MB 1A Practical 16: Dynamic programming

Andrew Firth

Aim: To write a dynamic programming algorithm for aligning two sequences.

In this practical, you will continue using the techniques you have already learnt (particularly for and while loops and if/else statements) to write a simple but powerful dynamic programming algorithm.

Notes:

- The most common mistakes seem to be (i) losing track of "{}"s, e.g. placing code after a closing "}" that should be within a "{}" function block, and (ii) forgetting that variables are undefined outside of the function {} in which they are defined.
- I tend to use "=" in my code where other people use "<-". Don't worry about this.

Reminder: If you define a variable inside a function, you will not be able to access the variable outside of that function.

For example, the `print(y)` line in this function will work:

```
my_function <- function(x){
  y = x + 1
  print(y)
}
my_function(12)
```

```
## [1] 13
```

The `print(y)` line after this function will error:

```
my_function <- function(x){
  y = x + 1
}
print(y)
my_function(12)
```

Design

We are going to create a program (R function) that reads in two nucleotide sequences (e.g. CGGATG and CAGTG) as well as "match", "mismatch" and "gap" scores. The program will then build a dynamic programming matrix, fill it out according to the sequences and scoring scheme, perform the trace back to find the optimal alignment, and write out the optimal alignment. We will start with a global alignment.

To start, open a new R script and save this (currently blank) script to an appropriately named file (e.g. DP_nt.R). Remember to check the "Source on Save" check box.

Reading in the parameters

Let's start with reading in the parameters: match_score, mismatch_score, gap_score, sequence_1 and sequence_2. We also want to check that they get read in as expected before proceeding, which we can do by printing them right back out again.

```
DP_nt <- function(match_score,mismatch_score,gap_score,sequence_1,sequence_2)

{
  print(match_score)
  print(mismatch_score)
  print(gap_score)
  print(sequence_1)
  print(sequence_2)
}
```

Save the script and make sure you've sourced it. Then let's see if it works by going to the console pane, and typing

```
DP_nt(1,-1,-2,"CGGATG","CAGTG")
```

```
## [1] 1
## [1] -1
## [1] -2
## [1] "CGGATG"
## [1] "CAGTG"
```

You should see exactly the same parameters that you typed in. Note that this is exactly the same as the global alignment example used in lecture notes.

Now that we know it is reading in the parameters correctly, you can if you like comment out the print statements in the script (add a "#" at the beginning of a line to prevent it from being executed).

Length of input sequences

Next we need to find the lengths of our input sequences. We can do this with the **nchar()** function.

```
nchar("ABCDEFG")
```

```
## [1] 7
```

So we can add the following to our script

```
# Length of input sequences.
len_seq1 = nchar(sequence_1)
len_seq2 = nchar(sequence_2)
```

Again, we should check that produces the expected results by (temporarily) adding the lines

```
print(len_seq1)
print(len_seq2)
```

to our script. Now if you run the script in the console pane, you should get the following

```
DP_nt(1,-1,-2,"CGGATG","CAGTG")
```

```
## [1] 6
## [1] 5
```

Converting the input sequences to character vectors

"CGGATG" is a chracter string, but we want to put it into a vector to make it easy to reference each position of each sequence. We can do this with the function **substr()**, e.g.

```
myWord = "ELEPHANT"
myWord_last4letters = substr(myWord,5,8)
print(myWord_last4letters)
```

```
## [1] "HANT"
```

The following code converts the two input sequences into vectors, one character per entry. You can add this to the DP_nt.R script:

```
# Convert sequences into vectors, one character per entry.
  seq1 <- vector(mode="character", length=len_seq1)
  for (i in 1:len_seq1)
  {
    seq1[i] <- substr(sequence_1,i,i)
  }
  seq2 <- vector(mode="character", length=len_seq2)
  for (i in 1:len_seq2)
  {
    seq2[i] <- substr(sequence_2,i,i)
  }
```

To check this we can add the temporary code

```
print(sequence_1)
print(sequence_2)
print(seq1)
print(seq2)
print(seq1[1])
print(seq2[1])
```

and run the script

```
DP_nt(1,-1,-2,"CGGATG","CAGTG")
```

```
## [1] "CGGATG"
## [1] "CAGTG"
## [1] "C" "G" "G" "A" "T" "G"
## [1] "C" "A" "G" "T" "G"
## [1] "C"
## [1] "C"
```

Note how the initial "single-word" input sequences are now split into individual characters which can be referenced as elements of a vector for each sequence.

In case you're wondering, your (clean) DP_nt.R script should now look like this:

```
 DP_nt <- function(match_score,mismatch_score,gap_score,sequence_1,sequence_2)

{

# Length of input sequences.
  len_seq1 = nchar(sequence_1)
  len_seq2 = nchar(sequence_2)

# Convert sequences into vectors, one character per entry.
```

```
  seq1 <- vector(mode="character", length=len_seq1)
  for (i in 1:len_seq1)
  {
    seq1[i] <- substr(sequence_1,i,i)
  }
  seq2 <- vector(mode="character", length=len_seq2)
  for (i in 1:len_seq2)
  {
    seq2[i] <- substr(sequence_2,i,i)
  }

 }
```

Initializing the dynamic programming matrices

For our dynamic programming grid, we want an $(m_2+1) \times (m_1+1)$ array to fill out with scores, where m_1 is the number of characters in sequence 1 and m_2 is the number of characters in sequence 2. We also need a way to keep track of the arrows showing the possible paths that we can take from one cell to another. One way to do this is to have three more $(m_2+1) \times (m_1+1)$ arrays, one for each of vertical (v), horizontal (h) and diagonal (d) arrows. If we can get to cell (j, i) (where j is vertical position and i is horizontal position) via a diagonal arrow from cell $(j-1, i-1)$ then we can put a "1" in cell (j, i) of the diagonal (d) traceback array, and otherwise we can put a "0" in cell (j, i). Similarly for the horizontal (h) and vertical (v) traceback arrays.

First we need to initialize our arrays to 0 – i.e. to make empty $(m_2+1) \times (m_1+1)$ matrices where every value is 0. Note that, for compatibility with the lectures, we have sequence 1 going across the top (i.e. there are len_seq1 + 1 columns) and sequence 2 going down the side (i.e. there are len_seq2 + 1 rows).

```
# Initialize matrices to 0.
# Make empty (m2 + 1) x (m1 + 1) matrices where every value is 0.

# The DP matrix will contain the scores.
  DPmatrix <- matrix(0,len_seq2+1,len_seq1+1)

# The traceback matrices will contain information about how we got to each cell.
  traceback_v <- matrix(0,len_seq2+1,len_seq1+1)
  traceback_h <- matrix(0,len_seq2+1,len_seq1+1)
  traceback_d <- matrix(0,len_seq2+1,len_seq1+1)
```

Since we are trying to perform global alignments, the next stage is to initialize the first row and first column. As shown in the lectures, these represent the scores for inserting gaps, with the $(j+1)$th row or $(i+1)$th column containing j or i, respectively, times the gap penalty.

Note that our indices are offset by +1 since the first column and first row correspond to initial gap characters. I.e. the first character of sequence_1 corresponds to column 2 and the first character of sequence_2 corresponds to row 2, etc. Thus, to reference the cell corresponding to the (jth,ith) characters, we use the indices $(j+1, i+1)$.

```
# Initialize the first row and first column of the arrays.

# For every column in the matrix, make the value in row 1 = gap_score * i, where
#   i+1 is the column number. These cells can only be accessed horizontally so we
#   update the horizontal traceback matrix traceback_h to 1 instead of 0 for each
#   cell.
DPmatrix[1,1] = 0
for (i in 1:len_seq1)
{
```

```
  DPmatrix[1,i+1] = gap_score * i
  traceback_h[1,i+1] = 1
}

# For every row in the matrix, make the value in column 1 = gap_score * j, where
#   j+1 is the row number. These cells can only be accessed vertically so we
#   update the vertical traceback matrix traceback_v to 1 instead of 0 for each
#   cell.
for (j in 1:len_seq2)
{
  DPmatrix[j+1,1] = gap_score * j
  traceback_v[j+1,1] = 1
}
```

Note, if we were doing a local alignment, we would skip this step since the arrays are already initialized to 0. (Remember for local alignments, we want the first row and first column filled out with zeros.)

Filling out the dynamic programming matrices

Now things get a little harder. We need to fill out the rest of the matrix using the dynamic programming rules covered in lectures:

Case 1 (diagonal arrow): $S_{j,i} = S_{j-1,i-1} + s(j,i)$, where $s(j,i)$ = match_score for a match or mismatch_score for a mismatch

Case 2 (horizontal arrow): $S_{j,i} = S_{j-1,i}$+ gap_score

Case 3 (vertical arrow): $S_{j,i} = S_{j,i-1}$+ gap_score

For each of the remaining cells in the matrix, we will try all three of these cases and see which gives the highest score. We will then add a "1" in this position to the diagonal, vertical or horizontal traceback matrix (or matrices) for the highest scoring direction (or directions).

See if you can work out what the different components of the following do, then add it to your DP_nt.R script:

```
# Fill out the rest of the matrix.
  for (i in 1:len_seq1)
  {
    for (j in 1:len_seq2)
    {
      if (seq1[i] == seq2[j])
      {
          s_ji = match_score
      } else {
          s_ji = mismatch_score
      }
      # Calculate case 1
      diagonal_score = DPmatrix[j,i] + s_ji
      # Calculate case 2
      horizontal_score = DPmatrix[j+1,i] + gap_score
      # Calculate case 3
      vertical_score = DPmatrix[j,i+1] + gap_score
      # Find the highest of these three scores
      max_score = max(c(vertical_score, horizontal_score,
                        diagonal_score))
```

```
      # Record this score
      DPmatrix[j+1,i+1] = max_score
      # Update from 0 to 1 at this position in the traceback matrix for the case
      #   (or cases) that produced the highest score.
      if (max_score == vertical_score)
      {
        traceback_v[j+1,i+1] = 1
      }
      if (max_score == horizontal_score)
      {
        traceback_h[j+1,i+1] = 1
      }
      if (max_score == diagonal_score)
      {
        traceback_d[j+1,i+1] = 1
      }
    }
  }

# Write out matrices (uncomment these lines for testing).
#print("DPmatrix")
#print(DPmatrix)
#print("vertical arrows")
#print(traceback_v)
#print("horizontal arrows")
#print(traceback_h)
#print("diagonal arrows")
#print(traceback_d)
```

If you now run the script with, e.g., the

```
#print("DPmatrix")
#print(DPmatrix)
```

lines uncommented, you should get

```
DP_nt(1,-1,-2,"CGGATG","CAGTG")
```

```
## [1] "DPmatrix"
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    0   -2   -4   -6   -8  -10  -12
## [2,]   -2    1   -1   -3   -5   -7   -9
## [3,]   -4   -1    0   -2   -2   -4   -6
## [4,]   -6   -3    0    1   -1   -3   -3
## [5,]   -8   -5   -2   -1    0    0   -2
## [6,]  -10   -7   -4   -1   -2   -1    1
```

You can also uncomment the **print()** lines for the traceback arrays and check they agree with the example aligning the same sequences in the lecture notes.

If something doesn't work, you can find the complete code further below and check your code agrees with it.

Performing the traceback

Now that we have the completed the DP and traceback arrays, we need to perform the traceback to read off the optimal alignment. For simplicity, we will just output one alignment if there are multiple possibilities,

giving traceback_d priority over traceback_h, and traceback_h priority over traceback_v (for no particular reason).

Since this is a global alignment, we're going to start the traceback from the bottom right cell - i.e. index [len_seq2+1,len_seq1+1]. We generate the alignment backwards, so at some point, we're going to need to reverse it. There are various ways to do this, but the way I've chosen is to have two character vectors, alnseq1 and alnseq2, one for each aligned sequence, and to fill in the vectors from the highest index working towards the lowest index, i.e. working from the end of the alignment towards the beginning. So what is the highest index? We don't know *a priori* how long the alignment will be, but we do know that the maximum possible length of the alignment is len_seq1 + len_seq2, so we can just start from index len_seq1 + len_seq2, work backwards, and if we have some space left over at the low-index end of the vectors after we have finished the traceback, then it doesn't really matter.

```
# Initialize aligned sequence vectors.
# Make empty vectors in which to store the aligned sequences.
alnseq1 <- vector(mode="character", length=len_seq1+len_seq2)
alnseq2 <- vector(mode="character", length=len_seq1+len_seq2)
```

See if you can work out what the different components of the following code do.

```
# Perform traceback to get the optimal alignment.

# alnpos is your current position in the alignment of the two sequences.
# We start at the end of the alignment.
alnpos = len_seq1 + len_seq2

# i0 and j0 are your current positions in the sequences you are aligning.
# We start at the bottom right cell - i.e. the ends of the sequences.
i0 = len_seq1
j0 = len_seq2

while (i0 >= 0 && j0 >= 0)
{
  print(paste(j0,i0))

  if (1 == traceback_d[j0+1,i0+1])
  {
    # If this position in the matrix scored highest when accessed diagonally.
    print("d")
    # Fill in the alignment with the nucleotides at the current position in
    #   sequence one and the current position in sequence two.
    alnseq1[alnpos] = seq1[i0]
    alnseq2[alnpos] = seq2[j0]
    # Move to the next position in sequence one and in sequence two.
    i0 = i0 - 1
    j0 = j0 - 1

  } else if (1 == traceback_h[j0+1,i0+1]) {
    # If this position in the matrix scored highest when accessed horizontally.
    print("h")
    # Fill in the alignment with the nucleotide at the current position in
    #   sequence one and a gap in sequence two.
    alnseq1[alnpos] = seq1[i0]
    alnseq2[alnpos] = "-"
    # Move to the next position in sequence one.
    i0 = i0 - 1
```

```
  } else if (1 == traceback_v[j0+1,i0+1]) {
    # If this position in the matrix scored highest when accessed vertically.
    print("v")
    # Fill in the alignment with the nucleotide at the current position in
    #   sequence two and a gap in sequence one.
    alnseq1[alnpos] = "-"
    alnseq2[alnpos] = seq2[j0]
    # Move to the next position in sequence two
    j0 = j0 - 1

  } else {
    print("Finished")
    break
  }
  alnpos = alnpos - 1
}
```

You should compare the traceback path with the example at the end of lecture 1 (also in the lecture notes).

i0 and j0 count the indices in the DP and traceback arrays. alnpos counts the index in the aligned sequence vectors alnseq1 and alnseq2. At each cell, we look through the traceback arrays to find the first arrow that we can follow back to a previous cell (remember that a "1" in a traceback array means there is an arrow and a "0" means there is no arrow), filling out the alnseq1 and alnseq2 vectors accordingly as we go. We finish when we end up in the upper left cell.

Note all the **print()** statements. You should get into the habit if printing things out often during software development so you can check every detail is working as expected before moving on. Once everything is working, you can comment out or remove these superfluous **print()** statements.

Have a think about the **while(){}** loop in the block of code above. In theory, we can exit the loop in two ways:

- If the while condition **(i0 >= 0 && j0 >= 0)** becomes false.
- Through the **break** statement in the last option of the **if/else** block.

Question: Which of these two exit possibilities is the one that is actually used?

When we initialized our traceback arrays, we set all values to 0 (i.e. all cells in the dynamic programming grid have no associated arrows). If we have written our code correctly, then the top left cell should still have no arrows associated with it, whereas any other cell should be tracebackable to the top left cell. Thus, our **if/else** block will only default to the last option (i.e. none of the traceback_v, _h and _d arrays containing a 1) if we have reached the top left cell. This is the cell [1,1], corresponding to i0 = j0 = 0. Thus, we should *always* exit the **while** loop via the **break** statement in the last option of the **if/else** block *before* i0 or j0 can become negative. Therefore the while condition **(i0 >= 0 && j0 >= 0)** should *never* become false.

So why do we use it? Well, it could be written differently; we could even write **while (TRUE) {}** to establish an infinite loop that could be exited only via an internal **break** statement. One advantage of writing it as **while (i0 >= 0 && j0 >= 0) {}** is that we have an extra safeguard in case of a glitch in the code to stop us running into negative i0/j0 values and trying to reference non-existent negative indices in the traceback and sequence arrays. But maybe the extra safeguard is not a good thing, as it would make us less likely to notice and fix the glitch ... :-)

Reading out the alignment

So now we have our aligned sequences in the vectors alnseq1 and alnseq2 and all that remains is to write out the alignment:

```
# Write out alignment.
print(paste(alnseq1,collapse = ""))
print(paste(alnseq2,collapse = ""))
```

The complete global alignment program

Your full (clean) DP_nt.R script should now look something like the following.

```
DP_nt <- function(match_score,mismatch_score,gap_score,sequence_1,sequence_2)

{

# Length of input sequences.
  len_seq1 = nchar(sequence_1)
  len_seq2 = nchar(sequence_2)

# Convert sequences into vectors, one character per entry.
  seq1 <- vector(mode="character", length=len_seq1)
  for (i in 1:len_seq1)
  {
    seq1[i] <- substr(sequence_1,i,i)
  }
  seq2 <- vector(mode="character", length=len_seq2)
  for (i in 1:len_seq2)
  {
    seq2[i] <- substr(sequence_2,i,i)
  }

# Initialize matrices to 0.

# The DP matrix contains the scores.
  DPmatrix <- matrix(0,len_seq2+1,len_seq1+1)

# The traceback matrices contain information about how we got to each cell.
  traceback_v <- matrix(0,len_seq2+1,len_seq1+1)
  traceback_h <- matrix(0,len_seq2+1,len_seq1+1)
  traceback_d <- matrix(0,len_seq2+1,len_seq1+1)

# Initialize the first row and first column of the arrays.
  DPmatrix[1,1] = 0
  for (i in 1:len_seq1)
  {
    DPmatrix[1,i+1] = gap_score * i
    traceback_h[1,i+1] = 1
  }
  for (j in 1:len_seq2)
  {
    DPmatrix[j+1,1] = gap_score * j
    traceback_v[j+1,1] = 1
  }
```

```
# Fill out the rest of the matrix.
  for (i in 1:len_seq1)
  {
    for (j in 1:len_seq2)
    {
      if (seq1[i] == seq2[j])
      {
          s_ji = match_score
      } else {
          s_ji = mismatch_score
      }
      horizontal_score = DPmatrix[j+1,i] + gap_score
      vertical_score = DPmatrix[j,i+1] + gap_score
      diagonal_score = DPmatrix[j,i] + s_ji
      max_score = max(c(vertical_score, horizontal_score,
                        diagonal_score))
      DPmatrix[j+1,i+1] = max_score
      if (max_score == vertical_score)
      {
        traceback_v[j+1,i+1] = 1
      }
      if (max_score == horizontal_score)
      {
        traceback_h[j+1,i+1] = 1
      }
      if (max_score == diagonal_score)
      {
        traceback_d[j+1,i+1] = 1
      }
    }
  }

# Initialize aligned sequence vectors.
  alnseq1 <- vector(mode="character", length=len_seq1+len_seq2)
  alnseq2 <- vector(mode="character", length=len_seq1+len_seq2)

# Perform traceback to get the optimal alignment.
  alnpos = len_seq1 + len_seq2
  i0 = len_seq1
  j0 = len_seq2
  while (i0 >= 0 && j0 >= 0)
  {
    if (1 == traceback_d[j0+1,i0+1])
    {
      alnseq1[alnpos] = seq1[i0]
      alnseq2[alnpos] = seq2[j0]
      i0 = i0 - 1
      j0 = j0 - 1
    } else if (1 == traceback_h[j0+1,i0+1]) {
      alnseq1[alnpos] = seq1[i0]
      alnseq2[alnpos] = "-"
      i0 = i0 - 1
    } else if (1 == traceback_v[j0+1,i0+1]) {
```

```
      alnseq1[alnpos] = "-"
      alnseq2[alnpos] = seq2[j0]
      j0 = j0 - 1
    } else {
      break
    }
    alnpos = alnpos - 1
  }

# Write out alignment.
  print(paste(alnseq1,collapse = ""))
  print(paste(alnseq2,collapse = ""))

}
```

Let's try running it on the sequences given in the lecture notes:

```
DP_nt(1,-1,-2,"CGGATG","CAGTG")
```

```
## [1] "CGGATG"
## [1] "CAG-TG"
```

Now try aligning some longer sequences, such as

AAAAAACUCCAAGGGGCCCCAGAACACCAAGGGGCCCAAAAACCAGAAGACAUCCACUCA

and

AAAAAACGAGAGAAGGGGCCGCAGGCGGGGCCCAAAGAAUGGUACCAA

```
DP_nt(1,-1,-2,"AAAAAACUCCAAGGGGCCCCAGAACACCAAGGGGCCCAAAAACCAGAAGACAUCCACUCA",
      "AAAAAACGAGAGAAGGGGCCGCAGGCGGGGCCCAAAGAAUGGUACCAA")
```

```
## [1] "AAAAAAC--UCCAAGGGGCCCCAGAACACCAAGGGGCCCAAAAACCAGAAGACAUCCACUCA"
## [1] "AAAAAACGAGAGAAGGGGCCGCAG----GC--GGGGCCC--AAA-GA-AUGGUA-CCA---A"
```

Note that pretty much all "real" sequence alignment programs will use affine gap penalties (i.e. penalty for a gap of length l is $o + (l-1)e$ where o is the gap opening penalty, e is the gap extension penalty and $o > e$) and so will get somewhat more biologically realistic alignments than we are getting with our linear gap penality scheme, e.g.

```
AAAAAAC--UCCAAGGGGCCCCAGAACACCAAGGGGCCCAAAAACCAGAAGACAUCCACUCA
AAAAAACGAGAGAAGGGGCCGCAG------GCGGGGCCCAAAGAAUGG--------UACCAA
```

But it is not too difficult to extend the above program to use affine gap penalties (you don't need to do this).

You could also think about what you would need to do to adapt this program to perform *semi-global* and/or *local* alignments.
