CS 533 MINI-PROJECT I

Loren B. Davis

March 3, 2012

This author implemented the project, a solver for Markov decision processes, in Haskell using a policy iteration algorithm. The paper presents a brief summary of the algorithm and program, as well as the results of several test cases and the solution to one model of the parking problem for certain reasonable assumptions, explained herein.

CAL SCR

1 Introduction

While this paper will not provide an in-depth discussion of the implementation details (for which see the attached code listing[1]), and assumes familiarity with the theory of Markov decision processes and policy iteration in particular, an explanation of how the former derives from the latter is in order, especially as this author used a subtly-different form of policy iteration from the one in the text[2]. This paper also assumes that the reader has a copy of the assignment[3].

Rather than reinvent the wheel, the final program uses the `hmatrix` Haskell package, which adds Haskell bindings to the system's linear algebra libraries. You might therefore need to install `libgsl` and `liblapack`, then run `cabal install hmatrix`, to test the program.

2 Policy Iteration

"For small state spaces," Russell and Norvig write in section 17.3, "policy evaluation using exact solution methods is often the most efficient approach." This is the approach taken here, after

[1] `LorenDavis-cs533-proj1.hs`

[2] All references in this paper are to Chapter 17 of (Russell & Norvig 2010). Bibliography omitted for brevity.

[3] http://www.engr.orst.edu/~afern/classes/cs533/hw/mdp-mini-project.pdf. Retrieved 10 February 2012.

testing on a problem with a slightly larger number of states and a known solution and determining that exact policy iteration was entirely feasible. For problems with many more states, a different solver would scale better.

For our purposes, a Markov decision process consists of n states in S, m actions in A, rewards as a function $R(s)$ of the current state, and a transition function $T(s, a, s')$ that returns the probability that taking action a in state s will put the agent in state s'. We also look only at infinite-horizon MDPs here, with discount factor γ.

Policy iteration starts with a policy, which is a function returning an action to take next. Under the assumptions we've made, the optimal policy for this model is provably stationary (*q.v.* section 17.1.1). Therefore, a policy is a mapping from states to actions. Our goal is to find a policy π that maximizes expected reward, over time and discounted by γ. We also need to introduce a utility function, $U_\pi(s)$, which returns the expected reward for following policy π starting in state s.

A bird's-eye view of policy iteration is that, given a policy π_i before iteration i, we determine the values of the utility function $U_i(s)$ for every state s if we follow policy π_i. Using these new utilities, we can then find, for each state, the action that maximizes expected utility. The actions we select for all states, based on our updated utilities, become our new policy, π_{i+1}. When this process reaches a fixed point, and we can no longer improve our policy, the algorithm halts.

2.1 Exact Policy Iteration as Linear Algebra

During each iteration, the policy iteration algorithm uses only single-step lookahead. This simplifies the Bellman equation (17.5) to

$$U_i(s) = R(s) + \gamma \sum_{s' \in S} T\left(s, \pi_i(s), s'\right) U_i\left(s'\right)$$

As the values of R, T and π_i are all known, this gives us a system of n linear equations with n unknowns, the values of $U_i(s)$.

$$\begin{aligned} U_i(s_0) = &\quad R(s_0) + \gamma T\left(s_0, \pi_i(s_0), s_0\right) U_i\left(s_0\right) + \ldots + \gamma T\left(s_0, \pi_i(s_0), s_{n-1}\right) U_i\left(s_{n-1}\right) \\ &\vdots \\ U_i(s_{n-1}) = &\quad R(s_{n-1}) + \gamma T\left(s_{n-1}, \pi_i(s_{n-1}), s_0\right) U_i\left(s_0\right) + \ldots + \gamma T\left(s_{n-1}, \pi_i(s_{n-1}), s_{n-1}\right) U_i\left(s_{n-1}\right) \end{aligned}$$

With a bit of rearrangement, this becomes

$$\begin{aligned} U_i(s_0) - \gamma T\left(s_0, \pi_i(s_0), s_0\right) U_i\left(s_0\right) - \ldots - \gamma T\left(s_0, \pi_i(s_0), s_{n-1}\right) U_i\left(s_{n-1}\right) = &\quad R(s_0) \\ &\vdots \\ U_i(s_{n-1}) - \gamma T\left(s_{n-1}, \pi_i(s_{n-1}), s_0\right) U_i\left(s_0\right) - \ldots - \gamma T\left(s_{n-1}, \pi_i(s_{n-1}), s_{n-1}\right) U_i\left(s_{n-1}\right) = &\quad R(s_{n-1}) \end{aligned}$$

If we take $\boldsymbol{I}$ to be the $n \times n$ identity matrix, $\boldsymbol{A}$ to be the matrix whose element in column j of row k is $\left[\gamma T\left(s_k, \pi_i(s_k), s_j\right) U_i\left(s_j\right)\right]$, $\boldsymbol{X}$ to be the column vector containing $U_i(s_0), \ldots, U_i(s_{n-1})$, and $\boldsymbol{B}$ to be the column vector containing $R(s_0), \ldots, R(s_{n-1})$, then we can express this in linear algebra as

$$(\boldsymbol{I} - \boldsymbol{A})\, \boldsymbol{X} = \boldsymbol{B}$$

We can solve this for $\boldsymbol{X}$ in $O\left(n^3\right)$ time, although because of the typical representation of terminal states, these systems are often underconstrained, preventing us from using fast methods such as LU-decomposition. The above is the only relevant equation not spelled out in section 17.3 of the text, and it, rather than the modified policy evaluation algorithm given there, forms the basis of this solver.

3 Implementation in Haskell

As specified, the inputs to the library call consist of a MDP description in the form $(|S|, |A|, R, T)$ and returns its output as a tuple (U, π, O) where U is the utility function, π the optimal policy, and O a human-readable list of each iteration's updates. Since this implementation finds an exact solution rather than an approximation, there is no error bound to report; the updates are pairs (s, a') from $S \times A$, representing the values $\pi_{i+1}(s) = a'$ that changed from the previous iteration. Since, for models with many states and actions, the return value becomes a large and cryptic expression, the library provides several utility functions to extract information in human-readable format, for example displaying only the nonzero entries of the usually-sparse transition matrices, or converting between the integer representation of a state and a more semantically-rich form, such as `A 2 True` in the parking-lot world.

For efficiency's sake, the program represents states internally as `Int` and functions as unboxed arrays. It makes judicious use of sequencing and memoization to eliminate overhead. The heart of the implementation is a tail-recursive function that takes in the MDP, the value of γ, and the current approximation π_i to the optimal policy. It also adds its updates to the list of previous updates. This algorithm returns, but does not take as input, a utility function U.

One difference between this implementation and the algorithm as described in the text is that the initial policy π_0 is arbitrary and deterministic, rather than random. Specifically, it initially takes the first action listed in each state. This eliminated the complexity of dealing with stateful computation in a pure functional program, and the algorithm still converges.

For the step involving linear-algebra, the program uses the `hmatrix` packages' Haskell bindings to the `LAPACK` library.

4 Test Cases and Microworlds

Modularization and a front-end for the program were not priorities at this stage; the source file is intended to run inside Haskell's read-evaluate-print loop. It does come with several different tests, most of which were exercises we had previously solved.

4.1 A Three-State Microworld

The smallest of these was the toy MDP described in exercise 17.10, and assigned as a homework problem. This world is so micro that the output is easy for a human to verify without processing.

Figure 1: The solution for exercise 17.10

```
*MDP> policyIterate test3 1.0
(array (0,2) [(0,7.4999999999999964),(1,-2.4999999999999973),(2,-4.999999999999998)],
array (0,2) [(0,1),(1,1),(2,0)],
["Step 1: take action 1 in state 0; take action 1 in state 1.",
"Step 2: Done."])
```

Interpreting this, we can verify its correctness: the agent wants to reach the terminal state (which here is state 0, not 3). This has a higher value than either state 1 or 2, and state 2, with twice the negative reward, also has twice the negative value. The optimal policy is, if in state 1, to attempt to reach state 0, and if in state 2, to attempt to reach state 1, as we determined on the homework.

One interesting result, which the author perhaps should have anticipated but did not, is that reducing the discount factor does not appear to change the ratio of utilities between the states. Instead, this reduces both the penalty for being in state 1 and the penalty for being in state 2 by the same amount.

4.2 A 4 × 3 Microworld

A second test was a slightly-simplified, deterministic version of the microword in figure 17.2 of the text. This was slightly less trivial a state space, and tested different code paths that required multiple iterations.

Figure 2: The solution for the 4x3 microworld

```
Step 1: take action 3 in state 0; take action 2 in state 1;
take action 1 in state 3; take action 3 in state 4;
take action 3 in state 5; take action 3 in state 7;
take action 3 in state 8.
Step 2: take action 1 in state 2; take action 2 in state 6.
Step 3: take action 1 in state 1; take action 2 in state 10.
Step 4: take action 2 in state 5; take action 1 in state 9.
Step 5: take action 1 in state 8.
Step 6: Done.
```

To provide some explanation for figure **??**, everything on step 1 is noise except for "take action 1 in state 3." This means that the solver has figured out that an agent in the square adjacent to the upper right corner should move there. In the next stage, the model updates the states to the

left and below to move one step closer to the positive reward. In the stage after that, the model propagates that insight to the states adjoining those, and so on, essentially following Dijkstra's algorithm until it reaches every state on the board. Note that, because this version of the MDP is deterministic rather than stochastic, there is no reason for the solver to avoid moving near the terminal state with the negative reward, and therefore the middle squares on the bottom row take this path instead of the one in the book. Similarly, utilities depend on distance from the goal state, and not on proximity to the dead-end state.

4.3 A 101 × 3 Microworld

The third test case was the microworld in figure 17.14(b), which is to say, the 101×3 microworld from the first homework. This both tested the scalability of the problem to one with roughly four times as many states as the one it was intended to solve, and, since we had already found the critical value of the discount function that determined its behavior, this made a good test that it was properly computing the optimal path.

The output of this test case is very verbose, but suffice it to say that, for $\gamma = 0.9843$, the optimal policy recommends action 0 (up) in state 0 (the home state), whereas for $\gamma = 0.9844$, the optimal policy recommends action 1 (down). The breakpoint is in fact $\gamma \approx 0.984\,398$. As expected, the values of states along the upper track steadily increase, while those along the lower track steadily decrease, with 50-point jumps past the entry points of both.

5 The Parking Domain

The encoding of the problem involved some slight differences from the suggested approach. First, it eliminated certain redundant states and actions; a car could be in any column of row A or B, and the space next to it could be open or free. Alternatively, it could be parked between 1 and n spaces from the store. This encoding used only $5n$ states, rather than $8n$. Exit and crash were special states, rather than separate actions, and either parking or crashing moved the agent to the exit state automatically. Exit, the only terminal state, had no reward and no escape.

It quickly became obvious that a large parking-domain model would be too tedious to encode by hand. Fortunately, we have computers to do that for us. Therefore, the program contains a function, `parkingProblem`, which generates a MDP model of a parking-domain problem with user-specified parameters.

5.1 Random Variables in the Model

One decision deserves particular elaboration. A reasonable first-order approximation for the behavior of other drivers is that they enter at a certain rate, park in the closest non-handicap space, and leave a fixed amount of time later. This is a Poisson process, and the number of other cars on the lot at any given time therefore has the distribution $X \sim \text{poisson}(\lambda)$, where λ is also the expected number of cars at any one moment. Let $F(x)$ be the cumulative distribution

function of X. As we're assuming that spaces fill up from column 2 outward, the probability that there will be space A 3 or B 3 is free is the probability that there are fewer than three cars parked in the lot (and thus that spaces A 2 and B 2 can hold them), plus half of the probability that there are exactly three, and the third is equally likely to be in either. This is the average of $F(2)$ and $F(3)$. The program calculates F from the probability mass function $f(x) = \frac{e^{-\lambda}\lambda^x}{x!}$, making use of memoization to avoid redundant computations.

5.2 Generating a Parking Problem

The following parameters define the MDP of a parking problem:

The number of rows is n. The probability that a handicap spot will be free is α. The reward for parking there, which will be negative, is β. The discount function γ is, to be pedantic, a parameter separate from the MDP, but can be considered part of the model. The reward for parking a given distance from the entrance, a positive number, is δ. The model arbitrarily represents this reward as inversely proportional to distance. The penalty for driving around without parking, which here is a small *positive* number, is ι. The reward for crashing into a parked car, a very large negative number, is κ. As previously mentioned, the expected number of other cars on the lot is λ.

Figure 3: The Transition Function for a four-space parking lot.

```
*MDP> decodeT 2 . showT $ parkingProblem 2 0.9 (-10) 1 0.01 (-100) 2
[((Exit,"Move",Exit),1.0),((Exit,"Park",Exit),1.0),
((Crash,"Move",Exit),1.0),((Crash,"Park",Exit),1.0),
((P 1,"Move",Exit),1.0),((P 1,"Park",Exit),1.0),
((P 2,"Move",Exit),1.0),((P 2,"Park",Exit),1.0),
((A 1 True,"Move",B 1 True),9.999999999999998e-2),((A 1 True,"Move",B 1 False),0.9),
((A 1 True,"Park",Crash),1.0),((A 2 True,"Move",A 1 True),9.999999999999998e-2),
((A 2 True,"Move",A 1 False),0.9),((A 2 True,"Park",Crash),1.0),
((A 1 False,"Move",B 1 True),9.999999999999998e-2),((A 1 False,"Move",B 1 False),0.9),
((A 1 False,"Park",P 1),1.0),((A 2 False,"Move",A 1 True),9.999999999999998e-2),
((A 2 False,"Move",A 1 False),0.9),((A 2 False,"Park",P 2),1.0),
((B 1 True,"Move",B 2 True),0.7293294335267746),
((B 1 True,"Move",B 2 False),0.2706705664732254),
((B 1 True,"Park",Crash),1.0),((B 2 True,"Move",A 2 True),0.7293294335267746),
((B 2 True,"Move",A 2 False),0.2706705664732254),((B 2 True,"Park",Crash),1.0),
((B 1 False,"Move",B 2 True),0.7293294335267746),
((B 1 False,"Move",B 2 False),0.2706705664732254),
((B 1 False,"Park",P 1),1.0),((B 2 False,"Move",A 2 True),0.7293294335267746),
((B 2 False,"Move",A 2 False),0.2706705664732254),((B 2 False,"Park",P 2),1.0)]
```

The minimal example in figure **??** demonstrates how this process works: the agent may either move or park. Moving takes it to the next state in the cycle, which will either adjoin an open space (False) or one that's been taken (True). The odds of this depend, for column 1, on α, otherwise, on the column number and λ, but note that the handicap space is much more likely to be free. Parking next to a free space transitions to the corresponding P i state, and from there to the Exit state no matter what; parking next to a filled space transitions to the Crash state. In either case, the agent collects its just deserts. The values of R are automatically-generated the same way.

6 Results

By altering these parameters, we can easily see how the agent adjusts its behavior to its new expectations. As a base case, we use the parameters:

```
*MDP> whenToPark 10 $ policyIterate (parkingProblem 10 0.9 (-10) 1 0.01 (-100) 2) 0.98
[A 2 False,B 2 False,B 3 False,B 4 False,B 5 False,B 6 False]
```

This code extracts the list of states in which the optimal policy would park (excluding noise such as irrelevant choices in terminal states). Note that it never parks in a handicap spot, and will pass up all but the best spot in row A because the front of row B is coming up soon, and it expects a better space there to be open.

6.1 Utilities

We omit a detailed breakdown of the utilities; suffice it to say that, in general, the As have significantly higher utilities than the Bs, since an agent in an A state will tend to be moving toward more desirable spaces, and Bs away, and that being next to a free space is much better than being next to an occupied one if your policy would take it. In this case, for example the state A 2 False has utility 0.48, compared to only 0.29 for A 2 True and 0.14 for B 6 true. This intuitively makes sense: if you're in front of the best space in the lot, and it's open, you're in luck. Even if it's taken, you'll be passing the second-best space next, then the third. Conversely, if the first six spaces in row B are all taken, you'll still have to circle around the far end of the lot before you can come back to the front.

6.2 Varying the Parameters

```
*MDP> whenToPark 10 $ policyIterate (parkingProblem 10 0.9 (-10) 1 0.01 (-100) 5) 0.98
[A 2 False,A 3 False,A 4 False,B 2 False,B 3 False,B 4 False,B 5 False,B 6 False,
B 7 False,B 8 False]
```

Increasing λ from 2 to 5 cars on average in the 20 spaces has a dramatic effect: the agent will now park in any of the first 3 non-handicap spaces in row A, or any of the first 7 in row B. It is now less willing to hope for a better space, particularly if those are behind it.

```
*MDP> whenToPark 10 $ policyIterate (parkingProblem 10 0.9 (-10) 1 0.01 (-100) 2) 0.95
[A 2 False,A 3 False,B 2 False,B 3 False,B 4 False,B 5 False,B 6 False,B 7 False,
B 8 False]
```

Decreasing the discount factor from 0.98 to 0.95 makes the agent weigh the short-term reward of parking in the near future more heavily, and the longer-term gain of finding a better space later less heavily.

```
*MDP> whenToPark 10 $ policyIterate (parkingProblem 10 0.9 (-10) 1 0.02 (-100) 2) 0.98
[A 2 False,A 3 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]
```

Doubling ι, the cost of driving, makes the agent less willing to drive: it will now take the two best spaces in A, and all but the worst space in B.

I note in passing that I can get the agent to take a handicap space by changing the reward to be inversely proportional to the *square* of the distance and making the parameters somewhat extreme; it is difficult to force it to do so in this version of the program. One set of parameters that accomplishes this is:

```
*MDP> whenToPark 10 $ policyIterate (parkingProblem 10 0.9 (-1) 1 0.1 (-100) 20) 1.0
[A 1 False,A 2 False,A 3 False,A 4 False,A 5 False,A 6 False,A 7 False,A 8 False,A 9 False,
A 10 False,B 1 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,B 10 False]
```

7 Acknowledgments

Thank you for a most stimulating exercise.