TOOL-USE AND TOOL-BUILDING: COMBINING DECISION SCIENCE TOPICS AND COMPUTER PROGRAMMING SKILLS IN A DSS COURSE

Dr. Michel Mitri, James Madison University, mitrimx@jmu.edu

ABSTRACT

There is utility in applying rigorous programming practice to DSS courses, combining object-oriented techniques, data structures, and decision science into a comprehensive pedagogical structure. By integrating classroom discussion of decision models (concept delivery), laboratory work using DSS tools to solve decision-theoretic problems (tool-use), and additional lab work using programming techniques to build DSS tools and implement their underlying decision models (tool-building), students are exposed to the mathematical foundations of decision science, gain experience applying them to realistic problems, and hone programming skills by implementing decision algorithms in an object-oriented framework.

Keywords: decision support systems, object-oriented programming, data structures, tool-use, tool-building.

INTRODUCTION

DSS courses are interdisciplinary in nature. In a business school, they are typically housed within a CIS or MIS department. Yet, much of their content involves topics taught in management science or decision science curricula, such as expected value/utility, multi-criterion decision modeling, linear and nonlinear optimization. AI topics are also discussed in DSS courses; drawn from psychology, logic, biology, software engineering, evolutionary science, and probability.

Due to this diversity of background, and differences in interests among DSS instructors, assigned lab work in these courses vary considerably. The author conducted a search of on-line DSS syllabi in U.S. universities to ascertain the software tools most commonly used in DSS courses. Via Infoseek and Google search engines, with the search criteria "Decision Support System" and "Syllabus", the search produced over 400 hits. These were pruned to include only syllabi from U.S. business school curricula of four-year and graduate institutions from the last five years (1997 – 2001), resulting in a final list of 47 syllabi. The syllabi were scanned to determine the software used for lab assignments. Table 1 shows the result of this scan.

The most common lab assignments involve use of spreadsheet software, followed closely by various types of expert system shells. Many assignments involve use of add-on software for spreadsheets. There are also some programming assignments (usually in Visual Basic); these are mostly extra coding to provide added functionality or user interfaces in support of spreadsheet or database applications. Various genetic algorithm and neural network optimization assignments are also included. Use of decision analytic software and AHP is common. In some rare occasions, students do not perform any lab work (only written assignments); however, usually students are required to make extensive use of DSS tools, and sometimes supplement the tools with program code to enhance functionality or user interfaces.

Type of Software Tool	#Occurrences
Spreadsheet	23
Expert System Shells (VPExpert, Exsys, M4, 1stClass, CLIPS, Level5Object, other RBS)	19
Spreadsheet Add-ons (@Risk, Crystal Ball, DPL, GA, Generator)	12
Decision Analysis Software (@Risk, DPL, Precision Tree, CB-Predictor, SMILE)	10
Programming (VB/VBA/Other)	9
Database	7
AI Optimization (NN, GA, Braincel, GA, GAMS, Generator)	6
AHP (Expert Choice)	5
Web Tools (HTML, Javascript)	5
Written Work Only (No Lab)	5
Financial Analysis (IFPS)	4
Unspecified Lab Work	4
Linear Programming Tools (NEOS, SAS/OR)	2
Other Software (Visionquest GDSS, AIMMS)	2

Table 1: A ranking of the most commonly used software tools for laboratory assignments in DSS courses.

However, none of the courses use object-oriented programming (OOP) techniques; nor do they require students to *create* the DSS generators themselves. This is understandable for MIS curricula that do not teach programming skills in their major (i.e. with a managerial focus). But, surprisingly, even CIS programs with OOP and data structures classes do not emphasize OOP in DSS courses. This is unfortunate, because much can be gained by requiring students to implement the underlying algorithms of decision models. Writing a program to implement decision models exposes the student to the detailed algorithm of the model, and gives an appreciation for how computers carry out these calculations. Programming such non-trivial applications gives students valuable practice to hone their software development skills. Applying these programming skills to DSS techniques encourages students to recognize connections between software design and decision science.

In addition, consider that many decision models and AI techniques specialize abstract data structures such as trees and graphs. In particular, the general tree data structure is the foundation for a wide variety of models, including decision trees, AHP, backward chaining in rule-based systems, game search, induction-based machine learning, frame-based architectures, and case-based reasoning. Students who write programs to implement these algorithms gain first-hand experience in applying these data structures to real-world, challenging, and interesting problems. This has further implications when done in an object-oriented framework. Students who build a general tree-node class (with parent-sibling links and recursive depth-first search routines) can then create subclasses to implement the specialized nodes for the specific decision models they are assigned to create. Thus, they use many object-oriented techniques such as inheritance, polymorphism, and encapsulation.

In conclusion, much can be gained by combining the *tool-use* flavor of most DSS lab work with actual programming of model algorithms; i.e *tool-building*. Tool-use gives students experience in developing applications based on a particular model. Tool-building gives students experience in creating the model itself.

The author uses this approach in delivery of DSS courses, and encourages others to do the same. The approach consists of the following steps:

1) Give students a programming assignment to create a general tree-node class with recursive depth-first search implementation.
2) For some number of decision-models to be covered in class:
   a) In the classroom, describe the decision model, its underlying theory, and its mathematical, logical, and/or algorithmic structure.
   b) Provide for the students a software tool with a graphical user interface (GUI) and assign them the task of creating applications using the tool (i.e. a *tool-use* assignment)
   c) Give students programming assignments to implement the decision model as a subclass of the generic tree-node class (i.e. a *tool-building* assignment).

With this approach students can write three or four programs in a typical semester. This approach will only work if students have OOP experience prior to taking the DSS class, and preferably if they have experienced programming the general tree data structure with recursive tree search. The following sections illustrate the above approach with an *expected utility decision tree*.

BUILDING A GENERAL TREE NODE CLASS AND APPLICATION

The general tree node is constructed using the typical left-child, right-sibling implementation. Each tree node includes a pointer to the first *child* and a pointer to the next *sibling*, as shown in figure 1. The tree-**node** class contains the members shown in figure 2.

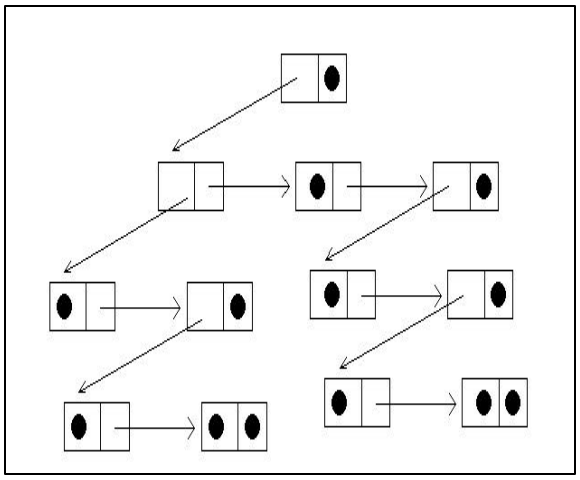

Figure 1: the general tree with left-child, right-sibling implementation

```
class node{
    private:                                  // data members
      char label[30];
      node* parent;                           // class composition
      node* left_child;
      node* right_sibling;
    public:                                   // member functions
      node();   ~node();
      void delete_links();
      virtual node* add_child();              // polymorphic
      node* find_node(char* lbl);  // recursive
      void link_child(node* child);
      void display_subtree(int);              // recursive
      virtual void display_extra_stuff();   // polymorphic
};
```

Figure 2: the class definition for the general tree-node

With this assignment, students practice object-oriented programming techniques including creation of member variables and member functions, polymorphic behavior, encapsulation, data hiding and class composition. Students also experience recursive functions and data structure concepts. Following this assignment, students start to learn specific decision models. They learn to use the conceptual framework of the model, get experience using tools that implement the model, and finally write programs (i.e. create classes that subclass the tree node) for implementing the model themselves. One example, an *expected-utility model*, is described in the following sections.

TEACHING EXPECTED UTILITY MODELS USING DECTREE

Probabilistic decision trees are commonly used for making decisions in the face of uncertainty. These tools have been used for applications in areas as diverse as project management, medical diagnosis, manufacturing, environmental risk assessment, and many other domains. The ubiquity of the decision tree model is evidence of its usefulness as a decision aid (1, 2, 3, 5).

Expected utility (EU) theory is a probabilistic approach for planning under uncertainty. It measures the expected utility (goodness) of an action, taking into account the probabilities of each of the consequences and their utilities.

Given a possible action to take *A* and a set of possible consequences $C_1 \ldots C_n$ of that action, the expected utility of *A* is calculated as:

$$EU(A) = \Sigma\, p(C_i|A) \times U(C_i)$$

where $p(C_i|A)$ is the probability of the *i*th consequence, given the action, and $U(C_i)$ is the utility (measure of goodness/badness) of that consequence.

Probabilistic decision trees form a graphic implementation of expected utility models. Several commercial applications exist that implement decision trees, including DPL, Precision Tree, and @Risk. For this DSS class, students use DECTREE, which provides a user-friendly format for implementation of decision tree models (4). DECTREE gives students an understanding of EU modeling and shows the desired behavior of the program that they will be assigned to write. With DECTREE, students construct the main components of EU decision trees, which include decision nodes (with choice branches), chance nodes (with outcome branches), and utility nodes (with utility expressions and variables).

A decision node represents a set of choices facing the decision-maker. Each decision node has a number of choice branches emanating from it. The DECTREE program automatically calculates expected utilities for each of these choice branches based on the above equation. This calculation is done via recursive tree-search, as described in the following section. Decision nodes in DECTREE are represented graphically with rectangular borders. The user assigns labels to the decision points and to each of the choice branches.

A chance node represents a point at which uncontrollable events may happen. These nodes have rounded borders, and can be labeled by the user. Branches from the chance nodes represent possible outcomes. Users assign labels and probability values to each of the chance nodes. The user can change these probability values at any time and study the effects of these changes on the expected utility values for all possible choices in the model.

A utility node represents the end point of a path of decisions and outcomes in a decision tree. Utility nodes contain values representing a measure of goodness or badness of the final outcomes of the path. The user enters, as the label of the utility node, a numeric value, a variable name, or a mathematical expression combining arithmetic operators, numbers, and/or variables. DECTREE calculates the value of a utility node based on the arithmetic expression entered by the user, then propagates this value up the tree in a recursive fashion according to the expected utility formulation and the probabilities of the chance nodes in the ancestral path of the tree.

Our company is trying to determine whether or not to build an E-Commerce web site. We are also trying to determine whether or not it is cost effective to do market research *prior to* deciding whether or not to build this web site. There is a cost of doing market research...this should be a utility variable. The market research will either result in favorable or unfavorable findings. The probabilities are 50% for favorable and 50% for unfavorable.

- A-priori (with no research conducted), there is a 50% probability that building the web-site will result in success, and 50% that building the web-site will result in failure.
- If market research results with favorable findings, there is 80% probability that building the site will result in success, and 20% that building the site will result in failure.
- If market research results in unfavorable findings, there is 20% probability that building the site will result in success, and 80% that building the site will result in failure.
- If the site is built, and is successful, it will bring the company a "Success Profit" (this is also a utility variable.
- If the site is built, and is a failure, it will cost the company a "Failure Loss" (this is also a utility variable)
- If no site is built, there are no costs or benefits that come to the company. (Utility value of zero).

Analyze the model under the following scenario:

- Market research cost is $150,000
- Successful web-site building brings in $2,000,000
- Unsuccessful web-site costs the company $1,000,000

Given the scenario described above, answer the following two questions: first, should market research be conducted, and second, should the web-site be built? Finally, by playing around with the profit-for-successful-plant variable, identify the profit required for reaching a break-even point for the decision to do market research prior to building.

Figure 3: A sample case for use with DECTREE

In the DSS course, students construct two decision trees. One is based on a case that is given them (see figure 3). The second decision tree solves a utility problem of their choice. The decision tree (shown in the DECTREE software) for the above scenario is shown in figure 4. Decision nodes are for doing market research and building the web site. Chance nodes are for the outcomes of market research and for the success/failure of the web-site. Utility nodes are the lowest-level nodes in the tree and involve utility variables for success-profit, failure-loss, and market-research-cost.

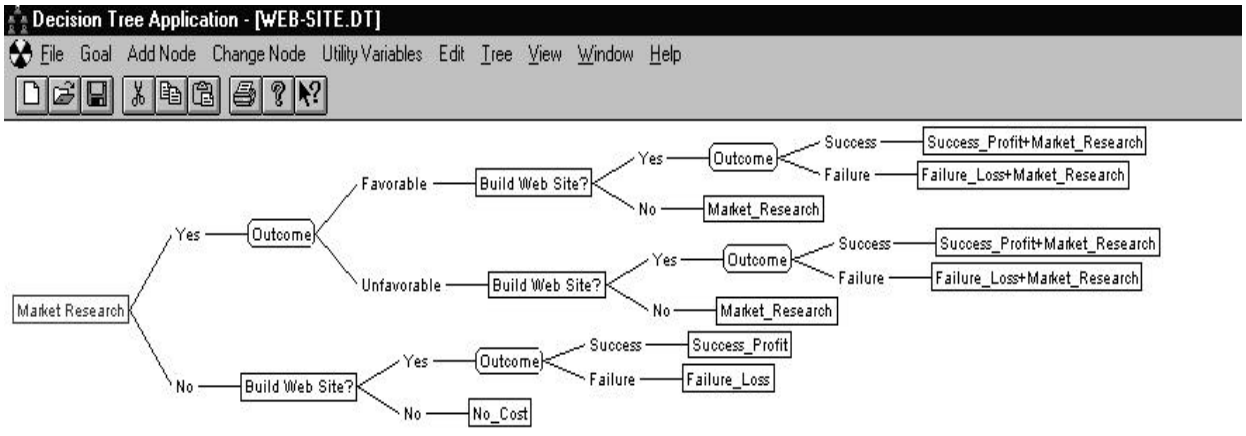

Figure 4: A DECTREE screen showing a decision tree for E-Commerce problem

Of course, this is a grossly oversimplified representation of a very complex problem. Nevertheless, it gives students the idea of decision trees, their implementation of expected utility, and most importantly the notion of how this is a special case of the general tree structure, complete with recursive search through the tree, which is how the overall expected utility of the decisions are calculated. The details of this algorithm are described below.

OBJECT-ORIENTED PROGRAMMING OF AN EXPECTED UTILITY MODEL

After students complete the tree-node programming exercise and the DECTREE assignment, they design and write the code for their own **decision tree node** class. Specifically, they create a subclass of the general tree node class (figure 5). This class inherits much of the functionality from the **node** class, the linkages to parents and siblings and the general tree-search algorithms. It specializes the node by including information regarding the specific type of node (decision node, choice branch, chance node, outcome branch, or utility node) and the value (utility or probability) for the node. Specialized functionality includes the calculation of expected utility in a recursive function, implementation of constraints regarding adding child nodes to a node based on the type of node (by polymorphic override), and display of data specific to the decision node class (also by polymorphic override). For many students, this is their first real taste of the usefulness of virtual functions in OOP.

The most challenging part of the programming exercise is calculation of the expected utility. This is a recursive function, which implements the formula in the expected utility equation shown above. The recursive aspect of this equation involves the term $U(C_i)$, which may either a simple value or the expected utility of an intermediate decision or chance node. If $U(C_i)$ is an expected utility, this requires a recursive function call. The *calculate_value* function performs different operations, depending on the type of node. Utility nodes simply return the value given by the user. Chance nodes calculate and return the expected utility by summing the values returned from the children (outcome branches) and store the result the *value* variable. Outcome branches calculate and return a value by multiplying the child's calculated value by the probability that is stored in the value variable.(note that there is only one child for this type of node). Choice branches store into the *value* variable the calculated value from the one and only child of this node, then return that value. Decision nodes store into the value variable the maximum value from all the children (i.e. choice branches) of this node, and return this value.

```
class decision_node: public node{   // inherits from node
   private:
     double value;
     int node_type;
   public:
     decision_node(); ~decision_node();
     double calculate_value();          // recursive
     void set_value();
     double get_value();
     virtual node* add_child();         // polymorphic
     virtual void display_extra_stuff(); // polymorphic
};
```

Figure 5: class definition for the decision tree-node

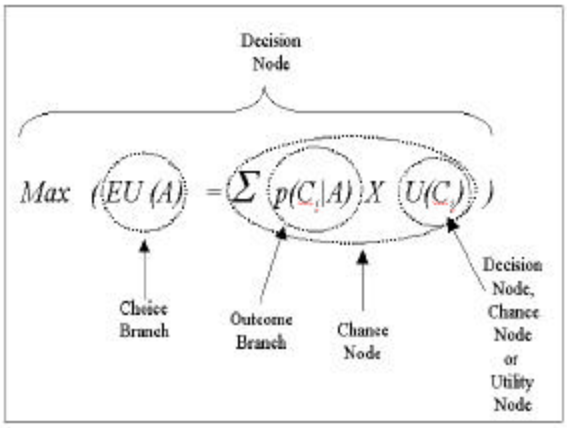

Figure 6: Contributions of various decision-tree nodes to the expected utility formula.

Figure 6 shows how each node contributes to the overall expected utility formula. The recursive aspect of the formula stems from the fact that a utility in the expression may be the result of a calculation at a decision node or chance node.

When students finish writing the software, they then test it with their own problem cases, or with a problem case provided by the instructor (such as the one described in figure 3). This gives students the satisfaction of having created their own DSS generator and then actually using it. Since they have an existing tool to work with (DECTREE), they can compare results of their program against the results of a DECTREE run.

CONCLUSION AND FUTURE DIRECTIONS

The pedagogical method described in this article brings an in-depth programming focus to decision support course work in a way that is not commonly seen in most DSS courses. Unlike most DSS courses, even those that include programming skills, this approach places strong emphasis on object-oriented techniques and applications of abstract data structures. Thus, students make the connection between inheritance/polymorpism/encapsulation, generic abstract data structures, and the decision models that they support. With this approach, students gain first-hand experience with *tool-use*, just as they do in most DSS courses. But in addition, students also gain experience with *tool-building*, and thus a deeper understanding of the implementational issues involved in creating DSS generators. This approach can be applied to a wide variety of decision models.

In the future, the author intends to apply this approach for more decision models. In addition, students will be exposed to using built-in Java or ActiveX tree controls in order to provide graphic user interfaces in their implementations. Currently, the programming assignments are console-based applications, with command-line interfaces. A graphic-oriented assignment will make for a more attractive and exciting programming experience, and will give students exposure to Windows programming skills including event-handling and component-based programming. Combining object-orientation, GUI component building, abstract data structures, and decision modeling should provide students with interdisciplinary topic coverage in a technically rigorous setting. The skills they gain from such exposure will increase their employability and give them the satisfaction and appreciation of having accomplished meaningful software development work in realistic problem domains.

REFERENCES

1. Clemen, R.T. (1996). Making Hard Decisions: An Introduction to Decision Analysis, Second Edition, Duxbury Press, Belmont, CA.
2. Dawes, R. (1988). Rational Choice in an Uncertain World, Harcourt Brace Jovanovic Publishers, Orlando, FL.
3. Langlotz, C.P. (1989). A Decision-Theoretic Approach to Heuristic Planning, PhD Dissertation, Report # STAN-CS-89-1295, Stanford University, Stanford CA.
4. Mitri, M. (1998). A Graphical Software Implementation of Decision-Theoretic Modeling for Expected Utility Problems and its use in an Information Systems Class Assignment. Proceedings of the IACIS Conference, Cancun Mexico.
5. North, D.W. (1990). A Tutorial Introduction to Decision Theory, in Readings in Uncertain Reasoning (ed. G. Shafer, J. Pearl), Morgan Kaufmann Publishers, San Mateo, CA.