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 policy can be found.

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.

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.