Recommendation as a Stochastic Sequential Decision Problem

Ronen I. Brafman
Dept. of Computer Science
Ben-Gurion University
Beer-Sheva 84105, Israel
brafman@cs.bgu.ac.il
+972 55 466 499

David Heckerman
Microsoft Research
One Microsoft Way
Redmond, WA 98052
heckerma@microsoft.com
+1 425 706 2662

Guy Shani
Dept. of Computer Science
Ben-Gurion University
Beer-Sheva 84105, Israel
shanigu@cs.bgu.ac.il
+972 55 670 556

Abstract

Recommender systems — systems that suggest to users in e-commerce sites items that might interest them — adopt a static view of the recommendation process and treat it as a prediction problem. In an earlier paper, we argued that it is more appropriate to view the problem of generating recommendations as a sequential decision problem and, consequently, that Markov decision processes (MDPs) provide a more appropriate model for recommender systems. MDPs introduce two benefits: they take into account the long-term effects of each recommendation, and they take into account the expected value of each recommendation. The use of MDPs in a commercial site raises three fundamental problems: providing an adequate initial model, updating this model online as new items (e.g., books) arrive, and coping with the enormous state-space of this model. In past work, we dealt with the first problem. In this paper we consider the second, and especially, the third problem, which is of greater concern to researchers in decision-theoretic planning. We show that although the model we consider has roughly 10^{11} states, we can quickly provide an approximate solution by utilizing its special structure. Our memory requirements – a serious concern for commercial online applications – are modest; and the overall resource requirements of our system are comparable to those of a well-known commercial recommender system that uses a simpler and less accurate model. Our system is one of a handful of deployed commercial recommender systems as well as one of a handful of MDP-based deployed systems. It has been running at www.mitos.co.il, a commercial online bookstore, since August, 2002.

Introduction

In many markets, consumers are faced with a wealth of products and information from which they can choose. To alleviate this problem, web sites attempt to help users by incorporating a *Recommender system* (Resnick & Varian 1997) that provides users with a list of items and/or web-pages that are likely to interest them. Once the user makes her choice, a new list of recommended items is presented. Thus, the recommendation process is a sequential process. Moreover, in many domains, user choices are sequential in nature, e.g., we buy a book by the author of a recent book we liked.

The sequential nature of the recommendation process was noticed in the past (Zimdars, Chickering, & Meek 2001). Taking this idea one step farther, we suggest that recommending is not simply a sequential prediction problem, but rather, a sequential decision problem. At each point the Recommender system makes a decision: which recommendation to issue. This decision should be optimized taking into account the sequential process involved and the optimization criteria suitable for the Recommender system. Thus, we suggest the use of Markov Decision Processes (MDP) (Puterman 1994), a well known stochastic model of sequential decisions. With this view in mind, a more sophisticated approach to Recommender systems emerges. First, one can take into account the utility of a particular recommendation – e.g., we might want to recommend a product that has a slightly lower probability of being bought, but generates higher profits. Second, we might suggest an item whose immediate reward is lower, but leads to more likely or more profitable rewards in the future. These considerations are taken into account automatically by any good or optimal policy generated for an MDP model of the recommendation process. In particular, an optimal policy will take into account the likelihood of a recommendation to be accepted by the user, the immediate value to the site of such an acceptance, and the long-term implications of this on the user's future choices. And these considerations are taken with the appropriate balance to ensure the generation of the maximal expected reward stream.

For instance, consider a site selling electronic appliances faced with the option to suggest a video camera with a success probability of 0.5, or a VCR with a probability of 0.6. The site may choose the camera, which is less profitable, because the camera has accessories that are likely to be purchased, whereas the VCR does not. If a video-game console is another option with a smaller success probability, the large profit from the likely future event of selling game cartridges may tip the balance toward this latter choice. Similarly, a book with a sequel or by an author that has many other titles may be likely to lead to additional purchases more than an average book.

Keeping these benefits in mind, we suggest an approach for the construction of an MDP-based Recommender system. To make this a viable approach, our formulation must address two key challenges. Our first problem is model gen-

eration. Standard reinforcement-learning techniques are inadequate in our case because their initial behavior is random. No commercial site will agree to a long (or even short) period of random recommendations. To handle this problem, we suggest the use of historical data describing user behavior in the same site *before* the introduction of a recommender system. Using this data, we learn a predictive stochastic model describing the sequential behavior of users without recommendations. This predictive model forms the basis for the initialization of the MDP. In (Shani, Brafman, & Heckerman 2002), where we first suggested the use of MDPs for modeling the recommendation process, we devoted our attention to the description and evaluation of a particular predictive model that performs quite well, and we demonstrated the clear predictive advantage of the sequential formulation. However, as we noted there, the choice of the predictive model is orthogonal to the MDP-based recommender system model, and any other good predictive model could be used. Thus, in this paper we simply assume an arbitrary predictive model as given.

Our second problem is the huge state-space that arises in our application domain, which is on the order of 10^{11} states. It is not possible to handle this state-space explicitly, both because of its space requirements – which would slow down the operation of the whole site – and its time requirements – which would make even off-line optimization impossible. To address this problem we resort to an approximate solution that makes use of the special structure of our state space. Indeed, the attempt to exploit structure to provide more efficient space and time performance of MDP solution algorithms has been a major theme in MDP-research in the decision-theoretic planning community for the past decade. This research produced interesting and potentially useful results. However, except for a small number of papers (e.g., (Meuleau *et al.* 1998)) it has not been inspired by real applications. As we shall see, the type of structure we use differs from most past examples, and allows us to provide reasonable approximations quite easily.

Our third problem is model update. In most applications of recommender systems, the set of items changes, typically because of the arrival of new products. We show how our particular approach is able to cope with this problem while maintaining its modest space and time requirements.

Validating our MDP approach is not simple. Most Recommender systems, such as dependency networks (Heckerman *et al.* 2000), are tested on historical data for their predictive accuracy. That is, the system is trained using historical data from sites that do not provide recommendations, and tested to see whether the recommendations conform to actual user behavior. In (Shani, Brafman, & Heckerman 2002), we showed that our particular predictive model yields more accurate predictions than a state-of-the-art commercial algorithm in this area. While this test provides some indication of the system's abilities, it does not test how user behavior is influenced by the system's suggestions or what percentage of recommendations are accepted by users. To obtain this data, one must employ the system in a real site with real users, and compare the performance of this site with and without the system (or with this and other systems). Such an

evaluation is currently underway, as our system has been deployed at the online book store Mitos (www.mitos.co.il). As commercial sites typically do not see themselves as experimental labs, this deployment can be viewed as somewhat of a success story in itself.

This paper is organized as follows: In Section 2 we provide some background on recommender systems, discuss the sequential nature of recommendations, and review the Markov decision processes model. In Section 3 we formulate the recommendation process using an MDP. In Section 4 we show how we can provide approximate solutions quickly. In Section 5 we explain how we update this model online. In Section 6 we show how our model compares to a well-known commercial recommender system in terms of its space and time performance. We conclude with a discussion of future work in Section 7.

Background

In this section we provide some background on recommender systems, collaborative filtering, and the MDP model.

Recommender Systems

Early in the 90's, when the Internet became widely used as a source of information, *information explosion* became an issue that needed addressing. Many web sites presenting a wide variety of content (such as articles, news stories or items to purchase) discovered that users had difficulties finding the items that interested them out of the total selection.

So called *recommender systems* (Resnick & Varian 1997), help users limit their search by supplying a list of items that might interest a specific user. Different approaches were suggested for supplying meaningful recommendations to users and some were implemented in modern sites (Schafer, Konstan, & Riedl 1999). Traditional data mining techniques such as association rules have been tried at the early stages of the development of recommender systems, but proved to be insufficient for the task. Approaches originating from the field of *information retrieval (IR)* rely on the *content* of the items (such as description, category, title, author) and therefore are known as *content-based recommendations* (Mooney & Roy 2000). These methods use some similarity score to match items based on their contents, then a list of similar items to the ones the user previously selected can be supplied. Another possibility is to avoid using information about the content, but rather use historical data gathered from other users in order to make a recommendation. These methods are widely known as *collaborative filtering (CF)* (Resnick *et al.* 1994). Many new systems try to create hybrid models that combine collaborative filtering and content-based recommendations together (Balabanovic & Shoham 1997).

Collaborative Filtering

The collaborative filtering approach originates in human behavior; people trying to find an interesting item they know little of (e.g. when trying to decide which movie to take from the video store), tend to rely on friends to recommend items

they tried and liked (good movies they saw). The person asking for advice is using a (small) community of friends that know her taste and can therefore make good predictions as to whether she will like a certain item. Over the net however, a larger community that can recommend items to our user is available, but the persons in this gigantic community know little or nothing about each other. The goal of a collaborative filtering engine is to identify those users whose taste in items is predictive of the taste of a certain person (usually called a *neighborhood*), and use their recommendations to construct a list of items interesting for her.

To build a user's neighborhood, these methods rely on a database of past users interactions with the system. Originally, the database consisted of items and some type of score given by the users who experienced those items (e.g. 5 stars to a great movie, 1 star to a horrible one). The user had to enter the site, grade a few items, and then a list of items she did not grade, but that the system predicts she will score highly, can be presented to her.[1] This approach is called *explicit ratings*, since users explicitly rate the items.

A rating system that uses data collected without actually asking the user to grade items is known as *implicit ratings*. A common approach assumes that people like what they buy. A binary grading method is used when a value of 1 is given to items the user has bought and 0 to other items. Many modern recommender systems successfully implement this approach. Claypool *et al.* (Claypool *et al.* 2001) suggested using other methods through a special web browser that kept track of user behavior such as the time spent looking at the web page, the scrolling of the page by the user, and movements of the mouse over the page, but failed to establish a method of rating that gave consistently better results.

Sequential Recommendations

It has been suggested before (Zimdars, Chickering, & Meek 2001) that one can look at the input data not as sets of ratings but also consider the order in which they were rated. Most recommender systems work in a sequential form: they suggest items to the user who can then take one of the recommendations. At the next stage a new list of recommended items is calculated and presented to the user. This process is clearly sequential by its nature - at each stage a new list is calculated based on the user's past ratings. Zimdars *et al.* suggested using standard algorithms such as dependency networks (Heckerman *et al.* 2000) by transforming the input data into a sequential form. The set of cases in the input data set for the training algorithm consists of sequences of items (i.e., merchandise bought, web-pages viewed, etc.). This data is transformed into a new data set that expresses this sequential structure using a standard *auto-regressive* process. Namely, it divides the data into two variable groups: source variables $X_{-k}, X_{-k+1}, ..., X_{-1}$ representing the last k items that were rated, and a target variable X_T that represents the next item in the sequence. This simple transformation is illustrated in Table 1 for $k = 3$ and an initial sequence of four items, x_1, x_2, x_3, x_4. We see that four distinct

Case ID	X_{-2}	X_{-1}	X_T
1	–	–	x_1
2	–	x_1	x_2
3	x_1	x_2	x_3
4	x_2	x_3	x_4

Table 1: Transforming the case x_1, x_2, x_3, x_4 using data expansion into 4 new cases

cases are obtained, representing the "state" prior to each purchase by the user. Next, a predictive model is built to predict X_T given $X_{-k}, ..., X_{-1}$. Dependency networks using a data set that was transformed using the auto-regressive approach showed superior predictive accuracy to non-sequential dependency networks.

Markov Decision Processes (MDPs)

An MDP is a model for sequential stochastic decision problems. As such, it is widely used in applications where an autonomous agent is influencing its surrounding environment through actions (e.g., a navigating robot). MDPs (Bellman 1962) have been known in the literature for quite some time, but due to some fundamental problems discussed below, not many commercial applications have been implemented.

An MDP is by definition a four-tuple: $\langle S, A, Rwd, tr \rangle$, where S is a set of states, A is a set of actions, Rwd is a reward function that assigns a real value to each state/action pair, and tr is the state-transition function, which provides the probability for a transition between every pair of states given each action.

In an MDP, the decision-maker's goal is to behave so that some function of its reward stream is maximized – typically the average reward or the sum of discounted reward. An optimal solution to the MDP is such a maximizing behavior. Formally, a stationary policy for an MDP π is a mapping from states to actions, specifying which action to perform at each state. Given such an optimal policy π, the agent need only establish what state s it is in and execute the action $a = \pi(s)$ at each stage of the decision process.

Various exact and approximate algorithms exist for computing an optimal policy. Below we briefly review policy-iteration (Howard 1960), which we use in our implementation. A basic concept in all approaches is that of the *value function*. The value function of a policy ρ, denoted V^ρ, assigns to each state s a value which corresponds to the expected infinite-step discounted sum of rewards obtained when using ρ starting from s. This function can be computed using the following set of equations:

$$V^\rho(s) = Rwd(s, \rho(s)) + \gamma \sum_{s_j \in S} tr(s, \rho(s), s_j) V^\rho(s_j)$$

where $0 < \gamma < 1$ is the discount factor.[2] The *optimal*

---

[1]An example of such a system can be found at http://www.movielens.umn.edu/.

[2]We use discounting mostly for mathematical convenience. True discounting of profit would have to take into account the actual times in which different books are purchased, leading to a semi-Markov model, which does not seem worth the extra effort involved.

value function, denoted V^*, assigns to each state s its value according to the optimal policy ρ^*. It can be computed (usually by means of an iterative process) based on the following set of equations:

$$V^*(s) = \max_{a \in A} [Rwd(s,a)) + \gamma \sum_{s_j \in S} tr(s,a,s_j) V^*(s_j)]$$

In policy-iteration we search the space of possible policies. We start with an initial policy $\pi_0(s) = argmax_{a \in A} Rwd(s,a)$. At each step we compute the value function based on the former policy and update the policy given the new value function.

$$V_i(s) = Rws(s, \pi_i(s)) + \gamma \sum_{s_j \in S} tr(s, \pi_i(s), s_j) V_i(s_j)$$

$$\pi_{i+1}(s) = argmax_{a \in A} Rwd(s,a) + \gamma \sum_{s_j \in S} tr(s,a,s_j) V_i(s_j)$$

Once the policy stops changing we have reached the optimal policy.

Solving MDPs is known to be a polynomial problem in the number of states (Puterman 1994). However, the number of states is exponential in the number of state variables, and often, the natural representation of the problem is at the variable level. This well known "curse of dimensionality" makes algorithms based on an explicit representation of the state-space impractical. Thus, a major research effort in the area of MDPs during this last decade has been on computing an optimal policy in a tractable manner using factored representations of the state space and other techniques (Boutilier, Dearden, & Goldszmidt 2000). Unfortunately, these methods do not seem applicable in our domain in which the structure of the state space is quite different – i.e., each state can be viewed as an assignment to a very small number of variables with very large domains. However, we were able to exploit the special structure of our state and action spaces using different (and much simpler) techniques. In addition, we introduced approximations that exploit the fact that most states – i.e, most item sequences – are highly unlikely to occur.

An MDP-Based Recommendation Strategy

Our ultimate goal is to construct a recommender system—a system that chooses a link, product, or other item to recommend to the user at all times. In this section, we describe how such a system can be based on an MDP. We assume that we are given a set of cases describing user behavior within a site that does not provide recommendations, as well as a probabilistic predictive model of a user acting without recommendations generated from this data. The set of cases is needed to support some of the approximations we make, and in particular, the lazy initialization approach we take. The predictive model provides the probability the user will purchase a particular item x given that her sequence of past purchases is $x_1, \ldots, x_k$. We denote this value by $Pr_{pred}(x|x_1, \ldots, x_k)$, where $k = 3$ in our case. There are various ways of constructing such a model (e.g., (Shani, Brafman, & Heckerman 2002; Kadie, Meek, & Heckerman 2002)).

Defining The MDP

The states of the MDP for our recommender system are k-tuples of items (e.g., books, CDs) some prefix of which may contain null values only, allowing us to model shorter sequences of purchases.

The actions of the MDP correspond to a recommendation of an item. One can consider multiple recommendations but, to keep our presentation simple, we start by discussing single recommendations first. When we recommend an item x', the user has three options:

- Accept this recommendation, thus transferring from state $\langle x_1, x_2, x_3 \rangle$ into $\langle x_2, x_3, x' \rangle$

- Select some non-recommended item x'', thus transferring the state $\langle x_1, x_2, x_3 \rangle$ into $\langle x_2, x_3, x'' \rangle$.

- Select nothing (e.g., when the user terminates the session).

The rewards in our MDP encode the utilities of selling items (or showing web pages) as defined by the site. Since the state encodes the list of items purchased, the reward depends on the current state only. For example, the reward for state $\langle x_1, x_2, x_3 \rangle$ can be the profit generated by the site from the sale of item x_3—the last item in the transaction sequence. While this is the reward function we have implemented, other choices have their merit, and we discuss this issue again later.

Notice that the stochastic element in our model is the user's actual choice given all possible options. The transition function for the MDP model:

$$tr^1_{MDP}(\langle x_1, x_2, x_3 \rangle, x', \langle x_2, x_3, x'' \rangle)$$

is the probability that the user will select item x'' given that item x' is recommended. We write tr^1_{MDP} to denote that only single item recommendations are used.

Unlike traditional model-based reinforcement learning algorithms that learn the proper values for the transition function and hence the optimal policy online, our system needs to be fairly accurate when it is first deployed. A for profit e-commerce[3] site is unlikely to use a recommender system that generates irrelevant recommendations for a long period, while waiting for it to converge to an optimal policy. We therefore need to initialize the transition function carefully. We can do so based on any good predictive model making the following assumptions:[4]

- A recommendation increases the probability that a user will buy an item. This probability is proportional to the probability that the user will buy this item in the absence of recommendations. This assumption is made by most CF models dealing with e-commerce sites. We denote the proportionality constant by α, where $\alpha > 1$.

- The probability that a user will buy an item that was not recommended is lower than the probability that she will

---

[3]We use the term e-commerce, but our system, and recommender systems in general, can be used in content sites, and other applications.

[4]In the implemented system, this is done using the Markov-chain predictive model of (Shani, Brafman, & Heckerman 2002)

buy it if it was recommended, but still proportional to it. We denote the proportionality constant by β, where $\beta < 1$.

- A third option is that the user will decide not to buy any item, thus remaining in the same state.

To allow for a simpler representation of the equations, for a state $s = \langle x_1, ..., x_k \rangle$ and a recommendation r let us denote by $s \cdot r$ the state $s' = \langle x_2, ..., x_k, r \rangle$. We use $tr_{predict}(s, s \cdot r)$ to denote the probability that the user will choose r next, given that its current state is s according to the predictive model in which recommendations are not considered, i.e., $Pr_{pred}(r|s)$. Thus,

$$tr^1_{MDP}(s, r, s \cdot r) = \alpha tr_{predict}(s, s \cdot r)$$

$$tr^1_{MDP}(s, r', s \cdot r) = \beta tr_{predict}(s, s \cdot r), \ r' \neq r$$

$$tr^1_{MDP}(s, r, s) = \\ 1 - tr^1_{MDP}(s, r, s \cdot r) - \sum_{r' \neq r} tr^1_{MDP}(s, r', s \cdot r)$$

where α and β are constant for combinations of an initial state and an item r. We do not see a reason to stipulate a particular relationship between α and β, but they must satisfy:

$$tr^1_{MDP}(s, r, s \cdot r) + \sum_{r' \neq r} tr^1_{MDP}(s, r', s \cdot r) < 1.$$

In our implemented system, we use a value of 0 for β. This allows us to use a constant, state-independent, value for α. Thus, initially, our recommendations are based strongly on the probability that an item will actually be bought. When $\beta > 0$, care is required to ensure that the sum of transition probabilities does not exceed 1.

Note that, initially, $tr^1_{MDP}(s, r', s \cdot r)$ does not depend on r' since our calculations are based on data that was collected without the benefit of recommendations.[5] This representation of the transition function allows us to keep at most two values for every pair of states, i.e., the probability that an item will be chosen when it is recommended and when it is not recommended. In fact, many states are not actually initialized due to the use a "lazy initialization" technique, described later. We actually maintain only two values for every state–item pair, with the number of items being much smaller than the number of states.

When moving to multiple recommendations we make the assumption that recommendations are not mutually dependent. Namely we assume that for every pair of sets of recommended items, R, R', we have that

$$tr_{MDP}(s, \{r\} \cup R, s \cdot r) = tr_{MDP}(s, \{r\} \cup R', s \cdot r)$$

This assumption might prove false. For example consider the case where the system "thinks" that the user is interested in an inexpensive cooking book. It can then recommend a few cooking books where most are very expensive and one

---

[5]Of course, if we have access to data that reflected the effect of recommendations, we can have a more accurate initial model.

is reasonably priced (but in no way cheap). The reasonably priced book will seem economic compared to the expensive ones, thus making the user more likely to take it.

Nevertheless, we make this assumption so as not to be forced to create a larger action space where actions are ordered combinations of recommendations. Taking the simple approach for representing the transition function we defined above, we still keep only two values for every state, item pair:

$$tr_{MDP}(s, r \in R, s \cdot r) = tr^1_{MDP}(s, r, s \cdot r)$$

$$tr_{MDP}(s, r \notin R, s \cdot r) = tr^1_{MDP}(s, r', s \cdot r), \\ \text{for all } r' \neq r$$

As before $tr_{MDP}(s, r \in R, s \cdot r)$ does not depend on r, and will not depend on R in the discussion that follows. We note again that these values are no more than smart initial values and would most likely require adjustments based on actual user behavior.

Solving The MDP

Having defined the MDP, we now consider how to solve it in order to obtain an optimal policy. Such a policy will, in effect, tell us what item to recommend given any sequence of user purchases. For the domains we studied, we found policy iteration (Howard 1960)—with a few approximations to be described—to be a tractable solution method. In fact, on tests using real data, we found that policy iteration terminates after a handful of iterations. This stems from the special nature of our state space and the approximations we made. In particular, fast convergence occurs because our iterations make use of the inherent directionality in the state space. That is, a state representing a short sequence cannot follow a state representing a longer sequence.

We have also found that the computation of the optimal policy is not heavily sensitive to variations in k — the number of past transactions we encapsulate in a state. As k increases, so does the number of states, but the number of positive entries in our transition matrix remains similar. Note that, at most, a state can have as many successors as there are items. When k is small, the number of observed successors for a state is indeed close to the number of items. When k is larger, however, the number of successors decreases considerably.

Although the number of iterations required is small, each iteration requires the computation of the expected rewards for every state, given the current policy. Even though we reduced the state space by limiting each state to hold the last k transactions, the state space remains large even when $k = 3$ and the number of items is roughly 5000. Thus, an accurate computation of an optimal policy is extremely time consuming. We reduce run time using a number of approximations.

As explained above, the vast majority of states in our models do not correspond to sequences that were observed in our training set. This fact holds because most combinations of items are extremely unlikely. For example, it is unlikely to find adjacent purchases of a science-fiction and a gardening book. This leads us to the use a lazy initialization

approach in which we maintain transition probabilities only for states for which a transition occurred in our training data. These transitions correspond to pairs of states of the form s and $s \cdot r$. Thus, the number of transitions required per state is bounded by the number of items (rather than by this number to the power of k).

Moreover, in our approximation, we do not compute a policy choice for a state that was not encountered in our training data. We use the immediate reward for such states as an approximation of their long-term expected value for the purpose of computing the value of a state that appeared in our training data. This approximation, although risky in general MDPs, is motivated by the fact that in our initial model, the probability of making a transition into an unencountered state is very low. Typically, there is a small number of items to which a transition is likely, and the probability of making a transition to any other state is very small. Moreover, the reward (i.e., profit) does not change significantly across different states, so, there are no "hidden treasures" in the future that we could miss. Thus, we can bound the value of such states and the loss incurred by using their immediate reward as their value, allowing us to make an informed choice as to which states can be ignored.

Of course, sometimes a recommendation must be generated for a state that was not encountered in the past. In that case, we compute the value of the policy for this state online. This requires us to estimate the transition probabilities for a state that did not appear in our training data. This is done using a number of generalization methods described in (Shani, Brafman, & Heckerman 2002).

We use another special feature of our representation of the MDP to allow for fast computations. As we have shown earlier, the computation of policy iteration requires the computation of the value function at each stage:

$V_{i+1}(s) = Rwd(s) + \gamma \max_R \sum_{s' \in S} tr(s, R, s')V_i(s') =$
$Rwd(s) + \gamma \max_R ($
$\quad \sum_{r \in R} tr_{MDP}(s, r \in R, s \cdot r)V_i(s \cdot r) +$
$\quad \sum_{r \notin R} tr_{MDP}(s, r \notin R, s \cdot r)V_i(s \cdot r))$

where $tr(s, r \in R, s \cdot r)$ and $tr(s, r \notin R, s \cdot r)$ follow the definitions above.

Notice that this equation requires maximization over the set of possible recommendations. The number of possible recommendations is n^κ, where n is the number of items and κ is the number of items we recommend each time. To handle this large action space, we make use of our independence assumption with respect to recommendation. Recall that we assumed that the probability that a user buys a particular item depends on her current state, the item, and whether or not this item is recommended. It does not depend on the identity of the other recommended items. The following method utilizes this fact to quickly generate the optimal set of recommendations for each state.

Let us define $\Delta(s, r)$ — the additional value of recommending r in state s:

$\Delta(s, r) = (tr(s, r \in R, s \cdot r) - tr(s, r \notin R, s \cdot r))V(s \cdot r)$

Now we can define

$R_{max\Delta}^{s, \kappa} = \{r_1, ..., r_\kappa | \Delta(s, r_1) \geq ... \geq \Delta(s, r_\kappa)$ and

$\forall r \neq r_i, \Delta(s, r_\kappa) \geq \Delta(s, r)\}$

$R_{max\Delta}^{s,\kappa}$ is the set of κ items that have the maximal $\Delta(s, r)$ values.

Theorem 1 $R_{max\Delta}^{s,\kappa}$ *is the set that maximizes* $V_{i+1}(s)$*, i.e.,*

$V_{i+1}(s) = Rwd(s) + \gamma ($
$\quad \sum_{r \in R_{max\Delta}^{s,\kappa}} tr(s, r \in R, s \cdot r)V_i(s \cdot r) +$
$\quad \sum_{r \notin R_{max\Delta}^{s,\kappa}} tr(s, r \notin R, s \cdot r)V_i(s \cdot r))$

Proof: Let us assume that there exists some other set of κ recommendations $R \neq R_{max\Delta}^{s,\kappa}$ that maximizes $V_{i+1}(s)$. For simplicity, we will assume that all Δ values are different. If that is not the case then R should be a set of recommendations not equivalent to $R_{max\Delta}^{s,\kappa}$. Let r be an item in R but not in $R_{max\Delta}^{s,\kappa}$ and r' be an item in $R_{max\Delta}^{s,\kappa}$ but not in R. Let R' be the set we get when we replace r with r' in R. We need only show that $V_{i+1}(s, R) < V_{i+1}(s, R')$.

$V_{i+1}(s, R') - V_{i+1}(s, R) =$
$Rwd(s) + \sum_{s'} tr(s, R, s')V_i(s') -$
$(Rwd(s) + \sum_{s'} tr(s, R', s')V_i(s')) =$
$\sum_{r'' \in R} tr(s, r'' \in R, s \cdot r'')V_i(s \cdot r) +$
$\sum_{r'' \notin R} tr(s, r'' \notin R, s \cdot r'')V_i(s \cdot r'') -$
$\sum_{r'' \in R'} tr(s, r'' \in R', s \cdot r'')V_i(s \cdot r) -$
$\sum_{r'' \notin R'} tr(s, r'' \notin R', s \cdot r'')V_i(s \cdot r'') =$
$tr(s, r \in R, s \cdot r)V_i(s \cdot r) - tr(s, r' \notin R, s \cdot r')V_i(s \cdot r') -$
$(tr(s, r' \in R', s \cdot r)V_i(s \cdot r) - tr(s, r \notin R', s \cdot r')V_i(s \cdot r)) =$
$\Delta(s, r) - \Delta(s, r') > 0$

∎

To compute $V_{i+1}(s)$ we therefore need to compute all $\Delta(s, r)$ and find $R_{max\Delta}^{s,\kappa}$, making the computation of $V_{i+1}(s)$ independent of the number of subsets (or even worse — ordered subsets) of κ items and even independent of κ itself. The complexity of finding the optimal policy when recommending multiple items at each stage under the assumptions mentioned above remains the same as the complexity of computing an optimal policy for single item recommendations.

Using this approach assumes the best recommendation to be that of an item whose appearance in the recommendations list enhances the likelihood of it being bought—known as the $lift$ of an item.

The formalization of the MDP described above causes the system to optimize the site profits. The model is rewarded for items that were purchased no matter whether the system recommended them or not. This formalization directs the system not to recommend items that are likely to be bought whether recommended or not, but rather to recommend items whose likelihood of being purchased is maximized if they are recommended. This approach might lead to a low acceptance rate of recommendations. This, by itself, is not problematic, as our goal is to maximize the accumulated rewards, not to obtain a high rate of accepted recommendation. However, one can argue that if recommendations are seldom followed, users will start ignoring them altogether, making the overall benefit 0. Our model does not capture such effects and one way to remedy this possible problem is

to alter the reward function so as to provide a certain immediate reward for the acceptance of a recommendation. Another way to handle this problem is to use the lift to order the items that have some minimal probability of being bought.

Updating The Model Online

Once the Recommender system is deployed with its initial model, we will need to update the model according to the actual observations. One approach would be to use a form of reinforcement learning — methods that improve the model after each recommendation is made. Although this idea is appealing since such models would need little administration to improve, the implementation requires more calls and computations by the recommender system online, which will lead to slower responses and is therefore undesirable. A simpler approach is to perform off-line updates at fixed time intervals. The site need only keep track of the recommendations and the user selections and, say, once a week use those statistics to build a new model and replace it with the old one. This is the approach we pursued.

In order to re-estimate the transition function the following values need recording:

- $count(s = \langle x_1, x_2, x_3 \rangle, s' = \langle x_2, x_3, x_4 \rangle)$ — the number of times the user selected x_4 after previously selecting x_1, x_2, x_3.

- $count(s = \langle x_1, x_2, x_3 \rangle, x_4, s' = \langle x_2, x_3, x_4 \rangle)$ — the number of times the user selected x_4 after it was recommended.

Using these two values we can compute a new approximation for the transition function:

$$c_{in}^{new}(s, r, s \cdot r) = c_{in}^{old}(s, r, s \cdot r) + count(s, r, s \cdot r)$$

$$c_{total}^{new}(s, s \cdot r) = c_{total}^{old}(s, r, s \cdot r) + count(s, s \cdot r)$$

$$c_{out}^{new}(s, r, s \cdot r) = c_{out}^{old}(s, r, s \cdot r) + count(s, s \cdot r) - count(s, r, s \cdot r)$$

$$tr(s, r \in R, s \cdot r) = \frac{c_{in}^{new}(s, r, s \cdot r)}{c_{total}^{new}(s, s \cdot r)}$$

$$tr(s, r \notin R, s \cdot r) = \frac{c_{out}^{new}(s, r, s \cdot r)}{c_{total}^{new}(s, s \cdot r)}$$

where $c_{in}(s, r, s \cdot r)$ is the number of times the r recommendations was taken, $c_{out}(s, r, s \cdot r)$ is the number of times the user took item r even though it wasn't recommended and $c_{total}(s, s \cdot r)$ is the number of times the user took item r while being in state s regardless of whether it was recommended or not. Note that at this stage the constants α and β no longer play a role, as they were used merely to initialize the model.

In order to ensure convergence to an optimal solution, the system requires every path in the model to be traversed a large number of times. If the system always returns the best recommendations only, then most values for $count(s, r, s \cdot r)$ would be 0, since most items will not appear in the list of the best recommendations available. Therefore, the system needs to recommend non-optimal items occasionally in order to get counts for those items. This problem is widely known in computational learning as the exploration vs. exploitation tradeoff. The system needs to decide when to explore unobserved options and when to exploit the data it has gathered in order to get rewards. Thus, it seems appropriate to select some constant ϵ, such that recommendations whose expected value is ϵ-close to optimal will be allowed—for example, by following a Boltzmann distribution:

$$Pr(choose(r_i)) = \frac{\exp \frac{V(s \cdot r_i)}{\tau}}{\sum_{j=1}^{n} exp \frac{V(s \cdot r_j)}{\tau}}$$

with an ϵ cutoff — meaning that only items whose value is within ϵ of the optimal value will be allowed. The exact value of ϵ can be determined by the site operators. The price we pay for this conservative exploration policy is that we are not guaranteed convergence to an optimal policy. However, commercial sites are not willing to accept a system that returns irrelevant recommendations for a noticeable period of time just because it improves itself in the future. Thus, significant exploration is not likely to be allowed by site operators in practice.

Since the system recommends more than one item at each state, a reasonable solution is to allow larger values of ϵ for the recommendations near the bottom of the recommendation list. For example, the system could always return the best recommendation first, but show items less likely to be purchased as the second and third items on the list.

Another problem that arises in recommender systems in general is adjusting to changes in the site. In e-commerce sites, items get frequently added and removed, and user trends shift making the model inaccurate after a while. Gathering the statistics shown above and improving the model once enough data has been gathered should readjust the transition function to follow the changes in users tastes. Once new items are added, users will start buying them and positive counts for them will appear. At this stage, the system adds new states for these new items, and the transition function is expanded to express the transitions for these new states. Of course, at first, the system is not able to recommend those new items, and this is known as the "cold start" problem (Good *et al.* 1999) in recommender systems. Removed items however need to be explicitly removed by the site administrator from the model. Even though a removed item transition probabilities will slowly diminish, it is better to set it to 0 manually to avoid generating a recommendation for an item that cannot be bought.

Performance Issues

Most modern model-based algorithms implement some type of optimization mechanism making the worst case complexity analysis uninteresting. The Microsoft Commerce Server Predictor (herein, Predictor), for example, allows you to build models based on a sample of the input data and limit the number of items used during the build process. We allowed similar options such as building a model for the top-n observed items and limiting the number of initialized states, and finally — optimizing the model for online recommendations, removing all the irrelevant data for that task.

It is however interesting to examine the amount of time required to build a model and the memory needed to hold such a model, as well as the number of predictions generated per second, as these parameters affect the performance of the recommender system in real e-commerce sites. We ran experiments on models differing by the number of items in the test data set thus generating different model sizes, build time and latency for predictions.

We compared our model to five *non*-sequential Predictor models. The Predictor enables you to limit the number of items for which decision trees are built. Using this feature, the number of items used in the models was set; we did not change the data set, just modified the "Input attribute fraction" and "Output attribute fraction" parameters to achieve the proper number of elements. Setting the "Input attribute fraction" parameter to 0.1, notifies the model build process to use only the top 10 percent observed items for building the model. The "Output attribute fraction" parameter sets the number of decision trees to be built. Similar methods are used by our MDP models to filter a portion of the items used when the model is built. Those models do not have the exact number of items as the MDP models since the Predictor method for selecting the number of items is different from our method, making it difficult to adjust the exact number. We allowed a maximal difference of 0.1 between the two models always in favor of the Predictor. Note that by using the *non*-sequential data for the Predictor, we provided it with an advantage, since models built from sequential data are slower to build, require more memory and supply less predictions per second.

number of items	15231	2661	1142	354	86
MDP	112	63	58	41	16
Predictor-NS	3504	631	177	80	25

Table 2: Required time (seconds) for model building.

Table 2 shows the time needed to build a new model based on the transactions data set used in our experiments for models. The Predictor value in this and in the following tables is for the most accurate Predictor model. We note that build time is the least important parameter when selecting a recommender system, as model building is an off-line task executed at long time intervals (say once a week at most) on a machine that does not affect the performance of the site. That being said, we note that our models are built faster than the corresponding Predictor models.

number of items	15231	2661	1142	354	86
MDP	138	74	55.7	33.3	11.4
Predictor-NS	50.1	26	25	22.3	18

Table 3: Required memory (MB) for building a model and generating recommendations.

Table 3 shows the amount of memory needed to build and store a model in MB. This parameter is more important, since using a large amount of memory causes the system to

slow down considerably. This is less important while building the model but is crucial on the web server holding the model for predictions where free space is usually used for caching ASP pages making the site responses quicker. Our models perform worse in this aspect. This is because Predictor models have a decision tree built for each item, while we have a state for sequences of items. Even though we used several optimization techniques mentioned above (e.g. not holding all states explicitly), the number of states far exceeds the number of decision trees. Many papers have been written discussing smart representations of the state and action space and the transition and value functions that could be leveraged for decreasing the needed system space (e.g., see (Koller & Parr 1999)).

number of items	15231	2661	1142	354	86
MDP	250	277	322	384	1030
Predictor-NS	23	74	175	322	1000

Table 4: Recommendations per second.

Table 4 shows the number of recommendations generated per second by the recommender system. This measure is crucial for online sites as it encapsulates the latency caused by waiting for the system to add recommendations to the page. If this latency is noticeable, no reasonable site administrator will use the recommender system. Modern sites are visited by large numbers of users every day. Generating a recommendation list for every user multiple times (say, every time she enters the shopping cart page or views an item description page) can cause the system to slow down. The Commerce Server Predictor sets the boundary at a minimum of 30 recommendations per second and an average of 50 recommendations. Our experiments show them to get far superior results except when the number of items is large. Still, our models perform better than the equivalent Predictor ones on most cases since we do almost no computations online; while predicting, the model simply finds the proper state and returns the state's pre-calculated list of recommendations. We note, though, that in order to decrease the memory requirements we could be forced to move to a different representation of the policy that might lead to some computational cost and hence, to decreased performance.

Overall we conclude that our approach is competitive with the leading commercial implementation of recommender systems — it builds faster, supplies fast recommendations, and can be adapted to require a reasonable amount of memory. Finally, it is worth noting that the predictive accuracy of our predictive model, discussed in (Shani, Brafman, & Heckerman 2002), is better than that of the *sequential* Predictor models.

Conclusions And Future Work

This paper describes a new model for recommender systems based on an MDP. A system based on this model is currently incorporated into the online book store Mitos (www.mitos.co.il). Users can receive recommendations in two different locations: users looking at the description of

a book are presented with a list of recommendations based solely on this book. Users adding items to the shopping cart receive recommendations based on the last k items added to the cart ordered by the time they were added – an example of the latter is shown in Figure 1 where the three book covers at the bottom are the recommended items. Every time a user is presented with a list of recommendations on either page, the system stores the recommendations that were presented and records whether the user took a recommended item. Once a week, an automated process is run to update the model given the data that was collected during the week, as described in this paper.

Figure 1: Recommendations in the shopping cart web page.

Our work presents one of a very small number of commercially deployed recommender systems to emerge from academic research, and one of a few examples of commercial systems that use MDPs. To provide the kind of performance required from an online commercial site, we used various approximations and, in particular, made heavy use of the special properties of our state space and its sequential origin. While the applicability of these techniques beyond recommender systems is not clear, it represents an interesting case study of a successful real system. Moreover, the sequential nature of our system stems from the fact that we need to maintain history of past purchases in order to obtain a Markovian state space. The need to record facts about the past in the current state arises in various domains, and has been discussed in a number of papers on handling non-Markovian reward functions (e.g., see (Bacchus, Boutilier, & Grove 1996; Thiebaux, Kabanza, & Slaney 2002)). Another interesting technique is our use of off-line data to initialize a model that can provide adequate initial performance.

In the future, we hope to improve our transition function on those states that are seldom encountered using generalization techniques, such as skipping and clustering, that are similar to the ones we employed in the predictive Markov chain model described in (Shani, Brafman, & Heckerman 2002). Other potential improvements are the use of a partially observable MDP to model the user. As a model, this is more appropriate than an MDP, as it allows us to explicitly model our uncertainty about the true state of the user. In fact, our current model can be viewed as approximating a particular POMDP by using a finite – rather than an unbounded – window of past history to define the current state. Of course, the computational and representational overhead of POMDPs are significant, and appropriate techniques for overcoming these problems must be developed.

References

Bacchus, F.; Boutilier, C.; and Grove, A. 1996. Rewarding behaviors. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*.

Balabanovic, M., and Shoham, Y. 1997. Combining content-based and collaborative recommendation. *Communications of the ACM* 40(3).

Bellman, R. E. 1962. *Dynamic Programming*. Princeton University Press.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic dynamic programming with factored representations. *Artificial Intelligence* 121(1–2):49–107.

Claypool, M.; Le, P.; Waseda, M.; and Brown, D. 2001. Implicit interest indicators. In *ACM Intelligent User Interfaces Conference*, 33–40.

Good, N.; Shefer, J. B.; Konstan, J. A.; Borchers, A.; Sarwar, B.; Herlocker, J.; and Riedl, J. 1999. Combining collaborative filtering with personal agents for better recommendations. In *Proc. AAAI'99*.

Heckerman, D.; Chickering, D. M.; Meek, C.; Rounthwaite, R.; and Kadie, C. 2000. Dependency networks for inference and collaborative filtering and data visualization. *Journal of Machine Learning Research* 1:49–75.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. MIT Press.

Kadie, C. M.; Meek, C.; and Heckerman, D. 2002. Cfw: A collaborative filtering system using posterios over weights of evidence. In *Proc. 18th Conf. on Uncertainty in AI (UAI'02)*, 242–250.

Koller, D., and Parr, R. 1999. Computing factored functions for policies in structured mdps. *Proc. 16th International Joint Conference on AI (IJCAI-99)* 1332–1339.

Meuleau, N.; Hauskrecht, M.; Kim, K.-E.; Peshkin, L.; Kaelbling, L. P.; Dean, T.; and Boutilier, C. 1998. Solving very large weakly coupled markov decision processes. In *Proc. AAAI'98*, 165–172.

Mooney, R. J., and Roy, L. 2000. Content-based book recommending using learning for text categorization. In *Proc. 5th ACM Conference on Digital Libraries*, 195–204.

Puterman, M. 1994. *Markov Decision Processes*. New York: Wiley.

Resnick, P., and Varian, H. R. 1997. Recommender systems. *Communications of the ACM* 40(3):56– 58.

Resnick, P.; Iacovou, N.; Suchak, M.; Bergstrom, P.; and Riedel, J. 1994. Grouplens: An open architecture for collaborative filtering of netnews. In *Proc. ACM 1994 Conf. on Computer Supported Cooperative Work*, 175–186.

Schafer, J. B.; Konstan, J.; and Riedl, J. 1999. Recommender systems in e-commerce. In *Proc. ACM E-Commerce 1999 Conf.*, 158–166.

Shani, G.; Brafman, R. I.; and Heckerman, D. 2002. An mdp-based recommender system. In *Proc. 18th Conf. on Uncertainty in AI (UAI'02)*, 453–460.

Thiebaux, S.; Kabanza, F.; and Slaney, J. 2002. Anytime state-based solution methods for decision processes with non-markovian rewards. In *Proc. 18th Conf. on Uncertainty in AI (UAI'02)*, 501–510.

Zimdars, A.; Chickering, D. M.; and Meek, C. 2001. Using temporal data for making recommendations. In *Proc. 17th Conf. on Uncertainty in AI*, 580–588.