Option Discovery in Hierarchical Reinforcement Learning for Training Large Factor Graphs for Information Extraction

April 13, 2009

Sameer Singh

SYNTHESIS PROJECT REPORT
Graduate Program Portfolio

Abstract

Since exact training and inference is not possible for most factor graphs, a number of techniques have been proposed to train models approximately, but they do not scale to large factor graphs used in recent work on joint inference on multiple information extraction tasks. SampleRank is an MCMC based training algorithm that performs competitively in accuracy and speed on these problems, but is sensitive to local minima in the scoring function. This problem has been averted by framing the problem in the reinforcement learning framework.

Both Reinforcement Learning and SampleRank are restricted in speed (and therefore accuracy) due to the proposal function. Naive proposal functions propose small changes to the configuration that do not contain information to accurately represent the evaluation of the move. If the proposal function is designed to propose larger moves in the configuration space, the number of possible moves increases dramatically, and domain knowledge is required to select useful moves, which may not be possible for most tasks. These problems lead to requiring a very large number of samples before good moves are discovered and high accuracy is obtained.

To avert these problems, MAP inference in factor graphs is reframed as hierarchical reinforcement learning, and a novel method for discovering options fast is introduced. Sample trajectories are analyzed to detect dependencies between primitive actions. These dependencies are exploited to extract the commonly occurring sequences efficiently, and are then abstracted to form closed loop stochastic policies. The method is efficient at discovering useful options, and can handle a large number of long trajectories. These options require fewer samples to reach regions of state space that are close to the goal state. Options discovered are shown to reduce the number of samples required for high accuracy on a real world information extraction dataset.

Andrew McCallum
Synthesis Reader

Andrew Barto
Synthesis Reader

1 Introduction

Factor graphs have been widely used for information extraction tasks. Many of the best results obtained on information extraction tasks like coreference, named entity recognition and segmentation have been obtained by modeling the data as factor graphs. Exact training and inference on factor graphs can be performed tractably only for graphs that are small or limited in structure (like linear chains, trees). For most applications, these restrictions that make the models tractable are also impractical. Instead, a number of techniques have been proposed to train models approximately. Even though these techniques have been used successfully for a number of years, they do not scale to large factor graphs.

The need for scaling to large factor graphs is further motivated with the recent drive to model several tasks "jointly". Tasks involving multiple information extraction steps have traditionally been solved using a pipeline architecture, where the output predictions of one stage are used as the input to the next. This sort of architecture is susceptible to cascading of errors from one stage to the next. To minimize this error, there has been recent interest in *joint inference* of multiple tasks (Poon and Domingos, 2007; Punyakanok et al., 2005; Wellner et al., 2004). Full joint inference usually results in exponentially large models for which even most approximate learning and inference methods become intractable (Poon et al., 2008).

SampleRank(Culotta, 2008; Wick and McCallum, 2008) is a Metropolis-Hastings based MCMC training algorithm that performs competitively in accuracy but is also typically faster than other current methods on these large factor graphs. Examination of SampleRank shows evidence of slowdown because it gets stuck in local minima of the scoring function, relying completely on random sampling to jump out of these cases. These local minima are plentiful for most scoring functions; escaping one local minimum is usually followed by getting stuck in another. This behavior leads to slow convergence since a lot of time is spent learning to escape the local minima, a problem that worsens as the size of the factor graph increases.

It was hypothesized that this behavior is caused by a single step lookahead of the scoring function while performing MCMC(Rohanimanesh et al., 2008). A simple improvement would involve a finite horizon local search to evaluate the next states. This only pushes the local minima to outside the finite horizon. Instead, evaluation of the state should correspond to the evaluation of the state space reachable from that state after infinite steps. This notion of the *value* of the state corresponds directly with the concept of "delayed reward" existing in the reinforcement learning literature.

Both Reinforcement Learning and SampleRank were found to be further restricted in both accuracy and speed due to definition of the proposal function itself. Next action is chosen in both methods by sampling the next states from the proposal function. If the changes that are proposed are small, they can only evaluate the local value function which may not correspond to the global value function. Thus these changes do not have enough information to represent the evaluation of the move, whether it is the ranking function as in SampleRank, or the long term expected reward, as in reinforcement learning. When the proposal function is designed to propose larger moves in the configuration space, the number of possible moves increases dramatically, which leads to requiring a large amount of sampling before good moves are found.

This work reframes the problem as Hierarchical Reinforcement Learning, by using semi-markov decision process (Semi-MDP). Semi-MDPs allows usage of *options*(Sutton et al., 1999) that can represent for more complicated proposal functions. Options are extended actions that are used in reinforcement learning for temporal abstraction, performing "lower-level" actions and leaving the higher-level decisions to the value function. By using this hierarchy of actions, the value function at the higher level is usually much smoother than when only the primitive actions are used.

Options can be designed so that they exploit domain knowledge to reduce the number of samples required. However this task is not trivial, as the options have to be created such that they do not create unnecessary redundancy with the primitive action, while at the same time span the space not covered by the primitive actions. The design of options is especially difficult for joint inference tasks, where options have to be designed that span multiple domains. There has been extensive work in option discovery (Mannor et al., 2004; Hengst, 2002; Pickett and Barto, 2002; Randlov, 1999), but they cannot be applied directly due to the size of our problems.

Extending on previous work, we introduce a novel method for option discovery that can be used to discover useful options quickly, and scales to the sizes of the factor graphs used for information extraction. Instead of incorporating option-discovery during learning, sample trajectories resulting from inference are analyzed to detect commonly occurring sequences of actions that are abstracted to form closed loop policies. Dependencies between actions are exploited to only explore sequences in which the actions are related. Sequences are restricted to linear chains in which the actions directly cause the next action.

Most of the successful option discovery algorithm work in a closed loop with the learning system, i.e. there is constant evaluation of the proposed options to decide it's inclusion in the final set of actions. Since the options have been defined such that they are closed under combination, they can be built upon to discover higher-level options than span larger regions of the configuration space. Even though these approaches are not explored in this work, in order to use our method for these purposes, it has been designed to be very efficient and fast. By restricting most of the analysis to single trajectory at a time, the method scales linearly with the number of trajectories.

A widely studied task for information extraction is that of citation matching (Poon and Domingos, 2007; Wellner et al., 2004). Citation matching involves segmenting citation strings into author, title and venue fields (*segmentation*), and clustering the citations that refer to the same paper into a single entity (*coreference*). Previous results have been promising (Poon and Domingos, 2007), and this task provides a good testbed for the option-discovery method. Options discovered automatically by our method were shown to improve accuracy and speed of SampleRank on a real world citation matching dataset.

Some background material on approximate training of factor graphs using SampleRank and introduction to reinforcement learning and options is given in Section 2. We describe our method in Section 3, with details on accumulation of trajectories, and identification of recurring sequences, and creation of options from them. We demonstrate the benefits of using our method on real world data on a two different domains in Section 4. Our method is compared to related literature in Section 5. We end by mentioning limitations of the method, and future plans to address them, in Section 6.

2 Background

In this section, we provide introductory background material to some of the frameworks and methods employed in this work.

2.1 CRF

A conditional random field (CRF)(Lafferty et al., 2001) is a discriminative factor graph that gives a conditional probability distribution over an assignment to a set of hidden variables **Y**=**y** given a set of observed variables **X**=**x**. That is, a CRF can be written in the form:

$$P(\mathbf{Y}=\mathbf{y}|\mathbf{X}=\mathbf{x}) = \frac{1}{Z_X}\prod_{\mathbf{Y_i}\in\mathcal{M}}\psi(\mathbf{X},\mathbf{y_i})$$

where Z_X is an input-dependent normalizing constant ensuring that the distribution sums to one. The structure of the CRF is determined by the factors $\psi(\mathbf{X}, \mathbf{y_i})$, that decompose the model into sets of variables $\mathbf{Y_i} \in \mathcal{M}$ where $\mathcal{M} \subseteq \mathcal{P}(\mathbf{Y})$[1]. A factor is a function that maps an arbitrary number of hidden ($\mathbf{y_i}$) and observed variables ($\mathbf{X}$) to a real-value. Typically CRF factors are log-linear combinations of a feature vector $\phi(\mathbf{X}, \mathbf{y_i}) = \{\phi_j(\mathbf{X}, \mathbf{y_i})\}$ and parameters $\theta = \{\theta_j\}$. In this case the factors are calculated by $\psi(\mathbf{X}, \mathbf{y_i}) = exp(\phi(\mathbf{X}, \mathbf{y_i}) \cdot \theta)$.

As an example, consider the name disambiguation problem, where a set of names have to be clustered such that each cluster contains names that refer to a single name. As the input observed variables, we are given the set of names as strings ($\mathbf{X} = \{x_i\}$, where each x_i is a string). There is one hidden variable y_i for every name x_i that represents the cluster number of the name. An example of the factors that may be created for a configuration of $\mathbf{Y}$ is the pairwise Affinity factors that are created for every pair of coreferent entities. In this case there exists a factor $\psi(\mathbf{X}, y^i, y^j)$ for every i, j s.t. $y_i = y_j$. Hence, $\mathcal{M}$ is simply the set of all pairs (y^i, y^j) s.t. $y_i = y_j$. A simple feature for this case could simply be the string edit distance between the two strings, $\text{StringDist}(x_i, x_j)$. Hence, $\psi(\mathbf{X}, y_i, y_j) = exp(\text{StringDist}(x_i, x_j) \times \theta)$. It should be mentioned here that this is merely one simple way of representing the problem as a CRF.

Learning of the parameters θ is typically performed with some form of gradient descent, which requires computing the derivative of the log-likelihood with respect to each parameter. However, the above gradient involves model expectation over features, requiring a summation over all configurations $\mathbf{Y} = \mathbf{y}$.

We shall describe an approximate learning method in Section 2.3 that avoids this problem. However, we first introduce Metropolis Hastings inference in CRFs.

2.2 MCMC Inference in CRFs

Leaving the problem of learning the parameters aside, even when the parameters θ are given, it is not trivial to calculate the probability of a given configuration due to the normalizing factor Z_X. Instead of finding the marginal probability of any configuration, we are usually interested in the MAP configuration, i.e. the configuration of the unobserved variables $\mathbf{Y}$ that has the highest probability as given by the parameters. To find this MAP configuration, we usually employ the MCMC based Metropolis Hastings algorithm.

Each unique setting $\mathbf{y}$ of the values of the hidden variables $\mathbf{Y}$ of the factor graph constitutes a single state for MCMC. Even though we cannot calculated $P(\mathbf{y})$ directly due to the normalizing factor, we can calculate $\frac{P(\mathbf{y})}{P(\mathbf{y'})}$ for any two settings $\mathbf{y}$ and $\mathbf{y'}$ of $\mathbf{Y}$ since the normalizing factor Z_X is independent of the values of the hidden variables. Specifically,

$$\frac{P(\mathbf{y})}{P(\mathbf{y'})} = \frac{\prod_{\mathbf{Y_i} \in \mathcal{M}} \psi(\mathbf{X}, \mathbf{y_i})}{\prod_{\mathbf{Y_i} \in \mathcal{M}} \psi(\mathbf{X}, \mathbf{y'_i})} = \prod_{\mathbf{Y_i} \in \mathcal{M}} \frac{\psi(\mathbf{X}, \mathbf{y_i})}{\psi(\mathbf{X}, \mathbf{y'_i})}$$

Hence we see that the ratios of individual factors can be taken to calculate this ratio of probabilities. This is further simplified by noticing that since the factors are log-linear combinations of feature vectors, the ratio between each factors reduces to the dot product of the parameters θ and the difference of the feature vectors between $\mathbf{y}$ and $\mathbf{y'}$.

The MCMC approach relies on a proposal distribution Q that, given the current configuration $\mathbf{y}$, stochastically proposes a neighbor configuration $\mathbf{y'}$. Once the proposal distribution Q is specified, and we can calculated $\frac{P(\mathbf{Y}=\mathbf{y})}{P(\mathbf{Y}=\mathbf{y'})}$ and $\frac{Q(\mathbf{y},\mathbf{y'})}{Q(\mathbf{y'},\mathbf{y})}$, we can employ Metropolis Hastings to reach the MAP configuration. It should be noted when using Metropolis Hastings to perform MAP inference,

[1] $\mathbf{y_i}$ represents the value for variables $\mathbf{Y_i}$

the restrictions to the proposal distribution are relaxed, not requiring strict ergodicity or detailed balance (Culotta, 2008; Wick and McCallum, 2008).

Continuing with the example given in the last section, we can define a proposal function as one that, given the current clustering[2], randomly chooses a name x_i, and randomly sets y_i to any number between $1 \dots n$, where n is the number of names. Since the name and cluster are picked uniformly, $\frac{Q(\mathbf{y},\mathbf{y}')}{Q(\mathbf{y}',\mathbf{y})} = 1$.

2.3 SampleRank

This section briefly introduces an approximate learning method for the parameters called SampleRank (Culotta, 2008; Wick and McCallum, 2008). SampleRank updates the parameters of the factor graph while performing inference on training data with Markov Chain Monte-Carlo through the space of possible configurations of the factor graph, as described in the last section. The updates are made with approximate gradients at each MCMC step. Each step of MCMC creates a neighbor configuration pair by modifying a configuration $\mathbf{y}$ to produce $\mathbf{y}'$ using the proposal distribution. Usually, the difference between configurations y' and y is small and computing their gradient is efficient (details in Wick and McCallum (2008)).

The update rule can be applied as follows:

$$\theta = \theta + \begin{cases} -\alpha\ \phi_{y,y'} & \text{if } \mathcal{F}(\mathbf{y}) > \mathcal{F}(\mathbf{y}') \wedge \theta \cdot \phi_{y,y'} > 0 \\ \alpha\ \phi_{y,y'} & \text{if } \mathcal{F}(\mathbf{y}) < \mathcal{F}(\mathbf{y}') \wedge \theta \cdot \phi_{y,y'} \leq 0 \end{cases}$$

where α is the learning rate, θ are the current parameters, $\phi_{y,y'}$ is the difference of the feature vectors between the two state representations, $\mathcal{F}(y)$ is a ground truth metric. $\mathcal{F}(y)$ measures the quality of the configuration $\mathbf{y}$, as compared to the ground truth configuration. As an example of this metric, we return to the coreference example mentioned in the earlier sections. A possible metric is the accuracy of the predicted clustering, i.e. the fraction of all pairs, the coreference of which is same in the predicted clustering and the true clustering.

2.4 Reinforcement Learning

In this section we briefly overview MDPs, and introduce basic reinforcement learning concepts and algorithms. Most of the discussion here is based on Sutton and Barto (1998). Reinforcement Learning (RL) refers to a class of problems in which an agent interacts with the environment and the objective is to learn a course of actions that optimizes a long term measure of a delayed reward signal. The most popular realization of RL has been in the context of Markov Decision Processes (MDPs).

A finite Markov Decision Process (MDP) is defined by a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbf{R}$ is the reward function, i.e. $\mathcal{R}(s, a, s')$ is the expected reward when action a is taken in state s and transitioning into state s', and $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition probability function, i.e. $\mathcal{P}^a(s, s')$ is the probability of reaching state s' if action a is taken in state s.

A stochastic policy π is defined as $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ where $\pi(s, a)$ is the probability of choosing action a as the next action when in state s. Following a policy on an MDP results in an expected discounted reward R_t^π accumulated over the run, where $R_t^\pi = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$. An optimal policy π^* is defined as the policy which maximizes the expected discounted reward accumulated on the MDP.

[2]As a reminder, a clustering is a complete set of clusters of the mentions, i.e. an assignment to $\mathbf{Y}$.

The optimal policy can be found using Q(λ) learning that combines Temporal Difference (TD) learning (Sutton, 1988) and Q-Learning (Watkins and Dayan, 1992). It maximizes a value function that is associated with the policy π, defined as $Q^{\pi} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbf{R}$ where $Q^{\pi}(s, a)$ represents the expected long term discounted reward starting at the state s, taking action a, and following the policy π. This algorithm provides a simple yet efficient method to learn the Q-values. It has strong theoretical convergence guarantees, and in practice finds the optimal policy fairly quickly.

Q(λ) is limited to very small domains since it involves storing a value for each state-action pair. This problem is averted by employing function approximation techniques in reinforcement learning (Sutton and Barto, 1998). When linear functional aproximator is used, the state-action pair $\langle s, a \rangle$ is now represented by a feature vector $\phi(s, a)$. The Q value is represented using a vector of parameters θ, i.e.

$$Q(s, a) = \sum_{\phi_k \in \phi(s,a)} \theta_k \phi_k$$

Instead of updating the Q values directly, the updates are made to the parameters θ using a perceptron style update.

$$\overrightarrow{\theta} \quad \leftarrow \quad \overrightarrow{\theta} + \alpha \left(r_{t+1} - Q(s_t, a_t) + \gamma \max_{a} Q(s_{t+1}, a) \right)$$

Q(λ) also utilizes eligibility traces, which represent the recency and the frequency of visits to that state and action, and are used for credit assignment (Sutton, 1988). Eligibility traces need to be handled differently with functional approximation, as described in Section 8.2,8.3 in Sutton and Barto (1998).

2.5 Options

A number methods have been suggested to introduce hierarchy into reinforcement learning that use *semi-Markov Decision Processes*(Sutton et al., 1999; Parr and Russell, 1998; Dietterich, 2000). For this work, we shall focus on the options framework. The options framework was introduced in Sutton et al. (1999) to include *temporal abstraction* of the actions. Options are specified by a tuple $\langle \mathcal{I}, \pi, \beta \rangle$, where $\mathcal{I}$ is the input set of states, π is a *partial* policy over the states, and $\beta : \mathcal{S} \rightarrow [0, 1]$ is the termination probability. The option is available for use in state s if $s \in \mathcal{I}$. Once the option is selected for use, the policy π is followed and is terminated at each state s' with probability $\beta(s')$. This formulation of options allows us to specify primitive actions as described in Section 2.4 as *one-step* options. For many tasks, the options are terminated after a fixed number of steps. This introduces dependence of β on the history, making the action non-Markov. Thus the term *semi-markov* was introduced which refers options that use policies that are over histories of states and actions, instead of being only over states.

Once a set of options selected for use, a policy $\mu(o, s)$ is created that, analogous to the policy used for MDPs, gives the probability of selecting the option o in state s. The option is then followed till termination, after which the next option is selected from the terminal state using the policy μ. An reinforcement learning technique can be applied to learn the policy μ, e.g. Q-Learning. If useful options have been selected, they *skip* over regions of space where the decision making is trivial, allowing the policy to focus on hard to learn areas.

2.6 Factorie

To implement our method, we used a state of the art probabilistic programming toolkit called Factorie (McCallum et al., 2008), which is short for "Factor graphs, Imperative, Extensible". Factorie provides a complete package for designing factor graphs in a way that allows imperative hookups for injection of domain knowledge. It provides a common framework for evaluating a number of learning and inference algorithms on various domains, and includes efficient implementations of SampleRank and Reinforcement Learning. It has been shown to improve upon earlier state of the art results in information extraction on both accuracy and speed.

Factors in Factorie are divided into neighbors and sufficient statistics. Neighbors define the structural aspect of the factors, while the sufficient statistics define the feature vector given the neighbors of a factor. MCMC techniques are carried out in Factorie using the proposal function, which can be specified by the designer. Given a proposed change, Factorie scores it by identifying the neighbors and scoring it using the parameters and the sufficient statistics. This is done effectively by calculating only the difference of the configurations, as shown in Section 2.2.

3 Method

In this section we shall describe our approach to option-discovery in context of the factor graphs domain. We shall begin with our definition of a proposer function and an action. The factor graph domain is then framed as a reinforcement learning problem. This is followed by description of the first part of the option discovery method, the trajectory accumulation. Finally, we give the details of the option-discovery method, and describe some of its assumptions.

3.1 Proposal Functions

The proposal function for MCMC proposes a change from the current configuration (called *move*), and returns it's forward-backward ratio to be used to calculate the acceptance probability. Our proposal function is constructed such that it consists of a number of Proposers. Each of the proposers can stochastically suggest a change of configuraton, or a *move*. The proposal function randomly chooses a single Proposer to select the next move, and thus in effect sampling from the union of all the moves that can be suggested from all the proposers.

To facilitate generalization while retaining the utility of Proposers, we consider Proposers that take the variables of the factor graphs as arguments. Given a set of argument variables, these proposers stochastically pick a move that takes the arguments into account, presumably making changes to them. If no argmuents are provided, Proposers can randomly pick arguments from the complete set of variables. Proposers then output the change they've made, alongwith the forward-backward ratio, and the set of variables that were *changed*. The tuple containing the Proposer with the input arguments, with the output change to the configuration and the changed variables is called a move. The utility of this formulation shall become evident later in the section.

As an example, consider a ShatterProposer for the coreference domain. This proposer takes as argument a random non-singleton cluster (or, in the absence of an argument, samples from the set of non-singleton clusters) and created singleton clusters out of every mention in that cluster. This is an example of a proposer that is not stochastic given the arguments. A single move suggested by this cluster would be the input cluster argument, and the mentions that were in the cluster, and the singleton clusters that were created for them. Notationally, a move can be written as "$Shatter(c1) \rightarrow \{x_1, x_2, x_3, c_7, c_8\}$". Any other move that changes variables that do not appear in this set of *changed* variables is completely independent of this move.

It should be noted that a single Proposer by itself need not span the complete feasible space, in fact the ShatterProposer cannot propose any moves at all if starting from the all-singleton configuration. However, it is required that the set of Proposers that the Proposal function chooses between should be able to span the feasible region of configuration space with non-zero probability in finite number of steps (*irreducibility*). For example, we can add another Proposer to our coreference example called MergeTwo, that, given two clusters, merges them into a single cluster. This results in our combined Proposal function spanning the space of all clustering.

This uniform representation allows us to specify combinations of proposers as another proposer. For example consider a proposer (MultipleProposer) that consists of a number of proposers that are called in order. The input arguments of the MutlipleProposer correspond to the input arguments of the first proposer. The arguments to the later proposers each depend on the changed arguments of the previous proposer that was called, i.e. we also construct a *Mapping* that stochastically proposes how the changed variables of the last move can be used to construct the input arguments of the next move. We also introduce a termination probability which is sampled before calling each of the proposers in the sequence.

As a simple example of our MultipleProposer for the coreference domain, consider a MergeTwoShatter proposer. This proposer consists of two proposers, MergeTwo, followed by Shatter, and takes as argument two clusters. First, the two argument clusters are merged using MergeTwo proposer. Then the merged cluster is used as an argument to Shatter, resulting in a number of singleton clusters. Our mapping from changed variable of the first proposer to the arguments of the second is hence the Exact mapping. This results in a proposer that, given two cluster, shatters both of them.

Multiple applications of the MultipleProposer with the same arguments can lead to a large number of possible states due to 1) each proposer in the sequence may suggest a random change based on the input argument, 2) each mapping may sample different arguments for the next proposer in the sequence, and 3) due to the termination probability, the MultipleProposer may terminate after calling different number of proposers. In a later section (3.5) we shall described in detail how MultipleProposers can be framed in the *options* framework.

Options, when defined as Multiple Proposers, depend a lot on the representation ability of the mappings. The simplest mappings consist of ones similar to the example given above, that look at variables as simple sets. A stochastic mapping may pick a random subset of the changed variables. More complicated mappings may look at individual variables to study their domain-specific properties, like BiggestCluster, or MostSimilarPairOfMentions. Mappings can be stochastic or deterministic, and also contribute to calculating the forward-backward ratio of the suggested move. For this work, we consider only the basic stochastic mapping functions.

Before the details of how these options can be discovered are described, we shall briefly discuss how the proposal function and the factor graph domain can be framed as a reinforcement learning problem.

3.2 Factor Graphs in Reinforcement Learning

The details of this section are presented in Rohanimanesh et al. (2008). Formally, we can define an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$ with set of states $\mathcal{S}$, set of actions $\mathcal{A}$, reward function $\mathcal{R}$, and transition probability function $\mathcal{P}$ formulating the MAP inference problem as follows:

• **States:** In context of CRF (or more generally, a factor graph) representation of the problem, a state is any assignment $\mathbf{y}$ to the variables $\mathbf{Y}$. For example, in the coreference domain discussed earlier, any arbitrary clustering corresponds to a state in the RL formulation.

• **Actions** Given a state s (e.g., an assignment of $\mathbf{Y}$ variables), an action can be defined as a constrained set of modifications to the variables in state s. This corresponds to the move that is the output of the proposer function as described in Section 2.4. Each possible next state s' is considered the deterministic result of the move, leading to a large number of deterministic actions. Note that we are not concerned with details of the proposal function, it may consist of merely primitive actions, or complicated multiple proposers as discussed in the previous section.

• **Reward Function** The reward function has to be designed in such a way that the policy optimizing the delayed reward reaches the MAP configuration. Since the space is combinatorial, rewards must be shaped in such a way that it facilitates efficient learning at every state. The reward function used is the residual improvement based on the performance metric $\mathcal{F}$(see Section 2.3) when system transitions from some state s to s':

$$R(s, s') = \mathcal{F}(s') - \mathcal{F}(s)$$

• **Transition Probability Function:** Recall that the actions in our system are samples generated from a proposal function $\mathcal{B}$, which may or may not contain options, and that each action (or *move*) uniquely identifies a next state in the system. Let the function that returns this next state deterministically be *simulate*(s,a). Thus, given the state s and the action a, the next state s' has the following probability:

$$\mathcal{P}^a(s, s') = \begin{cases} 0 & s' \neq \text{simulate}(s, a) \\ 1 & s' = \text{simulate}(s, a) \end{cases}$$

To generalize over the state/action space, a similar linear function approximator as used in Section 2.1 is used. The Q value for a state s and an action a is represented by the difference of the features between the current state s and the next state s' that is reached by action a, i.e.

$Q(s, a) = \theta \cdot \phi(s, s')$, where s' =simulate(s, a).

• **Algorithm:** Once the state, action space and the reward function have been defined, we can now employ various RL learning algorithms in order to learn the policy for reaching MAP configuration when the system is initiated to some arbitrary configuration. Our learning algorithm builds on Watkin's Q(λ) algorithm described in Watkins (1989); Watkins and Dayan (1992) and Section 2.4, and shown in detail in Algorithm 1 and Rohanimanesh et al. (2008).

During testing we do not have access to the ground truth labels and thus the reward signal is not available. Since the value function represents the expected discounted reward for each action, we choose the action that maximizes our value function of the next state. Being greedy with respect to the current value function in the absence of rewards is used in policy improvement (Section 5.3 in Sutton and Barto (1998)), and was also used during testing in Zhang and Dietterich (2000).

3.3 Trajectory Accumulation

The first step for option discovery for our approach is Trajectory Accumulation. A trajectory is a list of moves that are chosen in a sample MCMC chain. As described in Section 3.1, a move contains a Proposer, a set of its input arguments, and a set of the output *changed* arguments. A useful trajectory is one that starts from a random, presumably bad, initial configuration, and contains the list of moves that changed the current configuration to a good one. Such a trajectory can be a result of almost any learning method, we use trajectories that result from SampleRank or reinforcement learning in this work. The trajectories that are accumulated in our method are from post-learning inference on the training data. Collecting trajectories for the purpose of option discovery during learning may be error prone since the parameters have not converged, or trajectories may be too long, since learning attempts to explore a large amount of the space.

The choice of accumulating trajectories after learning has some more advantages. Since it is independent of the learning mechanism, our option discovery method can be applied to any learning

Algorithm 1 Modified Watkin's-Q(λ) for Factor Graphs

```
1: Input: Performance metric $\mathcal{F}$, proposal function $\mathcal{B}$
2: Initialize $\overrightarrow{\theta}$ and $\overrightarrow{e} = \overrightarrow{0}$
3: repeat {For every episode}
4:     $s \leftarrow$ random initial clustering
5:     Sample $n$ actions $a \leftarrow \mathcal{B}(s)$; collect action samples in $A_{\mathcal{B}}(s)$
6:     for samples $a \in A_{\mathcal{B}}(s)$ do
7:         $s' \leftarrow Simulate(s, a)$
8:         $\phi(s, s') \leftarrow$ set of features between $s, s'$
9:         $Q(s, a) \leftarrow \sum_{\phi_i \in \phi(s,s')} \theta(i)\phi_i$
10:    end for
11:    repeat {For every step of the episode}
12:        if with Probability $(1 - \epsilon)$ then
13:            $a \leftarrow \arg\max_{a'} Q(s, a')$
14:            $s' \leftarrow Simulate(s, a)$
15:            $\overrightarrow{e} \leftarrow \gamma\lambda\overrightarrow{e}$
16:        else
17:            Sample a random action $a \leftarrow \mathcal{B}(s)$
18:            $s' \leftarrow Simulate(s, a)$
19:            $\overrightarrow{e} \leftarrow \overrightarrow{0}$
20:        end if
21:        $\forall \phi_i \in \phi(s, s') : e(i) \leftarrow e(i) + \phi_i$        {Accumulating eligibility traces}
22:        Observe reward $r = \mathcal{F}(s) - \mathcal{F}(s')$
23:        $\delta \leftarrow r - Q(s, a)$
24:        Sample $n$ actions $a \leftarrow \mathcal{B}(s')$; collect action samples in $A_{\mathcal{B}}(s')$
25:        for samples $a \in A_{\mathcal{B}}(s')$ do
26:            $s" \leftarrow Simulate(s', a)$
27:            $\phi(s', s'') \leftarrow$ set of features between $s', s''$
28:            $Q(s', a) \leftarrow \sum_{\phi_i \in \phi(s',s'')} \theta(i)\phi_i$
29:        end for
30:        $a \leftarrow \arg\max_{a'} Q(s', a')$
31:        $\delta \leftarrow \delta + \gamma Q(s', a)$
32:        $\overrightarrow{\theta} \leftarrow \overrightarrow{\theta} + \alpha\delta\overrightarrow{e}$
33:        $s \leftarrow s'$
34:    until end of episode
35: until end of training
```

mechanism. This is in contrast to other methods that discover options during learning (Hengst, 2002; Iba, 1989; Randlov, 1999) and are tied to the learning algorithm. Our approach is more similar to work in macro actions (Whitehall, 1989; Pickett and Barto, 2002; Botea et al., 2005) where the macro-operator discovery was a *passive* component merely observing the actions of the learning agent, and suggesting macros at the end.

There are a number of different ways this allows us to use option discovery. As option discovery may get expensive for larger trajectories, training and trajectory generation can be applied to subsets of the whole data, and then learned options may be used on the whole dataset. This is also important since learning with only primitive proposers may not be possible on the whole dataset, and the options discovered on the subsets may be used to facilitate learning. A variation of such an approach was used in Botea et al. (2005). This approach is also useful when some powerful learning methods are not tractable on the whole dataset, but can be used to learn useful options on a subset of the training data that can be used to improve learning of a faster method on the whole dataset. For example, by learning options after training on reinforcement learning on a subset, we may obtain options that we can use in SampleRank that avert the delayed reward problem associated with it, as described in Section 1.

It should also be noted that the proposers and options are problem dependent, but are domain independent, and hence may even generalize across domains. For example, the proposers and the options may be defined on the abstract coreference/clustering problem, but can be learned from and applied to different domains, like name disambiguation, citation coreference, etc. We shall describe such an application in Section 4.

3.4 Option Discovery

Given a set of sample trajectories on the training data, the option discovery steps are divided into four steps that are described in detail in the following sections. As the option discovery method is explained, we also give an example of each step in Figure 1.

3.4.1 Dependency Graph

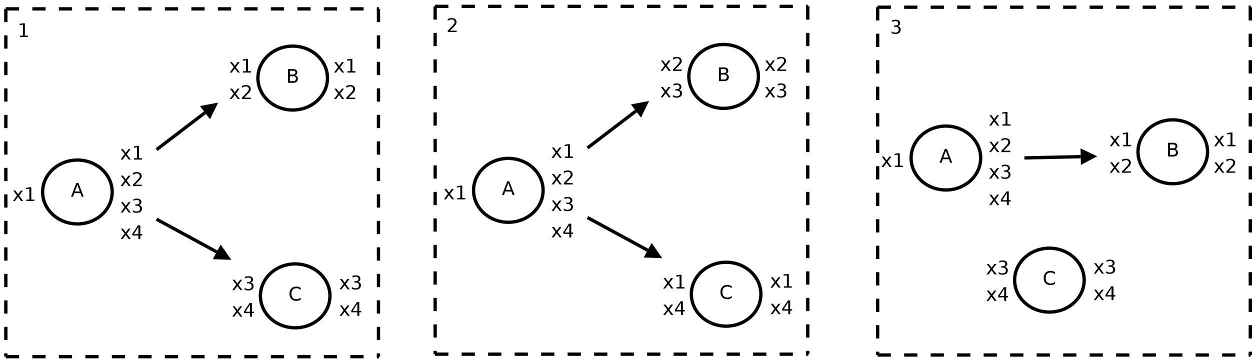
In most of the factor graph domains and corresponding proposal functions, every move involves a change to a small subset of the hidden variables. This results in moves that change subsets that are completely independent of each other, and hence those moves may appear in any relative order. This information is not captured by trajectories as each trajectory contains merely one of the possible orderings between the moves. Ideally, we would like to construct a directed acyclic graph that captures which move should come before which, and inherently represent independency between moves.

The problem of dependency graph creation can be addressed by looking at all the available trajectories, and for every pair of moves, deciding the presence and direction of an edge between the pair. This method is computationally slow, and highly sensitive to noise in the trajectories. Many moves may not appear often enough to reliably construct this graph.

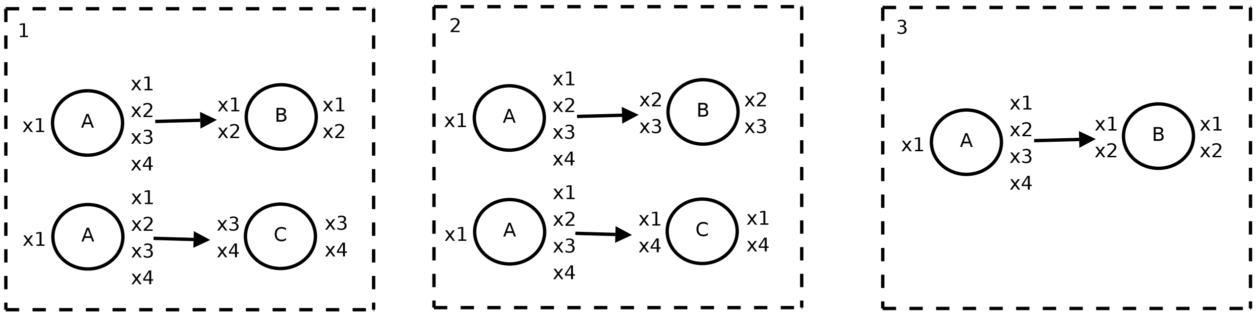
We use a simpler method of restricting the analysis to within the trajectories. Instead of treating each move as independent, we look at the input and the changed variables. A decision is made as follows: A move x is placed after move y iff at least one of the input arguments of x appears in the changed variables of y, and y comes before x in the trajectory. Thus one dependency graph is created for each trajectory. This algorithm runs in $O(nml)$ time, where m is the number of trajectories, n is the maximum number of variables changed in a move, and l is the maximum number of moves in a trajectory. Hence it is linear in the total number of moves in the trajectory.

1) $A(x1) \rightarrow \{x1, x2, x3, x4\}, B(x1, x2) \rightarrow \{x1, x2\}, C(x3, x4) \rightarrow \{x3, x4\}$
2) $A(x1) \rightarrow \{x1, x2, x3, x4\}, C(x1, x4) \rightarrow \{x1, x4\}, B(x2, x3) \rightarrow \{x2, x3\}$
3) $C(x3, x4) \rightarrow \{x3, x4\}, A(x1) \rightarrow \{x1, x2, x3, x4\}, B(x1, x2) \rightarrow \{x1, x2\}$

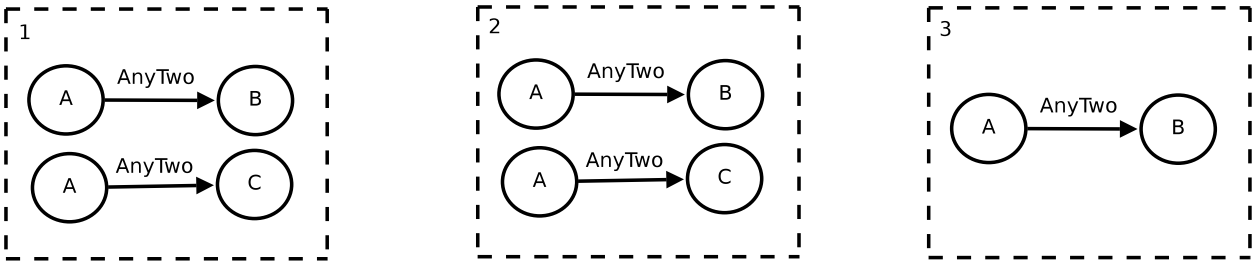
(a) Trajectories, each move written as *Proposer*(*input arguments*)→{*changed variables*}



(b) Dependency Graphs of the three trajectories



(c) Sequences extracted from the Dependency Graphs



(d) Abstracted Sequences

1) $A \rightarrow AnyTwo \rightarrow B$ Score: $2 \times 3 = 6$
2) $A \rightarrow AnyTwo \rightarrow C$ Score: $2 \times 2 = 4$

(e) Options

Figure 1: Example of option-discovery on three small trajectories, with Proposers A, B and C.

As an example, consider the trajectories given in Figure 1a. By looking at the inputs to moves using proposers B and C, we realize that they cannot come before moves using A for the first two trajectories since they depend on the output of A. For the last trajectory, C can come in any order compared to the rest of the moves. The corresponding dependency graph is shown in Figure 1b. Note that there is no generalization across moves or proposers.

3.4.2 Identification of Sequences

Once the dependency graph has been constructed for each trajectory, sequences of moves in the graph correspond to moves that necessarily have to follow one after the other as they depend on each other. Thus these sequences are ideal candidates for potential options.

The possible number of such sequences is still exponential, and hence any analysis that only analyzes one graph at a time shall be too expensive. Since we want the running time to be linear in the number of trajectories (or graphs), most methods that analyze all the graphs cannot be used either. This leads to restricting the classes of sequences that we use for options, specifically, we only consider sequences in which each move has a single parent, except for the first move, which has multiple or no parents. These sequences represent linear chains of moves where each move is caused only by the previous one, and directly causes the next. The first move may be the exception as it may cause a number of different moves. This class of sequences corresponds directly to our definition of options, as described in Section 3.1.

The number of such sequences is $O(l)$, since for each node in the graph, there is only one possible sequence which it is at the tail of. The algorithm hence runs in $O(nl^2)$, or more specifically, linear the number of nodes in all the extracted sequences. By using an upper threshold for the maximum sequence length, this reduces to $O(nl)$. Figure 1c shows the sequences of moves that are extracted from the dependency graphs given in Figure 1b.

3.4.3 Abstraction and Accumulation of Sequences

Until this step our method treats all moves as indistinguishable, and does not accumulate anything across trajectories or proposers. In this step, each of the sequences are abstracted, and accumulated across trajectories to evaluate them as candidate options.

A straightforward way to accumulate the moves across trajectories is to treat each tuple of input variables, proposers and changed variable as unique. Hence two sequences are considered the same if they contain the exact same sequence of moves, i.e. they changed the variables in the same order in every proposer in the sequence. This may work well for very small domains (like the one shown in Figure 1) but shall fail to generalize to domains containing larger numbers of variables, and highly stochastic proposers.

Instead of directly comparing moves, we would like to treat sequences as same if they are behaving the same, irrespective of the variables involved. To formalize the notion of the behavior of a sequence, we return to *Mapping* introduced earlier in Section 3.1. For every move x that follows y in any sequence, we identify a mapping that, given the changed variables of y, identifies a mapping to stochastically generate the inputs of x. Currently this involves searching a list of predefined mapping functions, and choosing the most specific mapping that applies. This is a simple algorithm that runs linearly in the total number of moves, and the maximum number of variables in a move. Figure 1d shows the abstracted sequences using the mapping AnyTwo that picks any two of the changed variables as input arguments. Two abstracted sequences are considered the same if their proposers and mappings are the same.

Once these abstract sequences (or candidate options) have been created, they are directly compared across trajectories and counted. By using hashing methods, this step is carried out in $O(k)$ amortized time, where k is the number of sequences, and the length of the sequences is upper bounded by a constant. Note that $k = O(nl)$.

3.4.4 Creation of Options

Once we have all possible candidate options with the number of times that they appear in the set of trajectories, each candidate's utility is evaluated. There are a number of learning method specific methods of evaluating utility of an option, for example, we can simply look at the accumulated reward for reinforcement learning. However, since our objective is to be agnostic to the learning method, we define our score based on the *cognitive savings* used in the macro-operator and option discovery work (Whitehall, 1989; Pickett and Barto, 2002). Cognitive savings score is simply defined as the number of occurrences of the option multiplied by it's length. An option that contains many actions and also occurs many times is useful.

Once the options have been scored, a suitable domain-dependent threshold is set to filter the options in linear time. These options can then be used as a normal proposer by specifying a MultipleProposer as defined in Section 3.1. The two candidate options for our example domain are scored and compared in Figure 1e.

By accumulating and scoring the form of the options that is independent of the variables, and taking the length of the options into account, we feel the effect of overfitting on training data are considerably reduced. If the proposers themselves are not domain specific (but may be problem specific) then the resulting options are likely to be useful across unknown data of same and different domains. However, since we restrict our analysis to trajectories, overfitting cannot be completely ruled out.

3.5 Usage of Options

Recall that an option is defined as a tuple $\langle \mathcal{I}, \pi, \beta \rangle$, where $\mathcal{I}$ are the initial states, π is the policy being followed while using this option, and $\beta(s)$ specifies the probability of termination of the option when in state s. We shall now describe the correspondence of MultipleProposers to options, and how the MultipleProposers are used.

Consider a MultipleProposer $\mathcal{M}$ with proposers $P_1, P_2, \ldots, P_n$ and mappings $m_{12}, m_{23}, \ldots, m_{n-1,n}$. It also has an associated probability of termination p_t. The option representing proposer $\mathcal{M}$ can be defined as follows:

- I: The initial set of this option is simply the set of states where a possible set of assignments to the arguments for P_1 exist. For example, if P_1 is the Shatter proposer mentioned earlier, the all singleton cluster state is not in the initial set of states.

- β: Since we are working with semi-markov options, β takes as an argument the history of the states that were encountered in this option. For i^{th} step, $\beta = p_t$ if $i \leq n$, else it is 1. β is also 1 if after any step i, P_i cannot accept any arguments using $m_{i-1,i}$ on the changed variables of P_{i-1}. For example, consider the case when P_{i-1} and P_i are both Shatter, and the mapping $m_{i-1,i}$ proposes any one of the changed variables in P_{i-1} as the input to P_i. Since the changed variables are singleton clusters, there are no valid inputs to P_i, and the option must terminate.

- π: As in the previous case, since the option is semi-markov, we allow the policy π to take the local history of the option into account. Specifically, given the current state s and the step number i, the probability distribution over the next states is given by $P_i\left(s, m_{i-1,i}\left(Y^{i-1}\right)\right)$, where Y^{i-1} are the changed variables by P_{i-1}, and P_i changes the configuration s using the arguments given to it by $m_{i-1,i}$.

Given a set of proposers that the proposal function uses, it picks a random proposer, ignoring whether it is a simple or a MultipleProposer. If a MultipleProposer is picked, it's sequence of proposers are called till any of the termination conditions are reached, proposing the change to the current state that is a combination of the changes proposed by all the constituent proposers. This is same as the way the options are employed in the options framework.

3.6 Limitations

One of the major objectives of the option discovery method is to be fast. Being an NP-hard problem, it is impossible to escape proposal of options that may not be very useful and add to the cost of sampling. Furthermore, as we discover options on sample trajectories, there is a possibility of overfitting, i.e. suggesting options that appear useful for the given trajectories, but do not generalize to the rest of the state space. For these reasons, we would like to evaluate the options, and add or remove options as may be required. This is motivation to make the discovery method fast, however, this speed comes at a cost. In most of our steps, we are making some strong assumptions, which we shall describe below.

The major speed up is obtained by using the dependency graph, which drastically reduces the combinations of actions that are considered as candidate options. As the dependency graph method runs independently for each trajectory and is based on observing the changed variables, random moves within a trajectory that happen to use the same variables may seem dependent due to noise. This can lead to options that may not be helpful. A possible solution is to explore all the trajectories while deciding the presence and direction of an edge, but this method is expensive.

Once the graph is constructed, the only sequences we consider are ones that are linear chains. It is possible to consider more complicated structures. For example, a decision-tree like structure can represent an option, where the next proposer is picked based on the results of the previous action. This leads to a exponential blowup, and is very sensitive to overfitting. A possible solution is to construct a general policy over the states and actions of the option.

Another assumption that reduces the running time from $\propto l^2$ to linear in l is the longest sequence assumption. When constructing sequences from the dependency graph, we use the longest sequences that can be constructed ending at a given proposer. For example for sequence $\langle a_1, a_2, a_3\rangle$, we consider sequences $\langle a_1, a_2\rangle$ and $\langle a_1, a_2, a_3\rangle$, and do not consider $\langle a_2, a_3\rangle$. This is a valid assumption as we would like the sequence of actions to be directly caused their previous one, except for the first action which may have multiple or no causes. For sequence $\langle a_2, a_3\rangle$, a_2 is caused by a_1, and thus including it in the sequence is reasonable.

A crucial step in our method is the process of mapping output variables of one action to the input arguments of the next. An exhaustive solution of this problem is to search through the space of possible mappings, and picking the one that matches a larger number of sequences. As an approximation we restrict our search using only the single pair of moves, and pick the mapping that samples the least. This allows us to discover options that are less stochastic, and hence are likely to propose good options from less samples.

These assumptions give us a huge benefit in asymptotic running time without hurting the utility, as shown in the following section.

Table 1: **Name Disambiguation dataset**

“Andrew McCallum”, “Andrew MacCallum”, “Angrew McCallum”, “McCallum”, “A. McCallum”
“Michael Wick”, “Mike Wick”, “Michael Andrew Wick”, “Wick”, “Wick”
“Khashayar Rohanemanesh”, “Khash R.”, “Kesh Rohanemanesh”
“Aron Culotta”, “Andrew Culotta”, “A. Culotta”, “Culotta McCallum”, “Culotta”, ”Culotta”
“Charles Sutton”, “Charles A. Sutton”, “Sutton”, “Sutton”

Table 2: Primitive Proposers used for the Coreference Domain

Proposers	**Input(s)**	**Description**	**Changed**
Shatter	c	Shatter cluster c into Singletons	Singleton clusters
MakeSingleton	m	Move m into it’s own cluster	m and it’s old cluster
MergeEntities	$c1, c2$	Move all mentions in $c2$ to $c1$	Mentions in $c1$
SimilarMention	m, c	Move mention closest to m in c to m’s cluster	c, moved mention

4 Results

The option discovery method was applied to trajectories obtained from SampleRank learning and MCMC inference. We describe the domains, our model, and the resulting options and improvements obtained by using them.

4.1 Name Disambiguation

To demonstrate the method, we first apply it to a synthetic toy coreference dataset of name disambiguation. Given a set of strings, the task is to cluster them such that each cluster of names refers to the same entity. The complete dataset is given in Table 1. Even though the dataset is small, it contains a lot of noise in the form of misspellings and overlapping clusters, making the task non-trivial.

There are two kinds of factors in the model, PairwiseAffinity and PairwiseRepulsion, each containing basic string comparison operations like matching, prefix comparison and containment. The former factor is instantiated between pairs of strings that are in the same predicted cluster, while the latter is instantiated between pairs of strings that are in different clusters. We include a number of proposers for the coreference domain which are listed in Table 2. Note that the notion of *distance* between mentions used in SimilarMention proposer uses the existing weights stored in

Table 3: **Name Disambiguation Results:** Average number of samples required and accepted moves for 50 runs of inference to reach within 3% of the asymptotic accuracy.

Proposers	**Samples Required**	**Accepted Moves**
Primitive Actions	615	31
+ Options	433	26

Figure 2: Example Options Discovered for Name Disambiguation

Options	**Description**	**Frequency**
Shatter-(AnyTwo)-MergeEntities	Shatter a cluster to form singleton clusters, but then look at the singleton clusters to find pairs that may be merged.	246
SimilarMention-(AnyTwo)-SimilarMention	Once $m1$ is used to extract $m2$ that is most similar to $m1$ in cluster c, the option suggests using either $m1$ or $m2$ to extract another mention from c.	189
SimilarMention-(AnyAll)-MergeEntities	Once a mention is moved into a cluster with a similar mention, find other clusters to merge it with	121
MergeEntities-(AnyOne)-MakeSingleton	Once two clusers are merged, the option decides to make singleton out of some mention in the merged clusters.	88
MakeSingleton-(AnyTwo)-SimilarMention	Once a mention m is extracted from a cluster c and made a singleton, the option suggests extracting the closest mention to m from c.	76

the factors, i.e. Distance(m_1, m_2)=PairwiseRepulsion(m_1, m_2)-PairwiseAffinity(m_1, m_2).

Training was performed for $50,000$ samples using SampleRank, with restarts initialized to random configurations. Inference was performed on the same training data for as many samples as it required to reach within 3% of the asymptotic F1[3], where F1 is the pairwise coreference decision F1 evaluation. The inference was repeated 50 times before aggregating the trajectories to create the options. The top two options were picked, and the experiment was repeated with options included in the set of proposers.

Since the data set is very small, getting high F1 is not a reliable benchmark by itself. Instead, we compare the average number of samples required to get a high score. These are shown in Table 3. Using options clearly requires less samples to reach the same accuracy.

The highest scoring options that were discovered are shown in Fig 2. AnyAll mapping is when the first input is used from any of the changed ones, while the second input is used from any of the rest of the variables. Only the first two options that are shown were used for result shown earlier. There were more options that were longer, but were not selected because of their low scores. Due to the size of the problem, the average length of the trajectory was 31. The utility of the method is evident from improvements shown by the options that were extracted from a small number of short trajectories.

[3]Asymptotic score is the ground truth score reached after 50,000 samples for training and testing.

4.2 Citation Coreference

The option-discovery method was applied to a real world citation matching Cora dataset. Cora dataset, created by Andrew McCallum, contains reference strings for research papers, and has been labelled for coreference and segmentation. We used the cleaned and segmented version[4] used in Poon and Domingos (2007) and available online. The dataset contains 1295 total mentions in 134 clusters, with a total of 36487 tokens.

The task of coreference refers to the task of identifying the citations that refer to the same paper. It is modeled the same way as described earlier for the generalized coreference problem, and as the name disambiguation problem. The two types of factors that are instantiated are the same as before. The features of the factors are based those used in isolated coreference in Poon and Domingos (2007). These features do not take into account the segmentated fields that is part of the data. The proposers for this task were the same as the ones used for the name disambiguation task (Table 2).

The Cora dataset was randomly split into three folds in a way that ensures mentions of the same ground truth cluster are in the same fold. Three fold cross validation was performed with five times. Each involved another 5 loops of training with 150,000 samples and 10 loops of inference till convergence. Thus overall 150 inference runs were performed.

Table 4: **Cora Coreference Results:** Average number of samples required and accepted moves for 50 loops of 3-fold cross validation runs of inference to reach within 5% of the asymptotic F1. Average F1 reached after 150 runs of 100,000 inference samples each are also shown.

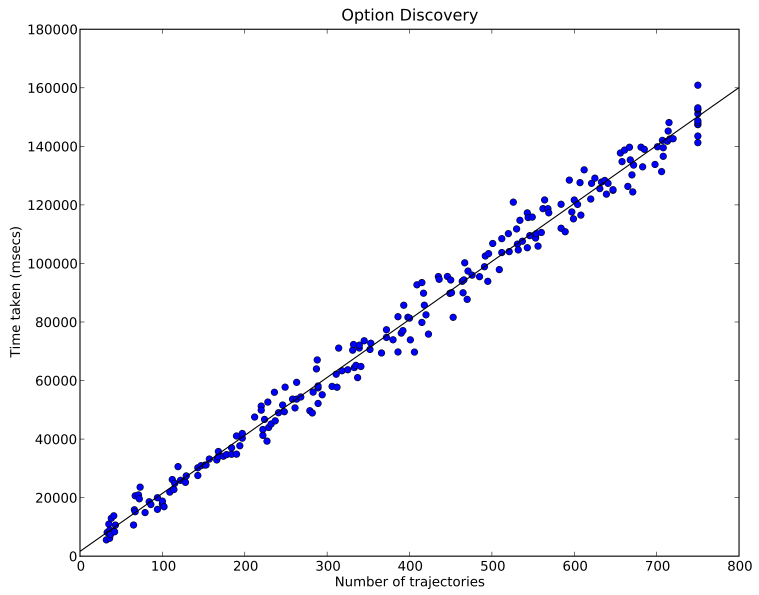
Proposers	**Samples**	**Accepted Moves**	**F1**
Primitive Actions	69203	780	92.4
+ Name Disambiguation Options	54078	746	92.8
+ Citation Matching Options	39791	699	93.3

Along with extracting options from the citation matching trajectories, the options used in the name disambiguation options were also used. Since the options are domain independent (any task that uses the same proposers can use the options discovered from those propsers), the options could be easily included. As earlier, we compare the number of samples required, and accepted, to reach 5% of the asymptotic score on the test set. We also compare the average score reached after 100,000 MCMC steps. These results are shown in Table 4. All improvements were found to be statistically significant at the 1% level using McNemar's test.

On convergence (200,000 steps for this domain), all methods were equivalent in terms of the F1 of the configuration reached. The benefit of options discovered by our method reduces sampling needed to explore the space, providing "shortcuts" to reach regions that have a high value. It may not be possible to discover options that perform better than SampleRank asymptotically, since SampleRank has been shown theoretically to converge to the MAP configuration with few restrictions on the proposers (Culotta, 2008).

[4]http://alchemy.cs.washington.edu/papers/poon07

Figure 3: Running Time of Option Discovery



4.3 Timing Results

The option discovery method was found to be very fast at discovering options. For name disambiguation it took a few seconds for 50 trajectories, each containing an average of 30 moves. For the Cora coreference task, 150 trajectories containing a total of 104923 moves caused the option-discovery algorithm to run for 35 seconds on average. This is acceptable running time compared to the time taken for training and inference for on the dataset (15 minutes). Since our method is fast, it is conceivable to include it as within a loop such that it dynamically adds, evaluates and retracts options.

To study the asymptotic running time, a larger set of trajectories was obtained from the Cora coreference task and the option discovery algorithm was run on random subsets. The increase in running time is almost linear in terms of number of trajectories, as can be seen in Fig 3. The black line denotes the best linear regression fit to the data (Correlation Coefficient, $r^2 = 0.9891$).

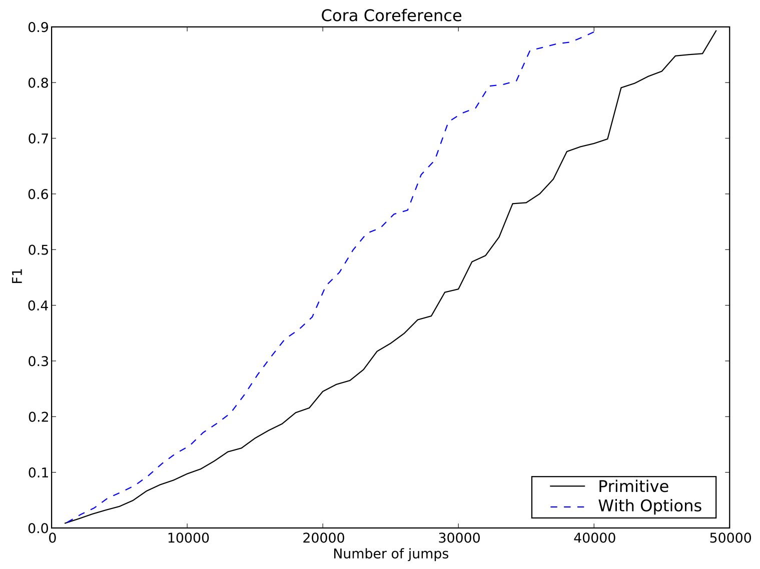
Adding options to the set of proposers did not significantly change the running time for fixed number of samples. Even though each sample with options involves multiple moves, the expensive part of evaluation is unrolling the factor graph and scoring the factors. For our experiments, the options did not create significantly larger factor graphs per sample as compared to using just primitive actions.

4.4 Discussion

The aggregate results shown above demonstrate how the options reduce the required number of samples, and reach convergence sooner. In this section we shall study the inference behavior when options are used, and contrasted with not using options.

An sample plot of the pairwise F1 during inference when starting from the same initial configuration is shown in Fig 4. If the initial slope is examined, we can see that the options help a lot initially, reaching states close to the goal very quickly. Once states close to the goals are reached,

Figure 4: Plot of Test F1 during Inference



it is better to propose smaller changes so that the configuration stays close to the goal. Thus, in this region, primitive actions seem to be more useful, and hence the utility of options is lost. The plots towards the end of inference for both the methods have similar slopes for this reason.

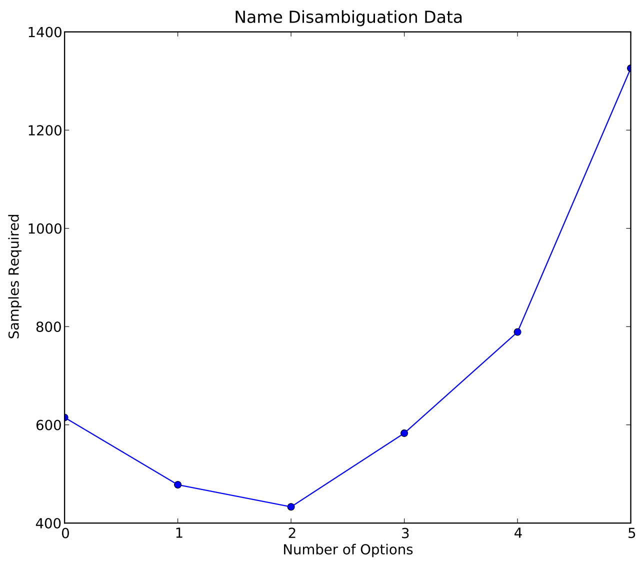
This tradeoff between cost of adding additional options versus benefit of bigger jumps provided by the options can be observed by looking at the behavior as number of options is varied. This has been shown by averaging over 50 runs in Fig 5 for the Name Disambiguation task. For some number of options (4 for our task), the utility of adding options is exceeded by the cost of sampling the options, and hence the performance becomes worse than using just the primitive actions. It is not clear how the ideal number of options can be chosen; a possible solution is to keep adding the highest scoring options until the desired accuracy is reached, or the performance starts degrading.

5 Related Work

Framing the MAP inference as a factor graph was first studied in Rohanimanesh et al. (2008). This formulation is similar to Zhang and Dietterich (1995) where they propose a reinforcement learning framework for solving combinatorial optimization problems. Similar to this approach, we also rely on generalization techniques in RL in order to directly approximate a policy over the unseen test domain. However, our formulation provides a framework that explicitly targets the MAP problem in large factor graphs and takes advantage of the log-linear representation of such models in order to employ a well studied class of generalization techniques in RL known as linear function approximation (Tadić, 1999; Precup et al., 2001). Learning a generalizable function approximator has been also used in standard search algorithms for efficiently guiding the search by learning evaluation functions from experience (Boyan and Moore, 2001).

Lot of the work in automated discovery of options does not take the reward into account. This allows them to generalize the same options to different reward functions, i.e. different goal states. QCut (Menache et al., 2002) identifies subgoals in the problem by finding bottleneck states in the

Figure 5: Varying the Number of Options Added



complete transition graph. LCut (Simsek et al., 2005) extends the work by only looking at sampled incomplete transition graph. Mannor et al. (2004), instead of identifying bottlenecks, clusters the states of the sampled transition graph, and trains options that reach neighboring clusters. In comparison, RN (Relative Novelty, Simsek and Barto (2004)) identifies states as subgoals if they allow *access* to state spaces that is novel to the agent. A similar idea of *landmarks* is used in Butz et al. (2004) to detect subgoals. This notion of *access* or *bottleneck* states does not apply to our transition graph for a couple of reasons. First, since our transition graph is very big, QCut cannot be applied, and a lot of exploration is required before meaningful sample graphs are obtained for LCut and RN. Second, we cannot assume existence of such *access* states or clusters since our transition graph is locally regular over the complete configuration space, unlike domains studied by these methods. Our method exploits the structure in the state space induced by the reward function.

There are a number of option-discovery methods that are used during or after learning that take the reward function into account. Randlov (1999) extends the usual reinforcement learning framework to include the previous action into the value function, i.e. $Q(s_t, a_{t-1}, a_t)$. This results in the choice of current action directly affecting the choice of the next action. From the usual options perception, this corresponds to a "two-step" option, which severely restricts applications where more complicated options are required. HEXQ (Hengst, 2002) analyses the frequency with which the state variables change, and constructs a hierarchy in which each level corresponds to a state variable. States that change the value of that variable are the sub-tasks for that level. HI-MAT Mehta et al. (2008) uses dynamic Bayesian network models to discover hierarchies for MAXQ (Dietterich, 2000) from trajectories. HI-MAT, HEXQ and other hierarchy discovery methods (Jonsson and Barto, 2006) rely on the changes to the feature values to detect action dependencies and subtasks. For our problem the only features that are calculated are between two states, and the features appearing in the state are only those that changed. Thus, a feature-based method may work in our domain.

Policy Blocks (Pickett and Barto, 2002) method of option-discovery is similar to our method. The options are discovered by identifying sequences that are common to a number of samples on the same underlying MDP, and is not concerned with subgoal-discovery. They also the *cognitive savings* score to filter options. However, they do not use action dependencies to prune the space to search to discover these sequences, leading to an exponential blowup. Also, the discovered options do not generalize to different domains. By restricting our method to the factor graph domain, we have sacrificed application to all reinforcement learning problems, but achieve signficant speedups.

Due to our definition of proposers and options, our work is similar to that of macro-operator discovery in planning. PLAND (Whitehall, 1989) is a macro-operator (or macro-action) discovery that looks at single long sequence of actions, and identifies substructures in it that can be used as macros. It does not restrict itself to sequences, but includes *loops* and *conditionals*. PLAND also filters out options using the *cognitive savings* measure. However, they consider the given sequence as strictly linear chain of events, which might lead to noise in the output. By constructing dependency graph, we ensure a lot of this noise is removed, and that we construct options from actions that appear related. MACLEARN(Iba, 1989) is one of the early macro-operator discovery algorithms that views actions as relational productions, and macros include mapping bindings of actions within the macro to each other. It requires deterministic actions, and performs searches to find the macros. More recently, our method is similar to Macro-FF (Botea et al., 2005). Macro-FF improves performce of AI Planners by searching for macro-operators on a simpler domain, and applying them to the complex domain. The system analyzes previous experience on a task and exploits domain structure.

Our work extends existing option discovery literature in reinforcement learning, and macro-action discovery work in planing, to design a fast and efficient algorithm to construct options for the novel task of MAP inference in factor graphs.

6 Future Work and Conclusions

This work frames the problem of MAP inference in factor graphs as a hierarchical reinforcement learning problem. This formulation allows design of *options* that can be used to overcome limitations of state of the art learning methods applied to this domain. We also present a very fast option-discovery method that was shown to improve speed on the citation matching dataset. However, there are a number of issues that need to be addressed before the method has widespread applicability.

Options that are discovered in this work are obtained by studying the trajectories composed of primitive actions. As a trivial extension, we can create options at a higher level of abstraction by analyzing trajectories consisting of options, which shall result in options composed of lower level options. This repeated application of the option discovery method is useful to evaluate and prune previously suggested options, while expanding the space of possible options that are being considered. This is aided by our formulation of the proposers, which is closed under combination. Since the proposed option discovery method is fast, it will not be a bottleneck in such a system.

One of the artifacts of analyzing trajectories to construct options is the bias of option discovery towards initial regions of state space. The initial region contains a lot of accepted moves since there are many possible good moves. The options worked very well in this region, quickly proposing good changes that subsume multiple primitive moves. Once the states near the goal are reached, however, accepted moves are rare. Hence the trajectories do not contiain ample data about this region to construct useful options. In future, this work can be extended to *weigh* the individual sequences according to where they appear in the trajectory, giving a higher weight to the tail. This shall reduce the number of samples required near the goal, possibly leading to a higher overall accuracy.

Our method relies on the domain knowledge that has been injected into the proposers and the mappings that are used to construct options. This is not demanding since proposers that are independent of the restrictions imposed on MCMC proposal functions can be easily designed. Options, on the other hand, can be tough to design without knowing the behavior of the underlying region of state space where the primitive Proposers are failing. This problem of specifying options is more important in joint inference of tasks, where domain experts of individual problems may be able specify options for their domain, but may not be able to design good options that span multiple tasks. In future, we would like to apply this method to joint inference tasks, and evaluate its utility compared to hand-crafted options.

References

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-ff: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res. (JAIR)*, 24:581–621.

Boyan, J. and Moore, A. W. (2001). Learning evaluation functions to improve optimization by local search. *J. Mach. Learn. Res.*, 1:77–112.

Butz, M., Swarup, S., and Goldberg, D. (2004). Effective online detection of task-independent landmarks. In *Online Proceedings for the ICML'04 Workshop on Predictive Representations of World Knowledge.*

Culotta, A. (2008). *Learning and inference in weighted logic with application to natural language processing.* PhD thesis, University of Massachusetts.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with hexq. In *ICML '02: Proceedings of the Nineteenth International Conference on Machine Learning*, pages 243–250, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Mach. Learn.*, 3(4):285–317.

Jonsson, A. and Barto, A. (2006). Causal graph based decomposition of factored mdps. *J. Mach. Learn. Res.*, 7:2259–2301.

Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Mannor, S., Menache, I., Hoze, A., and Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 71, New York, NY, USA. ACM.

McCallum, A., Rohanimanesh, K., Wick, M., Schultz, K., and Singh, S. (2008). Factorie: Efficient probabilistic programming via imperative declarations of structure, inference and learning. In *Neural Information Processing Systems(NIPS) Workshop on Probabilistic Programming*, Vancouver, BC, Canda.

Mehta, N., Ray, S., Tadepalli, P., and Dietterich, T. (2008). Automatic discovery and transfer of maxq hierarchies. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 648–655, New York, NY, USA. ACM.

Menache, I., Mannor, S., and Shimkin, N. (2002). Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *Machine Learning: ECML 2002, 13th European Conference on Machine Learning, volume 2430 of LectureNotes in Computer Science*, pages 295–306. Springer.

Parr, R. and Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 1043–1049, Cambridge, MA, USA. MIT Press.

Pickett, M. and Barto, A. G. (2002). Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning. In *ICML '02: Proceedings of the Nineteenth International Conference on Machine Learning*, pages 506–513, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Poon, H. and Domingos, P. (2007). Joint inference in information extraction. In *AAAI Conference on Artificial Intelligence*, pages 913–918.

Poon, H., Domingos, P., and Sumner, M. (2008). A general method for reducing the complexity of relational inference and its application to mcmc. In Fox, D. and Gomes, C. P., editors, *AAAI*, pages 1075–1080. AAAI Press.

Precup, D., Sutton, R. S., and Dasgupta, S. (2001). Off-policy temporal difference learning with function approximation. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 417–424, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Punyakanok, V., Roth, D., and tau Yih, W. (2005). The necessity of syntactic parsing for semantic role labeling. In *IJCAI*, pages 1117–1123.

Randlov, J. (1999). Learning macro-actions in reinforcement learning. In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pages 1045–1051, Cambridge, MA, USA. MIT Press.

Rohanimanesh, K., Wick, M., Singh, S., and McCallum, A. (2008). Reinforcement learning for map inference in large factor graphs. Technical report, University of Massachusetts.

Simsek, O. and Barto, A. G. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 95, New York, NY, USA. ACM.

Simsek, O., Wolfe, A. P., and Barto, A. G. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 816–823, New York, NY, USA. ACM.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, pages 9–44.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* The MIT Press.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Tadić, V. (1999). Convergence analysis of temporal-difference learning algorithms with linear function approximation. In *COLT '99: Proceedings of the twelfth annual conference on Computational learning theory*, pages 193–202, New York, NY, USA. ACM.

Watkins, C. J. (1989). *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Wellner, B., McCallum, A., Peng, F., and Hay, M. (2004). An integrated, conditional model of information extraction and coreference with application to citation matching. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 593–601, Arlington, Virginia, United States. AUAI Press.

Whitehall, B. L. (1989). Substructure discovery of macro-operators. In *Proc. of the Fifth Conference on Artificial Intelligence Applications CAIA-89*, pages 231–238, Miami, FL.

Wick, M. and McCallum, A. (2008). Approximate learning for conditional random fields with mcmc. Technical report, University of Massachusetts.

Zhang, W. and Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120. Morgan Kaufmann.

Zhang, W. and Dietterich, T. G. (2000). Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. *Journal of Artificial Intelligence Reseach*, 1.