Planning with Conceptual Models Mined from User Behavior

Thomas J. Walsh and Michael L. Littman
Department of Computer Science, Rutgers University, 110 Frelinghuysen Rd., Piscataway, NJ 08854
{thomaswa,mlittman}@cs.rutgers.edu

Abstract

This work considers the problem of learning to perform efficient sequential queries from traces of user behavior. We define a restricted conceptual modeling language and outline how to learn models in this language from behavior traces. We then show how to plan instantiations of concepts in such models and discuss how this planning is affected by different optimality criteria. We examine simple example domains as well as a real world domain involving Amazon web services.

Introduction

In this work, we consider the problem of an agent learning to perform an efficient sequence of "query operations" (e.g. against a database or web services) in a complex domain from traces of user(s) performing queries in the same domain. In the real world, the increasing availability of web services for performing composite tasks such as making travel arrangements and participating in auctions makes this problem of particular interest to researchers. The use of learning agents in these environments is particularly appealing as the diversity and changing functionalities that characterize such domains make building and maintaining "hard-coded" agents prohibitive. Although agents could potentially learn about these environments through direct experience, the complexity of learning representations of these domains without guidance is potentially burdensome. Instead, we focus here on a system that is provided with traces of user queries (such as the one in Figure 1a involving a simple flight reservation) and a simple modeling language to develop a corresponding conceptual model. The goal of the system is to create and execute a plan (which may be different than any instantiated plan in the example traces) that results in a qualitatively judged instantiation of a goal concept (e.g. reserve the *cheapest* flight). We describe an algorithm, MOPLEX (MOdeling, PLanning, and EXecution), that builds a conceptual model consistent with the trace domain using a restricted modeling language and then plans a sequence of queries (which may require parameters from the agent's knowledge base) that will result in the instantiation of the desired "goal concept".

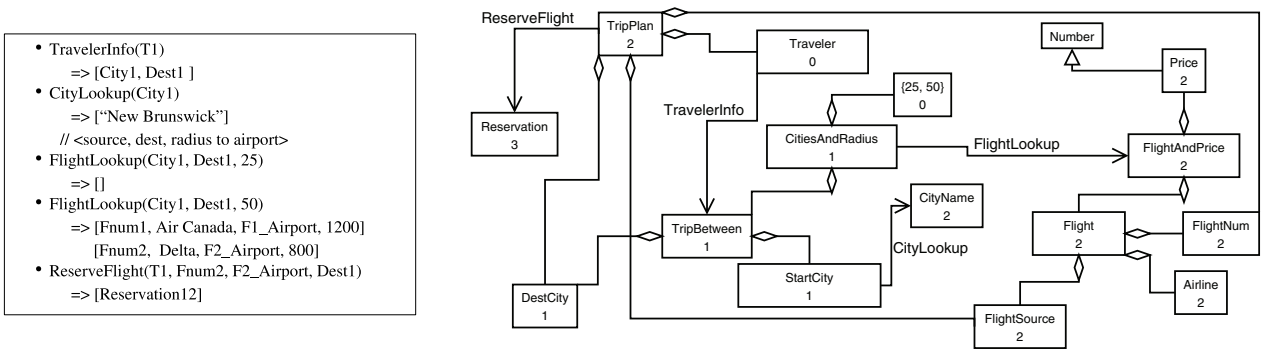
In this work, we give a detailed description of MOPLEX's modeling stage and identify our modeling language as a restricted subset of a particular Description Logic. We then discuss varying definitions of "optimality" for the planning stage and how our choice of planning criterion affects both behavior and computation. Throughout these discussions, we refer to several small-scale "toy" examples, but at the end, we show MOPLEX successfully modeling a complex real world scenario involving the Amazon web services [1].

Related Work

Our approach is certainly not the first to combine the ideas of conceptual modeling and planning. For instance, previous research has investigated using Description Logics (languages often used for conceptual modeling) with action formalisms (Baader *et al.* 2005) and extending relational representations to the reinforcement-learning paradigm (Dzeroski, De Raedt, & Driessens 2001). The problem we address in this work differs because we wish to learn from user traces and are concerned with acquiring and choosing specific instances of a concept based on some qualitative criteria (e.g. the *cheapest* flight), rather than being given the goal instance beforehand and planning at a completely conceptual level. The subject of using a concept model learned from an expert plan to induce a general rule-based policy has also previously been addressed (Martin & Geffner 2004). Although we are also learning a conceptual model based on user behavior, our approach builds a model of the dynamics of its environment and can therefore discover better policies than the ones used in the trace. Our approach is also robust to action failure, has defined rules for choosing between multiple instances of a concept, and uses a more restricted set of constructors than this related work.

The problem we have described is also closely related to the field of workflow induction (van der Aalst & Weijters 2004), which attempts to model real-world processes by analyzing user behaviors. Almost all the works in this field have employed a variant of the Petri Net (Murata 1989) data structure as their base model, including Workflow-Nets (van der Aalst, Weijters, & Maruster 2004), Time Interval Petri Nets (Bulitko & Wilkins 2005), and ADONIS Petri Nets (Herbst 2000). These different data structures provide

[1] http://www.amazon.com/aws



(a) (b)

Figure 1: (a) A sample trace of user queries. (b) The mined conceptual model, along with instantiation costs. Labels were chosen for human readability and were not automatically generated.

varying levels of expressiveness, and often require powerful machine-learning techniques such as genetic algorithms, decision trees, and statistical data mining to learn the “rules” that govern the behavior of the workflow. While this work in learning Petri Nets has garnered several interesting empirical results, such as accurate prediction of fire spreading patterns (Bulitko & Wilkins 2005), only a few theoretical results have been produced (Agrawal, Gunopulos, & Leymann 1998; van der Aalst, Weijters, & Maruster 2004). The focus of this paper differs from the previous workflow induction works in that it is concerned with modeling structured data with a Description Logic (as opposed to the largely propositional Petri Net), considers only a simple set of firing rules (ANDs of parts or a query execution), and considers the problems of planning and executing using these models after they have been learned.

Modeling

We first define the modeling language used by MOPLEX. We then describe how to learn models of this form from user traces.

Conceptual Syntax and Semantics

Conceptual models, which describe environments in terms of *concepts* and *relations*, are used in many facets of computer science (Mylopoulos 1998). The complexity of reasoning with such models varies greatly with the set of constructors allowed for building concepts. In MOPLEX, we used a constructor set corresponding to a restricted *subset* of the Description Logic $\mathcal{ALCO}$ (Baader *et al.* 2003) (constructors for conjunction ($\sqcap$), role range restrictions ($\forall$), and special versions of set enumeration (*oneOf*) and a version of part-whole ($\oplus$)). Formally, the modeling language employed by MOPLEX can be written as:

$$C \leftarrow T|C_p|C \sqcap C|(\forall R.C)|oneOf\{z_i...z_n\}|\oplus C$$
$$R \leftarrow R_p|hasPart_i$$

where $z_i...z_n$ are grounded numbers (that is we allow concepts based on individuals only when those individuals are numbers). The special $\oplus$ constructor can be translated logically as:

$$C \equiv \oplus C' \equiv \exists hasPart_i \sqcap \forall hasPart_i.C'$$

In other words, a concept C has part C' if the existence of its $hasPart_i$ role is required and all of the fillers of this role must be of type C'. Notice that such a constructor has a different meaning than intersection (which is often intuitively confused with the notion of parts) because intersection defines a concept that is a refinement of (subsumed by) other concepts (e.g. Father $\equiv$ Parent $\sqcap$ Male).

While this language can capture a domain at a conceptual level (“Employee” rather than “Alice”), we will find it necessary throughout this work to reason about specific *instances* of a concept. An *interpretation* of a concept, denoted C^I, is the set of instances from the interpretation of a domain, Δ^I, that are described by the concept C ($C^I \subseteq \Delta^I$). We say a concept C *covers* an instance i iff $i \in C^I$. In this work, we will often need to pick a specific instance from this interpretation and bind it to the concept for the purpose of planning. We will refer to a concept that has participated in such a binding as *instantiated*.

Other than the special part-whole relationship, all other relations in the modeling language above can be thought of as “query relationships” that link the input and output concepts for the queries in the user trace. We settled on this modeling language because it sufficiently captured our example domains while maintaining tractability in the operations needed to build and plan using the resultant model. We mark concepts that are defined solely by the “one-of” constructor as *enumerable* and those that have one or more parts defined by the “one-of” constructor as *partially enumerable*. In order to accomplish our goal of finding the “best” instance of a goal concept, the base conceptual model is augmented using *rules* mined from the user trace. While the system used in the running example uses only a simple rule-mining algorithm (decide if bigger or smaller numbers are preferred), more complex data-mining algorithms could be used to mine richer rules.

Model Construction

The construction of the model itself is accomplished using the following algorithm. Throughout this delineation we will refer to the example in Figure 1.

1. Begin with a concept library containing all primitive concepts that define known instances (Traveler in our exam-

ple) and a special *Number* concept.

2. Check the concept library for a concept whose *parts* cover the *non-numeric* parameters of the query. The system assumes that the only parameters for a query it will see will have come from another query's output, or be numeric. If a covering concept exists, call it the "input concept". Otherwise, if the query already has a known "input concept" that does not match the current parameters (a possibility if many prior queries can produce the input concept), then "merge" the parts of the two concepts that are common to this query. If neither of the above scenarios apply, then create a new input concept with a part for each input parameter.

   Example: ReserveFlight(T1, Fnum2, F2_Airport, Dest1) requires us to have a concept that covers $\{\oplus Traveler \sqcap \oplus FlightNum \sqcap \oplus FlightSource \sqcap \oplus DestCity\}$. Since no such concept exists, the concept *TripPlan* is created with exactly that definition.

3. If the query parameters contain a number and the input concept is not marked as *partially-enumerable* then create a new input concept defined as *oldInputConcept* $\sqcap\oplus$ *oneOf*$\{n\}$ where n is number and $\oplus$ is the "part-of" constructor. Mark this new input concept as *partially-enumerable* and the "oneOf" concept as *enumerable*. If the corresponding concept already exists, add the numeric parameter to the enumeration list if it does not already reside there.

   Example: FlightLookup(City1, Dest1, 25). From prior queries we already have the concept $TripBetween \equiv (\oplus StartCity \sqcap \oplus DestCity)$ but the numeric attribute (25) necessitates the construction of a different input concept $CitiesAndRadius = \oplus TripBetween \sqcap \oplus oneOf\{25\}$.

4. If the output of the query was not empty, check the concept library for a concept whose parts cover the *non-numeric* instances in the output. If no such concept exists, create a new one defined with parts corresponding to the output instances' concepts (which may themselves force the creation of new primitive concepts since we may not have seen them before). This check is essentially the same as the one used in the input concept rule, but the new concept parts may not already be associated with concepts and the resulting concept is marked as the *output concept*.

   Example: TravelerInfo(T1) → [City1, Dest1]. Since City1 and Dest1 are not instances of any known concept, we create new primitive concepts to cover them, *StartCity* and *DestCity* and create the new concept $TripBetween \equiv \oplus StartCity \sqcap \oplus DestCity$ as the output concept.

5. If the output instance contains a number, make a new output concept with two parts, the previous output concept and a hyponym of the *Number* concept. This new concept is then designated as the output concept.

   Example: FlightLookup(City1, Dest1, 50) → [Fnum1, AirCanada, F1_Airport, 1200].... The previous step, would have created the *Flight* concept, which covers the non-numeric parts, but the "1200" attribute needs to be associated not with a specific flight, but with the association between the flight and the traveler (it's a price that may change). To model it, we create the concept $FlightAndPrice \equiv \oplus Flight \sqcap \oplus Price$ with $Price \sqsubseteq Number$.

6. If the input/output concept pair is new, add a restriction on the fillers of the role name associated with the query to the input concept ($\forall$ *queryName.OutputConcept*).

   Example: FlightLookup(City1, Dest1, 50) → [...]. The input and output concepts of this query have been determined by previous rules to be *CitiesAndRadius* and *FlightAndPrice*, respectively. Now, we augment the *CitiesAndRadius* definition to include the FlightLookup role, that is $CitiesAndRadius \equiv \oplus TripBetween \sqcap \oplus oneOf 25, 50 \sqcap \forall FlightLookup.FlightAndPrice$.

7. Finally, if multiple instances of the *output* concept exist, then look ahead in the trace to see which one of these instances are used first. Create a simple rule based on the numeric attributes of this concept (prefer either small or large numbers) that augments the output concept, allowing MOPLEX to choose between multiple instances. These rules are outside the scope of the conceptual modeling language we have described and should be thought of as augmenting the concept model, rather than being an actual part of the modeling engine.

   Example: FlightLookup(City1, Dest1, 50) → [Fnum1, AirCanada, F1_Airport, 1200], [Fnum2, Delta, F2_Airport, 800]. There are two instances of the output concept *FlightAndPrice*, so we need a rule that would allow a planning agent to decide between them. Looking ahead in the trace, we see that Fnum2 and F2_Airport are used next before any of the other associated pieces of either instance, so we conclude that the second instance is preferable to the former, and augment the concept definition of *FlightAndPrice* with a rule "favor instances with smaller fillers of the Price part" (that is, choose the cheapest flight first).

The end result of this modeling in our simple example is depicted in the UML diagram in Figure 1b.

Planning

We now consider three different types of planning criteria that could be used to instantiate a goal concept in a conceptual model mined by MOPLEX. These techniques should be thought of as modular components that can be used depending on the criteria the user wishes to minimize. The first criterion assumes that each query will succeed, but that every query has a cost, and attempts to minimize the total cost. The other two criteria considered are probabilistic in nature, one that tries to minimize the expected cost given that queries may succeed or fail, and one that simply tries to minimize the probability of failing at all. In all cases, we assume that some non-empty set of concepts (e.g. Traveler) is already instantiated and that the goal concept is known.

Minimizing Query Costs

The first planning case we consider is one where every query has a cost and query failure is not anticipated. If all query

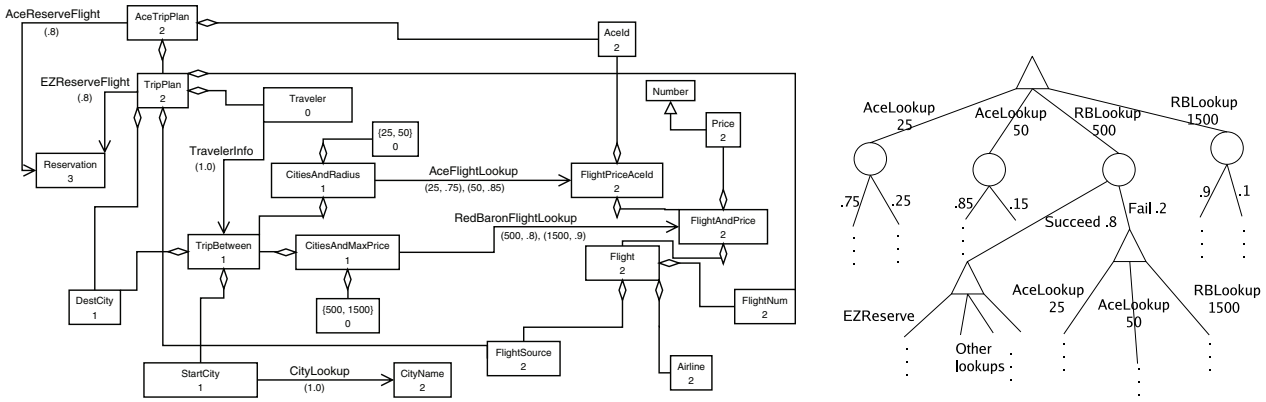


Figure 2: (a) A probabilistic travel-reservation domain. Query success probabilities are conditioned on their numeric parameters. The values in the concept boxes are from the non-probabilistic "Minimize Query Cost" criterion. (b) A portion of the Expectimax tree corresponding to this environment. In this domain, Expectimax will choose to use the AceFlightLookup(...50) because the success of this query will provide two chances to complete the task, making the expected cost of this query smaller than RedBaronLookup(...1500), which has a higher single step success probability. The "minimize probability of any failures" criterion leads to the latter choice.

costs are equal, planning in this case reduces to finding the fastest way (smallest number of queries) to instantiate the goal concept. In either case, this task can be accomplished by viewing the conceptual model itself as a graph and considering a moving "frontier" of instantiated nodes starting with only the initially instantiated primitive concepts. Effective planning in this setting requires a set of rules for how to instantiate nodes in the graph and a notion of cost for instantiating these nodes. In MOPLEX, the costs are associated with the three edge "types":

1. **Whole to Part:** If a concept is instantiated then all its parts and conjoined primitive components can be instantiated with cost 0.

2. **Parts to Whole:** If all the parts and conjoined primitive components defining a concept are either instantiated or *enumerable*, then that concept itself can be instantiated with cost 0.

3. **Queries:** A concept at the end of a "query edge" can be instantiated by adding that query to the plan as long as the concept the edge emanates from is marked as instantiated. The cost added to the plan by this edge is equal to the cost of the query.

The query cost structure could easily be conditioned on the numeric parameters as well (FlightLookup(...25) costs less than FlightLookup(...50)), which would allow the agent to choose parameters as well as queries. For simplicity, we ignore this aspect, and simply assume the agent chooses numbers in the order they appear in the enumeration list, though we deal with the parameter choice issue in subsequent probabilistic planning settings. With the cost structure laid out above, planning is done using a variant of breadth-first search in the concept lattice itself. Examples of total costs calculated using single step query costs of 1 for all queries in the travel-reservation domain appear in Figure 1b.

Minimizing Expected Cost

Our next two planning criteria are for the probabilistic planning setting, where each combination of query numeric input (e.g. FlightLookup, 25) is annotated with a probability that such a query call leads to a successful instantiation of the output concept. That is, if a call to a query with a given numeric parameter returns *null*, we consider the previous call to be a failure. Otherwise it is a success. We assume once a query succeeds or fails with a given set of parameters, the outcome will always be the same with that set. Since we are conditioning probabilities based on the parameters, we allow the agent in the probabilistic setting to choose numeric parameters for queries (from concepts built with the *oneOf* constructor). The success probabilities can easily be mined from user traces, giving us a probabilistic version of the conceptual model. Figure 2(a) provides an example of such a model based on an expansion of the earlier domain where there are now two ways to look up a flight (the Ace query or the Red Baron query, representing two different services). There are also two ways to reserve a flight (Ace and EZ) but only flights looked up through Ace can be booked by Ace. Each of those queries has a certain success probability given their numeric parameters, as depicted in the diagram. We assume in this work that the outcome of a query is independent of other queries, an assumption that will probably be violated in real life, but whose effects are somewhat mitigated by the conditioning on parameters (which are indirectly linked to future success). Conditioning probabilities on query/number pairs also has the effect of encouraging the system to seek the best parameters as well as the best queries.

In this probabilistic setting, using the same cost structure as defined in the previous section, we now consider the problem of minimizing expected cost. Unfortunately, this setting forces us to consider all combinations of query success and failure possibilities. One way to look at this problem is to consider a possible worlds model, that is, a set of possible

interpretations of the conceptual model, one for each possible setting of success or failure on each query relation. When viewed in this light, the problem lends itself to the search algorithm, Expectimax (Russell & Norvig 2002). Expectimax works by building out a tree of *choice* and *chance* points. A piece of such a tree corresponding to our example is provided in Figure 2(b). At the choice points, the agent must select from a set of actions, whose expected values are computed from the bottom (termination) up, corresponding to our agent choosing which query to use and with which numeric parameters. At the chance nodes, a probabilistic event occurs that shapes the agent's expected value from that point forward. This event corresponds to the success or failure of a query as defined above. In terms of the possible worlds model, these chance points determine which of the possible worlds the agent really is in. Expectimax can be used to determine the minimal expected cost strategy for the instantiation of a conceptual model with parameters as defined as above. In the probabilistic travel-reservation domain, Expectimax's first attempt to flight lookup is an AceFlightLookup with a radius of 50. If this lookup succeeds, it will then have two chances to book this flight, keeping the expected cost low. We note that this strategy does not maximize the probability of zero failures, a topic we will return to shortly. Unfortunately, the runtime of Expectimax is exponential in the size of the model, and therefore not practical for large domains. We now investigate a more practical, though less thorough, probabilistic criterion.

Minimizing Probability of Any Failure

In light of the computational burden mentioned above, we turn our attention to a computationally easier planning problem in the probabilistic setting, determining the strategy that is least likely to see *any* failures. That is, we assume a single failure truncates the search tree above, and do not consider decision making after this failure. While this criterion does not necessarily lead to intuitively optimal behavior, it does lend itself to an efficient algorithm for generating such policies. Specifically, one can use a variant of breadth first search that keeps track of the maximal probability (maximized over numeric parameters) at each concept that it plans to instantiate. This approach generally ignores the cost of queries, since query failure is assumed to lead to an infinite cost trap, but the smallest cost path can be used as a tie breaker in this scenario. In the probabilistic travel-reservation domain, this criteria will lead the agent to use Red Baron FlightLookup with a max-price of 1500 (a different choice then Expectimax), because this choice will maximize the probability of having no failures.

Execution

In the execution phase, MOPLEX attempts to run the plan produced in the previous phase. During execution, the system tries each of the planned queries, keeping track of the instantiated concepts, reacting when a query comes back empty, and choosing between instances when multiple instances of a concept exist. Query failure may require *replanning* using an updated model and whatever planning criteria were used in the original planning phase. MOPLEX chooses between multiple instances of a concept using the simple rules trained in the modeling phase or, in the case of instantiating an *enumerable* part, chooses the first number in the list not yet tried on this instantiation of the node (planning with query costs) or using the best probabilistic choice of parameter (probabilistic planning criteria). It then continues executing the queries in the plan using the chosen instance. Finally, if a planned query comes back empty, MOPLEX uninstantiates the nodes that brought it to this point in "causal order". That is, it uninstantiates each concept on the path to the concept that caused the failure, also uninstantiating any parts of such concepts along the way. This process continues until MOPLEX backtracks to a point where it chose one instance over another, either by rule or by choosing a number from an enumeration list and then replanning.

Web Services: An Application

As mentioned in the introduction, the recent proliferation of web services and other internet API's has provided a fertile experimental landscape for agents, like MOPLEX, that can model and reason about parameterized queries. MOPLEX's modeling engine, in particular, makes it well suited for the ever changing landscape of the world wide web, where changes often occur faster than adequate conceptual models are made available. Therefore, we considered a small problem involving the Amazon Web Services. The only instantiated concept at the onset is an email address for an Amazon user. The goal of the system is to look up the corresponding user's wish list, pick the item which the user wants in the highest quantity, and get these items at the cheapest cost. We used the simple Query Cost criterion in this setting and created sample traces by hand. The learned conceptual model for this domain, with planned costs, is provided in Figure 3 (some non-essential parameters are omitted for simplicity of presentation). The path MOPLEX chooses in this scenario bypasses the unnecessary CustomerSearch and CustListLookup queries. The success of MOPLEX in this real world domain motivates future work on more complex problems using Web Services.

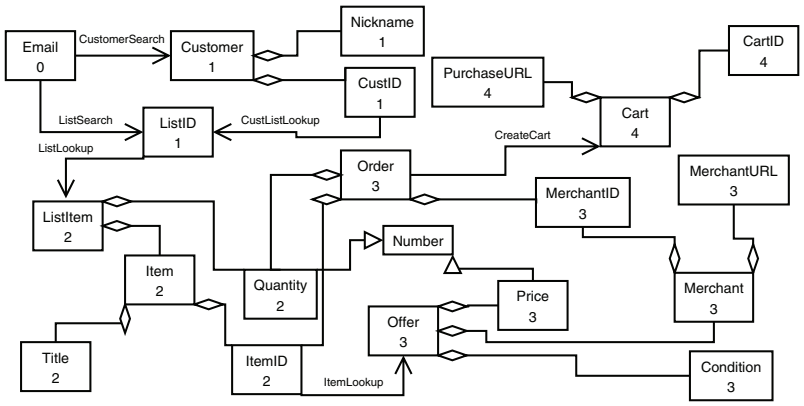
Conclusions and Future Work

In this work, we have proposed a system, MOPLEX, for learning a conceptual model of a domain from user traces and then planning and executing tasks in that domain. We have couched our system in the literature on learning conceptual models, particularly the field of Workflow Induction/Process Mining. We have given an algorithm for mining a conceptual model expressed in a particular Description Logic from traces. We discussed several criteria for evaluating plans in such domains, with and without probabilistic effects. After evaluating MOPLEX in two simple travel arrangement scenarios, we showed the ability of MOPLEX to model a real world scenario involving a series of calls to Amazon Web Services.

The work we have presented here is still, in large part, preliminary. There are various extensions that we are considering to enhance the expressiveness and applicability of

- CustomerSearch(Jane@DoeCorp.com)
  => [ID577, Janey]
- CustListLookup(ID577)
  => [ID345]
- ListSearch(Jane@DoeCorp.com)
  => [ID345] //smarter search
- ListLookup(ID345)
  => [[BeanBagChair, ID111] ,3]
  [[DeluxeOfficeChair, ID222] ,1]
- ItemLookup(ID111)
  => [35, [IDM2, www.goodDeal.com], new]
  [55, [IDM3, www.badDeal.com], new]
- CreateCart(ID111, IDM2 ,3)
  => [www.amazon... , IDC543]

(a)



(b)

Figure 3: An example (a) trace and (b) conceptual model mined from users browsing and buying items off of Amazon WishLists. The planning costs use the "minimize query cost" criterion. Labels were chosen for human readability, though in this example, suitable labels could potentially have been mined from the XML documents passed by the Web Services.

the system. First, we would like to investigate using more expressive modeling languages than the current constructor set. The current restricted constructor set was chosen because it models all the problems we have considered so far and still allows for efficient conceptual reasoning (i.e. subsumption reasoning). In addition, we plan to investigate more powerful data mining techniques to uncover more complex preference rules than the simple number based system used now (e.g. judging offers on their price and condition). We will also investigate more planning criteria than the three investigated in this work, again focusing on the tradeoff between "optimality" and computation. Particularly, we wish to consider more moderate truncations of the Expectimax tree than the "minimize probability of any failure" criteria, which essentially truncated the tree after every failure. Finally, we are looking into deploying MOPLEX in more complex real world domains using web services.

Acknowledgements

We would like to acknowledge the support of DARPA IPTO in this effort as well as the Integrated Learning project team, whose examples motivated our research in this area. Finally, we thank Alex Borgida for many helpful conversations on conceptual modeling.

References

Agrawal, R.; Gunopulos, D.; and Leymann, F. 1998. Mining process models from workflow logs. In *International Conference on Extending Database Technology*, 469–483. London, UK: Springer-Verlag.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

Baader, F.; Lutz, C.; Milicic, M.; Sattler, U.; and Wolter, F. 2005. Integrating description logics and action formalisms: First results. In *AAAI-05*.

Bulitko, V., and Wilkins, D. 2005. Machine learning for time interval Petri nets. In *Australian Joint Conference on Artificial Intelligence*, 959–965. Springer-Verlag.

Dzeroski, S.; De Raedt, L.; and Driessens, K. 2001. Relational reinforcement learning. *Machine Learning* 43(1):7–52.

Herbst, J. 2000. A machine learning approach to workflow management. In *European Conference on Machine Learning*, 183–194. London, UK: Springer-Verlag.

Martin, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Journal of Applied Intelligence* 20:9–19.

Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4):541–580.

Mylopoulos, J. 1998. Information modeling in the time of the revolution. *Information Systems* 23(3–4):127–155.

Russell, S. J., and Norvig, P. 2002. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition.

van der Aalst, W. M. P., and Weijters, A. J. M. M. 2004. Process mining: A research agenda. *Computers in Industry* 53(3):231–244.

van der Aalst, W.; Weijters, T.; and Maruster, L. 2004. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering* 16(9):1128–1142.