In *The 20th International FLAIRS Conference* (FLAIRS-07),
Key West, Forida, May 2007.

Guiding Inference with Policy Search Reinforcement Learning

Matthew E. Taylor
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
mtaylor@cs.utexas.edu

Cynthia Matuszek, Pace Reagan Smith, and Michael Witbrock
Cycorp, Inc.
3721 Executive Center Drive
Austin, TX 78731
{cynthia,pace,witbrock}@cyc.com

Abstract

Symbolic reasoning is a well understood and effective approach to handling reasoning over formally represented knowledge; however, simple symbolic inference systems necessarily slow as complexity and ground facts grow. As automated approaches to ontology-building become more prevalent and sophisticated, knowledge base systems become larger and more complex, necessitating techniques for faster inference. This work uses reinforcement learning, a statistical machine learning technique, to learn control laws which guide inference. We implement our learning method in ResearchCyc, a very large knowledge base with millions of assertions. A large set of test queries, some of which require tens of thousands of inference steps to answer, can be answered faster after training over an independent set of training queries. Furthermore, this learned inference module outperforms ResearchCyc's integrated inference module, a module that has been hand-tuned with considerable effort.

Introduction

Logical reasoning systems have a long history of success and have proven to be powerful tools for assisting in solving certain classes of problems. However, those successes are limited by the computational complexity of naïve inference methods, which may grow exponentially with the number of inferential rules and amount of available background knowledge. As automated learning and knowledge acquisition techniques (e.g. (Etzioni *et al.* 2004; Matuszek *et al.* 2005)) make very large knowledge bases available, performing inference efficiently over large amounts of knowledge becomes progressively more crucial. This paper demonstrates that it is possible to learn to guide inference efficiently via *reinforcement learning*, a popular statistical machine learning technique.

Reinforcement learning (Sutton & Barto 1998) (RL) is a general machine learning technique that has enjoyed success in many domains. An RL task is typically framed as an *agent* interacting with an unknown (or under-specified) environment. Over time, the agent attempts to learn when to take actions such that an external reward signal is maximized.

Many sequential choices must be made when performing complex inferences (e.g. what piece of data to consider, or what information should be combined). This paper describes work utilizing RL to train an *RL Tactician*, an inference module that helps direct inferences. The job of the Tactician is to order inference decisions and thus guide inference towards answers more efficiently. It would be quite difficult to determine an optimal inference path for a complex query to support training a learner with classical machine learning. However, RL is an appropriate machine learning technique for optimizing inference as it is relatively simple to provide feedback to a learner about how efficiently it is able to respond to a set of training queries.

Our inference-learning method is implemented and tested within ResearchCyc, a freely available[1] version of Cyc (Lenat 1995). We show that the speed of inference can be significantly improved over time when training on queries drawn from the Cyc knowledge base by effectively learning low-level control laws. Additionally, the RL Tactician is also able to outperform Cyc's built-in, hand-tuned tactician.

Inference in Cyc

Inference in Cyc is different from most inference engines because it was designed and optimized to work over a large knowledge base with thousands of predicates and hundreds of thousands of constants. Inference in Cyc is sound but is not always complete: in practice, memory- or time-based cutoffs are typically used to limit very large or long-running queries. Additionally, the logic is "n^{th} order," as variables and quantifiers can be nested arbitrarily deeply inside of background knowledge or queries (Ramachandran *et al.* 2005).

Cyc's inference engine is composed of approximately a thousand specialized reasoners, called *inference modules*, designed to handle commonly occurring classes of problem and sub-problem. Modules range from those that handle extremely general cases, such as subsumption reasoning, to very specific modules which perform efficient reasoning for only a single predicate. An *inference harness* breaks a problem down into sub-problems and selects among the modules that may apply to each problem, as well as choosing follow-up approaches, pruning entire branches of search based on expected productivity, and allocating computational resources. The behavior of the inference harness is defined by a set of manually coded heuristics. As the complexity of the problem grows, that set of heuristics becomes more complex and more difficult to develop effectively by human evaluation of the problem space; detailed analysis of a set of test cases shows that overall time spent to achieve a set of answers could be improved by up to 50% with better search policy.

Cyc's inference harness is composed of three main high-level components: the Strategist, Tactician, and Worker. The Strategist's primary function is to keep track of resource constraints, such as memory or time, and interrupt the inference if a constraint is violated. A *tactic* is a single quantum of work that can be performed in the course of producing results for a query, such as splitting a conjunction into multiple clauses, looking up the truth value of a fully-bound clause, or finding appropriate bindings for a partial sentence via indexing. At any given time during the solving of a query, there are typically multiple logically correct possible actions. Different orderings of these tactics can lead to solutions in radi-

[1]http://research.cyc.com/

cally different amounts of time, which is why we believe this approach has the potential to improve overall inference performance. The Worker is responsible for executing tactics as directed by the Tactician. The majority of inference reasoning in Cyc takes place in the Tactician, and thus this paper will focus on speeding up the Tactician.

The Tactician used for backward inference in Cyc is called the Balanced Tactician, which heuristically selects tactics in a best-first manner. The features that the Balanced Tactician uses to score tactics are tactic type, productivity, completeness, and preference. There are eight tactic types which describes the kind of operation to be performed, such as "split" (i.e. split a conjunction into two sub-problems). *Productivity* is a heuristic that is related to the expected number of answers that will be generated by the execution of a tactic; lower productivity tactics are typically preferred because they reduce the state space of the inference. *Completeness* can take on three discrete values and is an estimate of whether the tactic is complete in the logical sense, i.e. it is expected to yield all true answers to the problem. *Preference* is a related feature that also estimates how likely a tactic is to return all the possible bindings. Completeness and Preference are mutually exclusive, depending on the tactic type. Therefore the Balanced Tactician scores each tactic based on the tactic's type, productivity, and either completeness or preference.

Speeding up inference is a particularly significant accomplishment as the Balanced Tactician has been hand-tuned for a number of years by Cyc programmers and even small improvements may dramatically reduce real-world running times. The Balanced Tactician also has access to heuristic approaches to tactic selection that are not based on productivity, completeness, and preference (for example, so-called "Magic Wand" tactics—a set of tactics which almost always fail, but which are so fast to try that the very small chance of success is worth the effort). Because these are not based on the usual feature set, the reinforcement learner does not have access to those tactics.

Learning

This section presents an overview of the reinforcement learning problem, a standard type of machine learning task. We next describe NEAT, the RL learning method utilized in this work, and then detail how NEAT trains the RL Tactician.

Reinforcement Learning

In reinforcement learning tasks, agents take sequential actions with the goal of maximizing a reward signal, which may be time delayed. RL is a popular machine learning method for tackling complex problems with limited feedback, such as robot control and game playing. Unlike other machine learning tasks, such as classification or regression, an RL agent typically does not have access to labeled training examples.

An RL agent repeatedly receives state information from its environment, performs an action, and then receives a reward signal. The agent acts according to some *policy* and attempts to improve its performance over time by modifying the policy to accumulate more reward. To be more precise we will utilize standard notation for Markov decision processes (MDPs) (Puterman 1994). The agent's knowledge of the current state of its environment, $s \in S$ is a vector of k *state variables*, so that $s = x_1, x_2, \ldots, x_k$. The agent has a set of actions, A, from which to choose. A reward function, $R : s \mapsto \mathbb{R}$, defines the instantaneous environmental reward of a state. A policy, $\pi : S \mapsto A$, fully defines how an agent interacts with its environment. The performance of an agent's policy is defined by how well it maximizes the received reward while following that policy.

RL agents often use a parameterized function to represent the policy; representing a policy in table is either difficult or impossible if the state space is either large or continuous. *Policy search* RL methods, a class of global optimization techniques, directly search the space of possible policies. These methods learn by tuning the parameters of a function representing the policy. NEAT is one such learning method.

NeuroEvolution of Augmenting Topologies (NEAT)[1]

This paper utilizes *NeuroEvolution of Augmenting Topologies* (NEAT) (Stanley & Miikkulainen 2002), a popular, freely available, method to evolve neural networks via a *genetic algorithm*. NEAT has had substantial success in RL domains like pole balancing (Stanley & Miikkulainen 2002) and virtual robot control (Taylor, Whiteson, & Stone 2006).

In many neuroevolutionary systems, weights of a neural network form an individual genome. A population of genomes is then evolved by evaluating each and selectively reproducing the fittest individuals through crossover and mutation. Most neuroevolutionary systems require the designer to manually determine the network's topology (i.e. how many hidden nodes there are and how they are connected). By contrast, NEAT automatically evolves the topology by learning both network weights and the network structure.

NEAT begins with a population of simple networks: inputs are connected directly to outputs without any hidden nodes. Two special mutation operators introduce new structure incrementally by adding hidden nodes and links to a network. Only structural mutations that improve performance tend to survive evolution. Thus NEAT tends to search through a minimal number of weight dimensions and find the appropriate level of complexity for the problem.

Since NEAT is a general purpose optimization technique, it can be applied to a wide variety of problems. When applied to reinforcement learning problems, NEAT typically evolves *action selectors*, in which the inputs to the network describe the agent's current state. There is one output for each available action and the agent chooses whichever action has the highest activation, breaking ties randomly.

RL Tactician

To learn a policy to control the RL Tactician, we utilize NEAT with a population of 100 policies and standard parameter settings (Stanley & Miikkulainen 2002). Initially, every neural network in the population has the same topology but a different set of random weights. The RL Tactician must handle five distinct types of tactics and thus every policy is composed of five networks (the 5 actions considered by the RL agent). Each network has five inputs (the 5 state variables considered by the RL agent as the current world state) and one output.

[1]This section is adapted from the original NEAT paper (Stanley & Miikkulainen 2002).

- 1 bias input is set to always output 0.03.
- 1 input node for the productivity, a real-valued number, scaled[2] in the range of 0 to 1.
- 3 input nodes describe either the completeness or preference level, depending on the tactic type. Each of these features has three discreet preference levels, which correspond to one input with a value of 0.97 and the other two inputs with a value of 0.03.
- 1 output holds the network's evaluation of the input tactic.
- Randomly weighted links fully connect the 5 inputs to the output.

The output of the neural network is the evaluation of the input tactic. Every tactic is independently evaluated to a real number in the range of [0, 1] and all tactics are returned by ResearchCyc in sorted order.

An appropriate fitness function must be chosen carefully when using policy search RL as subtle changes in the problem definition may lead to large changes in resultant behavior. To reduce the time to first answer, a fitness based on the number of tactics used to find the first answer is most appropriate. Although not all tactics types in Cyc take precisely the same amount of time, they are approximately equivalent. An alternative approach would be to measure the CPU time needed to find an answer, but this is a machine-dependant metric and is significantly more complex to implement. Results suggest that fitness based on tactics is appropriate by demonstrating that the number of tactics used is closely correlated with the total inference time.

Given a set of queries, our goal is to minimize the number of tactics required to find an answer for each query (or decide that the query is unanswerable). However, the RL Tactician should be penalized if it fails to answer a query that the Balanced Tactician was able to answer; otherwise, the RL Tactician may maximize fitness by quickly deciding that many queries are unanswerable. Thus we have a tunable parameter, γ, which specifies the penalty for not answering an answerable query. When answering a set of queries, the fitness of a RL Tactician is defined as

$$\sum_{queries} \text{-}\,(\textit{\#Tactics}) - (\gamma \times \textit{\#QueriesMissed}). \tag{1}$$

In our experiments, we set γ to 10,000; this is a significant penalty, as most queries are answered by the Balanced Tactician with fewer than 10,000 tactics.

Experimental Methodology

This section details how the RL Tactician interfaces with ResearchCyc as well as how experiments were conducted. A number of assumptions are made for either learning feasibility or ease of implementation:

- The RL Tactician provides the inference Engine with an ordered list of tactics. However, the inference Engine may use meta-reasoning to decide to execute only a subset of these tactics. Ideally, the RL Tactician should be able to decide directly which tactics to execute in reducing inference time. However, this has been left to future work as we anticipate that the improvement would be minor.
- The Inference Engine does a substantial amount of meta-reasoning about what tactics should be discarded and not given to the RL Tactician as a possible choice. Removing this meta-reasoning has the potential to make the problem more difficult, but may also increase the speed of inference as the amount of higher-level reasoning required is reduced.
- The Cyc Inference Engine provides a large number of user-settable parameters, which control a variety of factors (such as defining what resource cutoffs to use or how to focus search). This experiment used a fixed set of parameters rather than those saved as part of the queries being used. This simplifies experimental setup and has the advantage that the longer-term goal is to have a single inference policy that works optimally in all cases, rather than requiring users to hand-tune engine behavior.
- The RL Tactician does not handle conditional queries, which are used to query for information about implications (e.g. does being mortal imply being a man), nor is it used when performing forward inferences (in which the consequences of a rule are asserted explicitly into the knowledge base). The RL Tactician should be able to learn appropriate control policies for both types of queries utilizing the same experimental setup, but the behavior is likely to be qualitatively different from standard backward inference.
- Of the eight tactic types available in ResearchCyc, the RL Tactician currently only handles five. Allowing one of the three remaining types will make the learning task only slightly harder but will require a substantial code change to the Cyc Inference Engine; the last two tactic types are transformations, and are discussed in the final point.
- No transformations are allowed during inference. Typically proofs which involve transformations are substantially more difficult as many more steps are required and is enabled by a user setting. We leave removing this restriction to future work as proofs with transformations are qualitatively different from removal-only proofs.

When transformations are enabled, the Tactician is allowed to apply a rule in the KB to a single-literal problem to produce a different problem. This increases the complexity of queries that can be answered at the expense of significantly increasing the inference space, and therefore the amount of time needed to answer a query. This more difficult class of inferences will likely benefit from similar learning techniques and will be addressed in future work.

When training, each policy must be assigned a fitness so that NEAT can correctly evolve better policies. When training over n queries, each policy in a population is used to control the RL Tactician while it is directing inference for n sequential queries. Algorithm 1 details how the RL Tactician interacts both with NEAT and ResearchCyc while learning.

[2] Valid productivity numbers range from 0 to 10^8 but low values typically indicate superior tactics. Thus precision is much more important at the low end of the range and we use a double log function: $in_{productivity} = 0.01 + log(1 + log(1 + productivity))$. This formula was chosen after initial experiments showed that it was difficult to learn with a linearly scaled productivity.

Algorithm 1 EVALUATING TACTICIAN POLICIES

```
1: while learning do
2:   for each policy to evaluate do
3:     $NumTactics \leftarrow 0$, $NumMissed \leftarrow 0$
4:     for each query $q$ do
5:       Direct the Inference Engine to find an answer to $q$
6:       while $q$ does not have an answer and the Inference Engine has not determined that $q$ is unanswerable do
7:         if only 1 tactic is available then
8:           Place the tactic in list, $l$
9:         else
10:          Pass features to the RL Tactician for each tactic
11:          RL Tactician evaluates each of the tactics with the current policy by calculating the output for every neural network, given the current state of the world
12:          The RL Tactician returns ordered list of tactics, $l$, to the Inference Engine
13:        end if
14:        Inference Engine executes tactics from $l$
15:      end while
16:      Increment $NumTactics$ by the number of tactic(s) used to answer query $q$
17:      if Balanced Tactician answered $q$ but RL Tactician did not then
18:        $NumMissed \leftarrow NumMissed + 1$
19:      end if
20:    end for
21:    $PolicyFitness \leftarrow NumTactics + NumMissed * \gamma$
22:  end for
23:  Evolve the population of policies with NEAT based on each policy's fitness
24: end while
```

When performing inference in ResearchCyc we use standard settings and a time cutoff of 150 seconds per query. The queries used are contained within ResearchCyc; queries that the RL Tactician cannot handle are filtered out (such as queries that require transformation).

The NEAT algorithm requires every policy to be evaluated to determine a fitness. However, this evaluation can be done in parallel, and thus with a population of 100 policies, up to 100 machines can be used concurrently. Our setup utilized 15-25 machines with identical hardware in a cluster.

NEAT is run as an external C process and a learned RL Tactician is fully described by the five neural networks that represent a policy. Thus it would be relatively easy to incorporate a trained policy directly into the ResearchCyc infrastructure, likely leading to additional speedup as communication overhead is removed.

Results and Discussion

Following the methodology discussed in the previous section, this section shows how the performance of a learning RL Tactician improves over time. When evaluating the results in this section, two points are important to keep in mind. The first is that the Balanced Tactician is a well-tuned module and we expect it to perform well, in general. The second is that RL Tactician has a smaller set of features available than the Balanced Tactician when making decisions. If the RL Tactician's features are augmented to be more similar to that of the Balanced Tactician, it is likely that the performance of the RL Tactician will increase.

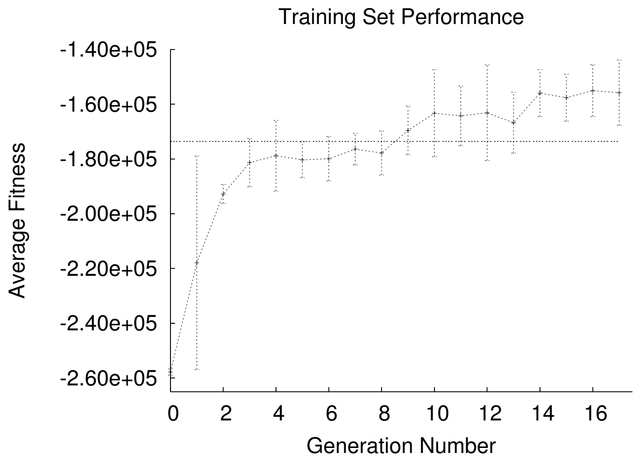

Figure 1: For each generation, the fitness of the best policy from three independent learning trials are averaged. This graph demonstrates that learned policies can outperform the Balanced Tactician on the training set of queries.

Figure 1 shows the best policy's performance of each generation on the training set of 382 queries. Three independent trials are averaged together to form the learning curve and error bars show a standard deviation. The line in the graph shows the performance of the Balanced Tactician on the same set of queries. Recall that we define the Balanced Tactician's fitness (Equation 1) as the number of tactics used, while the RL Tactician's fitness includes a penalty for missed queries. When using 20 machines each generation roughly takes between twelve and three hours of wall clock time, where initial generations take longest because the less-sophisticated policies take much longer to answer queries. We utilized a cluster of 20 machines to make the experiments feasible.

Figure 2 shows the same two sets of policies on the independent test set, made of 381 queries. After generation 14 all three of the best policies in each of the learning trials were able to outperform the Balanced Tactician. This shows NEAT successfully trained neural networks to control inference such that the fitness score for the RL Tactician outperformed the hand coded tactician.

Although we measured performance based on the number of tactics executed, our motivation for this work was reducing the amount of time needed to perform inference on a set of queries. We took the top performing policies at the beginning and end of the first training curve and ran each four times over the set of test queries. CPU times for these two policies are reported in Table 1. The Balanced Tactician outperforms the RL Tactician because it is implemented within the Cyc Inference Engine. However, as the number of tactics are reduced between the first and twentieth generations the time spent is reduced. A Student's t-test confirms that the difference between the two learned policies is statistically significant ($p < 3.7 \times 10^{-6}$). This demonstrates that execution time is indeed correlated with our fitness function and suggests that once the learned RL Tactician is incorporated into the ResearchCyc inference harness, wall clock time for answering this set of test queries will be reduced.

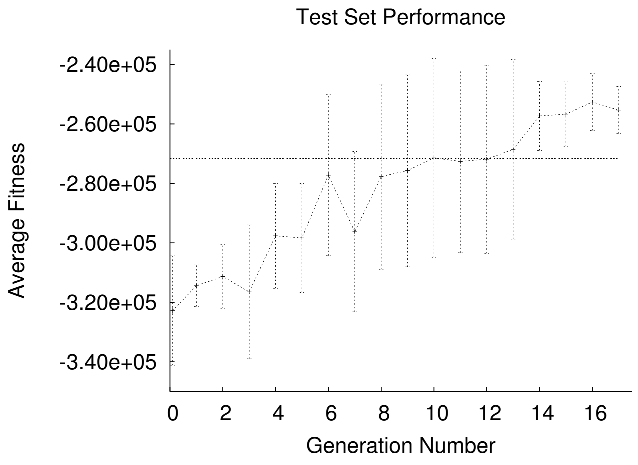

Figure 2: Each of the best policies found per generation, as measured on the training set, are used to perform inference on the test set. The resulting fitnesses are averaged and show that the best learned policies, as selected by training set performance, are also about to outperform the Balanced Tactician on the novel set of test queries.

CPU Time

Policy	Ave. Time (sec.)	Std. Dev. (sec.)
Best of Gen. 1	2012.94	74.28
Best of Gen. 20	1356.01	33.63
Balanced Tactician	1270.76	24.74

Table 1: This table reports the average CPU time answering the test query set. Polices used are: the highest fitness policy from the first generation of the first trial, the highest fitness policy from the 20^{th} generation of the first trial, and the hand-coded Balanced Tactician.

Although the fitness formula does not explicitly credit answering queries that were unanswerable by the Balanced Tactician, we found that during trials a number of policies learned to answer training queries that the Balanced Tactician did not answer. This suggests that tuning the fitness equation would allow the RL Tactician to maximize the number of queries that it answered in the specified time limit rather than minimizing the number of tactics used.

In a second set of experiments we modified the RL Tactician so that only tactics with a score of at least 0.3 are returned. We hypothesized that this second implementation would allow the tactician to learn to discard poor tactics faster and be able to give up on unanswerable problems by assigning a low evaluation to all tactics. However, three independent trials of this second implement produced results very similar to our first implementation which did not exclude any tactics. These supplemental results are omitted for space reasons.

Implementation in Cyc

When the RL Tactician was implemented within the Cyc inference engine, the previously unseen test corpus was answered faster, as expected. The RL Tactician was approximately twice as fast as the Balanced Tactician in time to first answer by median averaging, but approximately 30% *slower* when the mean was taken (see Figure 3). The RL Tactician successfully improved time to first answer for the majority of the test cases, although the RL Tactician actually performed less well on certain very long-running outliers which

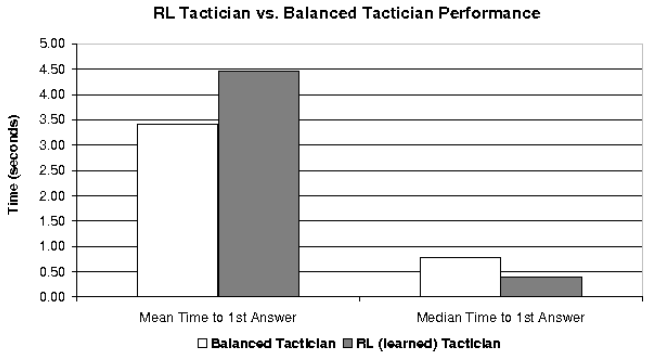

Figure 3: The relative performance of time to first answer for the learned RL Tactician and the Balanced Tactician. The learned tactician improves on the performance of the Balanced Tactician in the most common (median) case, although the Balanced Tactician does slightly better on average (mean).

accounted for a substantial fraction of the total time spent. Completeness (total answers produced) was comparable—of 1648 queries, the Balanced Tactician answered 13 that the RL Tactician failed to answer, and the RL Tactician answered 9 that the Balanced Tactician failed to answer.

Further analysis showed that a majority of the cases in which the RL Tactician showed improvement were queries in which a single high-level tactic was being selected differently by the two tacticians. The top-level tactic being selected by the RL Tactician is an iterative problem-solver that produces initial results quickly, while the Balanced Tactician was selecting a non-iterative version which produces *all* results slightly more quickly, but the first few results more slowly.

These results represent successful learning, as the evaluation function used to train the RL Tactician measured time to first answer and total answerable queries. Since the first answer to a query is frequently sufficient for the user's purposes, this also represents an improvement in the usability of the Cyc inference engine for applications in which responsiveness in user interactions is desirable.

Future Work

One future direction for this work is to further improve the RL Tactician speed. The first step will be to enhance the feature representation so that the learner may learn better policies. One such enhancement would be to allow the RL Tactician to directly compare tactics simultaneously, as the Balanced Tactician does, rather than independently evaluating each tactic. Another future direction is to adjust the evaluation function being used by the learner to take total time, as well as time to first answer, into account. Although requiring the learner to speed up both measurements is a potentially complex problem, the success achieved in improving time to first answer with relatively little training time is promising.

An additional goal is to handle more types of queries, particularly those that involve transformations and forward inference, as these types of queries are likely to benefit substantially. These different kinds of inference may require different behavior from the RL Tactician. In this case, different types of tacticians could be trained independently and then one would be utilized, depending on the type of query the Inference Engine was currently processing.

As well as extending the evaluation function and training on larger sets of queries, it would be worth attempting to optimize over a more representative training corpus. This work has utilized queries saved as part of the ResearchCyc infrastructure, but they are not necessarily representative of the types of queries that Cyc is typically required to answer. A corpus of queries may be constructed by recording the queries actually run by Cyc users and then used to train an RL Tactician. Such a method is more likely to yield a tactician that is able to decrease the running time of Cyc when interacting with users.

Related Work

Humans may inject meta-reasoning into the inference process, such as with *Logic Programming* (Genesereth & Ginsberg 1985), in order to speed up inference. Pattern matching, a primary example being *Rete* (Forgy 1982), operates by constructing a network to determine which rules should be triggered. Memory efficiency is sacrificed for lower computation complexity. There are other algorithms which are able to perform faster or with less memory (e.g. *Treat* (Miranker 1987)) which rely on similar approaches. Our work differers from that of logic programming and pattern matching as we rely on learning techniques rather than static solutions.

More interesting are approaches that learn control rules via learning, such as by using explanation-based learning (Minton 1990; Zelle & Mooney 1993). Another popular approach for speeding up inference is that of *chuncking*, which is based on a psychological theory about how humans make efficient use of short term memory. Soar (Laird, Newell, & Rosenbloom 1987) is one system that makes use of chuncking by deductively learning chunks. That is, it learns meta-rules in the form of: $if\ \langle situation\rangle\ then\ use\ \langle rule\rangle$, enabling the inference module to bias its search through inference space with rule preferences. This work differs because it utilizes an "off the shelf" data-driven statistical learning method. Additionally, this work produces speedups over a large number of complex queries in a large knowledge base, outperforming a hand-tuned module designed to optimize inference time.

Most similar to our research is work (Asgharbeygi *et al.* 2005) that uses *relational reinforcement learning* (RRL) to guide forward inference in an agent interacting with the world. In this task, the agent greedily proves as many things as possible before time expires. Different proofs are given different utilities by the task designers and the agent's goal is to maximize utility. Our work differs for three primary reasons. Firstly, the RL Tactician learns to reduce the amount of time needed to find a proof (if one exists), rather than trying to determine which proofs are most useful out of a set of possible proofs. Secondly, by utilizing the existing Cyc knowledge base rather than sensations from a simulated agent, many of our proofs take thousand of inference tactics, as opposed to learning only one- or two-step proofs. Lastly, RL is more general than RRL, which relies on human-defined relationships between different objects in the environment and thus may be more broadly applicable.

Conclusion

This work demonstrates that an existing reinforcement learning technique can be successfully used to guide inference to find answers to queries. Not only was the performance on training and test sets of queries increased over time, but the learned inference module was able to outperform the hand-coded inference module within ResearchCyc. This work shows that reinforcement learning may be a practical method to increase inference performance in large knowledge base systems for multi-step queries.

Acknowledgments

We would like to thank Robert Kahlert, Kevin Knight, and the anonymous reviewers for helpful comments and suggestions. This research was supported in part by NSF award EIA-0303609 and Cycorp, Inc.

References

Asgharbeygi, N.; Nejati, N.; Langley, P.; and Arai, S. 2005. Guiding inference through relational reinforcement learning. In *Inductive Logic Programming: 15th International Conference*.

Etzioni, O.; Cafarella, M.; Downey, D.; Kok, S.; Popescu, A.-M.; Shaked, T.; Soderland, S.; Weld, D. S.; and Yates, A. 2004. Web-scale information extraction in KnowItAll (preliminary results). In *WWW*, 100–110.

Forgy, C. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19:17–37.

Genesereth, M. R., and Ginsberg, M. L. 1985. Logic programming. *Communications of the ACM* 28(9):933–941.

Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. Soar: an architecture for general intelligence. *Artif. Intell.* 33(1):1–64.

Lenat, D. B. 1995. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM* 38(11):33–38.

Matuszek, C.; Witbrock, M.; Kahlert, R. C.; Jabral, J.; Schneider, D.; Shah, C.; and Lenat, D. 2005. Searching for common sense: Populating cyc from the web. In *Proceedings of the 20th National Conference on Artificial Intelligence*.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42(2-3):363–391.

Miranker, D. P. 1987. Treat: A better match algorithm for ai production system matching. In *AAAI*, 42–47.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc.

Ramachandran, D., Reagan, P., and Goolsby, K. 2005. First-orderized researchcyc: Expressivity and efficiency in a common-sense ontology. In *Papers from the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications*.

Stanley, K. O., and Miikkulainen, R. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation* 10(2):99–127.

Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. MIT Press.

Taylor, M.; Whiteson, S.; and Stone, P. 2006. Comparing evolutionary and temporal difference methods for reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1321–28.

Zelle, J., and Mooney, R. 1993. Combining FOIL and EBG to speed-up logic programming. In Bajcsy, R., ed., *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1106–1113. Morgan Kaufmann.