UNIVERSITY OF PITTSBURGH
CS 1501: ALGORITHM IMPLEMENTATION

Dynamically Cutting Cloths

Zach Sadler

April 22, 2013

1 PROBLEM DESCRIPTION

The *Cloth Cutting Problem* is an optimization problem with real-world applications:

> A textile company has a rectangular sheet of cloth with dimensions *width* x *height*.
> The company also has a set of patterns, **P**, each $\mathbf{P}_i$ with its own width, height, and value (m_i, n_i, and v_i respectively).
> The company can only make full length cuts on a given piece of cloth- at increments $1 \leq i \leq k$, where k is either the cloth's full width or full height- so that each cut creates two rectangular subcloths.
> Which sequence of cuts should the company make to maximize the total value of patterns created?

Obviously, the problem is trivial if there are no patterns or the cloth is of non-positive area, so assume that the width and height are positive for the cloth and each pattern, and that each pattern has a positive value. One more important detail is that neither the cloths nor the patterns may be rotated. For example, if one of the patterns is of size 2 x 5, one may not make a pattern of size 5 x 2 and attempt to sell it for the same value (or indeed, any value at all).

1.1 EXAMPLE

Let **P** be a set of patterns, as explained above. Let *Subproblem*(subWidth, subHeight) be the maximum total value for a cloth of size subWidth x subHeight. Now we can give a concrete example.

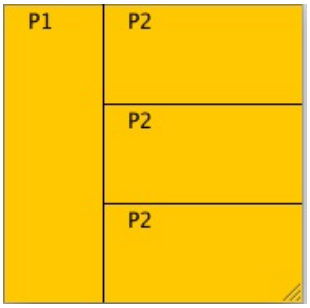

Figure 1.1: The optimal cutting for the example problem given below

Let $\mathbf{P}_1$ be a 2 x 6 piece of cloth with value 4, let $\mathbf{P}_2$ be a 4 x 2 piece of cloth with value 3, and $\mathbf{P}_3$ be a 5 x 3 piece of cloth with value 5. If the initial cloth is of size 6 x 6, then the optimal value is created by a vertical cut at $x = 2$, and three horizontal cuts, at $y = 2$, $y = 4$, and $y = 6$, resulting in $v_1 + 3 \cdot v_3 = 13$ value (see Figure 1.1 for a visual solution).

It is easy to see that this is a non-trivial problem, even for small amounts of patterns and a small total cloth size. For example, simply increasing the width by 1 and keeping the same set of patterns in the previous problem yields a much different result, as shown in Figure 1.2.

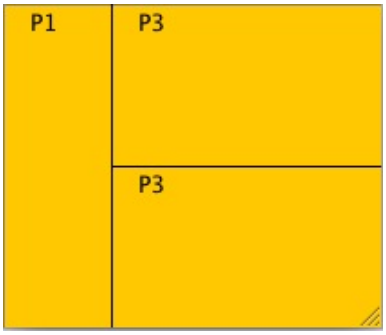

Figure 1.2: A very different solution, despite very little change

2 Naive Recursive Approach

Clearly, if such a small change can alter the result even with just three patterns, it's apparent that one will need to search exhaustively to find the solution. In the naive approach, one might think that knowing the previous problem will not impact the result of the current problem, as we saw in Figure 1.1 and 1.2. So, one would implement a top-down recursive approach as follows:

2.1 Code Listing - Naive

```
Subproblem(subWidth, subHeight) {
  patternMax, horizontalMax, verticalMax = 0
  for (i = 1... numberOfPatterns)
    if (patterns_i.value ≥ patternMax)
      patternsMax = patterns_i.value

  for (i = 1... subWidth) {
    temp = Subproblem(subWidth, i) + Subproblem(subWidth, subHeight − i)
    if (temp ≥ horizontalMax) { horizontalMax = temp }
  }
  for (i = 1... subHeight) {
    temp = Subproblem(i, subHeight) + Subproblem(subWidth − i, subHeight)
    if (temp ≥ verticalMax) { verticalMax = temp }
  }
  return max(patternMax, horizontalMax, verticalMax)
}
```

2.2 Analysis of Naive Recursive Algorithm

The basic idea of the code listed above is to start with the current cloth, try each pattern on it and save the maximum value, then try each cut (recursively) both horizontally and vertically (saving the resulting maximum of each pair of subcloths), and return the maximum of these three values.

This algorithm has major inefficiencies. Let's say that *Subproblem*(m, n) took δ recursive calls to solve. Then to solve Δ = *Subproblem*(m+1, n), we can see that along the way I will make a vertical cut at the first increment, leaving a cloth of size of m x n and another of size 1 x n. We already know that the cloth of size m x n will take δ recursive calls, and the cloth of size 1 x n will also require non-trivial work, and so naturally $\Delta > \delta$.

However, we will also have to perform a vertical cut at increment m, yielding a cloth of size m x n and another of size 1 x n. So once again we will have to perform more than δ recursive calls, which means that $\Delta > 2 \cdot \delta$, or in other words, $T(n) = 2 \cdot T(n-1) + 2 \cdot \theta(n)$, (where $\theta(n)$ accounts for the calls required for a cloth of size 1 x n) which means this algorithm is exponential.

Later in this paper we will see experimentation that confirms this analysis. Also, note that this analysis only took into account the adjustment of a single variable (either width or height- not both), increasing both values would result in an even worse case scenario, as *Subproblem*(m+1, n+1) requires more than four times the recursive calls of *Subproblem*(m, n).

3 Dynamic Bottom-up Approach

Our naive algorithm has some serious issues stemming from the fact that it repeats so many of the same subproblems. Consider if, instead of starting from the largest problem and continuously breaking it down into the same sized subproblems again and again, we reversed

our thought process. If we start from the smallest size and work our way up, we can keep track of the results of these smaller subproblems and then we can just look up the answer later instead of recalculating each time. Instead of solving the same subproblem (like 1x1) for every single larger subcloth, we would only have to solve it once and then remember the value,

This type of approach would dramatically reduce the number of recursive calls needed, so surely it must require some dramatic changes in our code, right?

3.1 Code Listing 2 - Dynamic

In order to make our program dynamic, we must make the following changes:

```
// stores the results of subproblem a x b in memos[a][b]
int [][] memos = new int[width + 1][height + 1]

// starts from the bottom, storing memos as we find answers to subproblems
Optimize() {
  // first initialize the memos to -1 as a flag to know we haven't been there yet
  for (i = 1... width)
    for (j = 1... height)
      memos[i][j] = -1

  // then start solving the subproblems and overwrite the flags with answers
  for (i = 1... width)
    for (j = 1... height)
      memos[i][j] = Subvalue(i, j)
}

Subvalue(subWidth, subHeight) {
  if (memos[subWidth][subHeight] > -1) {
    return memos[subWidth][subHeight]
  }
  ...
  // code from Subproblem before
  ...
}
```

3.2 Analysis of Dynamic Bottom-up Algorithm

Wait, what? That's all the changes we make? It hardly seems like anything at all, so let's see if the runtime changed to any noticeable degree.

As before, let's say that *Subvalue*(m, n) took δ recursive calls. Then to solve Δ = *Subvalue*(m + 1, n), we can see that we will take δ calls to solve up to a m x n subcloth. From there, if we

make any vertical cut on the cloth of size (m + 1) x n then we will be at a previously solved subproblem, and finished after a constant-time lookup from the memos. Thus we just have to solve (from the bottom up), the cloths of size (m + 1) x i, for $1 \leq i \leq n$, which will take θ(n) because of memoization. So our recurrence relation is $T(n) = T(n-1) + c \cdot \theta(n)$, (for some constant c) which is quadratic.

As another way of looking at it, just consider that we call Subvalue() $n \cdot m$ times, and while there is some additional recursion going on within those calls, with the memoization happening at constant-speed, the 'extra' recursion becomes negligible and the full algorithm takes $\Theta(n^2)$ time.

3.3 Technical Issues

The code listing provided is almost exactly what I have in my submitted project (which received full credit), and so it really is as easy as it sounds. However, one might have noticed that this solution only gives the *value* of the optimal cloth cutting sequence; not the sequence of cuts itself.

In order to track the proper sequence of cuts, a little more bookkeeping is required. I have a class called 'Cut' which stores information about the best cuts which were found at each subproblem. 'Cut' tracks whether it was vertical or horizontal, the size of the cloth it cut, the position it cut (the increment from 1 to the width or height), and the absolute offset of the subcloth compared to the original, full cloth. When I return the maximum value of *patternMax*, *horizontalMax*, and *verticalMax*, if I made a horizontal or vertical cut I store the information about the current subproblem to an ArrayList of Cuts. Remember that since we only solve each subproblem once, this means that there will be a *unique* cut for each $(x, y) \in width \times height$.

Then, after I've solved all the subproblems and filled out my 2D array of memos, I do one final run from top to bottom. I retreive the cut corresponding to the largest suproblem, then using the information stored inside it I retrieve the next cut recursively (by adjusting my current subWidth and subHeight by where I made the cut) and keep track of my absolute offset from the origin, to be able to display it later graphically. As an example, if the cut on a subcloth of size 20 x 20 is a vertical cut with 'position' = 3 then I will recursively retrieve the cut for subcloth size 17 x 20 (which then has absolute offset $(3,0)$), and after backtracing, recursively retrieve the cut for subcloth size 3 x 20 (which has absolute offset $(0,0)$).

I recognize that this final run does cause some overhead, but it is very minimal compared to the *Subvalue* method, which is itself very minimal compared to the naive approach.

4 Results from Experimentation

I adjusted my program so that I could turn memoization on and off, so that I could compare the number of recursive calls made when memoization was enabled versus disabled. The results were **truly staggering**, and confirmed my analyses of the runtimes:

These results were calculated while running the program with the same dataset (four patterns: $\mathbf{P}_1$ a 2 x 2 with value 1; $\mathbf{P}_2$ a 2 x 6 with value 4; and $\mathbf{P}_3$ a 4 x2 with value 3; $\mathbf{P}_4$ a 5 x 3

Table 4.1: Results from a Small Set of Experiments

Item	2 x 5	3 x 5	4 x 5	6 x 5	8 x 5	10 x 5
Memoized	60	105	160	300	480	700
Naive	1,052	6,573	35,224	786,548	14,420,144	234,156,620

with value 5). I ran each test three times to confirm the results, although there is no randomness to this deterministic algorithm, so there is no need for uncertainty about the number of recursive calls required.

These are the values (the maximum amount of money to be earned from the cloth) for each size:

Table 4.2: The Maximum Value to be Gained

Capacity	2 x 5	3 x 5	4 x 5	6 x 5	8 x 5	10 x 5
Value	2	2	6	9	12	17

4.1 Memoized Experiment Analysis

Using the data provided in Table 4.1, I found a quadratic formula that matches the data points for the memoized algorithm (R and R^2 values of 1):

$$5x^2 + 20x + 6.818 \cdot 10^{-14}$$

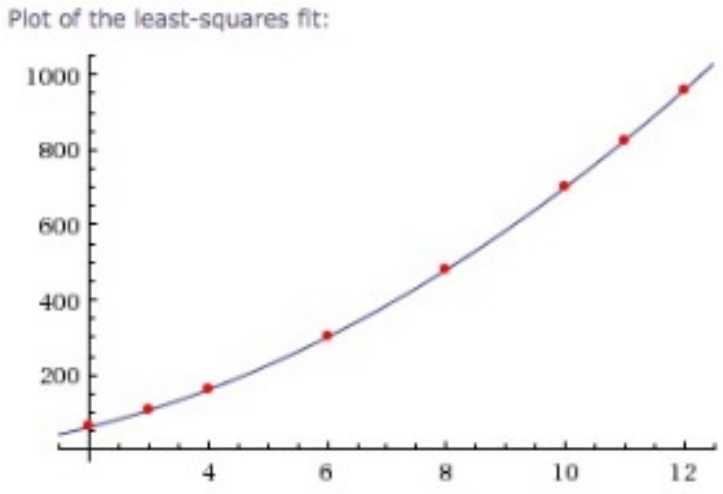

Figure 4.1: The curve of $5x^2 + 20x + 6.818 \cdot 10^{-14}$ intersecting each data point

Although this is a relatively small data set it is evident that the memoized approach has a quadratic runtime.

Similarly, there is an exponential function that matches the data points for the naive algorithm (again, R and R^2 values of 1):

$$206.505\mathbf{e}^{1.39412x}$$

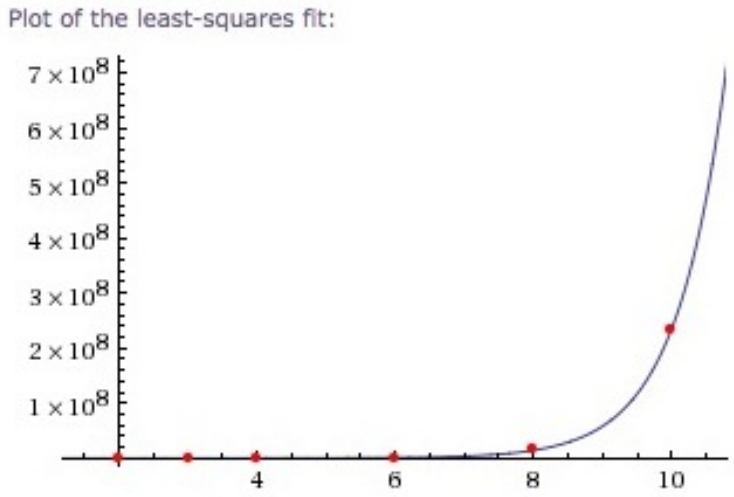

Figure 4.2: The curve of $206.505\mathbf{e}^{1.39412x}$ intersecting each data point

This is an obviously exponential function, if the data points were not enough of a clue to begin with.

5 A QUICK ASIDE

In case the graphs and the small experiment still did not give you a scale of the enormous numbers in play with the exponential function, consider the following:

$$206.505\mathbf{e}^{1.39412\cdot\mathbf{20}} = 2.66\cdot 10^{14}$$

that means that if we simply double the width of the cloth used in the final experiment (from 10 x 5 to 20 x 5), we get a number so large that if it took a **millisecond** to do each recursive call, then it would still take about eight and a half thousand years to compute the answer.

This is why it's important to find an efficient algorithm, because with the dynamic approach:

$$5\cdot\mathbf{20}^2 + 20\cdot\mathbf{20} + 6.818\cdot 10^{-14} = 2400$$

which means that if each recursive call took **3 years** the dynamic algorithm would *still* beat the naive approach.

The carpenter says to measure twice and cut once. The computer scientist says to make a program that can measure 2400 times in an instant, cut 8 times, and maximize your profit.