Department of Computing Science and Mathematics
University of Stirling

Writing Decision Trees for the CGT Viewer Program

Richard Bland
Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA
Scotland

Claire E Beechey
Department of Computing Science and Mathematics
University of Stirling

Dawn Dowding
Hull York Medical School
University of York

Technical Report CSM-163

ISSN 1460-9673

October 2003

Department of Computing Science and Mathematics
University of Stirling

Writing Decision Trees for the CGT Viewer Program

Richard Bland
Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA
Scotland

Claire E Beechey
Department of Computing Science and Mathematics
University of Stirling

Dawn Dowding
Hull York Medical School
University of York

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email r.bland@cs.stir.ac.uk

Technical Report CSM-163

ISSN 1460-9673

October 2003

Contents

List of Figures

List of Tables

Abstract

This paper is a tutorial and manual for authors of Decision Trees to be displayed by the CGT Tree Viewer program. It assumes that readers understand the basic concepts of Decision Trees and have sufficient skills to create and edit plain text files. After reading this document, you should be able to write your own Decision Tree and get the CGT program to display it.

Acknowledgements

The work reported here is part of the project *The Development and Evaluation of a Computerised Clinical Guidance Tree for Benign Prostatic Hyperplasia and Hypertension*, funded by the Chief Scientist's Office of the Scottish Executive with a grant for £150K over three-and-a-half years from March 2000. This project is being undertaken by staff from the Stirling University Department of Nursing and Midwifery (Dawn Dowding, now at Hull York Medical School, University of York; Audrey Morrison, Anne Taylor, and Pat Thomson), Forth Valley Primary Care Research Group (Chris Mair and Richard Simpson), Department of Psychology (Vivien Swanson), and Department of Computing Science & Mathematics (Claire Beechey and Richard Bland).

1 Introduction

Almost every situation in human affairs involves choices between alternative courses of action. A *Decision Tree* is a standard way of displaying the choices that can be made in a given situation, together with the possible outcomes of those choices. For discussions of Decision Trees, see [4], [5], [2], and [9].

The focus of our work is principally on the information-presenting role of Decision Trees. Decision Trees in the medical domain have been referred to as “Clinical Guidance Trees” (CGTs): see, for example, [5]. As the name suggests, CGTs have usually been seen as a tool for the medical professional. Our emphasis is different. Instead of using CGTs to help experts, we have been looking at their potential for informing patients, enabling them to have a more complete understanding of their treatment. Our current project (see [3]) has been testing their utility for conveying information about two particular medical conditions. A Decision Tree was written for each condition, and a computer program displayed the trees to patients participating in the study.

Our program, the *CGT Tree Viewer* or *CGT* for short, was developed at Stirling for the project. It can, however, display any Decision Tree that is written using CGT’s input conventions. Other workers, for example, have developed trees for Hepatitis C and for Menorrhagia. This document describes how to prepare a tree for CGT.

Before you begin, you should be sure that CGT does what you want. Our program is optimised for presenting information to lay users. It can give them detailed information and help. It can even tailor the tree to the particular circumstances of the user. It can do this because it uses an extended version of the Decision Tree model. This is described in a companion paper, [1]. CGT does not, however, enable experts to explore the the technical properties of trees. For example, it does not do sensitivity analysis. Other tools should be used for this kind of technical exploration.

2 A Very Simple Tree

Figure 1 shows a very simple Decision Tree. The tree is made up of three kinds of *node*, connected by branching lines. The name or label of each node is shown on the line leading up to the node. At the right-hand-side of the tree we have *Terminal nodes*, shown as small vertical bars. These represent the various final outcomes. We also have a *Decision node*, shown as a square. This is a branching point at which a decision can be made: this Sunday afternoon, should we go out for a walk, or read the paper? Finally, there is a *Chance node*, shown as a dot. This is also a branching point, but it is one where fate, chance, or other outside agency, determine the outcome: we ourselves are unable to influence what the weather will do. It may rain, or it may be a fine day.

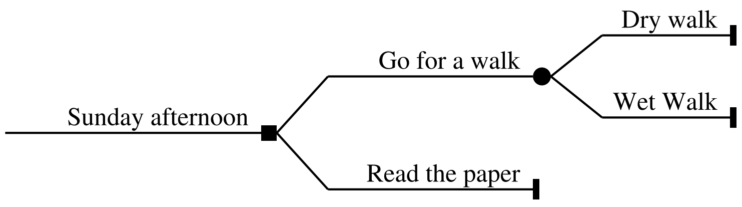


Figure 1: Should we go for a walk, or read the paper?

The leftmost node in the tree is the *root* node. Nodes that are connected by lines are *parent* (to the left) and *child* (to the right). The children of the same parent are called *siblings*. If we trace lines backwards (to the left) from any child node we go though the *ancestors* of the node, all the way back to the root. Obviously, the root is an ancestor of all the other nodes in the tree.

Chance and Decision nodes must have children, to represent the different possible outcomes. A Terminal node, as the name implies, must not have children.

Associated with the children of a Chance node are *Probabilities*: estimates of the relative likelihood of the various branches. These must sum to 1.0. For example, suppose that we think that the probability of rain this Sunday is 0.7, and hence the probability of a fine day is 0.3. Of course, the children of a Decision node don't have probabilities: the choice between the children is under human control, not chance.

Associated with Terminal nodes are *Payoffs*: the value of that outcome. By asking users to attach their own utilities to the various outcomes, we can extend the Decision Tree method to areas where there are no obvious numerical "payoffs". The CGT program invites users to give the various possible outcome values between 100 (most desirable) and zero (most undesirable). For example, consider the possible outcomes on Sunday afternoon. Here the question is whether we should go out for a walk, or stay at home to read the Sunday paper. The latter is safe but dull. The success of the walk will depend on the weather: if it clears up, the walk will be very enjoyable, but less so if it rains. Suppose the user supplies utilities such as:

- Walk, it rains (rather disagreeable: 30)
- Walk, weather fine (very pleasant: 80)
- Read the paper at home (rather boring really: 40)

The resulting tree with its probabilities and payoffs is shown in Figure 2.

How could we code this tree for display by the CGT program? We would prepare a plain ASCII text file, using any convenient editor such as TextPad (`http://www.textpad.com/`) or Notepad (supplied free with Windows). Microsoft Word can be used at a pinch, but you must stop it from being too helpful: in particular, you must insist that it saves the file as plain text rather than as a Word document. The file can be called whatever you like, but with extension `.tree` . For example, we could call it `Sunday.tree` . It can be placed anywhere in the filestore, but a convenient place is the `Treefiles` folder of the CGT installation.

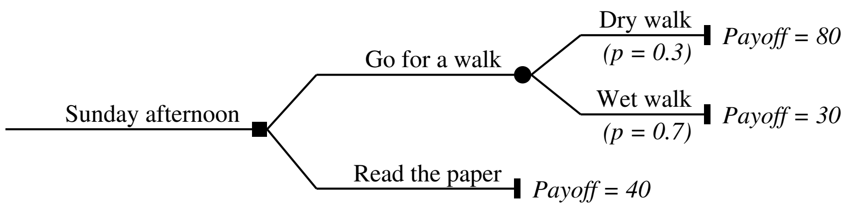


Figure 2: Adding probabilities and payoffs

Figure 3 shows the text of a file that codes the tree of Figure 2 for display by the CGT program. In Figure 3 the lines are numbered at the left-hand side. These numbers are not part of the file: they are included for ease of discussion here.

We see from Figure 3 that the tree file is composed of *fields.* For example, line 1 contains the `TreeName` field. Each field begins with its distinctive *tag*, followed immediately by a colon and then the contents of the field. There may be spaces after the colon but not before it. The tag must be given exactly: for example the `TreeName` tag cannot be given as `Treename`. The tag must be the first non-whitespace[1] characters on the line. The contents of the field can, if necessary, spread over several lines. (The field is terminated by the next tag or by the end of the tree file.)

The fields in the tree file are of two types. First come the *tree fields* and then the *node fields.* In the Sunday tree, for example, the tree fields are on lines 1 to 4 of Figure 3. The remainder of the file contains node fields.

There are three essential tree fields: they can be seen at the top of Figure 3.

TreeName A short label for the tree, used to provide a label for the button that shows the tree.

IntroText The program displays this text before starting the main display. It should be used to provide an introduction to the condition modelled by the tree. The text uses our *DisplayText* format: this is described in Section 5, but for the moment we can think of it as some flavour of HTML, the markup language used for World-Wide Web pages. This means that we can add formatting information to the text. In this example the text is completely plain.

RequiresVersion The program has gone through several versions, with different features and successively greater capabilities. This manual describes trees that can be run by versions 1.90 and later. The example in Figure 3 cites version 2.31, which makes it compatible with all distributed versions of CGT.

The remainder of the file describes the five nodes of our simple example. Each node is described by a group of node fields. In our example we have used only the essential node fields.

ID A short identifier for the node. Must not contain spaces.

ShortLabel The ShortLabel is the short text that will be displayed on the node's branch when the tree diagram is drawn. Due to diagram space constraints, it should be no more than 15 characters. Where the same outcome appears at more than one point in the tree, it is important to ensure that the same ShortLabel text is used. This ensures that these outcomes are treated as one outcome when the user is asked to weight the possible outcomes.

ParentID The ID of the parent node (the node immediately to the left) to which this node is attached. In the case of the root node, this field is left blank.

Type Indicates the node type: one of `Decision`, `Chance`, `Terminal`, or `Question` (we shall discuss Question nodes later).

[1]Space characters, tabs, and newlines are "whitespace" characters.

```
TreeName: Sunday walk
IntroText: This tree deals with the problem of deciding
 what to do on Sunday afternoon.
RequiresVersion: 2.31

 ID: 0
 ShortLabel: Sunday afternoon
 ParentID:
 Type:Decision
 UserText: We are deciding what to do on Sunday afternoon,
   and are considering reading the Sunday paper.
   <p> However, we are also considering going for a walk.
   <p> It is cloudy at present.

 ID: 1
 ShortLabel: Go for a walk
 ParentID: 0
 Type: Chance
 UserText:
   If we go for a walk, the weather may clear
   and the sun may come out: or it may turn out wet.

   ID: 1.1
   ShortLabel: Dry walk
   ParentID: 1
   Type: Terminal
   Prob: 0.3
   Payoff: 80
   UserText: The weather clears and we have a pleasant walk.

   ID: 1.2
   ShortLabel: Wet walk
   ParentID: 1
   Type: Terminal
   Prob: #
   Payoff: 30
   UserText: The weather worsens and it comes on to rain.
     We have a wet walk.

 ID: 2
 ShortLabel: Read the paper
 ParentID: 0
 Type: Terminal
 Payoff: 40
 UserText: Reading the Sunday papers is OK, up to a point,
   but it leaves you feeling vaguely dissatisfied.
```

Figure 3: The file `Sunday.tree`

Prob This is necessary for nodes which are children of a Chance node. The probability can either be expressed as a simple number less than 1 (e.g. 0.4) or as an expression (we shall discuss expressions later). The probabilities of the children of a Chance node must add to 1, obviously. Optionally, the probability one of the children may be specified using the hash character (`#`). (See line 35 of Figure 3.) This means "whatever's left". For example, if (as in Figure 3) there are two children, one with a specified probability of 0.3, then the hash sign evaluates to the probability required to add up to 1: in this case, 0.7.

Payoff A number between 0 and 100, representing the utility value or weighting given to the outcome. It can be left blank, because the value will be supplied by the user at runtime, but it is better practice to supply a default value to be used if the user skips the utility-elicitation phase of the program (which we describe later).

UserText This text provides a detailed text explanation of the situation at each node. As with the IntroText tree field, the text uses our DisplayText format, providing a number of ways in which the text's formatting can be specified: these are described in Section 5. In this example the text is almost completely plain: but lines 12 and 13 begin with the HTML tag `<p>`, indicating a paragraph break. (A display of these two breaks can be seen in Figure 9.)

The text in this field can also be split into sections: this is described later, in Subsection 5.4.

There are no comments in this example file, which is not very good practice. In a tree of any size the author should include comments to note any points that may not be obvious from the text. The name of the tree author and the revision history are good examples. Comments begin with the character pair `*/` as their first non-whitespace characters. Those characters and the following characters, up to and including the end of the line, are ignored when the tree is read by CGT.

This simple example has introduced only a few of the possible tree features that CGT can display. We shall cover the others in due course, but in the next section we look at how CGT would display the simple tree of Figure 3.

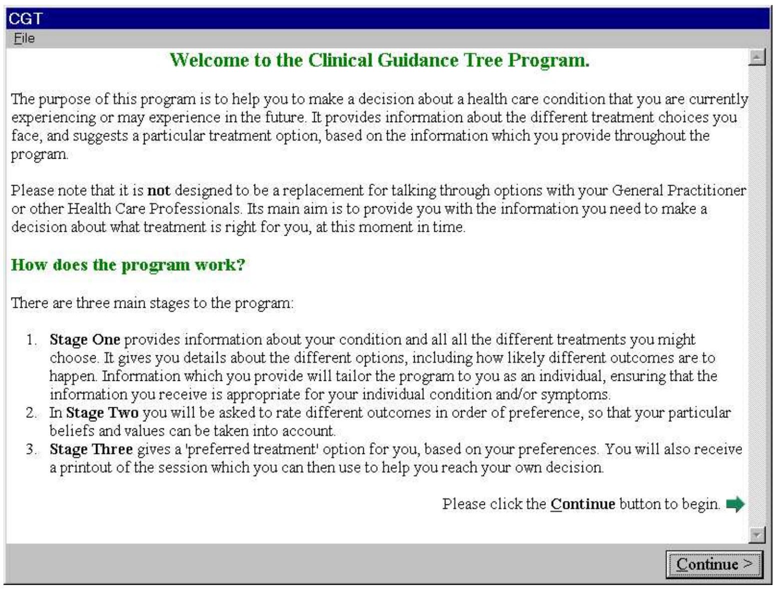
3 Viewing the Very Simple Tree

3.1 Introduction

The CGT Tree Viewer program is available from the authors. It runs on Windows PCs, and is installed simply by inserting the distribution CD and following the on-screen instructions.

If your CGT viewer has been installed in the normal way, you can run it from the Windows Start button. In this section we see what would happen if we ran the CGT Viewer program using the simple tree of Figure 3 as input.

When the program runs, it displays a standard introductory screen, shown in Figure 4. The text on this screen is not drawn from a tree file, but from a standard file in the CGT installation. All trees viewed on the computer will use this file. The file, `..\CGT\data1_90\HelpFiles\cgtIntro.html`, contains plain text (using HTML) and can be edited, if you wish.



Welcome to the Clinical Guidance Tree Program.

The purpose of this program is to help you to make a decision about a health care condition that you are currently experiencing or may experience in the future. It provides information about the different treatment choices you face, and suggests a particular treatment option, based on the information which you provide throughout the program.

Please note that it is **not** designed to be a replacement for talking through options with your General Practitioner or other Health Care Professionals. Its main aim is to provide you with the information you need to make a decision about what treatment is right for you, at this moment in time.

How does the program work?

There are three main stages to the program:

1. **Stage One** provides information about your condition and all all the different treatments you might choose. It gives you details about the different options, including how likely different outcomes are to happen. Information which you provide will tailor the program to you as an individual, ensuring that the information you receive is appropriate for your individual condition and/or symptoms.
2. In **Stage Two** you will be asked to rate different outcomes in order of preference, so that your particular beliefs and values can be taken into account.
3. **Stage Three** gives a 'preferred treatment' option for you, based on your preferences. You will also receive a printout of the session which you can then use to help you reach your own decision.

Please click the **Continue** button to begin.

Figure 4: Viewing the Simple Tree: Welcome Screen

When the user clicks the "Continue" button, the program displays a simple log-in screen, shown in Figure 5. The user enters any short identifier (preferably one that he or she has been given by your research team, or else your records will get muddled). This is recorded by CGT's Logging system.

When the user clicks the "OK" button, the program displays a screen to enable the user to pick a tree to open, shown in Figure 6. The screen shows the standard trees: here, our Benign Prostatic Hyperplasia and Hypertension trees. These options are compiled-in to the current version of the program: a future version will allow this to be customised. In any case the user can use the screen to navigate to any `.tree` file and open it.

If you have just changed your tree file, or have written one from scratch, there will probably be errors in it. If so you will see a warning screen, and you should look in the folder containing your tree file for the `errorlog.txt` file. Read that file, work out what you have done wrong, and edit the tree file. Now cross your fingers and re-open it from CGT (you will need to go back to the screen shown in Figure 6). From now on we assume that the tree is free from errors.

When the file opens, the user sees a screen showing the text from the `IntroText` tree field (lines 2 and 3 of Figure 3). This is shown in Figure 7. Clearly this is a very dull screen, because the contents of the field are so simple: but it can be as elaborate as you wish. We shall discuss the available methods later, but as a contrast Figure 8, from our Hypertension tree, gives some idea of the possible effects.

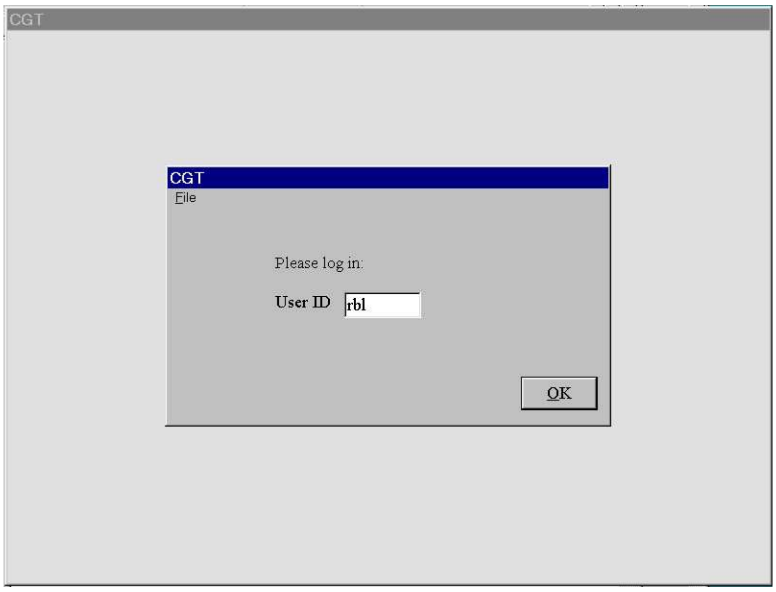


Figure 5: Viewing the Simple Tree: Login Screen

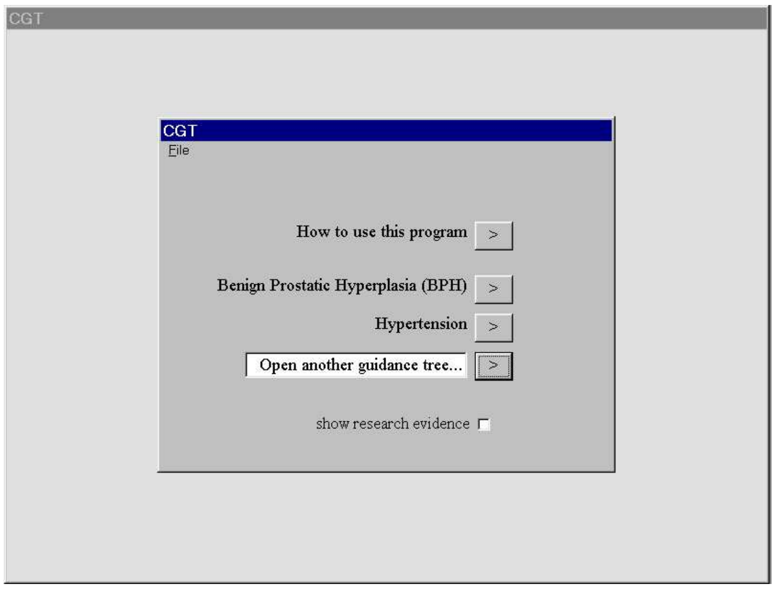
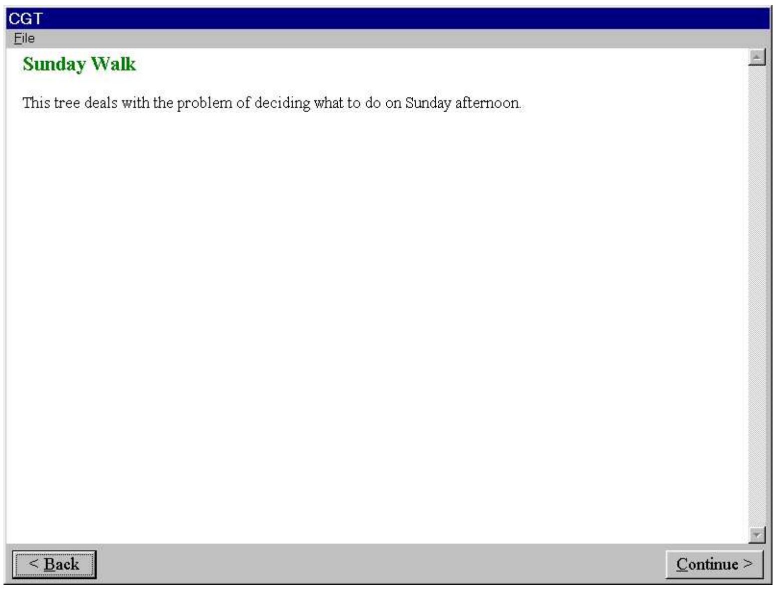


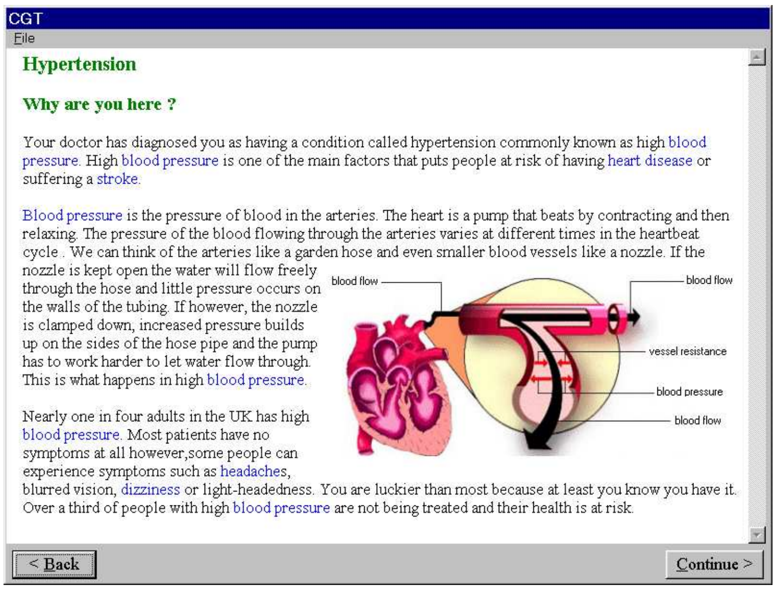
Figure 6: Viewing the Simple Tree: Choose Tree Screen



Sunday Walk

This tree deals with the problem of deciding what to do on Sunday afternoon.

Figure 7: Viewing the Simple Tree: Intro Text Screen



Hypertension

Why are you here ?

Your doctor has diagnosed you as having a condition called hypertension commonly known as high blood pressure. High blood pressure is one of the main factors that puts people at risk of having heart disease or suffering a stroke.

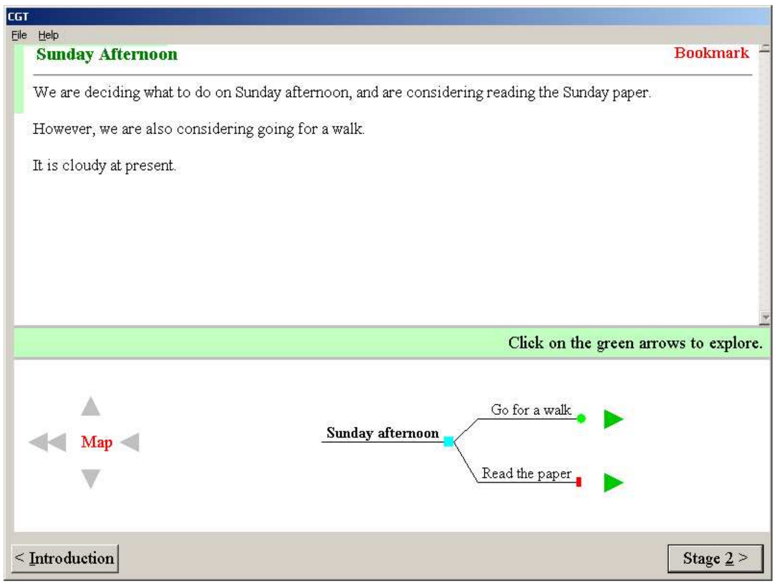
Blood pressure is the pressure of blood in the arteries. The heart is a pump that beats by contracting and then relaxing. The pressure of the blood flowing through the arteries varies at different times in the heartbeat cycle . We can think of the arteries like a garden hose and even smaller blood vessels like a nozzle. If the nozzle is kept open the water will flow freely through the hose and little pressure occurs on the walls of the tubing. If however, the nozzle is clamped down, increased pressure builds up on the sides of the hose pipe and the pump has to work harder to let water flow through. This is what happens in high blood pressure.

Nearly one in four adults in the UK has high blood pressure. Most patients have no symptoms at all however,some people can experience symptoms such as headaches, blurred vision, dizziness or light-headedness. You are luckier than most because at least you know you have it. Over a third of people with high blood pressure are not being treated and their health is at risk.

Figure 8: Hypertension Tree: Intro Text Screen

3.2 WaT mode

As the user moves through the CGT program, they move through a number of *viewing modes*. At the moment, they are in *Intro mode*, but when they click the "Continue" button they move into *Walk-a-Tree mode* or *WaT mode*. This is perhaps the most important mode from the point of view of user exploration: the user can explore the tree and browse back and forward. WaT mode is the main area in which the user can see the extended explanations that are one of the distinctive features of our Decision-tree model ([1]). Of course, in our sample tree the UserText fields are very simple, and so the WaT mode screens are rather bare. Figure 9 shows the first WaT screen for the sample tree.



We are deciding what to do on Sunday afternoon, and are considering reading the Sunday paper.

However, we are also considering going for a walk.

It is cloudy at present.

Click on the green arrows to explore.

Figure 9: Sunday Tree: First WaT Mode Screen

In WaT mode there is always a *current node*: the node that the user has just reached. We see this node ("Sunday afternoon") in bold in the lower pane of Figure 9. The upper pane displays the node's UserText field. We see from the top-right corner of the screen that the user can *bookmark* any node. (A bookmarked node's UserText is printed in full in the printout that the user can obtain at the end of the session.)

A heading for the current node is displayed above the UserText in the upper pane. This heading will be the ShortLabel, unless the author has included a LongLabel field for the node, containing plain text, which will be used instead of the ShortLabel. The LongLabel field is optional.

In the lower pane the user always sees the current node, with its ShortLabel. The parent node is also always visible (unless, as here, the current node is the root). The display will also show as many sibling links and child nodes as can be fitted into the space. In this case there are no siblings and only two childen, so eveything possible is being displayed.

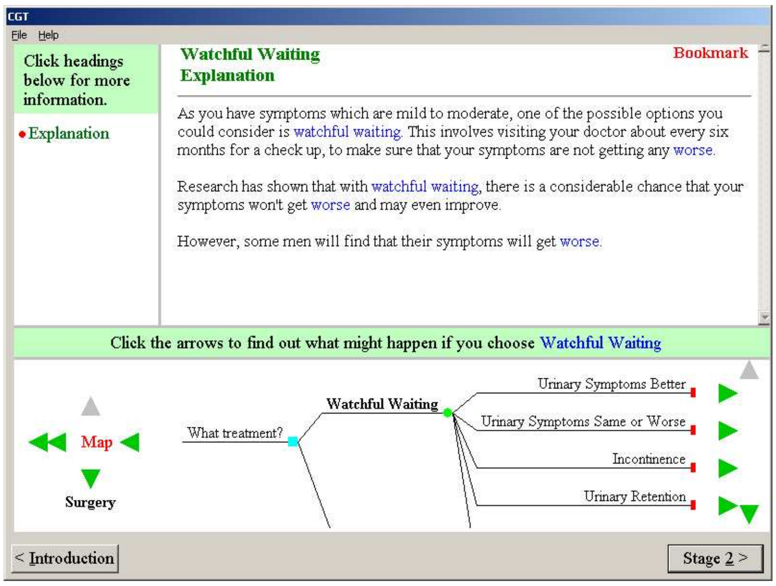
The upper and lower panes are separated by instruction text: this can be seen in Figures 9 ("Click on the green arrows to explore") and 10 ("Click the arrows to see what might happen if you chose Watchful Waiting"). These instruction texts can be a standard text, or can be set for each node by the tree author, using the InstructionText field. The field contains plain text. The default values for this field are different for each type of node. They can be customised for an installation (in the file `data1_90\default.txt`).

There are navigational methods that allow the user to move about the tree: for example, to move to one of the current node's childen. As a result of any move the "current node" is the one to which the user has just moved.

Within WaT mode, the following moves are possible. Some of them use the *DVD control*, found at the left-hand side of the lower pane in WaT mode. See, for example, the left-hand side of the lower pane in Figure 9.

to a child Click on the child's ShortLabel, or on the green arrow to the right of the child.

to a child out of view Click on the scroll arrows to find the child, then proceed as before. These scroll arrows, at the extreme right of the lower pane, only appear when needed. They aren't present in Figure 9, but can be seen in Figure 10, a screen from one of our other trees.



As you have symptoms which are mild to moderate, one of the possible options you could consider is watchful waiting. This involves visiting your doctor about every six months for a check up, to make sure that your symptoms are not getting any worse.

Research has shown that with watchful waiting, there is a considerable chance that your symptoms won't get worse and may even improve.

However, some men will find that their symptoms will get worse.

Figure 10: WaT Mode: A node with many children

to the parent Click on the parent's ShortLabel, or on the single green left-arrow in the DVD control.

to a sibling Click on either the single green up-arrow or down-arrow in the DVD control, as appropriate. The arrow(s) will be labelled with the ShortLabel of the adjacent sibling. Again, Figure 10 shows an example: the DVD's down arrow is labelled "Surgery," the current node's sibling.

to the root of the tree Click on the double green left-arrow in the DVD control.

These various arrows are green when enabled, and gray otherwise. For example, if the current node has no siblings, the DVD's "sibling" arrows are gray.

The DVD control also contains a clickable label, "Map". This allows the user to see, on a temporary screen, the current node within the context of the whole tree, drawn as a conventional wire-diagram. This is a kind of preview of *Show-a-Tree mode* or *SaT mode*, which we shall describe in due course.

The user can move forward out of WaT mode by clicking the "Stage 2" button at the bottom-right of any WaT screen.

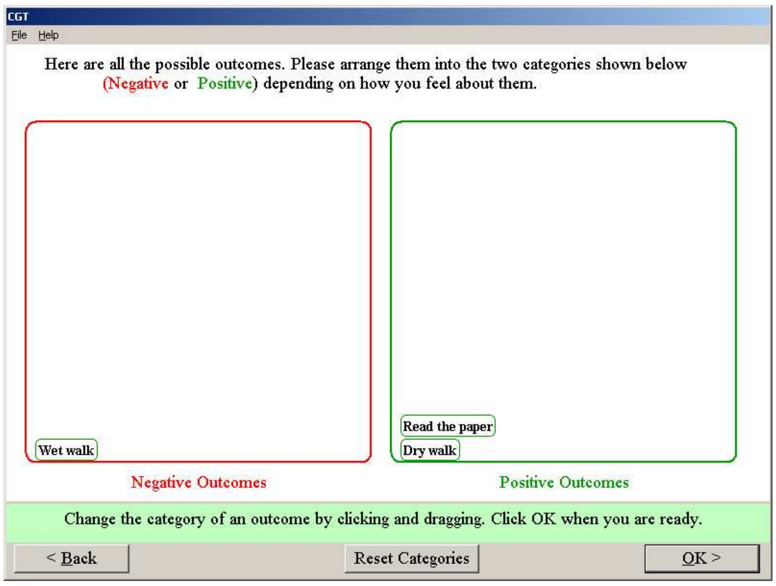
3.3 Utility Elicitation

On leaving WaT the user enters *Utility Elicitation mode* ("Stage 2"). Here the user is asked to provide numerical values, on a scale of 0 to 100, for every possible distinct outcome. Precisely, they are asked to rate every distinct ShortLabel from the Terminal nodes. (Readers may recall that different terminal nodes with the same outcome, such as "Restored to full health", should be given the same ShortLabel.)

First, the user is asked to separate "Positive" (nice) outcomes from "Negative" (nasty) outcomes by moving the nodes' ShortLabels between two panels on the screen. (We need to distinguish between Positive and Negative outcomes in case any nodes are *composed*. We deal with Composition later.) This step can be seen in Figure 11. Secondly, the user is asked to scale the outcomes by dragging them on a ruler. This step can be seen in Figure 12. In this second step, the user is prevented from mixing Positive and Negative outcomes on the scale. All Positive outcomes have to be ranked above all Negative ones.

When the user has scaled all the outcomes, they click the "Stage 3" button to leave Utility elicitation and enter *Show-a-tree* or *SaT* mode.

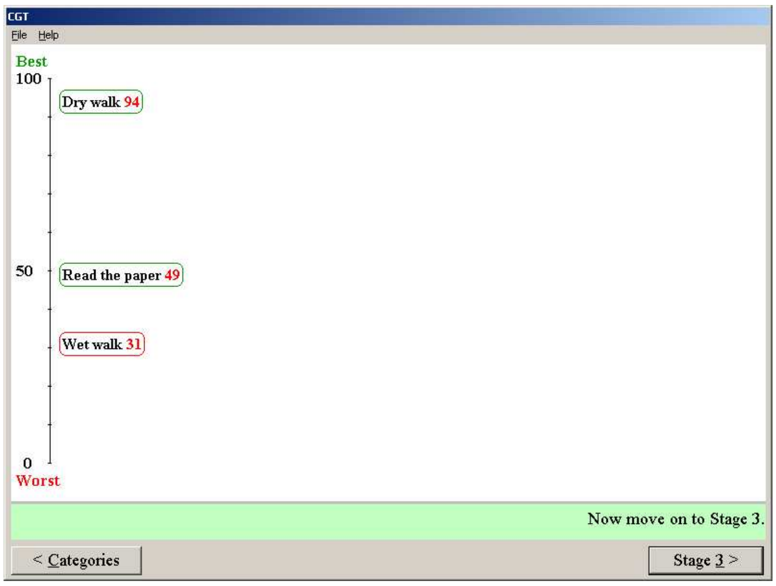
The user can skip Utility elicitation if they wish, by clicking "Stage 3" without dragging any outcomes to the scale. If so, the default values — the values in the Payoff fields of the Terminal nodes — are used.



Here are all the possible outcomes. Please arrange them into the two categories shown below (Negative or Positive) depending on how you feel about them.

Change the category of an outcome by clicking and dragging. Click OK when you are ready.

Figure 11: Utility Elicitation: Desirable and undesirable outcomes



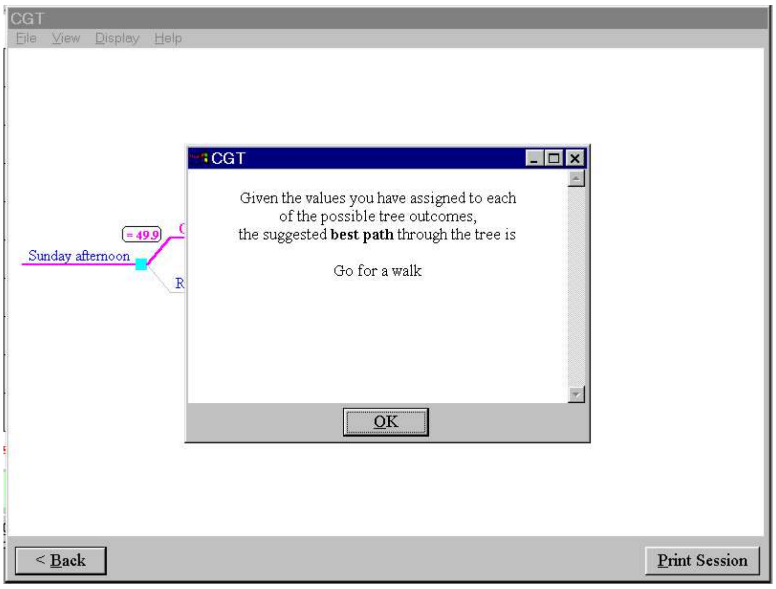
Now move on to Stage 3.

Figure 12: Utility Elicitation: Scaling outcomes

3.4 SaT mode

In Show-a-Tree (SaT) mode the user sees the “conventional” view of the Decision Tree, in which the whole tree is visible at once (perhaps with the aid of scroll bars) and the optimal path has been calculated, using the probabilities and utilities as they now stand.

First the user sees a superimposed window giving the optimal path. This is shown in Figure 13. Once



Given the values you have assigned to each
of the possible tree outcomes,
the suggested **best path** through the tree is

Go for a walk

Figure 13: Show-a-Tree mode: optimal path

this superimposed window has been dismissed (or slid aside) the user can see the tree, with the computed vales. This is shown in Figure 14. There is a range of options at this point. By clicking right-mouse on a

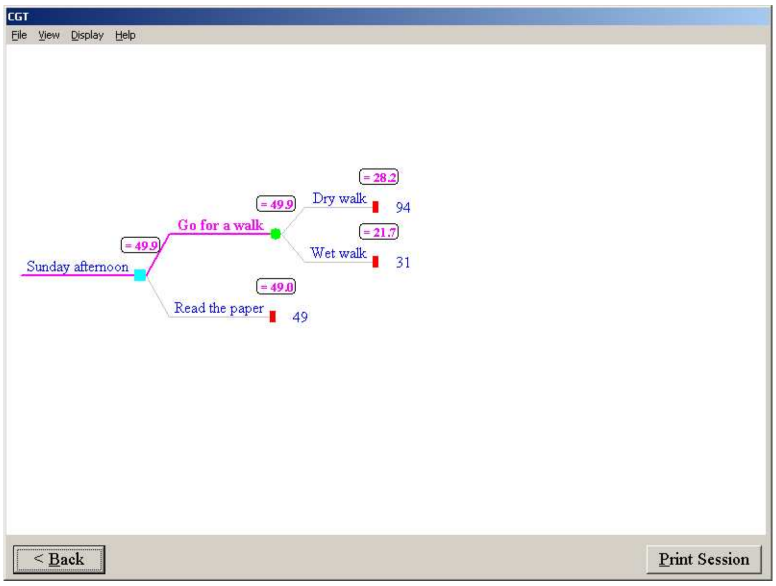


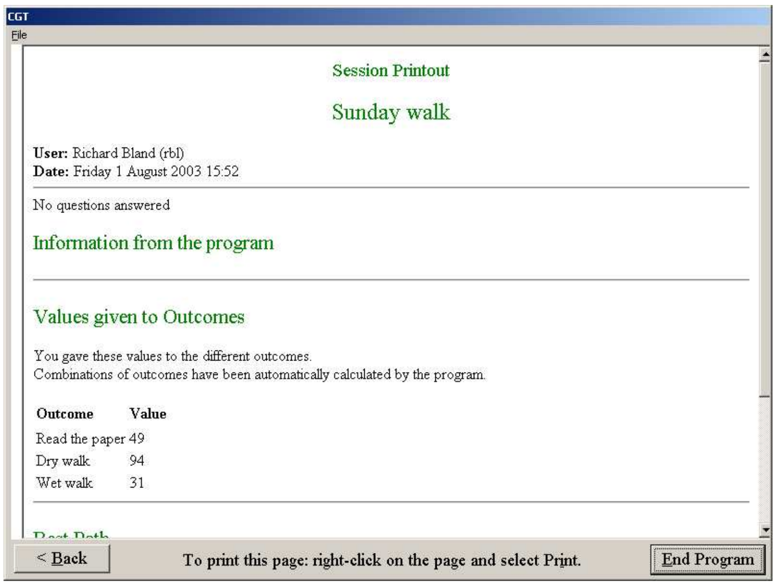
Figure 14: Show-a-Tree mode: the display

non-terminal node, the user can expand or collapse the display of branches. Using the View and Display menus, the user can change the level of detail and colours. Most importantly, by clicking right-mouse on a Terminal node the user can change its payoff, and then (from the View menu) re-compute the optimal path.

3.5 Printing

The user is free to move back and forward through the modes. If, for example, they wish to revisit WaT mode to browse the various explanatory texts, change their utilities, and then recompute the optimal path, they are free to do so.

As a final step the user can print a record of their session, using the "Print Session" button at the bottom right-hand of the SaT screen (see, for example, Figure 14). The result is shown in Figure 15. The program has in fact created an HTML file and is displaying it in the current window. The user can



Session Printout

Sunday walk

User: Richard Bland (rbl)
Date: Friday 1 August 2003 15:52

No questions answered

Information from the program

Values given to Outcomes

You gave these values to the different outcomes.
Combinations of outcomes have been automatically calculated by the program.

Outcome	Value
Read the paper	49
Dry walk	94
Wet walk	31

To print this page: right-click on the page and select Print.

Figure 15: The Session Printout

send the file to the ordinary Windows print queue by right-clicking on the window, and can stop the program by clicking the "End Program" button. Or, if the user wishes to go round again, they can work back to any point and re-explore the tree — or even load a different tree.

4 Variables

So far, our examples have had probabilities and payoffs that were constant values, quoted literally. Thus the probability of fine weather on a Sunday walk was given as 0.3 on line 27 of the tree in Figure 3. However, in any tree of a realistic size a tree author is likely to want to use a value several times in the same tree, and it is much safer to do this using a *variable*. For example, we could set up the variable (say) `pFine` to have the value 0.3, and write the variable's name in the tree at each point where we want to use that value. This would ensure that all references to the probability of fine weather would use the same value. Then, if later research showed that the true value was 0.25, we would just have to make that change at the single point in the tree where the variable was set up.

We can declare this variable with the TreeVariable tree field. For example, we could write

```
TreeVariable: pFine = 0.3
```

Now the Prob field in line 27 of Figure 3 can be written

```
Prob: pFine
```

You can declare as many variables as you like, each in its own TreeVariable field. Obviously, each variable must have a unique name. Names of variables consist of letters and digits (no spaces), starting with a letter. The case of letters is ignored, so (say) `pFine` is the same as `Pfine`. You don't need to supply a value when declaring the variable, provided you have supplied some other way for the variable to have a value before it is used. We shall see how to do this later. Variables are "global," that is, a particular variable can be used anywhere and always has a single value.

Variables in CGT are numeric (technically, signed Real values). We shall see later, in Subsection 5.3 and Section 7, how to output their values in displayed text, and how to use their values to manipulate standard phrases in such texts.

Frequently a tree author will want to manipulate variables arithmetically. For example, the default values of a set of payoffs may be derived from calculations based on (for example) market prices. But it would be very foolish for the author to enter these calculated values literally. If the underlying values were to change, then the tree author would need to recalculate the payoffs by hand and re-enter them into the tree. This is a very error-prone procedure. A much better approach is to define variables to hold the underlying values, and then get CGT to compute the payoffs.

Suppose, for example, we have a tree in which the user might or might not buy a company's shares. Perhaps the company may be about to strike oil, in which case its shares will rise in value from £100 to £2000. In this particular case we could declare variables like this:

```
TreeVariable: initialInvestment = 1000
TreeVariable: shareNow = 100
TreeVariable: shareIfYes = 2000
```

Having set up these variables, at the node "Buy, strikes oil", we can change the way that the payoff is specified. Instead of calculating the payoff (£19,000) and writing

```
Payoff: 19000
```

we could write

```
Payoff:  initialInvestment*(shareIfYes/shareNow - 1)
```

These formulas are simple examples of an idea that will be familiar from many other computing tools: formulas in spreadsheets, for example. Just like spreadsheet formulas, a user (here, a tree author) will use standard arithmetic operations, probably some common functions (logarithms, for example), and also conditionals using `if` and `else`. In CGT, we call such a formula an *Evaluable Expression* (or just *Evaluable*), that is, a formula ("expression") that can be worked-out ("evaluated") when its value is needed. We shall go into greater detail about these in Section 6. For the moment we just note that these expressions can occur in TreeVariable, Prob and Payoff fields (and in some other fields that we shall encounter later).

5 Display Text

DisplayText is a format that we use in a number of fields, both tree fields and node fields. It enables the tree author to specify formatting of displayed text and the inclusion of illustrations. It has three main features:

- HTML elements can be used
- A tree-specific dictionary can provide a glossary
- Standard blocks of text can be specified using *macros*

5.1 HTML

HTML is the language in which WWW pages are written([6]). The CGT DisplayText format can handle any standard HTML formatting element (apart from frames) and so tree authors can employ these elements to enhance text in fields such as UserText. (Hypertext links are not supported, however.)

HTML is a *mark-up* language: i.e. ordinary text is "marked-up" by the insertion of "markers" in the text. These markers describe how the text is to be formatted when viewed by an *HTML browser*. A browser is a program that enables a user to view the text in its formatted form. Netscape and Internet Explorer are examples of browsers. When the browser reads the text, it ignores the text's original layout and spacing and obeys only the inserted mark-up. The CGT program uses its own built-in browser to render DisplayText.

In HTML the markers are called *tags*. A tag is written as a short keyword enclosed in diamond brackets. For example, the keyword for *em*phasis is <em>. Most tags occur in on-off pairs: the <em> tag is an example. The "off" tag has a slash in front of the keyword: so we use <em> to switch emphasis on, and </em> to switch it off. HTML tags are not case-sensitive: for example, `<em>` is the same as `<EM>`.

Table 1 gives the basic HTML tags that you are likely to need.

On	*Off*	*Meaning*
<h1>	</h1>	Heading text, level 1
<h2>	</h2>	Heading text, level 2
<h3>	</h3>	Heading text, level 3
<br>		Line break
<p>		Paragraph break (with blank line)
<ol>	</ol>	Ordered (numbered) list
<ul>	</ul>	Unordered (bulleted) list
<li>		List item
<em>	</em>	Emphasis (italics)

Table 1: Useful HTML tags

Here's an example of the use of the list tags:

```
<ul>
  <li> This is the first list item
  <li> This is the second list item
</ul>
```

this produces an effect like:

- This is the first list item
- This is the second list item

```
   <h3>Why are you here ?</h3>
   Your doctor has diagnosed you as having a condition
   called hypertension commonly known as high blood
   pressure. High blood pressure is one of the main
   factors that puts people at risk of having heart
   disease or suffering a stroke.

   <p>Blood pressure is the pressure of blood in the
   arteries. The heart is a pump that beats by contracting
   and then relaxing. The pressure of the blood flowing
   through the arteries varies at different times in the
   heartbeat cycle
   <img alt = 'blood_pressure'
src= 'bloodpressure2.gif'
align=right>.
   We can think of the arteries like a garden hose
   and even smaller blood vessels like a nozzle.
   If the nozzle is kept open the water will flow freely
   through the hose and little pressure occurs on the
   walls of the tubing. If however, the nozzle is clamped
   down, increased pressure builds up on the sides of
   the hose pipe and the pump has to work harder to
   let water flow through. This is what happens in
   high blood pressure.

   <p>Nearly one in four adults in the UK has high
   blood pressure. Most patients have no symptoms at
   all however, some people can experience symptoms
   such as headaches, blurred vision, dizziness or
   light-headedness. You are luckier than most because
   at least you know you have it. Over a third of
   people with high blood pressure are not being treated
   and their health is at risk.
```

Figure 16: DisplayText for Figure 8

More advanced tags are beyond the scope of this text. There are dozens of good HTML texts, such as [6]). As an example, however, Figure 16 gives the marked-up text for the Hypertension tree's IntroText field. The display of this can be seen in Figure 8. This text includes a picture, contained in the graphic file `bloodpressure2.gif`. This file (which is in the same folder as the tree file itself) is included using the HTML `img` tag.

5.2 The Dictionary

The texts in the tree are likely to involve specialised vocabulary: medical terms, for example. The tree file can specify a *Dictionary*: a list of words and phrases (a "term"), each with an extended explanation (the definition) attached. When the CGT program is displaying DisplayText it will recognise any word or phrase that is in the dictionary and will print it in a distinctive font, currently blue. (We call this *Blue Clickable Text* or *BCT*.) If the user clicks on the blue text, they see a new window displaying the definition.

The definitions can use DisplayText format: so they can contain HTML markup and can contain words or phrases that are themselves in the dictionary. This means that the display of the definition can itself contain Blue Clickable Text.

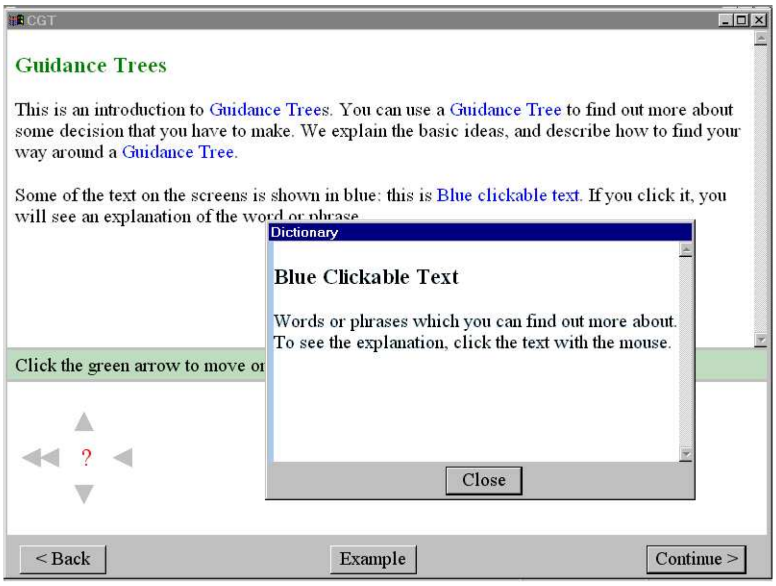
To use a Dictionary, the tree must specify the name of a file containing the definitions. This is done with the DictionaryFile tree field. For example:

```
DictionaryFile: Hypertension.dic
```

In this example, the file is in the same folder as the tree file. A path can be specified in the field, however, if the dictionary file is in a different folder. The file must be a plain text file. Its format is simple:

- Start each new term/definition pair on a new line.
- Dictionary terms and their definitions must be separated by the '|' (vertical bar) character. (On the keyboard this character is found to the left of the z key.)
- Dictionary terms can be phrases, like "Respiratory tract".
- Definitions may span several lines if necessary, BUT must not contain line breaks. Either use '/' at the end of a line to split up a long definition, or use line-wrapping in your chosen editor's display options. For example in Notepad use "Word Wrap" from the Edit menu to view the text.
- Definitions can themselves contain dictionary terms.
- Definitions can be written in HTML if desired. If no HTML tags are used, the text will be displayed as plain text.

Figure 17 shows a screen where the user has clicked on a BCT phrase.



This is an introduction to Guidance Trees. You can use a Guidance Tree to find out more about some decision that you have to make. We explain the basic ideas, and describe how to find your way around a Guidance Tree.

Some of the text on the screens is shown in blue: this is Blue clickable text. If you click it, you will see an explanation of the word or phrase.

Words or phrases which you can find out more about. To see the explanation, click the text with the mouse.

Figure 17: Clicking a dictionary term

5.3 Introduction to Macros

It is often the case that a tree author will wish to make use of some standard form of words at a number of different places in the tree. Suppose, for example, that every time a probability is quoted in a user text, it is to be explained, as in "the probability of this is about 0.2, or 20 cases in 100" or "the probability of this is about 0.05, or 5 cases in 100." Writing these standard forms is tiresome and error-prone: it is hard to maintain a consistent style, particularly if more than one author is involved, or if it is decided to adopt a different style.

In addition, a tree author is very likely to want to use the values of tree variables (introduced in Section 4) in the text. Developing the example in the previous paragraph, suppose that there is a tree variable `pEvent` that currently has the value 0.2. The tree author could write "the probability of this is about 0.2," but if the value of `pEvent` were to change, then the text as it stands would be wrong. It would be far better if the author had some method of instructing CGT to insert the current value of the variable in the text.

CGT meets both of these needs (recurring phrases, and dynamic use of tree variables), by the use of *Macroprocessing.* This allows the tree author to define *Macros* that expand into standard forms, manipulating and displaying tree variables as they are expanded. These expansions and manipulations do not happen statically as the tree is written, or as the tree is being read in, but take place dynamically, as the text is being displayed. Thus the the same fragment of DisplayText may be displayed differently to different users.

We give a fuller discussion of macros in Section 7, below.

5.4 Splitting the UserText Field

The simple examples of UserText fields that we saw in Figure 3 each contained a single piece of text. In practice it is better to split the material into sections, each of which has a label that acts like a button: the user can click on the label to see that section. Figure 18 shows this in the root node of our Hypertension tree. In Figure 18 the lines are numbered at the left-hand side. These numbers are not part of the file: they are included for ease of discussion here.

We see from Figure 18 that there are three sections within the UserText field. Each is enclosed in curly brackets (`{}`). The first section starts on line 7, the second on line 18, and the third on line 30. The sections each begin with the button label, terminated by the '`|`' (vertical bar) character. Respectively, these are "Normal Blood Pressure," "Types of Blood Pressure," and "Possible treatments."

Figure 19 shows the effect when the node is displayed in WaT mode. The three headings are displayed at the left-hand side of the upper pane, and are clickable (which is why we call them "button labels"). The currently-selected section is indicated by a (red) dot.

5.5 The DocumentingText Field

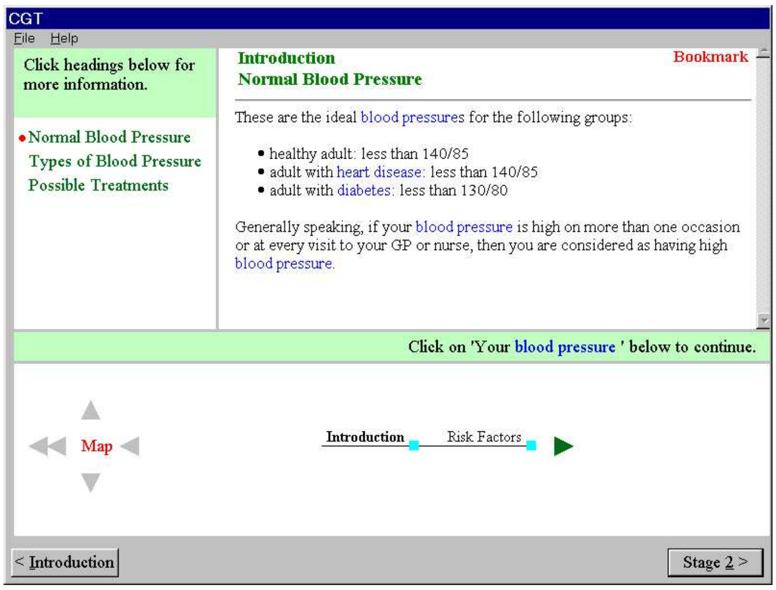
The (optional) DocumentingText node field is similar to the UserText field: it too contains text that explains the node. However, the UserText material is intended to be shown to the ordinary lay user of the tree, while the DocumentingText contains technical notes for the tree author(s) or their peers. For example, it could be used to give bibliographic and other information about the studies consulted by the author while writing this part of the tree. This facility to store technical supporting documentation as part of the tree is very useful. It means that the tree author's own notes are stored in the most relevant place: next to the tree structure itself.

The field can contain DisplayText, but cannot be split into sections like the UserText field. Figure 20 gives an example.

DocumentingText is only visible if the user checked a small box on the screen for choosing a file: see Figure 6. If that box is checked, then "Research Evidence" appears as an extra button on WaT mode screens, after the last button produced by splitting the UserText field (see Subsection 5.4, above). For example, in Figure 19 it would appear as a fourth clickable heading, under "Possible Treatments," on the left hand side of the upper pane.

```
ID:Root
ShortLabel: Introduction
LongLabel: Introduction
ParentID:
Type: Decision
UserText:
  {Normal Blood Pressure| These are the ideal blood
    pressures for the following groups:
    <ul>
    <li>healthy adult: less than 140/85
    <li>adult with heart disease: less than 140/85
    <li>adult with diabetes: less than 130/80
    </ul>
    Generally speaking, if your blood pressure is high
    on more than one occasion or at every visit to your
    GP or nurse, then you are considered as having
    high blood pressure.}
  {Types of Blood Pressure|There are different types
    and causes of high blood pressure.
    <p>Essential or primary high blood pressure is
    the most common type of high blood pressure
    and the type that you have.
    There is no specific disease process involved
    and in 9 out of 10 people there is no single cause.
    It is probably a result of a number of contributing
    factors, some of which we can control such as our
    lifestyle, others which we cannot such as hereditary
    factors or age. <p>Secondary high blood pressure is
    related to a specific disease process and is less common.}
  {Possible Treatments|There are people whose risk of
    heart disease and stroke is  not high enough to mean
    that they need treatment with tablets.
    In these people it is suggested that they make changes
    to their lifestyle.
    <p>
    In about 25% of people who  need to take tablets,
    their blood pressure will reduce by taking one tablet.
    However, other people may require a second or third tablet.
    <p>
    If you make changes to your lifestyle, then this may
    reduce the need for you to have to take more than one
    tablet for your blood pressure.}
```

Figure 18: Splitting the UserText field



Click headings below for more information.

● Normal Blood Pressure
Types of Blood Pressure
Possible Treatments

Introduction
Normal Blood Pressure

These are the ideal blood pressures for the following groups:

- healthy adult: less than 140/85
- adult with heart disease: less than 140/85
- adult with diabetes: less than 130/80

Generally speaking, if your blood pressure is high on more than one occasion or at every visit to your GP or nurse, then you are considered as having high blood pressure.

Click on 'Your blood pressure ' below to continue.

Figure 19: Splitting UserText

```
ID:Root
ShortLabel: Introduction
LongLabel: Introduction
ParentID:
Type: Decision
UserText:
{Normal Blood Pressure|
  ...}
{Types of Blood Pressure|
  ....}
{Possible Treatments|
  ...}
DocumentingText:
  The figures for normal blood pressure have been taken
  from ...
  <p>
  In 17 unconfounded trials of pharmacological treatment ...
  <H3>References:</H3>
  British Cardiac Society, ....
  <p>
  Collins R MacMahon S ...
  ...
```

Figure 20: The DocumentingText field

6 Expressions

Section 4 introduced the concept of the *Evaluable Expression* (or just *Evaluable*), that is, a formula ("expression") that can be worked-out ("evaluated") when its value is needed. For example, if we have a variable `pFine` (the probability of Fine Weather), we can have the Evaluable `(1-pFine)`, giving, obviously, the probability of rain. These Evaluables can be used in many places in a tree: for example, in the Prob and Payoff fields. In this section we give a complete account of CGT Evaluables.

1. Evaluable expressions can use addition (`+`), subtraction (`-`), multiplication(`*`), division(`/`), raising to a power (`x^y`, x is raised to the power y), the functions `log(x)` (natural logarithm) and `exp(x)` (e^x, antilog), and the use of brackets. These operators have their usual relative priorities. Raising to a power is evaluated right-to-left, so `2^3^2` gives 512 and not 64.

2. An Evaluable can, in addition to representing a numeric expression, also represent a Boolean expression, i.e. one that evaluates to either `true` or `false`. To make this possible we use use the convention that zero represents `false` and any other value represents `true`. We can use the relational operators (`> >= < <=`), the equality operators (`==` and `!=`), and the logical operators (`and`, `or`, and `not`). We can now write such expressions as

   ```
   male and age > 15
   ```

   (assuming we have declared variables `male` and `age`). This expression will be `true` if `male` is non-zero (i.e. `true`) and `age` is greater than 15.

   Take care not to write the is-equal-to operator `==` as `=` . This mistake, notorious in C and C++, has catastrophic consequences in CGT too. Suppose you meant to write

   ```
   IF sex == male THEN risk = 0.8 ELSE risk = 0.5 FI
   ```

   but instead you used the assignment operator in the condition, and wrote

   ```
   IF sex = male THEN risk = 0.8 ELSE risk = 0.5 FI
   ```

   what happens is not what you expect, or what you want. In the condition, the current value of `male` is *assigned to* `sex` (not compared with it). If that value is non-zero (i.e. `true`), then the condition is `true` and `risk` always gets the value 0.8, regardless of the previous value of `sex`. Or, if the current value of `male` is zero (i.e. `false`), then the condition is `false` and `risk` always gets the value 0.5, regardless of the previous value of `sex`. This is a disaster.

3. For convenience, there are two Boolean constants, `true` and `false`. The first has some fixed non-zero value (in fact its value is 1), and the second has the fixed value zero. The user cannot define variables with these names, and the names cannot be used on the left-hand side of an assignment.

4. The assignment operator (`=`) can be part of an Evaluable. Thus, in the line

   ```
   annualSalary = monthlySalary * 12
   ```

   the *whole of* the line is an Evaluable expression. The value of an assignment operator is the value that was placed in the variable on the left-hand side. Thus in the example above, if `monthlySalary` has the value (say) 1,500 then the value 18,000 is placed in the variable `annualSalary`, and 18,000 is the value of the whole expression. The assignment operator is evaluated right-to-left, so the Evaluable

   ```
   i = j = k = l = m = n = 0
   ```

   has the effect of setting all the variables `i` to `n` to zero. (And the whole expression has the value zero.)

5. An Evaluable expression can consist of more than one sub-expression, separated by semicolons. The value of the whole Evaluable is the value of the last sub-expression: the rest are discarded. Although only the last sub-expression gives the value of the Evaluable, the others may have useful effects. So, for example, in the following Evaluable

   ```
   annualSalary = monthlySalary * 12 ; annualSalary > 30000
   ```

   we set the value of the variable `annualSalary` and test it: the value of the Evaluable is `true` if the annual salary is greater than 30,000, and `false` otherwise.

6. An Evaluable can contain an *if-expression.* There are two forms: an if-expression is either

```
    IF expression1 THEN expression2 FI
```

or

```
    IF expression1 THEN expression2 ELSE expression3 FI
```

In both cases, don't forget the `FI`, which has the effect of terminating the `IF`. In both forms, if `expression1` evaluates to `true`, then `expression2` is evaluated and its value is the value of the if-expression. If `expression1` evaluates to `false`, then in the second form `expression3` is evaluated and its value is the value of the if-expression; in the first form, nothing is evaluated and the if-expression has no value. (The reserved words `IF` etc are not case-sensitive: e.g. `IF` can be written as `if`.)

These syntax rules allow a tree author to use a variety of styles of programming. For example, suppose that we have a variable `age` and wish to create a derived variable `ageCategory`. This is to have the value 1 for people below 20, 2 for 20 to 39, and 3 otherwise. Here are two ways of expressing this: the author can user whichever they find clearer and easier to write.

```
    IF age < 20 THEN                  ageCategory =
      ageCategory = 1                   IF age < 20 THEN 1
    ELSE IF age < 40                    ELSE IF age < 40 THEN 2
      ageCategory = 2                   ELSE 3 FI FI
    ELSE
      ageCategory = 3
      FI FI
```

In the style on the left, the expressions inside the `IF`s are written as assignments. These could be sequences of assignments. Suppose we have two probabilities, `p1` and `p2`, that should have different values for the age categories. We could write this as follows (taking arbitrary values for the probabilities):

```
    IF age < 20 THEN
      ageCategory = 1 ; p1 = 0.2 ; p2 = 0.15
    ELSE IF age < 40
      ageCategory = 2 ; p1 = 0.3 ; p2 = 0.45
    ELSE
      ageCategory = 3 ; p1 = 0.5 ; p2 = 0.55
      FI FI
```

7. Table 2 shows the operators in descending order of precedence. In that Table, horizontal lines separate the levels of precedence, so operators with the same precedence (e.g. multiply and divide) do not have a line between them.

8. A variable that has been declared but not given a value is *undefined.* This does *not* mean that it has some special numeric value. It means that it does not have a numeric value at all. Any operation, except one, on an undefined value yields an undefined value. So if `y` is undefined then `y+2` is undefined, and `y==0` is undefined (and not `false`, surprisingly). The single exception is the Boolean function `defined`: thus `defined(y)` is either `true` or `false`. For convenience, we define the constant, `undefined`. This constant is (of course) undefined. There is only one useful thing that can be done with this: to assign it to a variable, thus:

   ```
   y = undefined
   ```

   This has the effect of making `y` undefined. Note that the Boolean expression `y == undefined` will always give the undefined value, whether `y` is defined or not, so this will not test the value of `y`. The only possible way of seeing if `y` is undefined is to use the `defined` function, as shown earlier in this paragraph.

   If the condition part of an `if` (the `expression1`) is undefined, a run-time error message is generated.

9. An Evaluable expression is evaluated when its value is needed. For example, an expression initialising a variable being declared in the TreeVariable field is evaluated when the tree is being read. Expressions in the Prob and Payoff fields of Terminal nodes are evaluated when the optimimum path is being calculated in SaT mode.

Of course, an expression may be evaluated several times. The tree author should remember this fact when changing the values of variables. For example, suppose that a variable called `Counter` was initialised to zero when it was declared. Suppose also that it occurs elsewhere in the tree, in the evaluable

```
Counter = Counter + 1
```

Obviously, every time this is evaluated, the value of the variable will increase by one. For example (anticipating material to be covered in Section 7) suppose that the UserText field of a node contains the text

```
You have viewed this node McEval(Counter = Counter + 1,0) time(s)
```

This gives the user a count of the number of times they have visited the node. (The count will only be correct if the variable's value is not changed elsewhere in the tree, of course.)

10. You may have been wondering how variables like `sex` and `income` got their (user specific) values in the first place. This will be explained in Section 8, on Question nodes. That Section also has more examples of Evaluables.

Operator	*Meaning*	*Example*
`()`	Brackets round (sub)expression	`(a + b)`
`exp`	e^x	`exp(Income)`
`log`	$log_e x$	`log(Income)`
`^`	Raise to power	`Income^2`
`*`	Multiply	`a*b`
`/`	Divide	`a/b`
`-`	Subtract	`a-b`
`+`	Add	`a+b`
`!=`	Not-equals	`sex != 1`
`<`	Less-than	`income < 1000`
`<=`	Less-than or equal-to	`income <= 1000`
`==`	Equal-to	`income == 1000`
`>`	Greater-than	`income > 1000`
`>=`	Greater-than or equal-to	`income >= 1000`
`defined`	Has valid value	`defined(income)`
`not`	Logical not	`not old`
`and`	Logical and	`male and old`
`or`	Logical or	`male or old`
`=`	Assign-to	`pRain = 1 - pFine`
`;`	End-expression	(see text)
`else`	Else part of conditional	(see text)
`fi`	End of conditional	(see text)
`if`	Start of conditional	(see text)
`then`	Then part of conditional	(see text)
The relational and logical operators all return zero (false) or 1 (true). Any non-zero value is true.		
The alphabetic operators are not case-sensitive		

Table 2: Operators in Evaluable Expressions

7 Macros

7.1 Introduction

We mentioned in Section 5 that DisplayText (the format used in a number of fields, notably the UserText field) can contain *macros*. In this section we give a complete account of the use of macros in CGT. Our method is based on the Unix macroprocessor `m4` developed by BM Kernighan and DM Richie. See [7] and [8].

At their simplest, macros are pieces of text that are replaced by other pieces of text as the field is being displayed. For example, if the tree author finds that they often repeat a stock phrase (like, say, "ask your Doctor if you would like to know more"), then instead of entering this in full every time, the tree author can define a shorthand form (like `AskDr`) and use that instead. When the program displays the text, the shorthand is expanded to its full form. The shorthand is called "the macro" and the full form is called "the macro expansion" or "the macro replacement text". Here is an example:

```
McDefine(AskDr,‘ask your Doctor if you would like to know more’)
This is a complicated subject: AskDr. In fact, you should always
AskDr about anything that worries you.
```

In this example, we see that the macro `AskDr` is first defined, then used. The definition is done with the *built-in macro* `McDefine`. The text would appear as:

```
This is a complicated subject: ask your Doctor if you would like to
know more. In fact, you should always ask your Doctor if you would like
to know more about anything that worries you.
```

The most important use for macros in CGT is to manipulate and display tree variables. To do this we use the built-in macro `McEval`, which evaluates Evaluable expressions (described in Section 6) and turns the result (a number) into text. For example, suppose we have a tree variable `pHeadache` (probability of Headache). We could then write:

```
 .. this occurs in McEval(pHeadache*100,0)% of cases
```

The output would of course depend on the current value of `pHeadache`, which will be formatted (by truncation, not rounding) to zero decimal places. If `pHeadache` had the value (say) 0.2, we would obtain the output

```
.. this occurs in 20% of cases
```

7.2 Defining a macro

Macros can be defined in two places: either in the in the GlobalMacros tree field, or "on the fly" within tree or node fields. Global macros define a set of macros which you can use throughout the entire tree file. For example, you may define a macro to produce the text for a UserText field heading. This heading text can then be changed on all nodes by simply changing the macro. In contrast, a macro defined within a node record field is local to that field and cannot be used outwith that field. Unlike the TreeVariable field, where a separate field is required for each variable, any number of macros can be specified, one after the other, within the GlobalMacros field. Later in this section we shall see an example of definitions like this.

As we saw in the example at the start of this section, new macros are defined using the built-in macro `McDefine`. Notice, incidentally, that the names of macros are case-sensitive. `McDefine` is the name of a macro: `mcdefine` is not.

In talking about macros, we refer to a macro's *arguments*: the text enclosed in brackets after the macro's name. The `AskDr` macro, discussed above, has no arguments. Other macros, however, have a bracketted list of items, separated by commas, after their name. These items are arguments. Suppose we have a macro `Swap(this,that)`. Here, there are two arguments: `this` and `that`.

Unlike most macros, `McDefine` does not produce any replacement text. Instead, its effect is that all subsequent occurrences of the first argument are replaced by the second argument, just as in the definition of the `AskDr` macro, above.

In that example, `McDefine`'s second argument was enclosed in special quote marks. These have the effect of preventing the replacement text from itself being scanned for macros. This is almost always what we want and so it is good practice to quote the replacement text in this way. The quote marks are the left-single-quote character (to the left of '1' on my keyboard) and the "ordinary" single-quote character (to the right of semicolon).

Now suppose we wish to define a macro that will itself take arguments. For example, suppose we want `IfThen(indigestion,week)` to turn into the phrase `If you experience indigestion, see your Doctor within the next week`, whereas `IfThen(pain,day)` is to turn into `If you experience pain, see your Doctor within the next day`. We can define this macro like this:

```
McDefine(IfThen,‘If you experience $1,
  see your Doctor within the next $2’)
```

In this definition the terms `$1` and `$2` stand for the first and second arguments found when the macro is actually used. Macros can have up to nine arguments, so definitions can use `$1` up to `$9`.

7.3 Other built-in macros

There are three other built-in macros, in addition to `McDefine`. These are

McChangeQuote This macro is used to change the macro quote characters from the default left-and-right single quote. (Perhaps you want to use the quote characters literally in the replacement text.) For example, `McChangeQuote(<,>)`. If used without arguments it re-instates the default quote characters ‘’. If used with arguments, there must be two arguments and each must be a single character.

McEval As we saw above, the first argument to this macro must be an Evaluable expression. The replacement text is a character representation of the floating-point value that results from the evaluation. The macro can be given a second argument, a single digit giving the maximum number of decimal places to be shown. (The value is truncated, not rounded. If you want rounding, add 0.5, 0.05, or whatever is appropriate.) If the second argument is missing, the system decides how many decimal places to use.

McIfElse This allows for the conditional insertion of text, depending on a comparison of pairs of other text values. For example, suppose we wanted to write either

```
.. see your Doctor about this
```

or

```
.. see your Practice Nurse about this
```

then we could write the macro

```
McDefine(SEE,‘see your McIfElse($1,D,Doctor,$1,P,Practice
                Nurse,ERROR) about this’)
```

and then subsequently write either `SEE(D)` or `SEE(P)` to produce the required phrases.

The basic forms of `McIfElse` have either three or four arguments. `McIfElse(a,b,c)` is replaced by `c` if the text `a` is the same as the text `b`, otherwise there is no replacement. `McIfElse(a,b,c,d)` is replaced by `c` if the text `a` is the same as the text `b`, otherwise the replacement is `d`. However, there are more elaborate forms: `McIfElse(a,b,c,d,e,f)` is replaced by `c` if the text `a` is the same as the text `b`, otherwise the replacement is `McIfElse(d,e,f)`. `McIfElse(a,b,c,d,e,f,g)` is replaced by `c` if the text `a` is the same as the text `b`, otherwise the replacement is `McIfElse(d,e,f,g)`.

One common mistake is to think that `McIfElse` is a way of testing the values of variables directly. Suppose the variable `sex` has the value 1 if the user is a man, and 2 if the user is a woman. The tree author then writes

```
Because you are a McIfElse(sex,1,man,woman)  */ Horribly wrong
```

This is wrong because `McIfElse` is a way of comparing text. The text "sex" is never equal to the text "1"! We could rewrite the example as

```
Because you are a McIfElse(McEval(sex,0),1,man,woman)
```

This will output `Because you are a man` if the variable `sex` has the value 1, and `Because you are a woman` otherwise. It works because `McEval(sex,0)` produces either the text "1" or "2".

7.4 Examples

Let us look at some examples. First, suppose the tree author defines a macro in the GlobalMacros tree field,

```
McDefine(MildAUA, ‘AUA symptom index 0-7 points’)
```

Then every time the tree author writes, for example

```
For MildAUA the likely outcome is ...
```

the program will display

```
For AUA symptom index 0-7 points the likely outcome is ...
```

As a second example, let us take the display of the `pEvent` variable mentioned above. Using macro-processing, the tree author simply replaces

```
the probability of this is about 0.2
```

by the following

```
the probability of this is about McEval(pEvent)
```

and every time the program displays this, it will substitute the correct current value of `pEvent`.

Now a more elaborate example. The details of this example rely on material to be covered later in Section 7, but it is perhaps useful here as a demonstration of the power of the technique. Suppose there is a tree variable `SymptomScore` that holds the current user's score in terms of some index (we shall see later how this can be done). If the following macro is defined in the GlobalMacros tree field,

```
McDefine(SaySymptoms,‘McIfElse(McEval(SymptomScore<8,0),1,mild,
  McEval(SymptomScore>20,0),1,severe,moderate) symptoms’)
```

Then the tree author can write such DisplayText as

```
Your symptom score is McEval(SymptomScore,0).
This means that you have SaySymptoms.
```

and the program will convert this to a text that is tailored to the user. If, for example, the user's symptom score is 15, then the program will output

```
Your symptom score is 15.
This means that you have moderate symptoms.
```

How does this example work? Let us unpack `SaySymptoms` from the inside outwards, following the steps of the evaluation. We begin by looking at the first argument of the `McIfElse`.

- `SymptomScore<8` is `false`, i.e. zero
- `McEval(0,0)` is zero formatted with no decimal places, i.e.‘0’. As such it is not the same as the second argument of the `McIfElse`, because that is ‘1’. So we skip the third argument and look at the fourth argument.
- `SymptomScore>20` is also `false`, i.e. zero
- `McEval(0,0)` once again is ‘0’, and once again it is not the same as the fifth argument of the `McIfElse`, because that is ‘1’. So we skip the sixth argument. This leaves the seventh argument, “moderate,” as the replacement text of the macro.

Now suppose the user's symptom score is 7. This should give the result “mild”. Once again let us unpack `SaySymptoms`, following the steps of the evaluation. We begin by looking at the first argument of the `McIfElse`.

- `SymptomScore<8` is `true`, i.e. 1
- `McEval(1,0)` is 1 formatted with no decimal places, i.e.‘1’. As such it is the same as the second argument of the `McIfElse`, because that is also ‘1’. So the third argument, “mild,” is the replacement text of the macro.

8 Asking Questions

We have seen how to declare and initialise a variable, using the TreeVariable tree field, and how the value can be used and changed in Evaluable expressions. In this section we see how a variable's value can be obtained at run-time from the user.

8.1 The Question node

Our method uses a new kind of node, the *Question node.* This node type is incorporated into the tree in the same way as the other types, by specifying the ID of its parent in the ParentID field. A Question node can have zero, one or two children, but not three or more children. If a question node has one child, that child can be a node of any type (Decision, Chance, Terminal or Question). If it has two children, then the first child must be a Question node and the second must be a non-Question node (i.e. Decision, Chance or Terminal).

We begin with a very simple example, in which we incorporate a Question Node in the simple tree of Section 2. We'll ask the user to supply their own estimate of the probability of rain. We could do this is a number of ways: the most obvious is simply to add a Question node to the front of the existing tree. The resulting tree with its probabilities and payoffs is shown in Figure 21. Notice that we represent a Question node as a diamond.

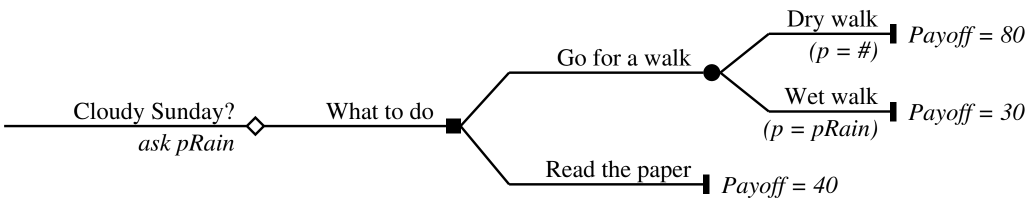


Figure 21: Adding a Question node

The coding of the tree file is shown in Figure 22. In Figure 22 the lines are numbered at the left-hand side. These numbers are not part of the file: they are included for ease of discussion here.

A Question node asks its question when it becomes the current node in WaT mode. This happens either because the user navigates to that node, either explicitly or by default. In the present example it happens by default, because the Question node is also the root. Figure 23 shows the display on the screen as the node asks its question. We see that the question occupies the lower pane of the window, replacing the normal WaT display of the current node. The upper pane is available for the display of the UserText field as normal.

In Figure 22 we note that line 4 declares a variable, `pRain` and gives it a default value. (This value will be used if the user skips the Question node.) Lines 6 to 15 are the Question node. We see that it has five fields not used in other node types. These are lines 11 to 15. In order, these fields are

QuestionText The text of the question to be asked. This should be just plain text (no HTML). This field is required.

InputType This field is required. The possible types are:

Edit(n) A box in which the user can type a number. The maximum number of characters is given in brackets.

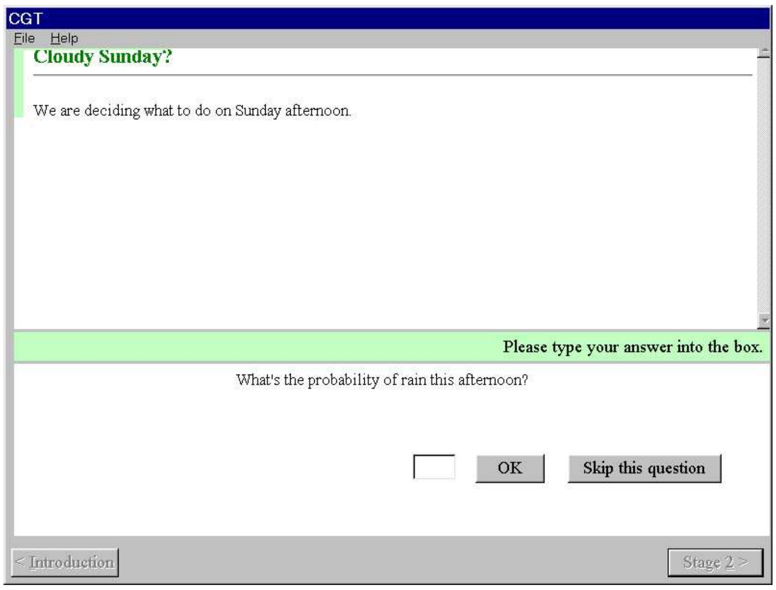
Combo(a,b,c, ...) A dropdown list of possible answers, which are given in brackets.

Radio(a,b,c) Radio buttons for the possible answers, which are given in brackets. The first button in the list is depressed when the list is displayed, and is the default answer. There can only be two or three answers in the list (if you need more, use a Combo).

HeightScale A slider for heights between 1.5 and two metres, given in both Metric and Imperial measure.

```
TreeName: Sunday walk
IntroText: This tree helps you decide what to do on Sunday afternoon.
RequiresVersion:2.4
TreeVariable: pRain = 0.7

ID: Root
ShortLabel: Cloudy Sunday?
ParentID:
Type:Question
UserText: We are deciding what to do on Sunday afternoon.
QuestionText: What's the probability of rain this afternoon?
InputType: Edit(4)
BoundVariable: pRain
ValidationRule: pRain >= 0.0 AND pRain <= 1.0
ValidationRuleText: That's not valid.  Enter a number between 0.0 and 1.0

ID: 1
ShortLabel: What to do
ParentID: Root
Type:Decision
UserText: So, the probability of rain is McEval(pRain,2).  Should we walk,
  or should we read the paper?  Explore the tree to see what's involved.

  ID: 2.1
  ShortLabel: Go for a walk
  ParentID: 1
  Type: Chance
  UserText:
    If we go for a walk, the weather may clear
    and the sun may come out: or it may turn out wet.

    ID: 3.1
    ShortLabel: Dry walk
    ParentID: 2.1
    Type: Terminal
    Prob: #
    Payoff: 80
    UserText: The weather clears and we have a pleasant walk.

    ID: 3.2
    ShortLabel: Wet walk
    ParentID: 2.1
    Type: Terminal
    Prob: pRain
    Payoff: 30
    UserText: The weather worsens and it rains.  We have a wet walk.

  ID: 2.2
  ShortLabel: Read the paper
  ParentID: 1
  Type: Terminal
  Payoff: 40
  UserText: Reading the Sunday papers is OK, up to a point,
    but it leaves you feeling vaguely dissatisfied.
```

Figure 22: The file `SundayQ.tree`



Cloudy Sunday?

We are deciding what to do on Sunday afternoon.

Please type your answer into the box.

What's the probability of rain this afternoon?

Figure 23: Asking a question

WeightScale A slider for weights between 40 and 145 kilograms, given in both Metric and Imperial measure.

BloodPressure A stacked pair of edit boxes labelled "Systolic" and "Diastolic".

BoundVariable The name of the variable that will receive the user's input. The variable must have been declared in a TreeVariable field. If the InputType is `BloodPressure` then there should be two variables names, separated by a vertical bar. This field is required.

The different InputTypes work as follows:

Edit(n) If the characters entered by the user make up a number (possibly signed and/or fractional), that number is transferred to the variable. Otherwise it's an error (see the ValidationRuleText field, below).

Combo(a,b,c, ...) A number corresponding to the position in the list is transferred. For example, if the user chooses the third item, the number 3 is transferred.

Radio(a,b,c) A button labelled "Yes" (case-insensitive) always has the value 1, and a button labelled "No" (case-insensitive) always has the value zero. Otherwise, a number corresponding to the position in the list is transferred. For example, if the user chooses the third item, the number 3 is transferred.

(And if you mix these conventions you will need to keep a steady head. {"Yes", "No", "Maybe"} will map to {1,0,3}, and {"No", "Maybe", "Yes"} will map to {0,2,1}, which are tolerable, but {"Maybe", "No", "Yes"} will map to {1,0,1}, which isn't good at all.)

HeightScale The selected value in metres is transferred.

WeightScale The selected value in kilograms is transferred.

BloodPressure As with Edit, above, with the Systolic value being transferred to the first variable and the Diastolic to the second.

ValidationRule This field is optional (but strongly recommended if you are using the Edit or BloodPressure InputTypes). If used, it should contain an Evaluable expression (see Section 6). If it evaluates to `false` (zero) then the input is treated as being in error (see the ValidationRuleText field, below). Obviously, the most direct way of using the field is to write a rule that tests the value(s) just entered by the user, as in line 14 of Figure 22. We shall see later that the field can contain programs with complex effects.

ValidationRuleText This field is optional. If used, it should contain plain text to be displayed if an error condition is detected. If the field is not present, then CGT uses its own error message. The error message is displayed in a pop-up box: when the user dismisses the box, they get another chance to input a value.

As Figure 23 shows, the user can opt to skip a question. If so, the next node becomes current, and the value of the BoundVariable is unaffected. Because users can skip questions in this way, it is highly desirable that the variables are given sensible default values (as in line 4 of of Figure 22).

8.2 Run-time behaviour

Question nodes have quite complex behaviour patterns. They have *Entry behaviour*: what they do when they become the current node. They also have *Exit behaviour*: what they do when they cease to be the current node. Included in their exit behaviour are the rules for deciding which node is the new current node. These behaviours are affected by whether the Question node is *satisfied*: that is, whether the user has given an answer to it (rather than skipping the question).

These behaviour patterns are defined elsewhere (see [1]). The idea behind them is that they should lead to behaviour that seems "natural" to the user. We do not give a detailed discussion here, because the tree author does not need to know these rules in detail. The author only needs to follow a conventional way of incorporating questions into trees. If this standard pattern is used, then the tree will behave "naturally". This "natural" behaviour is basically obvious (though the rules for implementing it are complex). The node should appear to recognise when it is being encountered for the first time, and when a user has browsed back to it. On the first visit, it asks its question. On subsequent visits it allows the user to change the previous response, or keep it. It should allow the user to skip a single question. If there is a run of questions, it should allow the user to skip all the questions.

The standard pattern for incorporating Question nodes is discussed in the following subsection.

8.3 Question sequences

A Question node can be the child of any type of node (except a Terminal, obviously). And, as we saw above, a Question node can have zero, one or two children, but not three or more children. If a question node has one child, that child can be a node of any type (Decision, Chance, Terminal or Question). If it has two children, then the first child must be a Question node and the second must be a non-Question node (i.e. Decision, Chance or Terminal).

If a tree author would like to ask a single question at some point in the tree, they simply insert a single Question node, much as we saw in the example above. If, however, a sequence of questions is needed, these should be written so that the sequence forms a side-branch on the tree. The first Question node in the sequence will have two children: the first child will be the second question in the sequence, and the second child will be a node of some other type (probably the root of some sub-tree). The subsequent Question nodes will only have one child, the next question in the sequence, until the last question is reached: this final node will be childless.

Let us extend our Sunday walk tree to provide an example. Suppose metereologists have produced a formula for our part of the world, whereby the probability of rain can be estimated by asking three questions:

- Did it rain yesterday?
- How much cloud cover is there?
- What's the relative humidity?

(The plausibility of this example is not important here.) Adding these questions to the tree gives us the configuration shown in Figure 24. We see the side-branch of questions at the top of the Figure.

Before we can code the tree for CGT, we need to know more about how the questions will be asked, and how the answers will be used. We can ask the questions in a number of ways. The first question is just a yes/no: we can use a radio control for this. Let's handle Cloud Cover by offering the user four alternatives (up to 25%, up to 50%, etc). Finally we can just ask the user to type-in the (relative) Humidity, as a number between 0 and 100 inclusive.

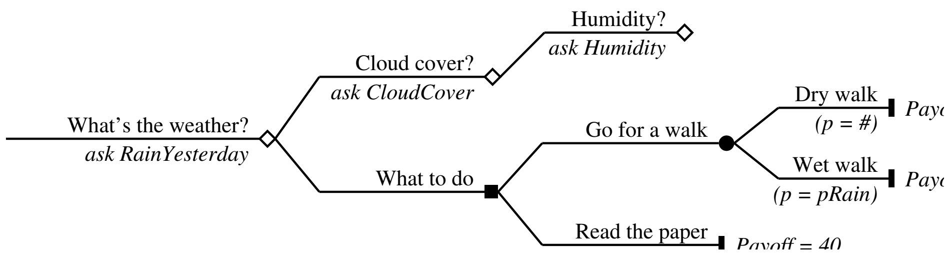


Figure 24: Adding a sequence of Question nodes

Turning to how the answers will be used to compute the probability, let's suppose the metereologists' formula is a regression equation, of the form

$$p = b_1Y + b_2C + b_3H + c \tag{1}$$

where Y is 1 if it rained yesterday, and zero otherwise, C is the percentage cloud cover, and H is the percentage relative humidity. As with any regression equation, the b's are the regression coefficients, and c is a constant. Let's say the metereologists have supplied values such as

$$p = 0.21Y + 0.0025C + 0.0023H + 0.2 \tag{2}$$

We shall need to declare at least four variables: RainYesterday, CloudCover, Humidity, and pRain. It is perhaps neater if we code the regression coefficients and constant as variables too. This gives us the eight declarations shown in Figure 25. (Once again these lines from the tree file have had line numbers added, for ease of discussion.) The first four declarations, on lines 4 to 7, have been given default values in case the user skips the questions. The last four declarations initialise the variables using values from the regression equation.

```
 4. TreeVariable: pRain = 0.7              */ Pessimistic default
 5. TreeVariable: RainYesterday = 0        */ Rain yesterday, 0 or 1
 6. TreeVariable: CloudCover = 50          */ Cloud cover, 0 .. 100
 7. TreeVariable: Humidity = 30            */ Humidity 0 .. 100
 8. TreeVariable: wtRainYesterday = 0.21   */ Regression coefficient
 9. TreeVariable: wtCloudCover = 0.0025    */ Regression coefficient
10. TreeVariable: wtHumidity = 0.0023      */ Regression coefficient
11. TreeVariable: const = 0.2              */ Regression constant
```

Figure 25: Estimating the probability of rain: declarations

We can now write the three Question nodes, shown (with added line numbers) in Figure 26.

- The first node, "Rain yesterday?" is straightforward. Because the InputType is a Radio, we don't need to check the user's input. Nor do we need to transform it, as it is already in the form we require, a zero/one variable.
- The second node, "Cloud cover?" is a little more complex. Because the InputType is a Combo, we don't need to check the user's input. But we do need to transform it, to map it from [1..4] to [0..100]. We can do this by a small program in the ValidationRule field (line 32 of Figure 26). This Evaluable has two sub-expressions: the first carries out the required conversion, and the second simply leaves the value `true` to make the ValidationRule "succeed". (In fact, this could be omitted, because the the first sub-expression always has a positive value and so is always "true". But it is clearer to leave the explicit `true` value, as we have here.)
- The third node, "Humidity?" must deal with two issues. The first is that the user can make mistakes in entering a percentage value in the Edit box: for example, they might enter 200, or

```
ID: Root
ShortLabel: What's the weather?
ParentID:
Type:Question
UserText: We are deciding what to do on Sunday afternoon.
  We need to ask a series of questions about the weather.
QuestionText: Did it rain yesterday?
InputType: Radio(No,Yes)
BoundVariable: RainYesterday

  ID: Q2
  ShortLabel: Cloud cover?
  ParentID: Root
  Type:Question
  UserText: To assess the probability of rain, it helps
    to know how cloudy it is.
  QuestionText: How much of the sky is cloudy?
  InputType: Combo(Up to 25%, Up to 50%, Up to 75%, Up to 100%)
  BoundVariable: CloudCover
  ValidationRule: CloudCover = CloudCover*25 - 12.5 ; true

    ID: Q3
    ShortLabel: Humidity?
    ParentID: Q2
    Type:Question
    UserText: To assess the probability of rain, it helps
      to know the humidity.
    QuestionText: What's the humidity?  (0 to 100)
    InputType: Edit(4)
    BoundVariable: Humidity
    ValidationRule: IF Humidity >= 0 AND Humidity <=100 THEN
                       pRain = wtRainYesterday * RainYesterday
                             + wtCloudCover * CloudCover
                             + wtHumidity * Humidity
                             + const ;
                       true
                    ELSE false FI
    ValidationRuleText: Humidity must be between 0 and 100
```

Figure 26: Estimating the probability of rain: questions

even a non-numeric value. Secondly, it is at this point that we need to compute `pRain` from the user's answers to all three questions. The resulting ValidationRule is lines 43 to 49 of Figure 26. If the user has made a mistake in entering the humidity, the Evaluable "fails" (and the user sees the ValidationRuleText of line 50). Otherwise, `pRain` is computed and `true` is left to ensure that the Evaluable "succeeds."

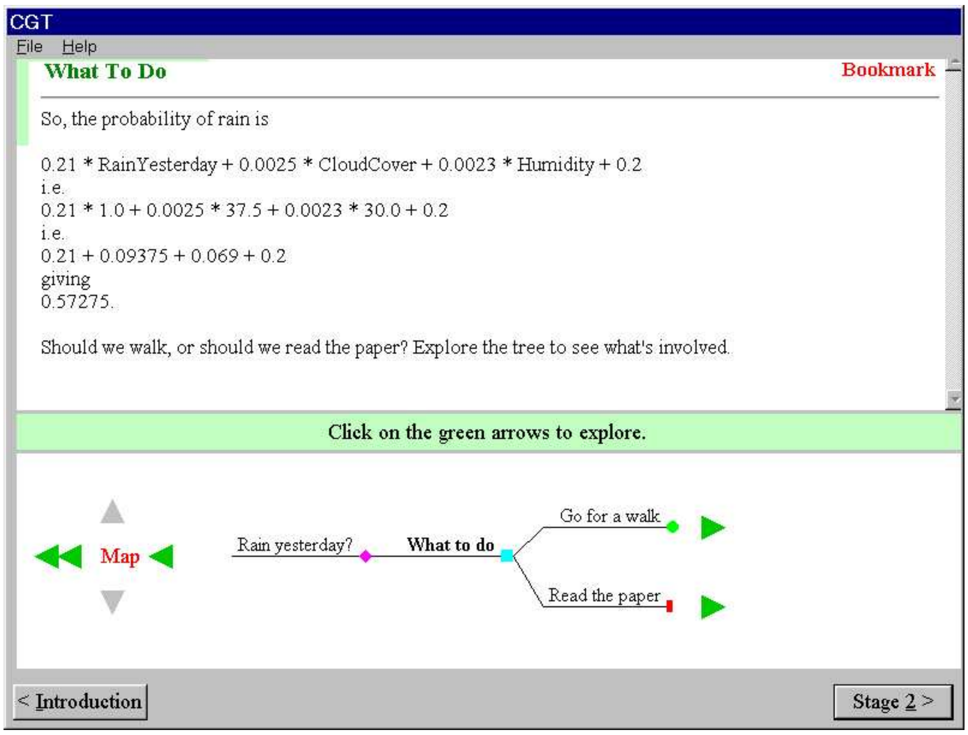
One benefit of using variables to hold all the values, constants as well as answers to questions, is that the computation in lines 44 to 47 is very clear: it is obvious that it follows the regression equation given above. Another benefit is that we can make the whole computation visible to the user. We can display all the values at the node following the sequence of questions. This is the node "What to do" in Figure 24. The text of this node is shown (with added line numbers) in Figure 27. We see that the UserText of the

```
52. ID: 1
53. ShortLabel: What to do
54. ParentID: Root
55. Type:Decision
56. UserText: So, the probability of rain is
57.   <p>
58.   McEval(wtRainYesterday) * RainYesterday +
59.   McEval(wtCloudCover) * CloudCover +
60.   McEval(wtHumidity) * Humidity +
61.   McEval(const)
62.   <br>i.e.<br>
63.   McEval(wtRainYesterday) * McEval(RainYesterday) +
64.   McEval(wtCloudCover) * McEval(CloudCover) +
65.   McEval(wtHumidity) * McEval(Humidity) +
66.   McEval(const)
67.   <br>i.e.<br>
68.   McEval(wtRainYesterday*RainYesterday) +
69.   McEval(wtCloudCover*CloudCover) +
70.   McEval(wtHumidity*Humidity) +
71.   McEval(const)
72.   <br>giving<br>
73.   McEval(pRain).
74.   <p>
75.   Should we walk, or should we read the paper?
76.   Explore the tree to see what's involved.
```

Figure 27: Echoing the data values using McEval

node makes heavy use of the McEval macro to walk the user through the computation. This is perhaps most useful while the tree author is debugging the tree: this amount of detail is probably too much for non-technical users. Figure 28 shows the screen image of this node, after the user has said that it did rain yesterday, the cloud cover is up 50%, and the humidity is 30%. The successive lines of output show the original regression equation, the substitution of the values for the variables, and finally the computation of the value for `pRain`.

Finally, a note on the visibility of Question nodes. We have seen that the correct way to code a sequence of contiguous questions is as an "upward" spur. In both WaT and SaT modes the user can see only the Question node forming the root of this spur. Thus, although Figure 24 shows us that "Cloud cover?" is a sibling of "What to do", the user cannot see "Cloud cover" when "What to do" is the current node in WaT mode. In fact, a Question node on a spur cannot be seen when any of its relatives is the current node. The only way of getting to a node on a spur, such as"Cloud cover," is through its parent question. As a result, the labelling and wording of the first question must make it clear that the remaining questions are present. Thus the node "What's the weather," the root of the spur of questions, has a dual function: it must ask the first question, and it must also signal to the user that it is the first in a series of related questions.


CGT

File Help

What To Do

Bookmark

So, the probability of rain is

0.21 * RainYesterday + 0.0025 * CloudCover + 0.0023 * Humidity + 0.2
i.e.
0.21 * 1.0 + 0.0025 * 37.5 + 0.0023 * 30.0 + 0.2
i.e.
0.21 + 0.09375 + 0.069 + 0.2
giving
0.57275.

Should we walk, or should we read the paper? Explore the tree to see what's involved.

Click on the green arrows to explore.

Figure 28: Echoing the data values, screen image

9 Dynamic reconfiguration: ExistsIf

We have seen that Question nodes give tree authors the ability to obtain information from the user. The values are stored in variables: using the facilities of Evaluables we can test the values of variables, modify them, and compute new variables. In our simple example we computed the probability of rain. In our Benign Prostatic Hyperplasia tree (see [3]) a sequence of seven questions was used to compute the American Urological Association (AUA) symptom score. We now build on these facilities to provide a feature that allows the tree to dynamically reconfigure itself.

This is achieved by using a new (optional) node field, *ExistsIf*. In this field, the tree author writes an Evaluable expression that will evaluate to `true` or `false` using the methods discussed above in Section 6.

Suppose, for example, that there are three possible treatments for a condition. These could be represented by three Chance nodes, the children of a Decision node. Suppose further that Treatment 1 would not be offered to a patient who was a sufferer from diabetes, and that we have a variable `diabetic`. A Question node has already set this variable to be `true` or `false`. The "Treatment 1" node could be coded as

```
ID: T1
ShortLabel: Treatment 1
ParentID: D1
ExistsIf: not diabetic
Type: Chance
...
```

Figure 29 shows the structure of this example. The Question node "Diabetic?" asks the question about diabetes (or, perhaps, that question may have been asked earlier); the Decision node "Possible treatments" presents the choice between the the three treatments; and the nodes "Treatment 1" to "Treatment 3" are the roots of the various possible outcomes of the treatments, shown as "... etc" in the diagram. The node "Treatment 1" will not be visible, in either WaT or SaT modes, if `diabetic` is `true`.

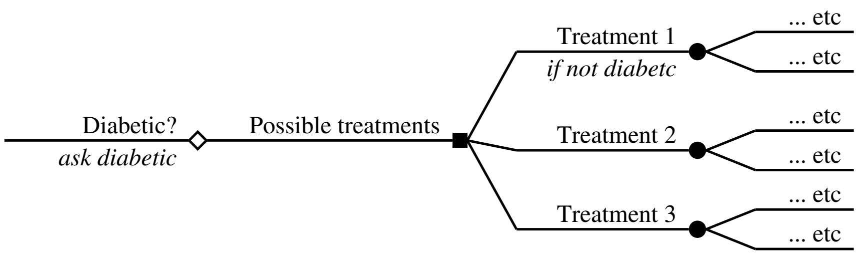


Figure 29: Using `ExistsIf` to exclude a branch

Of course, we could also have arranged matters so that diabetic and non-diabetic users see completely different versions of the tree, with only the root segment of the tree in common. Such an arrangement is illustrated in Figure 30. Both the diabetic and the non-diabetic see two possible treatments, but they are completely different. It should be clear that we can create patterns of arbitrary complexity, suggesting treatments to patients only if they meet complex conditions. For example, a treatment based only on diet could be restricted to patients whose symptoms are not too severe and who have already expressed willingness to modify their diet.

Any type of node can have the ExistsIf field. However, there is a structural constraint. Clearly, under all circumstances every Chance or Decision node must have at least one child surviving the ExistsIf tests. We can see at a glance that this is true for the trees of Figures 29 and 30. However, consider Figure 31. It is not clear that this tree always has a right-hand side: for a user who is neither tall nor thin, the node "Possible treatments" will have no children. But to see the possible problem we have to understand the meanings of the words "tall" and "thin." Only the human reader can do this: the CGT program

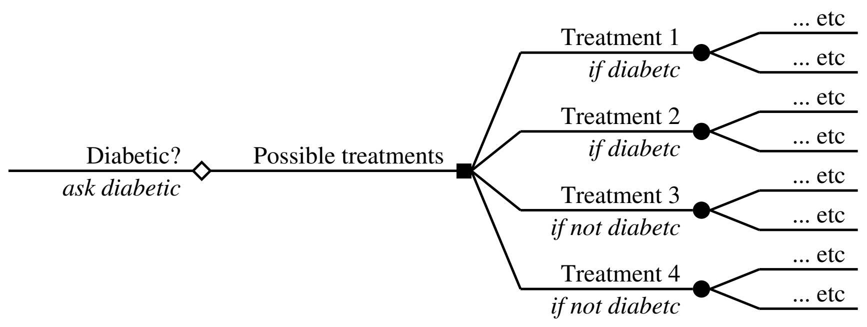


Figure 30: Using `ExistsIf` to provide alternative trees

can't. Also, to know if the problem is real, we have to know whether the users will in fact include people who are neither tall not thin. The program certainly doesn't know this. In general the program cannot determine when it reads the tree whether there is a potential problem. It is the tree author's responsibility to consider this issue carefully when designing the tree. The current version of the program does not treat a childless non-Terminal node as an error: in WaT mode it just displays it, but fails when it enters SaT mode.

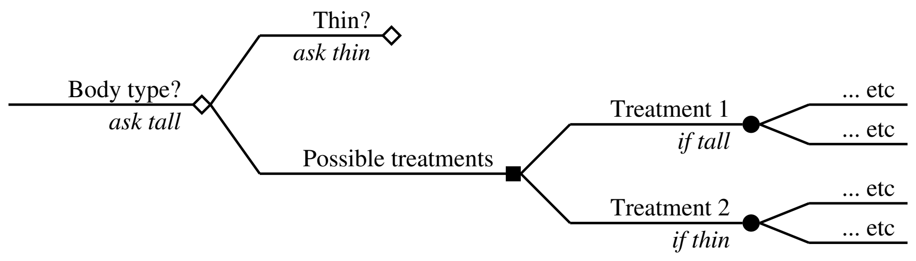


Figure 31: A structural problem with `ExistsIf`?

10 Composed nodes

In this section we describe our way of simplifying and customising the display of Terminal nodes. The text of this section borrows from the discussion in our earlier paper (see [1]) but goes into much greater detail about the actual coding for the CGT viewer program.

10.1 Basic ideas

The method is used where there are a number of independent events that can co-occur to make up an outcome. For example, let us return to our theme of the Sunday walk. Consider the fragments shown in Figure 32. We have three Chance nodes dealing with the possible outcomes of a decision to go for a walk. We see that it may rain, and it may be cold. So the walk may be "Nice", or the walker may get cold, or wet, or both. An equivalent tree can be constructed by flattening the tree, removing some of the Chance nodes, to give a more convenient representation.

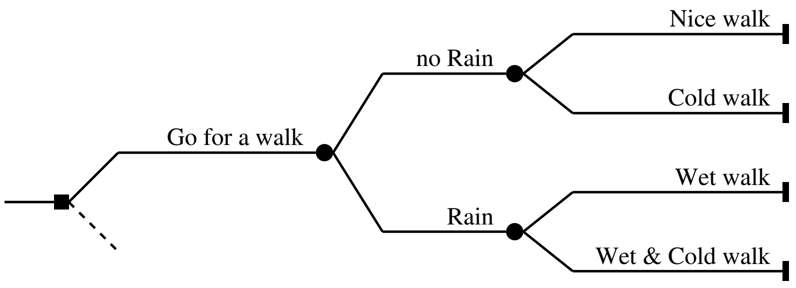


Figure 32: Outcomes of a Sunday walk

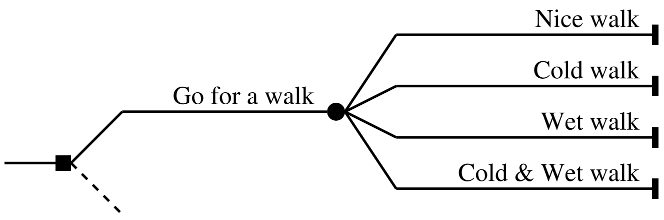


Figure 33: Outcomes of a Sunday walk, flattened

The general form is this: if we have n events that are not mutually exclusive, then there are 2^n possible combinations. Thus if the walker might have got tired as well, there would have been eight (2^3) possible outcomes, and thus eight Terminal nodes. This is shown in Figure 34. Obviously, the number of terminal nodes can become very large.

Clearly this combinatorial explosion can give problems to both the tree author and the user of the tree. Consider first the tree author, who has eight Terminal nodes to code. Even in the three-event case of Cold, Wet, and Tired, any one event, such as Cold, occurs four times. This means that any UserText (Section 5) relating to Cold will have to be repeated four times, and the same will apply to Wet and Tired. This makes the tree hard to write and very hard to maintain. It is all too easy for the various descriptions to become inconsistent.

Turning to the tree user, they are very likely to be daunted and confused by the large number of branches that they are offered. This confusion will be made worse by the fact that the nodes will contain repeated material. The user is likely to lose track of whether they have looked at some particular node before. A further problem for the user will be the number of utilities that they have to supply. As we saw in Section 3.3 , the user is asked to position each outcome on an analogue scale. This was shown in

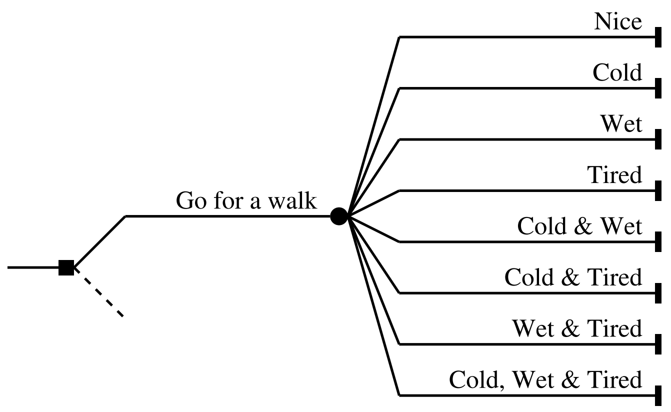


Figure 34: (Too many) Outcomes of a Sunday walk

Figure 12, where there were only three outcomes and the task was easy. But if there are a large number of outcomes then the task will become very tedious for the user. If those outcomes contain repeated words or phrases, as here, the user may be so discouraged as to be reluctant to complete the task.

There is also a more arcane problem for both author and user. The labels for apparently simple outcomes mean more than they say. For example, the outcome of Figure 34, "Cold," is not simply that the walker is cold. Instead, it is that the walker is cold (but not wet or tired). The difference is important. For example, the author must remember that if the probabilities of being cold, wet, and tired are respectively p_c, p_w, and p_t, the probability of the "Cold" node is not p_c (surprisingly). Instead it is

$$p_c - p_{c\wedge w} - p_{c\wedge t} + p_{c\wedge w\wedge t}$$

that is, the probability of being only cold (and not wet or tired). The user has a difficulty too. When he or she is looking at the "Cold" node, they have to remember that Coldness occurs at three other terminals as well.

Our method for addressing these various problems is the *composed node*. The tree author indicates that a particular outcome is "composed" of two or more other outcomes. This is done using the ShortLabel field of the node. Thus if we have two "composing" nodes with ShortLabels respectively `Cold` and `Wet`, then the author gives the composed node the ShortLabel `[Cold][Wet]`. We see that the composed node's ShortLabel is composed of the ShortLabels of of the composing nodes, each enclosed in square brackets. Although the author still has to write a node for every possible outcome, the material is easier to handle, as we shall see.

The tree author does not need to supply a node (whether composed or not) for every combination that is mathematically possible: only for those that actually occur. Further, the author can use a mixture of composition and writing a node normally: thus, in the example in Figure 34, the author can code some of the combinations as composed nodes and others as normal Terminal nodes. Note that composed nodes must come after the nodes that compose them.

As a simple example we shall follow the obvious strategy of simply composing the last four outcomes of Figure 34. This is shown in Figure 35. The ShortLabels of the last four nodes use the square-bracket convention, indicating that they are composed, and the lines are dashed to indicate that their visibility is different.

The key facts about composed nodes are these:

- In WaT mode, the user cannot see any composed node directly: it does not appear as a child or sibling of any current node, and it cannot become the current node. Hence, in WaT mode the user would not be able to see the bottom four nodes in Figure 35.

- When a composing node is the current node in WaT mode, the display of its UserText is supplemented by the UserTexts of composed nodes containing it, and also by the UserTexts of its

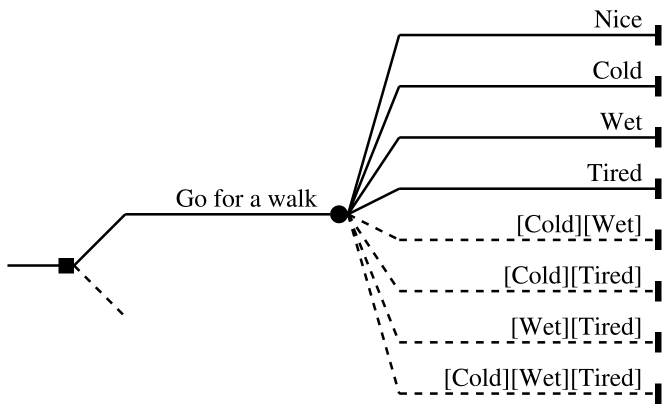


Figure 35: Outcomes of a Sunday walk using composition

co-composing node(s). Thus the basic WaT display of `Cold`, for example, would contain the User-Text for all of

`Cold` (the current node)

`Wet` (co-composing with the current node)

`[Cold][Wet]` (a composed node containing the current node)

`Tired` (co-composing with the current node)

`[Cold][Tired]` (a composed node containing the current node)

`[Cold][Wet][Tired]` (ditto)

These texts can be combined in a number of different ways. This is described later in this section.

- Composed nodes are not shown in the "Map" display that can be produced in WaT mode.
- The user does not see the composed nodes at the Utility Elicitation stage. Instead, the program computes payoffs for them, based on the payoffs of the composing nodes. The method is described in [1], pages 28-31. The author should supply default payoffs, however, in case the user skips utility elictation. Figure 36 shows utility elicitation for the five non-composed outcomes of the current tree: the program will compute the four composed outcomes.
- All nodes not removed by ExistsIf (Section 9), whether composed or not, continue to appear in SaT mode. Figure 37 shows a portion of the full tree: the user can see all eight branches of the "Walk" node.
- The tree author must supply probabilities for all Terminal nodes, composed or not.

10.2 Writing the tree

We now turn to coding the tree file for the example in Figure 35. Obviously there is a variety of ways in which this could be coded: we shall explore the possibilities as the discussion proceeds.

We begin by coding the tree fields. These are principally declarations of variables for the probabilities. The fields are shown in Figure 38. Once again, line numbers have been added for ease of discussion here.

Line 5 introduces a new Tree field, the NeutralPoint field. This field is required for trees that contain composed nodes. It gives a value above which all Payoffs in the tree are to be considered "Positive" outcomes, and below which they are to be considered "Negative" outcomes. We have already seen this idea in operation in Section 3.3. It is crucial for composed nodes, because the program must have

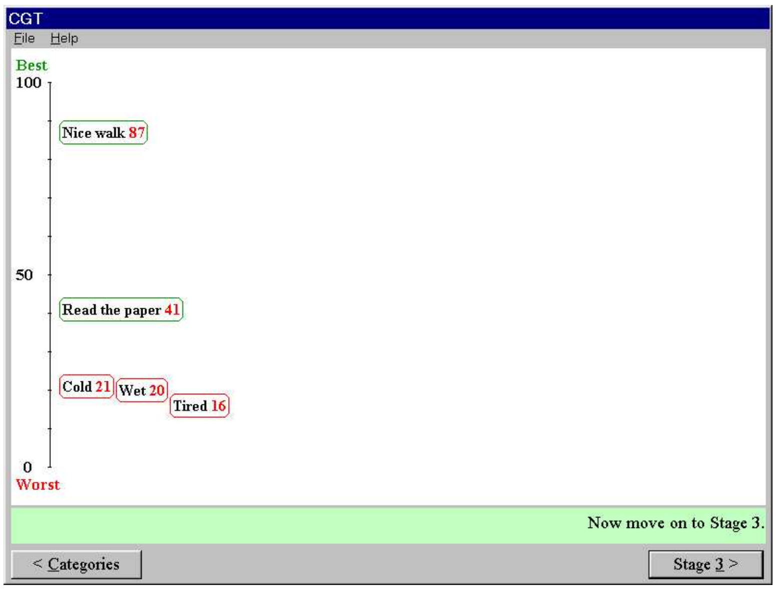


Figure 36: Scaling doesn't show composed nodes

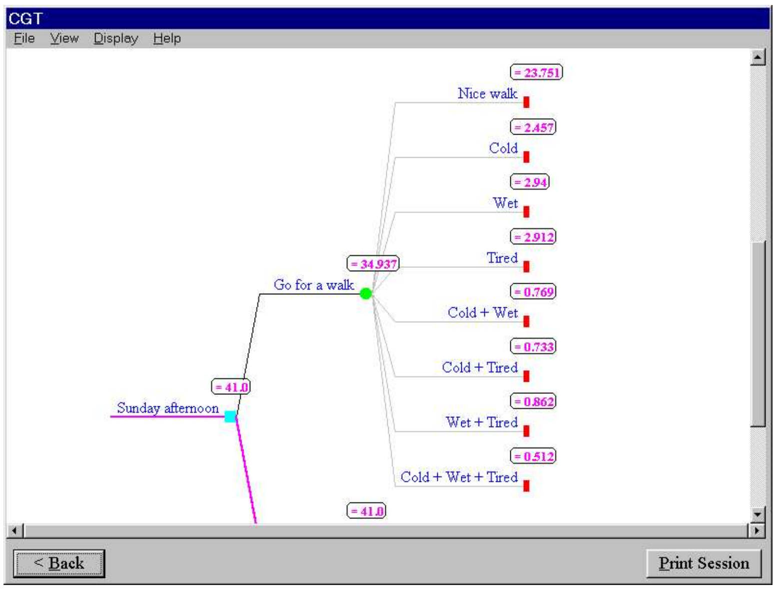


Figure 37: All nodes appear in SaT mode

```
 1. TreeName: Sunday walk (composed nodes)
 2. IntroText: This tree deals with the problem of deciding
 3.  what to do on Sunday afternoon.
 4. RequiresVersion: 2.6
 5. NeutralPoint: 35
 6. TreeVariable: pCold = 0.3
 7. TreeVariable: pWet = 0.35
 8. TreeVariable: pTired = 0.4
 9. TreeVariable: pCnW = pCold*pWet
10. TreeVariable: pCnT = pCold*pTired
11. TreeVariable: pWnT = pWet*pTired
12. TreeVariable: pCnWnT = pCold*pWet*pTired
13. TreeVariable: pCnode = pCold - pCnW -pCnT + pCnWnT
14. TreeVariable: pWnode = pWet - pCnW -pWnT + pCnWnT
15. TreeVariable: pTnode = pTired - pCnT -pWnT + pCnWnT
16. TreeVariable: pCWnode = pCnW - pCnWnT
17. TreeVariable: pCTnode = pCnT - pCnWnT
18. TreeVariable: pWTnode = pWnT - pCnWnT
19. TreeVariable: pCWTnode = pCnWnT
20. GlobalMacros: McDefine(TwoDP,‘McEval($1+0.005,2)’)
21. ComposedNodeJoiningText: <em>Combined with this we may have:</em>
```

Figure 38: Tree fields for a tree with composition

such a neutral point in order to know how to compute payoffs for composed nodes. For example, the composition of two negative outcomes must be more negative than either, and the composition of two positive outcomes must be more positive than either. In the present example, the neutral point is 35, so we infer that that author will give values less than 35 to Cold, Wet, etc, and values greater than 35 to "Read the paper" and "Nice Walk."

Our tree will have eight children of a Chance node, and so needs eight probabilities. Lines 6 to 19 declare a series of variables spelling out rather pedantically how these eight probabilites are computed, using a probability model in which the three events "Walker gets cold," "Walker gets wet," and "Walker gets tired" are assumed to be independent (however implausible this might be in real life). Values for these three probabilities are set in lines 6 to 8. The remainder of the lines use elementary probability theory. Lines 9 to 12 compute probabilities for joint events, so for example line 9 computes $p_{c \wedge w}$, the probabilty "Walker gets cold and wet". Finally lines 13 to 19 compute the probability for each node, so for example line 16 computes `pCWnode`, the probability "Walker gets cold and wet (but not tired)," which is the probability for the node `[Cold][Wet]`. These various calculations could be expressed more briefly, but our method has the merit of making the steps explicit.

Line 20 uses the GlobalMacros field to declare a macro (Section 7) called `TwoDP`, which prints its argument rounded (not truncated) to two decimal places. For example, `TwoDP(0.147)` will print as `0.15`. We shall make repeated use of this macro later in the tree: it will make it easier to write the various UserTexts.

Line 21 introduces another new Tree field, the ComposedNodesJoiningText field. This field is optional: it sets the text that is used to introduce the various UserTexts that are associated with the UserText of a composing node when that node is the current node in WaT mode. We shall see later where it is displayed on the screen. The default value for this field can be customised for an installation (in the file `data1_90\default.txt`): in the standard installation the default is a blank string. The tree author can also set this text for individual composed nodes, using the JoiningText node field, which contains DisplayText.

Now we look at the coding of the nodes. Figures 39 and 40 show the section of the tree file containing the eight nodes of interest: these are the nodes shown at the right-hand-side of Figure 35. The four composing nodes are shown in Figure 39, and the four composed nodes are shown in Figure 40. Once again line numbers have been added.

The four composing nodes, Figure 39, are straightforward. In each case the UserText contains

```
ID: 1.1
  ShortLabel: Nice walk
  ParentID: 1
  Type: Terminal
  Prob: #
  Payoff: 80
  UserText: We may have a pleasant walk.  The birds will
    sing and the flowers will bloom.  The probability of
    this is
    TwoDP(1 - pCnode  - pWnode  - pTnode
            - pCWnode  - pCTnode  - pWTnode
            - pCWTnode).

ID: 1.2
  ShortLabel: Cold
  ParentID: 1
  Type: Terminal
  Prob: pCnode
  Payoff: 30
  UserText: It may be rather chilly.  The probability of
    this is TwoDP(pCnode).

ID: 1.3
  ShortLabel: Wet
  ParentID: 1
  Type: Terminal
  Prob: pWnode
  Payoff: 30
  UserText: It may turn out wet. The sandwiches
    will be soggy and inedible.  The probability of
    this is TwoDP(pWnode).

ID: 1.4
  ShortLabel: Tired
  ParentID: 1
  Type: Terminal
  Prob: pTnode
  Payoff: 30
  UserText: We may get tired.  This will make us
    snappish and bad company.  The probability of
    this is TwoDP(pTnode).
```

Figure 39: Composing nodes

```
ID: 1.5
  ShortLabel: [Cold][Wet]
  ParentID: 1
  Type: Terminal
  Prob: pCWnode
  Payoff: 20
  UserText: We may get both cold and wet. The probability of
    this is TwoDP(pCWnode).

ID: 1.6
  ShortLabel: [Cold][Tired]
  ParentID: 1
  Type: Terminal
  Prob: pCTnode
  Payoff: 20
  UserText: We may get both cold and tired. The probability of
    this is TwoDP(pCTnode).

ID: 1.7
  ShortLabel: [Wet][Tired]
  ParentID: 1
  Type: Terminal
  Prob: pWTnode
  Payoff: 20
  UserText: We may get both wet and tired. The probability of
    this is TwoDP(pWTnode).

ID: 1.8
  ShortLabel: [Cold][Wet][Tired]
  ParentID: 1
  Type: Terminal
  Prob: pCWTnode
  Payoff: 15
  UserText: We may get all of cold, wet, and tired.
    The probability of this is TwoDP(pCWTnode).
```

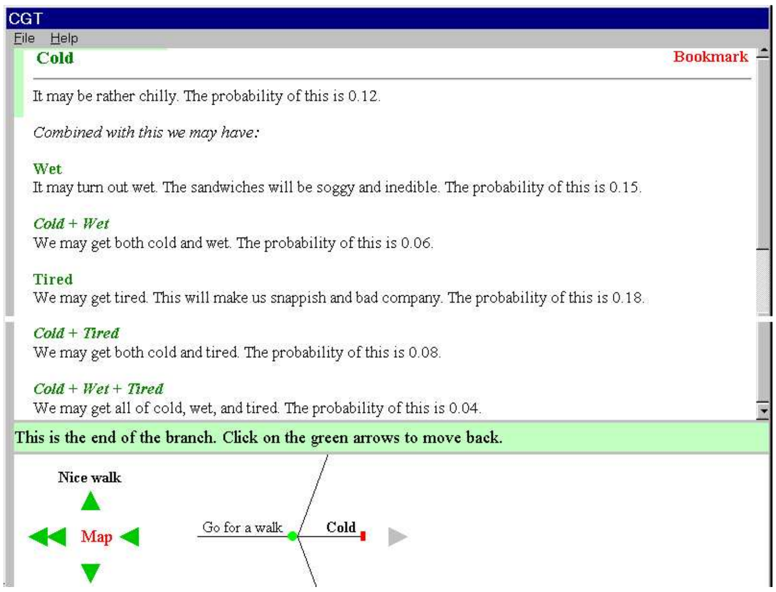
Figure 40: Composed nodes

material specific to one particular outcome: thus the Cold node (lines 54 to 61) deals only with being cold (and not wet or tired). The four composed nodes, too, are very simple: the only new feature is the way their ShortLabels declare them to be composed by using the square-bracket notation. Once again each node's UserText gives information about that outcome alone.

When these nodes are viewed in WaT mode, remember that only half of them are visible. The composed nodes have disappeared as separate nodes. When the user looks at an ordinary node, such as Nice Walk (lines 41 to 52 of Figure 39), nothing remarkable happens. The ordinary process of macro expansion (Section 7) gives us UserText saying

> We may have a pleasant walk. The birds will sing and the flowers will bloom. The probability of this is 0.27.

However, when the user looks at a composing node such as Cold, the program displays UserText from that node and from five other nodes: the two nodes that are co-composing with it, and the three nodes that it composes. The default display of these texts is shown in Figure 41. (This Figure is a composite of two screen-shots: the combined UserText is too big to fit in the upper pane, and the vertical scroll bar must be used.) The ComposedNodesJoiningText, "Combined with this we may have:" can be seen under the first UserText. The other UserTexts are each introduced by their ShortLabels: those for the composed nodes are in italic and contain a plus sign, so that [Cold][Wet] is rendered as "Cold+Wet."



Cold

It may be rather chilly. The probability of this is 0.12.

Combined with this we may have:

Wet
It may turn out wet. The sandwiches will be soggy and inedible. The probability of this is 0.15.

Cold + Wet
We may get both cold and wet. The probability of this is 0.06.

Tired
We may get tired. This will make us snappish and bad company. The probability of this is 0.18.

Cold + Tired
We may get both cold and tired. The probability of this is 0.08.

Cold + Wet + Tired
We may get all of cold, wet, and tired. The probability of this is 0.04.

This is the end of the branch. Click on the green arrows to move back.

Figure 41: Composed UserTexts: default display (combined screen-shots)

This default display is comprehensive, but the reader may feel that it does not flow very well. However, there are a number of ways in which the display can be modified.

The simplest modification to the format of the merged UserTexts is to let them flow together. This can be done using the ComposedHeadings tree field. If the tree fields contain the line

```
ComposedHeadings: No
```

then the headings (and their associated HTML paragraph tags) are removed from the merged text. The ComposedNodesJoiningText is also removed. In our present example, the text at the Cold node would consist of the six UserTexts we have already seen, in the same order, but as one continuous paragraph:

> It may be rather chilly. The probability of this is 0.12. It may turn out wet. The sandwiches will be soggy and inedible. The probability of this is 0.15. We may get both cold and wet. The probability of this is 0.06. We may get tired. This will make us snappish and bad company. The probability of this is 0.18. We may get both cold and tired. The probability of this is 0.08. We may get all of cold, wet, and tired. The probability of this is 0.04.

We can correct the rather breathless style by adding HTML markup. For example, we can simply add paragraph breaks by placing the `<p>` tag at the start of each node's UserText field. This greatly improves the readability of the text:

> It may be rather chilly. The probability of this is 0.12.
>
> It may turn out wet. The sandwiches will be soggy and inedible. The probability of this is 0.15.
>
> We may get both cold and wet. The probability of this is 0.06.
>
> We may get tired. This will make us snappish and bad company. The probability of this is 0.18.
>
> We may get both cold and tired. The probability of this is 0.08.
>
> We may get all of cold, wet, and tired. The probability of this is 0.04.

Another possibility is to write a macro that will produce the first UserText (the composing node's own UserText) on its own, followed by a bulleted list of the remaining UserTexts. Such a macro, McList, is shown in Figure 42, and sample invocations are shown in Figure 43. Once again line numbers have been added. The display of an invoking node is shown in Figure 44.

```
22.   McDefine(McFirst,YES)
23.   McDefine(McList,‘McIfElse(McFirst,YES,
24.     ‘$1
25.     <p>A number of other things could be associated with
26.     this outcome: these are
27.     <ul>
28.     McDefine(‘McFirst’,NO)’,
29.     <li>$1)’)
```

Figure 42: McList: Macro to produce a bulleted list in combined UserTexts

The macro definition is the most complex part of this. The McList macro's effect must differ on its first invocation from its second and subsequent invocations within a node. To achieve this we define another macro, McFirst, to use as a variable. Initially it has the value `YES`. McFirst tests this macro, using McIfElse. If it is set to `YES`, then McFirst outputs its argument, then a connecting text, then the HTML to start a bulleted list, and finally it sets McFirst to `NO`. On subsequent calls within a node, the macro just outputs the HTML for a bullet, followed by its argument. Clearly this is a technique that can be adapted to produce a variety of effects.

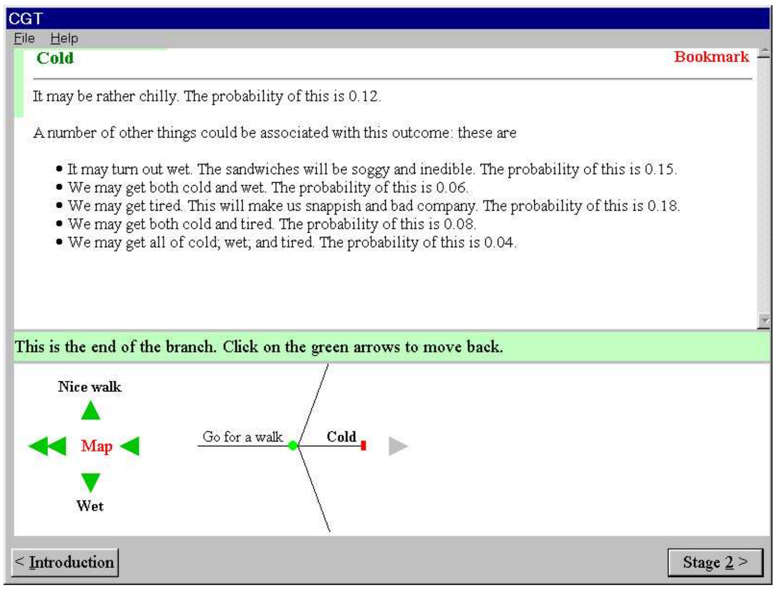
Another possibility is to use the JoiningText. This node field can only be used at composed nodes. For example, if we have composing nodes A and B, this field could be used at the composed node [A][B]. It can contain DisplayText. Its effect would be seen when viewing one of the composing nodes (A or B) in WaT mode: the ShortLabel of the other composing node(s) would be preceded by the text in the field.

We saw in Section 5.4 that the UserText field of any node can be split into multiple texts, each with its own button label. This facility can be used with composed nodes. Using these buttons extends the rules that we have seen so far. The full rules are:

- When showing the UserText for the current node, the node's siblings are checked for composed nodes which are composed from the current node. (Remember, the current node may be a composing node, but it cannot be a composed node, because composed nodes cannot become current.)
- If such a composed node or nodes are found, UserTexts from both the composed node and the other composing node(s) are appended to the current node's UserText as follows:
    - If the UserText button label(s) at a composed node match those of its composing nodes, for each matching button the current node's button text is followed by the ComposedNodesJoiningText, if any. (Joining texts may be suppressed by setting the tree field ComposedHeadings to `No`).

```
ID: 1.3
  ShortLabel: Wet
  ParentID: 1
  Type: Terminal
  Prob: pWnode
  Payoff: 30
  UserText: McList(‘It may turn out wet. The sandwiches
    will be soggy and inedible.  The probability of
    this is TwoDP(pWnode).’)

ID: 1.4
  ShortLabel: Tired
  ParentID: 1
  Type: Terminal
  Prob: pTnode
  Payoff: 30
  UserText: McList(‘We may get tired.  This will make us
    snappish and bad company.  The probability of
    this is TwoDP(pTnode).’)

ID: 1.5
  ShortLabel: [Cold][Wet]
  ParentID: 1
  Type: Terminal
  Prob: pCWnode
  Payoff: 20
  UserText: McList(‘We may get both cold and wet. The probability of
    this is TwoDP(pCWnode).’)

ID: 1.6
  ShortLabel: [Cold][Tired]
  ParentID: 1
  Type: Terminal
  Prob: pCTnode
  Payoff: 20
  UserText: McList(‘We may get both cold and tired. The probability of
    this is TwoDP(pCTnode).’)
```

Figure 43: Nodes that invoke McList


CGT

File Help

Cold

Bookmark

It may be rather chilly. The probability of this is 0.12.

A number of other things could be associated with this outcome: these are

- It may turn out wet. The sandwiches will be soggy and inedible. The probability of this is 0.15.
- We may get both cold and wet. The probability of this is 0.06.
- We may get tired. This will make us snappish and bad company. The probability of this is 0.18.
- We may get both cold and tired. The probability of this is 0.08.
- We may get all of cold; wet; and tired. The probability of this is 0.04.

This is the end of the branch. Click on the green arrows to move back.

Figure 44: Viewing composed UserTexts with McList

Then follows the matching button text for the other composing node(s), and for the composed node(s). Unless ComposedHeadings is `No`, each will be prefaced by their ShortLabels: for composing nodes the JoiningText (if any) of the composed node will precede the ShortLabel. If a button text is empty, the headings that would have preceded that text are omitted from the display.

– If button label(s) at a composed node do not match those of the composing node that is current, those label(s) are added as separate button(s) to the display of the current node. The texts from other composing nodes are not appended.

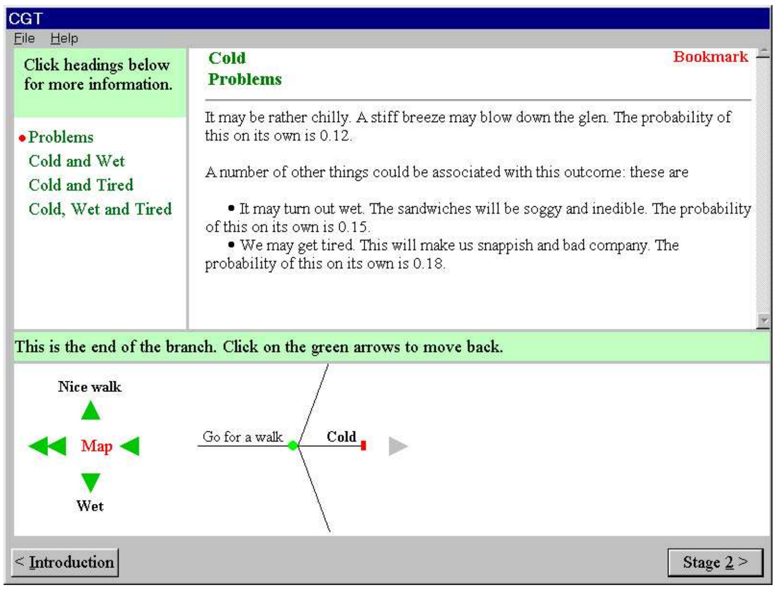
We now extend the handling of the current example, to use the new facilites. Each of the UserText fields of the seven unpleasant outcomes has been rewritten. The composing nodes are shown in Figure 45, the composed nodes in Figure 46, both with added line numbers. The display of an invoking node is shown in Figure 47. We see that the UserTexts of the composing nodes have been collected together on a single screen because they and their composed nodes share the same button label, “Problems” (lines 69, 79 and 89 of Figure 45). This will happen whichever of the three composing nodes is being viewed: the UserText from the other two will be appended to its own UserText. This will make up the text in the main pane of the display. However, there are no UserTexts from the composed nodes in the main pane. This is because the composed nodes have null text associated with the “Problems” button label (lines 99, 109, 119 and 129 of Figure 46). Instead, the display will also have buttons for the texts of the composed nodes, each of which has its own unique button label (lines 100, 110, 120 and 130 of Figure 46). These buttons can be seen at the left-hand side of the upper pane in Figure 47. Clicking one of these buttons brings up the associated UserText. This is shown in Figure 46. Because the composed nodes' UserTexts never appear together, the McList macro is not needed and the texts are left plain.

```
ID: 1.2
  ShortLabel: Cold
  ParentID: 1
  Type: Terminal
  Prob: pCnode
  Payoff: 30
  UserText: {Problems|McList(‘It may be rather chilly.  A stiff breeze
    may blow down the glen.  The probability of
    this on its own is TwoDP(pCnode).’)}

ID: 1.3
  ShortLabel: Wet
  ParentID: 1
  Type: Terminal
  Prob: pWnode
  Payoff: 30
  UserText: {Problems|McList(‘It may turn out wet. The sandwiches
    will be soggy and inedible.  The probability of
    this on its own is TwoDP(pWnode).’)}

ID: 1.4
  ShortLabel: Tired
  ParentID: 1
  Type: Terminal
  Prob: pTnode
  Payoff: 30
  UserText: {Problems|McList(‘We may get tired.  This will make us
    snappish and bad company.  The probability of
    this on its own is TwoDP(pTnode).’)}
```

Figure 45: Composing nodes using button labels

```
ID: 1.5
  ShortLabel: [Cold][Wet]
  ParentID: 1
  Type: Terminal
  Prob: pCWnode
  Payoff: 20
  UserText: {Problems|}
    {Cold and Wet|We may get both cold and wet. The
     probability of this is TwoDP(pCWnode).}

 ID: 1.6
   ShortLabel: [Cold][Tired]
   ParentID: 1
   Type: Terminal
   Prob: pCTnode
   Payoff: 20
   UserText: {Problems|}
     {Cold and Tired|We may get both cold and tired. The probability of
     this is TwoDP(pCTnode).}

 ID: 1.7
   ShortLabel: [Wet][Tired]
   ParentID: 1
   Type: Terminal
   Prob: pWTnode
   Payoff: 20
   UserText: {Problems|}
     {Wet and Tired|We may get both wet and tired. The probability of
     this is TwoDP(pWTnode).}

 ID: 1.8
   ShortLabel: [Cold][Wet][Tired]
   ParentID: 1
   Type: Terminal
   Prob: pCWTnode
   Payoff: 15
   UserText: {Problems|}
     {Cold, Wet and Tired|We may get all of cold; wet; and tired.
     The probability of this is TwoDP(pCWTnode).}
```

Figure 46: Composed nodes using button labels


CGT

File Help

Click headings below for more information.

● Problems
Cold and Wet
Cold and Tired
Cold, Wet and Tired

Cold
Problems

Bookmark

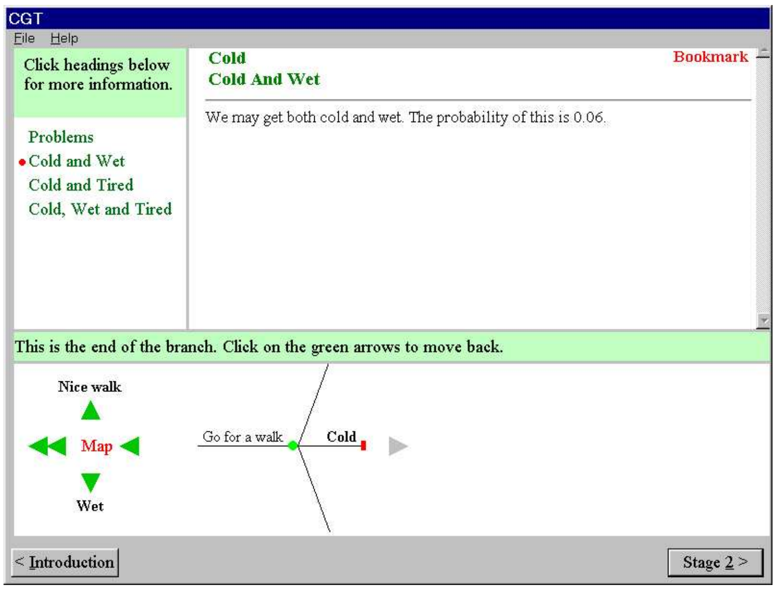
It may be rather chilly. A stiff breeze may blow down the glen. The probability of this on its own is 0.12.

A number of other things could be associated with this outcome: these are

- It may turn out wet. The sandwiches will be soggy and inedible. The probability of this on its own is 0.15.
- We may get tired. This will make us snappish and bad company. The probability of this on its own is 0.18.

This is the end of the branch. Click on the green arrows to move back.

Figure 47: Viewing composed UserTexts with button labels


CGT

File Help

Click headings below for more information.

Problems
● Cold and Wet
Cold and Tired
Cold, Wet and Tired

Cold
Cold And Wet

Bookmark

We may get both cold and wet. The probability of this is 0.06.

This is the end of the branch. Click on the green arrows to move back.

Figure 48: Viewing button text

10.3 Composition using hidden nodes

Thus far in our discussion all our composing nodes have been visible to the user. However, it is sometimes convenient to use some outcomes which never occur on their own. Such nodes are "hidden" because the user never sees them (not even in SaT mode). These outcomes act as building blocks for the construction of the actual outcomes, using composition.

We take a simple example, using the same domain as our previous example. Our prospective walk is up Dumyat ("Dum-eye-at"), the local hill. It may rain, or it may not. If it doesn't rain, we'll have a good view, but there's a (slight) risk of sunburn. A straightforward representation of this part of the tree is given in Figure 49. In Figure 50 we have simplified the structure by flattening it, adding an extra (hidden) node, and re-casting one of the original nodes as a composed node. The resulting tree file is shown in Figures 51 (the tree fields) and 52 (the nodes corresponding to Figure 50). As usual, line numbers have been added.

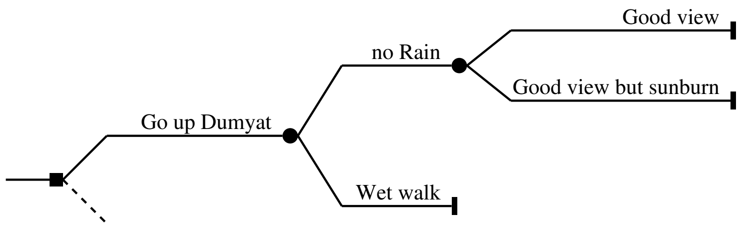


Figure 49: Up Dumyat, original version

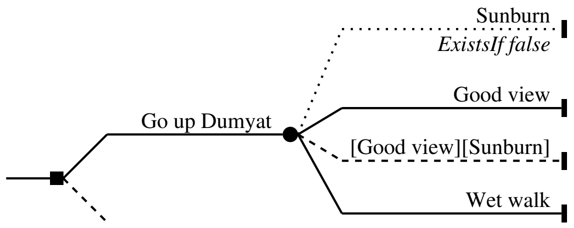


Figure 50: Up Dumyat, flattened and with hidden node

```
 1. TreeName: Dumyat
 2. IntroText: This tree deals with the problem of deciding
 3.  what to do on Sunday afternoon.
 4. RequiresVersion: 2.6
 5. NeutralPoint: 35
 6. TreeVariable: pGoodView = 0.6
 7. TreeVariable: pGoodnSun = 0.1
 8. TreeVariable: pRain = 0.3
 9. ComposedNodeJoiningText:
10. ComposedHeadings: Yes
```

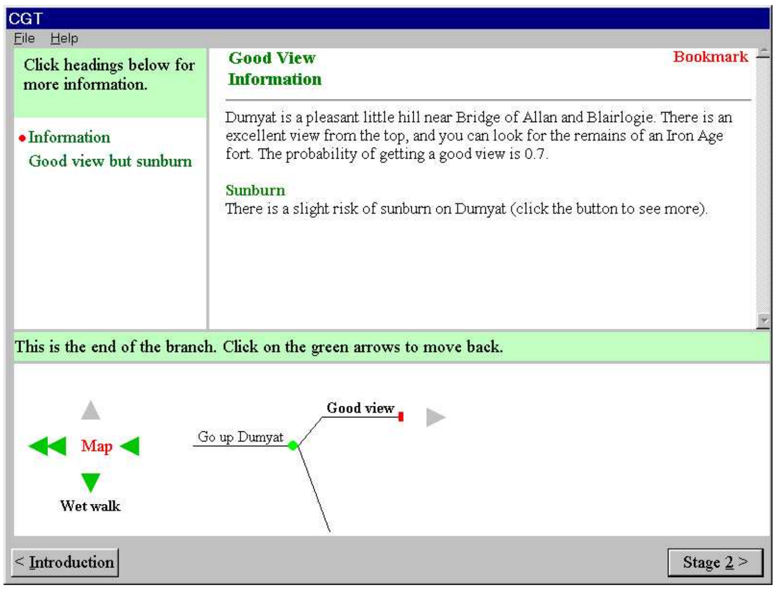
Figure 51: Up Dumyat: tree fields

```
ID: 1.1
  */ Hidden node
  ShortLabel: Sunburn
  ParentID: 1
  Type: Terminal
  ExistsIf: false
  ScaleIf: true
  Prob: 0
  Payoff: 30
  UserText: {Information|There is a slight risk of sunburn
    on Dumyat (click the button to see more).}

ID: 1.2
  ShortLabel: Good view
  ParentID: 1
  Type: Terminal
  Prob: pGoodView
  Payoff: 70
  UserText: {Information|Dumyat is a pleasant little hill
    near Bridge of Allan and Blairlogie.  There is an
    excellent view from the top, and you can look for
    the remains of an Iron Age fort.  The probability of
    getting a good view is McEval(pGoodView+PGoodnSun,1).}

ID: 1.3
  ShortLabel: [Good View][Sunburn]
  ParentID: 1
  Type: Terminal
  Prob: pGoodnSun
  Payoff: 60
  UserText: {Information|}
    {Good view but sunburn|If you do go up Dumyat, take
    your suncream (or a hat) if it looks as though the sun
    might come out.  Without protection, the risk of sunburn
    is McEval(pGoodnSun).}

ID: 1.4
  ShortLabel: Wet walk
  ParentID: 1
  Type: Terminal
  Prob: pRain
  Payoff: 20
  UserText: {Information|Unfortunately, it's quite likely
    to rain on Dumyat.  The probability is McEval(pRain).}
```

Figure 52: Up Dumyat: Composition with a hidden node

Here, the new material is the coding of the hidden node, shown on lines 32 to 41 of Figure 52. The node is hidden because it has an ExistsIf field (Section 9) set to `false` (line 36). This means that it always hidden: we could make it visible under some conditions by specifying those conditions, as an Evaluable expression, in the field. Normally, of course, such a node disappears completely: in particular, it would not be visible when the user is invited to scale the possible outcomes at the Utility Elicitation stage. But in this case we certainly do want the user to see it when scaling: we need it to be scaled because it is a composing node. We force scaling using the ScaleIf field (line 37). This is the first time we have seen this field. Like ExistsIf, it should contain an Evaluable. The node will be presented for scaling if the Evaluable evaluates to `true`, obviously. Here we always want it to be scaled. The node also contains Prob and Payoff fields: these are included for completeness but have no function in this example.

When the tree is viewed in WaT mode, the user will see only two nodes, "Good View" and "Wet walk." However, the UserText visible at "Good view" will collect together the UserTexts of three nodes, forming a natural grouping of all the information on sunny outcomes of the walk. The UserTexts of these three nodes use Button labels to control the display: the method is exactly the same as in the previous example (see Figures 45 to 48). The display is shown in Figure 53. We can see that we have information on the good view, noting the risk of sunburn. Further information about the sunburn risk is available, by clicking the Button label "Good view but sunburn".



Click headings below for more information.

• Information
Good view but sunburn

Good View
Information

Dumyat is a pleasant little hill near Bridge of Allan and Blairlogie. There is an excellent view from the top, and you can look for the remains of an Iron Age fort. The probability of getting a good view is 0.7.

Sunburn
There is a slight risk of sunburn on Dumyat (click the button to see more).

This is the end of the branch. Click on the green arrows to move back.

Figure 53: Viewing a node composed by a hidden node

When the user advances to Utility Elicitation, they see four items to scale: there is "Read the paper," which comes from a different branch, and "Wet walk," "Sunburn," and "Good view." These are straightforward and natural items, easy to scale. Going on into SaT mode, the user will see everything except the hidden node. This is shown in Figure 54: comparing this with Figure 50, we see that the intended structure has been achieved.

We can now give a more formal discussion of a hidden node.

- A Terminal node is "hidden" when it has an ExistsIf field, and the Evaluable expression in the field evaluates to `false`. If the Evaluable involves variables, then the node may be hidden only under particular circumstances, usually the particular characteristics of the user. A hidden node represents an outcome that does not occur on its own when the ExistIf is `true`. It may occur in conjunction with some other outcome: in this case the combination will be represented by another node (a composed node).

- If a node that can be hidden is a composing node, then it should have a ScaleIf field. The Evaluable in the field must evaluate to `true` when the node is hidden *and* any of the nodes it composes are not hidden.

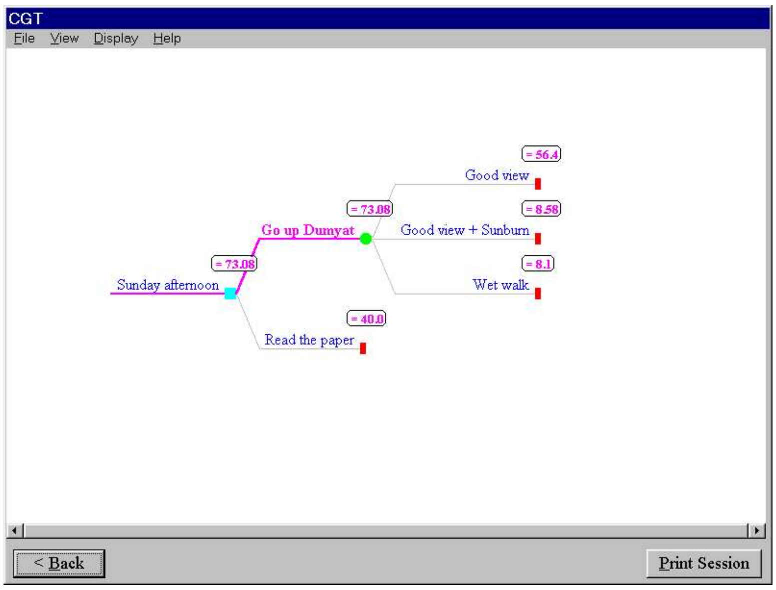


Figure 54: Hidden node invisible in SaT mode

- The Prob field of a node that can be hidden must evaluate to zero when the node is hidden.
- The Payoff field of a node that can be hidden is ignored when the node is hidden, but should be specified nevertheless.

In our example, we gave the user a slightly simpler tree by using the hidden node. In many cases, however, the gains can be substantial. These are cases where the possible combinations are very numerous, like our Cold, Wet, Tired example above. Consider a case in which a particular medical treatment for a condition has a number of possible side-effects that may occur together: for example, Drowsiness, Headache, ..., Drowsiness and Headache, etc. The number of possibilities rises sharply. Consider the case where all the possibilities are dichotomies, i.e. the treatment either does or does not relieve the condition, and the side-effects also either do or do not occur. Assume also that all combinations occur in practice. The situation when there are three side-effects is shown in Figure 55. As we would expect, with four dichotomies there are 16 outcomes, each represented by a visible node that the user will have to scale, and which they will have to keep distinct in their heads. Obviously, this is impractical. However, using hidden nodes and composition we arrive at Figure 56. In this case the user has only two visible nodes, and there are only five items to scale. Between them, the two visible nodes can display the information from all the nodes, in a structured and clear way.

The gains for the user should be obvious. However, the tree author still has quite a lot to do. There are now 19 nodes to code instead of the original 16, but at least the structure of the individual nodes is reasonably simple. This example obviously generalises to cases with more than three side-effects. The situation where the side effects are not independent raises no problems: if a particular combination does not occur, simply omit the corresponding composed node.

The tree author may have some difficulty in assigning probabilities to the various combinations. The clinical (or other) literature may not give enough guidance for empirical probabilities to be assigned to every branch. In such cases the author will have to seek statistical advice about the effect of any likely correlations or interactions between the different effects for which there are published data, and the best way of imputing values to the combinations. However, these difficulties exist in writing any decision tree. They are the same whether or not the author uses composition.

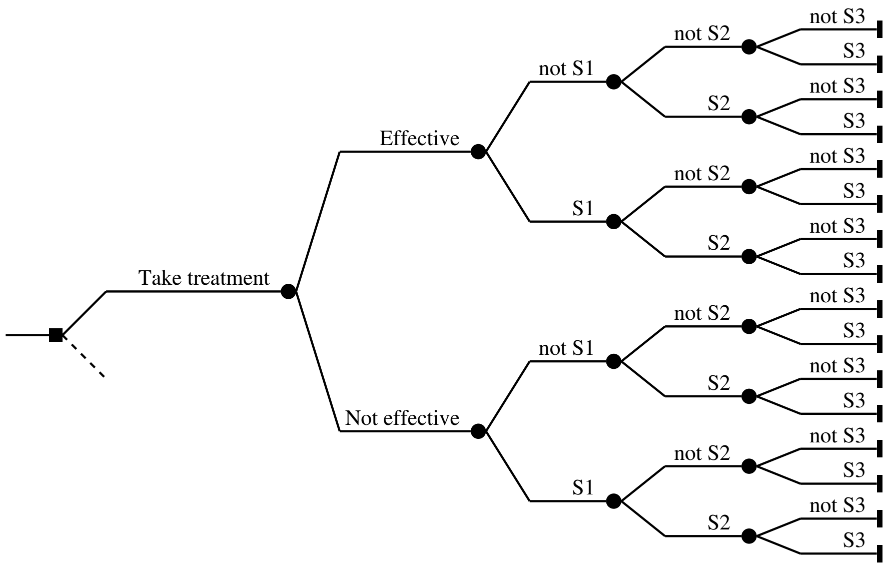


Figure 55: Treatment and three side-effects, original version

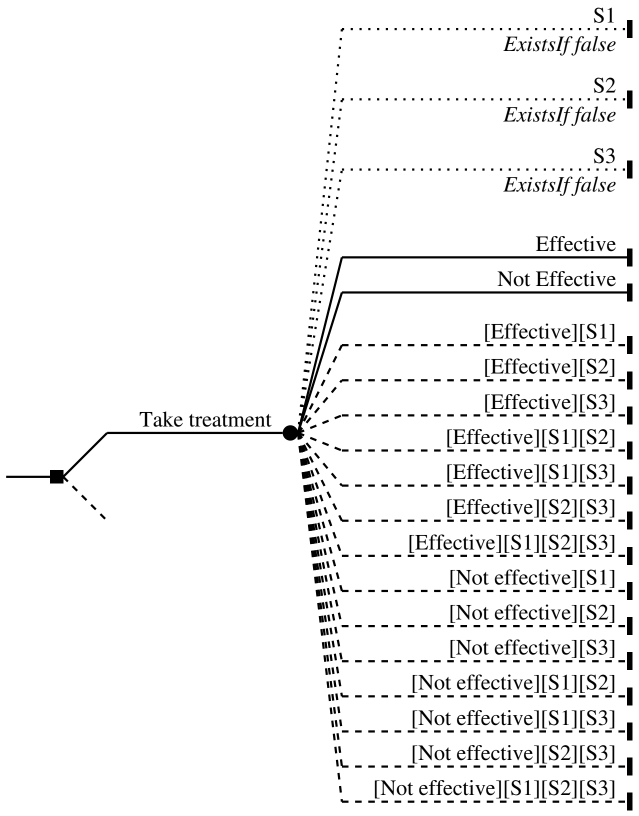


Figure 56: Treatment and three side-effects, flattened and with hidden nodes

11 Fields reference

Field	**Status**	**Contents**	**Usage**
ComposedNodesJoiningText:	Optional	Plain text	Text to connect current node's UserText to other nodes' UserTexts in WaT mode
DictionaryFile:	Optional	Filespec	File containing definitions of BCT terms
GlobalMacros:	Optional	Macro definitions	Define global macros using McDefine. May have multiple definitions
IntroText:	Required	DisplayText	Text for introductory screen
NeutralPoint:	Optional	Number	A number between 0 and 100, above which all Payoffs are "positive outcomes" and below which they are "negative outcomes." Required if the tree contains composed nodes.
RequiresVersion	Required	Decimal number	Program version for which tree was written: for all trees complying with this manual, may be any number between 1.96 and 2.6, but should not be higher than the version installed on your machine
TreeName:	Required	Plain text	Short descriptive phrase
TreeVariable:	Optional	Name [= expression]	Declare name of a variable, optionally supply initial value, which may be an expression
See also the Index entries for particular fields			

Table 3: Tree fields

Field	**Node type**			
	Decision	**Chance**	**Terminal**	**Question**
BoundVariable	N	N	N	Y
DocumentingText	O	O	O	O
ExistsIf	O	O	O	O
ID	Y	Y	Y	Y
InputType	N	N	N	Y
InstructionText	O	O	O	O
JoiningText	O for composed nodes only			
LongLabel	O	O	O	O
ParentID	Y	Y	Y	Y
Payoff	N	N	O	N
Prob	Y for children of chance nodes only			
QuestionText	N	N	N	Y
ScaleIf	N	N	O	N
ShortLabel	Y	Y	Y	Y
Type	Y	Y	Y	Y
UserText	Y	Y	Y	Y
ValidationRule	N	N	N	O
ValidationRuleText	N	N	N	O
Key: Y= Yes (Required) N = No (Not allowed) O= Optional See also Table 5 and Index entries for particular fields				

Table 4: Node fields for particular node types

Field	**Contents**	**Purpose**
BoundVariable	Variable name	Answer to question is stored in this variable
DocumentingText	DisplayText	Technical documentation for the node
ExistsIf	Evaluable	If evaluates to `false`, node and its children are not part of the tree (but may be composing Terminal)
ID	Plain text	Identifier for node, used to link tree. Should not contain spaces
InputType	Question type	One of `Edit(n)`, `Combo`, `Radio`, `HeightScale`, `WeightScale`, or`BloodPressure`
InstructionText	Plain text	Text between upper and lower panes in WaT mode
JoiningText	Plain text	Text introducing button text of composing node(s) when a co-composing node is the current node in WaT mode
LongLabel	Plain text	Heading for WaT mode display. No fixed limit but more than 60 characters would look wrong
ParentID	Plain text	Identifier of parent node, used to link tree
Payoff	Evaluable	The payoff of a Terminal node. Must evaluate to a number between 0 and 100 exclusive
Prob	Evaluable or '#'	The probability of a child of a Chance node. Must evaluate to a number between 0 and 1 inclusive. The probabilities of the children of a particular node must sum to 1
QuestionText	DisplayText	Text in upper pane when Question node is viewed in WaT mode
ScaleIf	Evaluable	If evaluates to `true`, then the node appears at Stage 2 unless the branch of the tree has been "pruned" because of an ExistsIf at a parent node. If the ScaleIf field is not present then the node will only be presented for scaling if it is visible (i.e. its ExistsIf, and those of all its parents, is `true`)
ShortLabel	Plain text	Up to 15 characters describing the node, or (for composed Terminal nodes) the ShortLabels of other Terminals each enclosed in square brackets
Type	Node type	One of `Decision`, `Chance`, `Question`, or `Terminal`
UserText	DisplayText	The text used in WaT mode when displaying the node, or when node is linked to the current node by composition. May be split into multiple button labels and button texts
ValidationRule	Evaluable	If evaluates to `false`, user is asked to re-enter answer to question
ValidationRuleText	Plain text	Text used if ValidationRule evaluates to `false`
See also Table 4 and Index entries for particular fields		

Table 5: Node fields, contents and meaning

References

[1] Richard Bland, Claire E Beechey, and Dawn Dowding. Extending the model of the decision tree. Technical Report CSM-162, Department of Computing Science and Mathematics, University of Stirling, August 2002. ISSN 1460-9673.

[2] G.B. Chapman and F.A. Sonnenberg. *Decision Making in Health Care: Theory, Psychology and Applications.* Cambridge University Press, Cambridge, 2000.

[3] Dawn Dowding, Vivien Swanson, Richard Bland, Pat Thomson, Chris Mair, Audrey Morrison, Anne Taylor, Claire Beechey, Richard Simpson, and Kate Niven. The development and preliminary evaluation of a decision aid based on decision analysis for two treatment conditions: Benign prostatic hyperplasia and hypertension. *Patient Education and Counseling*, ?:?, accepted.

[4] Dawn Dowding and Carl Thompson. Decision analysis. In Carl Thompson and Dawn Dowding, editors, *Clinical Decision Making and Judgement in Nursing*, pages 131–146. Churchill Livingstone, Edinburgh, 2002.

[5] J Dowie. Decision analysis in guideline development and clinical practice: the 'Clinical Guidance Tree'. In H.K. Selbmann, editor, *Guidelines in Health Care*, pages 162–193. Nomos-Verlagsgesellschaft, Baden-Baden, 1998.

[6] Ian S. Graham. *HTML 4.0 Sourcebook: a complete guide to HTML 4.0 and HTML extensions.* John Wiley and Sons, New York, 1998.

[7] Brian W Kernighan and P J Plauger. *Software Tools.* Addison-Wesley, Reading, Mass, 1976.

[8] Brian W Kernighan and Dennis M Ritchie. The m4 macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1977.

[9] Huw Llewelyn and Anthony Hopkins, editors. *Analysing How We Reach Clinical Decisions.* Royal College of Physicians, London, 1993.

Index