C

Look-Ahead Generate and Test (Test and Generate - T & G)

This appendix presents a Turbo Prolog program implementing a partial look_ahead generate and test (also called test and generate) methodology to solve the integer linear programming problem. All the variables data in this program are expected to be integer with the exception of the objective function coefficients.

```
/* Prog4 */

/* Solve the integer program  max cx
                              s.t.
                              Ax 'restriction' b
                              L <= x <= U

Method:    tighten the bounds L and U of each variable by looking at the image,
           through the linear constraints, of the bounds L' and U' of the other
           variables. Generate all values between Lower (L) and Upper (U) bounds,
           then proceed similarly with each of the other variables. Test whether
           each integer vector thus generated satisfy the linear constraints.

Notation:  the T-Prolog variables A,B,C,X,L,U refer to the integer programming
           variables and parameters described above,
           ..._i refer to the i-th row, ..._j to the j-th column, "times" denotes
           the scalar product;

           leftmost parameters are typically "input" parameters (except in
           min_and_max).

Timing:    feasible values are written to a file, this feature is  eliminated
           during computation experiments to remove the I/O overhead.        */


domains

file=output
i=integer

li=i*
lli=li*

r=real

lr=r*
llr=lr*

s=symbol
ls=s*


predicates

candidate_bound(i,i,i,i,i,i,i)
enumerate_coordinate(li,li,li,lli,ls,li)
integer_generator(i,i,i)
integer_program(li,li,li,lli,ls,li,li,i)
left_hand_side_bound(li,li,li,i)
min_and_max(i,I,i,i)
right_handside_feasibility(i,s,i)
reduce_A(lli,lli)
reduce_interval(li,li,lli,ls,li,i,i)
sum_product(li,li,i)
test_linear_constraints(li,lli,ls,li)
update_bounds(li,li,li,s,i,i,i)
update_B(i,lli,li,li)
```

```
goal
clearwindow,
openwrite(output, "c:prog4.out"),
writedevice(output),
time(0,0,0,0),


/*------------------------------ Input data ------------------------------*/

integer_program(X,
[0,0,0,0],
[1,1,1,1],
[
[     50,180,98,90],
[1050,890,1450,1400],
[1,1,1,1],
[0,1,0,0]],
["<","<",">","="],
[350,3500,3,1],
[133,101,185,263],Z),

fail;
time (H,M,S,T),
write("***Mn=",M," Sec=",S," Cent=",T, "***"),nl,
closefile(output),
write("***Mn=",M," Sec=",S," Cent=",T, "***"),nl.

clauses

integer_program(X,L,U,A,R,B,C,Z) :-
      enumerate_coordinate(X,L,U,A,R,B),
      test_linear_constraints(X,A,R,B),
      sum_product(X,C,Z),
      write("Z=  ",Z,"  X= ",X),nl.

/*------------------------ enumerate each variable ------------------------*/

enumerate_coordinate([],[],[],_,_,_).
enumerate_coordinate([X_j|X],[L_j|L],[U_j|U],A,R,B) :-
      reduce_interval([L_j|L],[U_j|U],A,R,B,Lower_Bound_j,Upper_Bound_j),!,
      integer_generator(X_j,Lower_Bound_j,Upper_Bound_j),
      update_B(X_j,A,B,New_B),
      reduce_A(A,New_A),
      enumerate_coordinate(X,L,U,New_A,R,New_B).


/*-------- reduce the interval over which each variable may range ----------*/

reduce_interval([L_j|_],[U_j|_],[],[],[],L_j,U_j).
reduce_interval([L_j|L],[U_j|U],[A_i|A],[R_i|R],[B_i|B],New_L_j,New_U_j):-
     update_bounds([L_j|L],[U_j|U],A_i,R_i,B_i,Better_L_j,Better_U_j),!,
     reduce_interval([Better_L_j|L],[Better_U_j|U],A,R,B,New_L_j,New_U_j).


/*--------------- update bounds of X_j using each constraint ---------------*/

update_bounds([L_j|_],[U_j|_],[A_j|_],_,_,L_j,U_j)    :-  A_j=0.

update_bounds([L_j|L],[U_j|U],[A_j|A],"<",B_i,Better_L_j,Better_U_j) :- A_j<>0,
      left_hand_side_bound(L,U,A,A_times_bound),
      candidate_bound(L_j,U_j,A_j,B_i,A_times_bound,Better_L_j,Better_U_j).
```

```
update_bounds([L_j|L],[U_j|U],[A_j|A],">",B_i,Better_L_j,Better_U_j) :- A_j<>0,
      left_hand_side_bound(U,L,A,A_times_bound),
      Neg_A_times_bound=-A_times_bound,                    /* Temporary exchange */
      Neg_B_i=-B_i,                                        /*  role of upper and */
      Neg_L_j=-U_j,                                        /*    lower bound     */
      Neg_U_j=-L_j,
      candidate_bound(Neg_L_j,Neg_U_j,A_j,Neg_B_i,Neg_A_times_bound,
                      Neg_Better_Lj,Neg_Better_Uj),
      Better_L_j=-Neg_Better_Uj,                           /*   restore upper    */
      Better_U_j=-Neg_Better_Lj.                           /* and lower bound    */

update_bounds([L_j|L],[U_j|U],[A_j|A],"=",B_i,Better_L_j,Better_U_j) :- A_j<>0,
      update_bounds([L_j|L],[U_j|U],[A_j|A],"<",B_i,Temp_L_j,Temp_U_j),
      update_bounds([Temp_L_j|L],[Temp_U_j|U],[A_j|A],">",B_i,Better_L_j,Better_U_j).


/*---- use bounds on uninstantiated variables for potential domain decrease --*/

left_hand_side_bound([],_,_,0).

left_hand_side_bound([L_j|L],[_|U],[A_j|A],New_A_times_bound) :-
      A_j >= 0,
      left_hand_side_bound(L,U,A,A_times_bound),
      New_A_times_bound = A_times_bound - L_j*A_j.

left_hand_side_bound([_|L],[U_j|U],[A_j|A],New_A_times_bound) :-
      A_j < 0,
      left_hand_side_bound(L,U,A,A_times_bound),
      New_A_times_bound = A_times_bound - U_j*A_j.


/*----using left hand-side, compute an alternative bound on variable X_j -----*/

candidate_bound(L_j,U_j,A_j,B_i,A_times_bound,L_j,New_U_j) :-
      A_j>0,
      Candidate_U_j=(B_i+A_times_bound)/A_j,
      min_and_max(New_U_j,Candidate_U_j,U_j,_).

candidate_bound(L_j,U_j,A_j,B_i,A_times_bound,New_L_j,U_j) :-
      A_j<0,
      Candidate_L_j=(B_i+A_times_bound)/A_j,
      min_and_max(_,Candidate_L_j,L_j,New_L_j).


/*-- compare old bound with candidate bounds and update if necessary ----------*/

min_and_max(X,X,Y,Y) :-  X <= Y.

min_and_max(Y,X,Y,X) :-  X > Y.


/*---------------------- generate an integer coordinate ----------------------*/

integer_generator(Lower_Bound,Lower_Bound,Upper_bound) :-
      Lower_Bound<=Upper_bound.

integer_generator(X,Lower_Bound,Upper_bound) :-
      Updated_Lower_Bound=Lower_Bound+1,
      Updated_Lower_Bound<=Upper_bound,
      integer_generator(X,Updated_Lower_Bound,Upper_bound).
```

```
/*--------------- update the right hand-sides of the constraints ---------------*/

update_B(_,[],[],[]).
update_B(X_j,[[A_j|_]|A],[B_j|B],[New_B_j|New_B]) :-
      New_B_j = -A_j*X_j + B_j,
      update_B(X_j,A,B,New_B).

/*------------------------- shrinks the constraint matrix ---------------------*/

reduce_A([],[]).
reduce_A([[_|New_A_i]|A],[New_A_i|New_A]) :-
      reduce_A(A,New_A).


/************************************ TEST ************************************/

/*------------------- use each constraint to test feasibility ------------------*/

test_linear_constraints(_,[],[],[]).
test_linear_constraints(X,[A_|A],[Restriction_i|Restriction],[B_i|B]) :-
      sum_product(X,A_,A_times_X),
      right_handside_feasibility(A_times_X,Restriction_i,B_i),
      test_linear_constraints(X,A,Restriction,B).

/*-------------------- general purpose inner vector product --------------------*/

sum_product([],[],0).
sum_product([X_j|X],[A_j|A],New_Sum) :-
      sum_product(X,A,Sum),
      New_Sum=X_j*A_j+Sum.

/*----------------- compare left with right side of each constraint ------------*/

right_handside_feasibility(Left_hand_side,"<",B) :-Left_hand_side<=B.
right_handside_feasibility(Left_hand_side,">",B) :-Left_hand_side>=B.
right_handside_feasibility(Left_hand_side,"=",B) :- Left_hand_side=B.
```

Appendix D

CLP($\Re$): Linear Program

This appendix presents a the computer program Prog5 for solving the continous linear program written in the experimental language, CLP($\Re$).

/* Prog5 */

```
/*------------------------------ Goal ---------------------------------*/

g( Z ) :-
     Epsilon = 1e-8,
     readdata( M, N, C , L, U, A, B, ),
     objective_bound(N, C, L, U, Z_l, Z_u),
     printf( "\n", [] ),
     printf( "Bounds: [ %d, %d ]\n", [Z_l, Z_u] ),
     ztime,
     constraints( M, N, X, A, B ),
     inbounds( N, X, L, U ),
     bisect( Z_l, Z_u, Z, Epsilon, N, X, C ),
     write (X),
     ctime( Time ),
     printf( "Time: %.2f secs.\n", [Time] ).


/******************************* File I/O ******************************/

readdata( M, N, C , L, U, A, B ):-
     see( 'budget.dat' ),
     read( M ),
     read( N ),
     printf(" # constraints: %d  , # variables: %d\n", [M, N]),
     readcjs(N, C),
     readlbs(N, L),
     readubs(N, U),
     readmatrix( M, N, A, B ),
     seen,
     printf( "Coeffs: ", [] ),
     write( A ),
     printf( "\n", [] ),
     printf( "Rhs:  ", [] ),
     write( B ).

readcjs(0, [] ).
readcjs(N, [C_j|C] ):-
     read(C_j),
     readcjs(N-1, C).

readlbs(0, []).
readlbs(N, [L_j|L]):-
     read(L_j),
     readlbs(N-1, L).

readubs(0, []).
readubs(N, [U_j|U]):-
     read(U_j),
     readubs(N-1, U).
```

```
readmatrix( 0, _, [], [] ).
readmatrix( M, N, [A_i|A],[B_i|B] ) :-
     readconstraint( N, A_i),
     read(B_i),
     readmatrix( M-1, N, A, B ).

     readconstraint(0,[]).
     readconstraint(N,[A_ij|A_i]) :-
          read(A_ij),
          readconstraint(N-1,A_i).


/******* Initial bounds on the value of the objective function *********/

objective_bound(0,_,_,_,0,0).

objective_bound(N,[C_j|C],[L_j|L],[U_j|U],MinCost,MaxCost):-
     Mincost1 = C_j * L_j,
     Maxcost1 = C_j * U_j,
     objective_bound( N-1, C, L, U, Nextmin, Nextmax ),
     MinCost = Mincost1 + Nextmin,
     MaxCost = Maxcost1 + Nextmax.


/**************** Explicit Linear Programming Constraints **************/

constraints( 0, _, _, _, _).
constraints( M, N, X, [A_i|A], [B_i|B] ) :-
     constraint( Sum, N, X, A_i ),
     Sum <= B_i,
     constraints( M-1, N, X, A, B ).


     constraint( 0, 0, [], [] ).
     constraint( Sum, N, [X_j|X], [A_ij|A_i] ):-
          constraint( OldSum, N-1, X, A_i),
          Sum = OldSum + X_j * A_ij.


/************************ Bounds on variables ************************/

inbounds( 0, _, _, _ ).

inbounds( N, [X_j|X], [L_j|L], [U_j|U] ) :-
     X_j >= L_j, X_j <= U_j,
     inbounds( N-1, X, L, U ).
```

```
/****************** Bisection of the objective value ******************/

/*-------------------- bisection bracket <= epsilon --------------------*/

bisect( L, U, U, Epsilon, _, _, _ ):-
    printf( " The optimum is between [%f,%f]\n", [L,U] ),
    U - L <= Epsilon.


/*--------------------- increase lower bracket -------------------------*/

bisect( L, U, Z, Epsilon, N, X, C ):-
     U - L > Epsilon,
     Mid = (U+L)/2,
     obj( Mid, N,  X, C),
     bisect( Mid, U, Z, N,  X, C ).


/*--------------------- decrease upper bracket -------------------------*/

bisect( L, U, Z, Epsilon, N, X, C ):-
     U - L > Epsilon,
     Mid = (U+L)/2,
     bisect( L, Mid, Z, N,  X, C ).

/************************ Objective Function ***********************/

obj( Thresh, N, X, C ):-
     obj_sum( C_X, N, X, C ),
     Thresh <= C_X.

     obj_sum( 0, 0, _, _ ).

     obj_sum( Sum, N, [X_j|X], [C_j|C] ) :-
          obj_sum( OldSum, N-1, X, C ),
          Sum = OldSum + X_j * C_j.
```

Appendix E

Survey of Expert Systems in Finance

The following list is compended from (Avignon, 1987; Bartee, 1988; Walker, 1990).

ATRANS (Automatic Funds Transfer System) is an expert system that automatically extracts information from unstructured, natural language messages, regarding bank-to-bank and customer transfers of money. Extracted information is then formatted according to SWIFT or custom bank message formats for automatic processing. The messages can be fed to ATRANS from a computer system that receives telexes or locally created unstructured, natural language messages. ATRANS processes money transfer messages only, but it can separate and process telexes containing multiple money transfers. (Cognitive Systems Inc.)

AUDITOR assists auditors in assessing a client's uncollectible accounts receivable. The use of AUDITOR as a decision support tool can underlay a firm's judgment process with a framework of consistent methods which can be independent of the biases of individual auditors. The program applies the knowledge domain of a sub-area of auditing to the generic inference engine. AUDITOR interacts with the user, asks for items of evidence, evaluates the new evidence with respect to what the system already knows, and then responds with recommendations or by asking for new evidence. (University of Illinois; Dungan 1985).

AUTHORIZER'S ASSISTANT was developed for American Express Corporation using ART. Its purpose is to help maintain convenience and safety to cardholders and merchants. American Express Corp. provides merchants who accept its card with an automated authorization service for charge approvals. Each hour, through this service, the company receives thousands of inquiries from merchants. The Authorizer's Assistant keeps this authorization system fast and reliable. The deployable system has about 1500 rules. An IBM-based computer system is used to handle most of the authorization inquiries. (Inference Corporation; Newquist, 1987).

BANK is a system developed to assist bank officers in qualifying mortgage applicants for various rates and schedules (Systems Designers Software, Inc.)

BUSINESSPLAN is a PC-based financial planning expert system for small businesses and self-employed professionals. It was designed for financial planners and their clients, has 7,500 rules and can handle up to 500 parameters. The system addresses risk management, income tax planning, and pension planning. BUISNESSPLAN utilizes artificial intelligence/expert systems technology to analyze the financial needs of the self-employed professional and business owner. The system not only analyzes qualitative and quantitative information, but also utilizes information about the planner's personal planning philosophy (Sterling Wentworth Corporation).

CASH TRADER is aimed at the management of portfolio of short-term "cash" instruments such as Certificates of Deposit and Repurchase Agreements. The institution dealing in these instruments has a "cash calendar" indicating expected supply and demand for cash within its own and its subsidiaries' operations. The objective is to balance the maturity of its market investments to the cash calendar (Arthur D. Little).

CASH VALUE is a knowledge-based capital projects planning package. It provides the user with cash flow forecasts, project evaluation and a detailed consultancy report based on project information and knowledge of finance and capital budgeting. Applications of this program include project investments, acquisitions and mergers, new ventures, and cost reduction programs (Heuros Ltd. and Hoskyns Ltd.)

CORPORATE FINANCIAL ANALYZER is a functional advisory system used to assist a financial professional in the analysis of the current and future financial states of a corporation. This system is designed to be incorporated into activities such as investment analysis, merger and acquisition planning, commercial loan approval, corporate financial management as well as other financial activities requiring the ability to perform "what if" analysis by specifying and assessing various hypothetical financial scenarios.

ELOISE (English Language-Oriented Indexing System for EDGAR) was developed for the U.S. Securities and Exchange Commission in connection with its EDGAR pilot. The system focuses on concepts found in proxy statements, a common type of SEC filing. ELOISE searches each document for specific concepts of interest. The system detects the presence of predefined concepts. ELOISE also provides some basic reference information about the document, and highlights the sentence that contains the selected concept (Arthur Andersen & Co.)

EQUITY TRADER is designed to assist equity traders in an organization where the actual trades in stocks are ordered by a portfolio manager and are executed by traders in communication with the different stockbrokers. The system will analyze the trader's buy and sell requirements in relation to the evolving state of the market, to detect orders that can be carried out at any time without risk, and to recommend brokers for each transaction (Arthur D. Little, Inc.)

EXPERT EXPORT is an expert for supporting export activities. A series of expert systems have been developed, aiming at the training of export business executives and contributes to the decision process in export activities, including: customs regulations, transport strategy selection and optimization, currency control regulation, security, sanitary control, and others. (Cognitech).

FAME (FinAncial Marketing Expertise) is a prototype expert system for consultation in financial marketing as it applies to the marketing of computers (Expert Systems Group, IBM Tokyo Research Laboratory).

FINANCIAL ADVISOR assesses company performance and recommends remedial action if necessary (Helix Products and Marketing, Ltd.)

FINANCIAL ADVISOR assists corporate executives in planning, formulating, evaluating and monitoring all types of capital intensive projects and products. It combines expert financial, economic, tax, and accounting experience, with sophisticated mathematic techniques to help untangle problems of enormous complexity without requiring the user to have expert knowledge of finance or how computers work (Palladian Software).

FINEX is an expert system for financial analysis (University of Southern California).

FINSTA is an expert system developed to aid in the design of accounting information system by developing aggregated financial statements from a set of accounts in order to improve management decision making. The system uses PROLOG to model the judgments of a management consultant (University of Southern California).

FINPLAN is a preliminary prototype program performing personal financial planning services. It is implemented primarily as a production system using expert system concepts. Two principle tasks of financial planning are addressed: (1) determining a sufficient amount of life insurance coverage and (2) providing recommendations regarding an investment portfolio for a given client-user. The determination of life insurance needs represents a class of numerically oriented problems, which are common in the domain of personal financial planning. FINPLAN offers a solution to this class of problems by integrating numerical algorithms with production rules. Recommendations regarding the client's investment portfolio are based primarily on inference procedures and other expert system concepts. FINPLAN selects appropriate investment classes by matching the characteristics of each investment class with the client's own temperature and goals (J.D. Pinc, Wright-Patterson AFB).

FOLIO is used by portfolio managers to assist in determining a clients investment goals and then selects the portfolios that best meet those goals (Stanford University).

FSA (Financial Statement Analyzer) was developed for the U.S. Securities and Exchange Commission in connection with its EDGAR pilot. EDGAR is a system that electronically receives SEC filings directly from filing companies. FSA is a prototype that analyzes the type of financial information contained in EDGAR's filings. The purpose of FSA is to perform consistent financial analysis, despite differences among individual statements. This analysis focuses on the calculation of standard financial ratios, such as the Quick Ratio. FSA reads the relevant financial tables and footnotes of companies like those contained in EDGAR's databases, and then presents the ratio calculation. FSA was developed using IntelliCorp's KEE software (Arthur Andersen & Co.)

FOREIGN EXCHANGE ADVISORY SYSTEM is a knowledge based system that provides expert financial advice to financial professionals involved in foreign exchange. This is a functional expert system for traders with financial organizations, treasurers of multinational corporations and international portfolio managers. This system specifically advises financial professionals in developing strategies for trading foreign-currency options. It provides advice for both the initial development of a strategy in foreign currency option trading as well as suggestions for future modifications to the option strategy as a result of currency price

movements in the marketplace. This system was developed using ART from Inference Corporation and requires either an ART run time or development system (The Athena Group).

GERI is an expert system for currency exchange risk management (Cognitech).

IGIRS is an expert system for pension plan processing. This system is capable of supporting the final phase of pension management administration: gathering and checking the carrier information for a pension case, and liquidating the pension file. Interfaces with databases are made using natural language (Cognitech).

INGOT is an expert system that assists in financial forecasting (Schonfeld & Associates).

LE COURTIER is a stock market advice system. The system offers advice based on a customer's answer to questions regarding age, income, desire for liquidity, current debts, future intentions, and the desired degree of risk (Cognitive Systems).

LENDING ADVISOR is an expert system designed to assist loan officers in evaluating the credit risks associated with commercial loan applications. The system does this by capturing the expertise of experienced loan officers and then providing that knowledge to other professionals. In this way, the Lending Advisor system can strengthen credit evaluations and thus decrease loan losses, improve the productivity of loan officers, increase management effectiveness and dramatically shorten training time (Syntelligence).

MORMAN is a system that will automate the processing of mortgage applications. An electronic version of the loan company's application form is shown on the screen and details are completed in consultation with the applicant. When the form is completed, the system checks that the proposed loan is in line with the company's lending policy, and if so, goes ahead with arranging the mortgage. When approval is given on the mortgage, the system will produce the appropriate letters, such as to employers, or existing mortgage holders (Financial and Corporate Modeling Consultants).

PALLADIAN CAPITAL INVESTMENT is an expert system to help managers make the right financial judgments. Using a wide range of sophisticated financial techniques, the system helps evaluate new product proposals, capacity expansion plans, and cost reduction proposals.

PALLADIAN FINANCIAL ADVISOR is designed to help managers make the right financial judgments. Using a wide range of sophisticated financial techniques, the system can help evaluate new product proposals, capacity expansion plans, and cost reduction proposals; decide between equipment make/buy and lease/ buy situations; and assess major strategic acquisitions and investments. The Palladian system also examines all input it is given, knows how to perform the needed calculations automatically, explores alternatives, and explains how it arrives at each step in the analysis. This makes the system essentially the same as having a team of financial experts on-staff at all times. Like expert consultants, the system tests every input assumption and conclusion against an extensive knowledge of general business practice (Palladian Software, Inc.)

PERSONAL FINANCIAL PLANNER prepares total financial plans for individuals with income in the range of $25,000 to $100,000. It contains multiple expert system modules for areas such as investment, education, and retirement. It incorporates a database management with backtracking capabilities (Arthur D. Little, Inc.)

PLANMAN is a financial planning system that runs on an IBM PC. It assists planners in developing a plan for an individual. It uses over 7500 decision rules to generate an extensive written report. PLANMAN accommodates the personal attitudes of the client and the planning philosophy of the user to create a custom document (Sterling Wentworth Corp.)

PLAN POWER is an expert systems-based financial planning tool. It is aimed at banks, brokerage firms and other financial institutions who want to offer financial planning services to their clients without increasing their professional staff. The program incorporates a database and spreadsheet as well as the inference mechanism and knowledge base needed by the expert systems side of the package. The knowledge base contains information on six financial areas: cash and credit management, risk management, tax planning, investment management, retirement planning, and estate conservation. The completed plan for each client will include recommendations in each of these areas. PLAN POWER also includes a computer text facility that enables the system to take findings, analysis, and recommendations of the expert system and generate a final plan in English (Applied Expert Systems, Inc.)

PORTFOLIO MANAGEMENT ADVISOR is a functional expert system that advises professional portfolio managers in the construction and maintenance of investment portfolios and is particularly well suited for banks, corporations, pension funds, mutual funds, etc. It allows a portfolio manager to incorporate qualitative reasoning into the analysis of the economy, the various markets, business sectors, and corporations as well as into the analysis of the investments themselves. This expert system was developed in ART from Inference Corp. and requires either an ART run time or development system (The Athena Group).

PREFACE-EXPERT is an expert system designed to diagnose new companies' projects from a financial point of view. It produces a series of common financial statistics, statements and forecasts. It proposes alternative plans, improving upon the initial plan by successive interactive simulation. (ESIEE; Senicourt, 1987).

ROME is a knowledge-based system under development which would support decision-making in the area of long-range financial planning. The program is still under development, and two ROME subsystems, ROMULUS (a natural language interface) and REMUS (the financial model reviewing expert) have already been implemented. The project is being sponsored by Digital Equipment Corporation (Carnegie-Mellon University).

STOCK OPTIONS ANALYSIS is an expert advisory system that includes corporate evaluations and stock and option pricing in the context of various economic and market conditions. Additionally, the system can formulate various hedging strategies and determine their effectiveness by probabilistically carrying out various hypothetical economic and market scenarios

at both the corporate level and at the marketable securities level. It makes use of three subsystems: a corporate finances analysis subsystem, a corporate stock analysis subsystem, and a stock option pricing subsystem (The Athena Group).

SYNTEL is a software tool that blends expert system technology with advanced concepts in programming languages, and has led to the development of powerful and practical expert systems; for example, one that assists financial institutions in evaluating the risk and return associated with asset and liability portfolios and another that supports the preparation of complex bids by engineering and construction firms (Syntelligence; Duda, 1987).

TAX ADVISOR, an AI-based package developed by Professors Michaelsen and Michie, provides estate tax planning advice for wealthy clients. It contains 275 rules and is programmed in InterLISP and EMYCIN.

TRADER'S ASSISTANT assists securities traders in assessing the state of the stock market, focusing on the instantaneous supply and demand of the stock, as well as the significance of current rumors and other hearsay (Arthur D. Little, Inc.)

XFIN (Financial Analysis Expert Advisor) is a module that evaluates the price and performance of recommended options. In the production version of SMARTX, XFIN will employ maximum negative cash flow, payoff period, present value, and rate of return coupled with an articulation of the payoff or benefit. The intent is to provide an option to perform sensitivity analysis.

Appendix F

Datasets.

This appendix presents the datasets used in the computation results of Chapter 4.
For convenience, the data for the two integer programs (G & T and T & G) was input in the format data(X,L,U,A1,B1,A2,B2,A3,B3,C,Z). The reason for accepting the data in this format was

(i) data had already been generated for a program accepting that format, and

(ii) the data is more readable in the older format (data(..)), but the program is greatly enhanced by accepting the new data format (integer_program(..)). To convert the datasets to the new format the small data interface module shown below was used.

```
/******************** DATA INTERFACE*********************/


predicates
data(li,li,li,lli,li,lli,li,lli,li,li,i)
rel(li,s,ls)
append(ls,ls,ls)
append(li,li,li)
append(lli,lli,lli)


clauses
data(X,L,U,A1,B1,A2,B2,A3,B3,C,Z) :-
     rel(B1,"<",R1),rel(B2,">",R2),rel(B3,"=",R3),
     append(R1,R2,R4), append(R4,R3,R),
     append(A1,A2,A4), append(A4,A3,A),
     append(B1,B2,B4), append(B4,B3,B),
     integer_program(X,L,U,A,R,B,C,Z).

rel([],_,[]).
rel([_|B],R_j,[R_j|R]):- rel(B,R_j,R).

append([],L,L).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).
```

The following capital budgeting datasets are used to test the influence of the number of variables, n on the computation time.

```
/* n=6, m=3  */
data(X,
[ 0,   0,   0,   0,   0,   0],                    /* Lower bounds l_j */
[ 1,   1,   1,   1,   1,   1],                    /* Upper bounds u_j */
[
[50, 180,  98,   6,   5,  38,     150]],          /* Capital requirements a_ij */
[350],                                            /* Total budget b_i */
[[ 1,   1,   1,   1,   1,   1]],                  /* Logic constraint a_ij */
[ 4],                                             /* Logic requirements b_i */
[[ 1,   0,   0,   0,   0,        0]],
[ 1],
[ 1,   1,   1,   1,   1,   1],                    /* Objective function c_j */
Z),
```

Computation Time (seconds):	Prog3	Prog4
	0.05	0.05

```
/* n=12, m=3 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  64,  65,  68, 72]],
[750],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1]],
[ 9],
[[ 0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0]],
[ 1],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	5.45	0.21

```
/* n=15, m=3*/
data(X,
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  64,  65,  68,  72,  99,  99, 99]],
[850],
[[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]],
[10],
[[ 0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0]],
[ 2],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	51.7	2.58

```
/*  n=18, m=3*/
data(X,
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  64,  65,  68,  72, 102,  60,  97,  95,  90,132]],
[1050],
[[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]],
[12],
[[ 0,  1,  0,  0,  0,  1,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0]],
[ 2],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	513.2	16.45

```
/* n = 24, m=3*/
data(X,
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  64,  65,  68,  72,  49, 181, 100,  63,  35, 242,  88,  74,  62,  67,  70, 70]],
[1500],
[[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]],
[18],
[[ 0,  1,  0,  0,  1,  0,  0,  1,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]],
[4],
[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	----	51.3

```
/* n = 27 , m=3*/
data(X,
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[
[ 50, 180, 98, 65, 38, 240, 90, 75, 64, 65, 68, 72, 50, 180, 98, 65, 38, 240, 90, 75, 64, 65, 68, 72, 75, 64, 65]],
[1650],
[[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]],
[21],
[[ 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]],
[4],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	----	120.7

```
/* n = 30  , m=3*/
data(X,
[ 0,  0,  0,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0,  0],
[ 1,  1,  1,  1,  1,  1,  1, 1, 1, 1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  1,  1],
[
[50,180, 98, 65, 38,240, 90,75,64,65,68,72,50,180,98,65,38,240,90,75,64,65,68,72,75,64,65,60, 70,80]],
[1720],
[[1,  1,  1,  1,  1,  1,  1, 1, 1, 1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  1,  1]],
[23],
[[0,  1,  0,  0,  0,  0,  0, 0, 0, 0, 0, 0, 0,  1, 0, 0, 0,  0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,  0,  1]],
[5],
[1,  1,  1,  1,  1,  1,  1, 1, 1, 1, 1, 1, 1,  1, 1, 1, 1,  1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  1,  1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	----	285.0

The following datasets are used to test the effect of a changing feasibility space f, measured by the constraining factor (CF) for a fixed matrix size.

```
data(X,
[  0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[  50,  180,   98,   65,   38,  240,   90,   75,   22,   18,   34,   26],
[1050, 890, 1450,  450,  275, 2100, 1400,  750,  102,   98,   99,  101],
[  50,  180,    0,    0,   38,  240,   90,   75,    0,    0,    0,    0],
[1050, 890,    0,    0,  275, 2100, 1400,  750,    0,    0,    0,    0],
[  15,   36,   19,   18,    7,   48,   21,   20,    4,    4,    7,    5]],
[750, 7680,  487, 5380,  207],
[[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    1,    0,    0,   -1,    0,    0,    0,    0,    0]],
[  11,   4,    0],
[[  0,    1,    0,    0,    0,    0,    0,    1,    0,    0,    0,    0],
[  1,    0,    0,   -1,    0,    0,    0,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    0,    1,    0,    0,    0,   -1,    0,    0],
[  1,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    1]],
[  1,    0,    0,    2],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	14.2	0.0

feasible points 0 (CF = 1)

```
data(X,
[  0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[ 50,  180,   98,   65,   38,  240,   90,   75,   22,   18,   34,   26],
[1050, 890, 1450,  450,  275, 2100, 1400,  750,  102,   98,   99,  101],
[ 50,  180,    0,    0,   38,  240,   90,   75,    0,    0,    0,    0],
[1050, 890,    0,    0,  275, 2100, 1400,  750,    0,    0,    0,    0],
[ 15,   36,   19,   18,    7,   48,   21,   20,    4,    4,    7,    5]],
[850, 7680,  547, 5380,  207],
[[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    0,    0,    1,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    1,    0,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    1,    0,    0,   -1,    0,    0,    0,    0,    0]],
[  8,    4,    0,    0,    0],
[[  1,    1,    1,    1,    1,    1,    1,    1,   -1,   -1,   -1,   -1],
[  1,    1,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0]],
[  2,    1],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	16.4	0.75

feasible points 41 (CF = 0.99)

```
data(X,
[      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[      50,   180,  98,   65,   38,   240,  90,   75,   22,   18,   34,   26],
[1050, 890,  1450, 450,  275,  2100, 1400, 750,  102,  98,   99,   101],
[      50,   180,  0,    0,    38,   240,  90,   75,   0,    0,    0,    0],
[1050, 890,  0,    0,    275,  2100, 1400, 750,  0,    0,    0,    0],
[      15,   36,   19,   18,   7,    48,   21,   20,   4,    4,    7,    5]],
[850,  7680, 547,  5380, 207],
[[     1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[      1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[      0,    0,    0,    0,    0,    1,    -1,   0,    0,    0,    0,    0],
[      0,    0,    0,    0,    1,    0,    -1,   0,    0,    0,    0,    0],
[      0,    0,    0,    1,    0,    0,    -1,   0,    0,    0,    0,    0]],
[      6,    3,    0,    0,    0],
[[     1,    1,    1,    1,    1,    1,    1,    1,    -1,   -1,   -1,   -1],
[      1,    1,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0]],
[      2,    1],
[      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	20.0	1.8

feasible points 165 (CF = 0.96)

```
data(X,
[      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[      50,   180,  98,   65,   38,   240,  90,   75,   22,   18,   34,   26],
[1050, 890,  1450, 450,  275,  2100, 1400, 750,  102,  98,   99,   101],
[      50,   180,  0,    0,    38,   240,  90,   75,   0,    0,    0,    0],
[1050, 890,  0,    0,    275,  2100, 1400, 750,  0,    0,    0,    0],
[      15,   36,   19,   18,   7,    48,   21,   20,   4,    4,    7,    5]],
[850,  7680, 547,  5380, 207],
[[     1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[      1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[      0,    0,    0,    0,    0,    1,    -1,   0,    0,    0,    0,    0],
[      0,    0,    0,    0,    1,    0,    -1,   0,    0,    0,    0,    0],
[      0,    0,    0,    1,    0,    0,    -1,   0,    0,    0,    0,    0]],
[      6,    3,    0,    0,    0],
[[     0,    0,    0,    0,    0,    0,    0,    1,    1,    0,    0,    0],
[      1,    1,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0]],
[      1,    1],
[      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	20.1	1.8

feasible points 204 (CF =0.95)

```
data(X,
[  0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[  50,  180,   98,   65,   38,  240,   90,   75,   22,   18,   34,   26],
[1050, 890, 1450,  450,  275, 2100, 1400,  750,  102,   98,   99,  101],
[  50,  180,    0,    0,   38,  240,   90,   75,    0,    0,    0,    0],
[1050, 890,    0,    0,  275, 2100, 1400,  750,    0,    0,    0,    0],
[  15,   36,   19,   18,    7,   48,   21,   20,    4,    4,    7,    5]],
[850, 7680,  547, 5380,  207],
[[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    0,    0,    1,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    1,    0,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    1,    0,    0,   -1,    0,    0,    0,    0,    0]],
[  1,    1,    0,    0,    0],
[[  0,    0,    0,    0,    0,    0,    0,    1,    1,    0,    0,    0],
[  1,    1,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0]],
[  1,    1],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	24.1	2.7

feasible points 534 (CF = 0.87)

```
data(X,
[  0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[  50,  180,   98,   65,   38,  240,   90,   75,   22,   18,   34,   26],
[1050, 890, 1450,  450,  275, 2100, 1400,  750,  102,   98,   99,  101],
[  50,  180,    0,    0,   38,  240,   90,   75,    0,    0,    0,    0],
[1050, 890,    0,    0,  275, 2100, 1400,  750,    0,    0,    0,    0],
[  15,   36,   19,   18,    7,   48,   21,   20,    4,    4,    7,    5],
[  40,    0,   90,    0,   30,    0,    0,   77,   21,   80,   44,   78]],
[950, 7680,  647, 5380,  207,  550],
[[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    0,    0,    1,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    1,    0,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    1,    0,    0,   -1,    0,    0,    0,    0,    0],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1]],
[  1,    1,    0,    0,    0,    1],
[ ],[    ],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	29.1	10.2

feasible points 2172 (CF = 0.47)

```
data(X,
[  0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[  50,  180,   98,   65,   38,  240,   90,   75,   22,   18,   34,   26],
[1050, 890, 1450,  450,  275, 2100, 1400,  750,  102,   98,   99,  101],
[  50,  180,    0,    0,   38,  240,   90,   75,    0,    0,    0,    0],
[1050, 890,    0,    0,  275, 2100, 1400,  750,    0,    0,    0,    0],
[  40,    0,   90,    0,   30,    0,    0,   77,   21,   80,   44,   78],
[  15,   36,   19,   18,    7,   48,   21,   20,    4,    4,    7,    5]],
[950, 9680,  747, 7380,  550,  507],
[[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    0,    0,    1,    0,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    1,    0,    0,    0,    0,    0,    0,    0],
[  0,    0,    0,    1,    0,    0,   -1,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    0,    1,    0,    0,    0,    0,    0,    0]],
[  1,    1,    0,    0,    0,    0],
[  ],[    ],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	35.0	14.0

feasible points 3072 (CF = 0.25)

```
data(X,
[  0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[
[ 50,  180,   98,   65,   38,  240,   90,   75,   22,   18,   34,   26],
[1050, 890, 1450,  450,  275, 2100, 1400,  750,  102,   98,   99,  101],
[ 50,  180,    0,    0,   38,  240,   90,   75,    0,    0,    0,    0],
[1050, 890,    0,    0,  275, 2100, 1400,  750,    0,    0,    0,    0],
[ 40,    0,   90,    0,   30,    0,    0,   77,   21,   80,   44,   78],
[ 15,   36,   19,   18,    7,   48,   21,   20,    4,    4,    7,    5]],
[950, 9680,  747, 7380,  550,  507],
[[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    0,    0,    1,    0,    0,    0,    0,    0,    0],
[  0,    0,    0,    0,    1,    0,    0,    0,    0,    0,    0,    0],
[  1,    0,    1,    1,    0,    1,    1,    0,    1,    1,    0,    1],
[  0,    0,    0,    1,    0,    0,    0,    0,    0,    0,    0,    0]],
[  1,    1,    0,    0,    1,    0],
[ ],[    ],
[  1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	38.2	18.9

feasible points 4,096 (CF = 0)

The following datasets are used to test the effect of changing the number constraints for a fixed variable problem; the feasible space (CF) was maintained at a constant value; # feasible coordinates approximately equal to 246, maintaining a CF of 0.94.

```
/* m=3 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  22,  18,  34, 26]],
[850],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1]],
[ 8],
[[ 1,   1,   1,   1,   1,   1,   1,   1,  -1,  -1,  -1,  -1]],
[ 2],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	7.15	1.65

```
/* m=6 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  22,  18,  34,  26],
[15,  18,  18,  15,  38,  24,  90,   7,  12,  18,  14, 16]],
[850,  433],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   1,   1]],
[ 8,   7],
[[ 1,   1,   1,   1,   1,   1,   1,   1,  -1,  -1,  -1,  -1],
[-1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,   1,   1,   1,   1]],
[ 2,  -2],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	10.8	2.7

```
/* m=9 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,  1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  22,  18,  34, 26],
[15,  18,  18,  15,  38,  24,  90,   7,  12,  18,  14, 16],
[25,  28,  28,  25,  28,  34,  45,  17,  22,  17,  34,  6]],
[850,433,520],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,  1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   1,  1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,  1],
[ 1,   0,   1,   1,   0,   1,   1,   0,   1,   1,   0,  1]],
[ 8,   7,   6,   2],
[[ 1,   1,   1,   1,   1,   1,   1,   1,  -1,  -1,  -1, -1],
[-1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,   1,   1,   1,  1]],
[ 2,  -2],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,  1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	14.4	3.3

```
/* m=12 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[
[ 50, 180,  98,  65,  38, 240,  90,  75,  22,  18,  34,  26],
[ 15,  18,  18,  15,  38,  24,  90,   7,  12,  18,  14,  16],
[ 25,  28,  28,  25,  28,  34,  45,  17,  22,  17,  34,   6],
[ 11,  31,  16,  15,  38,  24,  69,  45,  22,  17,  14,  19]],
[850,433, 520,390],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1],
[ 0,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 0,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1]],
[ 8,   7,   6,   5,   7,   2],
[[ 1,   1,   1,   1,   1,   1,   1,   1,  -1,  -1,  -1,  -1],
[ -1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,   1,   1,   1,   1]],
[ 2,  -2],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	18.5	4.1

```
/* m=15 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  22,  18,  34,  26],
[15,  18,  18,  15,  38,  24,  90,   7,  12,  18,  14,  16],
[25,  28,  28,  25,  28,  34,  45,  17,  22,  17,  34,   6],
[11,  31,  16,  15,  38,  24,  69,  45,  22,  17,  14,  19],
[50, 130,  87,  66,  49, 117,  88,  75,  42,  68,  34,  36]],
[850,433, 520, 390,940],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1],
[ 0,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 0,   1,   1,   1,   0,   1,   0,   1,   1,   1,   0,   1],
[ 0,   1,   1,   1,   0,   1,   0,   1,   0,   1,   0,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   1,   1]],
[ 8,   7,   6,   5,   7,   4,   3,   7],
[[ 1,   1,   1,   1,   1,   1,   1,   1,  -1,  -1,  -1,  -1],
[-1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,   1,   1,   1,   1]],
[ 2,  -2],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	22.4	4.9

```
/* m=18 */
data(X,
[ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[
[50, 180,  98,  65,  38, 240,  90,  75,  22,  18,  34,  26],
[15,  18,  18,  15,  38,  24,  90,   7,  12,  18,  14,  16],
[25,  28,  28,  25,  28,  34,  45,  17,  22,  17,  34,   6],
[11,  31,  16,  15,  38,  24,  69,  45,  22,  17,  14,  19],
[50, 130,  87,  66,  49, 117,  88,  75,  42,  68,  34,  36],
[33,  19,  43,  75,   8,  24,  35,  27,  12,  37,  14,  16]],
[850,433, 520, 390, 940,510],
[[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   1,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1],
[ 0,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
[ 0,   1,   1,   1,   0,   1,   0,   1,   1,   1,   0,   1],
[ 1,   0,   1,   0,   1,   0,   1,   1,   1,   0,   1,   1],
[ 0,   1,   1,   1,   0,   1,   0,   1,   0,   1,   0,   1],
[ 0,   0,   1,   1,   0,   1,   0,   1,   0,   1,   0,   1],
[ 1,   1,   1,   1,   0,   1,   1,   1,   1,   1,   0,   1]],
[ 8,   7,   6,   5,   7,   4,   4,   3,   2,   6],
[[ 1,   1,   1,   1,   1,   1,   1,   1,  -1,  -1,  -1,  -1],
[-1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,   1,   1,   1,   1]],
[ 2,  -2],
[ 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
Z),
```

Computation Time (seconds):	Prog3	Prog4
	26.3	5.65

The following datasets were used to run the CLP($\Re$) experiments.

7.
25.
4. 7. 0.2. 45. 12. 1. 0.75. 0.48. 0.51. 0.51. 0.62. 0.35. 0.27. 0.18. 0.35. 0.78. 1.2. 7.2. 0.01. 5.9. 2.2. 0.59. 0.8. 0.9. 1.2.
90. 5. 20. 5. 5. 20. 10. 15. 10. 7. 4. 5. 12. 4. 3. 40. 5. 10. 10. 9. 11. 8. 10. 4. 2.
300. 8. 20. 15. 10. 20. 20. 40. 35. 14. 12. 7. 20. 15. 23. 200. 9. 18. 14. 12. 14. 22. 14. 17. 19.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 585.
0. 1. 1. 0. 1. 0. 1. 0. 0. 1. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 305.
1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 0. 239.
8. 0.3. 2. 5. 6.7. 0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 65. 0.20. 1. 10. 0.9. 30. 0.5. 0.8. 0.9. 0.1. 5. 9145.
2. 9. 3.4. 66. 4. 0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 7. 0. 1. 17. 0.5. 10. 1. 1. 0. 1. 0. 2134.
1. 1.1. 3. 81. 2. 0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 65. 1. 0. 1. 0. 11. 20. 1.7. 1. 1. 1. 1305.
2. 4. 6. 0.1. 13. 0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 65. 0. 1. 0. 0.3. 1.2. 30. 10. 1. 1. 1. 3234.

7.
20.
1. 0.75. 0.48. 0.51. 0.51. 0.62. 0.35. 0.27. 0.18. 0.35. 0.78. 1.2. 7.2. 0.01. 5.9. 2.2. 0.59. 0.8. 0.9. 1.2.
90. 5. 20. 5. 5. 20. 10. 15. 10. 7. 40. 5. 10. 10. 9. 11. 8. 10. 4. 2.
300. 8. 20. 15. 10. 20. 20. 40. 35. 14. 200. 9. 18. 14. 12. 14. 22. 14. 17. 19.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 585.
0. 1. 0. 0. 1. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 305.
0. 1. 1. 0. 0. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 0. 239.
0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 65. 0.20. 1. 10. 0.9. 30. 0.5. 0.8. 0.9. 0.1. 5. 9145.
0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 7. 0. 1. 17. 0.5. 10. 1. 1. 0. 1. 0. 2134.
0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 65. 1. 0. 1. 0. 11. 20. 1.7. 1. 1. 1. 1305.
0.2. 0. 0. 0. 0. 0. 0.3. 0.3. 0.1. 65. 0. 1. 0. 0.3. 1.2. 30. 10. 1. 1. 1. 3234.

7.
15.
0.62. 0.35. 0.27. 0.18. 0.35. 0.78. 1.2. 7.2. 0.01. 5.9. 2.2. 0.59. 0.8. 0.9. 1.2.
20. 10. 15. 10. 7. 40. 5. 10. 10. 9. 11. 8. 10. 4. 2.
20. 20. 40. 35. 14. 200. 9. 18. 14. 12. 14. 22. 14. 17. 19.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 585.
0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 305.
1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 0. 239.
0. 0.3. 0.3. 0.1. 65. 0.20. 1. 10. 0.9. 30. 0.5. 0.8. 0.9. 0.1. 5. 9145.
0. 0.3. 0.3. 0.1. 7. 0. 1. 17. 0.5. 10. 1. 1. 0. 1. 0. 2134.
0. 0.3. 0.3. 0.1. 65. 1. 0. 1. 0. 11. 20. 1.7. 1. 1. 1. 1305.
0. 0.3. 0.3. 0.1. 65. 0. 1. 0. 0.3. 1.2. 30. 10. 1. 1. 1. 3234.

7.
10.
0.78. 1.2. 7.2. 0.01. 5.9. 2.2. 0.59. 0.8. 0.9. 1.2.
90. 5. 20. 5. 5. 20. 10. 15. 10. 7.
300. 8. 20. 15. 10. 20. 20. 40. 35. 14.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 585.
1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 305.
0. 1. 1. 0. 0. 1. 1. 0. 0. 0. 239.
0.20. 1. 10. 0.9. 30. 0.5. 0.8. 0.9. 0.1. 5. 9145.
0. 1. 17. 0.5. 10. 1. 1. 0. 1. 0. 2134.
1. 0. 1. 0. 11. 20. 1.7. 1. 1. 1. 1305.
0. 1. 0. 0.3. 1.2. 30. 10. 1. 1. 1. 3234.

7.
5.
2.2. 0.59. 0.8. 0.9. 1.2.
20. 10. 15. 10. 7.
20. 20. 40. 35. 14.
1. 1. 1. 1. 1. 585.
0. 1. 1. 1. 1. 305.
1. 1. 0. 0. 0. 239.
0.5. 0.8. 0.9. 0.1. 5. 9145.
1. 1. 0. 1. 0. 2134.
20. 1.7. 1. 1. 1. 1305.
30. 10. 1. 1. 1. 3234.

Bibliography

Al-Zobaidie, A., J.B. Grimson, " Expert Systems and Database Systems: How Can They Serve Each Other?" *Expert Systems*, February 1987.

Avignon, *7th international workshop: Expert Systems and their Applications*, Gerfau, Paris 1987.

Avis, D., and V. Chvátal. "Notes on Bland's pivoting rule," *Mathematical Programming Study*, vol. 8, pp. 24-34, 1978.

Balas, Egon, "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," *Operations Research 13*, No. 4, 1965, pp. 517-549.

Bartee, Thomas C., *Expert Systems and Artificial Intelligence: Applications and Management*, Howard W. Sams & Company, 1988.

Baumol, W. and Quandt, R., " Investment and discount rates under capital rationing - A programming approach, *Economic Journal*, 1965.

Bell Canada University Liaison grant *"Expert Systems for Resource Allocation"*, University of Ottawa, 1991.

Bielawski, Larry: Robert Lewand, *Expert Systems development: building PC-based applications*, Wellesley, Mass.: QED information Sciences, 1988.

Bierman, H. Jr, *Implementing Capital Budgeting Techniques*, Ballinger Publishing Company, 1988.

Bonczek, R.H., C.W Holsapple, and A.B Whinston, *Foundations of Decisions Support Systems*, New York: Academic Press, 1981.

Brans J. P and Ph. Vincke: "A preference ranking organisation method. The PROMETHEE method for MCDM", *Management Science*, vol. 31, no. 6, 1985, pp. 647-656.

Bromwich, M, *The Economics of Capital Budgeting*, Pittman Publishing Limited, 1979

Brown, R.G, J.W Chinneck, G.M Karam. "Optimization with constraint programming systems", *Publications in Operations Research series*, no. 9 1989.

Buchanan, B.G. and Shortliffe, E.H. (1984) *Rule Based Expert Systems. The Mycin Experiment of the Stanford Heuristic Programming Project*, Addison-Wesley.

Buchanan, B.G. (1986) Expert Systems: "Working systems and the research literature", *Expert Systems*, vol. 3, No. 1.

Cafolla, Ralph and Albert D. Kauffman, *Turbo Prolog: Step by Step*, Merrill publishing company, Columbus, Ohio, 1989.

Clark J. J, *Capital Budgeting*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey ,1984

Clocksin, W.F. and C.S. Mellish, *Programming in PROLOG*. New York: Springer, 1981.

Cohen, Jacques, "Constraint Logic Programming Languages," *Comm. A.C.M.*, vol.33, no. 7, pp. 52-68, July 1990.

Colmerauer, A., *Prolog and Infinite Trees*, Groupe Intelligence Artificielle, Faculte des Science de Luminy, Université Aix-Marseilles II, 1982.

Cuena, J., and A. Garcia-Serrano, "An Expert System for financial risk evaluation and decision making" *7th International Workshop: Expert Systems & Their Applications, Avignon 87*, Gerfau, Paris, 1987.

Dungan, Chris W. and Chandlers, John S., "AUDITOR: a micro-computer-based expert system to support auditors in the field," *Expert systems journal*, october 1985, pp 210-226.

Dantzig, G. B., *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

Dean, J., *Managerial Economics*. Englewood Cliffs, N. J., Practice-Hall, 1951.

Dean, J. *Capital Budgeting*. New York: Columbia University Press, 1951.

Duda, R.O., *Knowledge-Based Expert Systems Come of Age*, Byte, 6 (9), 1981, pp. 238-281.

Edwards, A., N.A.D. Connel, *Expert Systems in Accounting*, Hertfordshire: Prentice-Hall international, 1989.

Expert Systems. 1985. Financial Expert Systems from Apex. *Expert Systems*, vol. 2, No. 4.

EXSYS, "Expert Systems Software", Exsys, Inc., P.O Box 11247, Albuquerque, NM, 1991.

Farguhar, P. H, "A survey of Multiattribute Utility Theory and Applications", *Studies in the Management Sciences*, vol 6, North-Holland Publishing Amsterdam, pp59-89, 1977.

Farragher, E. "Capital Budgeting Practice of non-industrial Firms." *The Engineering Economist* 31, 1986, pp. 293-302.

Feigenbaum, E.A., *Knowledge Engineering for the 1980's*, Computer science dept., Stanford University, 1982.

Fogler, H. R., "Overkill in Capital Budgeting Technique?" *Financial Management 1*, No 1, 1972, pp. 52-62.

Forsyth, J. D. and D. C. Owen, Capital Rationing Methods, *Capital Budgeting Under Conditions of Uncertainty*, Martinus Nijhoff Publishing, 1981.

Fourier, J-B. J., *Solution d'une question particulière du calcul des inégalités*, Oeuvres II pp. 317-328, 1826.

Freuder, E.C., "Synthesizing Constraint Expressions" *Communications of ACM*, No 21, pp. 958-966, 1978.

Garey, M. R., and D. S. Johnson *Computers and Intractibility: A Guide to the thoery of NP-Completeness*, Freeman, 1979.

Garfinkel, R. S., and G.L Nemhauser, *The Set Partitioning Problem: Set Covering with Equality Constraints*, Opns. Res. 17, pp. 848-856, 1969.

Garfinkel, R. S. and G.L Nemhauser, *Integer Programming*, John Wiley & sons, New York, 1972.

Geoffrion A. M, J. S, Dyer and A. Feinberg, "An Interactive Approach for Multiple Optimization with Applications to the Operation of an Academic Department", *Management Science*, vol 19, no 4, 1972.

Gevarter, William B., *An overview of Expert Systems*, National Bureau of Standards Report #NBSIR 82-2505, May 1982.

Giannesini, Francis, Henry Kanoui, Robert Pasero, Michel Van Caneghem (Foreword by Alain Colmerauer), *Prolog*, Groupe Intelligence Artificielle, Faculte des Science de Luminy, Aux-Marseille II, 13288 Marseille cedex 2, France, Addison-Wesley Publishing Company, 1986.

Goldfarb, D. and J. K. Reid, "A practical steepest-edge simplex algorithm," *Mathematical programming*, vol. 12, pp. 361-371, 1977.

Greenberg, H. J., *The Development of An Intelligent Mathematical Programming System*, Washington Operations Research/Management Science (WORMSC), 1987.

Guranani, C., "Capital Budgeting: Theory and practice" *The Engineering Economist* 30, 1984, pp. 19-46.

Hamidi, B. and R. Tremolieres, *Intelligence Artificielle et Combinatoire: Comment Faire de la combinatoire sous Prolog?*, Technical Report, Institut d'Administration des Entreprises, Université d'Aix-Marseille III, France, 1989.

Haralick, R.M. and G.L. Elliot, "Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14: pp. 263-313, 1980.

Harmon, Paul, David King, *Expert Systems: AI in business*, New York: J. Wiley, 1985.

Harmon, P., Rex Maus, William Morrissey, *Expert Systems: tools and applications*, New York: Wiley, 1988.

Hayes-Roth, F., "Knowledge-based Expert Systems." *Computer*, October 1984.

Hayes-Roth, F., D. A. Waterman and D. B. Lenat, *Building Expert Systems*, Addison-Wesley Publishing Co., Inc., Massachusetts, 1983.

Ho, J. K. and E. Loute, "Computational experience with advanced implementation of decomposition algorithms for linear programming," *Mathematical programming*, vol. 27, pp. 283-290, 1983.

Hoffman, A., M. Mannos, D. Sokoloswky and N. Wigmann, *Computational Experience in Solving Linear Programs."* SIAM J., vol. 1, pp. 1-33, 1953.

Holsapple, Cyde W., Andrew B. Whinston, *Business Expert Systems*, Irwin,Inc.,1987.

International Workshop Expert Systems & Their Application (1987: Avignon, France), *Expert Systems and their application: 7th International Workshop, Avignon, France, May 13, 14 and 15, 1987*, Paris: Imprime par Gerfau, 1987.

Jaap Spronk, "Interactive Multiple Goal Programming as an Aid for Capital Budgeting and Financial Planning with Multiple Goals", *Capital Budgeting Under Condisions of Uncertainty*, Martinus Nijhoff Publishing, 1981.

Jack, W., "An interactive graphical approach to linear financial methods", *J. opl Res. soc.* 36.5, 1985.

Jaffar, J. and Lassez, J-L. "Constraint Logic Programming, In *Proceedings of the fourth ACM Symposium of the principles of Programming Languages*, (Munich, 1987), pp 111-119.

Jaffar, J. Spiro Michaylov, Peter Stuckey and Roland H.C Yap, *Research Report: The CLR(R) Language and System*, IBM Research Division, Yorktown Heights, NY 10598, 1990.

Johnson, L., E.T. Keravnou, *Expert Systems technology: a guide*, Turnbridge Wells: Abacus, 1985.

Karmarkar, N., "A new polynomial-time Algorithm for Linear Programming," *Combinatorica*, vol. 4, no. 4, 1984.

Klahr, Philip: Donald A. Waterman, *Expert Systems: techniques, tools, and applications*, Reading, Mass.; Donald mills, Ont.: Addison-Wesley pub. Co., 1986.

Kowalski, R.A, *Predicate Logic as Programming Language*, Amsterdam: North-Holland, 1977.

Kowalski R., *Logic for Problem Solving*, New York: Elsevier North Holland, 1979.

Lassez, C., Constraint Logic Programming, *Byte Magazine*, Aug 1987, pp. 171-176.

Lassez, C., McAloon, K. and Yap, R. "Constraint Logic Programming and Options Trading", *IEEE Expert 2(3)*, Special Issue on Financial Software, 1987, pp. 42-45.

Lassez, J-L, and M. J. Maher, *On Fourier's Algorithm for Linear Arithmetic Constraints*, IBM Research Report, T. J. Watson Research Center, 1988.

Lehner, P. E., S. W. Barth. "Expert Systems on Microcomputers." *Expert Systems*, October 1985.

Leibowitz, J., "Useful Approach for Evaluating Expert Systems." *Expert Systems*, April 1986.

Leibowitz, Jay, *Expert Systems for business and management*, Englewood Cliffs, NJ: Prentice Hall, 1990.

Leler, Wm., *Constraint programming Languages: Their Specification and Generation*, Addison-Wesley, Reading, Mass.

Little, J.D.C., K.G. Murty, D.W. Sweeney, and C. Karel, " An Algorithm for the travelling salesman problem", *Opns. Res. 11*, pp. 979-989, 1963.

Lorie, J. and L. Savage, "Three Problems in Capital Rationing", *Journals of Business*, Oct 1955, Vol.28, pp. 229-239.

MacCrimmon K. R, *An Overview of Multiple Objective Decision Making in MCDM*, Cochrane, James C, and Milen Zeleny (ebc), University of South Carolina press, Columbia s.c. 1973.

Martello, S. and P. Toth "Algorithms for Knapsack problems", *Surveys in Combinatorial Optimization*, 1987.

Martin, J. and S. Oxman, *Building Expert Systems*, Prentice Hall, Englewood Cliffs, NJ, 1988.

Mateus, G. R. and C. L. Bornstein, *Dominance Criteria for the Capacitated Warehouse Location Problem*, Journal of the Operational Research Society, Vol. 42, No. 2, pp. 145-149, 1991.

Mettrey, W., "An Assesment of Tools for Building Large Knowledge-based Systems." *AI Magazine*, Winter 1987.

Mini Euroconference [2nd: 1985: Lunten, Netherlands], *Expert Systems and AI in DSS: Proceedings of the second Mini Euroconference, Lunten, The Netherlands, 17-20 Nov, 1985*, Kluwer Academic Publishers, 1987.

Mockler, Robert J., *Knowledge-Based systems for Management decisions*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988. pp.23

Murray, R. and A. Kester, "A presolver for the AT&T KORBX Mathematical Programming system" *TIMS/ORSA Annual Conference*, Denver, CO 1988.

Myers, S. C., "Interactions for Corporate Financing and Investment Decisions-Implications for Capital Budgeting," *Journal of Finance* 29 (March 1974), pp. 1-25.

Naish, L., *Negation and Control in Prolog*. PhD thesis, University of Melbourne, Australia, 1985.

Naslund, B., "A Model of Capital Budgeting under Risk" *Journal of Business* 39 (1986) pp. 257-271.

Negotia, C.V., *Expert Systems and Fuzzy Systems*. Menlo Park, CA: Benjamin/Cummings, 1985.

Newell, A., and Simon, H. A., *Human Problem Solving*, Englewood Cliffs, NJ: Prentice-Hall, 1972.

Newquist, H., American Express and AI: Don't leave home without them. *AI Expert* 2, no. 4 pp. 63-65, April, 1987.

Nijcamp, P. and Spronk, J., Interactive Multiple Goal Programming, *Report7803/A, Centre for Research in Business Economics, Erasmus University, Rotterdam*, 1978.

Nijcamp, P. and Spronk, J., "Goal Programming for Decision Making", *Ricerca Operativa*, special issue on Multi Criteria Decision Making, 1978.

Nilsson, N. J., *Principles of Artificial Intelligence*, Morgan Kaufman, San Mateo, CA, 1980.

O'Keefe, Richard A., *The Craft of Prolog*, The MIT Press, Cambridge, Massachusetts, 1990.

Pinches, G. "Myopia, Capital Budgeting and Decision Making" *Financial Management* 1982, pp. 6-19.

Pissarides, S. "*Capital Budgeting: the Mathematical Programming Approach*" Master's thesis, Systems Science Program, University of Ottawa, in preparation.

Richard F. Deenro, R. W. Spahr and J. E. Herbert, "Preference Trade-offs in capital Budgeting Decisions", *IIE Transactions* vol. 17 no.4, 1985, pp. 332-337.

Robinson, J. A., "A machine oriented logic based on the resolution principle," *Journal of the Association for Computing Machinery*, Vol. 12, pp. 23-41, 1965.

Robinson, J. A., *Logic: Form and Function*, North Holland, New York, 1979.

Rolston, David W., *Principles of artificial intelligence and expert systems development*, McGraw-Hill, Inc. 1988.

Saaty, T. L, *The Analytic Hierchy Process*, McGraw-Hill, 1980.

Salo, A. and Hamalainen, R. P, "A modelling and decision aid for supporting telecomunications investments", *1989 IEEE international conference on the systems, man and cybernetics*, conference proceeding p.115-18 vol 1.

Savory, Stuart E., Barbara Chapman, *Expert Systems in the organization: an introduction for decision-makers*, Halsted Press, 1988.

Senicourt, P., "Preface™-Expert: Progiciel D'expertise Financiere et D'analyse pour la creation d'entreprises" *7th International Workshop: Expert Systems & Their Applications, Avignon 87*, Gerfau, Paris 1987.

Shambin, J.E., and G.T. Stevens Jr. *Operations Research: A Fundamental Approach.* New York: McGraw-Hill, 1974.

Silverman, G.Barry, *Expert Systems for business*, Reading, Mass.: Addison-Wesley, 1987.

Srinivasan, V. and A. D, Shocker, "Linear Programming Techniques for Multidimensional Analysis of Preferences", *Phychometrica*, vol 38. pp 337-369, 1973.

Sterling, L. and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, Cambridge, Mass, 1986.

Tanimoto, S. L., *The Elements of Artificial Intelligence: an introduction using LISP*, Computer Science Press, Maryland, 1987.

Turban, E., *Decision Support and Expert Systems*, MacMillan, 1990.

Turban, E., *Integrating Expert Systems and Mathematical Programming*, TIMS Conference XXX in Rio de Janeiro, August 1991.

Tversky Amos and D. Kahneman , "Judgement Under Uncertainty: Heuristics and Biases", *Science*, vol. 185, pp. 1124-1131, 1974.

Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, MIT press, 1989.

Vasant Dhar, Nicky Ranganathan, "Integer programming vs. Expert Systems: An Experimental Comparison" *Comm. A.C.M.*, vol. 33. no. 3, pp. 323-336, Mar. 1990.

Vasant Dhar, ALbert Crocker, "Knowledge-Based Decision Support in Business: Issues and a Solution" *IEEE Expert*, Spring 1988, pages 53-62.

VP-Expert, *Rule-Based Expert System Development Tool*, Paperback Software, 1989.

Walker, A., M. McCord, J. F. Sowa, and W. Wilson, *Knowledge Systems and Prolog*, Addison-Wesley Publishing Company, Inc., 1987.

Walker, T. C. and R. K. Miller, *Expert systems handbook: an assessment of technology and applications*, Fairmont Press, Lilburn, GA, 1990.

Weingartner H. M., *Mathematical Programming and the analysis of the capital budgeting problems*, Prentice-Hall, 1963.

Weingartner H. M., "Capital Budgeting of Interrelated Projects: Survey and synthesis," *Management Science* 12 (1966), pp 213-244.

Weingartner H. M., "Capital Rationing: n Authors in Search of a Plot," *Journal of Finance* 32 (1977), pp 1403-1431.

Wilkes, F. M , *Capital Budgeting Techniques*, Wiley (1983).

Wolfe, P., and L. Cutler, "Experiments in Linear Programming," *Recent Advances in Mathematical Programming*, McGraw Hill, New York, pp. 177-200, 1963.

Zadeh, L.A., "Fuzzy Logic," *IEEE Computer*, April 1988, pages 83-93.

Zadeh, L.A., *The Management of uncertainty in Expert Systems*. Proceedings, First Symposium on Application of Expert Systems in Emergency management Operation, FEMA/NBS, Washington, D.C., April 1985.

Zeleny, M : *Linear Multiobjective Programming*, Ph.D. thesis, University of Rochester, New York, 1972.