Reinforcement Learning: An Algorithmic Approach to Optimal Planning

Shane M. Conway
`shane.conway@gmail.com`

June 10, 2012

Abstract

How can we effectively plan for long-term goals? Many optimization problems require multi-step problems, where classic approaches to optimization fall short. I give a short introduction to Reinforcement Learning, including both discrete and continuous time, as well as finite and infinite domain problems. I try to demonstrate the functioning of RL through examples.

Contents

1 Preface: How not to be greedy and achieve your goals **3**

2 Introduction **5**
- 2.1 Optimal Control Theory . . . 5
- 2.2 Artificial Intelligence . . . 5
  - 2.2.1 Learn by Playing: Samuel's Checkers (1959) . . . 5
- 2.3 Overview . . . 6

3 Reinforcement Learning **7**
- 3.1 Intuition . . . 7
- 3.2 Formal definition . . . 8
- 3.3 Markov Decision Process . . . 9

4 Dynamic Programming, or Intelligent Brute Force Guessing **12**
- 4.1 Computational Design Pattern: Dependent Divide and Conquer . . . 12
  - 4.1.1 Example: Fibonacci Series . . . 13
- 4.2 Mathematical Formalism: Bellman's Principle of Optimality . . . 14
  - 4.2.1 Example: 0/1 Knapsack Problem . . . 15
- 4.3 Value Iteration . . . 17
  - 4.3.1 Example: Grid World . . . 17
- 4.4 Policy Iteration . . . 17

5 Temporal Difference Methods **19**
- 5.1 Q-Learning . . . 19
- 5.2 SARSA . . . 19

6 Infinite Domains **20**
- 6.1 The Curse of Dimensionality . . . 20

7 Continuous Time **21**

8 Applications **22**
- 8.1 Example: Shortest Path . . . 22
- 8.2 Example: Soduku . . . 22
- 8.3 Example: TD-Gammon . . . 23
- 8.4 Example: Multi-Period Portfolio Optimization . . . 23

9 Planning and Modelling **24**

10 Exploration vs. Exploitation **25**
10.1 The Multi-Armed Bandit Problem 25

11 Summary **26**
11.1 Resources 26
11.2 Further Reading 26
11.3 Appendix 27

Chapter 1

Preface: How not to be greedy and achieve your goals

In just a few centuries, the people of Easter Island wiped out their forest, drove their plants and animals to extinction, and saw their complex society spiral into chaos and cannibalism.

Easter Island's isolated head statues have often posed a mystery to scientists. How could a population creative enough to carve these works of art have completely disappeared?

The Polynesian population of Easter Island can thus be viewed as having greedily consumed short-term rewards without properly considering the long-term consequences.

http://www.hartford-hwp.com/archives/24/042.html

Reinforcement learning

Given the attraction of short-term rewards, how can we instead learn to trade-off appropriately and achieve a long term goal?

Samual's checker playing program. http://webdocs.cs.ualberta.ca/ sutton/papers/sutton-88.pdf

How do you become an expert at something? In Malcolm Gladwell's "Outliers", we are told that a few things are crucial: luck, practice, and ... He placed a great emphasis on the "10,000-hour rule".

Suppose that you want to learn to write, or to play the piano. We can picture the domineering instructor dolling our rewards and punishments in response to our performance.

Most introductory overviews of machine learning focus on a divide between *supervised* and *unsupervised* learning. These methods are typically mathematical and abstract. The most common method of learning from data is linear regression through ordinary least squares.

Supervised learning covers problems where the target variable is known in the training data. This covers models ranging from linear regression to neural networks. This makes identifying the underlying function a simple matter of trying to fit the target to the features, and given enough representative data and a proper understanding of the state space, we should be able to fully determine the function.

Unsupervised learning covers the case when we do not know the target variable, and want to learn the structure of the data. The most popular model here is k-means clustering.

One way to think about how reinforcement learning differs from supervised learning is that it tends to do a better job on tasks that evolve over time. It chooses an action based on the current state, with some idea of what worked before, rather than simply looking at all the data and trying to find the "best fit".

Reinforcement Learning is ...

> This was simply the idea of a learning system that wants something, that adapts its behavior in order to maximize a special signal from its environment. This was the idea of a "hedonistic" learning system, or, as we would say now, the idea of reinforcement learning.

(Reinforcement Learning, Preface, http://webdocs.cs.ualberta.ca/ sutton/book/ebook/node2.html)

Reinforcement Learning is an approach to learning that attempts to maximize a cumulative reward based on a set of actions and states. The techniques are very popular within operations research and control theory. It does not fall under the traditional paradigms of supervised because correct outputs are never provided. But it provdes more structure to the problem than unsupervised learning. We are not simply letting the data speak; we have a specific objective in mind.

There are several different methods for optimizing a reinforcement learning problem; I will be focusing on Dynamic Programming. Dynamic programming (or DP) is a powerful optimization technique that consists of breaking a problem down into smaller sub-problems, where the sub-problems are <em>not independent</em>. This is useful both in mathematics (especially fields like economics and operations research) and computer science (where the algorithm's complexity can be substantially improved through methods such as memoization). In this context, "programming" actually means "planning", as dynamic programming consists of forming a table of solutions.

Reinforcement learning may at first seem like an overly structured approach to learning from data. It requires you to explicitly define a state, reward, action, value, and policy. Wouldn't it just be easier to "let the data speak", as we do with something like k-means clustering?

Any model imposes a structure, whether that's clear or not.

Furthermore, different models are appropriate for different problems. And many of the major successes in recent projects have employed a combination of models.

Chapter 2

Introduction

Before diving into the details, I always like to put things in context. It is often easier to understand something when you can compare it with things that you already know. Human reason is a tool for classification.

Reinforcement Learning was essentially derived independently from two different fields: optimal control theory and artificial intelligence.

Compared to other optimization methods:

- Reinforcement learning is typically applied to *multi-period problems*. There should be some sequence of events that require decision making, and the goal is to find the optimal solution across the full period. By comparison, methods such convex optimization are limited to *single-period problems*: where there are not repeated decisions, but one decision that seeks the best solution.
- Reinforcement learning assumes an agent that can make decisions.

2.1 Optimal Control Theory

Optimal Control Theory

Control theory also introduces the idea of *approximate dynamic programming* to deal with the confining aspects of dynamic programming.

2.2 Artificial Intelligence

The two areas of artificial intelligence that became especially popular for reinforcement learning was game playing and robotics.

2.2.1 Learn by Playing: Samuel's Checkers (1959)

Claude Shannon

The canonical example of learning to play games is Arthur Samuel's checker's algorithm from ...

http://www.cs.unm.edu/ terran/downloads/classes/cs529-s11/papers/samuel$_1$959$_B.pdf$

http://ccar.colorado.edu/ leben/pdf/thesis$_powell.pdf$

Later we will look at one of the most successful game-playing programs, TD-Gammon (Tesauro, 1995).

2.3 Overview

I will cover reinforcement learning systematically, starting with the simplest discrete-space model using Dynamic Programming and moving towards continuous-space and high-dimensional problems.

Section 2 provides a genearl overview to the reinforcement learning problem. Section 3 introduction the first solution method, *Dynamic Programming*, a method used across many different fields. Section 4 builds on this by adding true RL models which don't require major assumptions.

Why another book on Reinforcement Learning? Sutton and Barto have already provided an incredibly good text on the subject. I won't begin to try to

The simple truth is that I fell in love with reinforcement learning. It is the most elegant and powerful method for learning that I have econountered, especially when combined with other existing models.

I hope that this book can provide some of the excitement that I have felt while learning and using this incredible method over the past few years.

Chapter 3

Reinforcement Learning

Survey paper: http://arxiv.org/pdf/cs/9605103.pdf

http://cs229.stanford.edu/notes/cs229-notes12.pdf http://www.tu-chemnitz.de/informatik/KI/scripts/ws09
//www.inf.ed.ac.uk/teaching/courses/rl/slides/4rllect10.pdfhttp : //www.cs.colostate.edu/ anderson/cs440/inde
*notes : reinflearn*1

Presentations: http://gandalf.psych.umn.edu/users/schrater/schrater$_l$*ab/courses/AI*2*/mdp*1*.pdfhttp* :
//gandalf.psych.umn.edu/users/schrater/schrater$_l$*ab/courses/AI*2*/rl*1*.pdf*

3.1 Intuition

Research into computational learning and intelligence is often inspired by natural processes. Neural networks were created to reproduce the inner functioning of the brain. Reinforcement learning reproduces the learning process of animals. It is closely related to evolutionary learning, although we will contrast the two methods to understand the differences.

The idea behind reinforcement learning can be found in many fields. It is a search algorithm: we want to search over an environment (or state space) to maximize our expected reward.

This also follows along the concept of *trial-and-error learning* from psychological learning literature, also known as the *Law of Effect* (as in Thorndike in 1911) (Marsland 293).

A prominant example of this is Operant Conditioning, as coined by B. F. Skinner in 1937. In Operant Conditioning

horndike was able to create a theory of learning based on his research with animals.[9] His doctoral dissertation, "Animal Intelligence: An Experimental Study of the Associative Processes in Animals", was the first in psychology where the subjects were nonhumans.[9] Thorndike was interested in whether animals could learn tasks through imitation or observation.[10] To test this, Thorndike created puzzle boxes. The puzzle boxes were approximately 20 inches long, 15 inches wide, and 12 inches tall.[11] Each box had a door that was pulled open by a weight attached to a string that ran over a pulley and was attached to the door.[11] The string attached to the door led to a lever or button inside the box.[11] When the animal pressed the bar or pulled the lever, the string attached to the door would cause the weight to lift and the door to open.[11] Thorndike's puzzle boxes were arranged so that the animal would be required to perform a certain response (pulling a lever or pushing a button), while he measured the amount of time it took them to escape.[9] Once the animal had performed the desired response they were allowed to escape and were also given a reward, usually food.[9] Thorndike primarily used cats in his puzzle boxes.

Consider the following simple example of Skinner's rat. The rat is always in the same state, but can choose between two actions: lever 1 and lever 2. Lever 1 provides a treat as a reward, while lever 2 provides an electric shock as a punishment.

```
> state <- 0
> action <- rnorm(1000)
> reward <- 0
```

3.2 Formal definition

> RL algorithms are methods for solving this kind of problem, that is, problems involving sequences of decisions in which each decision affects what opportunities are available later, in which the effects need not be deterministic, and in which there are long-term goals. RL methods are intended to address the kind of learning and decision making problems that people and animals face in their normal, everyday lives.

The basic structure of reinformcement learning can be seen as a cycle, as in figure 3.1. In general, reinforcement learning is an iterative process where an *agent* takes *actions* in an *environment* an receives a *reward*.

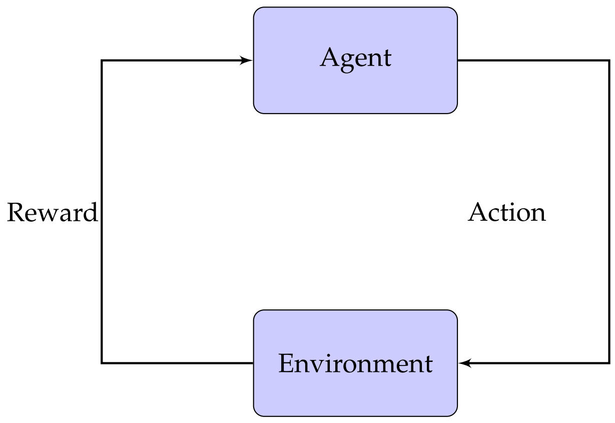


Figure 3.1: The reinforcement learning cycle, from agent to environment, back to agent.

Based on the representation in figure 3.1, we can define the basic building blocks of a reinforcement learning problem as:

- S is a finite set of states
- A is finite set of actions
- R is a reward function

IN general, we want to maximize the expected return, which can be defined as:

$$R(s) = r(s_0) + r_{t-2} + r_{t-3} + \cdots + r_{t-N} \tag{3.1}$$

We want to penalize ...

$$R_t = r_{t-1} + \gamma r_{t-2} + \gamma^2 r_{t-3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t-k-1} \tag{3.2}$$

Define a policy π
We define a *value function* to maximize the expected return:

$$V^{\pi}(s) = E[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots | s_0 = s, \pi] \tag{3.3}$$

We can rewrite this as a recurrence relation, which is known as the *Bellman Equation*:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s') V^{\pi}(s') \tag{3.4}$$

We can use *backup diagrams* to describe the RL process.
[Described on p.71 of B/S]

3.3 Markov Decision Process

A Markov Decision Process (MDP) is one of the most common methods for reinfocement learning. Beyond what we have already described in the prior

> Most RL research is conducted within the mathematical framework of Markov decision processes (MDPs). MDPs involve a decision-making agent interacting with its environment so as to maximize the cumulative reward it receives over time. The agent perceives aspects of the environment's state and selects actions. The agent may estimate a value function and use it to construct better and better decision-making policies over time.

http://webdocs.cs.ualberta.ca/ sutton/RL-FAQ.html
The MDP is a simple case as it adheres to the *markov property*, that "the future is independent of the past except through the present". That is to say that predicting the future is only dependent on the present state, or more formally:

$$P[s_{t+1}|s_t] = P[s_{t+1}|1, ..., s_t] \tag{3.5}$$

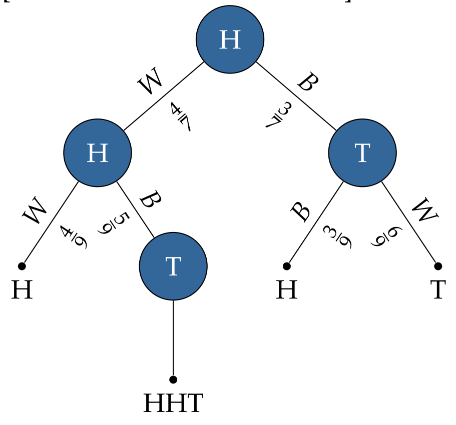
Many people will be familiar with the idea of a Markov process from stochastic processes. The discrete Markov process (MP) consists of a tuple (S, P):

- S is a finite set of states.
- P is a state transition probability matrix.

A common example from finance and other fields is the idea of a *random walk* (first introduced to finance by Louis Bachelier in "The Theory of Speculation" in 1900).
If we consider

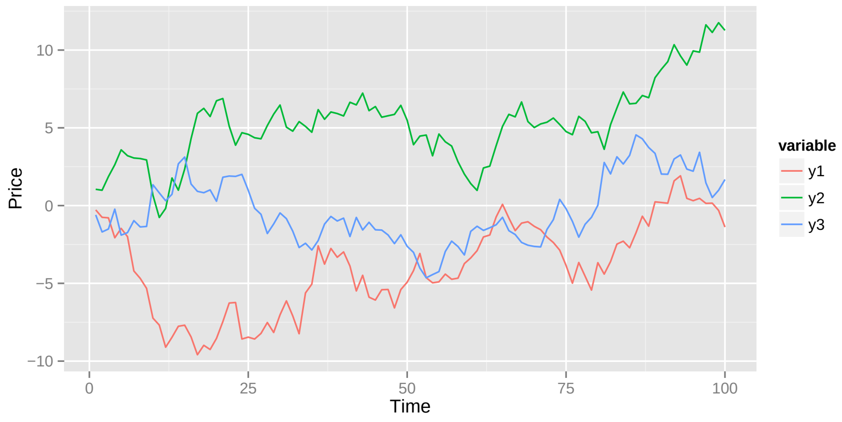
$$\begin{tikzcd}
0 \arrow[loop above, "p_{00}"] & 1 \arrow[l, "p_{10}"'] \arrow[loop above, "p_{11}"] & 2 \arrow[l, "p_{21}"'] \arrow[loop above, "p_{22}"] & {} \arrow[l, "p_{32}"'] & g \arrow[l, "p_{gg-1}"'] \arrow[loop above, "p_{gg}"]
\end{tikzcd}$$

[Show random walk as a tree]



Now we can build this chain out further and view how it evolves over time. Imagine that every coin flip relates to a price moving up or down one tick.

```
> y1 <- cumsum(rnorm(100))
> y2 <- cumsum(rnorm(100))
> y3 <- cumsum(rnorm(100))
> random.walk <- data.frame(time = 1:100, y1 = y1,
+     y2 = y2, y3 = y3)
> random.walk <- melt(random.walk, "time")
> print(ggplot(random.walk, aes(x = time, y = value,
+     group = variable, color = variable)) + geom_line() +
+     xlab("Time") + ylab("Price"))
```



The discrete MDP extends the Markov process further by including the idea of a utility or value that should be maximized. This consists of a tuple (S, A, P_{sa}, R, γ).

- S is a finite set of states
- A is finite set of actions
- P is a state transition probability matrix
- R is a reward function

- γ is a discount factor

Once we have a way of characterizing the *environment* E, we then need to try to find the optimal behavior:

- $\pi^*(s)$ is the optimal *policy*, which defines what steps to take to go from state s to some future state (such as a terminal state) in order ot maximize the rewards.
- $V^*(s)$ is the expected value for following the optimal policy π^* starting in the state s.
- $Q^*(a, s)$ is the optimal q-function which defines what happens given an action a in the state s.

Keep in mind that we are considering the *expected value* ($E[V]$) for the given policy: this is another way of saying the average value.

http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching$_{f}iles/MDP.pdf$

How can we solve an finite, discrete MDP?

Chapter 4

Dynamic Programming, or Intelligent Brute Force Guessing

History of dynamic programming:

http://www.eng.tau.ac.il/ ami/cd/or50/1526-5463-2002-50-01-0048.pdf

"The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies." (B/S p.89)

Now that we have a conceptual understanding of how reinforcement learning works, we can take a look at the most common method: *dynamic programming* (DP).

There are two standard methods: top down and bottom up.

Dynamic programming can be summarized as:

- Recursion
- Memoization
- Brute force

4.1 Computational Design Pattern: Dependent Divide and Conquer

A quick search of CRAN (including the Optimization view) will reveal that there are no dynamic programming packages available. This is because dynamic programming is a *design technique* rather than a specific algorithm; it is specifically tailored to each given problem.

From a design perspective, dynamic programming has a lot in common with divide-and-conquer, but is specifically designed to address problems where the subproblems are not independent. A nice summary of this distinction can be found in "Introduction to Algorithms":

> Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems...divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when

subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

4.1.1 Example: Fibonacci Series

The Fibonacci Sequence is one of the most popular introductory examples used in teaching programming. This entails a function where the output is simply the sum of the preceding two values:

$$F_n = F_{n-1} + F_{n-2} \tag{4.1}$$

Which results in a sequence that looks like:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... \tag{4.2}$$

A classic approach to solve this problem is to use recursion:

```
> fib1 <- function(n) {
+     if (n == 0)
+         return(0)
+     else if (n == 1)
+         return(1)
+     else {
+         return(fib1(n - 1) + fib1(n - 2))
+     }
+ }
```

I have included a print statement at the top of the function so that it's easy to see how this function is called. Unfortunately, the computational complexity of this algorithm is exponential. As an example, for <code>fib(5)</code>, we end up calling <code>fib(2)</code> three times.

The dynamic programming solution calls each value *once* and stores the values (memoization, also see <a href="http://cran.r-project.org/web/packages/memoise/index.html">Hadley Wickham's memoise package for R</a>). This might seem overly simplistic when you consider the recursive algorithm, but in this case we need to loop and maintain state, which is a very imperative approach.

```
> fib2 <- function(n) {
+     fibs <- c(0, 1)
+     if (n > 1) {
+         for (i in 3:n) {
+             fibs[i] = fibs[i - 1] + fibs[i - 2]
+         }
+     }
+     return(fibs[n])
+ }
```

This second algorithm is linear time complexity, and its performance is noticeably better even on small values for n.

```
> library(rbenchmark)
> benchmark(fib1(10))

      test replications elapsed relative user.self sys.self
1 fib1(10)          100    0.04        1     0.041        0
  user.child sys.child
1          0         0

> benchmark(fib2(10))

      test replications elapsed relative user.self sys.self
1 fib2(10)          100   0.006        1     0.005        0
  user.child sys.child
1          0         0
```

This demonstrates both the performance improvement that is possible, and also the dependence structure that is common in dynamic programming applications: the state of each subsequent value depends on prior states.

This also demonstrates another important characteristic of dynamic programming: <i>the trade-off between space and time</i>.

4.2 Mathematical Formalism: Bellman's Principle of Optimality

As ... explain:

> Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

Dynamic programming may be necessary when a static optimization isn't possible. This regularly occurs in economics and finance; an early example of its application can be found in <a href="http://pages.stern.nyu.edu/ eelton/papers/71-may.pdf">Elton and Gruber (1971) "Dynamic Programming Applications in Finance"</a>. While DP is more general than a static approach, it can also suffer from the curse of dimensionality.

http://ocw.mit.edu/courses/sloan-school-of-management/15-450-analytics-of-finance-fall-2010/lecture-notes/MIT15$_4$50*F*10$_l$*ec*05.*pdf*

The key to understanding how dynamic programming can lead to optimal solutions in markov decision process with dependent steps is the concept of <a href="http://en.wikipedia.org/wiki/Bellman$_{equ}$
strong > BellmanOptimality < /strong >< /a > .Thisisnamedafter < ahref = "http :
//en.wikipedia.org/wiki/Richard$_B$*ellman" > RichardBellman < /a >, whoinventeddynamicprogrammingandwrotea*
ahref = "http : //www.ingre.unimore.it/or/corsi/vecchi$_c$*orsi/complementiro/materialedidattico/originidp.pdf" >*
"RichardBellmanontheBirthofDynamicProgramming" < /a > formoredetail).

> Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. (from Bellman, 1957, Chap. III.3.)

$$V(x) = \max_{a \in \Gamma(x)} \{F(x,a) + \beta V(T(x,a))\}. \tag{4.3}$$

4.2.1 Example: 0/1 Knapsack Problem

How can we use this for optimization? The <a href="http://en.wikipedia.org/wiki/Knapsack$_{p}roblem'' >$ $knapsackproblem < /a > isaverywellknownexampleofdynamicprogrammingthataddressesthisquestion.Thisisaresourc$ $athiefentersahouseandwantstostealasmanyvaluableitemsaspossiblebutcanonlycarrysomuchweight.$

<img alt="" src="http://imgs.xkcd.com/comics/np$_{c}omplete.png''class = ''aligncenter''/ >$

Presented with this problem, we can think of several different approaches. We might start with a <i>greedy</i> strategy: rank all the items by value, and add them to the knapsack until we can't fit any more. This won't find the optimal solution, because there will likely be a combination of items each of which is less valuable, but whose combined value will be greater. Alternatively, we might simply iterate through every single combination of items, but while this finds the optimal solution, it also grows with exponential complexity as we have more items in the set. So we can use Dynamic Programming:

Define a set S of [latex]n[/latex] items each with a value and weight [latex]v$_{i}, w_{i}[/latex]respectively.Wewanttosolv$

$$maximize \sum_{i=0}^{n} v_i \tag{4.4}$$

Subject to:

$$\sum_{i=0}^{n} w_i \leq W \tag{4.5}$$

The dynamic programming solution follows these steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the optimal solution to subproblems.
3. Step backwards and find the global optimal value.

```
> knapsack <- function(w, v, W) {
+     A <- matrix(rep(0, (W + 1) * (length(w) +
+         1)), ncol = W + 1)
+     for (j in 1:length(w)) {
+         for (Y in 1:W) {
+             if (w[j] > Y)
+                 A[j + 1, Y + 1] = A[j, Y + 1]
+             else A[j + 1, Y + 1] = max(A[j, Y +
+                 1], v[j] + A[j, Y - w[j] + 1])
+         }
+     }
+     return(A)
+ }
> optimal.weights <- function(w, W, A) {
+     amount = rep(0, length(w))
+     a = A[nrow(A), ncol(A)]
+     j = length(w)
+     Y = W
+     while (a > 0) {
+         while (A[j + 1, Y + 1] == a) {
+             j = j - 1
+         }
+         j = j + 1
+         amount[j] = 1
+         Y = Y - w[j]
+         j = j - 1
+         a = A[j + 1, Y + 1]
+     }
+     return(amount)
+ }
```

We can test this on a simple example of 7 items with different weights an values:

```
> w = c(1, 1, 1, 1, 2, 2, 3)
> v = c(1, 1, 2, 3, 1, 3, 5)
> W = 7
> A <- knapsack(w, v, W)
> best.value <- A[nrow(A), ncol(A)]
> weights <- optimal.weights(w, W, A)
```

We can look at the matrix A to see how dynamic program stores values in a table rather than recomputing the values repeatedly:

```
> A

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    0    0    0    0    0    0    0    0
[2,]    0    1    1    1    1    1    1    1
[3,]    0    1    2    2    2    2    2    2
[4,]    0    2    3    4    4    4    4    4
[5,]    0    3    5    6    7    7    7    7
[6,]    0    3    5    6    7    7    8    8
[7,]    0    3    5    6    8    9   10   10
[8,]    0    3    5    6    8   10   11   13
```

Where the best total value when satisfying the weight constraint in the example.

There are many examples of <a href="http://rosettacode.org/wiki/Knapsack$_p$*roblem*/0 − 1" > *solutionstotheknapsackproblemonRosettaCode* < /*a* > .*Ishouldalsopointoutthat* < *ahref* = "*https* : //*sites.google.com*/*site*/*mikescoderama*/*Home*/0 − 1 − *knapsack* − *problem* − *in* − *p*" > "*Mike*′*sCoderama*"*hasapostcoveringthisinPython* < /*a* > .

Now that we have built up an understanding of how dynamic programming can solve sequential problems by breaking the problem into smaller parts, we can look at the first solution for a discrete MDP.

"We can easily obtain optimal policies once we have found the optimal value functions, V* or Q*, which solve the Bellman optimality equations:

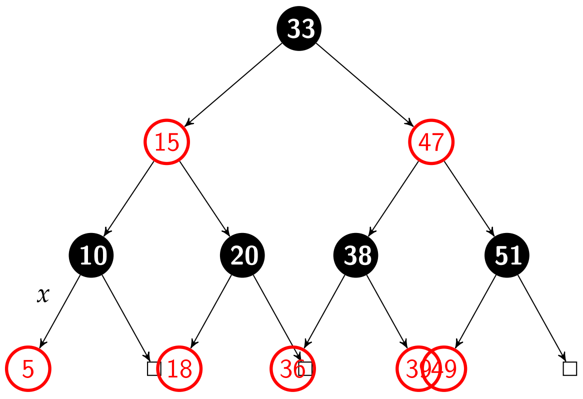
4.3 Value Iteration

Value iteration

Pseudocode from http://arxiv.org/pdf/cs/9605103.pdf

```
initialize $V(s)$ arbitrarily
loop until policy good enough
  loop for s ∈ S
    loop for a ∈ A
      $Q(s;a) := R(s;a) + \gamma \sum_{s' \in S} T(s;a;s')V(s')$
    $V(s) := max_a Q(s;a)$
  end loop
end loop
```

Here we can think about how the value function V^{π} relates to the q-function $Q^{\pi}(s,a)$?



4.3.1 Example: Grid World

"Grid World" is a canonical example in reinforcement learning. Consider a robot that is trying to navigate a maze. This example is from Sutton and Barto, although similar examples can be found in "Artificial Intelligence", Tom Mitchel, ...

Suppose that a robot can move north, south, east, or west with equal probablity initially. How can it find the optimal policy to the safe zone.

4.4 Policy Iteration

Policy iteration is a method for finding an optimal policy by iteratively computing the values for a policy through *policy evaluation* and then improving the policy through *policy improvement*.

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \tag{4.6}$$

Pseudo code (from Sutton and Barto, 92):
http://www.ics.uci.edu/ csp/r42a-mdp$_{r}eport.pdf$
1. Initialization

$$V(s) \in \mathbb{R} \tag{4.7}$$

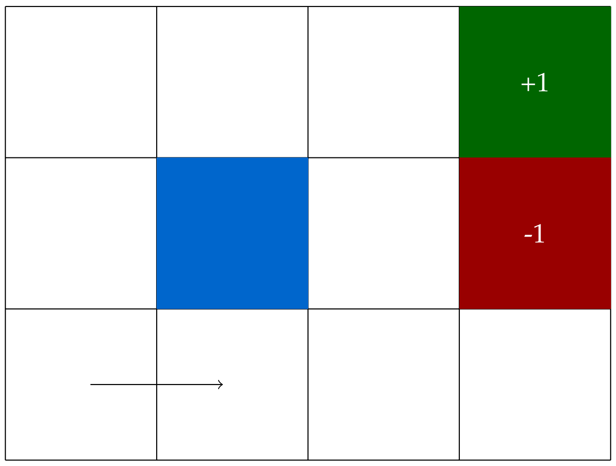


Figure 4.1: Grid World example.

```
choose an arbitrary policy $\pi'$
loop
  $\pi := \pi'$
  compute the value function of policy $\pi$:
    solve the linear equations
      $V_\pi(s) := R(s;\pi(s)) + \gamma \sum_{s' \in S} T(s;\pi(s);s') V_\pi(s')$
  improve the policy at each state:
      $\pi'(s)$
until $\pi = \pi'$
```

Chapter 5

Temporal Difference Methods

http://homes.cs.washington.edu/ todorov/courses/amath579/VanRoy$_n$$otes.pdf$

I will focus on *temporal difference (TD)* methods.

http://webdocs.cs.ualberta.ca/ sutton/papers/sutton-88.pdf

We can consider the current state, and average across all of the actions that can be taken, leaving the policy to sort this out for itself (the state-value function, V (s)), or we can consider the current state and each possible action that can be taken separately, the action-value function, Q(s, a).

Marsland, Stephen (2011-07-16). Machine Learning: An Algorithmic Perspective (Page 305). Chapman and Hall/CRC. Kindle Edition.

The Q(s, a) version looks very similar, except that we have to include the action information. In both cases we are using the difference between the current and previous estimates, which is why these methods have the name of temporal difference (TD) methods.

Marsland, Stephen (2011-07-16). Machine Learning: An Algorithmic Perspective (Page 306). Chapman and Hall/CRC. Kindle Edition.

5.1 Q-Learning

5.2 SARSA

Chapter 6

Infinite Domains

6.1 The Curse of Dimensionality

Given that dynamic programming is a brute force algorithm, it is fundamentally prone to suffer from the *curse of dimensionality*.

Chapter 7

Continuous Time

Thus far we have only considered discrete time processes.

Chapter 8

Applications

8.1 Example: Shortest Path

The Traveling Salesman Problem (TSP) is one of the most famous problems in theoretical computer science.

$$\begin{tikzcd}
 & & D \arrow[dl, "10"] \arrow[dd, leftrightarrow, "4"] \arrow[dddr, leftrightarrow, bend left, "10"] & \\
 & B \arrow[dl, "10"] \arrow[dd, leftrightarrow, "5"] & & \\
P \arrow[dr, leftrightarrow, "4"] & & C \arrow[ul, "3"] \arrow[dl, leftrightarrow, "9"] & \\
 & M & & L \arrow[ll, "10"]
\end{tikzcd}$$

8.2 Example: Soduku

Learning to play a game is one of the most straight-forward ways to think about RL.

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Unsolved Sudoku

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Solved Sudoku

8.3 Example: TD-Gammon

TD-Gammon remains one of the most famous examples of reinforcement learning.

8.4 Example: Multi-Period Portfolio Optimization

Consider the classic mean-variance portfolio optimization as

Chapter 9

Planning and Modelling

Model vs. Table

Convergence requires an infinite number of visits to each state, so an alternative approximation is to replace the table with a model, such as a neural network.

http://www.bcs.rochester.edu/people/robbie/jacobslab/cheat$_{s}heet/ModelBasedRL.pdf$

Chapter 10

Exploration vs. Exploitation

10.1 The Multi-Armed Bandit Problem

Chapter 11

Summary

Dynamic programming is a powerful technique both for finding optimal solutions to problems where steps have dependence, and at improving algorithm performance by trading off space for time. I will finish off this topic in the next post by looking in more detail at the topic of Bellman's Optimality Principle and at an example from economics.

11.1 Resources

I rely primarily on three sources for this analysis, all of which I highly recommend:

- Mario Miranda and Paul Fackler <a href="http://www.amazon.com/gp/product/0262134209/ref=as$_l i_s s_t l$?ie UTF8camp = 1789creative = 390957creativeASIN = 0262134209linkCode = as2tag = statalgo − 20" > "AppliedComputationalEconomicsandFinance" < /a > .Givesaconciseoverviewofcomputation ahref = "http : //www.amazon.com/gp/product/0262033844/ref = as$_l i_s s_t l$?ie = UTF8camp = 1789creative = 390957creativeASIN = 0262033844linkCode = as2tag = statalgo − 20" > "IntroductiontoAlgorithms" < /a > .Thisisaclassictextoncomputeralgorithms.

- Richard Sutton and Andrew Barto<a href="http://www.amazon.com/gp/product/0262193981/ref=as$_l i_s s_t l$?ie = UTF8camp = 1789creative = 390957creativeASIN = 0262193981linkCode = as2tag = statalgo − 20" > "ReinforcementLearning : AnIntroduction" < /a > (< ahref = "http : //webdocs.cs.ualberta.ca/ sutton/boo book.html" > htmlversionavailableforfree < /a >)Themostpopularintroductorytextonthesubject, lackinginsomeofthe
  In addition, I would point out:

- <a href="http://www.amazon.com/gp/product/0691152705/ref=as$_l i_s s_t l$?ie = UTF8camp = 1789creative = 390957creativeASIN = 0691152705linkCode = as2tag = actusfideicom" > "InPursuitoftheTravelingSalesman : MathematicsattheLimitsofComputation" < /a > isanexcellentrecentchronicleo see < ahref = "http : //www.amazon.com/gp/product/0470171553/ref = as$_l i_s s_t l$?ie = UTF8camp = 1789creative = 390957creativeASIN = 0470171553linkCode = as2tag = statalgo − 20" > "ApproximateDynamicProgramming : SolvingtheCursesofDimensionality" < /a > fortreatmentofthismethod.

11.2 Further Reading

http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/rl-survey.html
http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a.pdf

There are several good books on the subject:

- R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction* MIT Press, Cambridge, MA, USA, 1998

Reinforcement learning is also covered in chapters of:

- Chapter 13 of M. Marsland "Machine Learning: An Algorithmic Perspective" ...
- Chapter 13 of T. Mitchell *"Machine Learning"* ...

11.3 Appendix

Some useful resources:

- http://www.saylor.org/site/wp-content/uploads/2011/06/Dynamic-Programming.pdf
- https://researchspace.auckland.ac.nz/bitstream/handle/2292/190/230.pdf
- https://stacks.stanford.edu/file/druid:zd335yg6884/YongyangCai$_t hesis - augmented.pdf http : //homes.cs.washington.edu/ todorov/papers/optimality_c hapter.pdf$

- http://www.gatsby.ucl.ac.uk/ dayan/papers/dw01.pdf
- http://www.cs.ubc.ca/ nando/550-2006/lectures/l3.pdf (and l2.pdf, etc.)
- http://www.nbu.bg/cogs/events/2000/Readings/Petrov/rltutorial.pdf
- http://www.cs.ubc.ca/ nando/550-2006/lectures/l3.pdf
- http://www.cs.tcd.ie/Saturnino.Luz/t/cs7032/solvemdps-notes.pdf
- http://homes.cs.washington.edu/ todorov/papers/optimality$_c hapter.pdf http : //neuro.bstu.by/ai/RL - 3.pdf$
- http://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/Dynamic/knapsack-dyn.htm
- http://artint.info/html/ArtInt$_2 62.html TomMitchell : http : //www.cs.cmu.edu/ epxing/Class/10701 - 10s/Lecture/MDPs_R L_0 4_2 6_2 010.pdf$

  http://courses.cs.washington.edu/courses/cse473/11au/ https://github.com/sinairv/Temporal-Difference-Learning

Bibliography

[Barto Bradtke Singh, 1995] Andrew G. Barto, Steven J. Bradtke, Satinder P. Singh Learning to act using real-time dynamic programming *Artificial Intelligence*, Volume 72, Issues 1-2, January 1995, Pages 81-138