Lecture 27: Dynamic Programming Part 2

11:00 AM, Mar 24, 2021

Contents

Objectives

By the end of these notes, you will know

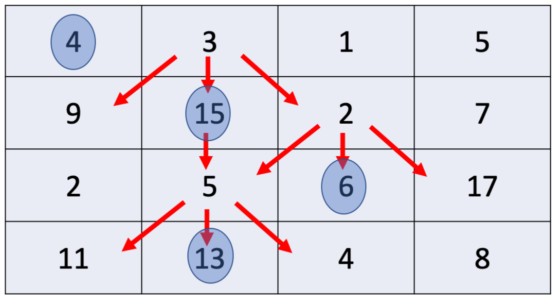
- how to approach a 2-dimensional dynamic programming problem

Last class, we searched for a way to optimize our selection of sweets from a display case. Our input data, the rating of each sweet, was uni-dimensional, in that we had a set of options to choose among that were ordered only in one dimension.

Today. we turn to a problem in which the options are ordered in two dimensions, each of which contributes to which information can be considered as we optimize our solution.

1 Maximizing Halloween Candy

Imagine that you live in a neighborhood that is laid out in a grid. It's Halloween, and you get to stop at once house on each east-west street (in each row) to collect candy. The idea is that you will start from some house in the top row, then make your way down to the bottom row. From each house you visit, the next one has to be either directly below or diagonally adjacent. The following image shows the idea (only some of the red "next" arrows are present to avoid clutter). The blue highlights show the optimal choice.



```
class Halloween() {

  def maxCandy(candyAvailable: Array[Array[Int]]): Int = {
    val height = candyAvailable.length - 1
    val width = candyAvailable(0).length - 1

    // get the available candy from the input, returning
    // 0 if the requested row/col is not in the table
    def candyVal(row: Int, col: Int): Int = {
      if ((row > height) | (col > width) | (col < 0))
        0
      else
        candyAvailable(row)(col)
    }

    def maxRec(row: Int, col: Int): Int = {
      if (row == height)
        candyVal(row, col)
      else
        candyVal(row, col) +
          max(maxRec(row + 1, col - 1),
            max(maxRec(row + 1, col), maxRec(row + 1, col + 1)))
    }

    // get the largest candy total from the top row
    List.range(0, width).map(c => maxRec(0, c)).max
  }
}
```

Figure 1: The unoptimized recursive solution

To see the computation that underlies the table, you can look at the following sample Excel sheet, which computes two tables from the input data (in gray in the sheet). The pink table shows the table that holds the max candy you can get at each house, assuming you filled the table from the top row to the last row. The green table shows the max-candy table, this time filled from the last row to the top row. Each table shows the same max answer (circled in blue). While the circled cell in each table is part of the optimal path (from the blue circles on the gray table), our code will still need to compute the actual path.

A DP-solution would compute one of these two tables (either choice works) to store the results of computing the max candy available at each house. The solution we build today will follow the formulas that generated the green table.

https://brown-cs18-master.github.io/content/lectures/27dynprog2/tables.xlsx

Figure 1 shows the un-optimized recursive program that solves this problem (the heart of the recursive function matches the computation in the pink/green tables). Figure 2 is the DP solution that computes the green table, storing it in an array. See the code comments for additional information.

```
class HalloweenLtoF() {  // LtoF stands for "last to first row"

  def maxCandy(candyAvailable: Array[Array[Int]]): Int = {
    val height = candyAvailable.length - 1
    val width = candyAvailable(0).length - 1
    // the array for holding previously computed values
    val candyComputed = Array.ofDim[Int](width+1, height+1)

    // get the available candy from the input, returning
    // 0 if the requested row/col is not in the table.
    def candyVal(row: Int, col: Int): Int = {
      if ((col > width) | (col < 0))
        0
      else
        candyComputed(row)(col)
    }

    // set up the base cases of the computed values
    for (col <- 0 to width) {
      candyComputed(height)(col) = candyAvailable(height)(col)
    }

    // set up the rest of the computed values
    for (row <- height - 1 to 0 by -1)
      for (col <- 0 to width)
        candyComputed(row)(col) =
          candyAvailable(row)(col) +
            max(candyVal(row + 1, col - 1),
              max(candyVal(row + 1, col), candyVal(row + 1, col + 1)))

    // lookup the computed answer
    // get the largest candy total from the top row
    List.range(0, width).map(c => candyComputed(0)(c)).max
  }
}
```

Figure 2: The optimized solution for computing only max candy available

1.1 Searching for paths

To also search for the optimal *path* (instead of just the optimal *value*), we need to compute a second table that holds the path of cells that got to the corresponding value in the green table. Slide 5 of the deck shows the path table in the upper right. Each cell in the darker-green table holds a list of tuples for the coordinates that were used to compute the max value in the lighter-green table. If you look at the list of coordinates in the top-left corner of the dark green table, you see that they correspond to the path taken with the blue circles on the gray table.

Figures 3 and 4 shows the code augmented with a second table (called `bestPath`) that mimics the dark green table. The two tables get computed simultaneously – as we figure out which cell to use to fill the next cell in the light-green table, we add that cell to the list accumulating in the dark-green table. This version has moved the code for initializing the arrays and populating them into helper functions. The main reason for that change is to align with the homework format, which was designed to show you a structuring technique that would also work in Java (where these methods would get called within the constructor).

2 Summary

What should you take from the DP segment?

- If you have a piece of functional (no mutation) code that makes the same call on the same inputs multiple times, you can save time by storing the previously-computed results in a data structure and retrieving them later.
- Search-based problems often satisfy this pattern. In search-based problems, we are looking to find (a) a solution, that (b) optimizes for some attribute of the data. In these notes, we have shown you how to make data structures to hold the results of both the optimized attribute and the solution paths as you run the program.
- A program modified to use dynamic programming will run only once on each unique input value. Dynamic programming saves on time by using more space.
- We used arrays as the data structure for previously-computed inputs here (rather than, say, hashtables) partly because you will often see DP problems solved with arrays (and we want you to be prepared for interviews). You could have used different data structures in practice.
- Starting from a working recursive solution without optimization can ease the process of writing the optimized solution.

3 Study Questions

1. Why did we say that DP only works if the original recursive program is functional (rather than mutating data)?
2. If you do write both the unoptimized and the optimized solutions, how might you use both in testing?

```
class HalloweenLtoFPaths(candyAvailable: Array[Array[Int]]) {
  val height = candyAvailable.length - 1
  val width = candyAvailable(0).length - 1
  val candyComputed = Array.ofDim[Int](width + 1, height + 1)
  // make lists of tuple of coordinates to store paths
  val bestPath = Array.ofDim[List[(Int, Int)]](width + 1, height + 1)

  // suppressed in this version for space
  def candyVal(row: Int, col: Int): Int = { ... }

  // a new helper to return which of col-1, col, and col+1 has
  // highest value in row -- use this to update both tables
  def bestCol(row: Int, col: Int): Int = {
    if (candyVal(row, col - 1) > candyVal(row, col))
      if (candyVal(row, col - 1) > candyVal(row, col + 1))
        col - 1
      else col + 1
    else if (candyVal(row, col) > candyVal(row, col + 1))
      col
    else col + 1
  }

  def initTable() = {
    // set up the base cases of the computed values
    for (col <- 0 to width) {
      candyComputed(height)(col) = candyAvailable(height)(col)
      bestPath(height)(col) = List((height, col))
    }
  }

  def fillTable() = {
    // set up the rest of the computed values
    for (row <- height - 1 to 0 by -1) {
      for (col <- 0 to width) {
        val maxCol = bestCol(row + 1, col)
        candyComputed(row)(col) =
            candyAvailable(row)(col) + candyVal(row + 1, maxCol)
        bestPath(row)(col) = (row, col) :: bestPath(row + 1)(maxCol)
      }
    }
  }

  // run the setup methods
  // in Scala, these don't really need to be methods. However, the homework
  // handout asks for them because this is how you would set up such a
  // problem in Java -- these methods would get called in the constructor
  initTable()
  fillTable()
}
```

Figure 3: Most of the solution that also computes paths.

```scala
// get the largest candy total from the top row of the candyComputed table
def maxCandy(): Int = {
  List.range(0, width).map(c => candyComputed(0)(c)).max
}

// get the path that leads to the max candy
def maxPath(): List[(Int, Int)] = {
  val indices = List.range(0, width)
  val topRowVals = indices.map(c => candyComputed(0)(c))
  val indexOfMax = topRowVals.indexOf(topRowVals.max)
  bestPath(0)(indexOfMax)
}
```

Figure 4: The rest of the paths-based solution .

3. DP seems to introduce a sizeable space overhead, especially if we were to be searching in large data. Think about our final Halloween solution, and the patterns of array cell access that get used in the final solution. Do the patterns suggest ways that we could use less space than our current approach (which needs two arrays, each the same dimensions as the input neighborhood).

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.