Job Search I: The McCall Search Model

Thomas J. Sargent and John Stachurski

March 15, 2020

1 Contents

- Overview 2
- The McCall Model 3
- Computing the Optimal Policy: Take 1 4
- Computing the Optimal Policy: Take 2 5
- Exercises 6
- Solutions 7

“Questioning a McCall worker is like having a conversation with an out-of-work friend: ‘Maybe you are setting your sights too high’, or ‘Why did you quit your old job before you had a new one lined up?’ This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them.” – Robert E. Lucas, Jr.

In addition to what’s in Anaconda, this lecture will need the following libraries:

In [1]: pip install --upgrade quantecon

2 Overview

The McCall search model [1] helped transform economists’ way of thinking about labor markets.

To clarify vague notions such as “involuntary” unemployment, McCall modeled the decision problem of unemployed agents directly, in terms of factors such as

- current and likely future wages
- impatience
- unemployment compensation
To solve the decision problem he used dynamic programming.

Here we set up McCall’s model and adopt the same solution method.

As we’ll see, McCall’s model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

Let’s start with some imports:

```python
In [2]: import numpy as np
    from numba import jit, jitclass, float64
    import matplotlib.pyplot as plt
    %matplotlib inline
    import quantecon asqe
    from quantecon.distributions import BetaBinomial
```

### 3 The McCall Model

An unemployed agent receives in each period a job offer at wage $w_t$.

The wage offer is a nonnegative function of some underlying state:

$$w_t = w(s_t) \quad \text{where} \quad s_t \in S$$

Here you should think of state process $\{s_t\}$ as some underlying, unspecified random factor that impacts on wages.

(Introducing an exogenous stochastic state process is a standard way for economists to inject randomness into their models.)

In this lecture, we adopt the following simple environment:

- $\{s_t\}$ is IID, with $q(s)$ being the probability of observing state $s$ in $S$ at each point in time, and
- the agent observes $s_t$ at the start of $t$ and hence knows $w_t = w(s_t)$,
- the set $S$ is finite.

(In later lectures, we will relax all of these assumptions.)

At time $t$, our agent has two choices:

1. Accept the offer and work permanently at constant wage $w_t$.
2. Reject the offer, receive unemployment compensation $c$, and reconsider next period.

The agent is infinitely lived and aims to maximize the expected discounted sum of earnings

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The constant $\beta$ lies in $(0, 1)$ and is called a **discount factor**.

The smaller is $\beta$, the more the agent discounts future utility relative to current utility.

The variable $y_t$ is income, equal to

- his/her wage $w_t$ when employed
- unemployment compensation $c$ when unemployed

The agent is assumed to know that $\{s_t\}$ is IID with common distribution $q$ and can use this when computing expectations.
3.1 A Trade-Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide optimally in the face of this trade-off, we use dynamic programming.

Dynamic programming can be thought of as a two-step procedure that

1. first assigns values to “states” and
2. then deduces optimal actions given those values

We’ll go through these steps in turn.

3.2 The Value Function

In order to optimally trade-off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to in next period (in this case, either employment or unemployment)

To weigh these two aspects of the decision problem, we need to assign values to states.

To this end, let $v^*(s)$ be the total lifetime value accruing to an unemployed worker who enters the current period unemployed when the state is $s \in \mathcal{S}$.

In particular, the agent has wage offer $w(s)$ in hand.

More precisely, $v^*(s)$ denotes the value of the objective function (1) when an agent in this situation makes optimal decisions now and at all future points in time.

Of course $v^*(s)$ is not trivial to calculate because we don’t yet know what decisions are optimal and what aren’t!

But think of $v^*$ as a function that assigns to each possible state $s$ the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function $v^*$ must satisfy the recursion

$$v^*(s) = \max \left\{ \frac{w(s)}{1 - \beta}, c + \beta \sum_{s' \in \mathcal{S}} v^*(s') q(s') \right\}$$  \hspace{1cm} (1)

for every possible $s$ in $\mathcal{S}$.

This important equation is a version of the Bellman equation, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

- the first term inside the max operation is the lifetime payoff from accepting current offer, since
\[
\frac{w(s)}{1 - \beta} = w(s) + \beta w(s) + \beta^2 w(s) + \cdots
\]

- the second term inside the max operation is the **continuation value**, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods.

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current state \( s \).

But this is precisely \( v^*(s) \), which is the l.h.s. of (1).

### 3.3 The Optimal Policy

Suppose for now that we are able to solve (1) for the unknown function \( v^* \).

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the r.h.s. of (1).

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given **any** \( s \), we can read off the corresponding best choice (accept or reject) by picking the max on the r.h.s. of (1).

Thus, we have a map from \( \mathbb{R} \) to \( \{0, 1\} \), with 1 meaning accept and 0 meaning reject.

We can write the policy as follows:

\[
\sigma(s) := 1 \left\{ \frac{w(s)}{1 - \beta} \geq c + \beta \sum_{s' \in S} v^*(s') q(s') \right\}
\]

Here \( 1\{P\} = 1 \) if statement \( P \) is true and equals 0 otherwise.

We can also write this as

\[
\sigma(s) := 1\{w(s) \geq \bar{w}\}
\]

where

\[
\bar{w} := (1 - \beta) \left\{ c + \beta \sum_{s'} v^*(s') q(s') \right\}
\]

Here \( \bar{w} \) (called the **reservation wage**) is a constant depending on \( \beta, c \) and the wage distribution.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

In view of (2), we can compute this reservation wage if we can compute the value function.
4 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each possible state $s \in \mathbb{S}$.

Let’s suppose that $\mathbb{S} = \{1, \ldots, n\}$.

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (1), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v^*(j)q(j) \right\} \text{ for } i = 1, \ldots, n$$  \hspace{1cm} (3)

4.1 The Algorithm

To compute this vector, we use successive approximations:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$.

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j)q(j) \right\} \text{ for } i = 1, \ldots, n$$  \hspace{1cm} (4)

Step 3: calculate a measure of the deviation between $v$ and $v'$, such as $\max_i |v(i) - v'(i)|$.

Step 4: if the deviation is larger than some fixed tolerance, set $v = v'$ and go to step 2, else continue.

Step 5: return $v$.

Let $\{v_k\}$ denote the sequence generated by this algorithm.

This sequence converges to the solution to (3) as $k \to \infty$, which is the value function $v^*$.

4.2 The Fixed Point Theory

What’s the mathematics behind these ideas?

First, one defines a mapping $T$ from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j)q(j) \right\} \text{ for } i = 1, \ldots, n$$  \hspace{1cm} (5)

(A new vector $Tv$ is obtained from given vector $v$ by evaluating the r.h.s. at each $i$)

The element $v_k$ in the sequence $\{v_k\}$ of successive approximations corresponds to $T^k v$.

- This is $T$ applied $k$ times, starting at the initial guess $v$.

One can show that the conditions of the Banach fixed point theorem are satisfied by $T$ on $\mathbb{R}^n$.

One implication is that $T$ has a unique fixed point in $\mathbb{R}^n$.

- That is, a unique vector $\bar{v}$ such that $T\bar{v} = \bar{v}$.

Moreover, it’s immediate from the definition of $T$ that this fixed point is $v^*$.

A second implication of the Banach contraction mapping theorem is that $\{T^k v\}$ converges to the fixed point $v^*$ regardless of $v$. 


4.3 Implementation

Our default for \( q \), the distribution of the state process, will be Beta-binomial.

```python
In [3]: n, a, b = 50, 200, 100  # default parameters
q_default = BetaBinomial(n, a, b).pdf()  # default choice of q
```

Our default set of values for wages will be

```python
In [4]: w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

Here’s a plot of the probabilities of different wage outcomes:

```python
In [5]: fig, ax = plt.subplots()
   ...: ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
   ...: ax.set_xlabel('wages')
   ...: ax.set_ylabel('probabilities')
   ...: plt.show()
```

![Plot of probabilities of different wage outcomes](image)

We are going to use Numba to accelerate our code.

- See, in particular, the discussion of `@jitclass` in our lecture on Numba.

The following helps Numba by providing some type

```python
In [6]: mccall_data = [
   ...: ('c', float64),  # unemployment compensation
   ...: ('β', float64),  # discount factor
   ...: ('w', float64[:]),  # array of wage values, \( w[i] = \) wage at state \( i \)
   ...: ('q', float64[:])  # array of probabilities
   ...: ]
```
Here’s a class that stores the data and computes the value on the right hand side of the Bellman equation (4).

Default parameter values are embedded in the class.

In [7]: @jitclass(mccall_data)
class McCallModel:
    def __init__(self, c=25, β=0.99, w=w_default, q=q_default):
        self.c, self.β = c, β
        self.w, self.q = w_default, q_default
    def bellman(self, i, v):
        """The r.h.s. of the Bellman equation at state i."
        # Simplify names
        c, β, w, q = self.c, self.β, self.w, self.q
        # Evaluate right hand side of Bellman equation
        max_value = max(w[i] / (1 - β), c + β * np.sum(v * q))
        return(max_value)

Based on these defaults, let’s try plotting the first few approximate value functions in the sequence \{T^k v}\.

We will start from guess \( v \) given by \( v(i) = w(i) / (1 - \beta) \), which is the value of accepting at every given wage.

Here’s a function to implement this:

In [8]: def plot_value_function_seq(mcm, ax, num_plots=6):
    """Plot a sequence of value functions.
    * mcm is an instance of McCallModel
    * ax is an axes object that implements a plot method.
    """
    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate \{i\}"
        # Update guess
        for i in range(n):
            v_next[i] = mcm.bellman(i, v)
            v[:] = v_next  # copy contents into v
        ax.legend(loc='lower right')

Now let’s create an instance of McCallModel and call the function:

In [9]: mcm = McCallModel()
You can see that convergence is occurring: successive iterates are getting closer together.

Here’s a more serious iteration effort to compute the limit, which continues until measured deviation between successive iterates is below \( \text{tol} \).

Once we obtain a good approximation to the limit, we will use it to calculate the reservation wage.

We’ll be using JIT compilation via Numba to turbocharge our loops.

```python
@jit(nopython=True)
def compute_reservation_wage(mcm, max_iter=500, tol=1e-6):
    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute the value function == #
    n = len(w)
    v = w / (1 - β)  # initial guess
    v_next = np.empty_like(v)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:
```

In [10]:
```
for i in range(n):
    v_next[i] = mcm.bellman(i, v)

error = np.max(np.abs(v_next - v))
i += 1

v[:] = v_next  # copy contents into v

# == Now compute the reservation wage == #
return (1 - β) * (c + β * np.sum(v * q))

The next line computes the reservation wage at the default parameters

In [11]: compute_reservation_wage(mcm)

Out[11]: 47.316499710024964

4.4 Comparative Statics

Now we know how to compute the reservation wage, let’s see how it varies with parameters. In particular, let’s look at what happens when we change β and c.

In [12]: grid_size = 25
R = np.empty((grid_size, grid_size))
c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        mcm = McCallModel(c=c, β=β)
        R[i, j] = compute_reservation_wage(mcm)

In [13]: fig, ax = plt.subplots()

    cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
    ctr1 = ax.contour(c_vals, β_vals, R.T)

    plt.clabel(ctr1, inline=1, fontsize=13)
    plt.colorbar(cs1, ax=ax)

    ax.set_title("reservation wage")
    ax.set_xlabel("$c$", fontsize=16)
    ax.set_ylabel("$\beta$", fontsize=16)

    ax.ticklabel_format(useOffset=False)

    plt.show()
As expected, the reservation wage increases both with patience and with unemployment compensation.

5 Computing the Optimal Policy: Take 2

The approach to dynamic programming just described is very standard and broadly applicable.

For this particular problem, there’s also an easier way, which circumvents the need to compute the value function.

Let $h$ denote the continuation value:

$$h = c + \beta \sum_{s'} v^*(s')q(s')$$  \hspace{1cm} (6)

The Bellman equation can now be written as

$$v^*(s') = \max \left\{ \frac{w(s')}{1 - \beta}, h \right\}$$

Substituting this last equation into (6) gives

$$h = c + \beta \sum_{s' \in S} \max \left\{ \frac{w(s')}{1 - \beta}, h \right\} q(s')$$  \hspace{1cm} (7)

This is a nonlinear equation that we can solve for $h$. 

10
As before, we will use successive approximations:
Step 1: pick an initial guess \( h \).
Step 2: compute the update \( h' \) via
\[
h' = c + \beta \sum_{s' \in S} \max \left\{ \frac{w(s')}{1 - \beta}, h \right\} q(s') \tag{8}
\]
Step 3: calculate the deviation \(|h - h'|\).
Step 4: if the deviation is larger than some fixed tolerance, set \( h = h' \) and go to step 2, else return \( h \).

Once again, one can use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we’re iterating on a single number, rather than an \( n \)-vector.

Here’s an implementation:

```python
In [14]: def compute_reservation_wage_two(mcm,
                      max_iter=500,
                      tol=1e-5):

    # Simplify names
    c, beta, w, q = mcm.c, mcm.beta, mcm.w, mcm.q

    # == First compute h == #
    h = np.sum(w * q) / (1 - beta)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:
        s = np.maximum(w / (1 - beta), h)
        h_next = c + beta * np.sum(s * q)
        error = np.abs(h_next - h)
        i += 1
        h = h_next

    # == Now compute the reservation wage == #

    return (1 - beta) * h
```

You can use this code to solve the exercise below.

### 6 Exercises

#### 6.1 Exercise 1

Compute the average duration of unemployment when \( \beta = 0.99 \) and \( c \) takes the following values
That is, start the agent off as unemployed, compute their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of $c$ in `c_vals`.

### 6.2 Exercise 2

The purpose of this exercise is to show how to replace the discrete wage offer distribution used above with a continuous distribution.

This is a significant topic because many convenient distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model.

Recall that $h$ in (6) denotes the value of not accepting a job in this period but then behaving optimally in all subsequent periods:

To shift to a continuous offer distribution, we can replace (6) by

$$h = c + \beta \int v'(s')q(s')ds'.$$

Equation (7) becomes

$$h = c + \beta \int \max\left\{\frac{w(s')}{1 - \beta}, h\right\} q(s')ds'.$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

Try to carry this out, setting

- the state sequence $\{s_t\}$ to be IID and standard normal and
- the wage function to be $w(s) = \exp(\mu + \sigma s)$.

You will need to implement a new version of the `McCallModel` class that assumes a lognormal wage distribution.

Calculate the integral by Monte Carlo, by averaging over a large number of wage draws.

For default parameters, use $c=25$, $\beta=0.99$, $\sigma=0.5$, $\mu=2.5$.

Once your code is working, investigate how the reservation wage changes with $c$ and $\beta$.

### 7 Solutions

#### 7.1 Exercise 1

Here’s one solution
In [15]: cdf = np.cumsum(q_default)

@jit(nopython=True)
def compute_stopping_time(w_bar, seed=1234):
    np.random.seed(seed)
    t = 1
    while True:
        # Generate a wage draw
        w = w_default[qe.random.draw(cdf)]
        # Stop when the draw is above the reservation wage
        if w >= w_bar:
            stopping_time = t
            break
        else:
            t += 1
    return stopping_time

@jit(nopython=True)
def compute_mean_stopping_time(w_bar, num_reps=100000):
    obs = np.empty(num_reps)
    for i in range(num_reps):
        obs[i] = compute_stopping_time(w_bar, seed=i)
    return obs.mean()

c_vals = np.linspace(10, 40, 25)
stop_times = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    mcm = McCallModel(c=c)
    w_bar = compute_reservation_wage_two(mcm)
    stop_times[i] = compute_mean_stopping_time(w_bar)

fig, ax = plt.subplots()
ax.plot(c_vals, stop_times, label="mean unemployment duration")
ax.set(xlabel="unemployment compensation", ylabel="months")
ax.legend()

plt.show()
7.2 Exercise 2

In [16]: mccall_data_continuous = [
    ('c', float64),  # unemployment compensation
    ('β', float64),  # discount factor
    ('σ', float64),  # scale parameter in lognormal distribution
    ('μ', float64),  # location parameter in lognormal distribution
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
] @jitclass(mccall_data_continuous)
class McCallModelContinuous:
    def __init__(self, c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000):
        self.c, self.β, self.σ, self.μ = c, β, σ, μ

        # Draw and store shocks
        np.random.seed(1234)
        s = np.random.randn(mc_size)
        self.w_draws = np.exp(μ + σ * s)

@jit(nopython=True)
def compute_reservation_wage_continuous(mcmc, max_iter=500, tol=1e-5):
    c, β, σ, μ, w_draws = mcmc.c, mcmc.β, mcmc.σ, mcmc.μ, mcmc.w_draws
    h = np.mean(w_draws) / (1 - β)  # initial guess
    i = 0
    error = tol + 1
while i < max_iter and error > tol:
    integral = np.mean(np.maximum(w_draws / (1 - β), h))
    h_next = c + β * integral
    error = np.abs(h_next - h)
    i += 1
    h = h_next

# == Now compute the reservation wage == #

return (1 - β) * h

Now we investigate how the reservation wage changes with $c$ and $β$.

We will do this using a contour plot.

In [17]: grid_size = 25
   : R = np.empty((grid_size, grid_size))
   : c_vals = np.linspace(10.0, 30.0, grid_size)
   : β_vals = np.linspace(0.9, 0.99, grid_size)
   : for i, c in enumerate(c_vals):
   :     for j, β in enumerate(β_vals):
   :         mcmc = McCallModelContinuous(c=c, β=β)
   :         R[i, j] = compute_reservation_wage_continuous(mcmc)

In [18]: fig, ax = plt.subplots()
   : cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
   : ctr1 = ax.contour(c_vals, β_vals, R.T)
   : plt.clabel(ctr1, inline=1, fontsize=13)
   : plt.colorbar(cs1, ax=ax)
   : ax.set_title("reservation wage")
   : ax.set_xlabel("$c$", fontsize=16)
   : ax.set_ylabel("$β$", fontsize=16)
   : ax.ticklabel_format(useOffset=False)
   : plt.show()
References