REFRAMING DECISION PROBLEMS: A GRAPH-GRAMMAR APPROACH

by

Shimon Schocken
Leonard N. Stern School of Business
New York University
New York, NY 10003

and

Christopher Jones
Faculty of Business Administration
Simon Fraser University
Burnaby, British Columbia

August 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-30

This research has been supported in part by a grant from the National Science and Engineering Research Council of Canada and by National Science Foundation Grant SES-8917966.

Reframing Decision Problems: A Graph-Grammar Approach

December 24, 1991

Abstract

One fundamental requirement in the expected utility model is that the preferences of rational persons should be independent of problem description. Yet an extensive body of research in descriptive decision theory indicates precisely the opposite: when the same problem is cast in two different, but normatively equivalent, "frames," people tend to change their preferences in a systematic and predictable way. In particular, alternative frames of the same *decision tree* are likely to invoke different sets of heuristics, biases, and risk-attitudes, in the user's mind. The paper presents a computational model in which decision-trees are cast as attributed graphs, and reframing operations on trees are implemented as graph-grammar productions. In addition to the basic functions of creating and analyzing decision-trees, the model offers a natural way to define a host of "debiasing mechanisms" using graphical programming techniques. Some of these mechanisms have appeared in the decision theory literature, whereas others were directly inspired by the novel use of graph grammars in modeling decision problems.

1 Graph-Grammars, Networks, and Decision-Trees

Researchers and practitioners of management science often use pictures to represent and analyze complex models. Indeed, there exist many situations in which "a picture is worth a thousand words," as the familiar saying proclaims. Examples include PERT/CPM graphs, data-flow diagrams, influence diagrams, semantic networks, decision-trees, and game-trees. From a functional standpoint, the above graphs are quite different from each other. Yet from a topological, or syntactical, perspective, they can all be seen as different instances of the same class – *attributed graphs.*

Briefly speaking, an attributed-graph is a graph whose nodes and edges are partitioned into different *types.* For example, a decision-tree graph consists of three types of nodes and two types of edges. The node-types are denoted hereafter `choice`, `chance`, and `outcome`, and the edge-types are denoted `echance` and `echoice`. `Choice` nodes represent choices among alternative courses of action (`echoice` edges), whereas `chance` nodes represent different outcomes of random events (`echance` edges). `Outcome` nodes represent final gains and losses, typically expressed in terms of monetary values or utilities.

Different node- and edge-types are characterized by different sets of *attributes*, designed to capture the specific nature of the graph in question. In the case of decision-trees, nodes of type `chance`, `choice`, and `outcome` are typically characterized by the attributes `label`, which represents the node's name, and `value`, which represents the current (expected) value of the subtree rooted at that node. Edges of type `echoice` are characterized by a single `label` attribute, whereas edges of type `echance` are characterized by a `label` and a `probability` attribute. Figure 1 illustrates all the types of nodes and edges that might appear in *any* decision tree graph.

The decision-tree in figure 1 was created with the help of `NETWORKS` [10] – a prototype computer-based environment for building and analyzing graph-based models. In `NETWORKS`, each graph object (node or edge) is characterized by an optional set of domain attributes, e.g., `value` and `probability`, and a mandatory set of system-attributes, e.g. `shape` and `size`. The latter attributes are used to control the display characteristics of the graph. For example, in a decision-tree graph, one can set the `shape` attribute of every `choice`, `chance`, and `outcome` node to the values `rectangle`, `circle`, and `diamond`, respectively.

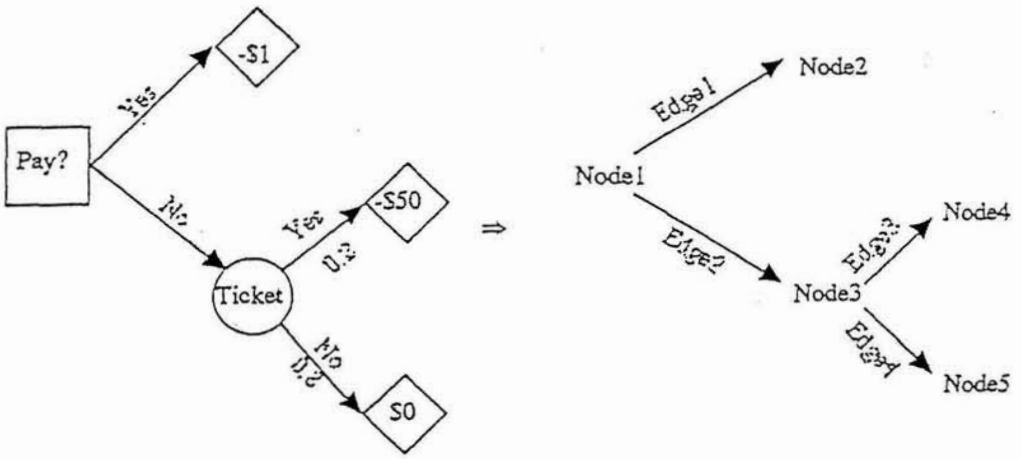

Figure 1: *Left:* A decision-tree that represents a familiar (and repetitive) urban dilemma: should I put money in the parking-meter? Parking-meters provide steady municipal revenues because most people prefer to pay, say, $1 to avoid an expected loss of, say, $10. *Right:* The same tree as an attributed-graph consisting of nine typed objects.

These special values cause NETWORKS to draw the corresponding objects in certain, predefined shapes.

Leaving the subject of graph drawing to a later stage, we now turn to a more fundamental question: how can we formally define generic families of attributed graphs? For example, what would be the formal definition of *decision-tree* graphs? We seek such a definition not for formality's sake; rather, we wish to develop a graphical foundation for building, analyzing, and manipulating decision-tree graphs, as well as other types of attributed graphs. This will enable us to specify and implement a variety of decision-theoretic analyses as graphical operations, using the language of graph grammars.

In order for a graph to qualify as a decision-tree, it must obey certain constraints. The graph objects must be of certain types, and the graph topology must form a hierarchy, each node having at most one incoming edge. These constraints can be described declaratively, using a logic-based formalism, or technically, via a set of data-structures. In this paper, however, we take an alternative approach, inspired by the theory of formal languages (e.g., [6]). Instead of specifying what it takes to *be* a decision-tree graph, we specify what it takes to *build* a decision-tree graph. More specifically, we wish to define a set of construction rules, or *productions*, that are guaranteed to produce and maintain valid decision-tree graphs. By "valid" we refer to attributed-graphs that obey the topological and typological constraints of decision-trees.

We base our constructive approach to modeling on a branch of formal languages called *graph-grammars* (Nagl, [17], [18], Göttler, [3], [4], [5]). Whereas *string-grammars* specify how to build syntactically correct sentences in a certain language, graph-grammars specify how to design and maintain syntactically correct graphs using a predefined set of productions. Our basic premise is that graph-grammars are well-suited to support the building and maintenance of certain families of management science models; in this paper, we demonstrate this proposition in the case of decision-trees.

The uninitiated reader is advised that graph-grammars entail programming and modeling styles which are quite different from those of conventional languages. This formalism will be presented below gradually, as it unfolds in the context of building a "package" for decision-tree modeling. The package, which is essentially a collection of graph-grammar productions, offers all the conventional services for building and analyzing decision-trees, as well as novel tree-manipulation techniques that were directly inspired by the use of

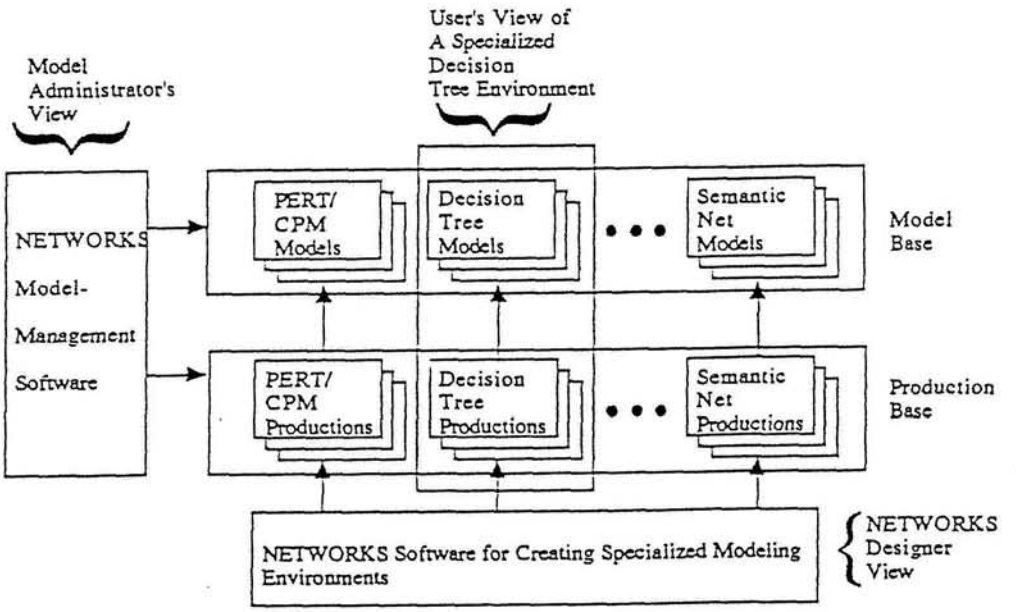

Figure 2: The "world of NETWORKS:" the *general* development environment (bottom block) can be used to create a variety of *specialized* modeling environments, like the decision-trees package enclosed in the vertical block. Through this package, an end-user who knows nothing about NETWORKS can create and maintain a library of decision-tree models. All the models and the productions are archived in a model-base and in a productions-base, respectively, which are managed by a models-management module (the vertical block on the left).

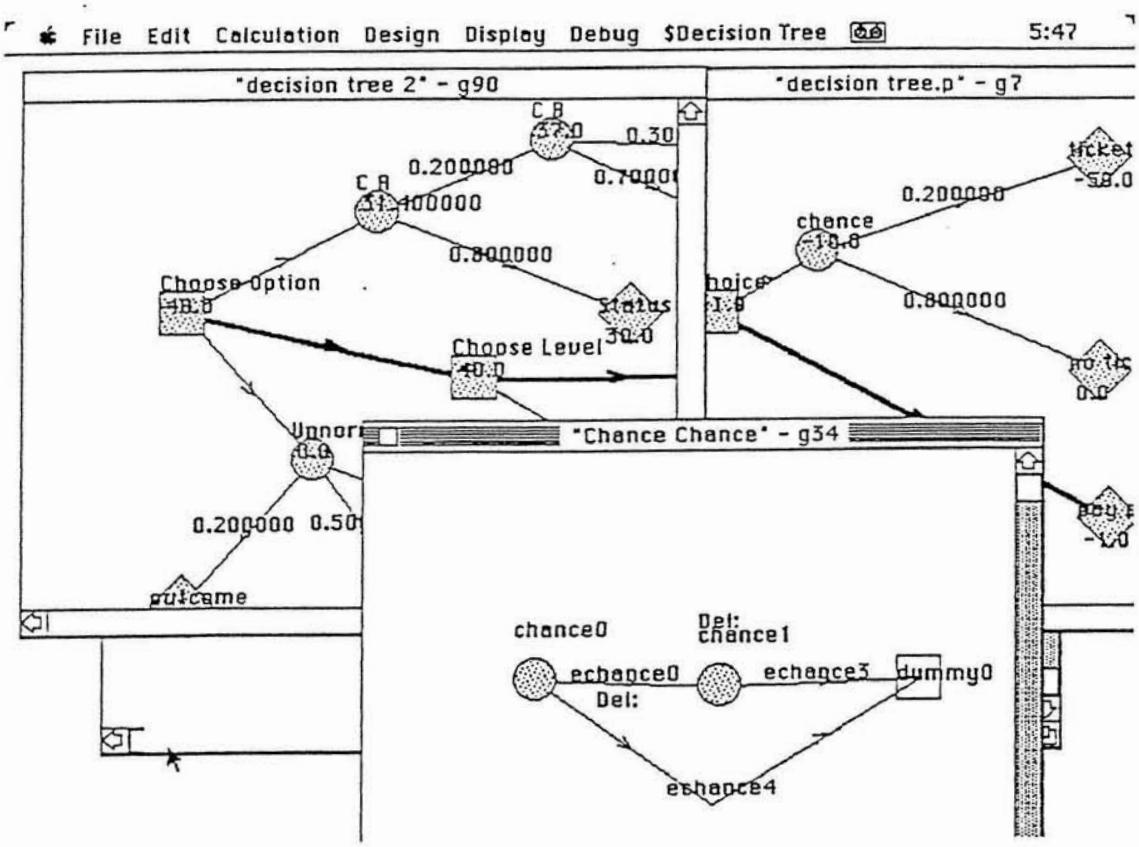

Figure 3: A snapshot of a typical NETWORKS session. The windows in the background contain two different, and possibly unrelated, target-graphs. The window in the foreground contains the production graph of CON, shown also in figure 11. The choice of the three graphs and the positioning of the windows are arbitrary.

graph-grammars. It is important to note, though, that we are not interested in presenting here yet another decision-tree package. Rather, our objective is to expose the reader to the benefits and limitations of the graph-grammar approach to modeling, using decision-trees as a familiar domain of application.

The work described here has been developed with the help of NETWORKS, an extended implementation of the graph-grammar formalism of Nagl and Göttler. Built by Jones ([10],[11]) to facilitate rapid creation of graph-based models, NETWORKS is a *generator of modeling environments*. The NETWORKS approach to modeling is unique in that both models *and* operations on models are implemented as instances of the same type of object – attributed graphs. This uniformity of expression provides extreme flexibility in terms of archiving, retrieving, and combining models and meta-models of different types and purposes. The general architecture of the NETWORKS environment is depicted in figure 2, and a snapshot of its user-interface is depicted in figure 3.

The plan of the paper is as follows. Section 2 presents the basic concepts and terminology that underlie the graph-grammars approach to modeling. Section 3 describes graph-grammar productions for *building* decision trees, whereas section 4 describes graph-grammar productions for *manipulating* decision-trees. Section 5 summarizes the research and discusses the pros and cons of using graph-grammars as a modeling tool in decision theory.

2 Basic Constructs

Our "low-level" modeling syntax was inspired by Prolog. Constants (otherwise called "labels," or "ground terms") are represented by identifiers beginning with lower-case letters, e.g. `graph18` or `edge193`. Variables (otherwise called "unknowns") are represented by identifiers beginning with upper-case letters, e.g. `GraphID` or `NodeType`.

2.1 The Graph Database

An attributed graph is a collection of typed nodes and edges. Each node in the graph is uniquely identified by the term `node(GraphID,NodeID,NodeType)`, representing the graph-identifier, the node-identifier, and the node-type, respectively (the existence of the first argument allows the system to manipulate multiple graphs, or models, simultaneously). In other words, any node is uniquely identified by its graph-identifier, node-identifier, and type. Similarly, each edge is uniquely identified by the term `edge(GraphID,EdgeID,EdgeType, FromNodeID,ToNodeID)`.[1] The last two arguments identify the nodes connected by the edge.

For example, the graph topology corresponding to the decision-tree depicted in figure 1 can be represented through the following database of `node` and `edge` terms:

```
node(tree1,node1,choice).
node(tree1,node2,outcome).
node(tree1,node3,chance).
node(tree1,node4,outcome).
node(tree1,node5,outcome).
edge(tree1,edge1,echoice,node1,node2).
edge(tree1,edge2,echoice,node1,node3).
edge(tree1,edge3,echance,node3,node4).
edge(tree1,edge4,echance,node3,node5).
```

It's important to emphasize that the graph database is neither created, nor is it ever seen, by the end-user (the model builder). The end-user draws his models on the screen by moving the mouse around, clicking menu items, and directly manipulating node- and edge-images. The graph database is created and maintained by the software automatically, as a transparent side-effect of the user's activities.

The `node(·)` and `edge(·)` terms play two different roles in the NETWORKS implementation. In their "ground" version, when they involve constants only, they serve as the building-blocks

[1] This is a shorthand notation of the term `edge(GraphID,EdgeID,EdgeType,FromGraph,FromNodeID, FromType, ToGraph,ToNodeID, ToType)`. The latter notation is necessary when edges cross the boundaries of a single graph – something that doesn't happen in this paper.

of the graph-database, as seen above. In their "predicate" version, when they involve one or more variables, they are used to do pattern-matching, Prolog-style. This is illustrated in the following three examples, which refer to figure 1. Example (1): the expression `node(tree1,node3,NodeType)` will instantiate the variable `NodeType` to the type of `node3`, which happens to be `chance`. Example (2): the expression `node(tree1,NodeID,outcome)` will match all the `outcome` nodes in `tree1` by repetitively binding the variable `NodeID` to the labels `node2`, `node4`, and `node5`. Example (3): the expression `edge(tree1,_,_, node3, NodeID)` will match all the children-nodes of `node3`, i.e., `node4` and `node5`. The two underscore characters indicate that in this particular predicate, the arguments `EdgeID` and `EdgeType` are immaterial, meaning that they can match anything.

2.2 Attributes

In order to retrieve the attribute values of specific graph objects, we use two general-purpose look-up functions: `na(·)`, for nodes, and `ea(·)`, for edges. The syntax of these functions is as follows:

```
na(AttributeID, GraphID, NodeID, NodeType)        (1)
ea(AttributeID, GraphID, EdgeID, EdgeType)        (2)
```

The `na(·)` and `ea(·)` functions are designed to return values, much like function calls in a traditional third-generation language. For example, the function call `ea(probability, tree1,edge4,echance)` will return the number 0.8 – the probability associated with that edge (see figure 1). In order to simplify the use of these functions, some of the variables in (1-2) are allowed to attain default-values, representing the "current-graph," the "current-node," and the "current-edge." To illustrate, the function `ea(probability,*,*,*)`, or `ea(probability)` for short, returns the value of the `probability` attribute of the current-edge in the current-graph. The notion of "currency" is maintained automatically by `NETWORKS`. A more formal definition of attributes and defaults can be found in [11].

The "contents" of an attribute can be either a constant, as in `probability` or `label`, or a user-defined formula, as in `value`. The formulas interact with other objects in the graph, resulting in a dynamic spread of activation similar to that of a spreadsheet program.

(Unlike spreadsheet formulas, though, our formulae understand about graph concepts.) To summarize, an attributed-graph is essentially a database of `node` and `edge` terms. These terms define a connected collection of typed objects, each characterized by a different set of user-defined attributes. The attribute values are computed "as-needed," to borrow a term from frame-oriented programming.

2.3 Formulas

The notion of graph-based formulas is central in our approach to modeling. In order to illustrate it, we'll describe how formulas are used to "roll-back," or "evaluate," a decision-tree graph. First, recall that in a decision-tree, each sub-tree represents a prospect, or a lottery, whose outcome is governed by a known probability distribution. Hence, the `value` attribute of each node x, denoted hereafter $v(x)$, is typically set to the expected-value of the prospect represented by the sub-tree rooted at x. This value is calculated by "rolling" the tree "backward," as follows. Suppose that x has $n \geq 0$ outgoing edges, leading to the children-nodes $x_1, \ldots, x_n$. If x is an `outcome` node ($n = 0$), $v(x)$ is a given constant. If x is a `choice` node, $v(x)$ is set to $\max\{v(x_1), \ldots, v(x_n)\}$, i.e., the value of the course of action that offers the highest reward at the choice junction rooted in x. Finally, if x is a `chance` node, $v(x)$ is set to $\sum_{j=1}^{n} p(x, x_j)v(x_j)$, where $p(x, x_j)$ denotes the value of the `probability` attribute associated with the edge (x, x_j). This formula computes the expected-value of the random-variable represented by x.

Two comments are in order here. First, note that the value formulas are recursive: the computation of $v(x)$ propagates from x all the way down to the leaves of the sub-tree rooted at x, where `outcome` nodes that carry constant values $v(\cdot)$ are encountered. Second, note that the roll-back procedure is based on normative assumptions on decision-making under uncertainty. That is to say, the procedure describes how an ideal automaton who follows the axioms of subjective probability and utility theory will evaluate the risky prospects that the tree represents. It is an embarrassing fact of reality, however, that most decision-makers, experts and laymen alike, exhibit systematic violations of these axioms. In fact, the very same problem, framed in two different (but normatively-equivalent) decision-trees, may well lead people to make two contradictory decisions. One of the objectives of this research is to use graph-grammar techniques to "debias," or at least highlight, such inconsistencies.

Going back to the roll-back procedure, we now show how graph-grammar formulas are used to compute the contents of the value attribute, namely the function $v(x)$. As we saw above, this function depends on the type of x. If x is a choice node, its value attribute will be computed by the formula:

$$\texttt{imax(edge(*,Edge,echoice,x,Y),na(value,*,Y,\_))} \tag{3}$$

If x is a chance *node*, its value attribute will be computed by the formula:

$$\begin{array}{l}\texttt{sum(}\ \texttt{edge(*,Edge,echance,x,Y),}\\ \qquad\texttt{ea(probability,*,Edge,echance)} * \texttt{na(value,*,Y,\_))}\end{array} \tag{4}$$

The function imax(P,V) (standing for "indexed-maximum") computes the largest value of the expression V, given all the possible instantiations of the predicate P. In the particular case of (3), P stands for the predicate edge(*,Edge,choice,x,Y), which is repeatedly instantiated to all the children-nodes of x in the current-graph. For each such instantiation, na(value,*,Y,_) returns the value attribute of the child-node (Y). (The *type* of the child-node is immaterial here, a fact which is denoted by the underscore character, indicating that NodeType in (1) can match anything). The glue that holds the two arguments edge(·) and na(·) together is the shared variable Y and the unification logic of Prolog.

In a similar vein, the function sum(P,V) accumulates the sum of the V values, given all the possible instantiations of the predicate P. In (4), this function is used to compute the expected-value of the random-variable represented by x, namely $\sum_{j=1}^{n} p(x, x_j)v(x_j)$. We leave it to the reader to verify that the declarative expression (4) indeed carries out this algebraic computation.

Hence, we see that recursive operations on trees such as "averaging out and folding back" (Raiffa, [19]) and "rolling back" (Howard, [7]) lend themselves nicely to declarative programming, in general, and to the NETWORKS language, in particular. Now, the fact that a Prolog-like language can be used to represent a set of nodes and edges is neither surprising, nor new. What sets NETWORKS apart from conventional logic-programs is its ability to understand (i) that the nodes and the edges have an important *graphical* interpretation; (ii) that the nodes and the edges have an important *modeling* interpretation; and (iii) that

these two interpretations are tightly interrelated. This, in a nutshell, is the key idea behind the NETWORKS approach to modeling.

2.4 Displaying Graphs

One feature that sets NETWORKS apart from other logic-programs is its ability to draw the database of nodes and edges that constitute what we call an "attributed graph." The display characteristics of the graph are determined by special system-attributes, called *bindings*, which are automatically assigned to every node created by the user. The bindings have names like `shape`, `color`, and `position`, and they contain either constant values or user-defined formulas, just like ordinary attributes. This flexibility gives the model designer full control over the display characteristics of his graphs.

In the case of decision-trees, the designer can program the system to highlight certain areas in the graph that deserve special attention from the user. For example, different colors can be used to distinguish promising courses of action from other edges in the graph. Specifically, recall that each node of type `choice` represents a choice among alternative courses of action, each leading to a different sub-tree, or prospect. When the tree is "rolled-back," the values of the sub-trees are computed, and one of them emerges as the course of action which offers the highest payoff in the decision junction under consideration. We call the edge that leads to that sub-tree the *optimal choice edge*, and the union of all these edges in the graph the *optimal choice path*. The optimal choice path is a road map that tells the user, in every choice junction along the way, which course of action will maximize his expected payoff.

Therefore, it would be nice to have a built-in mechanism that continuously highlights the optimal choice path implied by the tree's data. For example, we can decide to color all the `echoice` edges green, with the exception that optimal edges are colored red. This coloring scheme can be implemented by setting the `color` binding of each `echoice` edge `(x,y)` in the tree to the following formula:

```
if((edge(*,*,echoice,X,Y),
    na(value,*,X,_)=na(value,*,Y,_)),
    red,
    green)
```

The value of the function `if(Condition,TrueExp,FalseExp)` is `TrueExp` if `Condition` is true, and `FalseExp` otherwise. In the above example, `Condition` is a conjunction of the two expressions `edge(*, *,echoice,X,Y)` and `na(value,*,X,_)=na(value,*,Y,_)`.[2] The first expression binds the variables `X` and `Y` to the two nodes that reside at the end-points of the edge whose color we wish to determine. The second expression is `true` if the `value` of both nodes is equal, indicating that the edge is optimal (recall that the value of a *parent* choice node is set to the maximum value of its children nodes. Hence, if x is a parent of y and $v(x) = v(y)$, the edge (x, y) is optimal). To sum up, if the conjunction is `true`, the value of the `if` expression will be `red`; otherwise, it will be `green`. Since the `if` expression is bound to the `color` attribute of the edge (x, y), the edge will be drawn on the screen in that color.

This example illustrates the notion of dynamic graphics, where images are automatically redrawn by formulas which are continuously recalculated. Since the formulas establish links among various graph objects, any change in one object immediately effects all the other objects that depend on it, either directly or indirectly. This comes particularly handy in the case of sensitivity analysis. If the user wishes to see what will be the impact of different probability or payoff assumptions on his decisions, all he has to do is select certain objects in the decision-tree graph and make the necessary changes in their attributes. These changes will propagate via the formulas throughout the entire tree, causing the system to "automatically" redraw the new optimal path implied by the new assumptions.

[2] In the Prolog language, the expression `(p,q)` reads `p` and `q`. Hence, the expression `if((p1,...,pn),x,y)` will evaluate to `x` if and only if all the predicates `p1,...,pn` are true. If at least one of these predicates is false, the expression will evaluate to `y`.

2.5 Productions

As is typically the case with programmable environments, NETWORKS has two types of "stakeholders": designers, and users. The designer (programmer) is the person who builds a specialized modeling environment for a certain family of models, say *vehicle routing.* The user (e.g. a transportation analyst) is the person who uses the environment to define, manipulate, and solve, *specific* transportation models. The designer has to have some basic understanding of the user's world, but the user needs know nothing about the world of graph-grammars. As far as the user is concerned, the modeling environment consists of an intelligent scratch pad that "understands" what it takes to build and analyze vehicle routing models.

In principle, end-users can build and manipulate graphical models in NETWORKS directly, by means of a graph-oriented editor, or indirectly, by invoking programmed *productions.* Graph-grammar productions are conceptually similar to general-purpose subroutines, or macros, in a conventional language. For example, a specialized environment for vehicle routing will consist of a library of productions designed to assist transportation analysts and dispatchers in their typical tasks, e.g. adding customers, deleting customers, rerouting subtours, changing capacity and demand constraints, etc. These productions will be written by the *designer* of the vehicle routing modeling environment, and will be accessible to the *end-users* via a graphical interface consisting of multiple windows and pull-down menus (see figure 3).

A production is a general purpose piece of code, designed to operate on a wide variety of *target* graphs, just like a spreadsheet macro can be applied to many different areas in the same spreadsheet. In spite of this functional similarity, though, spreadsheet-macros and graph-grammar productions are conceptually far apart. A macro is essentially a stored sequence of keystrokes, whereas a production is a *graph*, quite similar to the target-graphs on which it is designed to operate.

Formally, a production $\mathcal{P}$ is a triplet $\mathcal{P} =< G^L, G^R, T >$[3]. When applied to a target-graph G, $\mathcal{P}$ checks if there is a subgraph $\overline{G}^L \subset G$ which is isomorphic to G^L. If such a subgraph is found, $\mathcal{P}$ transforms G into a new graph, in which $\overline{G}^L$ is replaced with G^R. This operation consists of two conceptual steps, as follows. First, the subgraph $\overline{G}^L$ as well as additional

[3]For historical reasons, G^L and G^R are called the "left-side" and the "right-side," respectively.

nodes and edges in its boundary (the "left side") are removed from G in order to create room, or *embed*, the new subgraph G^R. The result is a "temporary" graph, denoted $G \setminus \overline{G}^L$ (G with $\overline{G}^L$ removed). Next, G^R as well as a new set of connecting edges (the "right side") are implanted in $G \setminus \overline{G}^L$. The exact details of this tricky surgery are specified by the production's *embedding transformation* – T.

The nodes and the edges of production-graphs are similar, but not identical, to those of ordinary graphs. To begin with, each object (node or edge) in a production-graph is characterized by all the user-defined attributes of its corresponding (matched) target-object. However, the *contents* of the attributes is somewhat different at the production level. Specifically, when a new node (or edge) is added to a target-graph via a production, one has to specify the *formulae* that will "reside" in its attributes. The specification mechanism is called an *attribute transformation* expression, which is essentially a meta-formulae, as it calculates target formulae rather than scaler values such as integers, reals, and strings of characters.

In addition to the attributes of its corresponding target-object, each object in a production-graph is characterized by four mandatory production-specific attributes: `$hand`, `$delete`, `$selectbefore`, and `$selectafter`. Finally, a production-graph can refer to two special object-types that are unique to productions only: `$universal`, and `$path`. The description of these "system" attributes and object-types are given in tables 1 and 2, respectively.

The special attributes and object-types enumerated in tables 1 and2 provide all the necessary building-blocks for encoding the two essential ingredients of graph-grammar programming: (i) the pattern-matching logic that identifies the graph-pieces on which the production should operate, and (ii) the manipulation logic that specifies the transformation that those graph-pieces should undergo. The resulting programming formalism is Turing-complete [17].

We see that one intriguing feature of productions is that just like the objects on which they operate – graphs – they themselves can be represented as graphs. This elegant idea went through several steps of refinement. In Nagl's [17] original version of graph-grammars, the embedding transformation was specified as a set of textual expressions. Göttler [3] showed how Nagl's graph-grammar formulation could be represented as a graph. Jones extended this work further and developed a demonstrable prototype. As a result of these extensions,

Attribute Name	Possible Values	Description
$hand	find-one	Find one corresponding copy of this object in the target-graph.
	find-all	Find all corresponding copies of this object in the target-graph.
	add-one	Add one copy of this object to the target-graph.
	add-all	Add one copy of this object to the target-graph for each corresponding connected set of find-one and find-all objects found in the target-graph.
$delete	delete	Delete the corresponding target object after the production is complete. For find-one objects, this is the default.
	retain	Do not delete the corresponding target object after the production is complete. For all objects except find-one nodes, this is the default value.
$selectbefore	true	The corresponding target object must have been selected. For find-one objects, this is the default value.
	false	The corresponding target object may not have been selected. For allobjects except find-one objects, this is the default value. For add-one objects, this attribute is ignored.
	⟨*Label*⟩	The corresponding target object must have been selected and assigned the specified selection label.
$selectafter	true	The corresponding target object will be selected after the production completes.
	false	The corresponding target object will not be selected after the production completes. This is the default value.
	⟨*Label*⟩	The corresponding target object will be selected and assigned the specified selection label after the production completes.

Table 1: Production-specific attributes, used to control "production programming." Note that the labels find-one, find-all, add-one and add-all replace the "historical" labels left, embedding, right and connecting that were used by Jones in [10] and [11]. We believe that the new labels are less cryptic and more informative than the original ones.

Node/Edge Type	*Description*
`$universal`	A node or edge in a production graph labeled `universal` will match *any* type of node or edge in the target-graph.
`$path`	An edge in a production-graph labeled `path` will match a path of of indeterminate length in the target-graph.

Table 2: Production-specific node- and edge-types, also used to control production programming.

the production-graphs that are illustrated below are considerably more concise than would be possible using Nagl's and Göttler original formulations.

Since a production-graph is a special instance of attributed graphs, the designer can create productions in NETWORKS via an editor specifically designed for production-graph programming. Once created and tested, the user can invoke the production quite simply, as follows. First, the user places the target-graph in a certain work-area (window). Next, he uses a pointing device like a mouse to "select" certain objects (nodes and edges) in the target-graph. Finally, the user invokes the production by pulling down a menu and clicking the mouse on the name of the production that he wants to invoke. This will cause the production to go to work on the selected objects in the target-graph.

The details of production-graph programming are somewhat obscure, and one needs to see several examples in order to appreciate the simplicity, power, and limitations, of this novel computational paradigm. In the next section we present many such examples, given in the context of decision tree graphs. In particular, we will show how decision trees can be built and edited using two families of productions: one for inserting nodes to, and the other for deleting nodes from, a tree-shaped graph.

3 Model Definition Productions

As with database management, model management activities fall into two distinct categories: model definition, and model manipulation. This section discusses graph-grammar mechanisms for defining, or rather building, graph-based models.

Since a decision-tree is a special case of an attributed-graph, we could have let users build models directly, by giving them unabridged access to a general-purpose graphical editor that allows the creation and connection of nodes and edges in a free-form fashion. However, this freedom of expression may well turn into chaos, as it would allow users to develop invalid models, namely attributed graphs that violate the topological and typological constraints of decision-trees. Hence, rather than providing the user with a set of primitives for defining unconstrained graphs, we seek to develop higher-order construction operators that understand the special nature of decision-trees. Our approach is inspired by research on syntax-directed editing [20], which first came to prominence in the context

of writing computer programs. A syntax-directed editor "is aware" of the special nature of the target-text on which it operates. For example, a specialized Pascal editor offers a variety of Pascal-specific editing services such as nested indentation, tests of variable declarations, and insertion of `IF` and `WHILE` templates with single keystrokes. In a similar vein, specialized editors can be created to support program-writing in any other programming language. These context-sensitive editors speed up the software development process, and, more importantly, promote the construction of correct and readable programs.

Just as a specialized editor can be tailored to support the process of writing Pascal programs, a specialized graphical environment can be tailored to support the process of building and maintaining decision-tree graphs. This is because decision-trees, like Pascal programs, are not born in a vacuum; they must obey a set of well-defined structural constraints. With that in mind, our approach to designing a graph-grammar environment for building decision-trees is based on three steps, as follows: (a) Enumerate all the generic operations that underlie the design and maintenance of valid decision-tree graphs; (b) Define each operation as a separate graph-grammar production; and (c) Wrap the resulting library of productions with a congenial user interface.

3.1 Insertion

The construction of any decision-tree graph can be seen as a sequential application of a subset of twelve generic insertion-operations, as follows:

> Insert a new {`choice`|`chance`|`outcome`} node as a *child*
> of an existing {`choice`|`chance`} node.
>
> Insert a new {`choice`|`chance`} node as a *parent*
> of an existing {`choice`|`chance`|`outcome`} node.

We assume that before an insertion-production has been invoked, the user has selected some node in the target-graph (the graph on which the production operates), using a pointing-device like a mouse. The selected node, which will become the production's "anchor," is denoted hereafter $\overline{x}$. Each one of the above twelve insertion-productions can be applied

either uniquely, or repetitively, with respect to $\overline{x}$. A *unique* insertion of a new node $\overline{y}$ as the child (parent) of $\overline{x}$ attaches one copy of $\overline{y}$ below (above) $\overline{x}$. A *repetitive* insertion of a new node $\overline{y}$ with respect to $\overline{x}$ attaches one copy of $\overline{y}$ below (above) every node in the tree whose `label` attribute is identical to that of $\overline{x}$.[4] To illustrate the resulting family of twenty four productions, we now turn to describe one representative example – *unique insertion of a new outcome node as a child of an existing choice node.* This production was chosen because of its relative simplicity; as we introduce additional productions, we will gradually increase their level of complexity.

Consider a decision-tree in which a certain `choice` node, labeled `drilling-decision`, leads to two children nodes, labeled `drill` and `no-drill`. Suppose that it is now required to refine this binary choice by adding to it a new node, labeled `drill-test`. If you were to carry out this editing operation using paper and pencil, how would you go about it? First, you would locate, or *select*, all the nodes in the tree that are labeled `drilling-decision`. Next, you would draw a new node labeled `drill-test` below one of the selected nodes, and connect both nodes with a new edge. Finally, you would repeat the exact same operation for each node selected in the first step. The production that carries out node insertion operates in precisely the same manner. The name of the production is `INS`, and its production-graph is depicted in figure 4.

The logic of the `INS` production is very similar to that of its "paper and pencil" version. Node x represents the `choice` node that is about to be extended. Node y represents the new `outcome` node that has to be connected to x. We assume that before the production has been applied, the user has already selected one or more nodes, denoted $\overline{x}$, in the target-graph. (Note that we use an overbar to distinguish a node in the target-graph, e.g., $\overline{x}$, from its corresponding node in the production-graph, e.g., x.) The specific operation of `INS` is determined by the special labels that mark its objects. The `selectbefore` label indicates that x must have been selected by the user before the production was invoked. The `find-all` label specifies that the production will operate on *all* the nodes selected by the user.

The `add-all` labels that mark the edge (x, y) and the node y are instructions to add copies of these generic objects to the target-graph. Specifically, they indicate that for each

[4] Repetitive insertion is a natural operation in decision trees. Since a tree might contain multiple copies of the same sub-tree (representing a prospect that can be reached at under different contingencies), the insertion of a new branch to that prospect must be repeated in all its occurrences in the tree.

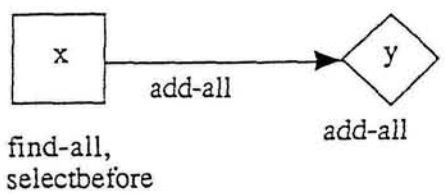

Figure 4: The insertion-production INS, designed to add a new `outcome` node to a selected `choice` node.

selected and found node $\overline{x}$, the production should (a) create a new outcome node – $\overline{y}$; and (b) connect $\overline{x}$ to $\overline{y}$ by a new `echoice` edge – $(\overline{x}, \overline{y})$. In other words, one copy of each `add-all` object will be added to each `find-all` object that was actually found in the target-graph. (Note in passing that `find-one` and `find-all` translate roughly to the notion of $\exists$ and $\forall$ in logic.)

We conclude this section with a few words about the user-interface of `INS`. In order to invoke this production, the user (model builder) should first identify the target-nodes that ought to be extended (multiple selection is done by pressing a special key while clicking on the nodes with the mouse). Having marked the nodes of interest, the user would pull down an "insertion operations" menu and select an entry entitled "insert a new outcome node." This entry will invoke the `INS` production, which will go to work on the selected areas in the target-graph.

3.2 Deletion

Deletion productions are designed to delete selected nodes from a decision-tree graph. As we did in the case of insertion, we assume that before a deletion-production is invoked, the user has already selected a certain target-node, denoted $\overline{x}$, as a candidate for deletion. The user may want to delete $\overline{x}$ in two alternative ways: (a) delete the node $\overline{x}$ and all but one of its children-nodes, which should be reconnected to $\overline{x}$'s parent-node; or (b) delete the node $\overline{x}$ and all the nodes that descend from it, i.e. the sub-tree rooted in $\overline{x}$. In this section we demonstrate the latter operation, which is implemented by a single production – `DEL` – whose graph is depicted in figure 5.

The node x, which is marked `find-one` and `selectbefore` in figure 5, represents the root of the sub-tree that has to be deleted. The `universal` label attached to y allows this production-node to match target-nodes of *any type*, meaning that the production is insensitive to the *type* of the children of the deleted node. Nodes x and y are connected by a `path` edge – a special production-level edge-type which represents a directed-path of indeterminate length in the target-graph. Finally, the label `find-all` which marks the path (x, y) as well as node y will cause the production to operate on *all* the paths, of any length, that emanate from $\overline{x}$ in the target-graph. In a similar vein, y will match all the nodes $\overline{y}$ that can be found along these paths.

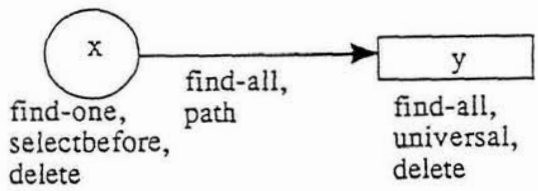

Figure 5: The deletion-production, DEL, designed to delete a selected chance node (x) and all the nodes and edges that descend from it.

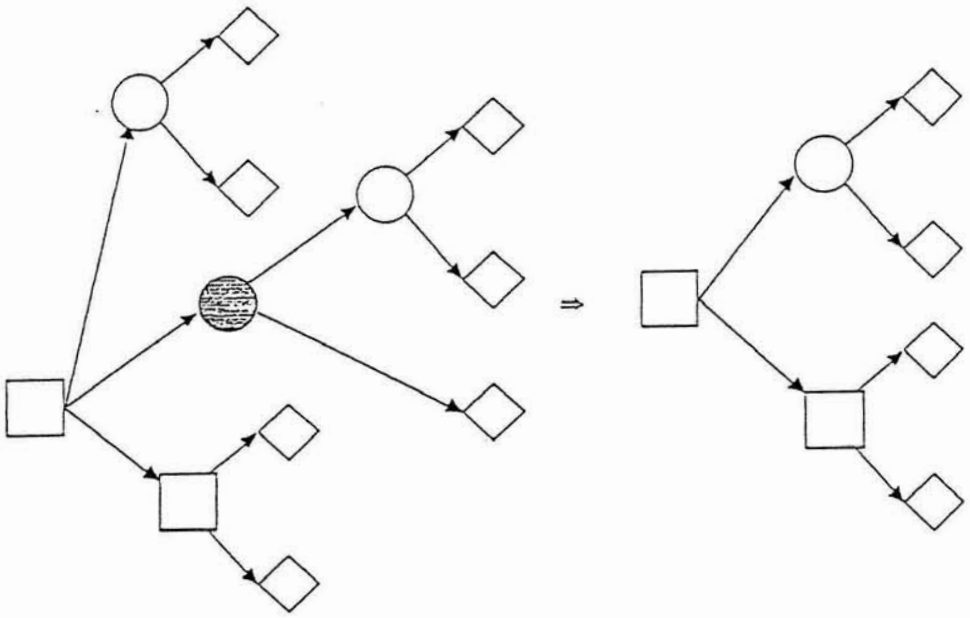

Figure 6: A decision-tree before and after applying the DEL production to the shaded node. The shaded node marks the node selected by the user.

It is interesting to note that all the labels that were mentioned thus far in the context of DEL are used for pattern matching only. In other words, these "passive" labels don't do anything – they just *enable* the operation of the production, similar to the left side of an IF condition THEN action rule. The only "active" instruction in DEL's definition is the delete label, which marks the node x. This will cause the production to delete the selected node ($\overline{x}$), all the edges attached to it, as well as all the nodes that descend from it and their attached edges, thus accomplishing the original plan for this production. The actual operation of the production is demonstrated in figure 6, where the user is assumed to have applied it to the shaded node in the left decision tree. When the production ends, the left tree is transformed into the right tree.

4 Model Manipulation Productions

The application of proper insertion and deletion productions is a necessary, but insufficient, condition for designing and maintaining *good* decision-tree graphs. In this section we turn to describe productions that are capable of *reframing* a decision tree in a number of different ways. We define "reframing" as an editing operation that transforms a given tree t into a new tree t' in such a way that leaves t and t' normatively equivalent, i.e. $U(t) = U(t')$, U being a standard Von-Neumann Morgenstern utility function.

The notions of reframing and equivalence in decision trees have been discussed in detail in the literature both from a *normative* perspective, e.g. [13] [14], [15], and [25], as well as from a *descriptive* perspective, most notably by Tversky and Khaneman (e.g. [27]). The normative line of research focuses primarily on the axiomatic validity of various reframing operations. The descriptive studies indicate that decision trees that are *normatively* equivalent are not necessarily *psychologically* equivalent, leading to decision making behavior which is sometimes inconsistent with expected utility theory.

Our own treatment of reframing is based on the premise that users of decision trees are not automatons, but rather human beings, guided by bounded rationality and equipped with limited computational devices [22]. In particular, we assume that different representations of the same decision tree might invoke different sets of cognitive biases in the user's mind. Furthermore, we follow Tversky and Khaneman [27] in assuming that users (a) will be unaware of the *existence* of alternative tree representations; (b) will not be willing to

go through the trouble of *constructing* such presentations; and (c) will be incapable of *comparing* the impact of different (but normatively equivalent) representations on their decisions.

Hence, the need for decision support arises here because "there is a human tendency to act on the most readily available frame, which may be attributed to the mental effort required to explore other alternatives" [27]. With that in mind, we wish to provide the user with an arsenal of reframing operations which we divide into four categories: pruning, optimizing, consolidating, and reversing. The remainder of the paper motivates the need for these operations, presents their conceptual definitions, and provides their graph-grammar implementations.

4.1 Pruning

A decision-tree model describes a sequence of *junctions*. Choice junctions represent alternative courses of action, whereas chance junctions represent alternative contingencies. When some of the alternatives that emanate from the same node overlap, the result is typically an unnecessarily cluttered model. For example, consider a chance node that branches into three probabilistic outcomes: a 50% chance of gaining \$100, a 25% chance of losing \$30, and a 25% chance of gaining \$100. Clearly, a more parsimonious description of the same prospect would be a two-way junction, representing a 75% chance of gaining \$100 versus a 25% chance of losing \$30. When a chance (choice) node has two or more outgoing edges that represent identical or similar contingencies (courses of action), we say that the node is *superfluous*.

Superfluous nodes arise because of three reasons. First, the decision problem that the tree represents may contain a genuine element of redundancy, in which case the "superfluous" nodes give an accurate picture of reality, and there is nothing wrong with it. Second, since superfluous nodes do not violate the topological restrictions of "being a tree," one can create them either unintentionally, or by bad design, through the use of valid tree-construction productions. Finally, superfluous nodes emerge as a degenerate side-effect of certain tree-manipulation productions, as we'll see in later in the paper.

Regardless of the reasons for their existence, though, it is necessary to be able to detect superfluous nodes, and, if the user so desires, proceed to delete their redundant edges and subtrees. We call this operation *pruning*. The following two problems motivate the need for a pruning production.

> **Problem 1:** The following game begins by spinning a roulette wheel which is numbered 1 to 50. If the lucky number is 10, 20, 30, 40, or 50, you win \$100. Otherwise, you win nothing. The cost of playing the game is \$10. Do you want to play?
>
> **Problem 2:** The following game begins by spinning a roulette wheel which is numbered 1 to 100. If the lucky number is between 1 and 10 (inclusive), you win \$100. Otherwise, you win nothing. The cost of playing the game is \$10. Do you want to play?

Note that both games represent the same prospect – a 10% chance of winning \$90 versus a 90% chance of losing \$10, problem 2 being the pruned version of problem 1. Yet the problems "look" different, as the frame of game 1 seems to offer a better "spread," and thus a better chance, of winning. So which problem gives a better representation of the true odds for winning – 1 or 2? We argue that at least on the grounds of parsimony and clarity, a pruned decision-tree is a better decision aide than a superfluous tree, provided of course that the pruning operation does not distort the original setting of the problem.

The pruning logic is simple. Let x be a node with n outgoing edges, leading to the children-nodes $x_1, \ldots, x_n$. Suppose, without loss of generality, that x_1 and x_2 are deemed redundant. Pruning consists of removing the edge (x, x_2) and the sub-tree rooted at x_2 from the sub-tree rooted at x. If x is of type `choice`, this completes the pruning operation. If x is of type `chance`, the probability of the removed alternative, $P(x, x_2)$, should be added to the probability of its remaining twin, namely to $P(x, x_1)$. If node x contains more than two identical alternatives, the same procedure can be applied repeatedly.

figure 8 depicts a graph-grammar production, `PC1`, designed to prune nodes of type `chance`. The key players here are the `chance` node, x, and its two children-nodes, x_1, and x_2 (node y will be discussed later). The three nodes are marked `find-one`, meaning that a copy of each must be found in the target-graph in order for the production to be applied. The `selectbefore` and `selectafter` labels of x indicate that the user must select this node

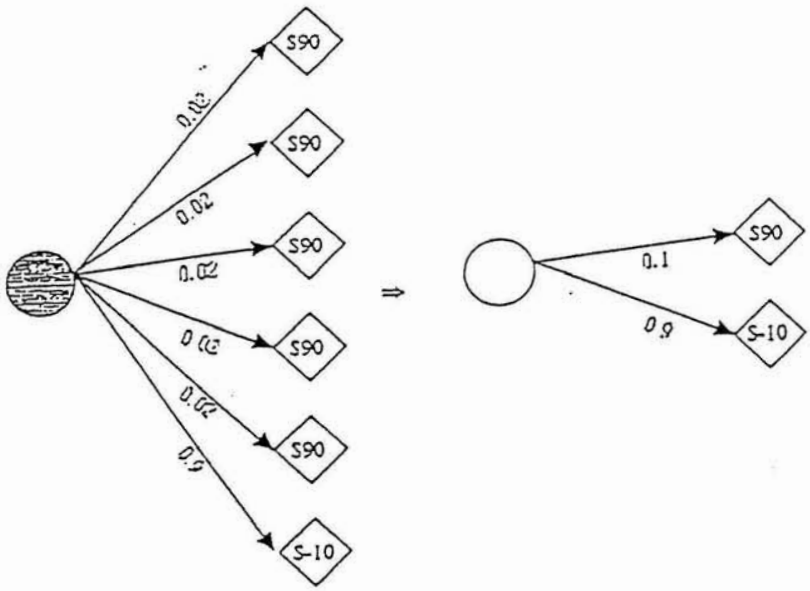

Figure 7: Problem 1 (left) and Problem 2 (right).

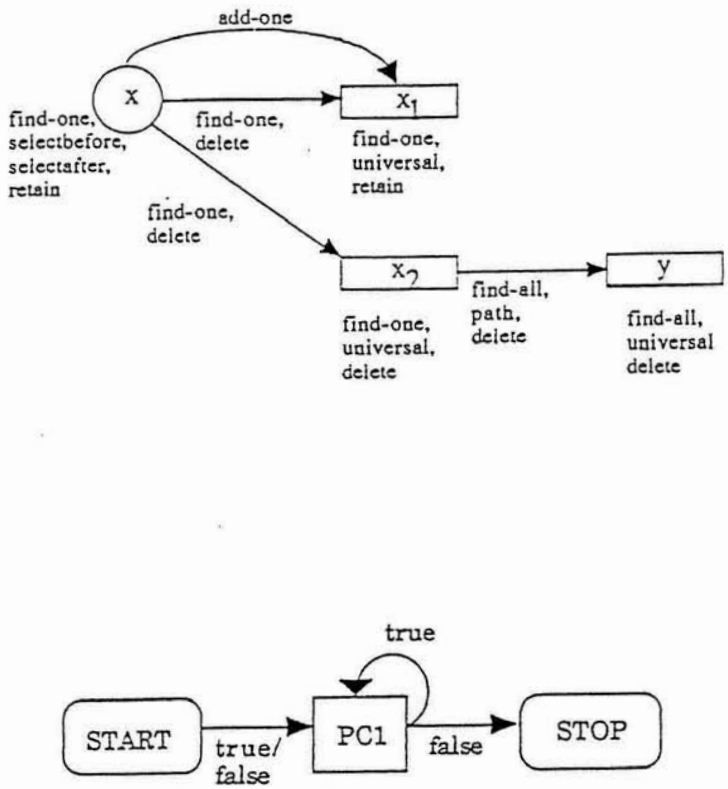

Figure 8: The pruning production PC1 (top) and the pruning program PC (bottom).

before the production starts, and that the node will remain selected after the production ends. The `universal` labels indicate that the production is insensitive to the *types* of x_1 and x_2, as long as they have the same type. Node x_2 is marked for deletion, but only if it is deemed redundant to x_1. The redundancy-test, which is not explicit in figure 8, is stored as an *applicability-predicate*, to be discussed shortly.

Note that the edge (x, x_1), which is marked for deletion, will be immediately replaced with a new edge, marked `add-one`. This structural trick is required because of a technical reason. Recall that the `probability` value associated with the edge (x, x_1) should be incremented by the probability attribute associated with (x, x_2). In `NETWORKS`, however, a production can only change the attributes of objects that are *added* to the graph. That's the reason behind the deletion and immediate reincarnation of the edge (x, x_1).

We now turn to discuss the applicability-predicate that tests whether or not nodes x_1 and x_2 are redundant. In order for a production to be applied, two conditions must be satisfied. First, there must be a topological match between the objects marked `find-one` and `find-all` in the production-graph, and corresponding objects in the target-graph. Second, the production's `applicability-predicate` – a graph-level attribute which is unique to production-graphs only – must evaluate to `true`. The `applicability-predicate` is essentially a logical expression that can be used to enforce additional constraints on the execution of a production. In the specific case of `PC1`, the applicability-predicate consists of the following expression:

$$\texttt{na(label,*,x1,\_)} = \texttt{na(label,*,x2,\_)} \tag{5}$$

Recalling that the definition of the `na` function is `na(AttributeID,GraphID,NodeID, NodeType)`, we see that (5) will be `true` if and only if the target-nodes corresponding to x_1 and x_2 have the same `label` values, implying redundancy. The underscore characters in the fourth argument indicate that the node-types of x_1 and x_2 are immaterial. It's important to emphasize that from a software engineering standpoint, other redundancy-tests can also be implemented with similar ease. For example, the application might render x_1 and x_2 redundant if $|v(x_1) - v(y_2)| < \epsilon$, for a certain tolerance level $\epsilon > 0$. Whichever redundancy-test we choose to adopt, the test can be easily implemented via an applicability-predicate such as (5).

Let us suppose then that (5) evaluates to `true`. This will cause the `PC1` production (top of figure 8) to delete node $\overline{x}_2$ and all the nodes that descend from it in the target-graph. The scope of the deletion operation is specified in the production-graph by the `path` edge (x_2, y) and the node y, which are both marked `find-all` and `delete`. Note, however, that `PC1` is a single-step deletion operator. As written, it is programmed to delete *one* redundant edge from a selected node, provided that such an edge exists. The case of $m > 2$ redundant nodes can be handled by writing a *program-graph* that applies `PC1` to the same node repetitively, until the node contains no further redundancy.

Program-graphs are the graph-grammar equivalent of flow-charts, except that their building blocks are productions, rather than instructions or subroutines (as in a procedural language). The flow of control of a program-graph is governed by the *values* that the productions return. Typically, each production is designed to either succeed, or fail; success indicates that the target-graph has been changed according to the production's specifications, whereas failure indicates that the production could not match its "left side" (G^L) with an isomorphic subgraph in the target-graph, and thus could not be applied (see section 2.5). Using the truth values that the productions return, program-graphs enable the implementation of repetitive graph manipulations in the spirit of `WHILE` and `REPEAT` loops.

In the specific case of pruning, the repetitive application of `PC1` to the same node is carried out by a program-graph named `PC` (bottom of figure 8). The logic of `PC` is as follows. First, `PC` invokes `PC1`, which then tries to detect a pair of duplicate nodes that emanate from the selected node. If such a pair is found, `PC1` proceeds to delete one of its members, returning the value `true`. This causes `PC` to invoke `PC1` once again to the same node. (Note that x_1 is marked both `selectbefore` and `selectafter` in `PC1`). The cycle continues until `PC1` returns the value `false`, indicating that the selected node contains no further redundancy. This, in turn, will cause `PC` to terminate its execution. The application of `PC` is demonstrated in figure 7, where the tree on the right represents the result of applying `PC` to the shaded node in the tree on the left.

We conclude this section with three comments on program-graphs. First, like `WHILE` and `REPEAT` constructs in a procedural language, they are theoretically not needed [2]; however, it is frequently more convenient to use a concise program-graph instead of a single, but hopelessly complex, production that accomplishes the same task. Second, program-graphs are a special case of attributed graphs. Therefore, they can be built and edited by the standard graph-oriented machinery of `NETWORKS`. Finally, program-graphs are invoked exactly

the same way as graph-grammar productions: directly, from menu-selections, or indirectly, from other program-graphs.

4.2 Optimizing

Each node in a decision-tree, say x, represents a junction of alternatives whose value, $v(x)$, depends (recursively) on the values of the children-nodes that emanate from x. Recalling that $v(\cdot)$ represents an *expected*, rather than a determinate, value, it is clear that $v(x) > v(y)$ does not necessarily imply that x is a "better" prospect than y. For example, it might be that one of y's children represents a risky prospect whose potential value is significantly higher than any one of the values of x's children. In such a case, we say that x and y are incomparable, because their relative attractiveness depends on subjective risk-attitudes that vary from one decision-maker to another.

There exist situations, however, in which it is possible to compare two sibling-nodes, x and y, and conclude that x dominates y under all possible states of nature, and under all possible (Von-Neumann Morgenstern) utility functions. For example, consider the decision-tree depicted in the left of figure 9. Node a_1 dominates node a_2 because it represents a prospect whose most pessimistic outcome, 10, is better than the most optimistic outcome of a_2, which is 8. In a similar vein, a_4 dominates a_1, and a_5 dominates a_4. Note that neither a_3 nor a_5 dominates each other.

In this section we present graph-grammar productions for detecting and eliminating inferior sub-trees from a decision-tree graph. In order to compute dynamically the notion of dominance (or inferiority), we assign to each node in the tree two additional attributes, named `upperValue` and `lowerValue`, whose values are denoted hereafter $v^+(x)$ and $v^-(x)$, respectively. These values depend on the type of x, as follows. If x is an `outcome` node, we define $v^+(x) = v^-(x) \stackrel{def}{=} v(x)$; If x is a `chance` or a `choice` node with children-nodes $x_1, \ldots, x_n$, we define $v^+(x) \stackrel{def}{=} \max\{v^+(x_1), \ldots, v^+(x_n)\}$ and $v^-(x) \stackrel{def}{=} \min\{v^-(x_1), \ldots, v^-(x_n)\}$.

The "dominates" relation can now be defined as follows: let x and y be two sibling-nodes (children of the same parent). Node x is said to dominate node y ($x \succ y$) if $v^-(x) > v^+(y)$.

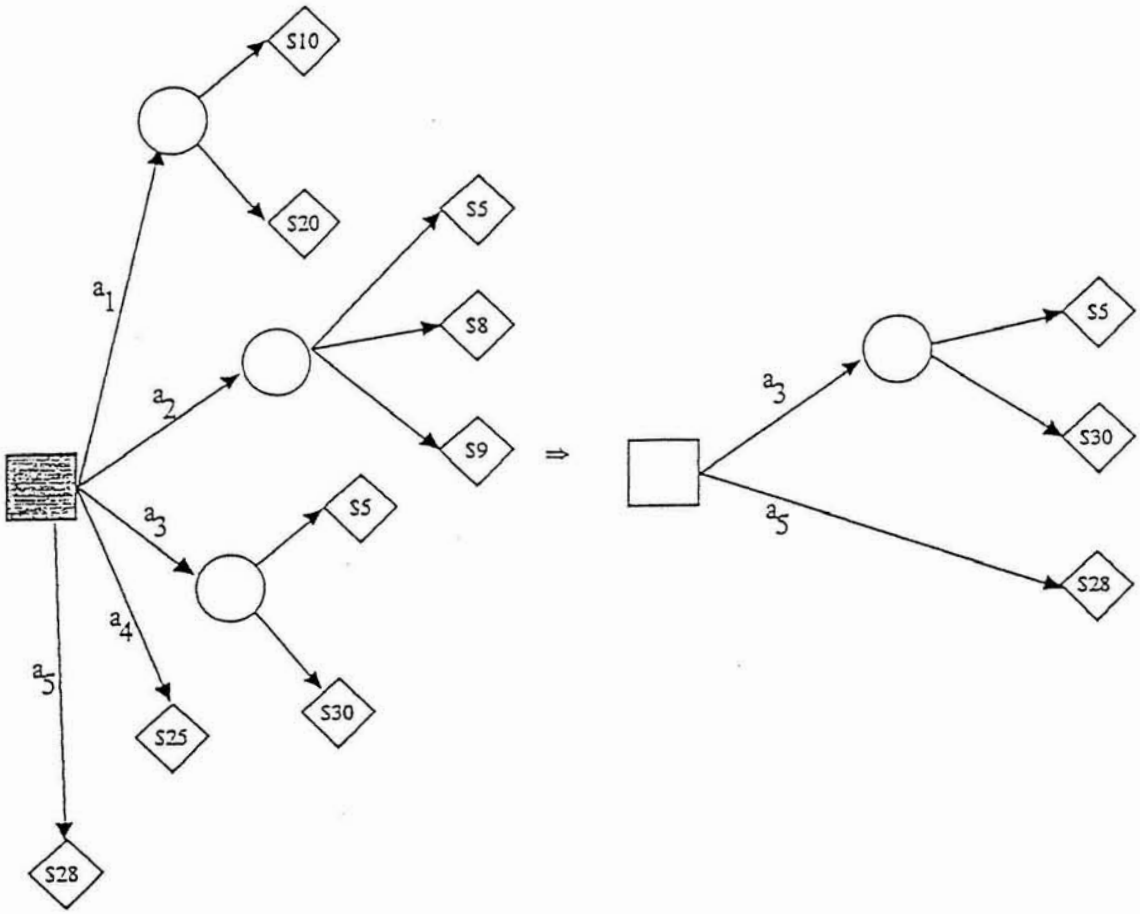

Figure 9: A decision-tree with inferior alternatives: a_1 dominates a_2, a_4 dominates a_1, and a_5 dominates a_4. The application of the OPT production to the shaded node would transform the left tree to the right tree.

It is easy to show that the $\succ$ relation is partial, irreflexive, transitive, and antisymmetric[5]. Hence, it forms a partial order on every set of sibling-nodes in a decision-tree graph.

The graph-grammar formulae for `upperValue` and `lowerValue` for a given `choice` or `chance` node x are as follows:

$$\texttt{upperValue}: \quad \texttt{imax}(\texttt{edge}(*,\_,\_,*,\texttt{X}),\texttt{na}(\texttt{upperValue},*,\texttt{X},\_)) \tag{6}$$

$$\texttt{lowerValue}: \quad \texttt{imin}(\texttt{edge}(*,\_,\_,*,\texttt{X}),\texttt{na}(\texttt{lowerValue},*,\texttt{X},\_)) \tag{7}$$

Given (6) and (7), the graph-grammar implementation of the relation $x_1 \succ x_2$ is as follows:

$$\texttt{na}(\texttt{lowerValue},*,\texttt{x1},\_) > \texttt{na}(\texttt{upperValue},*,\texttt{x2},\_) \tag{8}$$

We see that (8) will be `true` if and only if the `lowerValue` of x_1 is greater than the `upperValue` of x_2, implying that x_1 dominates x_2 (or, equivalently, that x_2 is inferior to x_1). Hence, the (8) predicate is essentially a *detector* of dominant and inferior nodes.

Once a set of nodes is found to be inferior in a certain sub-tree, it should be deleted from the graph. This operation is carried out by a production, named `OPT1`, and a program graph, named `OPT`, which are essentially identical to the pruning production `PC1` and the pruning program `PC`, respectively, from section 4.1 (see figure 8). The only difference between the two pairs is that the applicability-predicate of `OPT1` is (8), rather than (5).

When the user applies `OPT` to a selected node, say x, `OPT` begins its execution by applying `OPT1` to the same node. `OPT1` then tries to find a pair of children-nodes, x_1 and x_2, that satisfy (8). If such a pair is found, `OPT1` will proceed to delete x_2 from the target-graph, returning the value `true` to its calling environment – `OPT`. This will cause `OPT` to apply `OPT1` to x once again, until x contains no additional inferior edges.

[5]Proof: *Partial:* Let $v^+(x) = 3$, $v^-(x) = 2$, $v^+(y) = 4$, and $v^-(y) = 1$. The data are such that neither $v^-(x) > v^+(y)$, nor $v^-(y) > v^+(x)$. Hence, x does not dominate y and y does not dominate x. *Transitive:* assume that $x \succ y$ and $y \succ z$. Hence, $v^-(x) > v^+(y)$ and $v^-(y) > v^+(z)$. Now, by definition, $v^+(y) \geq v^-(y)$. Hence, we get $v^-(x) > v^+(y) \geq v^-(y) > v^+(z)$. Hence, $v^-(x) > v^+(z)$ and $x \succ z$. *Antisymmetric:* let $x \succ y$. Assume that $y \succ x$. Hence, $v^-(x) > v^+(y)$ and $v^-(y) > v^+(x)$. By definition, $v^+(x) \geq v^-(x)$, and $v^+(y) \geq v^-(y)$. Hence, we get $v^-(x) > v^+(y) \geq v^-(y) > v^+(x) \geq v^-(x)$, leading to $v^+(x) > v^+(x)$, which is a contradiction.

Since the production-graphs of `OPT` and `OPT1` are identical to those of `PC` and `PC1`, we will not repeat them here. To illustrate the execution of `OPT`, consider the decision tree graphs depicted in figure 9. If the user were to apply `OPT` to the shaded node in the tree on the left, he would end up with the tree on the right.

4.3 Consolidating

Decision-trees often contain a series of two or more consecutive nodes of the same type. A sequence of two `chance` nodes represents two consecutive random events. A sequence of two `choice` nodes represents two consecutive decisions. If a sequence of nodes of the same type provides an accurate description of the user's problem, then there is nothing wrong with it. For example, consider a `chance` node labeled `poll-results`, followed by a `chance` node labeled `election-results`. This sequence makes perfect sense in a certain context, and therefore it should not be altered. On the other hand, there exist situations in which a sequential presentation of nodes of the same type serves to befog, or even distort, an otherwise simple problem. Consider the following example:

> **Problem 3:** You may enter a two-stage game of chance whose outcome depends on the number drawn from spinning a roulette wheel which is numbered 1 to 100. In the first stage of the game, the wheel is spun. If the lucky number is in the range 1-16, you enter the second stage of the game. Otherwise, the game is over and you win nothing. In the second stage of the game, the wheel is spun again. If the lucky number is even, you win $500. Otherwise, you win nothing.
>
> **Problem 4:** You may enter a game of chance whose outcome depends on the number drawn from spinning a roulette wheel which is numbered 1 to 100. If the lucky number is in the range 1-8, you win $500. Otherwise, you win nothing.

Note that problems 3 and 4 (see figure 10) offer the same prospect – an 8% chance of winning $100. In spite of this normative equivalence, though, descriptive decision theory predicts that most people who are given this option would prefer to play game 3 on game 4. If this is indeed the case, the seller of the game could make it appear more attractive if he

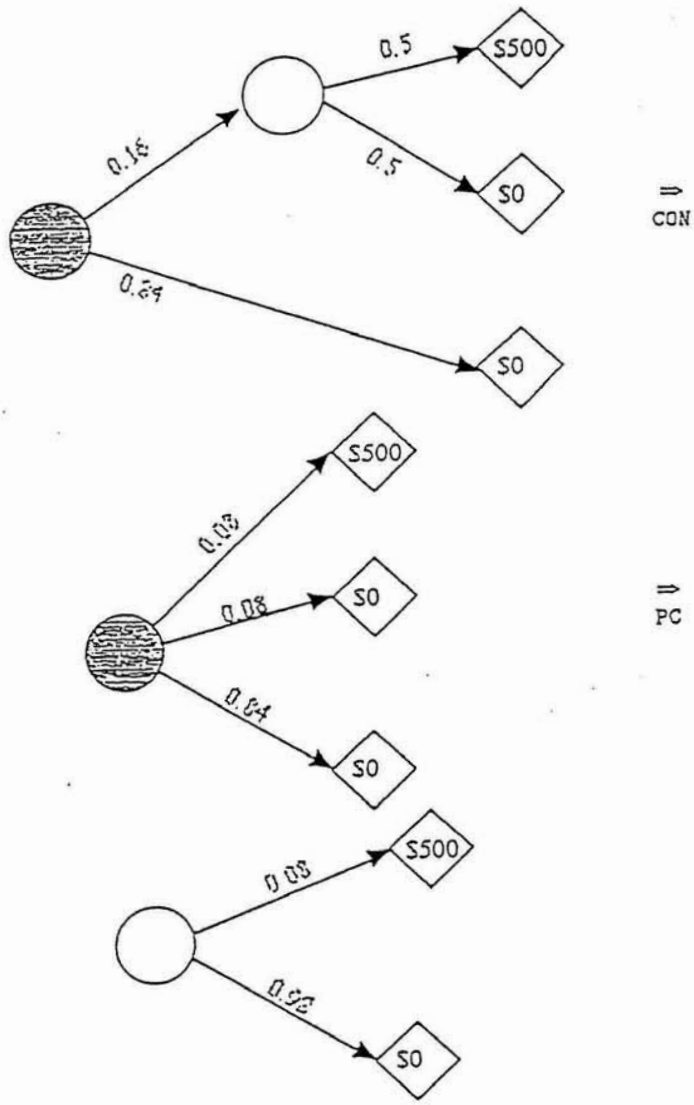

Figure 10: Problem 3 (top) and Problem 4 (bottom). The middle tree is an intermediate result, discussed in the paper.

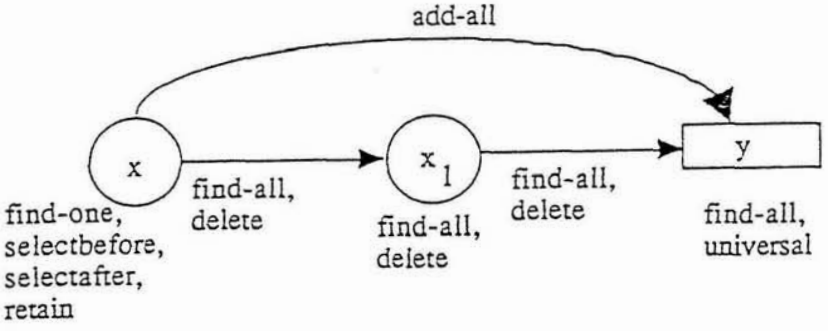

Figure 11: The consolidation production CON, designed to consolidate two consecutive chance nodes into a single chance node.

could cast it in terms of problem 3, rather than in terms of problem 4. Once again, we see that the frame of the problem – the topology of its decision-tree – plays a significant role in the user's decisions.

There may be several reasons why people prefer game 3 over game 4. First, it can be argued that game 3 is simply more interesting than game 4: although both games offer exactly the same value, the former game offers more excitement along the way. A more compelling explanation, inspired by Tversky and Khaneman's "pseudocertainty effect," [26] goes as follows. When people are presented with a sequence of two risky prospects, they tend to evaluate each prospect separately. As a result, the second prospect in game 3 is analyzed with a false feeling of certainty. In other words, the 50% chance of winning in the second stage tends to overshadow the fact that there is only a 16% chance of ever reaching that stage.

How can we eliminate, or at least reduce, the adverse impact of the pseudocertainty effect? Taking a graph-grammar approach, we provide the user with an optional production designed to detect paths that contain series of two or more consecutive chance nodes, and, if the user so desires, collapse them into single chance nodes. Formally speaking, let x be a node of type chance with n outgoing edges, leading to the children-nodes $x_1, \ldots, x_n$. Without loss of generality, assume that x_1 is also of type chance, and denote its children-nodes $x_{11}, \ldots, x_{1m}$. Our goal is to remove x_1 from the sub-tree rooted in x, and reconnect all of x_1's children directly to x. The probability value of each reconnected edge, (x, x_{1j}), $j = 1, \ldots, m$, should be set to the product $p(x, x_1) \cdot p(x_1, x_{1j})$ (the joint-probability that both x_1 and x_{1j} have occurred). We call this operation *consolidation*.

The production that carries out this transformation is called `CON` (figure 11). The production-graph consists of three nodes: x and x_1, representing the two connected chance nodes, and y, a typical child of x_1. Node x is marked `selectbefore`, `selectafter`, and `find-one`, indicating that (a) it must be selected before the production starts; (b) it will remain selected after the production has ended; and (c) only one such node is sought after in the target-graph. Node x_1 is marked `find-all` and `delete`, indicating that *all* such nodes should be found and then deleted from the target-graph. Note that the *types* of x and x_1 are the same. This constraint need not be specified explicitly, since both nodes appear as circles in `CON`'s production-graph.

The typical child of the deleted node is represented in the production-graph by y. The `universal` label indicates that the type of this node is immaterial for the production. The `add-all` label for edge (x, y) indicates that this is a *new* edge, to be added to the graph by the production. The value of the `probability` attribute of edge (x, y) should be set to the product of the `probability` attributes of the edges corresponding to (x, x_1) and (x_1, y) in the target-graph. This is achieved through the following attribute transformation expression:

$$\begin{aligned} &\texttt{ea(probability,*,(x,x1),echance)*} \\ &\texttt{ea(probability,*,(x1,y),echance)} \end{aligned} \tag{9}$$

The consolidating example (figure 10) illustrates how the "output" of one production can be piped into another production as "input." Denoting the trees in the figure from top to bottom t_1, t_2, and t_3, the overall graph manipulation can be described in terms of the chain $t_1 \xrightarrow{CON} t_2 \xrightarrow{PC} t_3$, or in terms of the functional form `t3=PC(CON(t1))`. Such combinations are possible because the inputs and outputs of all productions (as well as the productions themselves) are instances of the same thing – attributed graphs.

4.4 Reversing

Let t be a decision-tree. If t' is the decision-tree obtained from t by (a) pruning all the nodes of t (section 4.1); and (b) consolidating all the branches of t (section 4.3), then t' contains only alternating sequences of `choice` and `chance` nodes. In other words, the decision-tree that emerges from pruning and consolidating operations has the "normal" game-theoretic form of a 2-player game, in which a person (`choice` nodes) plays against nature (`chance` nodes). Typically, the order of the nodes is dictated by temporal constraints: player 1 makes the first move, player 2 makes the second move, player 1 makes the third move, and so on. In other words, the sequence of decisions and consequences unfolds in a fixed order which is determined by the rules of the game. There exist situations, however, in which there is a certain degree of latitude regarding the ordering of the moves. In these cases, the model builder would benefit from a production that enables him to *reverse* the direction of some nodes and edges in a sensible way, without violating the essential characteristics of the underlying decision problem.

Furthermore, note that most decision-trees are *not* cast, at least initially, in their normal form. As we mentioned elsewhere in the paper, many trees contain "genuine" sequences of chance-chance or choice-choice branches that the user may not want to consolidate, perhaps in order to preserve the original setting of the problem. Here, too, it may be desirable to reverse the order of some nodes, for two different reasons. First, node-reversal is a useful editing operation that comes handy in correcting or modifying the structure of an existing tree. Second, node-reversal is an effective analytic tool; with it, the user can create alternative frames of the same decision problem, gaining new insights into the problem's structure.

With that in mind, we seek to provide the user with four generic reversal operations, as follows:

1. reverse a `chance-chance` sequence
2. reverse a `choice-choice` sequence
3. reverse a `choice-chance` sequence
4. reverse a `chance-choice` sequence

Although the four operations are equally important from a functional standpoint, some are more interesting from a graph-grammar perspective. (1) is interesting because it involves recalculation of probabilities, using Bayes rule. (2) is a deterministic version of (1). (3) is interesting because it reverses nodes of different types, whereas (4) is the inverse of (3). Technically speaking, each one of the four operations represents a graph manipulation that is significantly more complex than what we've seen thus far in this paper. Therefore, and because of space limitations, we'll present here the implementation of one illustrative example – *reversing a chance-chance sequence*. The need for this operation can be motivated by the following example:

> **Problem 5:** A seasonal virus is known to infect one predisposed person out of every 100 people in the population. The virus causes a mild illness that lasts a few days. A new vaccine that completely eliminates the virus attack costs $100. You have just undergone a test which came out positive, indicating that you are predisposed. The test's hit-rate (positive result when the person is predisposed) is 80%. The test's false alarm-rate (positive

result when the person is not predisposed) is 20%. Will you purchase the vaccine?

The top left tree in figure 12 gives a compact description of the problem's data, showing clearly the *clinical* characteristics (type I and II errors) of the test. At the same time, the tree fails to answer the key question here, which is not clinical, but diagnostic, in nature: *what are the chances that I am predisposed, given that the test comes out positive?* In order to answer this question, one needs to transform the tree from its present, clinical, frame (top left tree in figure 12) into its dual, diagnostic frame (bottom left tree in figure 12). As we see, the chance of being predisposed if the test says so is strikingly low – less than 4%.

This is not to say that the top left tree in figure 12 is invalid or misleading. In its present frame, the tree serves the interests of one party, namely the vaccine's manufacturer. The other party involved – the prospective user who is considering taking the test – can learn nothing from the tree about the *actual* validity of the test. In order to make a reasoned decision, the user must reverse the sequence of the nodes `pre-disposed` and `test-results` to the sequence `test-results`, followed by `pre-disposed`. This reversal operation, which involves a delicate graph manipulation and an application of Bayes rule, is clearly beyond the bounded rationality of most decision-makers. Hence, an automated aid, in the form of a reversal production, is called for.

This is yet another example in which the given frame of a decision-tree does not lend itself to answering all the relevant questions that may be posed against it. In other words, even though the tree *contains* all the raw information necessary for reaching a reasoned decision, key parts of this information are implicit and not readily accessible.

The general case of node reversal is depicted in figure 13. Let t be a sub-tree with a root-node of type `chance`, denoted x. Node x has n outgoing edges, (x, y_i), $i = 1, \ldots, n$, each leading to a `chance` node y_i. The edges are parameterized by the probability distribution of X, i.e., by the set of values $P(x_i)$, $i = i, \ldots, n$. Each of the y_i's has m outgoing edges, (y_i, z_{ij}), $j = 1, \ldots, m$, leading to a node z_{ij} which may be of any type. Note that each of the y_i's represents the *same* random variable – Y – whose probability distribution is conditioned on the occurrence of the random event X. Hence, each edge (y_i, z_{ij}) is parameterized by the conditional probability $P(y_j|x_i)$, $j = 1, \ldots, m$, $i = i, \ldots, n$. For the sake of brevity, we denote this tree $t = (q, x, y, z)$ (q and z are nodes of any type, whereas x and y must be of type `chance`). With this notation, the goal of the reversal operation is to transform the

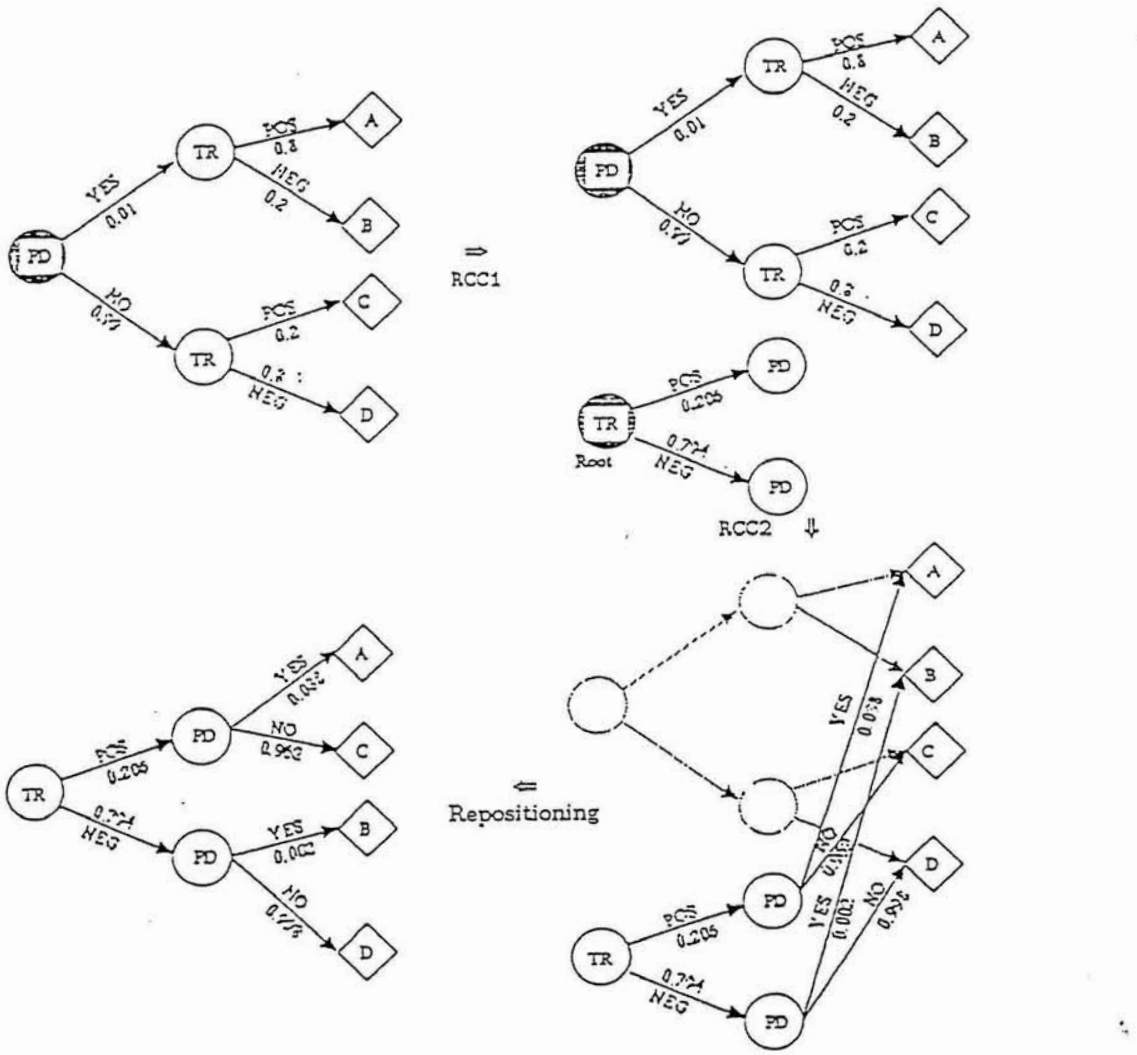

Figure 12: Example of the application of program-graph RCC (figure 14) to reverse a chance-chance sequence. PD and TR refer to the random variables predisposed and test-result, respectively (problem 5). Shaded nodes are nodes that have been selected, either by the production or by the user. Shaded edges (lower right) represent edges that have been deleted by the production.

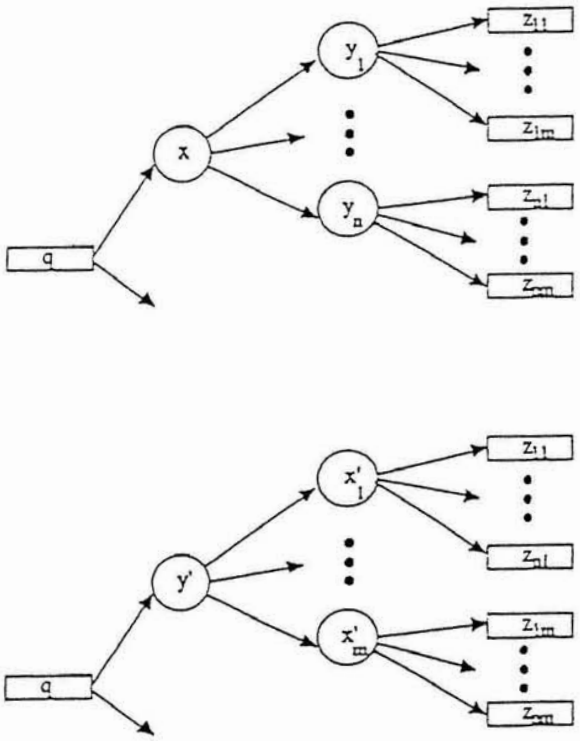

Figure 13: The goal of the reversal operation is to reverse the order of two consecutive chance nodes, i.e. to transform the top tree, denoted (q, x, y, z), to the bottom tree, denoted (q, y', x', z). q and z may be of any type, whereas x and y are assumed to be of type chance.

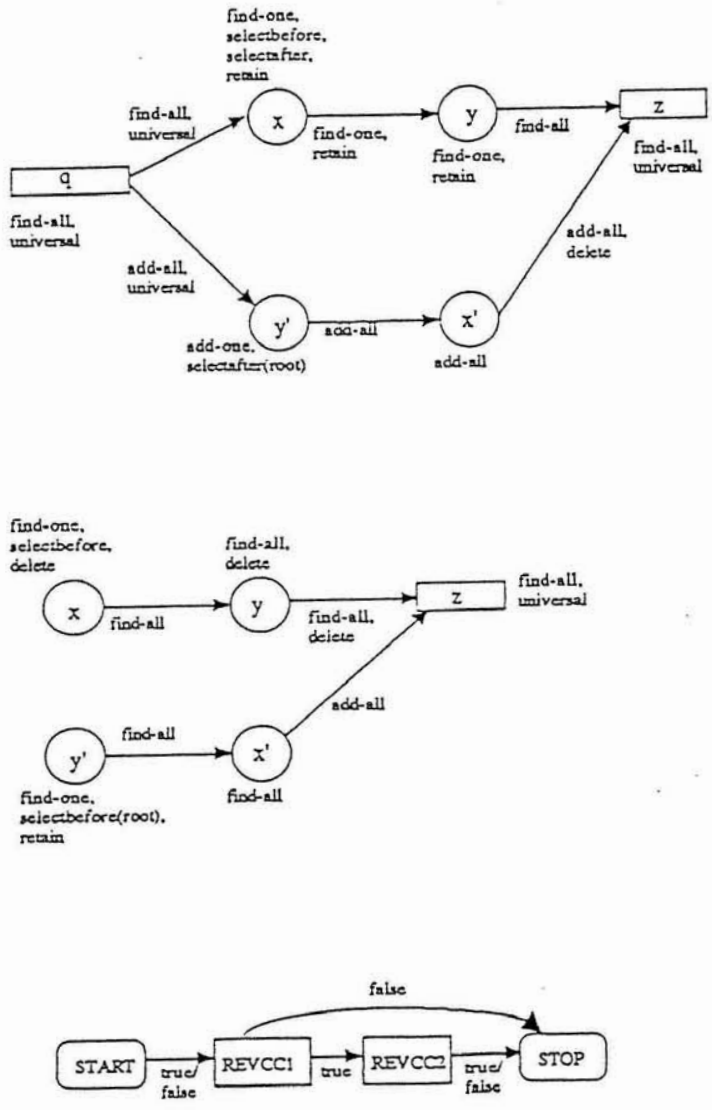

Figure 14: The RCC program (bottom), designed to reverse a chance-chance sequence. This program consists of two productions, RCC1 (top), and RCC2 (middle) executed in sequence.

sub-tree $t = (q, x, y, z)$ into the sub-tree $t' = (q, y', x', z)$ and, of course, carry out all the necessary probability calculations implied by the reversal. It's important to remember that t will typically be embedded in a larger tree, adding to the complexity of this manipulation.

The overall reversal operation is carried out by a graph-grammar program, named RCC, which applies two productions, named RCC1 and RCC2, to the target-graph (see figure 14). We assume that before the program has been invoked, the user has selected a certain target-node, denoted $\overline{x}$, as the operation's anchor, or "pivot." If $\overline{x}$ has no child node of type chance, the RCC1 production will fail to match its left hand side on the target-graph. As a result, the production as well as the RCC program will terminate their execution (see bottom of figure 14), and the tree will remain intact. If RCC1 succeeds to match the sequence x, y on the target-graph, it will proceed to create a reversed copy of this pair, denoted (y', x') in the production-graph. The second production – RCC2 – links the edges that immanent from the newly created node x' to the children of the old node y, and then deletes the old sequence x, y from the target-graph.

The logic of the productions RCC1 and RCC2 is rather simple, but they involve complex probability calculations that require several graph grammar tricks that we haven't seen yet. In order to avoid clutter, we delay the step-by-step description of these productions to a technical appendix. Readers who are not interested in the details of graph-grammar programming can skip this material without losing the main thread of the paper.

5 Conclusion

This section summarizes our work along the two dimensions that characterized the entire paper: graph-grammars (engineering) and decision theory (application).

Graph Grammars as a Modeling Tool: Generic families of models (like decision trees) can be built and manipulated in two different ways: through general-purpose languages, like Pascal or C, or through dedicated packages, like Arborist [1] or Supertree [16]. Each implementation vehicle offers a different set of pros and cons. General-purpose languages are flexible, but hard to use, whereas specialized packages are user-friendly, but functionally limited. In this paper we presented an interim solution to model building, in the form of a

GBMS (Graph-Based Modeling System). We argue that the GBMS approach offers both the flexibility of free-form programming, on the one hand, and the predictability and ease of use of specialized modeling environments, on the other.

In particular, the graph-grammar approach to modeling offers the following tangible benefits. *Flexibility*: graph-grammars are Turing-complete, meaning that they are just as powerful as any general-purpose programming language. *Formality*: the use of graph-grammars enables us to define all the permissible manipulations on a certain family of models precisely and unambiguously, using a mathematical language. *Executability*: the graph-grammar specifications can be implemented on a computer, so that model definitions can be directly executed. *Elegance:* in a graph-grammar, both graphs and operations on graphs are defined in terms of a uniform language – graphs. *Modularity:* New productions can be easily defined to manipulate models in ways not envisioned by the original designers. *Generality:* The graph-grammar formalism is a general-purpose modeling tool; it can be used to construct *any* attributed-graph, not just decision-trees. Hence, decision-tree models built in a GBMS can be archived and managed along with other graph-grammar models, forming a "model-base."

The latter point is quite important. In addition to its ability to support the construction and manipulation of decision-trees, NETWORKS can be applied to many other modeling domains, e.g. influence diagrams, game trees, and mathematical modelling [9]. Moreover, the system allows models from different paradigms not only to coexist, but also to interact. For example, the contents of a value attribute of an outcome node in a decision tree model might be calculated by another model, e.g., the optimal objective function of a linear programming problem. The link between the two models can be easily established, as the attributes of one graph are allowed to refer to attributes in any other graph in the model-base. This connectivity, along with the ability to work on different models in multiple windows, enable the implementation of many ideas in model management that up to now were considered quite abstract.

The decision tree "package" that resulted from this research was implemented in NETWORKS in about one week. NETWORKS runs on a Macintosh computer model II and requires at least 4 MB of main memory. It is written on AAIS Prolog, and can interface with native Prolog code.

Decision Theory: One fundamental requirement in the normative theory of decision making is that the preferences of rational persons should be independent of problem description. The expected utility model and the theory of subjective probability provide mathematical tools that are completely devoid of any "graphical" or "presentation" contexts. However, numerous studies on *actual* (rather than *normative*) decision making under uncertainty revealed a persistent and predictable framing effect. Tversky and Khaneman, who studies this phenomenon in detail, have summarized their findings as follows:

> "Individuals who face a decision problem and have a definite preference (i) might have a different preference in a different framing of the same problem, (ii) are normally unaware of alternative frames and of their potential effects on the relative attractiveness of options, (iii) would wish their preferences to be independent of frame; but (iv) are often uncertain how to resolve detected inconsistencies. In some cases, the advantage of one frame becomes evident once the competing frames are compared, but in other cases it is not obvious which preferences should be abandoned." [27]

We argue that this passage should motivate the development of a new breed of decision support systems – systems that not only support the technical aspects of building decision models, but are also sensitive to the cognitive limitations and biases that creep into their normal use. The "intelligent" decision tree package presented in this paper is a step in this direction. In addition to the standard functions of building, editing, and analyzing decision trees, we have developed a library of productions that enable the user to manipulate decision trees and create alternative frames of the same problem, thus gaining more insight into the problem's structure and into the decision maker's own set of preferences.

To summarize, the reframing productions that were presented in this paper fall into four categories: pruning, consolidating, optimizing, and reversing. *Pruning* consists of removing superfluous nodes and edges from an otherwise well-structured tree. *Consolidating* is the act of collapsing a branch of two or more nodes of the same type into a single node. *Optimizing* consists of removing sub-trees that have no impact on the optimal choice path. *Reversing* deals with altering the order of nodes and edges in the tree. One additional operation that we haven't implemented yet can be termed *combining* – the act of merging two separate decision trees into a single tree. This graph manipulation will be a powerful debiasing mechanism, as it would allow users to overcome a natural inability to analyze

concurrent decisions, a bias which was reported and analyzed in [27]. Combining graphs is also a challenging operation from a graph-grammar perspective, and we intend to report about it in future work.

6 Appendix: Node Reversal Productions

This appendix describes the productions RCC1 and RCC2, whose respective graphs are depicted in figure 14. These productions are designed to reverse the order of two consecutive chance nodes in a decision tree graph – a transformation which is depicted symbolically in figure 13. With that figure in mind, the RCC1 production performs the following operations:

1. Create the node y'.
2. Create the nodes x'_j, $j = 1, \ldots, m$. (Each of the m new nodes is a copy of the old x node.)
3. Create the edges (y', x'_j), $j = 1, \ldots, m$
4. Set the probability attribute of each edge (y', x'_j) to $P(y'_j)$ via the formula $\sum_{i=1}^{n} P(y_j|x_i)P(x_i)$.

The second production, RCC2, performs the following operations (see figure 13 and bottom right of figure 12):

1. For each new node x'_j, $j = 1, \ldots, m$, create a new set of edges, (x'_j, z_{ij}), $i = 1, \ldots, n$.
2. Set the probability attribute of each edge (x'_j, z_{ij}) to $P(x'_i|y'_j)$ via the formula $P(y_j|x_i) \cdot P(x_i)/P(y'_j)$, $j = 1, \ldots, m$, $i = 1, \ldots, n$
3. Delete the old x node and its outgoing edges (x, y_j), $j = 1, \ldots, m$.

Production RCC1: The production (top of figure 14) begins its operation by looking for an edge $(\overline{x}, \overline{y})$ in the target-graph such that $\overline{x}$ was selecetd by the user and both $\overline{x}$ and $\overline{y}$

are of type chance. The edge and its two end-nodes are labelled `find-one` and `retain`, since only one instance of them should be found in the target-graph, and they should not be deleted (at least temporarily, as we'll see shortly). Node x is labelled `selectbefore`, since it must have been selected before the production can proceed, and `selectafter`, in preparation for the second production (RCC2).

After x and y have been matched with corresponding nodes in the target-graph, the production proceeds to add a new node, denoted y', to the graph. The node is labeled `selectafter(root)` – a label which will help the next production (RCC2) distinguish between the old root x and the new root y', which are both selected (`root` is an arbitrary label chosen by the production designer). Since at the end of the reversal operation y' must inherit the parent of x, both nodes are connected to q (a node of any type, i.e. `universal`) in the production-graph.

The node q and its outgoing edges are labeled `find-all` and `add-all`, rather than `find-one` and `add-one`, because of a subtle contingency. The problem is that $\overline{x}$ might be the root of the overall decision tree, in which case q will not match any node in the target-graph. In such an event, if q were labeled `find-one`, the production would fail to match and thus terminate its operation. The `find-all` label, on the other hand, is more liberal, as it instructs the production to seek 0, 1, or more such nodes in the target-graph.

Recalling that nodes x and y represent two random variables, our next task is to link every possible outcome (outgoing edge) of y to an identical copy of x. The reversed nodes are denoted y' and x', and the reversal logic is carried out by the three edges (y,z), (y',x'), and (x',z). The labels of these edges cause the production to (a) enumerate (find-all) the possible contingencies of y, copy them (add-all) to y', and then attach their corresponding outcomes to x'. The curious pair of labels (`add-all,delete`) which marks (x',z) forces the system to add one copy of x' to each (found) outcome of y.

The value of the `label` attribute of edge (y',x') is set to the value of the `label` attribute of edge (y,z) by the attribute transformation expression:

$$\texttt{ea(label,*,(y,z),echoice)} \tag{10}$$

Finally, the value of the `probability` attribute of (y',x') (for a certain outgoing edge j), which represents $P(y'_j)$, should be calculated according to the following formula:

$$P(y'_j) = \sum_{i=1}^{n} P(x_i)P(y_j|x_i) \tag{11}$$

The graph-grammar implementation of this formula is as follows:

```
sum((edge(*,Edge1,echance,x,Y),
     edge(*,Edge2,echance,Y,Z),
     ea(label,*,Edge2,echance)=ea(label,*,(y',x'),echance)),
    ea(probability,*,Edge1,echance)*ea(probability,*,Edge2,echance))
```

In shorthand, this is essentially a `sum(P,V)` function in which the selector `P` consists of the three boolean conjuncts (`edge`, `edge`, and `ea=ea`, and `V` is the numeric expression `ea * ea`. The function sums up all the `ea * ea` values for which the three conjuncts are true, - where:

1. The first conjunct, `edge(*,Edge1,echance,x,Y)`, finds a child `Y` of `x`. This corresponds to x_i.
2. The second conjunct, `edge(*,Edge2,echance,Y,Z)`, finds a child `Z`, of the `Y` which was found above. This corresponds to $y_j|x_i$.
3. The third conjunct insists that the `label` of the edge (y, z) (`Edge2`) be equal to that of (y', x'). Recalling that we are calculating $P(y_j)$ for some j, this conjunct assures that we only use those conditional probabilities, $P(y_k|x_i)$, for which $k = j$.
4. Given the nodes and edges that the selectors have matched, the expressions `ea(probability,*,Edge1,echance)` and `ea(probability,*,Edge2,echance))` correspond to the probabilities $P(x_i)$ and $P(y_j|x_i)$, respectively.

Hence, the overall `sum` function evaluates to the value of 11.

Production RCC2: Once production RCC1 has completed, a new node, represented by y', will have been added to the graph. In addition, new edges will have been added to the graph, leading from y' to x', one edge for each possible outcome of y. Furthermore, the probability values of each of those edges (the $P(y_j)$'s) will have been calculated. What remains to be accomplished is to connect the x's to the children of the original y. This operation is carried out by RCC2.

If the selected nodes x and y' are matched in the target-graph, the production deletes the former and retain the latter. Next, the production proceeds to match the edges (y, z), (x', z), and (y', x'). The `find-all` and `add-all` labels ensure that *all* the children of y (the z's) will be reconnected to x', subject to the requirement that the edge (y, z) has the same label as the edge (y', x'). This is forced through the applicability predicate:

$$\texttt{ea(label, *, (y, z), echance)} = \texttt{ea(label, *, (y', x'), echance)} \tag{12}$$

Next, the labels that emanate from node x' are bound to the original labels that emanate from x. This is done by setting the `label` attribute of the edge (x', z) to the following attribute transformation expression:

$$\texttt{ea(label, *, (x, y), echance)} \tag{13}$$

Finally, the `probability` attribute of (x', z), which corresponds to $P(x_i|y_j)$ for some i and j, must be calculated through Bayes rule: $P(x_i|y_j) = P(x_i)P(y_j|x_i)/P(y_j)$. Now, the values $P(x_i)$, $P(y_j|x_i)$, and $P(y_j)$ are already stored in the graph in the `probability` attributes of the edges (x, y), (y, z), and (y', x'), respectively (the latter was computed by the RCC1 production). Hence, RCC2 computes the probability of (x', z) through the following attribute transformation expression:

$$\begin{aligned}&\texttt{ea(probability, *, (x, y), echance)*}\\ &\quad\texttt{ea(probability, *, (y, z), echance)/}\\ &\qquad\texttt{ea(probability, *, (y', x'), echance)}\end{aligned} \tag{14}$$

Since the topology of the production takes care of all the necessary matchings, the indices associated with each of these edges match up automatically, and there is no need to write any explicit matching predicates (as we have done in RCC1).

References

[1] Arborist decision-tree Software, Texas Instruments, Inc., PO Box 2909, Mail Station 2240, Austin, TX 78769.

[2] Bunke, H. 1982. On the generative power of sequential and parallel programmed graph-grammars. *Computing* **29**, 89-112.

[3] Göttler, H. 1979. Semantical description by two-level graph-grammars for quasihierarchical graphs. *Applied Computer Science* **13**, 207-209.

[4] Göttler, H. 1983. Attributed graph-grammars for graphics. In *Graph-Grammars and their Application to Computer Science*, pp. 130-142. H. Ehrig, M. Nagl, and G. Rozenberg (eds.), *Lecture Notes in Computer Science* **153**, G. Goos and J. Hartmanis (series eds.) Springer-Verlag, Berlin.

[5] Göttler, H. 1987. Graph-grammars and diagram editing. in *Graph-Grammars and their Application to Computer Science*, Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A. (Eds.), Springer-Verlag, Berlin.

[6] Hopcroft, J. E. and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Massachusetts.

[7] Howard, R.A. 1968. The Foundations of Decision Analysis. *IEEE Transactions on Systems Science and Cybernetics,* SSC-4, 211-219.

[8] Hutchins, E. L., Hollan, J. D. and D. A. Norman. 1986. Direct Manipulation Interfaces, in *User Centered System Design: New Perspectives on Human-Computer Interaction*, Norman, D. A. And S. W. Draper (eds.), Lawrence Erlbaum, Hillsdale, New Jersey, 87-124.

[9] Jones, C. V. 1989. An Example Based Introduction to Graph-Based Modeling, *Proceedings of the Twenty-Third Annual Hawaii Conference on the System Sciences*, Kona, HI, pp. 433-442.

[10] Jones, C. V. 1990. An Introduction to Graph-Based Modeling Systems, Part I: Overview. *ORSA Journal on Computing*, **2**:2, 136-151.

[11] Jones, C. V. 1990. An Introduction to Graph-Based Modeling Systems, Part II: Graph-Grammars and the Implementation, forthcoming, *ORSA Journal on Computing.*

[12] Jones, C. V. 1990. An Integrated Modeling Environment Based on Attributed Graphs and Graph-Grammars, forthcoming *Decision Support Systems.*

[13] Lavalle, I.H. 1978. *Fundamentals of Decision Analysis*, Holt, Reinhart & Winston, Inc., New York.

[14] Lavalle, I.H., and Fishburn, P.C. 1987. Equivalent decision-trees and Their Associated Strategy Sets. *Theory and Decision*, **23**, 37-63.

[15] Lavalle, I.H., and Wapman, K.R., 1986. Rolling back Decision Trees Requires the Independence Axiom. *Management Science* **32**: 382-385.

[16] McNamee, P. and J. Celona, 1987. *Decision Analysis for the Professional with Supertree*, Scientific Press, Palo Alto, CA.

[17] Nagl, M. 1976. Formal languages of labelled graphs. *Computing*, **16**, 113-137.

[18] Nagl, M. 1987. Set theoretic approaches to graph grammars. In *Graph-Grammars and their Application to Computer Science*, Ehrig, H., Nagl, M. Rozenberg, G. and A. Rosenfeld (eds.), 41-54, Springer-Verlag, Berlin.

[19] Raiffa, H. 1968. Decision Analysis. New York: Random House, p. 129.

[20] Reps, T. and T. Teitelbaum. 1984. The synthesizer generator, *Proc. ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments. ACM Sigplan Notices*, **19**:5, 42-48.

[21] Shneiderman, B. 1983. Direct manipulation: a step beyond programming languages, *IEEE Computer*, **16**:8, 57-69.

[22] Simon, H.A., *Q.J. Econ.* **69**,99 (1955), *Psychol. Rev.* **63**, **129** (1956).

[23] Suppowit, K. J. and E. M. Reingold. 1982. The complexity of drawing trees nicely. *Acta Informatica*, **18**, 377-392.

[24] Tamassia, R., Di Battista, G., and C. Batini. 1988. Automatic graph drawing and readability of diagrams. *IEEE Trans. on Systems, Man and Cybernetics*, **18**:1, 61-79.

[25] Thompson, G.L. 1972. Simplification of games in extensive form. *Internat. J. Game Theory* **1**, 147-159.

[26] Tversky, A. and Khaneman, D. 1974. Judgement Under Uncertainty: Heuristics and Biases. *Science* **185**, 1124-1131.

[27] Tversky, A. and Khaneman, D. 1981. The Framing of Decisions and the Psychology of Choice. *Science* **211**, 453-458.

[28] Vaucher, J. G. 1980. Pretty-printing of trees. *Software Practice and Experience*, **10**, 553-561.

[29] Warfield, J. N. 1977. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man and Cybernetics*, **SMC-7**, 505-523.