

CSc 445: Homework 5 Solutions

April 14, 2008

1 Properties of solution to the matrix chain problem

a Prove that any parenthesization of n elements has exactly $n - 1$ pairs of parenthesis.

Solution The proof is by induction

Base case: Let $n=2$. The only way two elements A_1 and A_2 can be parenthesized is (A_1A_2) . Hence there is only one $(n-1)$ parenthesis pair.

Induction Hypothesis: Let the proposition be true for $n = k$. Hence, $k - 1$ pairs of parenthesis are required to parenthesize k elements.

Weak Induction: Now, consider $(k+1)$ elements A_1, A_2, \dots, A_{k+1} . Let these $k+1$ elements be parenthesized as $((A_1, A_2, \dots, A_j)(A_{j+1}, A_{j+2}, \dots, A_{k+1}))$ for some $j \neq k$. The two partitions consist of j and $(k + 1 - j)$ elements respectively and contain fewer than k elements. Hence, from the induction hypothesis, they require $j - 1$ and $k - j$ pairs of parenthesis respectively. Hence, the total number of pairs of parenthesis required for the above parenthesization is $j - 1 + k - j + 1 = k$. The 1 is for the combining the two sub problems. Hence, the proof.

b Count the number of times a given table entry at $m[i, j]$ is referenced during an execution of the `MatrixChainOrder`.

Solution Looking at the nested for loops in the `MatrixChainOrder` given in CLRS, we can write the following summation:

The outer-most loop on l runs from 2 to n ,

or,

$$\sum_{l=2}^n 1$$

The next loop runs i from 1 to $n - l + 1$,

$$\sum_{l=2}^n \sum_{i=1}^{n-l+1} 1$$

The inner loop runs k from i to $j - 1$, but it also references two elements of m other than $m[i, j]$:

$m[i, k]$ and $m[k + 1, j]$. So, we add 2 to our total for each iteration of this loop.

$$\sum_{l=2}^n \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 2$$

This can be unrolled to include only l and n ,

$$-2\left(\sum_{l=2}^n l^2\right) + 2\left(\sum_{l=2}^n l(n+2)\right) - 2\left(\sum_{l=2}^n n+1\right)$$

or,

$$-2\left(\sum_{l=2}^n l^2\right) + 2(n+2)\left(\sum_{l=2}^n l\right) - 2(n^2 - 1)$$

We now use the appendix A.1 and A.3 to make this as a sum running from 1 and 0 respectively.

Altering the indices, we get

$$-2\left(\sum_{l=0}^n l^2\right) + 2 + (2n+4)\left(\sum_{l=1}^n l\right) - 2n - 4 - 2n^2 + 2$$

Plugging in the respective closed forms for the summation, we get

$$-\frac{n(n+1)(2n+1)}{3} + n(n+1)(n+2) - 2n^2 - 2n$$

Now, if we expand everything and sum up the like terms we will get $\frac{n^3-n}{3}$ as required.

2 Appalachian trail problem.

Solution First question to ask is about the optimal substructure. How does an optimal solution look like?

Like I mentioned in the previous homework's solution about VHS tapes, there are two common types of problems one that requires us to pick a permutation and one type that asks us to select a subset. This problem is a typical example of the subset selection.

So, back to examining the optimal solution \mathcal{O}^* . We do know one thing, it may or may not contain the last shelter s_n . Therefore, there are two cases.

If \mathcal{O}^* does not contain the last shelter, then the solution to the problem is the same as the one for the first $n - 1$ shelters .i.e.,

$$A(n) = A(n - 1)$$

If \mathcal{O}^* does contain the last shelter, then we have to cast off the shelters that are within the 10 mile mark from the n^{th} shelter. If we have a function f , that returns the next valid shelter,

we can solve the subproblem upto this shelter .i.e.,

$$A(n) = v_n + A(f(n))$$

This leads us to the familiar cut paste argument of why the above two cases we explored ought to exhibit optimal substructure .i.e., If the subproblem is not optimal, then replace with the better one and get a contradiction to the current problem etc.

We are ready to define the recursion now:

$$A(n) = \max\{A(n - 1), v_n + A(f(n))\}$$

To convert this into an algorithm, we first observe that we have to come up with a look up table for the $f(i)$ values for the shelters on the trail.

Since we are given the shelters along the trail in order, we have a sorted list that we scan through filling up the $f(i)$ entries with the last valid entry we saw. This clearly takes $\mathcal{O}(n)$.

Next, we initialize a storage structure, an array for storing the $A()$ values: $A(0) = 0$ and $A(1) = v_1$.

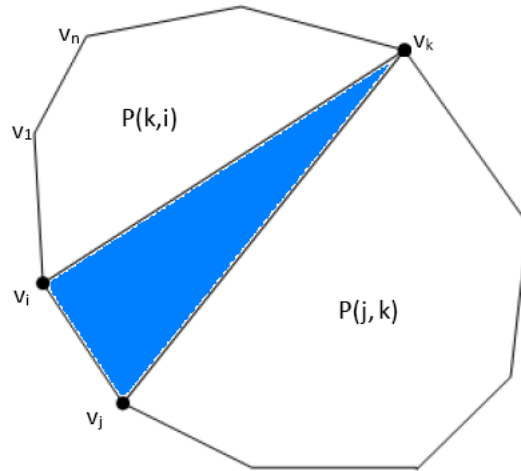
Looping from shelter 2 to n , we build up the $A()$ values using the recurrence. At each iteration, we do a constant time work for looking up $A(i - 1)$, $f(i)$ and $A(f(i))$, all of which are precomputed. Essentially, we use the lookup table to fill out our current entry and the table itself is filled from left to right.

So, our total time overall is $\mathcal{O}(n)$.

3 Triangulation using dynamic programming.

Solution a First we assume that the given polygon is a simple convex polygon. So, no holes in the polygon and it is convex. It is not too hard to solve with this restriction either but let's not get into that here.

Next, observe that any edge in polygon is in some triangle of the triangulation. Also, no two diagonals cut each other. With that in mind, we notice that triangulation of $\mathcal{P} : \{v_1, v_2, \dots, v_n\}$ with vertices named in counter-clockwise order reduces to choosing a certain permutation of diagonals.



Optimal substructure: Now let's ask the question what does the optimal solution look like. Suppose an oracle tells us that an triangle of $v_i, v_j, v_k, i < j < k$ exists in the optimal triangulation. Then the problem on polygon \mathcal{P} is split into two sub problems plus the triangle in the middle, $w(\Delta(v_i, v_j, v_k))$. We can now use the cut and paste argument to this. The subproblem $\mathcal{P} : \{v_j, \dots, v_k\}$ be solved using the same recursion and it should be optimal. Otherwise, if there is some other way of solving this, we can as well apply that method to the first problem on $\mathcal{P} : \{v_1, v_2, \dots, v_n\}$ and come up with a better solution than the optimal one.

The above observations lead us to the recursion:

$$P(v_1, \dots, v_n) = \begin{cases} w(\Delta(v_i, v_{i+1}, v_{i+2})) & \text{when } n = 3 \\ \min_{1 \leq i < j < k \leq n} \{P(v_j, \dots, v_k) + P(v_k, \dots, v_i) + w(\Delta(v_i, v_j, v_k))\} & \text{otherwise} \end{cases}$$

The following algorithm runs in $\mathcal{O}(n^3)$:

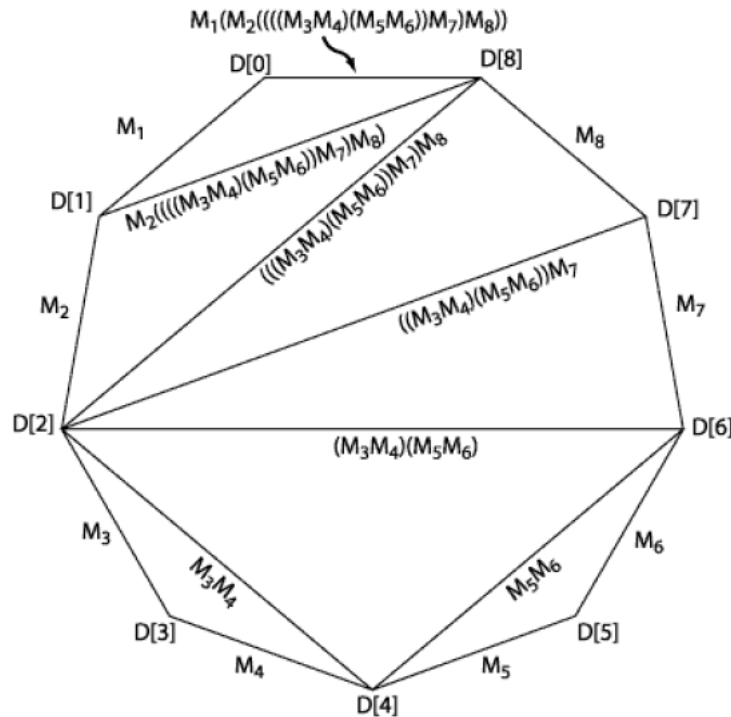
```

Triangulate ( $P[v_1, \dots, v_n], n$ )
for  $1 \leq i \leq n - 1$  do
    for  $i < j \leq n$  do
        if ( $i == j - 1$ ) then
             $\mathcal{P}(v_i, v_j) = 0$ 
        else
             $\mathcal{P}(v_i, v_j) = \infty$ 
            for  $i < k < j$  do
                 $\mathcal{P}(v_i, v_j) = \min\{\mathcal{P}(v_i, v_j), w(\Delta(v_i, v_j, v_k)) + \mathcal{P}(v_i, \dots, v_k) + \mathcal{P}(v_k, \dots, v_j)\}$ 

```

Solution b We can memoize the information in a look up table the same way as done in the MemoizedMatrixChain pseudocode. Literally, we substitute the $p_{i-1}p_k p_j$ with $w(\Delta(v_{i-1}, v_k, v_j))$ in line 6 of the innermost for loop. This also runs in $\mathcal{O}(n^3)$ time.

Note:As a final note for the problem, we can visualize this triangulation as an instance of the matrix chain problem. We rename the edges of the polygon as individual matrices and then look at each subpolygon as a product of all the edges (matrices) that constitute it. The lookup table is used to fill out the entries for the triangulation exactly as is done in the `MatrixChain` problem. The following figure illustrates the idea of visualizing the triangulation as an instance of `MatrixChain`:



4. Buying GOOG Yet Again

Again we are given an array of n daily differentials in stock prices, which we denote $A[1], \dots, A[n]$. We want to find the value of the contiguous subrange with maximum sum, which we may write unambiguously as

$$\max_{1 \leq i \leq j \leq n} \left\{ \sum_{k=i}^j A[k] \right\}.$$

Note that in the above formulation, i and j are both unknown termini of the subrange.

Of course we also wish to use this as an opportunity to improve our dynamic-programming fu (although as mentioned briefly in Homework 3, there is a linear-time divide-and-conquer solution). The first step is to break down the problem, looking for optimal substructure so that we can develop a recursive formulation, with a view to finding overlapping subproblems.

In order to break down the problem, let's define some notation. First, we shall consider how to solve the problem more generally, for an m -element prefix of the array A , $m \leq n$. Let

$v(m)$ denote, for any $1 \leq m \leq n$, the maximum value of any contiguous subrange for in $A[1], \dots, A[m]$. We can express this like so:

$$v(m) := \max_{1 \leq i \leq j \leq m} \left\{ \sum_{k=i}^j A[k] \right\}.$$

Can we somehow compute v recursively? The base cases are that $v(1) = 0$ if $A[1] < 0$, and $v(1) = A[1]$ if $A[1] \geq 0$; that is, if the stock goes up on the first day, then that differential is as much as we can make, whereas if the stock drops, we can make no profit.

Can we relate $v(m)$ somehow to $v(m - 1)$? We need optimal substructure to do so. If the maximum value subrange in $A[1], \dots, A[m]$ does not include day m , then clearly $v(m) = v(m - 1)$. That is, if the best subrange spans days i to j and $j < m$, then obviously the best subrange in days 1 to $m - 1$ can include the same range of days and hence will be no lower; nor can it be greater, for if it were then that same subrange is a part of days 1 to m , which contradicts the notion that we were considering the max subrange in days 1 to m . So optimal substructure holds for computing $v(m)$ provided that day m itself is not in the desired subrange.

Things are more complicated if day m is part of the best subrange. As a concrete example, if $v(m)$ is fifty dollars, and $A[m]$ is ten dollars, it is entirely possible that there is, say, a 45-dollar subrange somewhere earlier in the array, and so the maximum-sum subrange for $A[1], \dots, A[m - 1]$ is not necessarily the max-sum subrange for $A[1], \dots, A[m]$ with element $A[m]$ cut off. In other words, day m might have been a good day for the stock price, and pushed a second-place subrange into first-place.

We need some extra formal notation to describe this situation. Define $w(m)$ to denote the value of the max-sum subrange that ends at day m , i.e.,

$$w(m) := \max_{1 \leq i \leq m} \left\{ \sum_{k=i}^m A[k] \right\}.$$

By pinning the ending day of the subrange to be m , we have only one unknown index i in this expression. Armed with this notation we can describe the optimal substructure for $v(m)$ when day m is part of the subrange. Let R be the max-sum subrange that begins somewhere in days 1 to $m - 1$, but is constrained to end on day $m - 1$. Range R might not be the most valuable subrange, but if we know that day m is part of the optimal solution for days 1 to m , then R must not be too bad; in fact R must be within $A[m]$ of the best, i.e., $v(m - 1)$. For if R were worth less than that, then its value, defined as $w(m - 1)$, plus the value of day m (which is $A[m]$) would be less than “some other subrange” (that of $v(m - 1)$, as we just said), contradicting our assumption that day m is truly part of the max-sum subrange. To conclude, if day m is part of the max-sum subrange, then $v(m) = w(m) = w(m - 1) + A[m]$, which states that the optimal subrange for days 1 to m is the optimal subrange ending at day $m - 1$, plus the change in price on day $A[m]$. This completes our description of optimal substructure.

Thus to compute $v(m)$ we may simply try both possibilities: that day m is, and is not, included in the optimal range. Hence the following recursive formulation:

$$v(m) = \begin{cases} \max\{0, A[1]\}, & \text{if } m = 1, \\ \max\{A[m] + w(m - 1), v(m - 1)\}, & \text{if } 1 < m \leq n \end{cases}$$

We also can express $w(m)$ recursively:

$$w(m) = \begin{cases} \max\{0, A[1]\}, & \text{if } m = 1, \\ \max\{0, w(m-1) + A[m]\}, & \text{if } 1 < m \leq n \end{cases}$$

The max-sum subarray constrained to persist until day m is obviously nonnegative (since a zero-value, zero-length subarray has greater value), and if day $A[m]$ does not force the sum negative, then the value is $A[m]$ plus the previous day's value; if there were a greater-valued range, then the previous day's range could be altered to be only $-A[m]$ away from it, contradicting the optimality of the previous result. Thus, w exhibits optimal substructure. We can compute w in a linear array in a bottom-up fashion, in linear time. Also we can compute v in another linear array in bottom-up fashion in linear time. Then, of course, the value of the globally optimal solution is found in $v(n)$.

We can reconstruct the subrange itself by looking in the v table starting from index n and working backwards: we can detect when an element m is included because in that case, $v(m) \neq v(m-1)$. Something like the following should work, although this pseudocode is a bit rough:

```

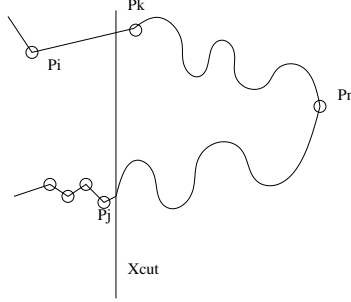
start ← 0, end ← 0
for k ← n downto 2
if end = 0 and v(k) ≠ v(k-1) then end ← k
if end ≠ 0 and v(k) = v(k-1) then begin start ← k + 1; break; end

```

5. Bitonic Euclidean TSP

Suppose we are given a set of n points $\mathcal{P} = \{p_1, \dots, p_n\}$ in the plane, and the first thing we do is sort them according to x -coordinate, so that p_1 is leftmost and p_n is rightmost; and we accept the textbook's invitation to assume "general position" with respect to x -coordinate: all x -coordinates are distinct. Also we note that the bitonic tour can be divided up into a simple "eastward" path from p_1 to p_n , and a simple "westward" path from p_n back to p_1 , where the paths' internal vertices are disjoint. (Although "east" and "west" suggest direction, we are not in fact interested in a directed path: the labels are arbitrary and intended only to distinguish these two nearly-disjoint portions of the tour.)

First we argue that this problem exhibits the optimal substructure property. Consider the globally optimal tour, and suppose we cut the tour with a vertical line at x_{cut} , which if it is between the two extrema p_1 and p_n must intersect both the eastward and westward portions of the tour. Let p_i be the rightmost vertex on the eastward path that is strictly to the left of x_{cut} , and let p_j be the rightmost vertex on the westward path strictly to the left of x_{cut} . Let p_k be the leftmost vertex to the right of (or on) x_{cut} .



Observe that p_k must be a member of the eastbound or westbound path, and thus must be adjacent to p_i or p_j (or both, iff $k = n$). If (as illustrated) p_k and p_i are adjacent, then the path from p_k around to p_j (through p_n) comprising vertices to the right of p_k must also be an optimal path; otherwise, if there were a shorter bitonic path with termini p_j, p_k , through the same vertices, then we could cut and paste it into our global optimal and lower its length, which contradicts the notion that we had an optimal solution. This argument holds regardless of whether p_i is to the right of p_j ; and so if p_k were adjacent to p_j on the globally optimal path (contrary to the illustration), for the same reasons the path from p_i around to p_k would be a shortest bitonic path through those vertices, and with those endpoints.

This leads to a nice recursive formulation, suitable for memoization. But first, some notation: we denote euclidean distance between points p_x and p_y as $d(p_x, p_y)$. Also, we denote the leftmost vertex to the right of both p_i and p_j as $r(p_i, p_j)$. Since the vertices are sorted left to right, $r(p_i, p_j) = p_{1+\max\{i, j\}}$, if it exists at all. Now define $c(p_i, p_j)$ to be the cost of a bitonic path starting at p_i , ending at p_j , and comprising all vertices to the right of both p_i and p_j . The base case occurs when $i = n$ or $j = n$, the turnaround point:

$$c(p_i, p_j) = \begin{cases} d(p_i, p_j), & \text{if } i = n \text{ or } j = n, \\ \min \left\{ \begin{array}{l} d(p_i, r(p_i, p_j)) + c(r(p_i, p_j), p_j), \\ c(p_i, r(p_i, p_j)) + d(r(p_i, p_j), p_j) \end{array} \right\}, & \text{if } i < n \text{ and } j < n. \end{cases}$$

Notice that since our definition always looks to the right, $c(p, q) = c(q, p)$ for any vertices p, q , so the c table may be triangular rather than square.

We may solve the global problem like so: the cost of the optimal euclidean bitonic tour is $d(p_1, p_2) + c(p_1, p_2)$. A recursive, memoized implementation fills in the c table in column-major order.

We can reconstruct the optimal solution if we also keep a table b of booleans such that $b(i, j)$ is true iff $r(p_i, p_j)$ is immediately adjacent to p_i in the overall solution; otherwise it must be immediately adjacent to p_j . We then can reconstruct the path like so:

```

initialize linked lists east, west
i ← 1, j ← 2
append(east,  $p_i$ ), append(west,  $p_j$ )
while i < n and j < n
    k ← 1 + max{i, j}
    if  $b(i, j)$  then      append(east,  $p_k$ ), i ← k

```

```

else
    append(west,  $p_k$ ),  $j \leftarrow k$ 
return merge(east, reverse(west))

```

In connection with writing this solution I (A.M.P.) wrote a toy MATLAB demonstration version that I will be happy to share with CSc 445 students on request.

6. Printing Neatly

To give credit where it is due, please know that the following solution is derived in large part from the solution manual.

We are given n words, of lengths l_1, \dots, l_n , and we assume no word is longer than will fit on a line, i.e., for all i , $l_i \leq M$.

Define $e(i, j)$ to be the number of extra spaces at the end of a line forced to contain words i through j . We may compute this number easily: by inspection, $e(i, j) = M - j + i - \sum_{k=i}^j l_k$. Note that for some values of i, j , clearly $e(i, j)$ could be negative.

Next we define $lc(i, j)$ to be the cost of including a line that contains words i through j . We may compute this like so:

$$lc(i, j) = \begin{cases} \infty & \text{if } e(i, j) < 0, \text{ i.e., the words don't fit,} \\ 0 & \text{if } j = n \text{ and } e(i, j) \geq 0, \\ (e(i, j))^3 & \text{otherwise.} \end{cases}$$

By making the cost of a line infinite when the words do not fit, we eliminate this possibility from occurring. By making the cost zero for the line containing the last word (provided the words all fit), we permit any number of spaces to occur at the end of this line. Thus we handle our special cases.

We observe optimal substructure in the problem like so. Suppose we have an optimal arrangement of the first j words, and suppose that the last line of our arrangement contains words i to j . Then the prior lines must contain an optimal arrangement of the first $i - 1$ words. For if there were a better arrangement of those first $i - 1$ words into lines, we could replace those lines of our supposedly optimal arrangement, and the cost would decrease, which is a contradiction.

Now let us define the cost to arrange the first j words to be $c(j)$. It seems sensible to define $c(0) = 0$. For larger arguments, we may observe that, if the last line contains words i through j then $c(j) = c(i - 1) + lc(i, j)$, which is simply the cost of the preceding lines plus the cost of the last line. Since we do not know what i is, we simply try all possibilities: $1 \leq i \leq j$. The above yields the following simple recursive formulation:

$$c(j) = \begin{cases} 0 & \text{if } j = 0, \\ \min_{1 \leq i \leq j} c(i - 1) + lc(i, j) & \text{if } j > 0. \end{cases}$$

The extra-space values e do not need $O(n^2)$ storage, just linear storage, if we store a cumulative sum of word lengths (details omitted). Because it is not recursively defined, the line-cost table lc can be filled in constant time for each entry, of which there are $O(n^2)$. In fact, by clever programming, we can eliminate both the e and lc tables (but we omit these details

too). The cost function $c(j)$ can be stored in a table that takes quadratic time to fill in a bottom-up fashion.

In order to reconstruct the optimal set of line breaks, it would be handy to store an additional table $\pi[j]$ in parallel to the table for c , and while computing $c(j)$, and searching for the index values i at which the minimum is found, for each such j we store in $\pi[j]$ the index i where the minimum occurs. Then to reconstruct the neatest paragraph, we break the lines at word $\pi[1], \pi[2]$, and so on.