Write once, solve thrice: Three gnarly problems, one solution

November 27, 2002

(See the HTML version of this document at

www.cs.utoronto.ca/ ˜ heap/Courses/270F02/A4/chains/chains.html)

Below are three problems that present themselves to the world as completely different. Somewhat surprisingly, the DP algorithms to solve these problems are strongly similar (I'll try to convince you of this a little later). If there is any justice in the world,[1] then the object-oriented features of C++ should allow you to express their similarity. Rather than write separate, repetitious solutions to these problems, you'll use C++ tools to write code that solves them generically.

The problems

Here are sketches of each problem, and a sketch (in English) of an algorithm to solve them.

Minimal triangulation of a convex polygon

A convex polygon has interior angles that are each strictly less than 180° (or π radians, if you like). A **triangulation** of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners), provided you never intersect another diagonal (except at a vertex), until all possible choices of diagonals have been used (see Figure 1).

Suppose a convex polygon has vertices v_0, …, v_n. In any triangulation we can assign a weight to each triangle: the length of its perimeter.[2] Let $w(i, j, k)$ denote the length of the perimeter of $\triangle v_i v_j v_k$. The cost of a triangulation is just the sum of the weights of its component triangles. We want to find a triangulation with the **minimum** cost.

Here's a sketch of how to find a minimum cost triangulation of a convex polygon with $n + 1$ vertices:

1. Number the vertices of your polygon from 0 to n.
2. Denote the cost of a minimal triangulation (which you haven't yet found) of your polygon as $c[0][n]$. Similarly, your polygon contains sub-polygons with vertices i through k (if k is at least 2 greater than i), and you can denote the weight of a minimal triangulation of such a sub-polygon as $c[i][k]$.
3. Possible values of $c[i][k]$ fall into two cases. **Case one:** if $k < i+2$ then the polygon with vertices v_i … v_k has fewer than 3 vertices, and no triangulation is possible, so an appropriate minimum triangulation cost is 0. **Case two:** if $k \geq i + 2$, then there are one or more choices of j where $i < j < k$. For each

[1] An unproved conjecture…

[2] Different weight functions may be used. If you make the weight of a triangle equal to its area, minimal triangulation becomes really easy

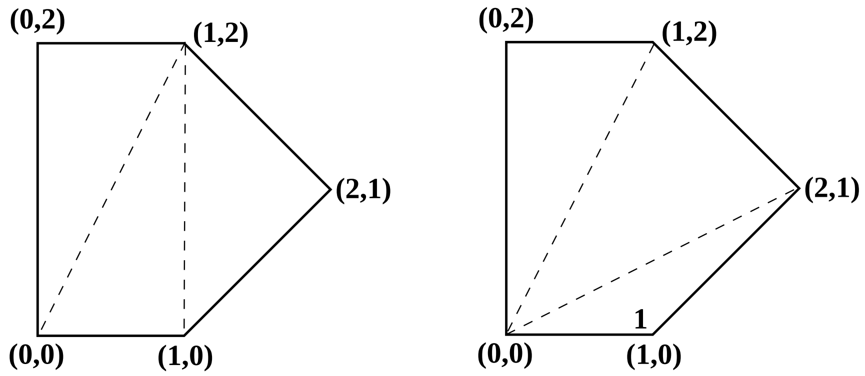

Figure 1: Two triangulations of the same convex pentagon. The triangulation on the left has a cost of $8 + 2\sqrt{2} + 2\sqrt{5}$ (approximately 15.30), the one on the right has a cost of $4 + 2\sqrt{2} + 4\sqrt{5}$ (approximately 15.77). If you index the vertices counter-clockwise from $(0, 0)$ (vertex 0) to $(0, 2)$ (vertex 4), then you could specify the triangulation on the left by listing the indices of its component triangles: `0 3 4`, `0 1 3`, and `1 2 3`. You could specify the triangulation on the right similarly: `0 3 4`, `0 2 3`, `0 1 2`.

choice of j, calculate $c[i][j] + c[j][k] + w(i, j, k)$ — the minimum over all appropriate j is $c[i][k]$. These two cases give the following recursive formula:

$$c[i][k] = \begin{cases} 0 & k < i + 2 \\ \min_{i<j<k}\{c[i][j] + c[j][k] + w(i, j, k)\} & \text{otherwise} \end{cases} \tag{1}$$

Record which index j corresponds to the minimum (you might denote it $m[i][k]$).

4. Suppose $m[0][n]$ is denoted j_0, then you know that $\triangle v_0 v_{j_0} v_n$ is part of a minimal triangulation. You can find more triangles in this minimal triangulation using $m[0][j_0]$ and $m[j_0][n]$. Continue until you've found all triangles in a minimal triangulation.

Although it is possible to build up a table of values $c[i][k]$ iteratively (as described in your course notes), for this assignment you must find $c[0][n]$ recursively, using memoization (mentioned in lecture) to prevent redundant calculations of the same value.

Optimal grouping of a matrix product

A matrix is a rectangular array of numbers. The product of two matrices, $M_1 M_2$ is defined, so long as M_1 has the same number of columns as M_2 has rows. Suppose M_1 has r_1 rows and c_1 columns, M_2 has r_2 rows and c_2 columns (c_1 must equal r_2), then the product $M_1 M2$ requires $r_1 r_2 c_2$ multiplications of array elements. Matrix multiplication is associative (although not commutative), so three or more matrices can be multiplied (so long as the appropriate adjacent dimensions match):

$$M_1 M_2 M_3 = (M_1 M_2) M_3 = M_1 (M_2 M_3).$$

Although the value of $M_1 M_2 M_3$ is the same whichever way you group them, the amount of work (multiplications of array elements) is not. For example, suppose M_1 is a 2×3 matrix (2 rows, 3 columns), M_2 is 3×4, and M_3 is 4×5. Then the grouping $(M_1 \times M_2)M_3$ costs: $24 + 40 = 64$ multiplications. The grouping $M_1(M_2 M_3)$ costs: $60 + 30 = 90$ multiplications. This disparity gets worse for longer chains of matrices. What is the best way to group

$$M_0 M_1 \ldots M_k?$$

Here is a first attempt at finding a minimum-cost grouping. In analogy with triangulation, denote the minimum cost of grouping matrices $M_0 \ldots M_k$ as $c[0][k]$, and the product $r_0 r_j c_k$ as $w(i, j, k)$. Then for each choice of j with $0 \leq j < k$ we check the value of $c[0][j-1] + c[j][k] + w(i,j,k)$, and take the minimum. This formula looks close to that for triangulation except for two things: the first part of the sum is $c[0][j-1]$ rather than $c[0][j]$ (since the matrices don't overlap), and the function $w(i,j,k)$ is different.

You can fix the first problem by changing notation slightly. **Define** $c[i][k]$ to be the minimum cost of grouping matrices $M_0 \ldots M_{k-1}$ (instead of $M_0 \cdots M_k$), so $w(i,j,k) = r_i r_j c_{k-1}$, and our recursive formula is now the same as Equation 1, except for $w(i,j,k)$ being different (and aren't there C++ features that might help with that?). You can use the same algorithm for both problems. The cost of a minimal grouping of matrices $M_0 \cdots M_n$ is $c[0][n+1]$.

Optimal structure of a BST

Suppose you need a binary search tree (BST) that makes searches as efficient as possible, given the frequency of the keys. Given a list of pairs

$$(K_0, f_0), (K_1, f_1), \ldots, (K_n, f_n),$$

...where the K_i are keys, $K_i < K_j$ (in some ordering) if $i < j$, and the f_i are integers specifying how frequently the keys occur. For each BST comprised of nodes containing these keys, denote the depth of the node containing K_i as d_i (with the depth of the root node being 1, its children being 2, and so on). The cost of such a BST is the sum of $d_i \times f_i$ (for all i) — you expect to have to traverse depth d_i f_i times retrieving the node with key K_i. What is the best way to construct a BST in order to minimize the cost? **Note:** For this assignment, BSTs have keys in all nodes, not just in the leaves.

Here is a first attempt. Denote the minimum cost of a BST comprised of nodes containing $K_i \ldots K_k$ as $c[i][k]$, and the weight added by selecting some K_j $(i \leq j \leq k)$ as the root with subtrees $K_i, \ldots, K_{j-1}$ and $K_{j+1}, \ldots, K_k$ as $w(i,j,k) = f_i + \cdots + f_j + \cdots + f_k$. You should draw some small BSTs to see why this $w(i,j,k)$ is suitable. Now, for each each choice of j with $i \leq j \leq k$ find $c[i][j-1] + c[j+1][k] + w(i,j,k)$. Again, this is close, but not identical, to the formula for triangulation and matrix multiplication, so you'll need to slightly change the notation.

Denote the minimum cost of a BST comprised of nodes containing $K_{i+1}, \ldots, K_{k-1}$ as $c[i][k]$, and $w(i,j,k)$ as the sum $f_{i+1} + \cdots + \cdots + f_{k-1}$. Now your recursive formula is Equation 1. If you set out to find a minimal BST for values $(K_0, f_0), \ldots, (K_n, f_n)$, the corresponding minimal cost will be $c[-1][n+1]$, and the algorithm is the same (except for the definition of $w(i,j,k)$) as the other two problems.

Your job

You'll need to download testChain.cpp, chain.h, Makefile. You will need to see the HTML version of this document:

www.cs.utoronto.ca/ ˜ heap/Courses/270F02/A4/chains/chains.html

in order to click or shift-click on the appropriate links (no known pencil, pen, stylus, or digit appears to work on the paper version).

Once you've downloaded the appropriate files, you have these tasks:

1. Implement `minPartition(int i, int k)` (see chain.h). This returns a minimal partition (a list of components, see below) of the range `i...k` of a chain. Some definitions:

   range `i...k` For a polygon, this means the (sub)polygon from vertex `i` to vertex `k`. For a matrix chain, this means the (sub)chain from matrix `i` to matrix `k-1`. For a BST this means the (sub)BST from node `i+1` to node `k-1`.

   component i j k A triple of integers that specify a component of a partition. For a polygon this specifies the triangle with vertices vertex i, vertex j, and vertex k. For a matrix chain this specifies the parenthesization of matrices i through $j-1$ and then of matrices j through $k-1$. For a BST this specifies that node j is the root of a (sub)tree with left child comprised of nodes $i+1$ through $j-1$ and right child comprised of nodes $j+1$ through $k-1$.

$w(i, j, k)$ The weight of component i j k. For a polygon, this is the perimeter of the triangle with vertices vertex i, vertex j, and vertex k. For a chain of matrices this is the product $r_i r_j c_{k-1}$, where r_i is the number of rows in matrix i, r_j the number of rows in matrix j, and c_{k-1} the number of columns in matrix $k - 1$. For a BST this is the sum of frequencies $f_{i+1} + \cdots + f_{k-1}$.

link The basic unit of a chain. For a vertexChain, this is a vertex with a pair of double values for its coordinates. For a matrixChain, this is a single matrix with a pair of doubles describing its dimensions (number of rows, number of columns). For a bstChain, this is a node with a pair of doubles representing its key and frequency.

2. Implement `partitionTally(componentNode *partitionList)` (see chain.h). This returns the sum of the weights $w(i, j, k)$ of components in the linked list `partitionList`.

3. Implement constructors for classes chain in a file called `chain.cpp`, vertexChain in a file called `vertexChain.cpp`, matrixChain in a file called `matrixChain.cpp`, and bstChain in a file called `bstChain.cpp`.

4. Implement any other functions, constants, or types needed to make testChain.cpp work when it is provided with input of the form described below. You may **not** change `testChain.cpp`. You may add to `chain.h`, but you may **not** change any functions or types originally declared there.

For a chain with n links (e.g. a polygon with n sides, a chain of n matrices, a BST with n nodes), your code should have worst-case $O(n^3)$ running time. Your solution should combine a recursive algorithm with memoization. Your solution should use generic code, rather than three repetitive solutions.

Here is what `testChains.cpp` expects on standard input:

1. One of the strings `polygon`, `matrix`, or `bst`, followed by white space. This indicates the type of chain that is being partitioned.

2. An integer (we'll call it n) followed by white space. This indicates the number of links in the chain.

3. n pairs of double literals, separated and followed by white space. These are the values of the links.

4. A single double literal, followed by white space. This represents the (approximate) calculated minimum value for partitioning this chain.

As an example, look at triangle.chn or any of the other examples in ../Code/TestChains. Once your code is working, you should be able to see the following test result:

```
./testChains < triangle.chn
Chain type: polygon     Minimal partition
```

What to submit

Assignments are due 11:59:59 on Friday, December the 6th.

You'll need to submit your `chain.cpp`, `vertexChain.cpp`, `matrixChain.cpp`, and `bstChain.cpp`. If you modify `chain.h` (you may add declarations, but not change those already present), then submit your versionof `chain.h`. It is your responsibility to ensure that when all these files are present in the same directory with the supplied Makefile (which you don't change nor submit), the command

```
make
```

...executes without error.

Once you're satisfied with your results, you may submit your work:

```
submit -c csc270h -a a4 chain.h chain.cpp vertexChain.cpp matrixChain.cpp bstChain.cpp
```

Your can check that your assignment has been submitted with

```
submit -l -c csc270h -a a3
```

You can replace a file of the same name that you have previously submitted with (using `vertexChain.cpp` as an example):

```
submit -c csc270h -a a3 -f vertexChain.cpp
```

Late assignments are not normally accepted, and *always* require a written explanation (e-mail is a form of writing).

Submit your own work! We detected a case of excessively similar code in A2, and we are pursuing a possible academic offense due to it. Don't be the next one.

1 Above and beyond

Warning: The material in this section won't (directly) earn you any marks.

Your course notes describe the similarity between the matrix chain multiplication problem and the optimal BST problem, however the suggested solution is iterative and doesn't qualify for this assignment. *Algorithms* by Cormen, Leiserson and Rivest (CL&R) describes the similarity between polygon triangulation, matrix chain multiplication, and parse trees. Be warned that CL&R use slightly different notation, so the formula for $c[i][k]$ might be off-by-one from this handout. CL&R also describe a single algorithm that solves an entire class of DP problems (CL&R, Section 26.4). You need some familiarity with abstract algebra to read this section.

The similarity of these three problems is discussed in www.ics.uci.edu/ ˜ eppstein/260/011023/, where all three are reduced to triangulations. Be warned that my formulation of the optimal BST structure allows keys to occupy all nodes (both internal and leaves), whereas on this page the keys occur only at the leaves.