CS 533 Project II

Loren B. Davis

March 23, 2012

This paper describes a library to solve problems in a parking-lot microworld. (Summary of the problem description omitted.) The author implemented the solver using model-based reinforcement learning, in Haskell. This paper discusses some limitations and issues of scalability.

1 Introduction

This paper assumes familiarity with the assignment and the author's implementation of Mini-Project I. Mini-Project II builds on the architecture of Project I, with some refactoring. Most significantly, the code is now separated into three modules, with the implementation of part I (very similar to the implementation of project I) in `MDP.hs`, the implementation of part II in `MDP-Sum.hs` and the implementation of part III in ADP.hs. The code for part II calls the code from part I, and the code from part III calls the code from parts I and II. As at one point the author profiled the code for part II and added bang patterns (forcing strict rather than lazy evaluation of some function arguments, where it would help), GHC therefore requires the `-XBangPatterns` flag to compile it. The author tested this code by first running `make` to compile the modules to optimized code, then `ghci -Wall -O2 -funbox-strict-fields -fllvm -msse2 -fobject-code -XBangPatterns ADP.hs` to run interactively. It would also be possible to compile with the `-e` switch, as the `lessBad` example in the makefile does. The makefile also contains an example of compiling with profiling turned on.

This project has three parts. As the first is virtually identical to the first mini-project, and so is the implementation, the author refers you to his previous paper. The major addition are a handful of utility functions to convert internal data into a human-readable form.

The second part involves simulation, and is covered in Section 2. The third involves reinforcement learning, and is covered in Section sec:reinforcement.

The source code uses a literate style of programming, and therefore full documentation for each of the functions below can be found in the comments there.

2 Policy Simulation

The MDPSim module implements a library to simulate arbitrary MDPS, and also a number of utility functions and a handful of examples related to the the parking problem specifically.

2.1 Simulating Arbitrary MDPs

The most important functions for this purpose are simulateMDP, bandit and evaluateParking-Policy.

The MDPSim.simulateMDP function takes an MDP, an agent policy, and several other parameters, such as the initial and terminal states (assumed to be unique, without loss of generality), and simulates a single run of the agent through that MDP. Its output is a list of (State, Action, Reward) triples representing the agent's situation at each time step and the next action it chose.

The MDPSim.bandit function takes an MDP, a discount factor γ, initial and terminal states, an agent policy, and runs the agent once through that MDP following that policy. Instead of returning detailed information about the results of the run, it calculates the time-discounted cumulative reward.

The MDPSim.evaluateParkingPolicy function runs an agent through a given number of simulations, starting from a randomly-determined state, then averages the cumulative rewards to measure the quality of a policy. Specifically, it generates a position in front of any parking space with uniform probability, and then simulates the drive action once, so that the agent is equally likely to start out in front of any space, and the probability that the space will be taken is realistic.

2.2 Application to the Parking Problem

The MDP.ParkingProblem function generates MDP descriptions of arbitrary parking problems; there are a few examples in the file, of which MDPSim.mdp4spaces is small enough to be easily verified by a human, and MDPSim.mdp10spaces is an interesting but reasonable size. Also helpful is the MDPSim.optimalPolicy function, which uses the policy iteration function from project I to solve a given MDP, then converts its output into a policy of the form expected by this project. The corresponding optimal policies for these examples are MDPSim.pi4spaces and MDPSim.pi10spaces.

The library also contains functions for generating random and handcrafted policies. The randomPolicy *r p* function will generate a policy for the parking problem with *r* rows that always parks with probability *p* and drives otherwise. Improving on this, neverCrashPolicy will always drive past a filled space, and lessBadPolicy will never parked in a handicapped space, will always park on an empty space in row 2, and will park in any other empty space in row *i* with probability p/i.

2.3 Examples

A sample run of an agent through the smaller example, using the optimal policy:

```
*ADP> simulateMDP mdp4spaces pi4spaces ( encodeState 2 (A 1 False) ) 0 10000
[(6,0,-1.0e-2),(10,0,-1.0e-2),(11,1,-1.0e-2),(3,1,0.5),(0,1,0.0)]
*ADP> decodeResults 2 it
[(A 1 False,"Drive"),(B 1 False,"Drive"),(B 2 False,"Park"),(P 2,"Park"),(Exit,"Park")]
```

The second line puts the results, one per time step, into human-readable form. The agent starts in row A, space 1, with the space empty. The optimal policy does not park in this spot, which is a handicap space. It drives to row B, space 1, which is also a handicap space. It keeps driving to row B, space 2, and finds it empty. It parks there, transitions to the state where it collects its reward for parking in row 2, and there to the terminal state. We can refer to the first line for the rewards: we accumulate small penalties of size ι until we park, then a reward of size $\frac{1}{2}$ for parking in row 2, then zero reward in the terminal state.

Here are some examples of policy evaluation:

```
*ADP> evaluateParkingPolicy mdp10spaces 0.98 (randomPolicy 10 0.6) 10000
-30.904503535201588

*ADP> evaluateParkingPolicy mdp10spaces 0.98 (neverCrashPolicy 10 0.6) 10000
-1.8747353656452381

*ADP> evaluateParkingPolicy mdp10spaces 0.98 (lessBadPolicy 10 1.0) 10000
4.4101490388976275e-2

*ADP> evaluateParkingPolicy mdp10spaces 0.98 pi10spaces 10000
0.1472353798151275
```

We see that a policy of randomly parking 60% of the time is terrible. Simply never crashing makes it substantially better, and the simple handcrafted policy barely breaks even. Finally, the optimal policy is about three times as good as the simple handcrafted policy.

2.4 Implementation Issues

By far the most difficult aspect of implementing this project in Haskell was the need to program with monads. It is impossible to do input, output or, as it turns out, probability in Haskell without encapsulating computations in monads. The only one-sentence description of monads the author was able to get Dr. Martin Erwig to accept was, "An object that obeys the rules of monads."[1] Be warned that the extremely simplified description of the purpose they serve in this project is "not the most general case." For some reason, very few people who are capable of understanding monads ever bother. As a result of this project, the author now personally understands monads much better than before; that is, precisely as well as he needed to in order to complete this project.

[1]Personal conversation, March 2012. And no, he didn't put me up to it.

One thing that monads can do (But not the most general!) is encapsulate computations that have to occur in a specific sequence within what is otherwise a pure functional language, in which no other computations have side-effects. For example, input and output have to occur in proper sequence because they have side effects, and the program would otherwise produce garbled nonsense. It so happens that Haskell's implementation of random numbers also has side-effects, because it alters the state of the random-number generator, even when the programmer does not require deterministic pseudo-random number generation and is totally indifferent to the order of calls to the RNG. This means that any randomly-generated double-precision number is not a `Double`, but an `IO Double`, which is a number that needs to be wrapped in a monad whose computation has side-effects. Furthermore, any computation that uses that value, such as the action some agent chooses in a given state, now becomes a computation that has side-effects as well, and so does anything that refers to it, in a chain that works like quarantine. This requires a substantial amount of very complicated glue to interact with anything: one thing you can specifically *not* do is turn the monadic computation into a precomputed value and then use it normally. So, one winds up writing code such as the following:

```
evaluateParkingPolicy :: MarkovDP -> Double -> Policy -> Int -> IO Double
evaluateParkingPolicy !mdp !gamma !policy !k = do
  utilities <- replicateM k ( bandit' mdp gamma 0 (initialState mdp) policy )
  return ((sum utilities) / fromIntegral k)
```

This code is hardly lengthy (APL would almost be proud), nor is its internal logic that complicated when one understands what it does. The `bandit` function expects an initial state argument, not an initial-state-whose-computation-has-side-effects argument, and a randomly-generated initial state has side effects because it's randomly-generated. So, as syntactic sugar, the program implements a new function, `bandit'`, which is just like `bandit` except that it expects its initial-state argument to be wrapped in a monad. Its *raison d'être* is specifically to simplify the above code. The first line of the function body runs the bandit computation k times on different initial states, with non-deterministic results, and concatenates the outcomes into a list of cumulative rewards whose computation has side effects. It stores that as `utilities`, which one can use as if it were an unadorned list in some but not all contexts within the function. The second line takes the mean of this list and returns it as a `Double` whose computation has side-effects. This is, however, sufficiently different from pure Haskell that figuring out which two lines to use is much trickier than it looks.

There was also one strange optimizer bug in `ghci 7.0.3` that caused a list of results to appear in reverse order; it was easier to slightly tweak the source so as not to expose it than to attempt to debug it.

3 Reinforcement Learning

Part III of the exercise concerned a reinforcement-learning agent. This implementation used an ε-greedy, model-based approach: the learning process read in a policy π and transformed it into an ε-greedy policy, which followed π with probability $1 - \varepsilon$ and chose an action uniformly at random with probability ε. It then ran that ε-greedy policy through k runs of a simulator (on which more below) and used the results to calculate a model of the reward and transition functions. Specifically, it made the assumption that R is a function of state and treated states never reached as having reward 0, and based its estimate of the transition probabilities $T(s, a, s')$ on the proportion of agents that started out in state s, took action a and transitioned to state s'. If a state-action pair was never explored, this implementation made the arbitrary decision to model it as going directly to the terminal state, which turned out to have consequences that will be discussed in section 3.2. It also assumes that the terminal state always transitions to itself. It then solved that model using the policy-iteration algorithm from part I and returned the optimal policy for that model.

The most important function in this module is `ADP.epsilonGreedyAgent`, which learns an approximation of the optimal policy. Also important is `ADP.fromMDP`, which converts an MDP into a simulator, which is an object that takes an agent policy and returns the results of one run through the world represented by that MDP, but completely hides the internal details of its model to the caller. (The current version of the algorithm does need to be told how many states and actions there are, but not to use the simulator.) In this implementation, the caller cannot specify an initial state. Otherwise, the logical course of action would be to simulate every state-action pair sufficient times to be confident that the learning process has derived the full transition function, which seemed less in the spirit of the assignment than what `epsilonGreedyAgent` currently does.

3.1 Examples

This code runs the reinforcement-learning process over 1,000 runs, starting with a random policy on the same 10-row example as in section 2.3.

```
*ADP> let simulator = fromMDP mdp10spaces
*ADP> let initialPolicy = randomPolicy 10 0.5
*ADP> let learnedPolicy = epsilonGreedyAgent 52 2 0 initialPolicy 0.98 simulator 0.0 5000
*ADP> epsilonGreedyAgent 52 2 0 initialPolicy 0.98 simulator 0.0 1000
```

We omit the output of this, but store it in a variable so we can refer to the same policy later rather than generate a different one:

```
*ADP> let learnedPolicy = it
*ADP> evaluateParkingPolicy mdp10spaces 0.98 learnedPolicy 10000
-5.3288869977143216e-2
```

The results are variable, sometimes slightly worse than the handcrafted policy and sometimes slightly better. What happens when we feed our learned policy back into the process for another 1,000 runs?

```
*ADP> epsilonGreedyAgent 52 2 0 learnedPolicy 0.98 simulator 0.2 1000
```

[Output redacted]

```
*ADP> let learnedPolicy' = it
*ADP> evaluateParkingPolicy mdp10spaces 0.98 learnedPolicy' 10000
3.253799040606868e-2
```

Although this appears to beat the handcrafted policy, the difference in performance between the learned policies and the handcrafted policy are not statistically significant. To get a significantly better result, one must run substantially longer, in this case, 10,000 iterations. Unfortunately, and especially in interpreted code, the algorithm does not scale well with k:

```
*ADP> epsilonGreedyAgent 52 2 0 learnedPolicy' 0.98 simulator 0.2 10000
```

[Output redacted]

```
*ADP> let betterPolicy = it
*ADP> evaluateParkingPolicy mdp10spaces 0.98 betterPolicy 10000
0.1171507299993456
```

This is two orders of magnitude better than before, and only $\approx$ 20% worse than the optimal policy. Even so, where is this difference coming from?

3.2 "Limitations"

Any difference in policy performance must be attributable to a difference in the actions chosen by the policy. Let us see when our policies tell us to park (slightly reformatted for clarity):

```
*ADP> decodePolicy 10 learnedPolicy
[Exit,Crash,P 1,P 2,P 3,P 4,P 5,P 6,P 7,P 8,P 9,P 10,
A 8 True,A 9 True,A 10 True,
A 2 False,A 3 False,A 4 False,A 5 False,
B 8 True,B 9 True,B 10 True,
B 2 False,B 3 False,B 4 False,B 5 False,B 6 False,B 7 False,B 8 False,B 9 False]
*ADP> decodePolicy 10 learnedPolicy'
[Exit,Crash,P 1,P 2,P 3,P 4,P 5,P 6,P 7,P 8,P 9,P 10,
A 8 True,A 9 True,A 10 True,
A 2 False,A 3 False,A 4 False,A 5 False,
B 9 True,B 10 True,
B 2 False,B 3 False,B 4 False,B 5 False,B 6 False,B 7 False,B 8 False,B 9 False]
*ADP> decodePolicy 10 betterPolicy
[Exit,Crash,P 1,P 2,P 3,P 4,P 5,P 6,P 7,P 8,P 9,P 10,
A 9 True,A 10 True,
A 2 False,A 3 False,A 4 False,A 5 False,
B 9 True,B 10 True,
B 2 False,B 3 False,B 4 False,B 5 False,B 6 False,B 7 False,B 8 False,B 9 False]
*ADP> decodePolicy 10 pi10spaces
[Exit,Crash,P 1,P 2,P 3,P 4,P 5,P 6,P 7,P 8,P 9,P 10,
A 2 False,A 3 False,A 4 False,A 5 False,
B 2 False,B 3 False,B 4 False,B 5 False,B 6 False,B 7 False,B 8 False,B 9 False]
```

Recall that the last policy is optimal. We can ignore the cases `Exit`, `Crash` and `P` *i*, as actions taken here are irrelevant; the implementation of policy iteration happens to pick the park action in such cases by default. We are interested in the cases labeled: column letter, row number, full or not. As you see, the policies all correctly deduce when to park in an empty space, but sometimes crash the car in a filled space at the far end of the row, and the more training the agent had, the further out it has to be to make this mistake. You now have enough information to diagnose the bug.

What happened (one silver lining of writing the code as a monad is that it's easy to insert instrumentation for debugging) is that the probability of a space being full decreases with distance. The agent will rarely drive that far out without parking somewhere else, unless it starts there, which it rarely will. Furthermore, an agent policy that isn't brain-dead won't take the park action in that situation, so that already-low probability will only occur when the agent additionally chooses to explore and then randomly picks the park action, with probability $\varepsilon/2$. What is going on here is that the agent has little or no training data for what would happen if one of those spaces were full, so it defaults to the assumption that the simulation would end with no further reward. The learning process has no domain-specific knowledge, and so doesn't know that the safe action is to drive rather than park. So, by an implementation quirk, it defaults to trying to park. It would be possible to fix this, but not in any general way. These are rare events, so they only reduce the score by a small amount, but the more of them there are, the more likely at least one of them is to occur. Their prevalence could be reduced by increasing the number of trials, increasing the probability for rare events to happen in the training data, or changing the problem itself so that there are fewer surprises such as this.