Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

The Maximal Rectangle Problem

Source Code Accompanies This Article. Download It Now.

- rectang.txt

The stepwise derivation of the algorithm David presents here illustrates a number of principles that are widely applicable in algorithm design.

By David Vandevoorde
April 01, 1998
URL:http://drdobbs.com/database/184410529

David, a software design engineer for Hewlett-Packard, is one of the founders and moderators of the comp.lang.c++.moderated Usenet forum and represents HP on the ANSI X3J16 C++ committee. David can be contacted at daveed@ vandevoorde.com.

The most enjoyable games often have the most straightforward rules. Similarly, some of the most fascinating problems can be stated most simply and most succinctly. I'll let you be the judge of whether the problem presented here is interesting or not, but you'll agree that it's easily described:

- *Given*: A two-dimensional array *b* (*M* rows, *N* columns) of Boolean values ("0" and "1").
- *Required*: Find the largest (most elements) rectangular subarray containing all ones.

Figure 1 illustrates a possible input array and the corresponding solution. (I'll often refer to a rectangular subarray as simply a "rectangle.")

Brute Force

Clearly, there are a finite number of distinct subarrays in the given array *b*. Hence, you could enumerate each of these subrectangles and test whether they uniformly consist of ones. Your intuition would be correct in rejecting such a solution for being too expensive, but for my purposes here, this brute force approach makes a nice baseline. It also provides me with an excuse to make a point later on.

Listing One (listings being on page 100) is pseudocode for this first algorithm. It enumerates all the subarrays of *b* by picking each element of *b* in turn as a possible lower-left corner *ll* of the final solution, and for each *ll* trying out every possible upper-right corner *ur*. Once *ll* and *ur* are chosen, the program tests whether or not the rectangle consists of all ones and whether or not that rectangle is larger than the best candidate solution you have found so far. Since the latter test is simpler, you can gain some time by doing it first: There is no need to scan a rectangle that is no larger than the best candidate subarray found so far. The helper function *area* computes the number of elements in a

rectangle defined by its lower-left and upper-right corner. The function *all_ones* returns True only if the given rectangle contains no zeros.

How expensive is this algorithm? It probably depends on the array you feed it. However, I'll limit myself to estimate the worst-case asymptotic complexity of the algorithms presented in this article. If you're not familiar with that sort of analysis or with the *O*-notation, you might want to refer to an algorithms book such as *Introduction to Algorithms*, by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest (McGraw-Hill, 1990). (In particular, Chapter 2 covers *O*-notation.)

Clearly there are *M*×*N* possible choices for *ll.* Each of these choices has between one and *M*×*N* corresponding upper-right corners: That means on average there will be *O(M*×*N) ur* choices for every *ll.* Hence the total number of subrectangles of *b* is of the order *O(M*^{2}×*N*^{2}*)*. The average number of elements in these subarrays is also *O(M*×*N),* and even though the algorithm only needs to scan a fraction of these subarrays, you can construct cases where the average number of operations performed per subarray of *b* is *O(M*×*N).* Hence, the worst case complexity of this algorithm is *O(M*^{3}×*N*^{3}*)*, which is not very good.

This algorithm requires very little extra storage other than the input array itself. Some of the faster algorithms I will present will not maintain that property (but I promise it won't be too bad).

Adding Direction

Though inefficient, the first algorithm provides a good basis upon which to build. To improve its performance, I'll add some direction to the search. Indeed, the algorithm in Listing One could enumerate the subrectangles in any random order and still find the correct solution. Instead, I now want to take advantage of the fact that if a small rectangle contains a zero, so will each of its surrounding rectangles. Therefore, I will grow rectangles for each possible lower-left corner. This growing process will only produce upper-right corners defining rectangles which contain all ones: The *all_ones* function of Listing One will no longer be necessary.

The resulting algorithm is listed in Listing Two, and the function *grow_ones* is at its heart. The first time through its outer *while* loop, this function scans to the right of the given lower-left corner *ll* for a chain of consecutive ones that is as long as possible. It remembers the right edge of that scan in *x_max*: When it tries to add a layer of ones in the next iteration of the outer loop, it will not attempt to scan a wider rectangle than the first one (since that would be pointless). Figure 2 shows the consecutive layers that are grown at the optimal lower-left point in Figure 1.

The *grow_ones* function will never examine an element of the array *b* more than once. However, for an array consisting of all ones, it will scan all the elements above and right of the given position *ll.* Consequently, this function's complexity is *O(M*×*N).* Since the main algorithm invokes *grow_ones* for every one of the elements in *b*, the total complexity of this approach is *O(M*^{2}×*N*^{2}*).*

Not good enough.

At this stage, the algorithm still does not require any significant amount of extra memory.

Remember!

If you play the role of a computer executing the algorithm in Listing Two on a sample array, you may notice that the *grow_ones* function often scans and rescans the same strip of ones. For example, in Figure 2, the element designated *ll* starts a strip of three ones. Some time after establishing this, the algorithm calls *grow_ ones* on the element *p* one position to the right on *ll.* Had the algorithm remembered that *ll* has three ones to its right, it would easily deduce -- without a loop -- that *p* is the left end of a sequence of two ones.

To make the algorithm remember the number of ones to the right of each element in a column of the array *b*, I introduce a cache *c*. Figure 3 shows consecutive cache contents near the solution rectangle. Updating this cache from one column to the next is not hard, but the initial values in the cache are easiest to set up if you start with the right-most column of *b*: In that case, you can start with a cache

consisting of all zeros. This explains why I changed the direction of the outermost loop in Listing Three. The cache is updated once per column by the procedure *update_cache*: the cost of one call to this procedure. However, it allows me to eliminate the inner loop of the *grow_ones* function, which, therefore, also reduces the cost per invocation. Combining all this, you find that this latest version of the solution costs $O(M^2 \times N)$.

This gets close to being acceptable. For example, many interesting numerical matrix algorithms have this sort of complexity. I'll now present an even better algorithm. As I hinted earlier, we are now in a situation where we need a significant amount of extra memory -- the cache -- to execute the algorithm. However, this also comes with a second advantage besides efficiency: Each element of the array *b* needs to be read only once!

Let me make one more observation about this incarnation of the solution: Although the problem statement is symmetric with respect to rows and columns, the complexity of the algorithm in Listing Three is not. Indeed, if *M* and *N* are significantly different, my earlier analysis predicts that it is better to look at the array *b* in landscape mode $(M<N)$ than in portrait mode $(M>N)$. You may therefore wonder whether the caching of horizontal scans could be combined with another cache for the vertical loop. Unfortunately, it's not that easy. There are a few algorithms that easily generalize from one dimension to two or more -- the Fast Fourier Transform (FFT) among them -- but they are the exception rather than the rule. To optimize the vertical loop, you need to take better advantage of the properties of the problem.

Exploiting Structure

Caching or remembering intermediate results is a general technique that applies to a wide variety of algorithms. It's also a technique that can often easily be integrated into existing algorithms. Ultimately, though, most optimal algorithms depend on the more unique features of a problem, and the maximal rectangle problem is no exception.

To find some additional structure in the problem, consider Figure 4, which depicts the cache *c* as a sort of profile of ones to the right of the current column. The previous algorithm examined every rectangle of ones that fits in that profile. However, some of these rectangles aren't as worth examining as others. For example, the dashed rectangle in Figure 4 is inside the profile described by the current contents of the cache, but there are larger rectangles in the profile which enclose the dashed one. My goal then is to examine only those rectangles that are inside the profile, but not contained by another rectangle in that profile. These rectangles were drawn in a slightly thicker line in the left half of Figure 4; the solution rectangle (in red) is one of these interesting rectangles.

Arrows to the left of the profile in Figure 4 depict the vertical extent of the interesting rectangles. The crucial observation to make here is that these extents nest -- if you traverse the arrow ends from bottom to top, you'll find that an arrow that started earlier than another arrow will finish no later that other arrow. You will also notice that arrows start when the profile widens, while they end when the profile narrows. So, opening one or more nested rectangles corresponds to seeing an increase in the cache value, while closing rectangles corresponds to seeing a decrease; if the cache does not change in value, no new rectangles are opened and no opened ones are closed.

If you are familiar with parsing programming languages with nested structures, you know that a stack is the natural data structure to handle the opening and closing of nesting levels. The stack I use in my final algorithm keeps track of where a profile widening was seen, and what the width prior to that widening was. This allows the algorithm to recover the width of already opened narrower rectangles when closing a wider one. Listing Four lists the details of the algorithm.

When the value of the cache reduces as the *y* coordinate increases, it is possible that multiple rectangles must be closed. That explains the innermost *do* loop. Each time a rectangle is closed, it is compared against the best rectangle so far and stored away if it is better. Because the *do* loop may pop one too many openings, it is followed by pseudocode that pushes back the last popped pair.

The cost of a single iteration of the *y* loop can be quite expensive due to the embedded *do* loop. However, each iteration of the *y* loop will never push more than one pair onto the stack. Since each iteration of the *do* loop will pop one such pair, the total number of times it will be executed for a

single *x* value cannot be more than *M.* Hence the innermost *for* loop has complexity *O(M).* Because *update_cache* has the same complexity and because both *update_cache* and the inner *for* loop are executed *N* times -- by the outer *for* loop -- the total complexity of this algorithm is *O(M*×*N).* The sort of thinking that went into the determination of the complexity of the inner *for* loop is sometimes called "amortized analysis" because the cost of the *do* loop is amortized over the number of push operations; Cormen et al. has more examples of this technique.

That complexity is optimal, because reading the contents of each element of the array requires at least *O(M*×*N)* operations (and if you miss reading even one element you might not find the largest rectangle). This algorithm still requires *O(M)* memory and still works when the elements of the given array *b* are given as an input stream.

Game Over

The stepwise derivation of this algorithm illustrates a number of principles that are widely applicable in algorithm design. In this case, each step provided an asymptotic improvement for the worst case, and resulted -- in my opinion -- in a more elegant algorithm. That is not always the case. Sometimes, a clever refinement will only provide a nonasymptotic improvement, but that, too, is often worthwhile. Furthermore, it is often the case that such refinements make the algorithm considerably less elegant and less concise.

A problem similar to that presented here was rather seriously studied in the continuous domain. That problem consists in finding the largest empty rectangle inside a given rectangle filled with points. An efficient algorithm for this problem builds on the concept of tents, which have some similarity to the cache/profile used in our solution.

Acknowledgment

Many thanks to Saul Rosenberg, Tom Halladay, and James Curran for making me dust off this algorithm to present it at an Object Development Group (http://www .objdev.org/) meeting in New York. This article is dedicated to Susan Rodger who first showed this problem to me in a most outstanding algorithms course at the Rensselaer Polytechnic Institute. David Francis and Tara Krishnaswamy contributed many corrections and ideas to this article.

References

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* New York: McGraw-Hill, 1990.

Orlowski, M. "A New Algorithm for the Largest Empty Rectangle Problem," *Algorithmica*, 1990, volume 5, 65-73.

Stroustrup, Bjarne. *The C++ Programming Language.* Third edition. Reading, MA: Addison-Wesley, 1997.

DDJ

Listing One

```
// Variables to keep track of the best rectangle so far: best_11 = (0, 0); best_ur = (-1, -1)
main algorithm:
   for 11.x = 0 .. N
      for 11.y = 0 .. M
         for ur.x = 11.x .. N
            for ur.y = 11.y .. M
               if area(11, ur)>area(best_11, best_ur)
               and all_ones(11, ur)
                  best_11 = 11; best_ur = ur
end main algorithm
define area(11, ur)
   if 11.x>ur.x or 11.y>ur.y // If ur is left of or
      return 0               // below 11: return 0
   else
```

```
      return (ur.x-ll.x+1)*(ur.y-ll.y+1)
define all_ones(ll, ur)
   for x = ll.x .. ur.x
      for y = ll.y .. ur.y
         if b[x, y]==0
            return false
   return true
```

Back to Article

Listing Two

```
// Variables to keep track of the best rectangle so far: best_ll = (0, 0); best_ur = (-1, -1)
main algorithm:
   for ll.x = 0 .. N-1
      for ll.y = 0 .. M-1
          ur = grow_ones(ll)
          if area(ll, ur)>area(best_ll, best_ur)
             best_ll = ll; best_ur = ur
end main algorithm
define grow_ones(ll)
   ur = (ll.x-1, ll.y-1) // Zero area ur-choice
   x_max = N // Right edge of growth zone
   y = ll.y-1
   while y+1<M and b[ll.x, y+1]!=0
      y = y+1; x = ll.x // Scan a new layer
      while x+1<x __ max and b[x+1, y]!=0
         x = x+1
      x_max = x
      if area(ll, (x, y))>area(ll, ur)
         ur = (x, y)
   return ur
```

Back to Article

Listing Three

```
// Variables to keep track of the best rectangle so far: best_ll = (0, 0); best_ur = (-1, -1)
// The cache starts with all zeros:
c[0 .. M-1] = 0
main algorithm:
   for ll.x = N-1 .. 0
      update_cache(ll.x)
      for ll.y = 0 .. M-1
          ur = grow_ones(ll)
          if area(ll, ur)>area(best_ll, best_ur)
             best_ll = ll; best_ur = ur
end main algorithm
define update_cache(x)
   for y = 0 .. M-1
      if b[x, y]!=0
         c[y] = c[y]+1
      else
         c[y] = 0
define grow_ones(ll)
   ur = (ll.x-1, ll.y-1) // Zero area ur-choice
   y = ll.y-1
   while y+1<M and b[y]!=0
      y = y+1; // Scan a new layer
      x = min(ll.x+c[y]-1, x_max) // Use the cache, Luke!
      x_max = x
      if area(ll, (x, y))>area(ll, ur)
         ur = (x, y)
   return ur
```

Back to Article

Listing Four

```
// Variables to keep track of the best rectangle so far: best_ll = (0, 0; best_ur = (-1, -1)
// The cache starts with all zeros:
c[0 .. M-1] = 0 // One extra element (closes all rectangles)
```

```
c[0 .. M-1] = 0 // One extra element (closes all rectangles)
main algorithm:
   for x = N-1 .. 0
      update_cache(x)
      width = 0 // Width of widest opened rectangle
      for y = 0 .. M
         if c[y]>width // Opening new rectangle(s)?
            push(y, width)
            width = c[y]
         if c[y]<width // Closing rectangle(s)?
            do
               (y0, w0)= pop()
               if width*(y-y0)>area(best_ll, best_ur)
                  best_ll = (x, y0)
                  best_ur = (x+width-1, y-1)
               width = w0
            until c[y]>=width
            width = c[y]
            if width!=0 // Popped an active "opening"?
               push(y0, width)
                        w0
```

Back to Article

The Maximal Rectangle Problem

By David Vandevoorde

Dr. Dobb's Journal April 1998

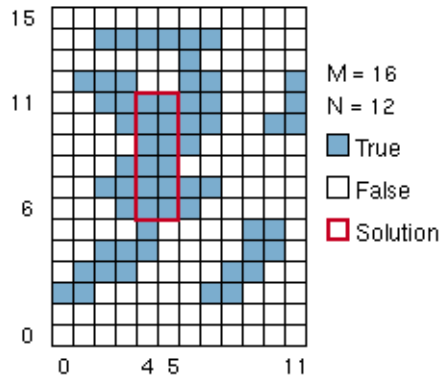

Figure 1: Possible input array and corresponding solution.

Copyright © 1998, Dr. Dobb's Journal

The Maximal Rectangle Problem

By David Vandevoorde

Dr. Dobb's Journal April 1998

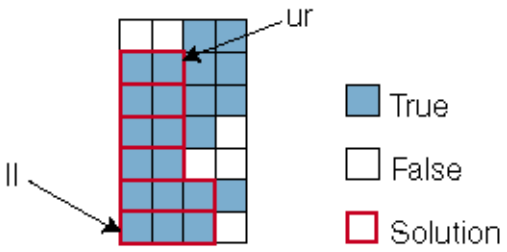

Figure 2: Consecutive layers that are "grown" at the optimal lower-left point in Figure 1.

The Maximal Rectangle Problem

By David Vandevoorde

Dr. Dobb's Journal April 1998

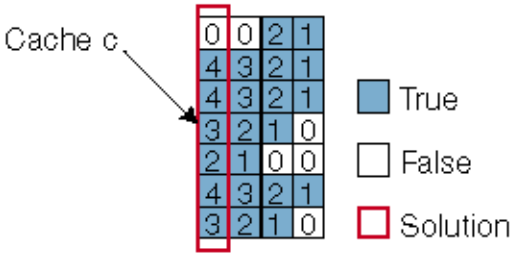

Figure 3: Consecutive cache contents near the solution rectangle.

The Maximal Rectangle Problem

By David Vandevoorde

Dr. Dobb's Journal April 1998

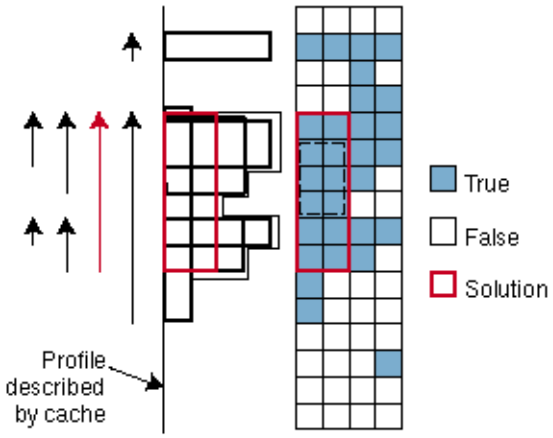

Figure 4: Cache c as a sort of profile of ones to the right of the current column.