A User and Designer Perspective on Multimodal Interaction in 3D environments

Supplementary material

DOCTORAATSPROEFSCHRIFT

2006 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Context Acquisition and Aggregation Supports the Realisation of Proactive Interaction

A Comparison between Decision Trees and Markov Models

Bijgevoegde stelling voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen: Informatica, te verdedigen door:

Joan DE BOECK

Promotor: Prof. dr. Karin CONINX

Universiteit Maastricht

universiteit hasselt

DOCTORAATSPROEFSCHRIFT

2006 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Context Acquisition and Aggregation Supports the Realisation of Proactive Interaction

A Comparison between Decision Trees and Markov Models

Bijgevoegde stelling voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen: Informatica, te verdedigen door:

Joan DE BOECK

Promotor: Prof. dr. Karin CONINX

Universiteit Maastricht

universiteit hasselt

D/2006/2451/48

Abstract

Over the last years, personal portable computer systems such as PDAs or laptops are being used in different contexts: at the office, during a meeting or at home. Switching between those contexts often requires the user to manually change different system settings, such as the audio volume, screen resolution, etc. In this work, we propose a first step towards a proactive user interface, predicting the next probable system changes, based upon previously learned samples. In particular, we will focus on two algorithms, decision trees and Markov models, that may support this proactive interaction. By elaborating on some practical scenarios, we compare both implementations by evaluating their respective outcomes.

Contents

Chapter 1

Introduction

Consider an employee with a laptop. At the office the computer is used with a dual monitor and the audio level soft, in order not to disturb his colleagues. Mostly, he's working on the files found on the server, to which he is constantly connected. When a telephone call is received, he needs his agenda tool and todo-list. Alternatively, when in a meeting, the computer is either used for a presentation or just for taking notes. Finally, at the end of the day, he takes his portable home, where it is mainly used for entertainment, using the full graphical and audio capabilities.

From the example, it may be clear that portable computer devices, such as PDAs or laptops, are more and more being used in a variety of situations. As these devices are easy to carry they may be applied in different contexts, in with different system settings for each context are necessary.

With the current generation of user interfaces, this versatility in settings is supported. However all settings must be manually changed each time a context switch occurs. Manufacturers often provide their hardware with extra tools that allow switching those settings all at once, but still those tools have to be configured manually. Moreover, as the user's behaviour and preferences can change over time, configurations must be manually changed at regular points in time.

We present a comparison of two approaches to create proactive user interfaces. Proactive user interfaces monitor the current context and suggest the next possible steps based upon the user's behaviour learned from previous examples. In the scenario above, if the employee enters the meeting room, the interface may conclude from previously learned data that there is a good chance that one of the two possible settings will be applied. As a consequence it suggests

to switch to one of either. The same is also true when the user arrives at home or at the office, or when a telephone call at the office is detected.

We will not focus on capturing the context data itself, but rather on comparing two algorithms that may be suitable to support proactive interfaces. In the next chapter, we will discuss some related work, which may be of importance in our research. Next, in chapter 3, we define the scope of this work and provide a detailed overview. Chapter 4 describes the scenarios that are used to compare both algorithms. The results and behaviours of the algorithms are discussed in chapter 5. We end our contribution by formulating our conclusions and suggestions for future work.

Chapter 2

Related Work

Proactive interfaces already can be found in several prototype application, for instance as demonstrated by Lieberman et al. As consumer electronics are becoming more and more complicated and a lot of users are unable to understand their full potential, they created 'Roadie' [Lieberman 06]; a prototype interface that will try to infer the user's goal by monitoring his or her actions. To make predictions, it will use EventNet [Espinosa 05], a plan recogniser that uses knowledge mined from the 'OpenMind Commonsense knowledge Base' [Singh 02], a knowledge base of 770,000 English sentences describing everyday life. Using Roadie, users can select their goal from a list of suggestions. Next, planning and commonsense reasoning is used to explain how to reach the goal. The answer is displayed in plain text, explaining the user's next action(s). If the user makes a mistake, Roadie will launch a debugging dialog for extra assistance.

Alternatively, Dey et al [Dey 04] propose *a CAPpella*, which uses programming by demonstration to teach a context-aware application. This program captures context data, such as raw audio or video from the environment, which can be replayed afterwards to indicate the relevant parts. After several iterations of recording and indicating the relevant parts, the application should be able to recognise a certain context (such as a meeting), from the recorded data, and hence perform the relevant actions for that context (such as launching a notepad to make notes during the meeting). The value of this work is that it learns to recognise a context from natural data; however the user still has to indicate which are the relevant parts.

Buyn and Cheverst propose the utilisation of context history together with user modeling and machine learning techniques to create proactive applica-

tions. In [Byun 04] they describe an experiment to examine the feasibility of their approach for supporting proactive adaptations in the context of an intelligent office environment. The application uses two different approaches to obtain proactivity: *proactive rule based adaptation*, and *proactive modeling adaptation*. The first uses simple predefined rules, but its limitation is that users have to reconfigure the system as their preferences change. The second approach automatically adapts those predefined rules when observation makes clear that the user's preferences have been changed. In order to learn the patterns of the user's behaviour, *decision trees* are used. Decision trees have the advantage to be much easier to understand by designers than other machine learning techniques such as neural networks. Buyn's conclusion is that context history has a concrete role for supporting proactive adaptation in ubiquitous computing environments.

Cook et al. developed a smart home that adapts itself to its inhabitants. The role of the prediction algorithms within the architecture is discussed in [Cook 03]. They use three different algorithms, one of which is a Markov model. A separate neural network is trained to select the prediction algorithm that would probably make the best prediction with the available data. The Markov model is generated from collected action sequences and used to predict the next action, given the current context. This was tested on synthetic data generated for 30 days, using separate scenarios for weekdays and weekends. The algorithm generated a 74% predictive accuracy on that data.

Petzold et al investigate the feasibility of 'in-door next location prediction' using a sequence of previously visited locations [Petzold 05]. They compare the efficiency of several prediction methods such as Bayesian networks, neural networks and Markov models. Markov models score fairly well with an average prediction accuracy of about 80%.

From the related work, it appears that decision trees and Markov models may both fit in our proactive interface, as both solutions have proven to provide adequate output with a reasonable amount of learning samples. Therefore, we investigate which of both may be most suitable in our application.

Chapter 3

Proposed Approach

3.1 Scope

As mentioned in the introduction, in this research we focus on the computer interface of a portable device. When a contextual change occurs (such as a telephone call, entering a meeting, being at home,...), the interface calculates the user's possible goal and proactively suggests the next possible actions as a result of previously recorded and processed learning samples.

We compare two approaches which may support proactive interaction: one implementation is based on a decision tree algorithm; the second is based upon a Markov Model. The details of both approaches are described in the next sections (section 3.2 and section 3.3).

It may be clear that the applications for those interfaces and the amount of system settings to monitor are nearly unlimited. However, we have chosen to work with a limited but relevant set of system settings and locations to ensure a manageable degree of complexity. We have defined the following attributes and values:

- Five applications that can be opened or closed: *Outlook*, *Internet Explorer*, *MS Word*, *Photoshop* and *PowerPoint*.
- Three values for the screen's backlight: *High*, *Medium* and *Low*
- Four audio levels: *Loud*, *Medium*, *Soft* and *Off*
- Three display themes: *Normal*, *High Contrast* and *Presentation*

- Three screen resolutions: *High Resolution*, *Projector* and *Dual Monitor*

Besides this, we also foresee three possible locations where the interface may be used in a different manner: *Office*, *Meeting Room* and *Home*. The latter context attributes such as 'location' may be handled slightly different by the system. Unlike the other attributes, the system cannot autonomously change these values, as they are 'read only'. Hence, a suggestion such as 'go to the meeting room', must not occur.

As we are not concerned with capturing the data itself in order to avoid the technical problems which rise in this domain, we use a simple 'Wizard Of Oz' user interface[1] as is described in section 4.1. In this form-based interface, we manually input the user's actions according to three predefined scenarios. We will evaluate those scenarios with both implementations, as is described in sections 4.2 through 4.4.

3.2 Decision Tree Implementation

3.2.1 General Approach

In a first part of the implementation we use *decision trees* [Mitchell 97] to learn from the user's actions. A decision tree is an internal data structure consisting of a 'root node', 'nodes', 'connections' (branches) and 'leaf nodes' as shown in figure 3.1. A node represents an attribute of the environment (e.g. location, screen resolution, ...), and may have one or more branches. Each branch contains a possible value for the attribute of its parent (e.g. 'home', 'meeting room' or 'office' for the node representing the attribute 'location'). Finally, a node that does not have children is called a leaf node, and it contains the final action.

A decision tree is built from a list of training examples. Such a training example is a collection of the current state of the attributes at the time of an action, together with that action. In the tree, the action of the training example will be reflected by the leaf node, while the values of the attributes can be found following the path to that node. Furthermore, a training example also contains information about the relevance of each attribute for the resulting action. It is not obvious for a computer to 'understand' which attributes

[1] We use the term 'Wizard Of Oz' here, because the application does not autonomously captures the context data, but instead the researcher has to manually input the data via a dialog-based interface.

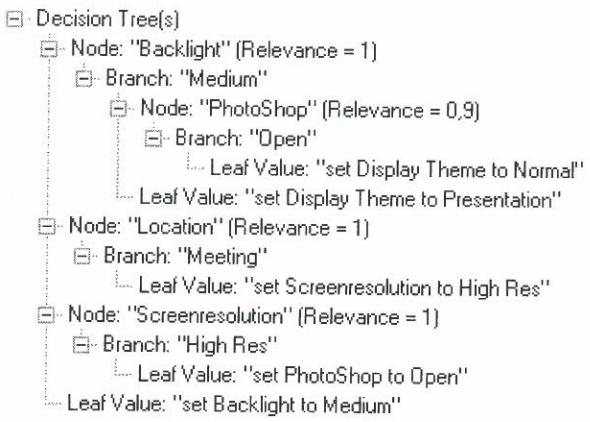

Figure 3.1: Example of a Decision Tree

are more relevant for a certain action than others. Therefore, a pragmatical approach is suggested in our implementation: attributes that are changed more recently have a higher relevance; attributes that are not changed are not relevant at all. The underlying thought is that the reason for the user's current action is probably more dependent on the attribute that lastly changed.

3.2.2 Building the Tree

With each new training example, the entire tree is recalculated. In a first step, we have to decide which attribute is placed at the root. Obviously, this is the attribute that is relevant for most training examples. Therefore, among all training examples, the attribute with the highest relevance and the highest frequency is chosen. When a training example is found that is incompatible with the current root node (having other relevant attributes), it will create a new sub-tree with a new root, relevant for that particular example. This way, tasks that differ too much from each other are stored in different subtrees. For each node representing an attribute is then a branch created for each value of that attribute.

For each branch the algorithm is recursively repeated: the most relevant attribute is chosen from the remaining training examples and is placed as a node in the particular subtree. When we get at the point where each of the remaining training examples contains the same action, this outcome is added as a leaf node. The result can be seen in figure 3.1. If more examples have a different outcome, but they cannot be further subdivided, the leave node contains all

possible actions, together with their probability. Obviously, the more often a value occurs in the list of training examples, the higher its probability gets.

3.2.3 Making Predictions

To make a prediction, the decision tree is queried to find the next possible action that can be proposed to the user. Upon the values we know for all attributes from the current application state, a suitable leaf node must be found. To find this leaf, we start with the root of the tree. With each node encountered, we follow the branch that has the same value for the given attribute as the value in the current application state. When a leaf node has been found, its value(s) is(are) returned as the result of the prediction. When no leaf node has been found, obviously there has never been such a learning sample, and hence there is no prediction.

It may also be possible to find a solution in more than one subtree. In that case, solutions are ordered according to the length of the path followed through the tree. The longer the path, the more attributes are successfully tested, and the more probable the outcome will be.

3.3 Markov Model Implementation

3.3.1 General Approach

In this part of the implementation, previously collected knowledge is stored in a first-order Markov model [Boyle 05, Souvignier 05], which may be seen as a state transition graph where each state in the diagram corresponds to a state of the system, and each transition is annotated with its probability, as shown in figure 3.2.

In our implementation, we divide the list of attributes in two classes: first the attributes that may be changed by a suggestion of the system (such as open applications, audio level, etc.); secondly, we consider the attributes that only may be detected by the system but cannot be changed, such as the location. Each attribute has a specific value, but additionally, when an attribute has not been changed since the startup of the system, its value is considered and stored in the model as 'non relevant' (N/R). This is important for the generalisation of an observed sequence of actions, because it ignores irrelevant values.

When the system is started, it resides in a known state, having a value for each attribute. As none of the attributes have been changed yet, this initial

state always maps to a state in the Markov model in which the value of each attribute is set to N/R. As soon as an attribute is changed, the new application state is compared with the internal Markov model. If there exists a state that *exactly* matches the current application state (taking into account the N/R values), this becomes the active state of the model. If there is no match between the current application state and the internal Markov model, we are probably in a new learning path, and hence the new state is added to the model. While extending the model or navigating through it, the transition probabilities are updated as described in the next paragraph.

3.3.2 Updating Transition Probabilities

Intuitively, the transition probabilities in a Markov Model could be easily calculated by counting how many times a transition had been occurred in the past with respect to the total number of transitions from that state. However, this would imply maintaining a history log for each state and each transition. Besides storing all history data, this intuitive approach would give an equal weight to all occurrences, independent whether they are old or very recent. In our approach, we assign a higher weight to recently occurred transitions, as they are probably more relevant in the learning process. A single exponential smoothing function weighting past occurrences with an exponentially decreasing factor is used, as described in [Rigole 07]. This function can be formulated as

$$\begin{pmatrix} S_1 \\ S_2 \\ \dots \\ S_n \end{pmatrix} = \alpha \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + (1-\alpha) \begin{pmatrix} S'_1 \\ S'_2 \\ \dots \\ S'_n \end{pmatrix}$$

With S_i, the newly calculated probability and S'_i the current probability in the model. x_i is either 1 or 0 dependent whether the transition was chosen or not. α is a real number between 0 and 1 that indicates the weight of the current transition with respect to those in history.

Consider the example in figure 3.2. Be $S'_1 = 0,2$; $S'_2 = 0,5$; $S'_3 = 0,3$ and α=0,1. Suppose transition two to be selected, then we can find the following values for S_1, S_2 and S_3.

$$\begin{pmatrix} \mathbf{0,18} \\ \mathbf{0,55} \\ \mathbf{0,27} \end{pmatrix} = 0,1 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + (1-0,1) \begin{pmatrix} 0,2 \\ 0,5 \\ 0,3 \end{pmatrix}$$

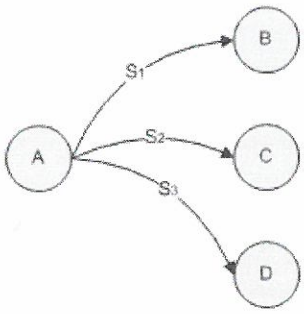

Figure 3.2: State transitions with their probabilities

Empirically, we have chosen α=0.1 when the current transition already exists in the model. When a new transition is defined (either to a new state or an existing state), the initial probability equals zero. To update the probability for this new transition, a higher weight α=0.4 is assigned in the previous formula.

3.3.3 Forecasting the User's Next actions

As soon as the user changes one of the attributes (e.g. the audio level or the location), the system tries to predict the user's goal, and formulates a suggestion to assist the user to reach this goal. The suggestion can be derived from the Markov Model as follows:

First, all transitions starting from the current new state are analysed in a depth-first manner until all end-states are found. While searching for the end-states, the probability for each end-state is calculated. As shown in figure 3.3, multiple paths can lead to the same target. In that case, the probabilities are added up. For our example, this results in:

$$P_{a,f} = S_1 \cdot S_3 \cdot S_5 + S_2 \cdot S_4 \cdot S_6$$

If a loop (a transition to a state that already has been analysed) is detected, that path is ignored.

The result of the step described in this section is an ordered list of all end-states together with their probabilities, which may be *directly* reached from the current state.

In the next step, all states in the Markov model are compared with the current application-state for similarities. This may find similar states from other

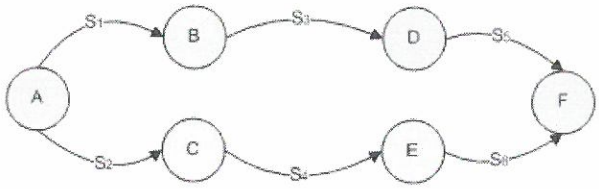

Figure 3.3: Multiple paths to the same target state

learning paths, resulting in a better generalisation of the learned samples. Therefore, for each state, we calculate a positive *and* a negative score as a result of the comparison of each individual attribute. Roughly spoken, if an attribute matches with the current state, the positive score is increased; in the other case the negative score is decreased. As explained earlier in this chapter, our Markov implementation also defines a value for 'not relevant' denoted as `N/R`, which results in a slightly more complex usage of the positive and negative scores, as is explained in table 3.1.

Finally, this step ends with a list of all states, ordered by descending positive values and negative values. The first two states in this list may be further elaborated on; provided that their total score (positive+negative) is above a certain threshold. Those state's respective end-states (with probabilities) are calculated in a similar way as described in the first step.

Finally, when the system has found all possible transitions to any relevant end-state with the respective probability, this ordered list is translated to concrete tasks that may assist the user, such as 'Open PowerPoint', 'Set Audio Volume to Loud', etc. It may be clear that end-states that suggest to adapt attributes that cannot be changed by the system (such as moving to another location) are left out.

Table 3.1: Factors to calculate state similarities

	Current App-State	State to Compare	Value
+ Factor	Changed ($\backsim$N/R)	Not Changed (N/R)	1
- Factor			-1
+ Factor	Changed ($\backsim$N/R)	Match ($\backsim$N/R)	2
- Factor			0
+ Factor	Changed ($\backsim$N/R)	Mismatch ($\backsim$N/R)	0
- Factor			-2
+ Factor	Not Changed (N/R)	Not Changed (N/R)	0
- Factor			0
+ Factor	Not Changed (N/R)	Match ($\backsim$N/R)	1
- Factor			0
+ Factor	Not Changed (N/R)	Mismatch ($\backsim$N/R)	0
- Factor			-1

For illustration purposes, table 3.1 shows an overview of all positive and negative factors. For instance, if the attribute of the application's current state has been explicitly changed, but the value of the attribute of the state we are comparing is not relevant (first row), the positive score is increased and the negative score is decreased by one. Alternatively, if it matches (second row), the positive score is increased by two, while the negative value is left unchanged. It may be clear for the reader that the values in this table are found empirically, and may be subject to change when applying this approach in a larger scale.

Chapter 4

Comparison

As described in section 3.1, this research focuses on a user interface of a PDA or a laptop, proactively suggesting the next possible actions that the user may perform. This suggestion is triggered by a context switch, and derived from previously learned examples. In order to compare both implementations, we have conducted some experiments, each focusing on a specific scenario. In the next section, we first describe the 'Wizard of Oz'-interface that will be used for the experiments. In the sections below, we then describe the focus of each scenario. In order to draw our conclusions, for each experiment we count the number of user actions, as well as the useful, useless and wrong suggestions.

Although the amount of samples in each experiment is rather limited, it may give a good impression of the value of both algorithms. We believe more samples within the same scenario would lead to slightly different relevances or probabilities, but will not end up with significantly more complex models. It may be clear however that those experiments will not give a proof of the behaviour of the algorithms with prolonged use. Therefore, we refer to the 'future work' section in this text.

4.1 Wizard of Oz Interface

In order not to cope with the 'detection' of context or machine state and the integration in a 'real' user interface, and in order to end up with manageable internal models, we have chosen for a limited set of attributes that are manually fed to the internal models via a simple dialog, as shown in figure 4.1.

It may be clear that this interface is only built for testing the underlying

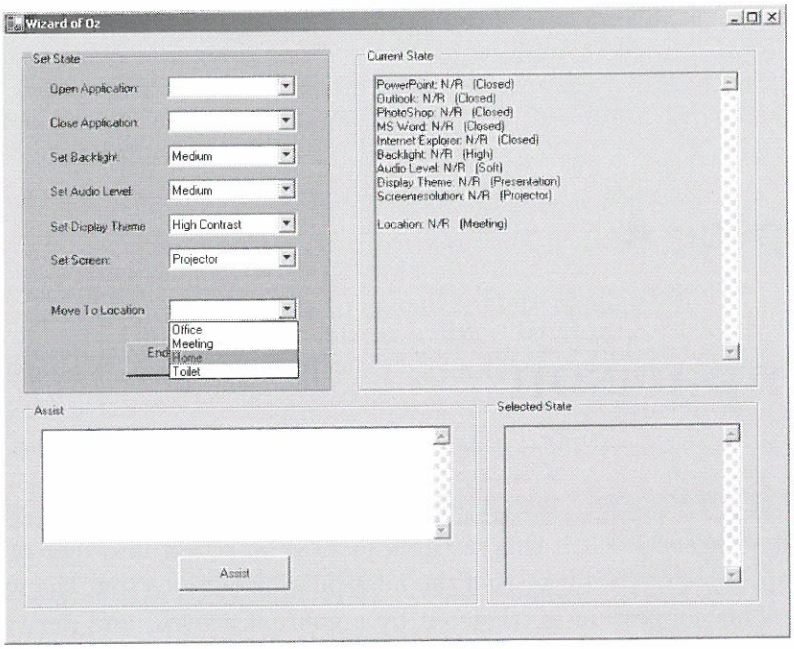

Figure 4.1: The 'Wizard Of Oz' interface

algorithms, and that ultimately the computer should be able to autonomously detect these (and other) system settings.

The first step for each sample, is to indicate the initial state of the application. This is the state the computer resides in, when he starts working. In our interface, this is done while the background of the top-left pane is dark (as shown in figure 4.1).

After finishing, by clicking the button, the combo-boxes are used to simulate the changing system settings. The top-right text-box constantly gives feedback about the current application-state.

As soon as the system is able to suggest some further actions, they are proposed in the bottom-most part of the window. As shown in figure 4.2, the listbox on the left gives the overview of all suggestions with their relevance. Relevances are calculated in percentages as a normalised result of internal calculations, but they must not be seen as an absolute chance. On the right, detailed information about the selected proposition is shown. Finally, by clicking the 'assist' button, the selected suggestion is accepted and the corresponding application state is established.

4.2 First Experiment

In the first scenario, we focus on the adaptations of the interface when a user moves to another location, independent of the machine's initial state. The experiment consisted of 5 subsequent trials, each starting from a slightly different initial computer state (initial screen resolution, initial audio level, initial location, etc...). In each trial the user moves to the meeting room where the audio level should be set to 'off'.

In total, this scenario required 9 actions for 5 trials. After the first trial in which, obviously, the audio level must be turned off explicitly, all sequent trials gave correct suggestions for both algorithms. Three times, a correct suggestion was proposed, and one time both algorithms should suggest to turn of the audio, but because the sound was already off in the initial state of that trial, the suggestion was suppressed.

4.3 Second Experiment

The second experiment is slightly more complex, as it focuses on the different computer settings when being at home or at the office. At home, the audio should be 'loud', the monitor is in 'high resolution' and mostly the application 'Internet Explorer' should be open, while 'Outlook' is closed. On the contrary, at the office, the audio level is set to 'low', there is a dual monitor setup and either 'Outlook', 'Word' or 'Photoshop' is open.

The experiment consisted of 10 trials simulating to go home and to arrive at the office. In total, the scenario required 43 actions. Both algorithms gave 23 desired suggestions. The Markov implementation gave 3 unnecessary suggestions, while the decision tree implementation gave 4. Those suggestions where all the result of the fact that there is no application that is *always* opened

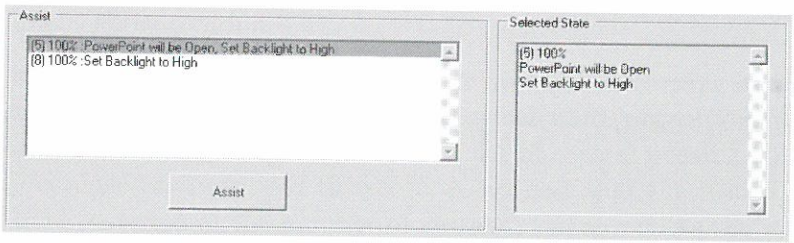

Figure 4.2: Example of a suggestion by the system

at the office. Therefore, both algorithms may suggest to open applications that are not necessary. In this experiment, however, we have to notice that the decision tree suggested to turn on the sound as a next action after a suggestion to turn off the sound had been accepted. Because this suggestion was at the very end of the scenario, when the desired state was reached, this was not recorded as a 'false' suggestion.

4.4 Third Experiment

In a third experiment we will compare how both algorithms may generalise learning paths from another situation, as well as how the algorithm behaves when exactly the same actions are performed in another order.

The first learning path contains changing the sound, theme and screen settings in an explicit location (e.g. meeting room). Next, the same sequence is desired, but now in an unspecified location. We see that the decision three has learned from the previous learning path, while the Markov model did not. In a third sequence, we desire again a similar sequence of actions, but now at an explicit location, other than the first sequence (e.g. office). Now we observe that both algorithms have correctly learned from the second sample. For the decision tree, the second learning sample is preferred above the first, as it is more general.

Finally, we define a new learning path of four actions in a first sample, and do the same actions but in reverse order in a second sample. We see that after some actions, the generalisation extension of our Markov implementation finds the similarities and does a correct suggestion, while the decision three algorithm does not make any suggestions. It may be clear that, both algorithms will make correct suggestions when the second time the reverse order is presented, because meanwhile, both have learned from the user's actions.

Chapter 5

Discussion

5.1 General Discussion

From the previous experiments, we can see that both implementations are behaving similar. In all scenarios the suggestions are nearly identical, and the number of unnecessary or false suggestions is very low. The experiments show that the decision tree implementation seems to be better in generalising and applying learned data in another context, which could also be expected when reasoning about the algorithm. As a drawback however, one can imagine that this may result in many general or false suggestions when using larger sample sets (such as turning on the sound directly after it is has been turned off, as we observed in experiment 2 in section 4.3).

A Markov-model at the other hand, is better to recall the exact learned machine state, and may be less suitable for generalising the learned data. In this context, however, the addition to search for similar states may improve this drawback. Moreover, because of the latter addition, the Markov implementation is less dependent on the order in which actions are presented, as we found in experiment three (section 4.4).

Finally, extrapolating our results to a larger number of learning samples and more complex environments, both models may calculate suggestions with a lot of actions to perform, or they may return a list of several possible final goals. This may confuse the user, and therefore we suggest that the results must be limited, both in terms of the number of suggestions, as well as the number of actions within a suggestion. For instance, only the four or five most relevant suggestions, limited to four or five subsequent steps may be shown to the user.

5.2 Performance Considerations

As both algorithms may run at a PDA, with limited resources, some consideration about performance may be useful. It is clear that our test examples are far too simple to measure the performance, but based upon the description of the algorithms in chapter 3, we can make predictions about the time complexity of both algorithms. For both algorithms, we describe the time complexity when adding a new learning sample as well as when finding a suggestion. A thorough optimisation of the algorithms, however, may imply that some parts of the algorithms can result in another time complexity.

5.2.1 Decision Tree

For each new sample, the decision tree must be rebuilt. For this purpose, all training examples have to be iterated once for each level in the tree. Searching in a well balanced tree is logarithmic with its number of nodes. With n the number of nodes and t the number of training samples, this results in:

$$O(t \cdot log(n)) \tag{5.1}$$

Searching for a single suggestion is performed by a single search operation in the tree, which is logarithmic with the complexity of the tree. Multiple suggestions leading to a desired end-state are executed as multiple independent search operations within the tree. With l the length of the path to the end state, this gives $O(1 \cdot \log(n))$. The value of l is typically small, as it has little sense to make tens of suggestions at once. Therefore, we can conclude:

$$O(log(n)) \tag{5.2}$$

5.2.2 Markov Implementation

When a new sample is presented, in a first step, the entire model is analysed to find whether the new application state already exists in the model or not. This process is linear with the number of states m in the model. If necessary, a new state is created, and the probabilities of the transitions of the previous state are updated. As this is only a local adaptation, it is independent of the model's complexity.

$$O(m) \tag{5.3}$$

To make a suggestion, from the current state, all possible end-nodes are searched for, and the respectable probabilities are calculated. As loops in the graph are ignored, this is a linear function with the number of nodes present in the sub-graph (s). In a second step the Markov model is entirely analysed to find states that are 'similar' to the current application state; this is an operation which is linear with the number of nodes in the graph. The time complexity is:

$$O(s \cdot m) \tag{5.4}$$

In a worst case scenario, s is equal to m (as it is a sub-graph), resulting in:

$$O(m^2) \tag{5.5}$$

As from our experiments, the complexity of the decision tree (n) and the Markov model (m) appear to be proportional, we can conclude from this reasoning that the decision tree implementation is a factor log(n) slower for adding new samples to the tree. Alternatively, our Markov implementation[1] is slower (quadratic compared to logarithmic) for calculating a suggestion.

[1] Also searching for similar states when making a suggestion

Chapter 6

Conclusion and Future Work

In this work, we compared two algorithms that may be used to support a proactive user interface. As a result of a user action or a context switch (such as moving to another location), the interface proactively suggest the following probable actions. We described the implementation of both a decision tree algorithm and a Markov model. Both algorithms were tested using a 'Wizard of Oz' interface which has been used to provide input according to three predefined scenarios.

From the results, we could conclude that both algorithms behave more or less in the same way, and that the differences are very small. However it appears that a decision tree is more suitable to generalise samples learned in a specific context, while a Markov model is more suitable when the learning sample must be recalled. Moreover, due to its nature, a decision tree may sometimes generate false suggestions.

When considering the performance of both algorithms, we also see little difference. Although this is a preliminary conclusion, decision trees appear to be less optimal to integrate new samples in a large model, while our Markov implementation is worse in order to calculate a new suggestion.

As the experiments look promising for both algorithms, a real and more extensive evaluation may be conducted. Therefore, it is necessary to build a tool that is able to capture and change the relevant computer attributes and detect the current location or context. As this future tool can then be easily used in practice, larger sample sets and more complex models can be built. Furthermore, a practical evaluation will also allow us to query end-users for their subjective satisfaction for either algorithm.

Acknowledgements

The author would like to thank Kristof Verpoorten for the very appreciated collaboration, as well as Prof. Dr. Kris Luyten for sharing his knowledge, discussing the experiments and providing valuable feedback.

Bibliography

[Boyle 05] RD Boyle. *Hidden Markov Models.* `http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/main.html`, 2005.

[Byun 04] H.E. Byun & K. Cheverst. *Utilising context history to support proactive adaptation.* In Applied Artificial Intelligence, vol. 18, nr. 6, pages 513–532, July 2004.

[Cook 03] D. Cook, M. Youngblood, E. Heierman, K. Gopalratnam, S. Rao, A. Litvin & F. Khawaja. *MavHome: An Agent-Based Smart Home*, 2003.

[Dey 04] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li & Daniel Hsu. *a CAPpella: Programming by Demonstration of Context-Aware Applications.* In Proceedings of CHI 2004, Vienna, Austria, April 2004.

[Espinosa 05] José Espinosa & Henry Lieberman. *EventNet: Inferring Temporal Relations Between Commonsense Events.* In Proceedings of MICAI 2005, pages 61–69, 2005.

[Lieberman 06] Henry Lieberman & José Espinosa. *A goal-oriented interface to consumer electronics using planning and commonsense reasoning.* In IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces, pages 226–233, New York, NY, USA, 2006. ACM Press.

[Mitchell 97] Tom Mitchell. Machine learning. McGraw-Hill Education (ISE Editions), October 1997.

[Petzold 05] Jan Petzold, Faruk Bagci, Wolfgang Trumler & Theo Ungerer. *Next Location Prediction Within a Smart Office Building.* In

Proceedings of ECHISE 2005 - 1st International Workshop on Exploiting Context Histories in Smart Environments (held in Conjunction with the Pervasive 2005 Conference), Munich, Germany, May 2005.

[Rigole 07] Peter Rigole, Tim Clerckx, Yolande Berbers & Karin Coninx. Task-driven automated component deployment for ambient intelligence environments. Submitted to the Elsevier Pervasive and Mobile Computing Journal (PMC), 2007.

[Singh 02] Push Singh, Thomas Lin, Erik Mueller, Grace Lim, Travell Perkins & Wan Li Zhu. *Open Mind Common Sense: Knowledge acquisition from the general public.* In roceedings of the First International Conference on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems, P. Irvine, CA, 2002.

[Souvignier 05] Bernd Souvignier. *Wiskunde 2 voor kunstmatige intelligentie, Deel III. Probabilistische Modellen.* `http://www.math.ru.nl/souvi/wiskunde2_05/les12.pdf`, 2005.

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten uit twee landen: de Universiteit Hasselt en de Universiteit Maastricht.
De opleidingen Informatica/Kennistechnologie/ICT en Biomedische Wetenschappen/Moleculaire Levenswetenschappen zijn reeds ondergebracht in dit samenwerkingsverband. Ook in andere wetenschapsdomeinen wordt de samenwerking tussen beide universiteiten bevorderd.

www.unimaas.nl

Universiteit Maastricht
Postbus 616
NL-6200 MD Maastricht
Tel.: 0031(0)43 388 222

www.uhasselt.be

Universiteit Hasselt | Campus Diepenbeek
Agoralaan | Gebouw D
BE-3590 Diepenbeek
Tel.: +32(0)11 26 81 11