

Probabilistic and Decision-Theoretic User Modeling in the Context of Software Customization

Depth Oral Report
Department of Computer Science, University of Toronto
Bowen Hui
bowen@cs.utoronto.ca

June 4, 2004

Abstract

Research in the field of *user modeling* has aimed to supersede the current “one-size-fits-all” trend in software development that forces users to change their behaviour according to the preprogrammed functions. This paper discusses aspects of user modeling and its relevance to *software customization*. In particular, we focus on user modeling techniques that utilize probabilistic and decision-theoretic models. We examine the fundamental limitations of each modeling technique with respect to the goals of user modeling and software customization. Finally, we propose to use *partially observable Markov decision processes* as the underlying framework to model users for customizing software, and we plan to apply techniques from *inverse reinforcement learning* and *preference elicitation* to learn the user’s utility function. We present our prototype of a typing assistant modeled as a partially observable Markov decision process. Lastly, we outline the future directions of how to incorporate inverse reinforcement and preference elicitation work into our project.

Contents

1	Approaches to Software Customization	3
1.1	Modeling Techniques	3
1.2	Outline	4
2	Background on Modeling Technologies	5
2.1	Bayesian Networks and Dynamic Bayesian Networks	5
2.2	Markov Decision Processes	8
2.3	Partially Observable Markov Decision Processes	10
2.4	Inverse Reinforcement Learning	14
2.5	Preference Elicitation	15
3	Applications	16
3.1	Customization Applications Using Rules, Heuristics, and Plan Libraries	16
3.2	Customization Applications Using Probability and Utility Theory	17
3.3	Customization Applications Using (Dynamic) Bayesian Networks and (Dynamic) Decision Networks	19
3.4	User Interaction Applications Using Markov Decision Processes	21
3.5	User Interaction Applications Using Partially Observable Markov Decision Processes	23
3.6	User Interaction Applications Using Inverse Reinforcement Learning	23
3.7	User Interaction Applications Using Preference Elicitation	24
4	Summary of Open Problems in User Modeling	25
4.1	Issues Across Models	25
4.2	Evaluation Difficulties	25
4.3	Learning Model Parameters	25
5	Our Proposed User Model for a Typing Assistant	26
5.1	Typing Assistant Model	26
5.2	Implementation	29
5.3	Simulations and Evaluations	29
6	Future Directions	30
6.1	Applying POMDPs to Various Software Customization Problems	30
6.2	Evaluation	30
6.3	Learning the Reward Function	31

1 Approaches to Software Customization

Software development has generally adopted a “one-size-fits-all” model in which software is designed for a few classes of user groups, rather than tailoring it to the needs of particular users. Clearly, different individuals with varying levels of expertise, different preferences, needs and goals require different kinds of services. As argued previously [Bru01a, HLM03], the dimensions that need to be considered as part of software design include the user’s goals, preferences, and skills. Several sub-fields in Computer Science have begun to address the need for *software customization*, each from its own perspective. These fields include software engineering (SE), human-computer interaction (HCI), and artificial intelligence (AI).

In SE, the focus is on customization at *design time*. Rather than packing all thinkable functionality into a single generic system, program families [KKLL99] attempt to define the set of allowable alternatives for a given set of features included in a family. This approach offers a design-level perspective on the set of alternative customizations supported by a program family, and does not easily translate to user needs, skills, and preferences. More recently, goal analysis was used to systematically generate design alternatives and scores from a user profile (skills and preferences) were used to choose among the best alternative [HLM03]. Nonetheless, the customization process does not end at the design stage. Even after software deployment, the system should recognize individual changes in user goals, preferences, and skills. For this reason, work in HCI and AI address the *run time* aspect of software customization, generally coming from opposite angles. Work from HCI concentrates on providing users with *adaptable* software so that the user remains in control and can directly manipulate the functionality and interface components of a software (e.g., [MBB02]). On the other hand, research in AI pushes for *adaptive* systems that monitor user actions in order to infer user goals and preferences, which might result in a revised user interface (e.g., [HBH⁺98, CV01]). We do not pursue the debate between adaptable versus adaptive systems. Instead, we focus on adaptive techniques while allowing for adaptable user control.

We approach the problem of run time software customization from a *user modeling* perspective. Our goal is to identify and monitor features of the user that are relevant to their tasks, so that the software is tailored toward the user’s needs by observing his behaviour or reactions to the system. We may think of this approach as building a software assistant (anthropomorphized or otherwise) that helps the user in accomplishing his tasks. In contrast to much of the AI work in user modeling (UM), we believe that the system should model both the system state as well as features of the user. One advantage in modeling explicit user features is that user profiles may be learned and reused in other applications. This approach will take us a step toward the generalization of software customization models.

1.1 Modeling Techniques

This paper surveys techniques and applications for run time software customization. Much of the work that deals with run time customization originates from the UM community. One of the objectives in this work is to determine the user’s current goal with respect to the system environment. Naturally, the system components and functionality are modeled as a plan from the point of view that a user moves along different parts of the system in order to achieve his goal. This approach is generally called *plan recognition*, where the system attempts to recognize the user’s plan and helps the user complete his plan more efficiently. As researchers began to acknowledge the importance of user specific information such as skills and preferences, methods in plan recognition were combined with the use of heuristics, probability theory, and utility theory. We review some of applications in these areas in Sections 3.1 and 3.2.

A major drawback with plan recognition approaches is that the underlying model is deterministic. Given that the software designer cannot anticipate all the necessary information or obstacles in a problem domain, *uncertainty* arises in both the problem formulation and the modeling of (any) user features. To explicitly model the sources of uncertainty in our problem and act optimally, we need to be able to express our beliefs over the possible situations while taking into consideration the history of actions and observations. Modeling with Bayesian networks (BNs) and dynamic Bayesian networks (DBNs) take this perspective. In this manner, the system environment is modeled as states in the BN, the user’s goals can be inferred from observing the user. Appropriate system actions can then be chosen according to some criterion, perhaps comparing its level of contribution against user-defined thresholds. Combined with the states of a BN, influence diagrams (IDs) can model the system’s actions and various factors that contribute to the utility of its actions. Then, system actions can be chosen to optimize expected utility

gain with respect to the inferred beliefs. In this manner, we can reason explicitly about the utility of a system action based on what it offers and when it is offered. A skill profile of the user can also be inferred in a similar manner. In Section 3.3, we will review applications that adopt this Bayesian perspective.

Modeling the software environment as states in a BN takes care of domain uncertainty, but the uncertainty in the user is still largely ignored. If we want to obtain user satisfaction, we must model features of the user that contribute toward that satisfaction. For example, if the system agent can infer the user’s goal exactly, should the agent just do it on the user’s behalf? Should the system employ scaffolding to bring the user closer to the goal? What if the user is highly independent and does not like to accept other people’s help? What if the user is highly dependent and may end up relying on the system for basic needs? Little work has touched upon this aspect of modeling using BNs, due to the difficulty in assessing human cognitive features. We mention two projects that incorporate system variables and user variables into the state space in Section 3.3.

Modeling user preferences is more complicated than just tracking past actions and observations. One complication is that there is often a large set of alternatives available to the user. It is infeasible to present all the options to the user or to ask the user for his preferences every time uncertainty arises. What’s more is that it matters *when* system help or interaction takes place, as it affects future preferences and behaviour. Imagine that the system agent has the option of interrupting and offering help to the user now or remaining quietly observing the user. If the system decides to offer help, but the user is just below the tolerance of welcoming interruptions, then the system has in fact sacrificed its chance to offer better help at a later point. In an alternate scenario, given that system interruption is welcomed, the system still must choose informative queries for the user because these queries have an effect on what the system chooses to ask at a later time. These problems illustrate the *sequential* nature inherent in UM.

To model sequential decision making, we turn to Markov decision processes (MDPs). MDPs model a set of actions and a set of utilities for each state-action pair for choosing the optimal action over the specified horizon. We are unaware of existing applications that use MDPs for software customization. But related applications – recommendation systems – are discussed in Section 3.4.

Realistically, because the state of the world is not fully observable, especially if we model the user’s cognitive variables, we need to account for *partial* observability. In other words, given a MDP of the world and that the world is partially observable, we cannot know with certainty the true state we are in. Therefore, we propose to use a partially observable Markov decision process (POMDP) as a more general model for software customization. One can think of a POMDP as keeping a distribution over possible states and takes actions with respect to that belief. The states in these models typically describe system variables and the agent’s environment. However, in UM, we believe that user variables also need to be modeled as part of the state space. Again, we are unaware of software customization applications with POMDPs, so instead, we discuss a robot application in Section 3.5.

For applications that interact jointly with the user and that act on the user’s behalf, a model that represents and learns what the user prefers is necessary. To do this in a POMDP means that the system needs to learn the user’s reward function. Given a model of the world and feedback on system behaviour, we turn to the literature in *inverse reinforcement learning* (IRL). However, the agent does not always receive feedback on its actions. At times, it may be necessary to elicit feedback from the user, or even actively seek information from the user. For this reason, we turn to *preference elicitation* (PE) to incorporate querying techniques into the IRL framework. Sections 3.6 and 3.7 review recommendation systems in these two areas. Altogether, we will see how the complexity and expressiveness of the various modeling techniques unfold.

1.2 Outline

We begin with a review of the various modeling techniques that lead up to our proposed view of run time software customization in Section 2. In Section 3, we survey applications that use the modeling techniques from Section 2. Whenever possible, we discuss applications specifically for customization. To date, the more general models (such as MDPs and POMDPs) do not yet have any application for customization. In those cases, we will discuss applications that involve human interaction. Section 4 summarizes the problems identified among the various approaches of UM in the context of software customization. To address these problems, Section 5 presents our prototype model of a typing assistant with initial simulations. Finally, Section 6 outlines the next steps in our project.

2 Background on Modeling Technologies

In this section, we review preliminary material for several modeling techniques. The complexity and generalizability unfolds as we progress through the models. To illustrate the differences among the modeling techniques, we will use a running example for a simple interface agent that monitors user actions.

2.1 Bayesian Networks and Dynamic Bayesian Networks

Everyday, we need to make a series of *decisions* based on what we observe in our environment. Probabilistic modeling allows us to capture and reason with the *uncertainty* in our decision making process and utilities let us express our preferences among the alternatives in these decisions. With this, we turn to several *graphical models* as the basis of our framework.

A *Bayesian network* (BN) [Pea88], or Bayes net, allows us to exploit independence by specifying direct conditional dependencies and inferring probabilistic relations in a joint distribution. A more general framework that builds on top of Bayes nets so that agent actions and utilities are explicitly modeled is called an *influence diagram* (ID) [HM84, Sha86]. To model dependencies across time, Bayes nets are extended to *dynamic* Bayes nets (DBNs) and decision networks are extended to dynamic decision networks (DDNs). We first review Bayes nets and DBNs and then turn to IDs.

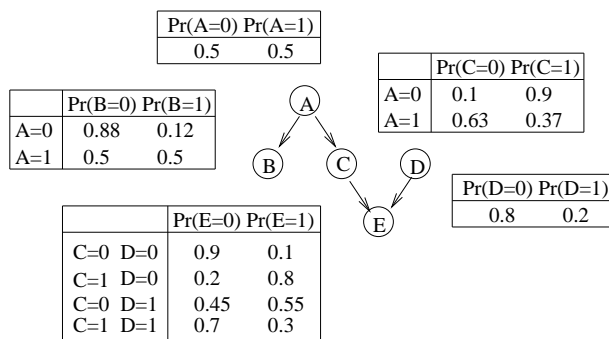


Figure 1: Simple Bayes Net

Bayes nets are directed acyclic graphs that consist of nodes which correspond to random variables and directed edges which correspond to causal influences. A simple Bayes net with five discrete variables is shown in Figure 1. For example, variable A has two values, zero parents, and two child nodes. The probability of A having either value is uniform, and it is expressed in a table called the *conditional probability table* (CPT). CPTs are defined over families in a Bayes net. The joint probability of a BN is defined as the product of local conditional probabilities, as in Equation (1).

$$Pr(V) = \prod_{X \in V} Pr(X | Parents(X)) \quad (1)$$

The joint of the BN in Figure 1 is: $Pr(A, B, C, D, E) = Pr(A)Pr(B|A)Pr(C|A)Pr(D)Pr(E|C, D)$. Each node in the BN is independent of its non-descendants given its parents. For example, E is independent of A given C and D . General independence can be tested with a graphical property called *d-separation* [Pea88].

Let us turn to an example involving a user interacting with software in Figure 2. If the user needs help, then he is likely to display signs, such as making mistakes often (e.g., backspaces), browsing for more information, or pausing because he is not sure know what to do. This is exhibited in the variable, “Need Help”. On the other hand, if a user is an easily distractible person, then he will show signs of distraction, such as browsing and pausing. This is exhibited in the variable “Easily Distractible”. Observations, the nodes that are drawn in double circles, are variables that can be fully observed. In contrast, “Need Help”, “Easily Distractible”, and “Browse” refer to user states, which cannot be observed. Note that distractibility of a person refers to a trait so its value persists over time, whereas neediness might change frequently. A BN cannot capture this distinction because it only has one “copy” of each variable. We see how this can be dealt with when we turn to DBNs.

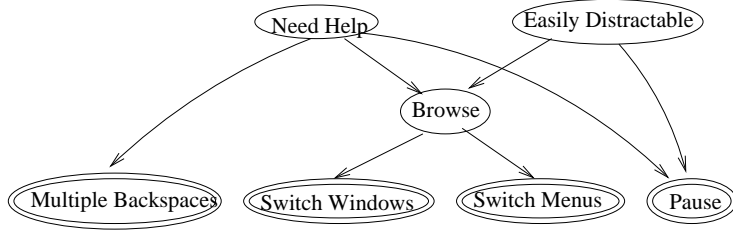


Figure 2: Interface agent example with Bayesian network.

With a Bayes net, we can represent our beliefs of the interaction between some random variables. In many cases, we are interested in *inferring* the likelihood of specific events after having made an observation, e.g., query for $Pr(X|Y = y)$. Because joint distributions of realistic applications tend to involve many variables, computing this query would require summing out or integrating all the other dependent variables in the joint. *Variable elimination* [Dec99] is a general algorithm for exact inference. Suppose we wanted to compute $Pr(C, E = 1)$ from the joint in Figure 2. We could straightforwardly compute the probability using Equation (2). If we leave the summations in the front of the equation, it would mean doing unnecessary computations such as summing different values of D when calculating $Pr(B|A)$. By splitting up the sums and pushing them in as far as possible, we get the expression in Equation (3). Notice we have moved $Pr(B|A)$ out of the summation over D . This idea exploits structural independence by moving variables outside of summations that are not needed locally. Unfortunately, optimal variable ordering is an NP-complete problem, so in practice, efficient orderings are approximated.

$$Pr(C, E = 1) = \sum_{A, B, D, E=1} Pr(A)Pr(B|A)Pr(C|A)Pr(D)Pr(E|C, D) \quad (2)$$

$$= \sum_A Pr(A)Pr(C|A) \sum_B Pr(B|A) \sum_D Pr(D) \sum_{E=1} Pr(E|C, D) \quad (3)$$

Another general exact inference algorithm is the *junction tree* algorithm [LS88, HD94]. From the Bayes net, it creates a secondary structure called the junction tree (alternatively called *join tree* or *clique tree*). The nodes in this structure basically correspond to cliques (fully and maximally connected) in a moralized, triangulated BN. Each clique has a distribution represented as a potential so that all the computations are done by multiplying and summing over potentials. The clique tree is initialized with the model's CPTs by creating a potential for each and multiplying them into appropriate cliques. When evidence is available, its observed value is entered into an appropriate clique. To ensure consistency, we perform message passing across the tree. To compute $Pr(V)$ for some variable set V , we first identify the clique that contains V and then sum out any unnecessary variables from the clique and normalize if necessary.

DBNs [DK89] add a temporal component to the model. Given a BN, we may observe a stream of events such that the values of its variables evolve over time. To model temporal dependencies more accurately, we model multiple time slices and allow variables to have influences across time. For example, a two-slice DBN consists of two copies of the underlying BN with arcs between them. To extend a BN into a DBN, we copy the BN slices over time, and draw in temporal dependencies. Continuing with the interface agent example in Figure 3, the variables referring to the user states carry over time. However, depending on the definition of the duration in a time slice, the observations may or may not have temporal dependencies. Consider the case where a time slice advances per system event. In that case, if the user started switching between windows, then in the next time step, the user is more likely to switch to more windows. However, if a time slice were defined over an interval (say, 5 seconds), whether the user is switching between windows now may have no influence on whether the user will be switching windows in the next time slice. For simplicity in illustration, we assume in Figure 3 that there are no dependencies among observations across time steps.

Inference in DBNs [Kja92, Mur02] now refers to computing the marginal at a specific time or over an interval, given evidence over an interval. Here, we restrict our attention to *monitoring*, which is to compute $Pr(X_t|y_{1:t})$. In the simplest case given a bounded horizon T , we could *unroll* the DBN over T

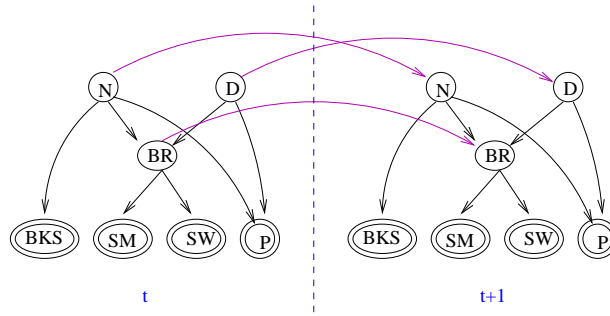


Figure 3: Interface agent example with dynamic Bayesian network. Notation: N is “Need Help”, D is “Easily Distractible”, BR is “Browse”, BKS is “Multiple Backspaces”, SM is “Switch Menus”, SW is “Switch Windows”, P is “Pause”.

time slices and then apply (static) BN inference. A more space and time efficient method is to maintain a two-slice DBN template. Inference starts by initializing slice 1 in the template with the prior distribution, observing evidence and marginalizing out slice 2 in the template, taking a new template and repeating this process. Initially, the prior distribution is defined by the CPTs in the model. Over time, the priors are the marginalized joint distributions computed in the previous inference step.

The junction tree algorithm (mentioned above) can also be used for DBN inference. We could create a join tree representing a two-slice DBN, initialize it with its CPTs, observe evidence, compute the marginal, and repeat these steps at each time slice. However, repeating these steps means creating a new join tree and initializing it at each slice. We could actually avoid doing this if the new observations was not “zeroed out”¹ by the previous observations. When we make a new observation, if it was not zeroed out previously, then we just enter it into the join tree as usual. Doing this saves the work of having to create and initialize a new tree. Otherwise, we retract the old observations by reinitializing the join tree and then entering the new observation as before.

A DBN typically involves unobservable variables which we want to infer the value of. This partial observability makes us keep track of information that is further than one time step ago. Therefore, even nicely factored models become completely correlated quickly, making exact inference infeasible. Figure 4 shows an example of a BN unrolled over 3 time steps. Even when the state variables are independent within a given time slice, once the model is unrolled, all of the variables become correlated. For larger, realistic DBNs, exact inference is computationally infeasible.

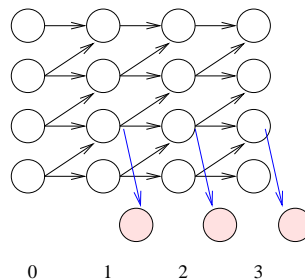


Figure 4: Unrolled BN for 3 time slices (taken from [BK98]p.2).

A general approximate inference algorithm for DBNs is the *Boyen-Koller* algorithm [BK98]. This algorithm enforces independence over parts of the Bayes net by introducing clusters over variables. Then it uses an exact inference procedure, such as the clique tree algorithm, to perform inference. At each time step when we roll up, we remove the correlations between clusters and force the clusters to be independent again. By doing so, we are in effect ensuring that the cliques remain the same size over time.

¹When a variable’s value is observed to be false, it gets the value 0, and otherwise it gets the value 1.

An ID [HM84, Sha86] extends a BN by adding actions and utilities. Figure 5 shows the corresponding ID for our BN model from Figure 2. We added an extra hidden node, “consider help”, to indicate the system’s inferred value of whether the user will consider help from the system or not. In the ID, a binary-valued action node is added for whether a help box should be presented to the user. This box contributes to the utility of the decision, which is also affected by whether the user needs help and whether the user is easily distracted. The expected utility of offering help is compared against the expected utility of not offering help, and the action with higher value is taken.

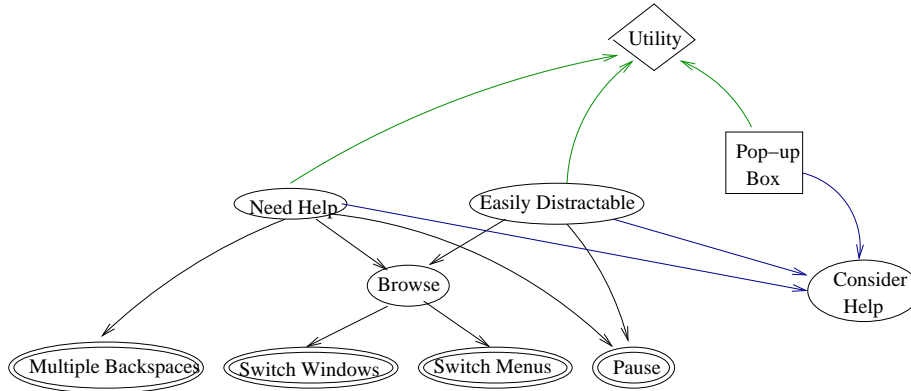


Figure 5: Interface agent example with influence diagram.

Although Figure 5 presents a one-shot decision, IDs can model multiple decisions to be made in sequence. An ID can have multiple action and utility nodes that depend on one another. However, it does not have a temporal component with repeated structural dependencies over time as we saw with DBNs. Alternatively, we can maintain our view of repeated decisions made over time slices. In this sense, we would view the model in Figure 5 as a decision made in a time step, and model its influences over time, including any temporal influences that actions and utilities may have. This idea is illustrated as a *dynamic decision network* (DDN) in Figure 6. In essence, DDNs offer a structured, compact representation of POMDPs [TS90].

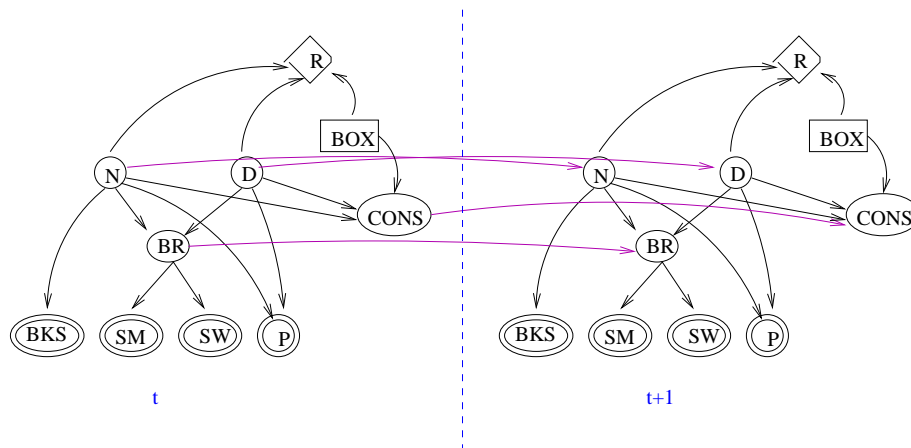


Figure 6: Interface agent example with dynamic decision network.

2.2 Markov Decision Processes

An MDP is a mathematical model for a expressing a sequential decision making problem under a decision-theoretic framework [Put94]. Formally, an MDP is a tuple consisting of $\langle S, A, T, R, h, \gamma \rangle$ such that:

- S , set of world states
- A , set of actions for the agent
- $T : S \times A \times S \rightarrow [0, 1]$, transition function rewritten as $T(s, a, s') = Pr(s'|s, a)$, $\forall s \in S$ and $\forall a \in A$
- $R : S \rightarrow \mathbb{R}$, agent's reward function², rewritten as $R(s)$, $\forall s \in S$
- h , the horizon of the problem
- $\gamma \in [0, 1]$, the discount factor at each time step

An agent acting in an MDP can fully observe its state space at every time step. In general, the states and actions in the MDP can be discrete or continuous, and finite or infinite. The transition function is a stochastic function that maps from states to states after taking some action. The state transition function is Markov so that $Pr(s_t|s_0, s_1, \dots, s_{t-1}, a) = Pr(s_t|s_{t-1}, a)$. States are associated with real-valued rewards which an agent can accumulate along its path. Common criteria for an agent to act optimally is by maximizing its total expected reward over a finite horizon, or total expected discounted reward over an infinite horizon. We express the optimality criterion as $E[\sum_{t=0}^h \gamma^t R(s_t)|s_0]$, where $0 \leq \gamma < 1$ and $h = \infty$ is used for an infinite horizon, and $\gamma = 1.0$ for finite horizon. From here on, unless otherwise specified, we will restrict our discussion to finite MDPs over an infinite horizon.

An action decision made at one point has consequences on where the agent goes to, what future actions should be taken thereafter, and the rewards collected. This *sequential* nature is inherent in many realistic decision making problems and can be captured in an agent's policy, $\pi : S \rightarrow A$. To evaluate among different policies, we define the value of a policy as:

$$V^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|s_0, \pi] \quad (4)$$

Then, a policy is optimal, denoted as π^* , if $V^{\pi^*}(s) \geq V^\pi(s), \forall s \in S, \forall \pi$. In particular, given a policy π , the value function V^π at a given time step, t , is defined as:

$$V_t^\pi(s) = R(s) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_{t-1}^\pi(s') \quad (5)$$

A dynamic programming (DP) formulation of value iteration shows that the value of being in a state can be computed based on the value of its previous state. In particular, we can rewrite the value function in terms of the value a state-action pair, as in Equations (6-7) [Bel57].

$$Q_t(s, a) = R(s) + \gamma \sum_{s' \in S} T(s, a, s') V_{t-1}(s') \quad (6)$$

$$V_t^*(s) = \max_{a \in A} Q_t(s, a) \quad (7)$$

Equation (6) is referred to as the *Q-value* function that represents the quality of a state-action pair. Then the optimal value function is the value of the action that maximizes the *Q* function.

Given the parameters of the model, we turn to the interface agent example in Figure 7. Because the world is fully observable, all the variables become part of the state in this MDP. (Note that it is not realistic to assume that an interface agent can observe a user's neediness, distractibility, level of browsing, or the level of considering help.) The size of the state space is the product of domain size of each state variable. The action variable, *BOX*, affects what the next state is. The transition function describes the probability of going from one state to another after taking each possible action. Reward, *U*, is collected as a function of the current state.

One standard solution of the MDP is to use *value iteration* which iteratively computes the value function of each state by implementing Equations (6-7). Initially, $V_0^* = R$. At each iteration step, we perform the Bellman backup, V_{t-1}^* , and take the best action corresponding to s at t . A sequence of value

²More generally, the reward function is $R : S \times A \times S \rightarrow \mathbb{R}$ so that actions and state transitions also contribute to rewards. For simplicity, the remaining discussion focuses on a reward function of the form $R(s)$. Note that the model and results can be extended to account for $R(s, a, s')$.

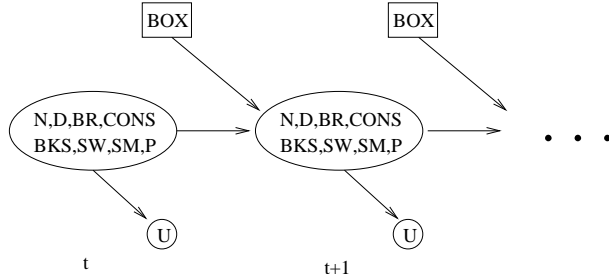


Figure 7: Interface agent example with Markov decision process.

functions is computed until convergence. With an infinite horizon, stopping at the t^{th} iteration when $\|V_t(s) - V_{t-1}(s)\|_\infty < \frac{\epsilon(1-\gamma)}{2\gamma}$ guarantees that $\|V_t(s) - V^*(s)\|_\infty < \epsilon$ [Put94]. From Equations (6-7), we can see that each iteration requires $O(|S|^2|A|)$ computations.

Another common approach to solving MDPs is *policy iteration* [How60]. At each time step, a sequence of improving policies is computed until $\pi_t(s) = \pi_{t-1}(s), \forall s \in S$. Value iteration converges linearly whereas policy iteration converges at least linearly and quadratically under some conditions [Put94]. The optimal value function can also be expressed as a set of linear inequalities, $V \geq R^{\pi^*} + \gamma P^{\pi^*} V$, so that a linear program can be used to solve for the optimal value function.

So far, the state space, S , takes on a *flat* representation. But if we decompose S into a set of features, then the state representation can be exponentially smaller. In particular, let $\{X_i\}$ be the set of state variables describing S so that all the combinations of x_i correspond to the enumeration of S . Then, $|S| = \prod_i |X_i|$. We refer to this compact representation as the *factored* representation of S . Similarly, the other parameters of the MDP can also take on a factored representation [BDH99].

On the other hand, the number of states is exponential in the number of variables. This problem is called the “curse of dimensionality” [Bel57]. The algorithms described above require that the complete state space be enumerated. There have been approaches that attempt to escape from the curse of dimensionality problem. First of all, note that the MDP can be represented as a DBN. As a DBN, the model is factored, and can result in as much as an exponential reduction in terms of space complexity. Now, taking context independence into consideration, the model parameters may be reduced further (the amount of reduction obviously depends on the nature of the problem). An algorithm that exploits context independence is SPUDD [HSAHB99], which uses algebraic decision diagrams (ADDs) as the underlying data structure in value iteration. Here, the model parameters, intermediate value functions, and intermediate policies are represented as an ADD – a decision tree-like structure that “merges” identical subcomponents across variables. To leverage further context independence, “similar” subcomponents (states, values, etc.) can be aggregated so that the result is a even more compact model [SAHB00]. In this way, value iteration can be modified to operate on the minimum and maximum values of the aggregated states to find a near-optimal solution.

Neuro-dynamic programming [BT96] describes a class of DP methods that attempts to tackle the curse of dimensionality, as well as the problem when an accurate MDP model is lacking. State feature vectors are used as a compact representation of S and they are input to a function approximator (e.g., a neural network, linear approximator) to approximate V^* . One could also partition the state space into subsets, possibly according to the “similarity” of state features, assuming each subset corresponds to a separate *subvalue* function. Each subvalue function can be approximated individually and then combined together to form a value function approximation over the entire state space. The overall methodology of neuro-DPs can also be used to approximate policies (since policy iteration requires computing V^π).

2.3 Partially Observable Markov Decision Processes

A POMDP [Dra62, Ast65] relaxes the full observability assumption of an MDP. In this way, the states in the POMDP are no longer fully observable. Intuitively, a POMDP has an underlying MDP. But because the agent cannot observe the state of the world entirely, it collects observations from the world to infer its current state. A POMDP is formally defined as tuple consisting of $\langle S, A, T, R, O, Z, h, \gamma \rangle$ such that:

- S , set of world states
- A , set of actions for the agent
- $T : S \times A \times S \rightarrow [0, 1]$, transition function, rewritten as $T(s, a, s') = Pr(s'|s, a)$, $\forall s \in S$ and $\forall a \in A$
- $R : S \rightarrow \mathbb{R}$, agent’s reward function, rewritten as $R(s)$, $\forall s \in S$
- O , set of observations from the world
- $Z : S \times A \times O \rightarrow [0, 1]$, observation function rewritten as $Z(s, a, o) = Pr(o|s, a)$, $\forall s \in S$, $\forall o \in O$, $\forall a \in A$
- h , the horizon of the problem
- $\gamma \in [0, 1]$, the discount factor at each time step

Note that in an MDP, $O = S$ and Z is deterministic. Our discussion will focus on finite POMDPs with infinite horizon, unless other specified. (Although see Section 2.5 for a POMDP model with continuous states, continuous actions, finite observations, and an infinite horizon.)

Since the world is partially observable, an agent needs to keep track of the *history* of actions and observations in order to infer its current state and course of action. However, keeping an entire history is cumbersome and specifying a policy over the space of histories can be difficult. One way around this is to maintain a belief state over the state space, $bel : S \rightarrow [0, 1]$, which serves as a sufficient summary of the history [Ast65]. To update the belief state, we use Bayes rule and condition bel on the previous action and observation as defined in Equation (8), where α denotes the normalizing constant.

$$bel_{t+1}(s_i) = \alpha \sum_{j=1}^n bel_t(s_j) T(s_j, a_t, s_i) Z(s_i, a_t, o_{t+1}) \quad (8)$$

To illustrate, we use the interface agent example shown in Figure 8. The main difference between this model and the MDP (Figure 7) is that observations are separated from state variables to indicate that some of the state variables in the MDP are, in fact, not observable. In addition, this POMDP would maintain a belief state over the possible states.

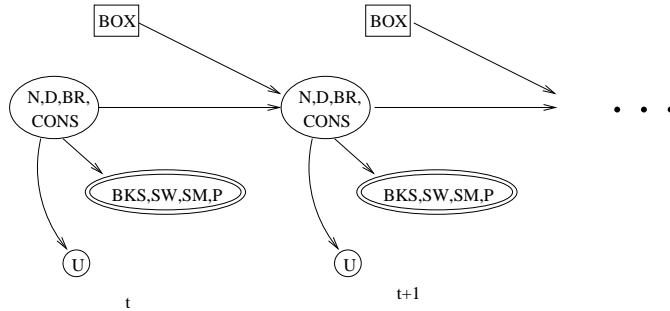


Figure 8: Interface agent example with partially observable Markov decision process.

Given a specific POMDP, our goal is to find a *policy* that maximizes the expected discounted sum of rewards attained by the system. Since the system state cannot generally be known with certainty, a policy maps either action-observation histories or belief states into choices of actions. Representing action-observation histories requires considering all “branches” of observations for every action at every time step. Graphically, we could represent this policy as a tree, rooted with an action node, with an arc for each possible observation leading to the next possible actions, and so forth. This is called a *conditional plan*, as each next best action is conditioned upon the observation made. Unfortunately, the size of a conditional plan grows exponentially with the horizon. We adopt the representation of a policy mapping from a continuous belief space to actions. Later, we revisit the history-based representation when we discuss finite state controllers.

To evaluate how good a policy is, we use the value function defined in Equation (9):

$$V^\pi(\text{bel}_0) = E\left[\sum_{t=0}^h \gamma^t \sum_{s \in S} \text{bel}_t(s) R(s, \pi(\text{bel}_t)) \mid \pi, \text{bel}_0\right] \quad (9)$$

Starting at bel_0 and executing policy π over h time steps, we compute the value of π by summing up the expected rewards in the respective belief states discounted at each time step. Given an initial belief state bel_0 , π_i is better than π_j if $V^{\pi_i}(\text{bel}_0) > V^{\pi_j}(\text{bel}_0)$. The optimal policy, π^* , is defined as in Equation (10):

$$\pi^* : V^{\pi^*}(\text{bel}) \geq V^\pi(\text{bel}), \forall \pi, \forall \text{bel} \quad (10)$$

For a simple problem with two states, s_1 and s_2 , we can represent a continuous belief state in one dimension and the value of being in a state in another dimension. The value of taking an action in a state is linear with respect to the belief state. Hence, we represent this value as a vector and call it an α vector. With multiple actions of varying values over the belief space, we have multiple α vectors as illustrated in Figure 9(a). Given a belief state b , the best action is the one with the highest value. In Figure 9(b), we see that the optimal policy is the set of actions of with the best value in the belief space. The optimal value function, V^* , is constructed by taking a combination of these α vectors, while getting rid of parts that are *dominated* by better values. In particular, V_h^* for the h^{th} iteration is obtained from the set of α vectors generated at that iteration by selecting the best ones for each belief state using $V_h^* = \max_{\alpha} \text{bel} \cdot \alpha$. With a continuous belief space, there are uncountably many belief states to check for a corresponding α vector. Fortunately, V^* is piecewise linear and convex (PWLC) [SS73] for finite horizon. Therefore, algorithms that represent a belief state only need to identify a finite number of belief states that have distinct optimal α vectors, because only a few of the possible ones will contribute to V^* .

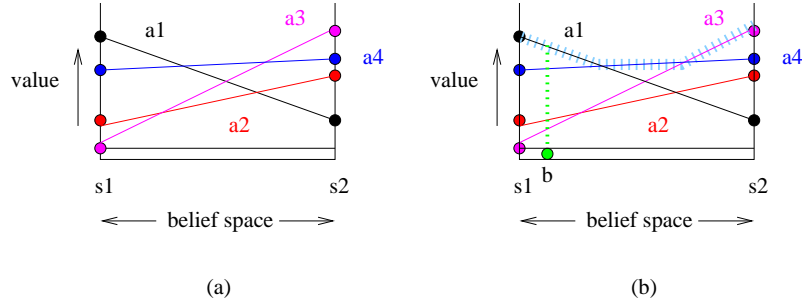


Figure 9: (a) Value functions in the belief space for multiple actions. (b) Computing the best action – one with the highest value – given belief state b .

The uncertainty inherent in POMDPs introduces three main sources of intractability. Two of them relate to an offline computational problem for α vectors and the third one relates to an online problem for monitoring the continuous belief state. We discuss each of these in turn and various state-of-the-art exact and approximate solutions to them.

In Figure 9, we carried out a one-step value computation for a problem with two states. We saw that the size of an α vector is the size of the state space, i.e., the size of the α vectors is exponential in the number of state variables (in a factored representation). In addition, the number of the α vectors imposes a computational bottleneck because there is one per conditional plan and they grow exponentially over the horizon. As this process repeats to the horizon, h , V_h^* can get complicated. Classic POMDP solutions provide exact solutions for finite horizons that address these two offline computational issues. Belief-state policies are generally constructed using dynamic programming (DP) algorithms that produce an optimal value function that implicitly associates an optimal action with each belief state. The DP formulation of the value function is state in Equation (11), where $r_a(\text{bel}) = \sum_s \text{bel}(s) R(s, a)$, $\text{Pr}(o|\text{bel}, a) = \sum_s \text{bel}(s) \text{Pr}(o|s, a)$, and bel_o^a is the posterior belief state upon taking a and observing o .

$$V^{k+1}(\text{bel}) = \max_a r_a(\text{bel}) + \gamma \sum_{o \in O} \text{Pr}(o|\text{bel}, a) V^k(\text{bel}_o^a) \quad (11)$$

The simplest exact solution is to exhaustively enumerate all the α vectors. This algorithm requires keeping track of $O(|A||\alpha(t-1)|^{|Z|})$ vectors at the t^{th} iteration. Hence, Monahan’s algorithm [Mon82] prunes the set of α vectors at each iteration by eliminating all the ones that are completely dominated before generating more at the next stage. Rather than pruning, Cheng’s and Littman’s algorithms only generate α vectors that guarantee value improvement. Cheng’s linear support algorithm [Che88] incrementally generates α vectors only if it has points in the belief space that “witnesses” its contribution to V^* . These witness points can be found by searching the belief space for the intersections of α vectors because those are the places with the most value improvement. However, this search might take exponential time because there is an exponential number of possible vertices in the belief space. By reformulating $V^{k+1}(bel)$ in terms of the Q functions, Littman’s witness algorithm [Lit94] searches through the observation space for witness points to find the set of α vectors for Q_a^h (which is also PWLC), and hence V_h^* . As a result, the search time of the algorithm reduces to $O(|O||\alpha(h-1)|)$. Zhang’s incremental pruning [CLZ97] performs pruning by decomposing $Q_a^h(bel)$ even further into $Q_{a,o}^h(bel)$ and $Q_a^h(bel)$ so that pruning the set of α vectors for these two separately results constant time improvements over pruning the original set of α vectors for $Q_a^h(bel)$.

These classic solutions compute the optimal policy over a finite horizon. It has been shown that this problem is EXP-hard in both $|S|$ and h [PT87], and these techniques have only been demonstrated on toy problems. In POMDPs with infinite horizons, the objective is to solve for an ϵ -optimal policy. However, even this problem has been shown to be undecidable [MCH03]. Approximations are necessary to apply POMDPs to large problems.

One class of solutions is to use methods with a finite “memory”. Rather than iteratively computing the optimal value function at each stage, we could iteratively search in the policy space for the optimal policy. For this, *finite state controllers* (FSCs) can be used to represent history-based policies, and several exact and approximate algorithms can be used to construct good FSCs [Han98, MPKK99, PB04]. An FSC is a directed graph $\pi = \langle \mathcal{N}, \mathcal{E} \rangle$ where each node $n \in \mathcal{N}$ is labeled by an action $a \in A$ and each edge $e \in \mathcal{E}$ is labeled by an observation $o \in O$. FSCs are a cyclic version of a conditional plan, so that each action node has an outgoing edge for each possible observation. A deterministic FSC policy $\pi = \langle \alpha, \beta \rangle$ where α is the action strategy of a node $\alpha(n) = a$ and β is the observation strategy of the successor node $\beta(n, o) = n'$. The value of an FSC π is defined in Equation (12).

$$V^\pi(n, s) = R(s, \alpha(n)) + \gamma \sum_o Pr(s'|s, \alpha(n)) Pr(o|s', \alpha(n)) V^\pi(\beta(n, o), s') \quad (12)$$

Extending to belief space to get $V(bel)$, the value of a belief state is the node with the highest value, $V(bel) = \max_n V(n, bel)$, where the value is taken with respect to the probabilities of the belief, $V(n, bel) = \sum_s bel(s) V(n, s)$. Then π^* has the value function in Equation (13).

$$V^*(bel) = \max_a R(bel, a) + \gamma \sum_o Pr(o|bel, a) V(bel_o^a) \quad (13)$$

Policy iteration (PI) [Han98] for FSCs starts with arbitrarily generating a initial FSC, π_0 , and incrementally improving on it by changing the graph structure until the policy converges optimally. Gradient ascent [MPKK99] is a solution technique for computing stochastic FSCs. A stochastic FSC is an FSC with stochastic action and observation strategies that are defined by distributions $\alpha(n, a) = Pr(a|n)$ and $\beta(n, o, n') = Pr(n'|n, o)$ respectively. An objective function is used to minimize the error over trajectories of the FSC at each iteration. Poupart & Boutilier [PB04] introduced an approximate algorithm called *bounded policy iteration* (BPI) which builds on Hansen’s algorithm but it bounds the size of the stochastic FSC at each iteration.

Finally, we turn to the third source of intractability – *belief state monitoring*. As mentioned above, if we do not want to maintain histories, we can alternately maintain a summary of the history via a belief state. Under this representation, $\pi : bel \rightarrow a$, and the system is required to update the belief *online* upon every new action and observation. This can often impose a severe online computational burden if $|S|$ is large. Belief state monitoring can often be made tractable if the system dynamics and observation model can be represented concisely using, say, a DBN [BK98] (cf. Section 2.1’s discussion on DBN approximate inference). Various belief state compression algorithms [RG03, PB03] have also been explored in making the problem more tractable. These algorithms exploit independence and sparsity in the belief space.

By making belief state monitoring more tractable, POMDP solution techniques that use a belief state representation are also made more tractable.

2.4 Inverse Reinforcement Learning

The problem of inverse reinforcement learning (IRL) is that there is an agent acting in a state space, S , by optimizing its reward function, R . However, the agent’s reward function is unknown and the agent only receives samples of rewards in various states. The goal of IRL is to determine an R that explains the observed behaviour, π^* (often assumed to be optimal). We are only aware of three AI approaches to IRL [NR00, AN04, CKO01]. We discuss each of these in turn.

Let MDP\(\mathbb{R}) be an MDP model with an unknown reward function. The first paper [NR00] presents three linear programming formulations under varying settings. Given a finite S in a known MDP\(\mathbb{R}) and a completely observed π^* , the first algorithm finds a set of R for the given model where the observed π^* is the optimal policy of the MDP with R . Using the Bellman equation (Equation (7)), the solutions of R satisfies the condition that: $(P_{a^*} - P_a)(I - \gamma P_{a^*})^{-1}R \geq 0$, where P are the transition matrices, $\pi^*(s) = a^*$, and γ is a discount factor. To prefer solutions whose policies are more similar to π^* , the objective function penalizes deviations from π^* by maximizing the difference between the quality of the best action, $Q^\pi(s, a^*)$, and the quality of the second best action, $\max_{a \in A \setminus a^*} Q^\pi(s, a)$, summed over all $s \in S$. In addition, to bias towards solutions with smaller rewards for simplicity, a penalty term of $-\lambda \|R\|_1$ is added to the maximization, where λ balances between the two goals.

The second algorithm assumes the same setting but the model has an infinite state space, i.e., $S = \mathbb{R}^n$, and the reward function is now $R : \mathbb{R}^n \rightarrow \mathbb{R}$, where $R(s) = \sum_{i=1}^d w_i \phi_i(s)$, for d known and bounded basis functions ϕ_i ’s and unknown parameters w_i ’s. A similar optimization is defined over a subspace of S . Since π^* is assumed optimal, $E_{s' \sim P_{s a^*}}[V^\pi(s')] \geq E_{s' \sim P_{s a}}[V^\pi(s')]$, $\forall s \in S, \forall a \neq a^*$. So the objective maximizes the total value summed over all states while penalizing ones that violate the inequality.

The third algorithm changes the setting and is only given observed trajectories of π^* and the ability to estimate V^π for any π . The goal is to find R such that π^* maximizes $E_{s_0 \sim D}[V^{\pi^*}(s_0)]$, for some initial state distribution, D . $V^\pi(s_0)$ is decomposed via the basis functions of R and is estimated as $\hat{V}^\pi(s_0) = \sum_i^d w_i \hat{V}_i^\pi(s_0)$ where each $\hat{V}_i^\pi(s_0) = \frac{1}{m} \sum_m \sum_{k=0}^K \gamma^k \phi_i(s_{m,k})$ for m simulated (or observed in the case of π^*) trajectories of length K . The algorithm keeps a set of known policies $\{\pi_1, \dots, \pi_j\}$ and iteratively solves for the parameters of R by constraining $\hat{V}^{\pi^*}(s_0) \geq \hat{V}^{\pi_j}(s_0), \forall j$. With the new R , a new π_{j+1} is found and added to the set. The algorithm terminates when a “satisfactory” R is found.

The second paper [AN04] takes a similar incremental approach as above. In an MDP\(\mathbb{R}), the unknown reward function is again assumed the structure $R = w^T \phi$. The setting is apprenticeship learning, where the agent is the learner and it observes an expert’s behaviour, π_E , and estimates the expected value of the trajectory using π_E . The goal is to simulate the expert’s behaviour by finding a policy that is near-optimal to π_E , evaluated on the unknown R , and then recover R . Starting with an arbitrary policy π_0 , at each iteration $t > 0$, the algorithm alternates between optimizing the parameters of $\hat{R}_{(t)}$ between the expected values of using π_E and $\pi_{(t-1)}$, and solving for an optimal policy $\pi_{(t)}$ via an independent MDP. The algorithm terminates when optimizing $\hat{R}_{(t)}$ has ϵ impact on the new policy and outputs the set of policies found at each iteration. An optimal policy can then be manually or automatically selected.

The third paper [CKO01] formalizes the underlying sequential decision problem as a decision tree from an agent’s perspective. Decision nodes in the tree are the agent’s actions. Chance nodes are nature nodes with a probability distribution over successor nodes. Terminal nodes, n , are utility values, $U(n)$, for the agent. An agent’s strategy, s , maps decision nodes to successor nodes. The expected utility of a strategy, $EU(s)$, is the weighted average of the utility values at the terminal nodes, where the weights are the probabilities of reaching the terminal nodes given s . The optimal strategy, s^* , maximizes the agent’s expected utility. Now, the agent’s utility function U is a linear combination of subutilities, $U(n) = \sum_{j=1}^m \alpha_{n,j} u_j$, normalized in $[0, 1]^m$. The space of possible utility functions for a rational agent, U^* , is constrained by a polytope defined by $EU(s^*) \geq EU(s), \forall s$. However, there are exponentially many different strategies, so the authors derive an alternate set of constraints that relaxes some assumptions and constrain a new utility density, $U_C \supseteq U^*$. Unlike the two previous approaches where an arbitrarily utility function in U_C is selected, here the algorithm make use of the user’s actions to further derive linear constraints in the space, without committing to a particular utility function. First, they impose a prior probability density function, $p(u_1, \dots, u_m)$ over U_C to represent their belief of the user’s utility function.

The p distribution could be constructed based on previously learned data. Each user action is treated as evidence on which p is conditioned to obtain the posterior, $q(u_1, \dots, u_m)$. However, q is typically a complex function. So instead, they generate samples from q via a Markov Chain Monte Carlo sampling algorithm. Finally, q is used to reason and predict the agent’s actions. A recommendation application using this algorithm is described in Section 3.6.

2.5 Preference Elicitation

One of the key aspects in modeling users is to learn their preferences. The focus of preference elicitation (PE) is to choose queries that reveals one’s preferences over a set of outcomes, S . A commonly used query type is *standard gamble queries* [Fre86], in which the user is asked: for what value, pr , is he indifferent between the definite outcome, s_i , and a gamble with probability pr of getting the best outcome, s_\top , and $1 - pr$ for getting the worst outcome, s_\perp . A principled way in evaluating the usefulness of a query is to compute its *expected value of information* (EVOI) [CKP00, BZM03]. In other words, compute the utility of knowing a response to the query, in expectation to getting each possible response. (Note that this EVOI is myopic.) Note, also, individual preferences are often costly to acquire due to a large outcome space and the continuous set of queries for selection ($pr \in [0, 1]$).

In addition, elicited preference information is often incomplete and inaccurate. Nonetheless, decisions must be made. Due to having incomplete utility information, the principle of maximum expected utility (MEU) cannot be directly applied to choose optimal actions. Rather than committing to a single user utility function (as in some of the IRL work in Section 2.4 [NR00, AN04]), we turn to approaches that explicitly models the system’s uncertainty about the user’s utility function by modeling a *distribution* over the set of utility functions [CK00, Bou02] and acting in expectation of this distribution determines the value of the action. However, we restrain from discussing myopic approaches and focus the rest of this section on a POMDP formulation of PE [Bou02] and associated computational bottlenecks in the model.

With the set of outcomes, S , and $|S| = n$, a user’s utility function is $u : S \rightarrow [0, 1]$. Then the space of all utility functions is $U = [0, 1]^n$. The state space in the POMDP is U and the belief space is a density P over U . Note that the user’s underlying utility does not change over time, so there is no state transition in this model. The set of actions available to the system is a set of queries, $q \in Q$, and a finite set of decisions, $d \in D$. Each query is parameterized by probability $l \in [0, 1]$ used in the standard gamble question. Since a query might burden the user, each query has a fixed cost, $c(q)$. The elicitation process terminates when the system takes a decision, d , and a terminal reward of $EU(d, u) = \sum_{i \in S} Pr_d(s_i)u(s_i)$ is received. The observations are the finite set of responses, $r \in R$, provided by the user. In the case of standard gamble queries, possible responses are {yes,no}. The observation function is the response model of each query, $Pr(r_q|q, u)$, with respect to the current state. Future stages are discounted by γ and decisions are computed over an infinite horizon.

Just as in Equation (13), the optimal value function of this POMDP is formulated in Equation (14).

$$V^*(P) = \max_{a \in A} R(P, a) + \gamma \sum_{r \in R} Pr(r|P, a) V^*(P_r^a) \quad (14)$$

When $a = d$, $R(P, d) = EU(d, P)$ and the remaining summands are dropped. When $a = q_i$, $R(P, q_i) = c(q_i(l))$ and P_r^q is the posterior density updated using Bayes rule. However, due to the continuous nature of the state and action space, the belief state and the optimal value function must be approximated. The approximation represents P as a mixture of uniform distributions and its number of components is kept manageable by pruning periodically. Then V^* is approximated via a quadratic function approximator or a neural network, and computed for individual components.

Although the continuous nature of PE exemplifies the complexities of POMDPs, this model addresses the sequential nature of PE and shows that approximations can be used to solve the POMDP and to obtain good policies. In addition, further computational gains can be leveraged by using compact representations of the utility function and belief state.

3 Applications

In this section, we survey applications in user modeling for software customization. The survey is organized with respect to the modeling techniques employed in the application. Sections 3.1 to 3.3 discuss applications for run time software customization, while Sections 3.4 to 3.7 draw on applications that have some kind of human interaction.

3.1 Customization Applications Using Rules, Heuristics, and Plan Libraries

The need for software customization has been acknowledged in the user modeling (UM) community and largely applied to areas such as natural language understanding and generation [Car01, ZL01], intelligent tutoring systems (ITS) [CGV02], adaptive hypermedia [Bru01b], and interface agents [LMM94]. The objective is to design systems that make interaction easier for the user to complete his tasks. In an ITS, for example, the system’s objective is to act as a tutor and help the user (student) by observing his actions that reveal what he has learned and what he needs improvement on. Traditional methods in UM use hardcoded rules, heuristics, and plans to infer the user’s current goal(s). This is usually achieved via *plan recognition*, which refers to inferring the goal by recognizing a plan that leads to it.

We illustrate the idea behind of plan recognition by showing an example of a *recipe* that encodes how a user can be distracted.³ We sketch this recipe in Figure 10 for an automatic help system. A user can be distracted by browsing or pausing repeatedly. If he is browsing, his application usage actions could be switching between menus, or switching between windows. The loops in these nodes represent that these actions need to be repeated, perhaps using a rule with a hardcoded time threshold, in order for the user to be considered as being distracted. In this recipe, there is no way to recognize a distracted user if he is not performing one of the leaf node actions, or a user who performs a different combination of these actions. In general, plan recognition systems store a large library of such recipes in attempt to model different courses of actions for multiple goals.

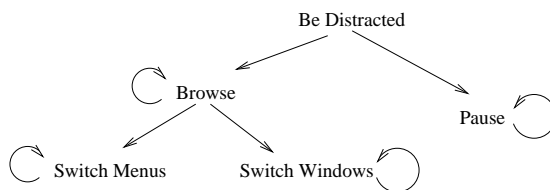


Figure 10: Interface agent example with plan recognition

This example illustrates a system that unobtrusively observes user actions. This approach of plan recognition is called *keyhole recognition*. When multiple recipes match the current course of action, the system must disambiguate from among the several possible goals. In such situations, the system may use strategies such as taking the one with the shortest path to the goal, or context specific information such as tracking the user’s current focus, or developing hierarchical plan libraries [LRS99, Wær97]. Note that maintaining multiple partial plans may seem attractive, but each step now implies keeping multiple candidates for multiple plans. A tradeoff must be made between including as many recipes as possible and the ability to prune away possible candidates at each step. Alternatively, the system may perform actions to help it help the user. This approach is called *intended recognition* [CPA81], and assumes a (currently unrealistic) cooperative environment in which the user collaborates on a *joint plan* with the agent.

Note that the determinism of system actions is a by-product of the rule-based and plan-based approaches because they do not allow enough flexibility in the models. We can bypass this by modeling uncertainty in our environment and thus choosing to act optimally with respect to the possible situations and how likely each of them are. Finally, due to relying on handcrafted, domain-specific plan libraries, even if they could be tweaked to work well in reported experiments, these approaches are expensive to generalize across domains and do not scale to real world systems [Car01].

³Being distracted is not a user goal per se, but it is important to recognize when a user gets side-tracked from his true goal.

However, a notable exception is the system Pearl [MP02], a personalized cognitive orthotics robot, which generates reminders based on a set of daily required activities and time constraints. The nature of the domain (daily routines) makes the system tractable. Reminders are formulated as temporal constraints rather than plans, so that the plans that get realized might change throughout the day based on observing the patient’s activities. The main heuristics used to evaluate when reminders are generated are the number of reminders, the timing (within specified constraints), and the spacing among reminders. In this system, the schedule of plan activities and patient preferences are manually updated by the patient’s caregiver.

Aside from exhibiting deviations from expected usage, another major problem in attempting to hard-code patterns is that users often change their intended goals without informing the agent (if they knew there was one). As a result, some preliminary work has begun investigating the use of machine learning techniques in classifying users into user groups/types (e.g., recommendation systems), predicting the user’s next action (e.g., [Mae97], [DH98], [KG00], [AZN98]), and learning to automatically build plan libraries (e.g., [Bau99]). The above systems do not reason about the utility of committing to various possible plans. They also do not reason about *when* to interrupt user (by helping or asking a question). Thus, these models do not consider the utility of its own interruption, nor does it model the future consequences of the interruption.

3.2 Customization Applications Using Probability and Utility Theory

In this section, we survey several UM applications that explicitly models uncertainty and the utility of system actions.

The focus of Fleming & Cohen [FC01] is to design systems that can engage in effective *mixed-initiative* collaboration with the user. Rather than passively waiting for appropriate opportunities to help or interrupt the user, they reason *when* the system should take initiative in requesting information. Under a cooperative setting, the model focuses on the decision of whether the system should ask the user for more information in order to further assist the user. The uncertainty modeled includes the probability of that the user understands the query, P_{UU} , the probability that the user has the knowledge to answer the query, P_{UK} , and the probability that the user could be made to understand the query, P_{UMU} . These estimates are assumed to be available from an underlying user model. Then, the system’s course of action due to the query has an expected utility, EU_{ask} , and the status quo has an expected utility, EU_{no_ask} . The situation where the user understands and can answer the question has the benefits $P_{UU}P_{UK}(EU_{ask} - EU_{no_ask})$ and the situation where the user does not understand but can be made to understand has the benefits $(1 - P_{UU}P_{UMU}P_{UK})(EU_{ask} - EU_{no_ask})$. Asking and engaging in a clarification dialogue have costs $C_{ask} + (1 - P_{UU})C_{clar}$. The system decides to ask the question if the total benefits outweighs its costs. The authors show the importance of the factors in this model by giving examples from sports scheduling and interactive machine translation. They mention extending the myopic expected utility calculation to account for the sequential nature of actions.

This work combines aspects of plan recognition, user modeling, and utility theory. However, the way the system makes decisions should be based on choosing the action with the highest expected utility. The scenarios in comparing benefits and costs are also not complete because there are 3 binary variables (UU,UK,UMU) making 8 outcomes and only two scenarios are considered. Furthermore, while a user model is assumed to provide the probability assessments, these probabilities are non-observable. Although the details depart from standard utility theory, the formulation considers one-step look ahead when a clarification dialogue is a worthwhile next-action. The paper also lacks an implementation.

An example of goal inference is exhibited by Brown, Santos Jr., & Banks [BJB98]. The application models a pilot’s landing process that can be done either automatically or manually, and when done manually, some system actions may be performed to help reduce the cognitive load of the user. They model the dependencies first using a “goal graph” decomposition which they convert to a BN after. The idea is to infer whether the user’s goal is to land automatically or manually, and whether the user wants his cognitive load reduced. Every second, the system makes a decision of whether to offer assistance to the user by computing the expected utility of each goal, based on the probability of doing an action, a , with respect to a set of observed user actions, E , and cognitive load, L , and the utility of an action with respect to a goal and the user’s skills and preferences, $U(G, a, U)$. In other words, $EU(G, U) = \sum_a Pr(a|E, L)U(G, a, U)$. The goal with the highest expected utility is thresholded against user defined values for autonomy and suggestion. The system completes the goal for the user if the expected utility

is above the autonomy threshold, suggests doing the automation if it is above the suggestion threshold, and does nothing otherwise. Although the authors mention that the system has been implemented, no evaluations were provided.

The authors omit discussion of converting between their goal graph representation to a BN. The decomposition of the model and the formulation of the decision rule are questionable in that the agent suggests the goal, *not* action, with the highest expected utility. The user’s actions should be treated as evidence, but the system is modeled to maintain a belief over actions based on some other evidence and cognitive load factors. It is also not clear whose actions these are, as the authors only mention that these are actions that achieve goals. The model should define the expected utility of a system action over the probability of the user having a particular goal, and then the action with the highest value can be compared against the various thresholds.

Next, we turn to a mobile application for interface customization. For a fixed application, an interface can be rendered in different ways depending on the constraints of various devices available. SUPPLE [GW04] is a system that defines a formal model of interface rendering as an optimization problem. The tuple $\langle I, D, T \rangle$ is the set of variables corresponding to interface requirements, device requirements, and traces of user’s usage patterns respectively. An interface is defined as a hierarchical layout of widgets and the design of the interface adopts an object-oriented programming perspective. I consists of information elements and interface constraints, while D consists of the set of available widgets, device constraints, the cost to manipulate state values via specific widgets, and the cost to navigate among widgets. Both I and D need to be defined at design time. T , on the other hand, keeps traces of the sets of elements the user works through and the changed values of these elements. This information needs to be updated at runtime, either in real time or over intervals. A cost function, $C(\phi, T)$, is defined for rendering ϕ and trace T in terms of navigation and manipulation costs of the entire T . In this sense, the cost of ϕ is defined deterministically according to old traces. To choose ϕ , the system uses a branch-and-bound constrained search that guarantees an optimal solution. The algorithm was reported to find the best rendering in between 0.2 to 13 seconds without performing a complete search. The system was able to generate different interfaces given different input mechanisms, screen sizes, and user traces.

This system formalizes different factors for interface customization succinctly as an optimization problem. As this problem was motivated for mobile devices, it is important to consider the size of the display in the customization process. In particular, the cost function is defined in terms of navigation and manipulation effort. The effort to manipulate the value of a widget is currently defined based on the particular widget and the intended value, but independently of the size of the screen display. Thus, a large monitor could end up having really tiny widgets even if there were no space limitations. However, the effort to manipulate small widgets could have a high cost and be generally discouraged. The cost of a rendering is defined with respect to available traces, which means they assume that past behaviour reflects future patterns. This is reasonable only if there are a large number of traces available for a particular user. At the same time, it is hard to avoid making this assumption since it is hard to predict user patterns accurately. If they were to predict future patterns, they could project from old traces and take the expected cost of ϕ with respect to the predicted traces. Collections of traces may also be used as priors or to determine user types. As traces accumulate for the existing user, a new rendering should be generated according to the updated trace information. One could periodically update the traces and re-render the interface, or alternatively reason explicitly about when the update should take place. The time taken to find an optimal rendering can be slow (13 seconds).

The last system we consider in this section is an interactive machine translation system called Transtype [LFL98] that provides word completions using probabilistic model. Given some text in the source language window, the user’s task is to type the translation in the target language window. At each time step (per letter typed), the system chooses a set of completion words to the user based on the words typed by the user and words in the source language. The system has an evaluator function that assigns a likelihood to each target word candidate based on a weighted linear combination of the language (trigram) model and the translation model (a probabilistic model that aligns the source and target language). The weight used is dependent on the context of the current text entry, so that words near the beginning of sentences would weigh the translation model more heavily while words at the end of sentences would weigh the language model more heavily. Candidate strings are searched in a small active lexicon first and then in the remaining lexicon upon failure. The system chooses a set of completion words based on the highest probabilities assigned by the evaluator function. A pilot user study was conducted in which the recorded

typing speeds were lengthened but the users felt the system helped them type faster. Based on the number of acceptances of suggestions made, the system allowed about 60% character savings in the study.

This is an interesting word prediction application that takes advantage of the available source text in a translation context. To speed up the search of possible matches, the system keeps a small active lexicon of 500 words apart from the remaining 380,000 words. In a probabilistic model, every word is a possible completion because it has non-zero probability. However, when composing the box of completion words, it was not mentioned whether there is a cut-off for displaying a maximum number of words to the user. From a usability aspect, there is also a trade-off in displaying more information to the user and the effort required of the user to search through all the suggestions. There is also a trade-off between adding more words at the end of a list and the time and distance required to select a suggestion that is further away from the current cursor position. Although the authors mention that the piloted users typed slower in the experiments, it was not mentioned how long each suggestion took to compose.

3.3 Customization Applications Using (Dynamic) Bayesian Networks and (Dynamic) Decision Networks

Many UM applications are beginning to acknowledge the value in modeling the user with BNs and DBNs. Software applications with user interaction can benefit from inferring the user's current goal or task, or predicting the user's future action or state [AZN98]. We omit discussing papers that propose models without implementations or evaluations. We begin surveying the Bayesian UM applications in ITS.

Conati & VanLehn [CV01] describe an ITS called SE-Coach that decides whether to prompt the student to self-explain a solution step or not via a scaffolding interface. A domain-specific BN is built from a set of physics exercises. Each problem contains the following nodes: goals, facts, rules, rule applications, possible self-explain prompts, and instruction steps. To track the user's focus, the sequence of solution steps to an exercise is masked, and are unfolded when either the student clicks on a step or it is inferred that the student has considered the step. Based on observing the student's reading time and previous system interaction, the system infers whether the student knows the particular facts and goals involved in a given step. These distributions represent the system's assessment of the student's domain knowledge. The system also infers whether the necessary rule application is taking place. Latency is used to infer whether the student is already self-explaining a step on his own or if the student is reading the instructions in the step. If the rule application probability is less than a pre-defined threshold τ , then the system suggests that the student read more carefully, or tailors the self-explain prompt with different informative browsers. After each exercise, the student model is saved and the inferred posteriors on rule nodes are used as priors in the next exercise. The only general information about the student in this user model is a variable η that represents the student's tendency to self-explain.

Although self-explanation is an effective pedagogical technique, not every student knows how to correctly and effectively self-explain. In particular, students who need extra help with a topic may not know why certain rules are used, even if the solution is presented to them. The variable η is fixed in the experiments and no discussion is provided as to how this value may be assessed. As part of the BN, it would be more interesting to infer a user's tendency to self-explain so that the system learns about the student's study habits. Using latency is a very loose measure of a student's reading time because he may be distracted. Different student behaviour is not discussed in this model. In particular, students are expected to use the system as intended in the design, but there is no mention of the possibility that a student may get distracted, switch to different exercises in the middle of a problem, get frustrated and quit, or get bored and quit. From this, we see that student with different behaviour and students who learn at different paces are not modeled. As a tutoring system, the adaptation should emphasize the student's study pace and problematic areas, rather than whether particular rule applications are used because those could be memorized by drills.

SE-Coach is embedded in a larger ITS called Andes. The decision-theoretic extension of Andes is called DT Tutor [MV00]. The goal of this project is to design a system that could replicate human tutor behaviour in selecting tutorial actions. The model is a DDN that encompasses domain facts, emotions (independence and morale), and domain rules. The system was implemented and simulated using exact and approximate inference. On average, DT tutor required 8 to 108 seconds for each action selection. The authors describe variations in the system tutorial selections and the resemblance to human selections.

It is interesting that DT Tutor considers the student's morale and independence in considering which

tutorial gets selected. In a human-human scenario, the tutor would infer the student’s morale by actions as well as visual feedback (facial and body language). However, the paper only mentions that the variables for moral and independence are influenced by tutor and student actions, without further details to explain what kinds of actions and how they might influence the variables. Also, even though the authors mentioned that real-time responses were necessary, the fastest time in the simulation still takes too long (8 seconds).

Zhou & Conati [ZC03] develop a probabilistic representation that models the cognitive structure of emotions in an *edutainment* system called Prime Climb. In Prime Climb, the student exercises factorization knowledge while climbing up a mountain. The student completes the quest after achieving each intermediate subtask posed by a pedagogical agent. Since student emotions are largely dependent on their personality traits and whether their goals are satisfied, the authors develop a DBN that infers student emotions (and domain knowledge) by monitoring student interaction patterns and whether they succeed in the quest. The focus of the paper is to learn a BN model that assesses student emotions online based on goals being satisfied. The BN is divided into groups of variables corresponding to the student’s personality, student goals, interaction patterns, and specific actions during the game. Using psychologically motivated personality variables and game-specific interaction pattern variables, the authors elicited the values of goal variables from focus groups. A field study was conducted to collect data to identify the structure of the BN that best explains the data. The speed for online inference was not provided.

Because the paper focuses on learning the BN to infer emotions, the overall role of the pedagogical agent is ignored. It is not clear in the current version of Prime Climb how much control the agent has in the system. For example, can the agent provide positive and negative feedback to the user? Can the agent choose the factorization exercise based on history? Can the agent help the user when it notices the user is stuck? Can the agent provide incentive for the user to concentrate on solving the exercises? Furthermore, can the agent challenge the user by increasing the level of difficulty? The next step would be to adapt the agent’s actions based on its interaction with the user and its assessment of the user’s emotional state.

Turning to automated assistance in general software use, we first discuss the Lumière system [HBH⁺98]. The authors present an ID showing the utility of automated assistance, influenced by factors including cost of assistance, user needs, user goals, user skills, task history, and application context. The rest of the paper focuses on developing a model to infer a user’s needs and goals. To determine heuristics for this inference, the authors conducted a “Wizard of Oz” study using Microsoft Excel and identified usage patterns through observing various users to infer when users need help and what specific task they were working on. They develop an elaborate DBN to recognize 40 goals in Excel using these observed patterns. Examples of goals are defining a custom chart and formatting the document. A distribution over these goals is maintained. Distributions of user profiles were also created and updated in the application, but no details were given as to what kinds of information was kept in the profiles. To bridge the gap between these action patterns and system events, they presented an intermediary language that maps events into model variables. At every time step, the system decides to provide automated help if the inferred probability of the user needing help is greater than a specified threshold. The authors considered different definitions of a time step and a cost-benefits decision policy, but comparisons were not provided.

Lumière was used to demonstrate the tractability and ideas for user modeling in a real-world application. The model presented is based on system variables only. Although a user’s competency profile and history can influence the probability the user needs help and the current goal, but it is not clear what kind of information is in this profile and the level of granularity of this information. Also, some of the decision parameters – the cost of assistance, the difficulty of a task, the threshold for providing help – are the same for all the users. Again, it would be interesting to infer user *types* based on system interaction. In addition, the authors mention extending the model to account for different modes of interaction, including observations from cameras and engaging into dialogues with users. They also mentioned developing a decision policy that accounts for the value and costs of automated assistance. In this way, they can incorporate system queries as one of the automated actions and explore active learning. Furthermore, the value of help may be adjusted depending on the user’s history, or traded off, depending on predicted user actions.

In general, automated assistance can be viewed as the system taking initiative to help the user, rather than viewing the system as a passive tool under complete user control. The area of *mixed-initiative* interaction formalizes initiative patterns between the system and its user(s). Horvitz [Hor99] identified twelve principles for designing mixed-initiative user interfaces. Some of the principles highlighted AI aspects, such as modeling explicit uncertainty and using a cost-benefits analysis in the decision rule. On

the other hand, some of the principles centered on HCI aspects, such as designing social agents and allowing the user to directly invoke and terminate automated assistance. An email-calendar testbed called LookOut was used to demonstrate these principles. Using the email content, the system applies a linear support vector machine to classify text into either having the goal of scheduling a meeting or not. With this belief of the user goal, $Pr(G)$, the system decides from among the actions of scheduling an appointment mentioned in the text, popping up a dialog to ask the user whether an appointment should be scheduled, or do nothing. Each action has two utility values associated with it corresponding to the cases given the goal is true, $u(A|G)$, and given the goal is false, $u(A|\bar{G})$. The optimal decision is to take the action with the highest expected utility. No evaluation was provided.

A neat trick was used to speed up the computation of real-time responses – thresholds for dialog and interruption are precomputed as the intersection points of these utilities and the action is chosen as the one with the highest expected utility at $Pr(G)$. Note that the representation of these utilities with respect to the probability of having a certain goal is directly analogous to the representation of α vectors in a POMDP. Also recall that the intersection points of the α vectors with the highest value also “thresholded” which optimal action to take with respect to the corresponding belief state. In theory, the action utilities can be extended to account for variance across different users so that different users can have different interruption thresholds. The current decision policy is decision-theoretic, but myopic, because it considers only the value of an action at the current time. For example, it can be better to engage in a dialogue earlier than to perform automated help later. But the system does not account for this.

3.4 User Interaction Applications Using Markov Decision Processes

Two notable problems in the models we reviewed from Section 3.3 are a lack of explicit user features and the use of myopic policies. In most cases, the models have neglected to tease apart the variables that are user dependent. In other words, in calculating the expected utility of system actions, the system does not consider what kind of user it is interacting with. Note that a user *feature* is different from a *goal*. A user feature is a trait, such as independence, that causes different observed behavioural pattern. Thus, these models do not tailor to different user types and make it difficult to extrapolate the user features that are relevant in other applications. Most of the decision policies discussed in these applications have handcrafted policies that compare an expected utility value to a predefined threshold. However, user preferences often are best expressed over trajectories of system behaviour, and these will vary over time. In this section, we turn to MDPs for a sequential account of decision making. Due to a lack of customization applications with MDPs, our discussion will focus on applications in domains involving user interaction.

Gorniak & Poole [GP00a, GP00b] propose an MDP that models a user for generic software using both system configuration states, S_C , and some internal user states, S_U . In this way, $S : S_C \times S_U$. Actions are system actions that change from one configuration to another and state transitions are probability distributions over world states. Finally, rewards are assigned to every state and action pair, $R : S \times A \rightarrow \mathbb{R}$. They assume that the user is *optimizing* (and hence, rational). If the states were fully observable, then the model would be solvable by standard MDP solutions. However, since the state space includes user states that are unobservable, the goal of this work is to approximate the MDP state space using observed state space, S_O (e.g., interface configuration), to predict future user actions.

Now, given observed states, S_O , an offline explicit state identification algorithm (OFESI, [GP00a]) infers the MDP state space based on state-action patterns. The goal of this algorithm is to introduce new states when they can explain a large fraction of observed application use. Based on a state-splitting algorithm [McC96], states are split in a binary and hierarchical fashion and the split only occurs when the information gained and the number of actions accounted for are greater than some thresholds. Substates are chosen via stochastic local search through the state space for some parameterized number of iterations. Each action corresponds to a search move, so that with probability p the move that maximizes information gain will be chosen, and with probability $1 - p$ a random move is selected.

At run-time, an online implicit state identification algorithm (ONISI, [GP00b]) predicts the user’s next action. Observed states, S_O , are a history of state-action pairs over a fixed-sized window. A ranking of actions per state at each time is computed as a weighted sum of the frequency of action in the state and the average length of the k longest sequences matching the immediate history in all recorded histories. These rankings are used to predict the next action in time. Experiments showed good predictive performance.

The model distinguishes between system states and user states. Although this work does not learn explicit user features, the algorithms can take a generic application, extrapolate the observed state space and user actions, and infer the underlying state space. Because user features are modeled explicitly, this framework can learn user preferences (as a function of those user features) more readily than frameworks which only model the user as usage patterns. In general, learning underlying user states would be hard to validate. Ideally, one could evaluate the effectiveness of OFESI by comparing its predictive performance with and without state splitting. Alternatively, the MDP user model could be generalized to a POMDP, since the user states are unobservable.

Jameson et al. [GHMR⁺01] focus on the decision of presenting instructions in a sequence of steps versus in a bundle. Instructions presented in *stepwise* mode gives one instruction at a time and waits for user acknowledgment before proceeding to the next instruction. Instructions presented in *bundled* mode gives all the instructions together. This process attempts to make it easy on the user’s working memory and is initially modeled as a BN. User experiments were conducted to gather real data to use for the CPTs, with variables such as presence of other distractions, execution time for the instructions, and error rate in following the instructions. The BN model is extended to an ID and then a more general MDP. The MDP model breaks down the decision of choosing a presentation mode for a set of instructions into smaller decisions of choosing a presentation mode for a subset of instructions and then repeating the process thereafter. Value iteration is used to solve for the optimal policy.

This model is based purely on system state variables, so the user’s response toward an initial presentation mode is not taken into consideration in the following configurations. The model also assumes that the presence of distraction is fully observable, which is likely to be unrealistic. To relax this assumption, the model can be extended as a POMDP where the system keeps a distribution over different levels of distraction and uses observations update the distribution.

In a later application, Bohnenberger & Jameson [BJ01] model an airport shopping navigation system using an MDP. There are 16 location sensors and 2 departure gates in a gridded map. The user may need to purchase a gift before leaving the airport by a certain time, and the user has an associated reliability factor for following directions correctly. Each state corresponds to whether a location sensor sensed that a gift has been bought or not. There are two additional goal states, totaling 34 states. The system displays directions between these states in either speech mode or map mode. Speech mode is assumed to result in faster movement while map mode is assumed to increase the likelihood of gift purchase. Stochastic transitions are defined among states, and the cost of each transition is a function of the time it would take the user to move from one state to the other, taking into consideration pedestrian traffic in shopping areas. The terminal reward received at one of the goal states is 0 if no gift is purchased and otherwise, the reward is some function of gift satisfaction and early arrival. Value iteration was used to determine the policy of choosing an action with the next location and best presentation mode. The authors show the system behaviour when varying user reliability (2-valued) and the importance of having a gift purchase (4-valued). Users with low reliability are steered away from high traffic areas, unless having a gift is extremely important. If the gift has little or no importance, the user is steered toward the terminal gate and instructions are given in map mode.

This paper demonstrated the use of MDPs while taking user specific input (user reliability and importance of gift purchase) into account. Although not mentioned, these two variables are assumed to be given and are probably embedded into the transition function and reward function respectively. However, the system does not model any changes in the environment (e.g., sudden increase in the traffic of people) and does not interact with the user or checks to see if the user followed the suggested instructions.

An MDP model was proposed for adapting recommendations in a recommendation system by Shani et al. [SBH02]. In general, recommenders are used to suggest items that are likely to be bought by the user based on what a group of similar users found interesting. Each time the user selects an item, the system can potentially make one or several recommendations, and the user can select an item from the list or browse elsewhere. The repeated scenario illustrates the sequentiality of the problem – the system selects recommendations based on the user’s past responses. Furthermore, the system might make suggestions that have lower immediate reward but greater long term utility or more value of information. For example, the system can choose to suggest an item that has lower profit (immediate) but it belongs to a series or a collection so the user is more likely to purchase multiple items. Note that the objective formulated in this model is of a different nature than that of software customization because it does not act in the best interest of the user.

In Shani et al.’s model, the states consist of a sequence of the three most recently bought items, $s = \langle x_1, x_2, x_3 \rangle$. Specific features about the items or the users were not modeled. The action is one of the possible items to suggest to the user. The transition function, $T(\langle x_1, x_2, x_3 \rangle, x', \langle x_2, x_3, x' \rangle)$, is proportional to the learned transition function of a corresponding predictive model of usage patterns, which is based on the same state definition and learning techniques such as skipping, clustering, and mixtures of components. The reward of arriving at each state, $R(\langle x_1, x_2, x_3 \rangle)$, is the profit earned by x_3 . Policy iteration was used to solve the MDP. An approximation of using immediate rewards rather than expected value for states that were not often visited by users was used to exploit sparsity in the data. Thus, the system chooses the action of recommending an item that has the highest expected value. The authors commented that computing a recommendation list of k items is too expensive, and they propose the simple augmentation by taking k items corresponding to the k best actions. They carried out a systematic evaluation of different versions of the predictive model that used different learning techniques (skipping, clustering, mixture of components). The evaluation of the recommendation system as a whole with real users was not presented in the paper, although it has been deployed.

This paper presents a novel formulation for recommendation systems as MDPs. Since it is in an MDP model, the choice of system actions considers trade-offs in the long term. The system could potentially make better informed decisions if the states modeled specific user features, such as features about items the user selected or browsed through. The model could also be extended to make suggestions of sets of items at a time.

3.5 User Interaction Applications Using Partially Observable Markov Decision Processes

Due to complexity bottlenecks, POMDPs have only been recently applied beyond toy problems. Here, we discuss an application with approximately 10 state variables, actions, and observations each.

Roy et al. [RPT01] recognize the usefulness of an MDP approach for generating policies in dialogue management. However, an MDP would still have problems with one-sided initiation on the user’s side and not being able to model the noise in human utterances. Hence, they propose a POMDP model for dialogue management and applied it to a robot in a nursing home. In particular, the robot infers user goals, such as wanting to know the time of day or tomorrow’s weather forecast, based on observations of user utterances. The specific states in the model are user actions, such as requesting for information, or subgoals, such as wanting weather information, that lead to the larger goals. The structure of the state model has a starting state that recognizes when a request from the user has begun, a set of possible subgoals and requests, and a terminating state when the request is completed. Available to the robot is a set of speech actions, such as asking information to be repeated or saying hello. The transition function is defined as usual, $T(s, a, s')$, while the reward function is $R : S \times A \times O \rightarrow \mathbb{R}$ – defined for system states, agent actions, as well as user observations. To solve the POMDP, they convert the model into an MDP by compressing the full belief state, $bel(s)$, into the most likely state, $argmax_s bel(s)$, and the entropy of the belief state, $H(bel(s)) = -\sum_s bel(s) \log bel(s)$. These two features are used as the (fully observable) state in the simpler MDP. Then value iteration is used to solve the simpler MDP, over infinite, discounted horizon.

Although the authors state it as one of the motivations for using a POMDP model, mixed-initiative interaction is still not present in system. Also, the POMDP was used to model uncertainty in user goals – which are unobservable. But the way the model is solved summarizes uncertainty and treats the states as fully observable. Although the details of the reward structure was not presented, different goal states seem to have rewards associated with them. Also, there is no user specific information into model. For example, the robot does not learn about different user’s usage patterns or vary costs of questioning for different users.

3.6 User Interaction Applications Using Inverse Reinforcement Learning

Due to the limited available works on IRL, this paper discusses a recommendation system that learns the user’s utility function. Chajewska et al. [CKO01] apply an IRL mechanism to learning the user’s utility function in the context of a recommendation system. The domain has two players (using game theory terminology), one being the user – who is an agent acting to maximize his own expected utility, and the

other being a software agent – who is also a rational agent but is aware of the user and can adapt its action according to the user’s response. Under a game perspective, the agent’s objective is *not* to act on the user’s behalf, but rather, to maximize system profit (like Shani et al. [SBH02]).

The recommendation is formulated as a decision tree, rooted with a user action (sign on to system or not), alternating with an agent action (offer discount or not), followed by a user action (buy product or not). If the last user action is to purchase a product, the tree is further split by a chance node of whether the product is enjoyable or not. Each leaf node has a utility value associated with it, defined as a linear combination of general subutilities, e.g., the utility of getting a bargain or enjoying the purchase. One recommendation session is represented by such a decision tree. The system models a distribution over the space of possible utilities. The details for constraining this utility space and sampling from the distribution is described in Section 2.4. With this information, the system tries to predict whether the user will purchase a product or not, and decides whether or not to offer a discount of the product. However, since the agent is not certain of the user’s utility, it takes a finite chain of samples of the distribution and uses each of these samples as initial states of multiple trees. The authors propose to view the problem as an *imperfect game* so that the system maintains an information set of many game trees and acts with respect to the expected value of the actions under this belief. In a series of experiments, the authors showed that the error between the true utility function and the mean estimate of the posterior distribution decreased drastically over repeated games. There were no results reporting computational performance.

This paper offers a general account of acting under utility uncertainty. It highlights many of the computational issues and proposes approximate solutions with quickly decreasing error rates.

3.7 User Interaction Applications Using Preference Elicitation

This section discusses a recommendation system with preference elicitation. Nguyen & Haddawy [NH98] apply a decision-theoretic formulation of preferences in movie recommendation domain. They suggest that long-term user preferences be formulated as part of the user’s value function, while short term preferences be formulated as hard constraints. The value function is a linear combination of duration, rating, and collectively, movie director, cast, and genre. The system has an initial database of movie preferences specified according to the value function. A new user is given a questionnaire and asked to fill in (binary) preferences for running time, rating, and movies he liked and disliked. The new user can select prototypical functions to represent his subvalue function for running time and rating. The list of sample movies are used as training instances in C5.0 to get user specific classifiers for the remaining variables. With this information, a partial preference structure is learned about the new user, and this structure is matched against others in the database in order to find the closest match to supplement the partially elicited information. The similarity between two preference structures is defined as the average count of the number of items that are ranked the same. The system would then recommend movies based on all the available information about the user, but the experiments did not discuss this. A database of 10 users’ preferences were collected. For evaluation, one of the existing user’s preferences were used as a “new” user. The system was able to match the new user with the true user in the database.

It is not clear what the authors mean by short term preferences because they do not provide examples of them. In particular, one might model movie duration, director, and genre as short term preferences, since they can change depending on what the user is in the mood for. There was no discussion as to how much data is needed in the database for good performance, or how much data needs to be elicited from new users, or whether more information can or needs to be elicited from users during application use. The paper mentions that a new user can select from several prototype value functions, but there is no example to show how this is done. The similarity metric between preference structures does not consider the relative importance of the different variables in the value function. It is also problematic when there are multiple equally good matches for a new user. Also, under this elicitation of preferences, users may get “stuck” in the same category. For example, if a user is only familiar with directors he likes and possibly a few directors that he dislikes, then the similarity metric will match him to someone else’s structure in the database. But if that other structure does not include new directors, then this new user will not have a chance to explore other kinds of movies.

4 Summary of Open Problems in User Modeling

After reviewing the state-of-the-art applications in Section 3, we summarize the difficulties and problems found into three categories: modeling, evaluation, and learning model parameters.

4.1 Issues Across Models

First of all, with regard to the quality of the proposed models, many of the surveyed applications traded off modeling the sequential nature of the problem using a more general model, with a simpler model and a myopic policy. Most of the systems also do not model the user explicitly, but restrict the state space to modeling the system environment only. Therefore, system utilities are not taken in expectation of the beliefs over user features or user types. As a side-effect of this choice, the system does not infer features about the user. Hence, it makes eliciting and learning about the user's preferences implicit.

The more complex the modeling technique, the smaller the size of the demonstrated application. The only truly large scale application is Lumière [HBH⁺98], while the rest are modeled on small problems. This suggests that the level of abstraction chosen in their DBN model allowed the application to scale well. Furthermore, this suggests that exploiting structural representation for MDP and POMDP models would be promising.

At the heart of user modeling is the learning of specific feature values for our users. However, most models do not attempt to learn this kind of information, and therefore, make it difficult to adapt the system to relevant user features, to differentiate amongst different user types, or to re-apply the user model in other applications. Generalizing user models is one of the areas we would like to explore. In addition, we would like to investigate leveraging information about a single user to speed up learning of the entire user group. In Section 5 and 6.1, we sketch models of interest to help us address these modeling issues.

4.2 Evaluation Difficulties

Due to inherent subjectivity, user evaluation of software has been an outstanding problem. Because this is not the focus of our work, we do not dwell on the issues here. However, note that many of the applications in our survey attempted to provide objective evaluation metrics – such as speed performance, accuracy in prediction, average percentage of suggestions accepted – to either parts of the model (e.g., [SBH02]) or the entire proposed model (e.g., [LFL98]). To provide at least some level of validity, we intend to evaluate our models in two ways: (i) via simulations and (ii) pilot user studies. We discuss these possibilities in Sections 5.3 and 6.2.

4.3 Learning Model Parameters

Under the decision-theoretic framework, there are several places that could use learning. The first is for learning the underlying model structure. For example, with empirical data, one could learn the strength of influences among state variables. The second is for learning the transition function. With empirical data of state and action sequences, transition models across states can be estimated. If we used a POMDP, then the third is for learning the observation function, using the same technique as learning the transition function. The difficulty in these three problems is that one must gather ample data to detect significance between candidates, otherwise the results would be somewhat arbitrary. Several applications in our survey have already explored these avenues (e.g., [AZN98, ZC03, GHMR⁺01, SBH02]). The fourth is for learning the utility function. In addition to the sparse data problem, the truly difficult part here is that people do not know their own utility function. In other words, we often do not know how to compute or express our preferences to exact numerical accuracy needed in the models. In Section 6.3, we discuss ways of learning the user's utility function.

5 Our Proposed User Model for a Typing Assistant

Consider formulating the software customization problem in a state diagram. At a given point, the user is at a state, S , trying to arrive at some other state, T . Figure 11 illustrates a simple example. In this figure, there are two states with positive rewards (drawn with plus signs) and one with negative rewards (drawn with minus signs). The shaded nodes highlight a typical usage path to T . This path is not necessarily the best way of getting to T from S , but it represents a typical sequence of actions for the user (possibly because it is either the only path known to the user or the only path available to the user). Other states are possible states already in the system, but they are currently unreachable from S due to some design decisions.

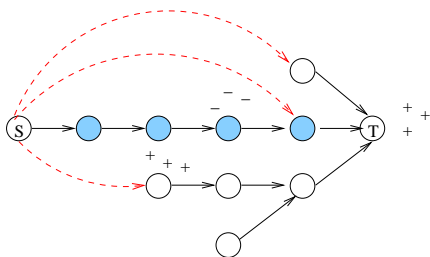


Figure 11: State diagram showing a typical action path from state (S) to state (T), with suggested paths drawn in (red) dashed lines.

The role of a customizing agent is to help the user go from S to G in a more effective way via a path with higher payoff. This can be done by making the user aware of existing paths, or creating new paths (i.e., linking to unreachable states). In a customization application, a path with higher payoff may correspond to a shorter path (e.g., the agent performs an automated task) or going through a more preferred state (e.g., the agent configures the interface based on known user preferences). Under the view of having an underlying assistant agent, there are two actors involved – the user and the customizing agent. Both actors travel in the same state space, S , but the transition function, T , is now affected by the joint actions of the user and agent. From the view of a POMDP, user actions can be modeled as observations, O , while agent actions can remain as system actions, A , in the model whose expected utility is maximized in the decision making process. Furthermore, if the state space in the problem includes system customizable states as well as user states (e.g., psychological features about the user) such that $S = S_C \times S_U$, then the states are partially observable and the model would also need to maintain a belief state.

The major obstacle in this problem is that we do not have a clear definition of the reward function, R , but the agent must nonetheless act as optimally as possible. Our goal is to learn R while earning significant actual rewards as early as possible. An inevitable issue in software customization is keeping up with changes and detecting inconsistencies in user goals, observed or elicited user skills, and observed or elicited preferences. We do not dwell on this latter point, although it is an important issue to investigate.

5.1 Typing Assistant Model

To demonstrate decision-theoretic modeling for software customization, we carved out the text completion task that is available in many existing text editors for writing documents or electronic mail. Such a tool could play an especially important role in aiding users with physical or cognitive impairment to maintain communication with others. From the system’s point of view, the user is typing an English text document and the agent can choose whether to suggest a set of completion words to the user or not. In addition to the system state, the factors at play are the user’s attitude toward the writing task as well as the writing environment (i.e., the software). The current model we present here is a simplified model for illustrative purposes and to test out the main ideas of our work. The model expressed as a DBN is shown in Figure 12.

The state variables in our model consists of a system variable and user variables. Our system variable is:

- $SDR(1, 2, 3, 4, 5)$: value of the slider bar – an explicit indicator of the level to which the user wants adaptive agent active

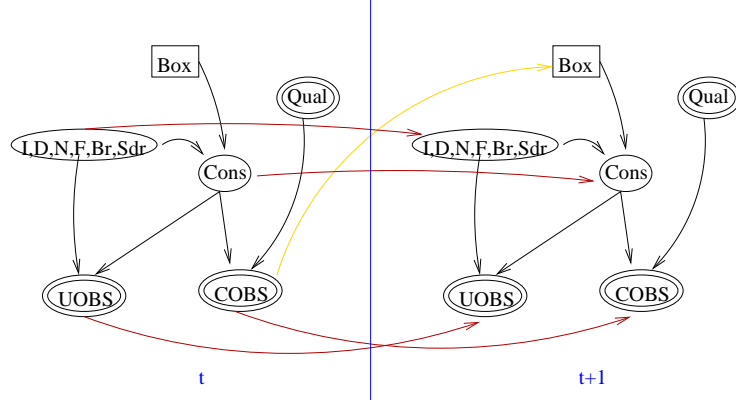


Figure 12: Event-based model (in DDN layout) of our typing assistant.

Our user variables are:

- $I(1, 2, 3)$: the independence level of the user as a general trait
- $D(1, 2, 3)$: the distractibility level of the user as a general trait
- $N(1, 2, 3)$: the degree to which the user needs help with the current word
- $F(1, 2, 3)$: the degree to which the user is frustrated with the software suggestions
- $BR(1, 2, 3)$: the amount of browsing the user is doing, which is dependent on N and D

The variables I , and D capture characteristics about the user that are less likely to change in the short term, each of them having a discrete value ranging from $[1,3]$. Variables F and N also have discrete values in $[1,3]$ but they change more frequently. BR is a variable that characterizes a set of behaviours common to both needing help and being distracted. The model also has the following hidden variable:

- $CONS(1, 2, 3)$: the degree to which the user is considering the help box, which influences the kinds of observations we expect to detect

This variable behaves like a “select” variable in the model. Depending on the value of its parent nodes, it selects amongst its children variables. The structure of $CONS$ will ensure mutual exclusivity amongst the various events.

To assess user attitudes, we defined relevant observations that are causally related to these user variables. For example, if the user is frustrated with the software (in the context of a typing task), the user may jam into the keyboard several times showing a sequence of “ls;jdalsj;dafdsalj” keys or perform mouse clicks rapidly. All the observations that indicate one of I, D, N, F, BR are shown as $UOBS$. These include: continuously pressing a key down, moving the mouse back and forth rapidly, pressing multiple backspaces, switching windows rapidly, surfing menus without selecting an item, pausing, adjusting the slider up, adjusting the slider down, and typing. Observations that indicate $CONS$ are shown as $COBS$. These include: accepting help, hovering over the box, and pausing when the box is up. Every observed event advances to the next time step. We refer to this version of the model as our “event-based” model because time is defined with respect to observed events.

The system action is BOX – a binary variable, which indicates whether the suggestion box is currently on the screen or not. Associated with the box is $QUAL \in [0, 1]$: a continuous-valued “objective utility” of the words in the suggestion box computed via a language bigram model. At each time step, the value of BOX depends on the computed policy. However, if the user was considering it (e.g., by hovering over the box), then BOX remains up at this time step. We made this policy choice to avoid a box that was under consideration from disappearing on the user.

The system keeps a belief, bel , over the state variables and makes decisions with respect to this belief. At each time step, observations are detected, bel is updated, the system puts together completion words

and assesses *QUAL* of this set of words, and the system decides whether to pop up *BOX* or not. Now, because the user attitudes are unlikely to change at each event, we abstract this model over a larger, 5 second time interval. We refer to this version of the model as the “interval-based” model, shown in Figure 13. The two versions of the model have a similar underlying causal structure. In the interval-based model, the observations are accumulated events over 5 seconds, and causal relationships between state variables and observation variables are defined according to the likelihood of observing a specific number of events given a particular attitude value. Since we would like to be able to make more than one suggestion per time step, the model now has multiple *copies* of the variables associated with *BOX* and the quality and consideration of it.

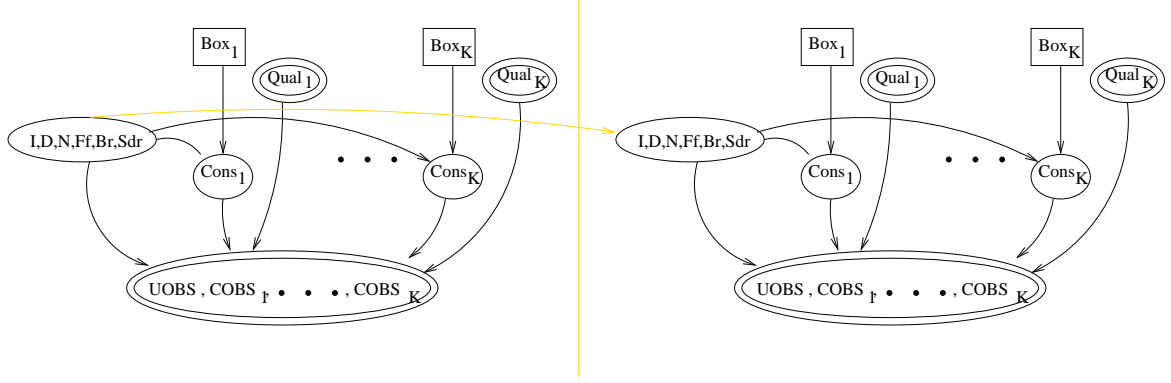


Figure 13: Interval-based model (in DDN layout) of our typing assistant.

To reason about making suggestions for the user, we consider the quality of the suggestion the system can offer (*QUAL*) and the inferred user attitudes (*I*, *D*, *N*, *F*). For instance, a person who is highly independent is less likely to accept help even if the quality of help is very good. On the other hand, a person who is needy and very dependent is likely to accept help even if the quality of help is not perfect. Thus, we define a subjective reward function, decomposed as follows: $U(I, D, N, F, QUAL) = U(I, F, QUAL) + U(D, QUAL) + U(N, QUAL)$. With each suggestion comes a cost of interruption, $Cost(I, D, N, F)$, which also varies according to different user types. Together, we define the following reward function that measures system actions at each decision point:

$$\begin{aligned}
 R &= 0, & \neg Pop \\
 &= U(I, D, N, F, QUAL) + Cost(I, D, N, F), & Pop \ \& \ Accept \\
 &= Cost(I, D, N, F), & Pop \ \& \ \neg Accept
 \end{aligned}$$

Our initial attempt at tackling this problem is to handcraft a reasonable myopic policy which allows us to assess our model and test the tractability of belief state monitoring. We compute the expected utility of making a suggestion, conditioned upon user accepting it:

$$EU(Pop) = EU(Pop|Accept)Pr(Accept) + EU(Pop|\neg Accept)Pr(\neg Accept)$$

The expected utilities $EU(Pop|Accept)$ and $EU(Pop|\neg Accept)$ are defined using R taken with respect to the belief of the user type, $bel(I, D, N, F)$. On the other hand, $EU(\neg Pop)$, the expected utility of not making the suggestion is zero. The policy is to suggest if $EU(Pop) \geq EU(\neg Pop)$.

In the above description, we have identified S , A , O , R , and bel of our POMDP model for a typing assistant. The transition and observation functions have been implicit in our DDNs and are currently handcrafted. To make the POMDP model more realistic, we will use a stochastic simulator based on the CPTs currently defined in the event-based model to simulate a more reasonable T and O functions in the interval-based model. Given an initial state distribution, infinite horizon, and discount factor, we will solve for an FSC using a factored version of the BPI algorithm discussed in Section 2.3 [PB04].

5.2 Implementation

Our system is implemented in Matlab, with the language module that selects completion words implemented in C. The CPTs of the event-based model are handcrafted. To fill in the CPTs of the interval-based model, we simulated the event-based model with distributions of temporal constraints and averaged the samples over all the iterations. For belief state monitoring, we implemented the clique tree algorithm to perform exact inference.

5.3 Simulations and Evaluations

The first evaluation we do is a simulation of different user types that exhibit different behavioural patterns. Recall that our model defines user features I, D, N, F . However, N and F are transient variables that do not directly reflect a user trait. On the other hand, a person who has a tendency to get frustrated (TF) will tend to have a higher value of F more frequently and a person who has a tendency to need help (TN) will tend to have a higher value more frequently. Both TN and TF can be binary variables, representing whether the person has the tendency, or not, to need help or to get frustrated respectively. Hence, we define the variables I, D, TN, TF as the possible profiles of user types in our model. We initialize the values for these user variables to simulate an “agreeable” type ($I = 1, D = 1, TN = 2, TF = 1$), an “easily aggravated” type ($I = 3, D = 3, TN = 1, TF = 2$), and a “neutral” type ($I = 2, D = 2, TN = 1, TF = 1$). Given a piece of text (2300 characters, about 3 paragraphs long) in the same genre as the corpus used to train the bigram model, we simulated the user typing from this text and generate observable events by sampling the event-based model. For comparison purposes, we used two additional policies of popping up the suggestion box when $Qual > 0$ and when $Qual > 0.6$. These policies do not take into consideration the belief about the user.

We will report the results for time performance in evaluating each online policy and the speed of belief state monitoring. We will also discuss the policy pattern in terms of the percentage of suggestions made, percentage of suggestions accepted, amount of work saved, and total reward.

Beyond the handcrafted policies, we will evaluate the model with an FSC that approximates the POMDP. With this policy, simulations can be run in the same way as described above and we can compare the quality of the policy. Finally, with a working interface, we would like to evaluate our system with real users. We elaborate on this point in Section 6.2.

6 Future Directions

In this section, we outline several directions of our project corresponding to the open problems identified in Section 4.

6.1 Applying POMDPs to Various Software Customization Problems

We believe that POMDPs provide the right model for software customization and we would like to demonstrate this point by modeling applications in different domains. In Section 5, we have fleshed out a model for a typing assistant under an environment where users can communicate in writing. This application was originally motivated by a project that develops cognitive aids for Alzheimers or traumatic brain injury patients [HLM03], even though our prototype is designed for a more general audience (including the average user, non-native English speakers, translators). To demonstrate that our approach works as a general user model, another application that would be interesting to pursue is a visualization assistant that automatically expands folders. This application is motivated by a colleague’s project in HCI [MDB04].

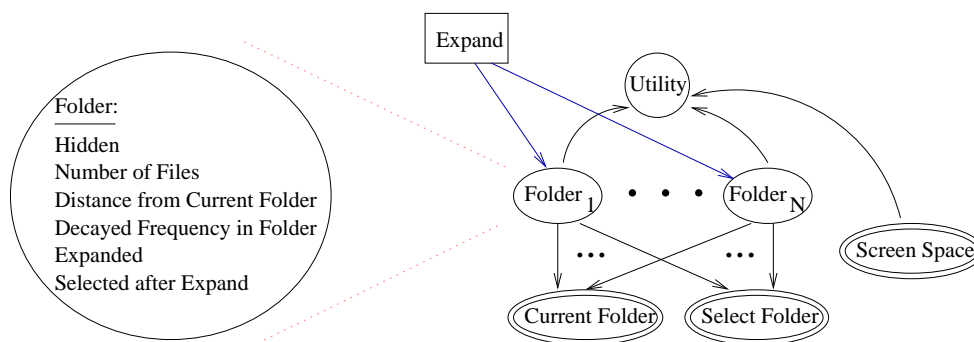


Figure 14: Preliminary model (in factored MDP layout) for visualization assistant.

The idea is to maximize the use of (desktop) screen space while considering the utility of the information presented to the user. For concreteness, we discuss the task of expanding file directories in a text display. Given n folders that may be expanded in limited screen space, the system must choose a subset of possibly nested folders to display on the screen. Each folder has attributes that keep track of its internal attributes (e.g., hidden or not, number of files to be listed, distance from current folder, frequency of use, whether it is currently expanded, and whether it was selected when it was expanded previously). Based on observing the user’s folder usage, these features get updated. Each folder would have an associated utility of how good it is to be expanded at the current time, and the system would choose a folder to expand by maximizing the expected utility. A preliminary model corresponding to this idea is illustrated in Figure 14.

6.2 Evaluation

In Section 5, the typing assistant model used an event based model to fill in the parameters of the interval based model. To assess user acceptance of the prototype, we would like to (i) prototype an interface (as in Figure 15), (ii) collect user data to learn model parameters (following methodologies of current work [AZN98, ZC03, GHMR⁺01]), and (iii) conduct laboratory user studies to get user feedback (e.g., [HBH⁺98]).

Learning the transition function in the model is a bit tricky because we would need to design a study that allows us to collect data of how the system states change as a function of user attitudes and how user attitudes change or persist over time. Recall that our model defines possible user types with the variables I, D, TN, TF , yielding $|I| \times |D| \times |TN| \times |TF| = 36$ user types. The hardest part of the evaluation is to know the type of the user participating in the experiment, or have a way to reveal his type. Aside from that, we would need to collect enough data for each user type. The experimental design of the study would also need to somehow simulate the varying levels of frustration, forcing the user to ask for help,

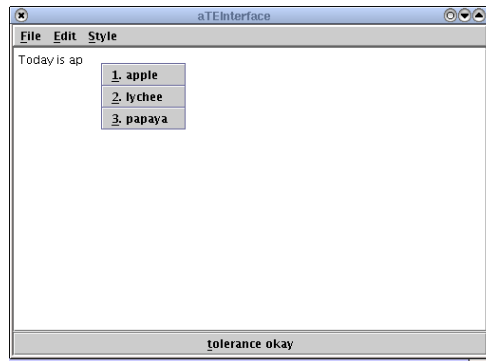


Figure 15: Mock-up of our adaptive text editor interface.

and distracting the user. The only problem for learning the observation function is to collect enough data for each user type. Once there is enough data, we could just take the average counts of observations as a function of their hidden parents. Learning the reward function in the model is even harder because it requires having the user reveal objectively to the system the value of being in each state. In reality, there is often much structure exhibited in a user’s reward function. In our model, we had assumed that the reward function is defined with respect to the variables $I, D, N, F, QUAL$, where $QUAL$ is continuous. (Note that even if we discretized $QUAL$ into, say, 10 values, this combination still yields 360 outcomes.) Therefore, it would be infeasible to pose a query to the user for a consistent, quantitative evaluation of each state. The general difficulty in learning the user’s reward function is elaborated in Section 6.3.

Conducting user studies amounts to carrying out a heuristic evaluation [Nie93] to assess usability, in addition to measuring some objective value such as typing effort saved and percentage of suggestions accepted (cf. Section 5.3 for simulation plans).

It would also be interesting to do an evaluation that compares the effectiveness of a static interface, an adaptable interface, and an adaptive interface, for different user types [FM04]. In the context of a typing assistant, a static interface would correspond to having a simple plan text editor (i.e., Microsoft Word would not work here), an adaptable interface that allows the user to manipulate the possible assistance levels via a slider widget, and an adaptive interface as the one we presented in Section 5.

6.3 Learning the Reward Function

The direction we would like to take for this project is to learn the user’s utility function in the context of software customization. Throughout this paper, we have described various models and motivated for the need to use POMDPs as a general model of the user interacting with software. Here, we describe how we model software customization as a POMDP.

In the interest of learning a user’s utility function, we view the problem as an IRL problem. That is, given a model of the world, the goal is to learn the user’s reward function (although not the *exact* function, as we elaborate below). There is a customizing agent whose job is to assist the user. The user may or may not be aware of this agent. Using only the system states would mean that the world is fully observable to the agent. However, as we would like to model user attitudes explicitly (as we did in Section 5 and in [GP00a, GP00b]), the world states would need to be decomposed as $S : S_C \times S_U$, where S_C is the set of customizable states and where S_U is the set of user states. In this case, since the agent cannot observe user states, the model is now partially observable.

The agent would have a set of actions, A , available at its disposal. The set of actions is a combination of actions that help the user with the current task and queries to the user to reveal information about his reward function. The user would have a set of actions, O , available as general actions in the software or responses to system queries. User actions may be higher level actions that exhibit behavioural pattern, such as pressing a key down continuously or moving the mouse back and forth quickly. System actions and user actions influence each other – the agent acts based on observing user actions and current state, while the user *reacts* to agent actions in the current state. This interaction highlights the difficulty in

modeling software customization.

To model system dynamics, we would need to have an idea of how the system states change. In particular, given only system states, S_C , it would be fairly straightforward to define a transition function, T , mapping S_C and agent actions onto S_C . Having added user states, S_U , makes the problem more complicated because of the unobservability of these variables. Consider the case if there were no adaptation at all. Then, system actions may be redefined as responses to user requests. In this case, there is still some problem in determining the state transitions across time step, but the uncertainty in S_C can be modeled via a belief state. In fact, this was our typing assistant model in Section 5. Now, we go back to the general case of allowing adaptation. The added complication is that system actions can directly influence multiple factors – they can directly manipulate system states, they can impact user attitudes towards the system, they can change the user’s behaviour pattern, and they can affect the system’s rewards. For these reasons, it would also be difficult to define an observation model, Z , because of the possible influences from system actions.

Fixing the objective of the agent, the parameters of the reward function, R , are predefined according to the domain. In particular, R should be a function of user attitude variables, S_U , as well as any “relevant” system states, S_C , possible user responses to being in the current state, O , and the action that was taken by the agent in the previous time step, A . Given particular user *types*, the structure of R can be handcrafted to reflect the user’s tolerance towards the system, user preferences of certain world states, and the cost of system interruptions at specific world states. Again, the goal here would not be to learn the *exact* R , but rather, we would keep a distribution over the space of reward functions, and use actions and observations to refine this distribution.

To act optimally, in this context, now refers to the agent choosing an action, $a \in A$, that maximizes the expected utility of outcomes with respect to the distribution of reward functions.

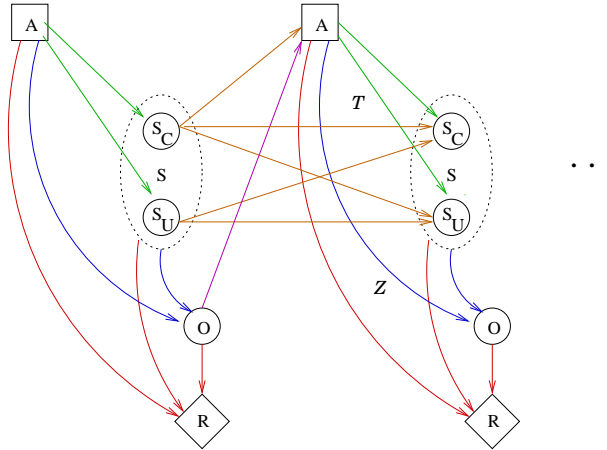


Figure 16: Influence diagram for software customization.

To summarize, the software customization problem cast as a POMDP with the influences illustrated in Figure 16 has the following parameters:

- $S : S_C \times S_U$, set of world states
- A , set of actions for the customizing agent
- $T : S \times A \times S \rightarrow [0, 1]$, transition function
- O , set of actions for the user
- $Z : S \times A \times O \rightarrow [0, 1]$, observation function
- $R : S \times A \times O \rightarrow \mathbb{R}$, agent’s reward function

As discussed above, the major difficulty is deriving or learning a reasonable model of the system dynamics. To tackle this problem, we hope to borrow techniques from preference elicitation and inverse reinforcement learning. An important issue is finding the right level of abstraction at which to model user reward functions and policies, and mapping system events into these. Furthermore, modeling uncertainty is bound to have computational bottlenecks. It is important to minimize the time it takes for the agent strategy to converge to a point where reasonable performance is possible because users are unlikely to tolerate “poor” system actions for long. It is also important to reduce online computational time because users are unlikely to want to wait for system reconfiguration. For these problems, we hope to adopt structural representations and adapt various approximate solution techniques for POMDPs.

References

- [AN04] Pieter Abeel and Andrew Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.
- [Ast65] K. J. Aström. Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Appl.*, 10:174–205, 1965.
- [AZN98] David W. Albrecht, Ingrid Zukerman, and Ann E. Nicholson. Bayesian models for keyhold plan recognition in an adventure game. *User Modeling and User-Adaptive Interaction*, 8:5–47, 1998.
- [Bau99] Mathias Bauer. Machine Learning for Plan Recognition. In *UM'99 Workshop on "Machine Learning for User Modeling"*, 1999.
- [BDH99] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.
- [Bel57] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.
- [BJ01] Thorsten Bohnenberger and Anthony Jameson. When policies are better than plans: Decision-theoretic planning of recommendation sequences. In *Proceedings of the International Conference on Intelligent User Interfaces*, 2001.
- [BJB98] Scott M. Brown, Eugene Santos Jr., and Sheila B. Banks. Utility theory-based user models for intelligent interface agents. In *Twelve Canadian Conference on Artificial Intelligence (AI'98)*, 1998.
- [BK98] Xavier Boyen and Daphne Koller. Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, 1998.
- [Bou02] Craig Boutilier. A POMDP formulation of preference elicitation problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 239–246, Edmonton, AB, 2002.
- [Bru01a] P. Brusilovsky. Adaptive hypermedia. *User Modeling and User Adapted Interaction*, pages 87–110, 2001.
- [Bru01b] Peter Brusilovsky. Adaptive hypermedia. *User Modeling and User-Adaptive Interaction*, 11(1-2):87–110, 2001.
- [BT96] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, Belmont, MA, 1996.
- [BZM03] Craig Boutilier, Richard S. Zemel, and Benjamin Marlin. Active collaborative filtering. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, Acapouco, 2003.
- [Car01] Sandra Carberry. Techniques for plan recognition. *User Modeling and User-Adaptive Interaction*, 11(1-2):31–48, 2001.
- [CGV02] Cristina Conati, Abigail Gertner, and Kurt VanLehn. Using bayesian networks to manage uncertainty in student modeling. *User Modeling and User-Adaptive Interaction*, 12, 2002.
- [Che88] Hsien-Te Cheng. *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia, Vancouver, 1988.
- [CK00] U. Chajewska and D. Koller. Utilities as random variables: Density estimation and structure discovery. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 63–71, 2000.

- [CKO01] Urszula Chajewska, Daphne Koller, and Dirk Ormoneit. Learning an agent’s utility function by observing behavior. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 35–42, 2001.
- [CKP00] U. Chajewska, D. Koller, and R. Parr. Making rational decisions using adaptive utility elicitation. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 363–369, 2000.
- [CLZ97] Anthony R. Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact method for POMDPs. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 54–61, Providence, RI, 1997.
- [CPA81] Phil R. Cohen, Ray C. Perrault, and James F. Allen. Beyond question-answering. In W. Lehnert and M. Ringle, editors, *Strategies for Natural Language Processing*, pages 245–274. Lawrence Erlbaum, Hillside NJ, 1981.
- [CV01] Cristina Conati and Kurt VanLehn. Providing adaptive support to the understanding of instructional material. In *Proceedings of the International Conference on Intelligent User Interfaces*, 2001.
- [Dec99] Rina Dechter. Bucket elimination: a unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.
- [DH98] D.B. Davison and H. Hirsh. Predicting sequences of user actions. Technical report, Rutgers State University of New York, 1998.
- [DK89] Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.
- [Dra62] A. Drake. *Observation of a Markov process through a noisy channel*. PhD thesis, Massachusetts Institute of Technology, 1962.
- [FC01] Michael Fleming and Robin Cohen. A user modeling approach to determining system initiative in mixed-initiative ai systems. In *Eighth International Conference on User Modeling*, pages 54–63, 2001.
- [FM04] Leah Findlater and Joanna McGrenere. A comparison of static, adaptive, and adaptable menus. In *ACM CHI*, 2004.
- [Fre86] Simon French. *Decision Theory*. Halsted Press, New York, 1986.
- [GHMR⁺01] Anthony Jameson Barbara Grobmann-Hutter, Leonie March, Ralf Rummer, Thorsten Bohnenberger, and Frank Wittig. When actions have consequences: Empirically based decision making for intelligent user interfaces. *Knowledge-Based Systems*, 14:75–92, 2001.
- [GP00a] Peter Gorniak and David Poole. Building a stochastic dynamic model of application use. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, 2000.
- [GP00b] Peter Gorniak and David Poole. Predicting future user actions by observing unmodified applications. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2000.
- [GW04] Krzysztof Gajos and Daniel S. Weld. Supple: Automatically generating user interfaces. In *Proceedings of the International Conference on Intelligent User Interfaces*, 2004.
- [Han98] Eric A. Hansen. Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 211–219, Madison, Wisconsin, 1998.
- [HBH⁺98] Eric Horvitz, Jack Breese, David Heckerman, David Hovel, and Koos Rommelse. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, 1998.

- [HD94] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedure guide. *Approximate Reasoning*, 11:1–158, 1994.
- [HLM03] Bowen Hui, Sotirios Liaskos, and John Mylopoulos. Requirements analysis for customizable software: A goals-skills-preferences framework. In *The Eleventh International Requirements Engineering Conference (RE'03)*, Monterey Bay, USA, September 2003.
- [HM84] Ronald A. Howard and James E. Matheson, editors. *Readings on the Principles and Applications of Decision Analysis*. Strategic Decision Group, Menlo Park, CA, 1984.
- [Hor99] Eric Horvitz. Principles of mixed-initiative user interfaces. In *CHI*, pages 159–166, 1999.
- [How60] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.
- [HSAHB99] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288, Stockholm, 1999.
- [KG00] Benjamin Korvemaker and Russ Greiner. Predicting UNIX command lines: Adjusting to user patterns. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2000.
- [Kja92] Uffe Kjaerulff. A computational scheme for reasoning in dynamic probabilistic networks. In *Proceedings of the Eighth Conference on Uncertainty in AI*, pages 121–129, Stanford, 1992.
- [KKLL99] K. Kang, S. Kim, J.J. Lee, and K.W. Lee. Feature-oriented engineering of pbx software for adaptability and reuseability. *Software Practice and Experience*, 29(10):875–896, 1999.
- [LFL98] Philippe Langlais, George Foster, and Guy Lapalme. TransType: a Computer-Aided Translation Typing System. In *xxx*, 1998.
- [Lit94] Michael L. Littman. The witness algorithm: Solving partially observable Markov decision processes. Technical Report CS-94-40, Brown University, Department of Computer Science, Providence, RI, December 1994.
- [LMM94] Y. Lashkari, M. Metral, and P. Maes. Collaborative interface agents. In *Proceedings of the Twelfth National Conference on AI*, volume 1, Seattle, WA, August 1994. AAAI Press.
- [LRS99] Neal Lesh, Charles Rich, and Candace L. Sidner. Using Plan Recognition in Human-Computer Collaboration. In *International Conference on User Modeling*, pages 23–32, June 1999.
- [LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *J. Roy. Stat. Soc. B*, 50:157–224, 1988.
- [Mae97] Pattie Maes. Agents that reduce work and information overload. In Jeffrey M. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1997.
- [MBB02] J. McGrenere, R.M. Baecker, and K.S. Booth. An evaluation of a multiple interface design solution for bloated software. In *Proceedings of ACM CHI 2002, ACM CHI Letters 4(1)*, pages 163–170, 2002.
- [McC96] Andrew R. McCallum. Instance-based state identification for reinforcement learning. Technical report, University of Rochester, USA, 1996.
- [MCH03] O. Madani, A. Condon, and S. Hanks. On the Undecidability of Probabilistic Planning and Infinite-Horizon Partially Observable Markov Decision Process Problems. *Artificial Intelligence, Special Issue on Planning with Uncertainty and Incomplete Information*, 147(1-2):5–34, 2003.

- [MDB04] Michael J. McGuffin, Gord Davison, and Ravin Balakrishnan. Expand-Ahead: Browsing trees with increased screen space use and faster drill-down, 2004. (Under review).
- [Mon82] George E. Monahan. A survey of partially observable Markov decision processes: Theory, models and algorithms. *Management Science*, 28:1–16, 1982.
- [MP02] Colleen E. McCarthy and Martha E. Pollack. A plan-based personalized cognitive orthotic. In *Proceedings of the 6th International Conference on AI Planning and Scheduling*, 2002.
- [MPKK99] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 427–436, Stockholm, 1999.
- [Mur02] Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, Computer Science Division, UC Berkeley, California, USA, 2002.
- [MV00] C. Murray and K. VanLehn. DT Tutor: A decision-theoretic dynamic approach for optimal selection of tutorial actions. In *Proceedings of Intelligent Tutoring Systems (ITS 2000), 5th International Conference*, Montréal, Canada, 2000. Springer.
- [NH98] Hien Nguyen and Peter Haddawy. The Decision-Theoretic Video Advisor. In *Working Notes of the AAAI-98 Workshop on Recommender Systems*, pages 77–80, July 1998.
- [Nie93] Jakob Nielsen. *Usability Engineering*. Academic Press, Inc., 1993.
- [NR00] Andrew Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.
- [PB03] Pascal Poupart and Craig Boutilier. Value-directed Compression of POMDPs. In *Proceedings of Conference on Neural Information Processing Systems*, 2003.
- [PB04] Pascal Poupart and Craig Boutilier. Bounded Finite State Controllers. In *Proceedings of Conference on Neural Information Processing Systems*, 2004.
- [Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- [PT87] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.
- [Put94] M. L. Puterman. *Markov Decision Problems*. Wiley, New York, 1994.
- [RG03] Nicholas Roy and Geoffrey Gordon. Exponential Family PCA for Belief Compression in POMDPs. In *Proceedings of Conference on Neural Information Processing Systems*, 2003.
- [RPT01] Nicholas Roy, Joelle Pineau, and Sebastian Thrun. Spoken dialog management for robots. In *Association for Computational Linguistics (ACL)*, 2001.
- [SAHB00] Robert St-Aubin, Jesse Hoey, and Craig Boutilier. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings of Conference on Neural Information Processing Systems*, pages 1089–1095, Denver, 2000.
- [SBH02] Guy Shani, Ronen I. Brafman, and David Heckerman. An MDP-Based Recommender System. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, 2002.
- [Sha86] Ross D. Shachter. Evaluating influence diagrams. *Operation Research*, 33(6):871–882, 1986.
- [SS73] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.
- [TS90] Joseph A. Tatman and Ross D. Shachter. Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 20(2):365–379, 1990.

- [Wær97] Annika Wærn. Local Plan Recognition in Direct Manipulation Interfaces. In *Proceedings of the International Conference on Intelligent User Interfaces*, 1997.
- [ZC03] Xiaoming Zhou and Cristina Conati. Inferring user goals from personality and behavior in a causal model of user affect. In *Proceedings of the International Conference on Intelligent User Interfaces*, 2003.
- [ZL01] Ingrid Zukerman and Diane Littman. Natural language processing and user modeling: Synergies and limitations. *User Modeling and User-Adaptive Interaction*, 11(1-2):129–158, 2001.