A Critique of the dissertation "Concurrent Hierarchical Reinforcement Learning"

Max Robinson MAX.ROBINSON@JHU.EDU
Johns Hopkins University,
Baltimore, MD 21218 USA

1. Introduction

(Marthi, 2006)

2. Related Work

3. Summary

Marthi touches on a myriad of topics as he discusses the research conducted in his thesis. Starting with background on "Flat" reinforcement learning, Marthi explains the foundations of the Semi-Markov Decision Process (SMDP), and partial programs upon which the research is built.

The first major contribution of the dissertation, the Concurrent ALisp language, is then explained. A description of the implementation follows. How learning algorithms are applied in Concurrent ALips then follows. These learning algorithms use the early foundations to describe how the learning using Concurrent ALisp is executed.

The experimental results then backup the principles of learning using Concurrent ALisp. The primary themes of partial programs, reward decomposition, and scalability are all tested.

3.1 Background

3.1.1 MDPs and SMDPs

The research done by Marthi, along with much for the reinforcement learning field, builds upon Markov decision processes or MDPs (Puterman, 1994) (Bertsekas & Tsitsiklis, 1996). MDPs are often used to model sequential decision making processes. An MDP can be defined as a tuple $M = (S, A, P, R, s_0)$. Each value of the tuple is defined as follows.

- S - state space
- A - action space
- P - transition distribution
- R - reward function. A function that maps a state, action, and next state $R(s, a, s')$ to a member in $\mathbb{R} \cup -\infty$
- s_0 - initial start state

For an MDP to be an accurate representation of the problem, two general properties have to be met or assumed. First, the current state must be derivable just from the last perception of the environment but the agent. Second, the Markov property is assumed. The Markov property states that the probability of entering a given state next only relies on the current state and the action taken from that state. No prior history before that state is taken into account.

Markov decision processes can be solved or estimated with a multitude of different algorithms and approaches. The solution to an MDP is known as a *policy*, denoted by π. A policy describes what actions an agent should take when in a given state. Two types of policies to focus on are stationary and non-stationary policies.

A stationary policy is one in which it depends only on the last state, $\pi(s)$. A non-stationary policy is one in which the action decision relies on additional information than just the current state. Marthi focuses on non-stationary policies as he notes that most hierarchical reinforcement learning breaks agent behavior into tasks. As a result, the goal of an agent might not be recoverable from just the environment state.

From MDPs, a modified version called a semi-Markov decision process (SMDP) can be described. An SMDP is an MDP that also includes a duration distribution for each state action pair. The reasoning behind adding a duration is that actions can take some amount of time to complete. From a hierarchical standpoint with tasks, one might imagine that a task takes a certain amount of time. The SMDP is build to incorporate that duration into the model.

3.1.2 Partial Programs

A goal for Marthi's research is to allow programmers to easily incorporate background knowledge into learning algorithms. To do this Marthi introduces ALisp (Andre & Russell, 2002) and partial programming. Partial programs aim to help programmers incorporate background knowledge while writing a program to learn based on the partial program.

A partial program can thought of as constraints on which policies can be searched for while learning. A policy that finds an optimal policy given these constraints can be considered hierarchically optimal. ALisp is a language in which partial programs can be written. In ALisp, the foundations for Concurrent ALisp, the construct of a choice statements is added to allow for non-determinism in the program execution. This nondeterminism is used for selecting which set of statements to run next and what actions to take in the environment. The program then corresponds to a set of policies that can run the program and choose what to do at the choice statements. The choices made at each statement use a *completion* of the program.

Partial programs when combined with an underlying MDP create an SMDP, called the induced SMDP. The states in the SMDP are derived from the state of the MDP and the *machine state*. The machine state can be thought of as program specific information about the partial program during execution. Actions in the SMDP are the choices made at the choice statements. The actions taken in the program then might result in some action being taken in the underlying environment. The reward for the SMDP is the reward gained during all actions between two choice statements.

By creating an SMDP from the partial program, algorithms for SMDPs can be applied to the induced SMDP. Marthi concludes that finding the hierarchically optimal policy corresponding to the partial program is equivalent to finding the optimal stationary policy for the induced SMDP. This means that by looking at the SMDP, a hierarchically optimal execution of the partial program can be found. Since the partial program corresponds to constraints on policies for the MDP, finding a hierarchically optimal execution for the partial program means that a hierarchically optimal policy has been found for the MDP.

Partial programs also allow for decomposition of the Q-function for learning algorithms. The Q-function can be thought of as the "action-value function". The Q-function is the expected reward if we start at $\mathbf{s}$, a history of states, and do action a, then follow π. For partial programs, the Q-function can be decomposed into three components, Q_r, Q_c, Q_e. These are the expected reward received while doing the current choice, after the current choice until the subroutine ends, and after the current subroutine respectively.

The motivation behind decomposing the Q-function is to simplify the learning process. Each part of the decomposition may only rely on a particular subset of variables that are part of the state. In these cases, state abstraction can be leveraged, and thus reduce the sample complexity of learning, since each value is learned independently.

3.2 Concurrent ALisp (CALisp)

In the case of multiple agents or a multieffector MDP, ALisp alone is no longer suitable for solving these problems. To attempt to make ALisp work with multiple agents, the partial program would then have to be written to account for multiple agents and require a much more complicated program with book keeping. This is due to the fact that the subroutine nature of ALisp is lost because each agent might be doing something different at every time step. Separate ALisp programs could be run instead to maintain the subroutine nature but in doing so it assumes there are no coordinated actions at a low-level.

Concurrent ALisp is designed to solve these problems. This is Marthi's first major contribution. To sidestep the flaws of a large ALisp program or separate ALisp programs, Concurrent ALisp takes a multithreading approach to solving the problem.

The multithreading approach allows for the partial program to spawn new threads and specify a specific subroutine that is then executed in that thread. When the subroutine finishes, the thread dies. These threads will run independently from one another until a choice or action statement is reached.

When a thread reaches a choice statement, the thread pauses execution and waits for other threads to arrive at a choice or action statement. All threads that are at a choice statement then make a joint choice. This choice is based on the *completion* that is provided to the partial program.

Threads also wait for each other at action statements. Once all threads are at an action statement, a joint action is taken. If some threads are at choice statements and others are at action statements, threads at the action statements wait, while threads at choice statements choose and continue until they are at an action statement.

Unlike separate programs, these joint choices allow for coordination between subroutines. The CALisp partial programs also retains is subroutine structure, while taking care of all the bookkeeping for threading. This allows for slight modifications to the program to be

made in order to adapt to multiple agents, rather than a complex rewrite. A multithreaded structure also exposes a threadwise parallel structure in addition to temporal structure which is later taken advantage of in the learning algorithms.

Marthi goes on to define the formal structure of CALisp in his dissertation. The reader is referred to the thesis for exact details on the formalization of CALisp. The formalization describes how one might implement CALisp.

3.3 Implementation of CALisp

As his second major contribution, Marthi created an implementation of his CALisp language. The implementation of CALisp for this thesis was written in Lisp. This section of the dissertation is similar to code documentation and language specifics. This includes things like specifying types, function calls, parameters for functions, objects, and required methods for these objects. As a result, readers are referred to the thesis for details of the exact implementation rather than including a summarization of the language particulars in this critique.

At a high level, the important part about the language implementation is the idea of extensibility. CALisp was implemented in an object-orient paradigm. Based on this paradigm, the implementation focused on defining objects that were necessary for any partial program to run, and objects that could be extended for additional functionality. Key examples of objects that were designed to be extensible are the Q-function, approximation architectures, and learning algorithm objects. These objects are described with a base implementation but also layout the functions needed and the behavior expected if one were to extend the object.

Extensibility is important when implementing a language. Rarely is a user going to want only the pre-built functions of a language. This would mean that a user of the language would have little control over the behavior of objects native to the language and has little recourse in building their own objects to try and modify those behaviors. By allowing for users to extend objects, the user is given much more control over how a partial program in the language might behave.

An example of extending a CALisp object is the ability to create a new approximation architecture. The language implementation comes with a default linear approximator. If a user wanted to instead use a neural network, the user could extend the approximation architecture object and implement the required methods. This is a very important feature for CALisp in order to allow CALisp to be used by others and to extend the life of CALisp.

3.4 Learning Algorithms in CALisp

Several algorithms have been developed to learn an optimal policy in a flat MDP. Two popular algorithms that are also leveraged by Marthi are Q-learning (Watkins & Dayan, 1992) and SARSA (Sutton & Barto, 1998). Both build on the idea of dynamic programming and temporal-difference methods to find a solution to the MDP. Temporal difference methods work by storing an estimation of the value function for the policy, $\hat{V}^{\pi}$. Then after an action is taken according to the policy, an update of the estimated value function is applied according to equation 1

$$\hat{V}^{\pi}(s) \leftarrow \hat{V}^{\pi}(s) + \eta(r + \hat{V}^{\pi}(s') - \hat{V}^{\pi}(s)) \tag{1}$$

where η is a learning rate and r is the reward observed.

Q-learning uses a temporal difference method but to estimate the optimal Q-function given an observation of a state, action, reward, and new state. The update performed is

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \eta(r + \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)) \tag{2}$$

where η is again a learning rate. A discount factor can be added to the equation to model the idea of future rewards being less valuable then current a reward.

SARSA is similar to Q-learning, but rather than updating the Q-function estimation based on the maximum value of (s', a'), the update is made according the estimated Q-function determined by the current policy, $\pi(s', a')$. This makes SARSA an on-policy algorithm while Q-learning is an off-policy algorithm. The SARSA update is described as

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \eta(r + \hat{Q}(s', \pi(s',a')) - \hat{Q}(s,a)) \tag{3}$$

where a' is chosen according to the policy.

These algorithms, while designed for MDPs can be modified for SMDPs by adding a discount term that scales according to the duration of the task. For Q-learning an updated equation 2 for SMDPs would be

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \eta(r + \gamma^{d} \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)) \tag{4}$$

where d is the duration of the task.

The modifications of these algorithms for SMDPs are important because, as discussed earlier, the partial programs written in CALisp with an underlying MDP can produce an induced SMDP. Thus, a simple way to write a learning algorithm for CALisp programs is to use an existing algorithm on the induced SMDP from the partial program.

The SMDP Q-learning algorithm could thus be used in CALisp to learn the optimal stationary policy on the SMDP. As a reminder, the optimal stationary policy of the SMDP is equivalent to the hierarchically optimal completion of the partial program. This then corresponds to a hierarchically optimal solution for the MDP. As a result, a hierarchically optimal solution can be found to the MDP by finding the optimal stationary policy of the induced SMDP.

3.4.1 Decomposed Learning

While applying a standard SMDP algorithm to CALisp works, Marthi points out that there is a better way. For the SMDP Q-learning, rewards are not decomposed in any way which means that should there be a positive and negative reward from different subroutines, all subroutines are penalized according to the combined rewards. In addition, the decomposition of the Q-function that is available as part of a partial program in ALisp is lost in CALisp because subroutines now run in separate threads.

However, these problems are solved if a programmer can provide a way of decomposing the reward signal into per-thread rewards. This allows specific threads, and thus subroutines, to be associated with particular rewards. This then allows a threadwise decomposition of the Q-function across subroutine boundaries, like in ALisp.

In order to make a threadwise decomposition, the variability in the number of threads has to be taken into account. Since threads are spawned only from other threads, this can be boiled down to taking into account a threads future rewards as well as rewards form uncreated descendants of the that thread. This is done by calculating the Q-function for a given thread by taking into account the relevant descendants of a thread. The reward received from the relevant descendants of threat t for a set of combined states (called a trajectory) $\vec{\omega}$ can be defined as a random variable $\chi^t(\omega)$. The threadwise Q-function can then be defined as

$$Q_t^{\psi}(\omega, u) \triangleq E[\chi^t(\vec{\omega})] \tag{5}$$

for a state ω, and a choice u based on a completion ψ. This means that the Q-component for t is the expected total reward from t and all descendants of t.

This then allows the combined Q-function for a CALisp program is proven by Marthi to be the sum of the threadwise Q-components in the current state. This is described as

$$Q^{\psi}(\omega, u) = \sum_{t \in T(\omega)} Q_t^{\psi}(\omega, u) \tag{6}$$

As a result, Marthi claims, an algorithm that learns the Q_t corresponding to an optimal completion for all threads that could be seen during execution, the optimal completion can be recovered from the learned Q_t's. This means that learning algorithms can target learning more specific Q-functions for subroutines possibly reduce the complexity of learning without losing the ability to learn the optimal completion.

A temporal decomposition can then be defined per thread decomposition, similar to the ALisp partial program decomposition of the Q-function discussed in 3.1.2. The biggest difference is that the temporal decomposition of the Q-function relies heavily on the implementation of CALisp since use of the machine state is used to determine which parts of the reward should be incorporated into which of the temporal Q-components.

The decomposition results in temporal Q-components per thread of $Q_{t,r}$, $Q_{t,c}$, $Q_{t,e}$. $Q_{t,r}$ is the reward gained doing a particular action, or rewards all rewards gained until a choice block. $Q_{t,c}$ is subsequent rewards in the current choice block. $Q_{t,e}$ is the rewards outside the choice block.

These temporal Q-Components are then shown to equal the total threadwise Q-component by

$$Q_t^{\psi}(\omega, u) = Q_{t,r}^{\psi}(\omega, u) + Q_{t,c}^{\psi}(\omega, u) + Q_{t,e}^{\psi}(\omega, u) \tag{7}$$

This then allows a learning algorithm to focus even more narrowly on learning just the temporal decomposition of the threadwise Q-function, while still corresponding to an optimal completion of the partial program.

3.4.2 SARSA Implementation

Marthi provides and implements two SARSA variants, one with threadwise decomposition and the other with temporal decomposition. Threadwise decomposition of SARSA is proved to converge to the optimal global Q-function, based on work done by Russell and Zimdars (?), and Singh et al. (?). Russell showed threadwise updates are equivalent to standard SARSA updates. Singh wrote the SARSA convergence theorem.

A conjecture is made that the temporally decomposed SARSA will converge the a optimal global Q-function. However no proof is provided. Even for ALisp, though, the temporal decomposition of SARSA has not been proved to converge.

3.5 Experimental Results

Three sets of experiments were run using CALisp, each with a different focus. The first focuses on the effect of incorporating prior knowledge through a partial program on learning. The second focuses on the performance between threadwise decomposition, temporal decomposition, and undecomposed algorithms. The third measures the scalability of algorithms.

3.5.1 Prior Knowledge experiment

To test the role of prior knowledge in the form of a partial program, five different partial programs are compared, labeled 1-5. Each algorithm adds slightly more prior knowledge than the last and the fifth uses incorrect prior knowledge.

1. equivalent to flat reinforcement learning.
2. specifies task hierarchies with no order.
3. adds some task ordering information
4. includes ordering information along with coordination information.
5. includes "faulty prior knowledge"

The environment for the test is a 3x2 grid world, where two peasants try to gather two of each type of resource. The resources are gold and wood. Despite the small world, it has over 4500 states and 49 actions per state.

The learning algorithm used is a tabular representation of Q-Learning. The tabular representation was used to avoid any possible benefit to learning outside of the partial program, such as a good function approximator or reward shaping. This is the reason why a small example is chosen for the tabular representation. Even though it is a small example, the Q-function tables still has over 2 x 10^5 entries.

The results of the experiment show that with increasing prior knowledge in the partial program, the rate of convergence to an optimal policy also increased. Partial program number 4 converged the quickest, while partial program number 1 took the longest. Partial program 5 learned quickly, but ultimately converged to a suboptimal policy because of the incorrect prior knowledge.

3.5.2 Decomposed learning experiments

Three learning algorithms are tested to explore the role of decomposed learning. The algorithms compared are an undecomposed, threadwise decomposed, and temporally decomposed Q-learning algorithms. For each algorithm, linear approximation and reward shaping were used to make learning more tractable. Boltzman exploration was used for the exploration policy for all algorithms.

Again, the environment is a resource gathering problem. In this case, the world is a 20x20 grid with 6 peasants. The goal was to gather 10 gold and 15 wood.

The results demonstrate that temporal decomposition converged to the optimal policy the fastest, very closely followed by threadwise decomposition. The difference between the two is difficult to see in the resulting graph provided, but the authors analysis seems believable. An explanation for the similar performance is based on the task tested having a relatively shallow task hierarchy. It is suggested that temporal decomposition might perform noticeably better than threadwise decomposition in these deeper hierarchy. However, no experimental results are provided on this front.

3.5.3 Scalability experiment

Two experiments are conducted to test scalability: varying the number of peasants to collect resources, and varying the size of the world. Again, the goal is for peasants to collect resources in the environment. The algorithm used in the experiment is the temporally decomposed learning algorithm from the previous experiment.

Varying the number of peasants in a 15x15 grid world showed two stages of performance. For a small number of peasant learning took fewer steps but grew fairly quickly as the number of peasants increased to about 10 peasants. After 10 peasants the number of steps it takes to learn remains relatively similar for the rest of the number of peasants up to 40, which was the max tested. The rate of growth in learning time (in steps), after 10, looks roughly linear with the number of peasants.

The reasoning given for the two different rates or learning is due to the task. Collisions provide a large negative reward, so for few peasants the learner might just need to learn to avoid collisions. As the number of peasants increase though, the learner might have to learn not to send multiple peasants to the same resource.

In varying the map size, grids with side length of about 2, 5, 10, 20, 40, 60, and 100 were used with 5 peasants. The results show that as the map grows the number of steps needed until optimality increases in a roughly learn fashion with the size of the grid side length.

Reasoning as to this linear growth is based in the time it takes to get updates at the higher level of choices. Each choice probably takes longer because the grid is getting larger. This then creates fewer high-level updates to the Q-function, thus needing more steps to learn an optimal policy.

4. Contributions

5. Relevant Algorithms

6. Applications of Concurrent Hierarchical RL

7. Conclusion

References

Andre, D., & Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Eighteenth National Conference on Artificial Intelligence*, pp. 119–125, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming* (1st edition). Athena Scientific.

Marthi, B. M. (2006). *Concurrent Hierarchical Reinforcement Learning*. Ph.D. thesis, Berkeley, CA, USA. AAI3253978.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st edition). John Wiley & Sons, Inc., New York, NY, USA.

Sutton, R. S., & Barto, A. G. (1998). *Introduction to Reinforcement Learning* (1st edition). MIT Press, Cambridge, MA, USA.

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