Faculty of Mathematical Sciences

University of Twente

University for Technical and Social Sciences

P.O. Box 217
7500 AE Enschede
The Netherlands
Phone: +31-53-4893400
Fax: +31-53-4893114
Email: memo@math.utwente.nl

Memorandum No. 1480

A branch-and-bound methodology within algebraic modelling systems

J.J. Bisschop, J.B.J. Heerink[1] and
G. Kloosterman

December 1998

ISSN 0169-2690

[1]Paragon Decision Technology, Haarlem, The Netherlands

A Branch-and-Bound Methodology within Algebraic Modeling Systems

J.J. Bisschop, J.B.J. Heerink and G.J. Kloosterman

Faculty of Mathematical Sciences
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

December 18, 1998

Abstract

Through the use of application-specific branch-and-bound directives it is possible to find solutions to combinatorial models that would otherwise be difficult or impossible to find by just using generic branch-and-bound techniques within the framework of mathematical programing. MINTO is an example of a system which offers the possibility to incorporate user-provided directives (written in C) to guide the branch-and-bound search. Its main focus, however, remains on mathematical programming models. The aim of this paper is to present a branch-and-bound methodology for particular combinatorial structures to be embedded inside an algebraic modeling language. One advantage is the increased scope of application. Another advantage is that directives are easier implemented at the modeling level than at the programming level.

KEYWORDS: Modeling Languages, Combinatorial Optimization, Discrete Optimization, Implicit Enumeration, Branch-and-Bound

MATHEMATICAL SUBJECT CLASSIFICATION: 68N20, 68N99

1 Introduction

Branch-and-bound methods are well-known enumeration schemes for solving combinatorial optimization models and mixed-integer programming models [11, 12]. These methods employ a search tree. Along each branch in this tree typically one or more decision variables are either fixed or further constrained. At each node in the tree there is a submodel to be solved in terms of all variables not yet fixed along the path from the node to the root of the search tree.

When the overall model to be solved is a linear program with binary and integer variables, then there are readily available solvers which can solve these

models quite efficiently. These solvers are based on a collection of techniques contained inside a generic branch-and-bound framework. There are models, however, for which these solvers are not applicable, either because the underlying model is not a mathematical program of the right type, or because the time required for the search is too exhorbitant. In these cases, there is a need for an alternative.

One such alternative is offered through the use of application-specific directives within a branch-and-bound framework. Through these directives it is possible i) to influence the shape of the search tree, ii) to select the order in which nodes are created, iii) to compute bounds which are not based on mathematical programming relaxations, and iv) to look for multiple (not necessarily optimal) solutions. As these directives are based on insight into the reality being modeled, they could be superior to generic search directives which are for the most part solely based on mathematical arguments. It is this last consideration that has inspired the development of a system such as MINTO [13, 14].

MINTO is a system geared at solving mixed integer linear programs, and relies heavily on its ability to solve linear programming relaxations at nodes in the search tree. It offers the possibility to replace default system procedures by new user-defined procedures to control the overall enumeration process. MINTO is without doubt a powerful system, but its demanding interface requires extensive programming skills in addition to the already required skills to design the appropriate search directives.

An alternative technology for the specification of application-specific directives within a branch-and-bound framework could be based on the use of modeling systems, such as AIMMS [2, 3], AMPL [9, 10] and GAMS [5, 6]. The modeling languages inside these systems were primarily designed for the specification of constraint-oriented models, but there are no inherent limitations to expand these languages to allow for the expression of branch-and-bound directives related to these models.

The aim of this paper is twofold. The first objective is to specify a conceptual branch-and-bound enumeration scheme in which system tasks are coupled to the execution of user-defined directives. The second objective is to present a proposal on how this conceptual solver can be employed within the framework of an algebraic modeling system. The algebraic modeling language AIMMS [3] is used for illustrative purposes, but any other language could be extended in a similar manner.

Some related work in the area of modeling languages has already been done by Bisschop and Fourer [4]. They explore a general variable subset enumeration approach, combined with directives to guide the search. Our proposal offers similar functionality, but has been extended to allow the modeler to set up an application-specific search tree.

A straightforward implementation of the modeling technology proposed in this paper will not compete in efficiency with a solver such as MINTO due to differences in data structures and execution principles. In first instance, only the ease and scope of specification should be seen as major advantages of using a modeling language approach. However, as compiled code generation enters into the arena of modeling systems, the efficiency argument in favor of C or FORTRAN-based technology may no longer remain valid.

The remainder of this paper is organized as follows: Section 2 gives a detailed overview of the branch-and-bound enumeration scheme thereby mak-

ing a distinction between user-defined routines and system-specific routines. Section 3 argues in favor of modeling languages when specifying branch-and-bound directives. Special emphasis is placed on the specification of the directives for particular set structures, namely the variable subset, the variable subtour and the variable partition. For each of these three set structures an example model plus selected directives are provided in Section 4, and have all been written in the AIMMS Modeling Language.

2 The Branch-and-Bound Enumeration Scheme

The principal idea behind each branch-and-bound enumeration scheme is to partition a problem into more restricted subproblems of smaller size. Each of these subproblems can then be reconsidered and partitioned again, until all subproblems have become easy to solve. The best solution encountered during this search is then also the best solution of the original problem. The major disadvantage of this approach is that the number of subproblems grows exponentially. However, not all subproblems need to be considered. Their number can be reduced by concluding infeasibility of a particular subproblem and all of its descendents. Another way to reduce the number of subproblems is to obtain a bound on the best attainable objective function value of the subproblem and all of its descendents. If this bound indicates that another solution found elsewhere in the tree is better, then the subproblem (together with its descendents) can be dropped.

The full specification of a branch-and-bound enumeration scheme is presented in the following three subsections. The first section describes a flowchart which captures the main enumeration algorithm. Two steps in this algorithm, namely the extensive processing of a current node and the generation of new nodes, are further elaborated in the subsequent two subsections.

2.1 Main Enumeration Algorithm

The implementation of an efficient enumeration algorithm in a programming language such as C or FORTRAN is a nontrivial task. There is quite an extensive bookkeeping task involved to keep track of the search tree (node creation, node deletion, parameters at node, etc). The proposal in this paper makes the assumption that the bookkeeping task is to be performed by a bookkeeping engine, and that only the specification of enumeration directives remains the responsibility of the person designing the algorithm.

Consider the flowchart in Figure 1. In this chart there are grey boxes, white boxes, striped boxes, solid one-way directed arrows and a dotted two-way arrow. The grey boxes describe actions to be implemented by the algorithm designer. The white boxes decribe actions to be performed by the bookkeeping engine. The striped boxes represent extensive tasks to be performed, for which further detail can be found in a separate flowchart elsewhere in this section. The solid arrows indicate the flow of the algorithm, while the dotted two-way arrow indicates a procedure call from within a grey box.

The meaning of each of the boxes is described as follows.

- The grey "Preprocess" box is provided to describe actions that may lead to

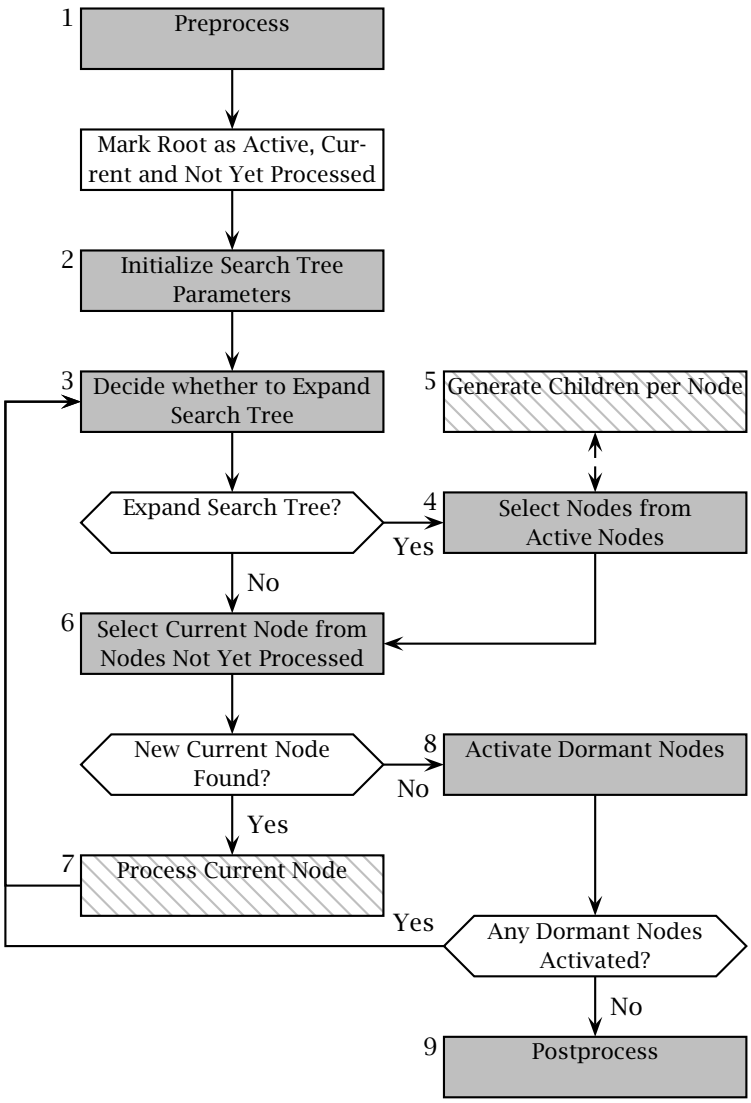

Figure 1: Flowchart of the Main Enumeration Algorithm

a tighter specification of the original problem or to the determination of method parameters. A typical action is to tighten bounds, lift constraints, or use a heuristic to find an incumbent solution and/or an incumbent objective function value to be used for subsequent subproblem elimination.

- The grey box with "Initialize Search Tree Parameters" is provided to set up and initialize (recursively defined) parameters to be used throughout the search tree.

- The white "Mark Root Node as Active, Current and Not Yet Processed" is a system action that begins as soon as the previous step has been completed. The bookkeeping engine is started, and the root node becomes Active, Current and Not Yet Processed. The Active status is maintained by the system until the node is no longer needed. A node looses its Active status once the subproblem at this node is infeasible, or does not pass the bound test when compared to the incumbent objective function value, or can no longer produce new subnodes (i.e. child nodes). The Not Yet Processed status is assigned by the system to every new subproblem in the search tree, and is changed into Processed as soon as the subproblem has been processed. The root node is also the initial value of the system parameter `BBCurrentNode`.

- The grey box entitled "Decide whether to Expand Search Tree" must contain the logic that controls the growth of the tree. It is possible to create new subproblems without first investigating the originating subproblems. It is also possible to reverse this process, and to investigate existing subproblems before creating additional new ones. The logic may depend on the current number of active nodes, the depth of existing active nodes or any parameter defined over nodes.

- The white "Expand Search Tree" box is a system action to check the logical value of the predefined system parameter `BBExpandSearchTree`, and act accordingly. The value of this parameter is set by the user in the previously described grey box.

- The grey box entitled "Select Nodes from Active Nodes" is a user procedure to let the search tree grow as directed. During the execution of this procedure one or more active nodes are selected in an iterative fashion, and for each selected node a call is made to the procedure `BBRegisterNewChild` to generate one or more new subproblems. Once the call is finished, the calling procedure continues its execution.

- The grey box with text "Select Current Node from Nodes Not Yet Processed" is a user-defined procedure that selects a single subproblem to be processed. Note that the Not Yet Processed status is completely under the control of the system. When a node is created, it automatically gets this status. It looses this status after the node has been processed.

- The white "New Current Node Found" box checks the value of the predefined element parameter `BBCurrentNode` (taken from the predefined set of `BBAllNodes`). When this parameter is empty, the system will still check whether there are any dormant nodes to be activated.

- The grey box with header "Activate Dormant Nodes" is a user-specified routine designed to activate nodes which were previously given the status of Dormant. This status was given because the node was about to be dropped as the result of a bound test, but either the computed bound or the incumbent value (or both) contained a cheating factor. Dropping would then not have been proper, thus a Dormant status (no longer Active) was assigned to temporarily remove these nodes from further consideration. In an iterative fashion all Dormant nodes are checked and, depending on a user-defined criterion, Dormant nodes can be made Active by calling the system procedure `BBWake`.

- The white (system) decision box to check whether there are "Any Dormant Nodes Activated" reacts to the result of the previous procedure Without any active or dormant nodes the search is definitely over, and possible postprocessing can begin. Otherwise, there is one or more active node to be considered for the generation of new subproblems.

- The grey "Postprocess" box allows the user to specify the computation of those parameters which depend on one or more of the incumbent solutions found during the search process. Once the postprocess step is completed, the entire search procedure is terminated.

2.2 Generate Children per Node

How to generate some or all of the children belonging to a selected active node is the most difficult task to be specified by the person providing search directives. As will be demonstrated in the next section, this task is somewhat simplified when the specification makes use of predefined modeling structures for subsets, ordered subsets, permutations, partitions, etc.

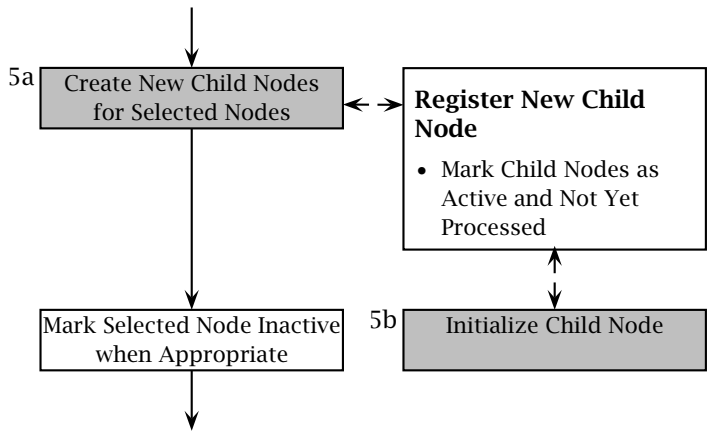

Figure 2: Flowchart of the Child Generation Algorithm

Consider the flowchart in Figure 2. The boxes in this chart can be described as follows.

- The grey box entitled "Create New Child Nodes for Selected Node" is a user-defined procedure to indicate how many and which child nodes are to be created for the selected node at hand. Each new child is named using the predefined naming function `BBRegisterNewChildNode` which allows for optional arguments to pass all necessary naming information.
- The grey box "Initialize Child Node" allows the user to initialize node parameters that are associated with the current child node (which is available through the parameter `BBCurrentChildNode`. This routine is automatically called by the bookkeeping engine for every newly registered child node.
- The white box "Mark Selected Node Inactive when Appropriate" is a system routine which marks the selected node as Inactive when no new child node was registered or when all children have been generated as indicated by the value of the system parameter `BBAllChildrenGenerated`.

2.3 Process Current Node

Perhaps the most elaborate task to be specified by the person providing search directives is how to process the current node. There are many ways to process a node. Any choice in this matter is not only problem dependent, but also depends on efficiency considerations (computational time required, likelihood of results, etc.).

Consider the flowchart in Figure 3. Inside the grey box there are several topics. For any particular subproblem in the search tree, not every topic needs to be considered and certainly not in the order specified. The following comments apply.

- Finding implications is an activity that checks whether new bounds and new restrictions can be added to the subproblem at hand based on the restrictions already set for each subproblem along the path to the root of the search tree. The designer must decide (and thus provide control over) whether or when it is worthwile finding implications for a particular subproblem.
- Infeasibility determination is an activity that checks whether the current subproblem and its descendents cannot have a feasible solution. Such a check is essentially wasted when infeasibility cannot be concluded. In case infeasibility is concluded, it should be registered by calling a system routine named `BBRegisterInfeasibility` after setting the predefined parameter `BBIsInfeasible` to `TRUE`. The system will then drop the current subproblem (plus any existing descendents) from the search tree.
- The determination of a local bound (plus possibly a local solution) is an activity that also might lead to the situation in which the system drops the current subproblem (plus any existing descendents). This takes place when a true local bound indicates that the subproblem plus its descendents are no longer candidates for a better solution. If the bound is not a true bound (something that must be communicated to the system through the predefined parameter `BBIsExactBoundComputation`), then the system will not drop the subproblem from the search tree, but will give it the

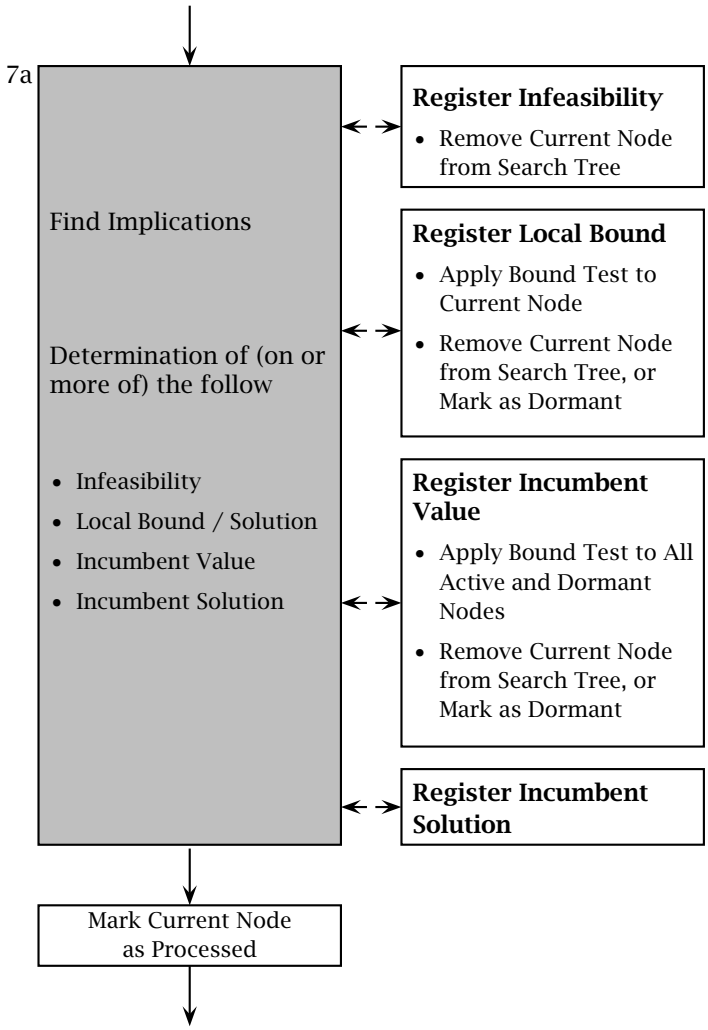

Figure 3: Flowchart of the Current Node Processing Algorithm

Dormant status. Bound computations in practice are rarely trivial, and are usually based on relaxation and/or local search techniques. Finally, the local bound itself must be communicated to the system by calling a system routine named `BBRegisterLocalBound`.

- An incumbent value is usually coupled to a feasible local solution. In that case, this feasible solution also becomes an incumbent solution. By calling the system routine `BBRegisterIncumbentValue` to register the incumbent value, the system will also check whether there are other subproblems that can be dropped from the search tree on the basis of this new incumbent value. By calling the system routine `BBRegisterIncumbentSolution` the system is informed about the fact that the current solution should be considered as an incumbent solution.

- Once the current subproblem has been processed, it is marked as Processed by the system. As a result, it will never be reconsidered for processing. Its Active status, however, remains unaffected so that the node can still be considered for the generation of new children.

2.4 Comparision to MINTO functionality

The main differences between the functionality of MINTO and the functionality of the branch-and-bound enumeration scheme in this paper, apart from the already mentioned language differences and the differences in the use of mathematical programming relaxations, are as follows.

- The child generation step as available in MINTO is linked to the node process step. While MINTO requires all new child nodes of the current node to be generated at once, the branch-and-bound enumeration scheme in this paper allows for multiple child generation phases for a single node by unlinking the child generation and node process step.

- The concept of Dormant nodes (and activation of Dormant nodes) allows for a phased solution process. The underlying tree remains intact during the transition from one phase to the next.

3 Branch-and-Bound Directives in a Modeling Language

The main aim of this paper is to present a branch-and-bound methodology for combinatorial models to be embedded inside an algebraic modeling language. Algebraic modeling languages are primarily designed for the specification of optimization models in terms of indexed constraints. Such notation allows for a representation that stays fairly close to the reality being modeled with the advantage of understandability and maintainability of the resulting symbolic form. This representation in terms of symbolic constraints is called the modeler's form by Fourer [8]. It is the task of the underlying modeling system to automatically translate this modeler's form into the algorithm's form required by solution algorithms. The algorithm's form is an abstract one with reference to just a single array of numbered variables and a single array of numbered

constraints. This representation is not easy to comprehend by a human being and thus also not easy to maintain. This is precisely the weakness of a system such as MINTO, which requires the user to specify both the model and the corresponding branch-and-bound directives in terms of the algorithm's form.

The currently available solvers inside modeling systems have been designed to solve mathematical programs consisting of mostly linear constraints and numeric variables which are either binary, integer or continuous. In this paper there is no longer the requirement that variables are numeric. It will be possible to specify different kinds of set variables which take their value from one or more known master sets. Such decision sets occur quite naturally when describing subselections and partitions. The reliance on standard solvers is then no longer possible, but in their place will be the general branch-and-bound search presented in this paper. Although not further discussed in the sequel, many models expressed in terms of set variables can be translated automatically to an equivalent representation in terms of numeric variables. Eventually, the two approaches (one based on set variables and the other based on numeric variables) will be fruitfully combined into a single branch-and-bound framework.

The branch-and-bound methodology in this paper is to be used as a general search mechanism with no a priori limitations on the shape of the search tree. Nevertheless, it turns out that for most models it is natural to use special constructs for the generation of children leading to prespecified shapes of the search tree. The three structures discussed in this paper are the subset, the subtour and the partition.

These structures together with predefined branch-and-bound identifiers are treated in the remainder of this section. A few detailed examples are provided in Section 4.

3.1 Variable Set Structures

Variable set structures, such as a variable subset, a variable subtour and a variable partition, are available as model identifiers of a particular type, each with its own attributes and methods. These methods are function or procedures that operate on the variable set structures, and are referenced in the same way as attributes, using the dot notation. It is expected that a large number of applications will rely on just a few of these variable set structures.

3.1.1 Variable Subset

The variable subset structure concerns the unknown selection of a subset of a known master set.

Attribute	Value-type
`SUBSET OF`	*set-identifier*
`INDEX`	*index-identifier*
`MINIMAL CARDINALITY`	*numerical-expression*
`MAXIMAL CARDINALITY`	*numerical-expression*

Table 1: `VARIABLE SUBSET` attributes

The SUBSET OF attribute is used to specify the master set with possible elements of the variable subset. The INDEX attribute provides the possibility to declare an index identifier over the variable subset. When used inside a branch-and-bound directive this index will only iterate over the elements that are in the variable subset that corresponds to the current node. The MINIMAL CARDINALITY and MAXIMAL CARDINALITY attributes directly restrict the cardinality of the variable subset. This information will be used by the branch-and-bound enumeration scheme, and will influence the size of the search tree.

Method	Value-type
IsSelected(i)	*boolean-function*
IsExcluded(i)	*boolean-function*
IsFree(i)	*boolean-function*
Select(i)	*procedure*
Exclude(i)	*procedure*
SetFree(i)	*procedure*

Table 2: VARIABLE SUBSET methods

The first three methods associated with the variable subset are used to interrogate the variable subset about its current contents. The boolean function IsSelected(i) returns TRUE if element i is in the variable subset corresponding to the current node. The function IsExcluded(i) can be used to determine whether a certain element i is excluded from the variable subset. All elements that are not selected and not excluded are considered to be free. This status is checked through the IsFree(i) method. An element i can be put in the variable subset corresponding to the current node (and all its descendants) by using the Select(i) method. An element i can be explicitely excluded from the variable subset by using the Exclude(i) method. Finally, an element i that has either been selected or excluded can be set free by using the SetFree(i) method.

3.1.2 Variable Subtour

The variable subtour structure is an unknown cyclically ordered subset of a known master set.

Attribute	Value-ype
SUBSET OF	*set-identifier*
INDEX	*index-identifier*
MINIMAL CARDINALITY	*numerical-expression*
MAXIMAL CARDINALITY	*numerical-expression*

Table 3: VARIABLE SUBTOUR attributes

The attributes of a variable subtour are the same as the previously discussed attributes of a variable subset.

The methods of a variable subtour are not the same as the previously discussed methods of a variable subset. When formulating a model that considers a variable subtour, it is natural to express the essential decisions in terms of links that together form the subtour. For this reason, the methods all have

Method	Value-type
IsSelected(i,j)	*boolean-function*
IsExcluded(i,j)	*boolean-function*
IsFree(i,j)	*boolean-function*
Select(i,j)	*procedure*
Exclude(i,j)	*procedure*
SetFree(i,j)	*procedure*

Table 4: VARIABLE SUBTOUR methods

two indices to represent the start and end element of the link. Based to this link representation, the branch-and-bound enumeration scheme imposes trivial checks on the search tree. One such check makes sure that every element in the tour has exactly one incoming and one outgoing link. As was the case with the VARIABLE SUBSET structure, the first three methods are used to retrieve information about the variable subtour at the current node. The boolean function IsSelected(i,j) returns TRUE if link (i, j) is part of the variable subtour. Similarly, the functions IsExcluded(i,j) and IsFree(i,j) provide information about the exclusion or 'not yet determined' status of link (i, j). For each status there is a corresponding method to set the status. The Select(i,j) method ensures that the link (i, j) is part of the variable subtour corresponding to the current node, while the Exclude(i,j) method is used to explicitely exclude link (i, j) from the subtour. A link (i, j) that has already been selected or excluded can be set free again using the SetFree(i,j) method.

3.1.3 Variable Partition

The variable partition structure is the unknown partition of a set of elements into groups of elements.

The variable partition attributes are for the most part related to the *groups* of elements.

Attribute	Value-type
ELEMENTS	*set-identifier*
GROUPS	*set-identifier*
INDEX	*index-identifier*
MINIMAL CARDINALITY	*numerical-expression*
MAXIMAL CARDINALITY	*numerical-expression*
MINIMAL CARDINALITY PER	*(indexed) numerical-expression*
MAXIMAL CARDINALITY PER	*(indexed) numerical-expression*

Table 5: VARIABLE PARTITION attributes

The ELEMENTS attribute refers to the master set of elements to be partitioned, while the GROUPS attribute refers to the master set of groups making up the partition. The INDEX attribute allows for an index to iterate over the groups in the variable partition corresponding to the current node. The number of non-empty groups in the variable partition can be restricted using the MINIMAL CARDINALITY and MAXIMAL CARDINALY attributes. Similarly, the attributes MINIMAL CARDINALITY PER and MAXIMAL CARDINALITY PER restrict the number of elements per group. All four cardinality restrictions will be used

by the branch-and-bound enumeration scheme to restrict the size of the search tree.

Method	Type
IsSelected(e,g)	boolean function
IsExcluded(e,g)	boolean function
IsFree(e)	boolean function
Select(e,g)	procedure
Exclude(e,g)	procedure
SetFree(e)	procedure

Table 6: VARIABLE SUBTOUR methods

The decisions to be taken when constructing a variable partition are expressed in terms of a single element e and a single group g, indicating the assignment of element e to group g. The boolean function IsSelected(e,g) indicates whether an element e is already assigned to a group g. The boolean function IsExcluded(e,g) is used to determine whether an element e is excluded from a group g. The attribute IsFree(e) is used to provide information about whether element e is assigned at all. As before, the three methods Select, Exclude and SetFree can be used to set the corresponding status.

3.2 Branch-and-Bound Method Declaration

A user-defined branch-and-bound method can be specified by declaring a new ENUMERATION METHOD identifier. The directives of the branch-and-bound method can be specified through its attributes, which correspond with the grey boxes in the three flowcharts described in the previous section.

Attribute	Value-ype	Flowchart
PREPROCESSING	*procedure*	Box 1
INITIALIZATION	*procedure*	Box 2
DECIDE ON EXPANSION	*procedure*	Box 3
SELECT EXPANSION NODES	*procedure*	Box 4
EXPAND SELECTED NODE	*procedure*	Box 5a
INITIALIZE CHILD NODE	*procedure*	Box 5b
SELECT CURRENT NODE	*procedure*	Box 6
PROCESS CURRENT NODE	*procedure*	Box 7a
ACTIVATE DORMANT NODES	*procedure*	Box 8
POSTPROCESS	*procedure*	Box 9

Table 7: BRANCH AND BOUND METHOD attributes

3.3 Branch-and-Bound Parameters

Special predeclared system identifiers can be used to interrogate the enumeration process or to pass additional information to the enumeration process. The value of these predeclared system identifiers are likely to vary from node to node. If so, the value that is passed to the model will be the value at the current node in the enumeration process. General branch-and-bound tolerances (like cutoff tolerances, absolute and relative optimality tolerances) will not be

discussed here, but are communicated to the enumeration scheme through the AIMMS option mechanism.

3.3.1 Predefined Sets and Parameters

The following predefined sets and parameters are available within the branch-and-bound enumeration scheme.

- BBAllNodes, a set identifier denoting the set of all nodes in the search tree.
- BBActiveNodes, a subset of the set BBAllNodes, denoting the set of all currently active nodes.
- BBDormantNodes, a subset of the set BBAllNodes, denoting the set of all currently sleeping nodes.
- BBCurrentNode, an element parameter in the set BBAllNodes, denoting the current node in the enumeration process.
- BBExpandSearchTree, a boolean value indicating whether the search tree should be expanded with new child nodes.
- BBAllChildrenGenerated, a boolean value indicating whether all children have been generated. Once set to TRUE during the child generation routine, the current node is marked as inactive. The default value for this identifier is FALSE.
- BBIncumbentSolutionValue, a real value that denotes the current incumbent solution value.
- BBExactBoundComputation, a boolean denoting whether the local bound computation is exact or not. This information is used by the enumeration engine to decide whether a node should be made *inactive* or *dormant* when its bound test is violated. The default value for this identifier is TRUE.

3.3.2 Tree Administration Routines

The following predefined tree administration procedures and functions are available either to get additional information or to activate the bookkeeping mechanism to perform certain tasks.

- BBParent(n), a function that returns the parent node of node n in the set BBAllNodes.
- BBAncestor(k), a function that returns the k-th ancestor node of node n in the set BBAllNodes.
- BBChildSet(n), a function that returns the set of all generated child nodes of node n.
- BBDepth(n), a function that returns the depth of node n in the search tree. The root node has depth 0.

- `BBNodeNumber(n)`, a function that returns the number of node n. Node numbers are consecutive integers denoting the order in which the nodes are created. The root node has number 1.

3.3.3 Child Identification Routines

The following predefined child identification procedures are available for use during the node creation and node processing phases.

- `BBExpandNode(n)`, a procedure that is called to generate children for node n.
- `BBRegisterNewChildNode(...)`, a procedure to generate a new child node for the current node. Information can be passed through optional arguments. These arguments are available when initializing the node through the functions `BBNumberOfArguments` and `BBArgument(i)`.
- `BBNumberOfArguments`, a function which returns the number of arguments that were passed when genererating the node.
- `BBArgument(i)`, a function which returns the i-th argument that was passed when generating the node.
- `BBCurrentChildNode`, an element parameter in the set `BBAllNodes`, denoting the current child node.

3.3.4 Registration Routines

The following predefined system registration procedures are available for use during the node processing phase.

- `BBRegisterInfeasibility`, a procedure to inform the bookkeeping process that the current node is *infeasible*.
- `BBRegisterLocalBound(x)`, a procedure to inform the bookkeeping process about the value of the local bound for the current node. Based on the value of `BBExactBoundComputation` the bookkeeping process will change the status of the node.
- `BBRegisterIncumbentValue(x)`, a procedure to inform the bookkeeping process about a (potential) new incumbent solution value.
- `BBRegisterIncumbentSolution`, a procedure to inform the bookkeeping process that the solution at the current node should be considered as the candidate solution of the original problem.

4 Examples

In this section there are three worked examples. The first example is complete in the sense that it demonstrates all directives necessary to specify the enumeration process. The last two examples contain a complete model description, but do not provide all required directives to steer the branch-and-bound process. Instead, they focus on the specification of the shape of the search tree.

4.1 A Subset Enumeration Example

The following example describes the same approach to solve a knsapsack problem as was presented in [4]. The knapsack needs to be filled with objects such that the total value of all objects in the knapsack is maximal and the total weight of all objects in the knapsack does not exceed a given maximum capacity. The declaration of the model is straightforward and natural. The formulation makes use of the VARIABLE SUBSET identifier.

4.1.1 Model Declaration

```
SET:
  Objects
    index : o;

PARAMETERS:
  Weigth
    index domain : o;
  Value
    index domain : o;
  Capacity;

VARIABLE SUBSET:
  KnapSack
    subset of : Orders
    index     : k;

CONSTRAINTS:
  CapacityRestriction
    definition : sum[ k, Weight(k) ] <= Capacity;

VARIABLE:
  TotalValue
    definition : sum[ k, Value(k) ];

MATHEMATICAL PROGRAM:
  FillKnapsack
    objective  : TotalValue
    direction  : maximize
    subject to : AllConstraints
    method     : EnumerateKnapsack;
```

4.1.2 Branch-and-Bound Method Declaration

In this example the directives specified below will constitute the entire branch-and-bound enumeration scheme to solve the initial knapsack problem.

```
BRANCH AND BOUND METHOD:
  EnumerateKnapsack
    preprocessing              :
    initialization             : Initialization
    decide on expansion        : DecideOnExpansion
    select expansion nodes     : SelectExpansionNodes
    expand selected node       : ExpandSelectedNode
    initialize child node      : InitializeChildNode
    select current node        : SelectCurrentNode
    process current node       : ProcessCurrentNode
    activate dormant nodes     :
    postprocess                : ;
```

4.1.3 Branch-and-Bound Directives Specification

There are three parameters that are to be registered at every node. Their declaration is as follows.

```
INDEX:
  n
    range : BBAllNodes;

PARAMETERS:
  CurrentValue
    index domain : n;
  PathCost
    index domain : n;
  CapAvail
    index domain : n;
```

The procedure `Initialization` is used to initialize the previously declared three parameters. Note that the initialization is specified for the root node, which at this phase in the enumeration process is available through the system parameter `BBCurrentNode`. Initialization of these parameters for all other nodes is performed in the procedure `InitializeChildNode`.

```
PROCEDURE Initialization:
BODY:
  CurrentValue(BBCurrentNode) := 0;
  PathCost(BBCurrentNode)     := 0;
  CapAvail(BBCurrentNode)     := Capacity;
ENDBODY;
```

In this example the search tree will only be expanded (at the current node) when there are still objects available for placement in the variable subset.

```
PROCEDURE DecideOnExpansion;
BODY:
  if ( count[ o | Knapsack.IsFree(o) ] = 0 ) then
    BBExpandSearchTree := FALSE;
  else
    BBExpandSearchTree := TRUE;
  endif;
ENDBODY;
```

In this example the procedure `SelectExpansionNodes` tells the branch-and-bound enumeration scheme only to consider the current node when generating new child nodes.

```
PROCEDURE SelectExpansionNodes
BODY:
  BBExpandNode(BBCurrentNode);
ENDBODY;
```

For every object that has not yet been selected (or excluded) a new child node is generated in the procedure `ExpandSelectedNode`. All possible child nodes are generated at once. The new object *o* that is added to the variable subset is passed as an argument when generating the child node. This argument is used again in the procedure `InitializeChildNode` to put the new object in the variable subset using the `Select` method.

```
PROCEDURE ExpandSelectedNode:
BODY:
  for (o | Knapsack.IsFree(o) ) do
    BBRegisterNewChildNode(o);
  endfor;
  BBAllChildrenGenerated := TRUE;
ENDBODY;
```

The three parameters for the current child node are initialized through the following recursive relations linking the current child nodes to their parent (i.e. the current node). In addition, the child node itself is characterized by selecting a specific object (i.e. the value of `NextObject`) and excluding all objects already selected by previous children.

```
PROCEDURE InitializeChildNode:
DECLARATION SECTION:
  ELEMENT PARAMETER:
    NextObject
      range : Objects
ENDSECTION;
BODY:
  ! initialize parameters at node
  CurrentValue(BBCurrentChildNode) = CurrentValue(BBCurrentNode) +
                                     Value(NextObject);
  PathCost(BBCurrentChildNode)     = PathCost(BBCurrentNode) +
                                     Weight(NextObject);
  CapAvail(BBCurrentChildNode)     = CapAvail(BBCurrentNode) -
                                     Weight(NextObject);

  ! specify selection and exclusion at node
  NextObject := BBArgument(1);
  for ( o | o < NextObject ) do
    Knapsack.Exclude(o);
  endfor;
  Knapsack.Select(NextObject);
ENDBODY;
```

The procedure `SelectCurrentNode` selects that active node for which the associated knapsack has maximal cardinality. Note that this selection criterion leads to a depth-first search strategy.

```
PROCEDURE SelectCurrentNode
BODY:
  BBCurrentNode := argmax[ a in BBActiveNodes, card(Knapsack) ];
ENDBODY;
```

During the process step the enumeration scheme will conclude infeasibility if the remaining capacity that is available in the knapsack associated with the current node is less than zero. Otherwise, a local bound will be computed and passed to bookkeeping engine by making a call to `BBRegisterLocalBound`. Since a feasible solution of the subproblem at the current node is also feasible for the original knapsack problem, this solution and its corresponding objective function value are also reported to the bookkeeping engine as a candidate incumbent solution and a candidate incumbent solution value, respectively.

```
PROCEDURE ProcessCurrentNode
DECLARATION SECTION:
  PARAMETER:
```

```
    LocalBound;
ENDSECTION;
BODY:
  ! infeasibility check
  if ( CapAvail(BBCurrentNode) < 0 ) then
    BBRegisterInfeasibility;
    return;
  endif;

  ! local bound computation
  LocalBound := CurrentValue(CurrentNode) + CapAvail(CurrentNode) *
                ( max[ o | Knapsack.IsFree(o), Value(o) / Weight(o) ] );

  ! register local bound and solution
  BBRegisterLocalBound(LocalBound);
  BBRegisterIncumbentValue(LocalBound);
  BBRegisterSolution;
ENDBODY;
```

4.2 A Subtour Enumeration Example

The subtour enumeration example considers the budgetted traveling salesman problem as was considered in [4]. A tour must be found such that number of cities visited is maximal while still satisfying an overall budget restriction.

4.2.1 Model Declaration

```
SET:
  Cities
    index : c, f, t;

PARAMETERS:
  LinkAvailable
    index domain : (f, t);
  TravelCost
    index domain : { (f, t) | LinkAvailable(f, t) };
  AvailableBudget;

VARIABLE SUBTOUR:
  Tour
    subset of : Cities

VARIABLE:
  NrCitiesVisisted
    definition : card(Tour);

CONSTRAINT:
  BudgetRestriction
    definition : sum[ c in Tour, TravelCost(c, c++1) ] <= AvailableBudget;

MATHEMATICAL PROGRAM:
  TravelAlongCities
    objective  : NrCitiesVisited
    direction  : maximize
    subject to : AllConstraints
    method     : EnumerateSubTour;
```

4.2.2 Branch-and-Bound Method Declaration

In this variable subtour example the focus is on the specification of the shape of the search tree. That is why only the three relevant directives are discussed below.

```
BRANCH AND BOUND METHOD:
  EnumerateSubTour
    ...
    select expansion nodes : ExpandCurrentNode
    expand selected node   : ExpandSelectedNode
    initialize child node  : InitializeChildNode
    ...;
```

4.2.3 Branch-and-Bound Directives Specification

The procedure `ExpandCurrentNode` only generates new child nodes for the current node.

```
PROCEDURE ExpandCurrentNode
BODY:
  BBExpandNode(CurrentNode);
ENDBODY;
```

The number of new child nodes to be created for a particular node is equal to the number of cities that are not yet visited in the current tour. All child nodes are created at once. Note that the unvisited city *c* is passed as an argument of the `BBRegisterNewChild` procedure.

```
PROCEDURE ExpandSelectedNode:
BODY:
  for (c | Tour.IsFree(c)) do
    BBRegisterNewChildNode(c);
  endfor;
  BBAllChildrenGenerated := TRUE;
ENDBODY;
```

Note that the name of the current child is retrieved using the `BBArgument` function and assigned to the local element parameter `NewCity`. Together with the last city in the variable subtour the value of `NewCity` is used to select the next link to be part of the variable subtour.

```
PROCEDURE InitializeChildNode:
DECLARATION SECTION:
  ELEMENT PARAMETERS:
    LastCity
      range : Cities
    NewCity
      range : Cities
ENDSECTION;
BODY:
  LastCity := last(Tour);
  NewCity  := BBArgument(1);
  Tour.Select(LastCity, NewCity);
ENDBODY;
```

4.3 A Partition Enumeration Example

This variable partition example considers a branch-and-bound enumeration scheme for the order batching problem as described in [1]. In this problem, orders are assigned to batches subject to a maximum batch size. Batches are filled in parallel. For each batch there is a cheapest way to retrieve the orders from a warehouse. This retrieval subproblem can be efficiently solved using an external procedure (referred to as `RatliffAndRosenthal` below). The overall objective is to minimize the makespan (i.e. the largest amount of time required for any batch).

4.3.1 Model Declaration

```
SETS:
  Orders
    index : o;
  Batches
    index : b;

PARAMETER:
  PickTime
    index domain : o;

VARIABLE PARTITION:
  OrderBatchAllocation
    elements                : Orders
    groups                  : Batches
    maximum cardinality per : ceil(card(Orders) / card(Batches));

FUNCTION RatliffAndRosenthal:
  argument list : (CurrentOrders)
  range         : nonnegative;
DECLARATION SECTION:
  SET:
    CurrentOrders
      subset of : Orders
      property  : Input;
ENDSECTION;
BODY:
  ...
  RatliffAndRosenthal := ...;
ENDBODY;

VARIABLE:
  TotalCost;
  BatchCost
    index domain : b
    definition   :
      RatliffAndRosenthal( {o | OrderBatchAllocation.IsSelected(o,b)} );

CONSTRAINT:
  TotalCostDefinition
    index domain : b
    definition   : TotalCost >= BatchCost(b);

MATHEMATICAL PROGRAM:
  OrderBatching
    objective  : TotalCost
    direction  : minimize
    subject to : AllConstraints
```

```
    method     : EnumerateOrderBatchAllocation;
```

4.3.2 Branch-and-Bound Method Declaration

In this variable partition example the focus is on the specification of the shape of the search tree. That is why only the three relevant directives are discussed below.

```
BRANCH AND BOUND METHOD:
  EnumerateOrderBatchAllocation
    ...
    select expansion nodes : SelectExpansionNodes
    expand selected node   : ExpandSelectedNode
    initialize child node  : InitializeChildNode
    ...;
```

4.3.3 Branch-and-Bound Directives Declaration

In this variable partition example only new child nodes are generated for the current node. This is expressed in the procedure `ExpandCurrentNode`.

```
PROCEDURE SelectExpansionNodes
BODY:
  BBExpandNode(BBCurrentNode);
ENDBODY;
```

All child nodes are generated at once in the following manner. Consider the first order that has not yet been assigned to the current node. A new child node is created for every non-empty batch that is not yet full. One more extra child is generated for the first empty batch. By constructing the search tree in this manner all possible assignments of orders to batches are enumerated [1]. Both the batch to be considered and the first order not yet assigned are passed as arguments of the routine `BBRegisterNewChild`.

```
PROCEDURE ExpandSelectedNode:
DECLARATION SECTION:
  ELEMENT PARAMETERS:
    FirstOrderNotYetAssigned
      range : Orders
    FirstEmptyBatch
      range : Batches
ENDSECTION;
BODY:
  FirstOrderNotYetAssigned :=
    first(o | not exists( b | OrderBatchAllocation.IsSelected(o,b)));
  FirstEmptyBatch          :=
    first(b | not exists( o | OrderBatchAllocation.IsSelected(o,b)));
  for (b | b <= FirstEmptyBatch) do
    BBRegisterNewChildNode(b, FirstOrderNotYetAssigned);
  endfor;
  BBAllChildrenGenerated := TRUE;
ENDBODY;
```

Note that the identification of the current child is retrieved using the `BBArgument` function. This identification information is assigned to the local element parameters `Batch` and `Order`, which are then used to assign `Order` to `Batch` using the `Select` method of the variable partition.

```
PROCEDURE InitializeChildNode:
DECLARATION SECTION:
  ELEMENT PARAMETERS:
    Batch
      range : Batches
    Order
      range : Orders
ENDSECTION;
BODY:
  Batch := BBArgument(1);
  Order := BBArgument(2);
  OrderBatchAllocation.Select(Order, Batch);
ENDBODY;
```

5 Conclusions

In this paper a branch-and-bound algorithmic framework is specified in detail to demonstrate the possibilities of providing search directives as part of the algorithm. The main purpose was to express these search directives within the framework of a modeling language. The examples in this paper have shown the feasibility of such an approach, although quite an extensive machinery is required to obtain the kind of flexibility required to specify the branch-and-bound directives for a wide class of problems. This paper forms a serious first step, but an implementation of the methodology proposed herein, coupled with a large number of examples, is required before stronger conclusions can be drawn.

References

[1] J.P. van den Berg, A.J.R.M. Gademann and H.H. Hoff, "An Order Batching Algorithm for Wave Picking in a Parallel-Aisle Warehouse," Report LPOM-96-10, University of Twente, The Netherlands, 1996.

[2] J.J. Bisschop and R. Entriken, "AIMMS: The Modeling System," Paragon Decision Technology, Haarlem, 1993.

[3] J.J. Bisschop and G.H.M. Roelofs, "AIMMS 3: The Language Reference,", Draft, Paragon Decision Technology, Haarlem 1998.

[4] J.J. Bisschop and R. Fourer, "New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages," Computational Optimization and Applications vol. 6, pp. 83–116, 1996.

[5] J.J. Bisschop and A. Meeraus, "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment," Mathematical Programming Study, vol. 20, pp. 1–29, 1982.

[6] A. Brooke, D. Kendrick and A. Meeraus, "GAMS: A User's Guide, release 2.25," Boyd & Fraser/The Scientific Press, Danvers, MA, 1992.

[7] CPLEX, "Using the CPLEX Callable Library," 1997, Version 5.0, ILOG Inc.

[8] R. Fourer, “Modeling Languages versus Matrix Generators for Linear Programming,” ACM Transactions on Mathematical Software, 9, pp. 143-183, 1983.

[9] R. Fourer, D.M. Gay and B.W. Kernighan, “A Modeling Language for Mathematical Programming,” Management Science, vol. 36, pp. 519–554, 1990.

[10] R. Fourer, D.M. Gay and B.W. Kernighan, “AMPL: A Modeling Language for Mathematical Programming,” Boy & Fraser/The Scientific Press: Danvers, MA, 1992.

[11] T. Ibaraki, “Enumerative Approaches to Combinatorial Optimization”, Part I, Baltzer, AG, 1987.

[12] G.L. Nemhauser and L.A. Wolsey “Integer and Combinatorial Optimization” Wiley , NJ, 1988.

[13] G.L. Nemhauser, M.W.P. Savelsbergh and G.S. SigisMundi, “MINTO, a Mixed INTeger Optimizer,” Oper. Res. Letters vol. 15, pp. 47–58, 1994.

[14] M.W.P. Savelsbergh and G.L. Nemhauser, “Functional description of MINTO, a Mixed INTeger Optimizer,” Report COC-91-03C, Georgia Institute of Technology.