Tutorial

Approximate Dynamic Programming

Biarri Applied Mathematics Conference, 12-13 November 2012

COMMERCIAL MATHEMATICS

www.biarri.com

Approximate dynamic programming will change your life!

- Quick dynamic programming refresher
- Stochastic dynamic programming
- The “curse of dimensionality”
- A general ADP algorithm
- Worked examples
- Future directions

- Key reference:

  Approximate Dynamic Programming: Solving the Curses of Dimensionality

  Second Edition, 2011. Warren B. Powell, Wiley Series in Probability and Statistics

A simple example

- Inventory ordering problem
- Data:
  - Starting stock level
  - Maximum stock level
  - Demand in each time period (deterministic)
  - Total cost of ordering n units of stock = *F(n)*, for some given function F
    - Typically non-linear, increasing
  - Cost per unit per time period of holding stock
  - Stock out cost
- Determine:
  - Quantity of stock to order in each period

Let's get our notation straight

S Discrete state space

A Discrete action space

Stages indexed by t : This often corresponds to time periods

Transition function: $S_{t+1} = S^M(S_t, a_t)$

The state we are in at stage t+1 depends on the previous state and action
In our example, the new inventory level depends on the previous level, quantity ordered and the demand

Contribution function: $C_t(S_t, a_t)$

The immediate cost (or value) of making decision a_t in state S_t. *F* in our example.

Value function: $V_t(S_t) = \min_{a_t}\{C_t(S_t, a_t) + V_{t+1}(S_{t+1})\}$

Total cost (or value) of making all the best decisions from here to the end

Solving simple dynamic programming problems is easy

- Code recursive function directly
- “Memo-ise” values for efficiency
- Return value function and the argument that achieves the optimal value

Extending to stochastic problems is sometimes easy

- Our example:
  - Demand is no longer constant, but rather given by a known, discrete probability distribution
- Notation:
  - Transition matrix: $p_t(S_{t+1} | S_t, a_t)$
    Probability that if we are in state S_t and take action a_t that we will next be in state S_{t+1}
    For our example this can be calculated from the demand distribution and our action
  - Value function: $V_t(S_t) = \min_{a_t} \left\{ C_t(S_t, a_t) + \sum_{s' \in S} p_t(s' | S_t, a_t) V_{t+1}(s') \right\}$

    alternatively: $V_t(S_t) = \min_{a_t} \{ C_t(S_t, a_t) + \mathrm{E}[V_{t+1}(S_{t+1}) | S_t, a_t] \}$

Solving simple stochastic dynamic programming problems is easy

- Code recursive function directly
- “Memo-ise” values for efficiency
- Return value function and the argument that achieves the optimal value

- The answer is an expected value and a **policy**. For each possible state, it specifies the optimal action.

Some problems are too hard to solve exactly

- Consider our example expanded to 10 product types:
  - If we have 1000 maximum units in stock for each product, we have 1001^{10} possible states
  - If demand for each can range from 0 to 200, we have 201^{10} possible outcomes
  - If we can order between up to 500 units at a time, we have 501^{10} possible actions
- These are the three curses of dimensionality:
  - State space – traditional curse of dimensionality for deterministic problems
  - Outcome space – we may not be able to compute our expectation
  - Decision space – LP and MIP regularly handle very large decision spaces

There is a way forward

- Outline approach:
  - Make an initial estimate of $V_t(S_t)$ for states S_t
  - Repeatedly choose "sample paths" of random outcomes
    - At each time step, make the optimal decision based on the estimates of $V_t(S_t)$
    - Update the estimates based on the observed actual values

- We produce a set of "value function approximations" which collectively define a policy, as we can make a decision a_t so as to minimise:

$$\min_{a_t}\left\{C_t(S_t, a_t) + \mathrm{E}[\widehat{V}_{t+1}(S_{t+1}) | S_t, a_t]\right\}$$

- Balance (known) short term costs with (estimated) long term costs
  - Using just short term costs is known as the myopic policy
  - It is surprisingly popular!

This tricky idea is best understood by some concrete examples

- Assigning prime mover / driver pairs to jobs:
  - Cost of assignment of driver to each job known: Travel time, specific payments, etc
  - New jobs arriving at random (with known distribution)
- Myopic policy: assign drivers to jobs based on known assignment costs
- ADP policy: use value function approximations for the value of drivers becoming available given drivers home base, hours served, etc

- Assigning blood to demands: elective, non/elective, blood type substitution
  - Demand and supply for each time period are random variables
- Myopic policy: assigns all blood possible
- ADP policy: keeps some blood back based on value of starting with some blood

Outline algorithm

Initialise $\widehat{V}_t(S_t)$ and initial state S_0^1

For n = 1…N

Choose a sample path ω^n

For t = 0…T

Solve $\widehat{v}_t = \min_{a_t}\left\{C_t(S_t^n, a_t) + \mathrm{E}[\widehat{V}_{t+1}(S_{t+1}^n) \middle| S_t^n, a_t]\right\}$

Let the action that achieves this minimum be $\widehat{a}_t^n$

Update a value function approximation: $\widehat{V}_t(S_t^n) = \widehat{v}_t$

Compute $S_{t+1}^n = S^M(S_t^n, a_t^n, \omega^n)$

That's fine in theory, but will it work in practice

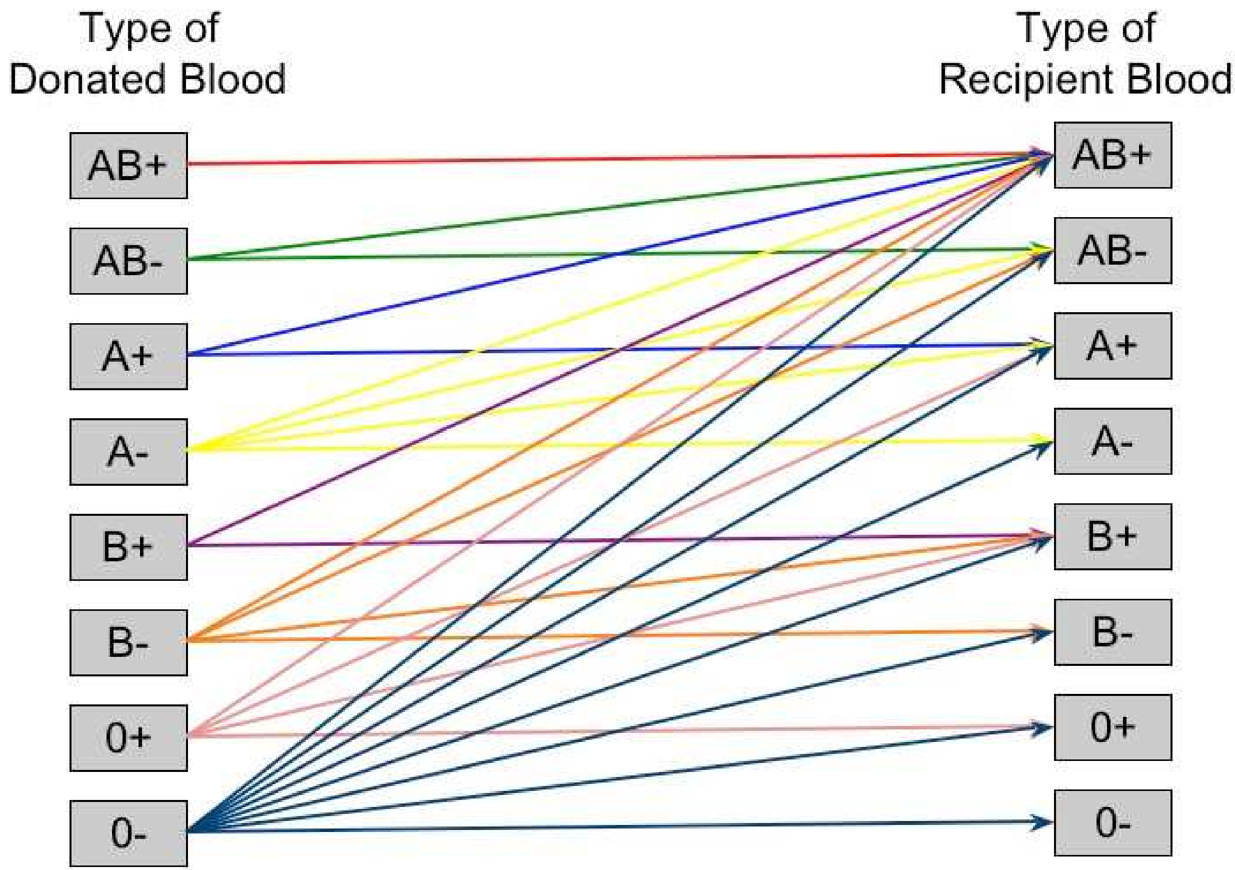
- The approach combines optimisation and simulation
- The expectation calculation may be still be intractable
- How do we choose initial values and an initial state?
- What about states we don't visit?
- Will it converge to a useful answer, and how long will that take?

- Some general tricks:
  - Separating the value function into linear components
  - Piecewise linearisation
  - Aggregation of the state space

And now for a detailed example

- Random weekly demand and supply of blood of different types
- Fixed number of weeks (or infinite horizon)
- Demand split into different types of procedures:
  - Urgent or Elective (85:15)
  - Blood substitution allowed or not (50:50)
- Known table of possible blood substitution
- Blood stored for up to 6 weeks
- Specified myopic value of using blood:
  - Filling urgent demand: 40
  - Filling elective demand: 20
  - Substituting blood: -10
  - Using O- blood as a substitute: 5

Blood substitution



Decision making

- Myopic decisions
  - Assign blood to demand so as to maximise the value at each stage
  - Simple network flow model
- The problem
  - Strongly favours using all blood
  - No incentive to keep some stock on hand to meet urgent demand in the next period
  - State, decision and outcome vectors all have large dimension

Decision making

- ADP approach: $\max_{x_t}\left\{C_t(S_t, x_t) + \mathrm{E}[\widehat{V}_{t+1}(S_{t+1}) | S_t, x_t]\right\}$
  - x_t represents the blood to demand assignments made at time t
  - C_t is the short term value of the assignments
  - The state S_t is the amount of each blood type in stock
- How do we estimate the value function?
  - Based the amount of each blood type/age combination left after we make a decision
  - Separated by blood type/age

$$\begin{aligned} \mathrm{E}[\widehat{V}_{t+1}(S_{t+1}) | S_t, x_t] &= \overline{V}_t(R_t^x) \\ &= \sum_b \overline{V}_{tb}(R_{tb}^x) \end{aligned}$$

  - Then use a piecewise linearisation – easily incorporated into network model
  - Remaining problem – how do we estimate the slopes of the piecewise linearisation
  - The answer – dual variables

The algorithm for calculating the value function approximations

Initialise piecewise linear approximations and blood levels

For n = 1 to N

- Set *level* to be a vector of blood starting levels (blood type / age)
- Simulate one week
  - Generate supply and demand
  - Calculate the assignment of blood to demand (use current value functions)
  - Roll forward unused blood
- Use dual variables to update the piecewise linear approximations around *level*

Updating the piecewise linearisation

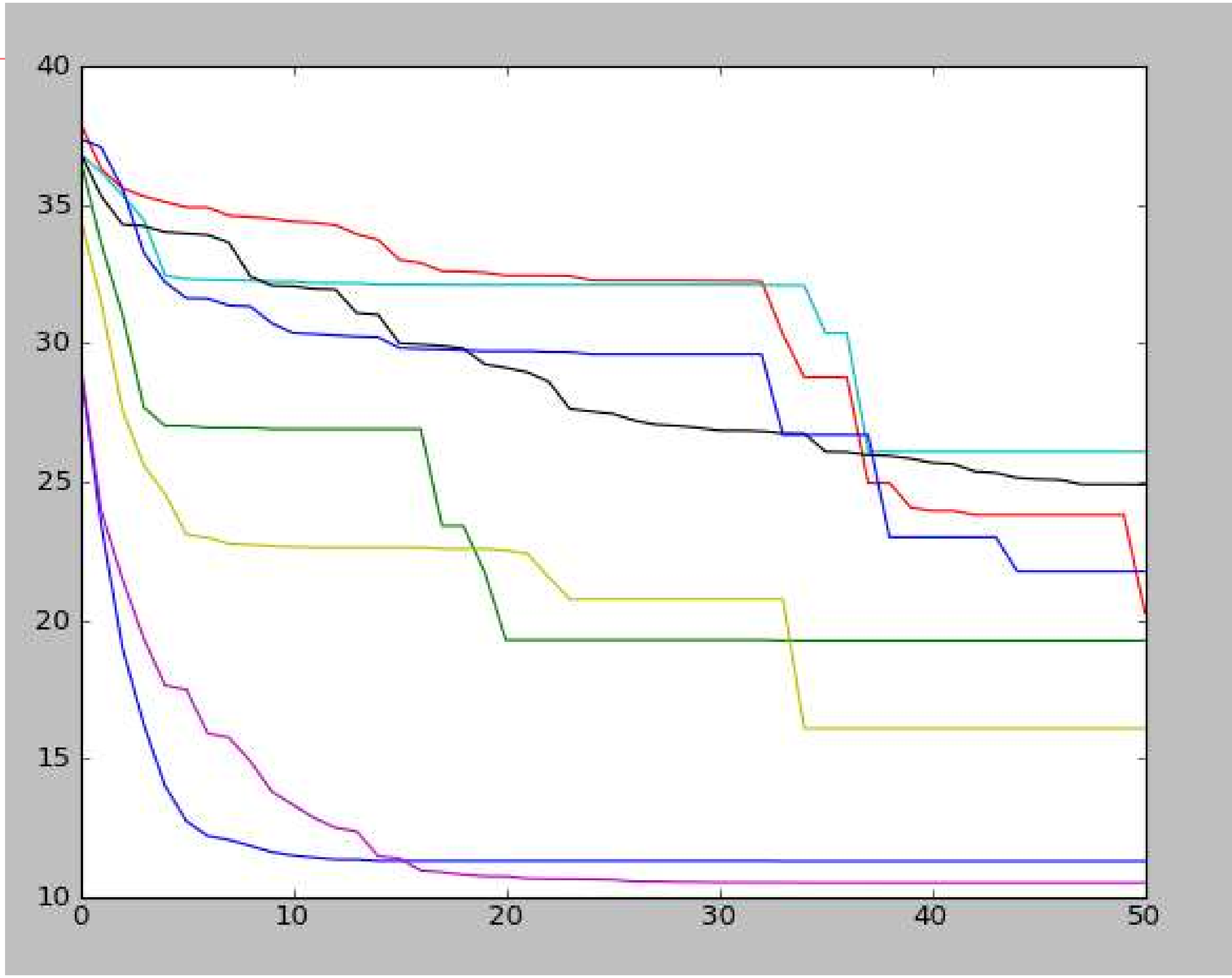
- Two key parameters
  - Width of window to update
    - A function of n
    - Wider in earlier iterations
  - Smoothing parameter
    - Weight on this iteration
    - Smaller over time
- Need to ensure values for each interval are monotonically decreasing

- Multiple value functions are updated at each step
  - This is common when using dual variables in resource allocation problems

Results

- 5000 time interval simulation
- Myopic:
  - Average objective: 5777.97
  - Average unmet demand: 48.91
  - Average unmet urgent demand: 30.30
- With value function approximations:
  - Average objective: 5981.34 (3.5% better)
  - Average unmet demand: 49.74
  - Average unmet urgent demand: 22.38

Value Function Approximations



ADP is still very much a black art

- Very problem specific approaches required
- Tuning of important parameters is itself a stochastic optimisation problem
- Very few proofs of convergence
  - It's fine in practice, but does it work in theory

- However…

- Very natural interpretation of the results
- Easy to demonstrate when it does better than myopic (with statistical significance)
- Relatively easy to implement, with similar data requirements as simulation

Questions?

Dr Michael Forbes
Optimisation Guru
michael.forbes@biarri.com