Worked Bioninformatics, Statistics, and Machine Learning Examples

Stephen Woloszynek

2020-02-24

Contents

1 Introduction

This is a bunch of stuff that I either complete wrote, lectured, or adapted during my time completing my PhD. It mostly consists material I used to intro-level statistics classes for biomedical engineers, some machine learning code, and topic model derivations related to my thesis work.

2 Dynamic Programming

2.1 Introduction

Dynamic programming makes computationally demanding problems manageable by dividing them into a set of subproblems. On its surface, this might sound like a divide-and-conquer approach, but it is in fact different. D&C solves *disjoint* subproblems recursively. Each subproblem must be calculated from scratch, and their results are combined to reach a final solution. This results in more work than necessary. DP approaches, on the other hand, solve *overlapping* subproblems and save their results for later use; hence, each subproblem needs to be calculated just once, and overlapping subproblems can inform each other to lessen the computational burden.

DP alogorithms are designed as follows (from Introduction to Algorithms, 3rd ed.):

1. Characterize the structure of an optimal solution
2. Recusively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

2.2 Rod cutting

Say we have a silver rod, and a piece of length i would net us p_i dollars, such that we have the following price table:

```
set.seed(12)

N <- 25
C <- 50
l <- seq_len(N)
p <- sort(sample(seq_len(C),N,replace=TRUE),decreasing=FALSE)

matrix(p,nrow=1,dimnames=list('price',l))
```

```
##       1 2 3 4 5 6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## price 2 4 5 5 8 8 13 16 18 18 24 26 26 27 27 28 34 38 39 42 43 44 46 46 48
```

We'd like to maximize our profit, so we need to figure out a way to cut our rod such that our pieces total to the largest value. We can do this recursively:

```
cut_recursive <- function(prices,length){
  if (length == 0) return(0)
  q <- -Inf
  for (i in seq_len(length)){
    #cat(sprintf('len=%s,val=%s\n',length-i,cut_recursive(prices,length-i)))
    q <- max(q,prices[i] + cut_recursive(prices,length-i))
  }
  return(q)
}
```

```
cut_recursive(p,5)
```

```
## [1] 10
```

```
cut_recursive(p,15)
```

```
## [1] 32
```

```
cut_recursive(p,20)
```

```
## [1] 42
```

```
cut_aux <- function(prices,length){
  if (R[length + 1] >= 0) return(R[length + 1])
  if (length == 0){
    q <- 0
  }else{
    q <- -Inf
    for (i in seq_len(length)){
      cat(sprintf('len=%s,val=%s\n',length-i,cut_aux(prices,length-i)))
      q <- max(q,prices[i] + cut_aux(prices,length-i))
    }
  }
  R[length+1] <<- q
  return(q)
}

cut_memoized <- function(prices,length){
```

```
  R <<- rep(-Inf,length + 1)
  return(cut_aux(prices,length))
}
```

```
cut_memoized(p,5)
```

```
## len=0,val=0
## len=1,val=2
## len=0,val=0
## len=2,val=4
## len=1,val=2
## len=0,val=0
## len=3,val=6
## len=2,val=4
## len=1,val=2
## len=0,val=0
## len=4,val=8
## len=3,val=6
## len=2,val=4
## len=1,val=2
## len=0,val=0
```

```
## [1] 10
```

```
cut_bottomup <- function(prices,length){
  r <- c(0,rep(-Inf,length))
  for (i in seq_len(length)){
    q <- -Inf
    for (j in seq_len(i))
      q <- max(q,prices[j] + r[i-j+1])
    r[i+1] <- q
  }
  return(r[length+1])
}
```

```
cut_bottomup(p,10)
```

```
## [1] 20
```

```
cut_bottomup(p,15)
```

```
## [1] 32
```

```
cut_bottomup(p,20)
```

```
## [1] 42
```

2.3 Fibonacci rabbits

Let's try and write a script that can solve a potentially computationally burdonsome problem – a problem involving reproducing rabbits. Say we start with 1 baby rabbit (age 0) at month 1. When the rabbit reaches 1 month of age, it can reproduce, producing a new baby rabbit the following month (month 3). On month 4, the baby rabbit can now reproduce, but our original rabbit will also reproduce, and so on. The rules are therefore

- Rabbits age each month
- Baby (age 0) rabbits cannot reproduce

And here is a diagram showing the process:

Now focus on the number of rabbits for each month: 1, 1, 2, 3, 5, 7. It's the fibonacci sequence, where the current months total is a sum of the previous month's total. We can therefore make a function that can *recursively* calculate the number of rabbits, using the finonacci sequence, only requiring the number of months the process will span across.

A quick aside: recursive algorithms are hard. They take some work to get a hang of them. I would not worry about either trying to write recursive algorithms or completely understanding how the code below works. The point of showing them is that there's often a natural way to tackle a programming problem, but it's not necessarily always the *best* way.

```
fib <- function(n){

  if (n==1 || n==2){
    return(1)
  }else{
    return(fib(n-1) + fib(n-2))
  }

}

fib(5)
```

```
## [1] 5
```

```
for (n in seq_len(25)) cat(fib(n),' ')
```

```
## 1  1  2  3  5  8  13  21  34  55  89  144  233  377  610  987  1597  2584  4181  6765  10946  17711
```

Let's change the problem a little bit. Let's assume now that the rabbits can die after k months. For $k = 3$, we'd have the following process:

We can write another recurssive algorithm to tackle this:

```
lifespan_inner <- function(n,y,Y){

  if (n==1){
    return(1)
  }else if (y==Y){
    return(lifespan_inner(n-1,y-1,Y))
  }else if(y==1){
    lifespan_inner(n-1,Y,Y)
  }else{
    return(lifespan_inner(n-1,y-1,Y) + lifespan_inner(n-1,Y,Y))
  }

}

lifespan <- function(n,y){

  return(lifespan_inner(n,y,y))

}

for (n in seq_len(25)) cat(lifespan(n,3),' ')
```

```
## 1  1  2  2  3  4  5  7  9  12  16  21  28  37  49  65  86  114  151  200  265  351  465  616  816
```

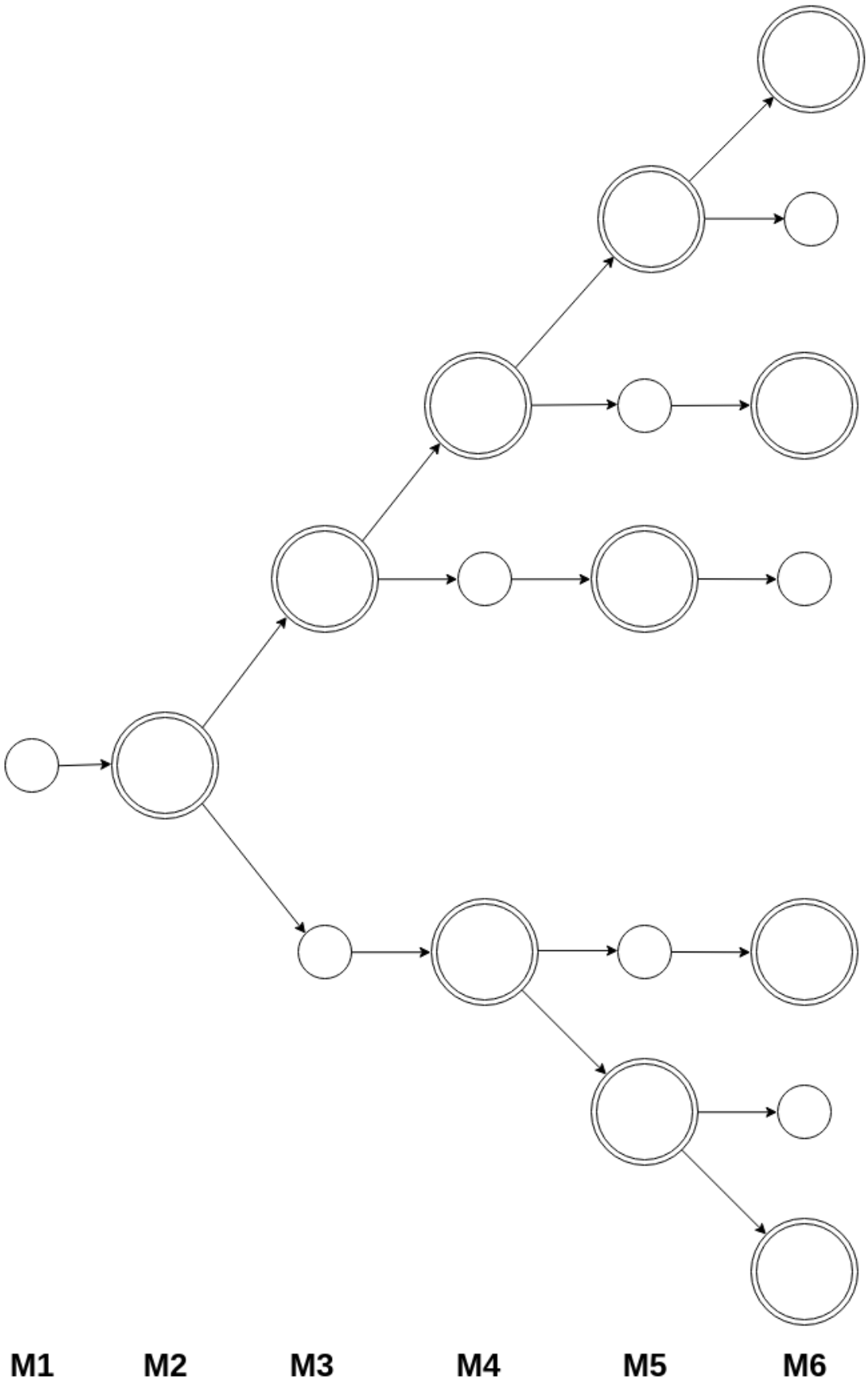

Figure 1:

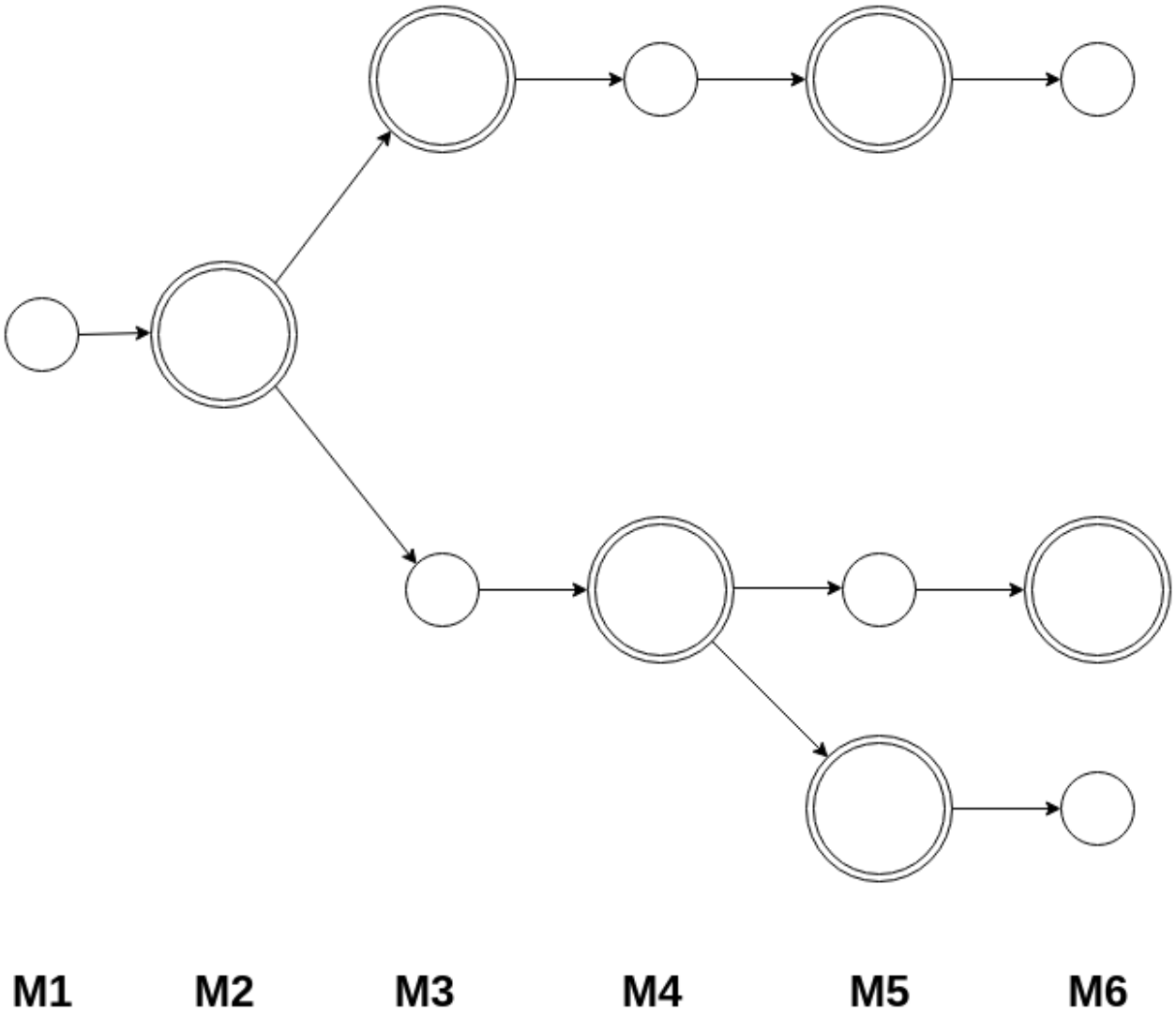

Figure 2:

But look how much time it takes if we ramp up the number of months to 60:

```
t1 <- Sys.time()
for (n in seq_len(60)) cat(lifespan(n,3),' ')
t2 <- Sys.time()
cat('Time elapsed:',round(t2-t1,1),'minutes.')
```

```
## Time elapsed: 5.3 minutes.
```

It's slow!

Now we'll use a *dynamic programming* approach. You're going to need dynamic programming later in the course for genomic sequence alignment, so it's worth exploring the type of speedup one can obtain with a quite intuitive method, particularly when aimed at computationally demanding tasks often seen in genomics.

Again, dynamic programming saves a ton of time and resources by sweeping through a problem as a set of smaller subproblems, while storing the results of each subproblem as you go. Think about the rabbit flow chart. If we wanted to know the number of rabbits present on month 1000, we'd have to add months 999 and 998 together, which require information from months 996 through 998, and so on. A recursive algorithm would calculate the result of 999 independently of 998, and *then* add them together. Dynamic programming, on the other hand, would have the results from those previously months stored, simply requiring us to look them up.

The game here involves the following:

1. Make a $n \times y$ matrix M, where n is the number of months and y is the rabbit's lifespan.
2. Row 1 will represent month 1, column 1 will represent baby rabbits, and column y will represent the final month of life for an adult rabbit.
3. Each subsequent row will be a running tally of the number of rabbits in each age group. Because each month is updated sequentially, you only need the information of a previous row (month) to update a current row (month).

```
dynprog <- function(m,y,p=FALSE){

   mat <- matrix(0,m,y)
   mat[1,1] <- 1

   for (i in 2:m){
      y1 <- mat[i-1,]
      y2 <- mat[i,]

      y2[1] <- sum(y1[-1])
      y2[-1] <- y1[-y]

      mat[i-1,] <- y1
      mat[i,] <- y2

      if (p){
        cat(sprintf('y%s:\t%s',i,paste0(mat[i,],collapse='\t')))
        line <- readline('')
      }
   }

   return(rowSums(mat))
}
```

Here are some example answers to check. Note that there may be some variability given the max integer number in R, which is set in the options. If you get the right answer for smaller parameterizations, then your

code is correct.

```
dynprog(25,5,TRUE)
```

```
## y2:  0   1   0   0   0
## y3:  1   0   1   0   0
## y4:  1   1   0   1   0
## y5:  2   1   1   0   1
## y6:  3   2   1   1   0
## y7:  4   3   2   1   1
## y8:  7   4   3   2   1
## y9:  10  7   4   3   2
## y10: 16  10  7   4   3
## y11: 24  16  10  7   4
## y12: 37  24  16  10  7
## y13: 57  37  24  16  10
## y14: 87  57  37  24  16
## y15: 134 87  57  37  24
## y16: 205 134 87  57  37
## y17: 315 205 134 87  57
## y18: 483 315 205 134 87
## y19: 741 483 315 205 134
## y20: 1137    741 483 315 205
## y21: 1744    1137    741 483 315
## y22: 2676    1744    1137    741 483
## y23: 4105    2676    1744    1137    741
## y24: 6298    4105    2676    1744    1137
## y25: 9662    6298    4105    2676    1744
```

```
##  [1]     1     1     2     3     5     7    11    17    26    40    61    94
## [13]   144   221   339   520   798  1224  1878  2881  4420  6781 10403 15960
## [25] 24485
```

```
dynprog(50,5)[50]
```

```
## [1] 1085554510
```

```
dynprog(70,6)[70]
```

```
## [1] 2.685139e+13
```