CPS260/BGT204.1 Algorithms in Computational Biology **September 9, 2003**

Lecture 5: Greedy Algorithms, Dynamic Programming

Lecturer: Pankaj K. Agarwal *Scribe: David Vaughn*

5.1 Optimization Problems

These are problems with many possible solutions, and the goal is to find the best (i.e. *optimal*) solution. Examples of optimization problems include:

Shortest Path: find the shortest path in a graph

Sequence Alignment: find the best possible alignment between two DNA sequences

Matrix Multiplication: find the best order in which to multiply n matrices

Two algorithmic models for solving optimization problems are *greedy algorithms* and *dymanic programming*. Often, both techniques can be used on the same problem, but there are some major differences in the results:

Greedy algorithms are generally faster, but do not always yield the optimal solution.

Dynamic programming is slower, but finds the optimum solution.

5.2 Greedy Algorithms

Basics. It is hard to formally define what is meant by a Greedy Algorithm, but one generally has these important features:

- it builds up a solution in small steps,
- at each step, it makes what *seems* to be the best choice based on local, readily available information.

As stated above, greedy algorithms do not generally find the optimum solution (i.e. change with minimum # of coins – see lecture 4), but for some problems it can be proven that they do, for example:

Minimum Spanning Tree: Given a connected (i.e. there is a path between every pair of nodes), undirected graph $G = (V, E)$, and an associated weight w for each every edge in E, find a subset $T \subseteq E$ such that $G' = (V, T)$ is connected and the sum of all the edge weights $w \in T$ is minimized.

Single Linkage Clustering: Given two clusters of points, find the distance between the closest pair of points, such that the two points are not in the same cluster.

Interval Packing: Given a set of overlapping intervals, find a largest subset of pairwise-disjoint intervals.

Example: Interval Packing Sometimes also called *interval scheduling*, it can be states as follows: *Given a set of n intervals $I = \{i_1, i_2, ..., i_n\}$, where the k^{th} interval starts at time s_k and finishes at time f_k, find a largest subset of intervals that do not overlap.*

The basic idea in a greedy algorithm for this problem is to use a simple rule, based on local information, to choose the first interval i to be in the solution. Then we rule out all intervals overlapping with i, and then make the next choice from the remaining intervals based on the same rule, and so on, until there are no intervals left. The only problem is, what rule should we choose. There are several options:

- always choose the interval that starts first (min s)
- always choose the interval that is the shortest (min $f - s$)
- always choose the interval that finishes first (min f)

It turns out that it is easy to find counterexamples showing that the first two rules don't always work, but the third rule works, leading to the algorithm described in Figure 5.1:

```
S = ∅
while I ≠ ∅
  remove from I the interval i with the smallest f
  insert i into S
  remove from I all intervals that overlap with i
return S
```

Figure 5.1: A greedy algorithm for interval packing.

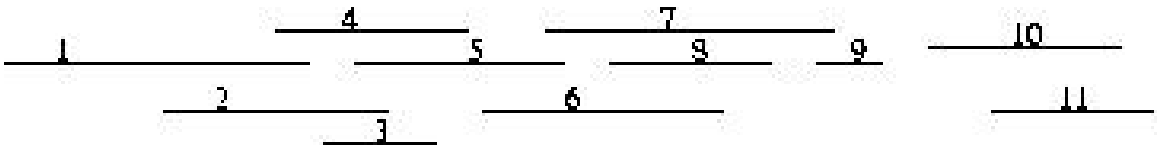

Figure 5.2: Interval packing example.

Let's trace through the algorithm on the above example:

1. Interval 1 finishes first, so it is put in S, and 1,2,4 are removed from I.
2. Of the remaining, 3 finishes first, so it goes to S, and 3 and 5 are removed from I.

3. 6 finishes next, so it goes in S, and 6,7,8 are removed from I.
4. 9 goes to S, and removed from I.
5. 10 goes to S, 10 and 11 are removed from I, and we are done.

Although this is a simple algorithm, it is not obvious that it will always work, so here is a brief proof:

Sketch of Proof The idea is to assume there is an optimal solution T, and then show that $|S| = |T|$. Let $i_1...i_k$ be the intervals in S in the order they were added to S. So $|S| = k$. Let $j_1...j_m$ be the the intervals in T. We must prove that $k = m$.

Lemma 1 *For every pair of intervals i_n and j_n, i does not finish after j (i.e. $f_{i_n} \leq f_{j_n}$).*

This is easy to prove by induction. For $n = 1$, this is true, since the algorithm chooses the earliest finishing interval to put in S first. Now assume that for $n - 1$ this is true. We also know that interval j_{n-1} finishes before interval j_n starts ($f_{j_{n-1}} \leq s_{j_n}$). This means that j_n is one of the options when our greedy algorithm chooses its n^{th} interval, and since the algorithm always chooses the interval with the smallest f, it must be true that $f_{i_n} \leq f_{j_n}$

Now we can prove by contradiction that S is optimal. If S is not optimal, then $|T| > |S|$, or $m > k$. From the lemma we know that $f_{i_k} \leq f_{j_k}$. But since $m > k$, there must be an interval j_{k+1} in T. This interval starts after j_k ends, and therefore after i_k ends. This means that after the greedy algorithm chose i_k, there was still a non-overlapping interval left in I, a contradiction to the stopping condition of the algorithm. Thus S is optimal. ■

5.3 Dynamic Programming

Basics. As stated before, there are problems where there is no simple, greedy solution. In these cases, a more systematic approach, dynamic programming, is called for. The basic idea of dynamic programming is to *show how to construct an optimum solution to a problem from optimum solutions to smaller subproblems.*

- This is usually accompanied by a *recurrence equation*, expressing the *cost* or *value* of the solution as the total cost of the subsolutions, plus the cost of the final step of constructing the solution from these subsolutions.
- Then we have a blueprint showing how to build an optimal solution from the bottom up, and an equation that gives the associated value of this solution

Therefore, for a problem to be amenable to dynamic programming, it must be *decomposable* into smaller subproblems. For example: Consider the problem of finding the shortest path between Durham and Washington, D.C. If we know that the path must go through South Hill, then we can express the problem in terms of 2 problems, namely, finding the shortest path from Durham to South Hill, and then from South Hill to Washington, D.C.

You might notice that this idea bears a striking resemblence to the the *divide-and-conquer* technique, such as is employed in the *merge-sort* algorithm. There are two important differences, however:

- The divide-and-conquer technique is usually applied to a problem with a unique solution, so the underlying problem is typically not an optimization problem.
- A divide-and-conquer algorithm considers each subproblem only once, whereas a dynamic-programming based algorithm may consider the same subproblem several times — thus the importance of *keeping track of the solution to each subproblem the first time it is solved* so as to avoid re-solving the same problem several times (which could lead to combinatorial explosion!)

Example: Matrix-chain multiplication. An $m \times n$ matrix A has m rows and n columns, and the element in the i^{th} row and the j^{th} column is denoted as a_{ij}.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots \\ a_{21} & a_{22} & \dots \\ \vdots & \vdots & a_{mn} \end{pmatrix}$$

Given an $\ell \times m$ matrix A and an $m \times n$ matrix B, the product of A and B is an $\ell \times n$ matrix C where c_{ij} is the *inner product* of the i^{th} row of A and the j^{th} column of B. That is,

$$\begin{pmatrix} a_{11} & a_{12} & \dots \\ a_{21} & a_{22} & \dots \\ \vdots & \vdots & a_{lm} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \dots \\ b_{21} & b_{22} & \dots \\ \vdots & \vdots & b_{mn} \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{m} a_{1k} \cdot b_{k1} & \cdots & \sum_{k=1}^{m} a_{1k} \cdot b_{kn} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{m} a_{\ell k} \cdot b_{k1} & \cdots & \sum_{k=1}^{m} a_{\ell k} \cdot b_{k\ell} \end{pmatrix}.$$

It should be clear that the number of columns in A must be equal to the number of rows in B, otherwise $A \times B$ is undefined. As you can see, the number of entries in C is $\ell \cdot n$, and the number of terms summed to calculate each entry is m, so we say that matrix multiplication is takes ℓmn multiplications, or is $O(\ell mn)$.

Now consider the problem of multiplying k matrices:

Matrix-Chain Multiplication Problem: Given an ordered list of k matrices $M_1 M_2 M_3 ... M_k$, where matrix M_i has dimension $p_{i-1} \times p_i$, parenthesize the list to indicate the order in which to multiply the matrices so that the total number of multiplications is minimized.

For example, consider the chain $M_1 M_2 M_3 M_4$ where the matrices are of dimensions 1×10, 10×1, 1×10, and 10×1 respectively. There are several orders in which they can be multiplied:

- If they are multiplied in the order $(M_1(M_2 M_3))M_4$, then the total # of operations is

  100 + 100 +10 = 210

- On the other hand, if they are multiplied in the order $(M_1 M_2)(M_3 M_4)$, then the total # of operations is

  10 + 10 + 1 = 21!

Clearly, order matters!

One first impulse (it would be wrong) would be to just consider every possible parenthesization of the chain. Let's see how many such choices are. If one has a chain of n matrices, it can be split into two chains $M_1...M_k$ and $M_{k+1}...M_n$ for any value k between 1 and n, and the two chains can be paranthesized independently $(M_1...M_k)(M_{k+1}...M_n)$. For this value of k, the total # of possible parenthesizations is the product of the total # from each of the two subchains. Summing over all k we obtain a recurrence equation expressing the total # of orderings:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

It turns out that $P(n)$ increases very rapidly with n. More specifically, $P(n) = C(n-1)$, where $C(n)$ are the *Catalan Numbers*:

$$C(n) = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix} \approx 4^n.$$

Obviously, it is unrealistic to search the whole space of possible parenthesizations. However the problem is *decomposable*:

- Remember how the recurrence equation above was formulated. It turns out that the same idea can help formulate a faster solution.
- Given a chain of n matrices, $M_1...M_k$, we know the optimal solution will perform some multiplication $M_{1..k}M_{k+1..n}$ last (here, $M_{i..j}$ is just shorthand for the matrix that results from the product $M_iM_{i+1}...M_j$).
- Therefore, we know that the cost of the optimal solution is just the cost of the optimal solution to $M_{1..k}$ plus the cost of the optimal solution to $M_{k+1..n}$ plus the cost of multiplying the two together.

Using this idea, we can write a recurrence equation for $C(i,j)$, the cost (i.e. number of multiplications) in the optimal solution to the problem of calculating $M_{i..j}$

$$C(i,j)) = \begin{cases} \min_{i \leq l \leq j} (C(i,l) + C(l,j) + p_{i-1}p_lp_j) & \text{if } i < j, \\ 0 & \text{if } i = j. \end{cases}$$

Thus, the goal is to find $C(1,n)$.

A recursive algorithm. This recurrence equation might lead one to write a simple recursive program to solve this problem, as shown in Figure 5.3:

The problem with this algorithm is that it revisits the same subproblems many times, and each time it does not remember what it did before, so it solves them again. This is obviously very inefficient.

The running time of this algorithm is given by the following recurrence equation, where $T(n)$ denotes the

```
MULTORDER(i, j)
  if i = j then
    return 0
  C(i, j) = ∞
  for l = i to j − 1 do
    q = MULTORDER(i, l) + MULTORDER(l + 1, j) + p_{i−1} p_l p_j
    c = min(q, c)
  return (c)
```

Figure 5.3: A recursive algorithm for matrix chain multiplication.

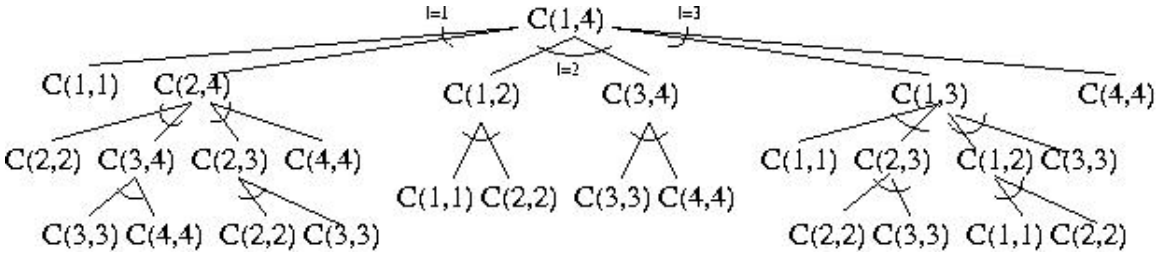

Figure 5.4: Recursion tree for multiplying 4 matrices.

running time of MULTORDER$(1, n)$ (more generally, for MULTORDER$(i, i + n - 1)$):

$$T(n) = \begin{cases} \sum_{l=1}^{n-1} (T(l) + T(n-l) + 1) & \text{if } n > 1, \\ 1 & \text{if } n = 1. \end{cases}$$

It can be shown that $T(n) > 2^n$, so clearly this inefficiency leads to a very slow algorithm.

In general, there are 2 pitfalls to recursive algorithms;

1. Re-solving the same problem many times (see figure 5.2).
2. Overhead of recursive calls: each time a function is called, the computer has to do a lot of shuffling around of variables, scopes, etc.

Modified recursive solution: Memoizing. If we modify the recursive solution to remember the solutions that have been computed already, we can speed up the running time immensely:

This technique of using a recursive algorithm together with a table to record values is sometimes called *Memoizing*. In this case, with three nested for-loops, it is easy to see the running time $T(n) \in O(n^3)$ This solution solves pitfall # 1 from above, but not # 2. An iteritive solution would be even faster.

A dynamic-programming based algorithm. The iterative solution described in Figure 5.6 is similar to the memoized solution, except the table of values is explicitly filled in from the bottom up.

```
MULTORDER(i, j)
  if i = j then
    return 0
  C(i, j) = ∞
  for l = i to j − 1 do
    if C(i, l) =↑
      MULTORDER(i, l)
    if C(l + 1, j) =↑
      MULTORDER(l + 1, j)
    q = C(i, l) + C(l + 1, j) + p_{i−1} p_l p_j
    c = min(q, c)
  return (c)

*note: ↑ means undefined, or not yet calculated
```

Figure 5.5: An efficient recursive algorithm for matrix chain multiplication.

```
MULTORDER (1, k)
  for i = 1 to k do
    C(i, i) = 0
  for l = 1 to k − 1 do
    for i = 1 to k − l do
      j = i + l
      C(i, j) = ∞
      for u = i to j − 1 do
        q = C(i, u) + C(u + 1, j) + p_{i−1} p_u p_j
        if q < C(i, j)
          C(i, j) = q
          S(i, j) = u
  return (C(1, k))
```

Figure 5.6: A dynamic-programming based algorithm for matrix chain multiplication.

Here, we have added an additional table $S(i, j)$ that allows us to retrieve the actual solution, in addition to the table that allows us to calculate its cost.

Now we will trace the solution of the original example, the chain $M_1M_2M_3M_4$ where the matrices are of dimensions 1×10, 10×1, 1×10, and 10×1 respectively. It will be helpful to remember that:

- $C(i, j)$ in the table denotes the minimum cost for computing the chain $M_{i..j}$
- Matrix M_i has dimensions $p_{i-1} \times p_i$ in this case, that means

$$p_0 = 1, p_1 = 10, p_2 = 1, p_3 = 10, p_4 = 1$$

Step-by-step through the algorithm.

1. First, all $C(i, i)$ are initialized to 0 (Figure 5.7).
2. Next, the optimum solutions for all subchains of length 2 are calculated (Figure 5.8).
3. Then, subchains of length 3 (Figure 5.9).
4. Finally, the optimum solution is found (Figure 5.10).

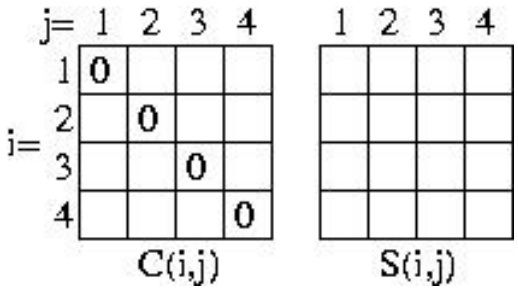

Figure 5.7: Tables C and S after $C(i, i)$ are initialized.

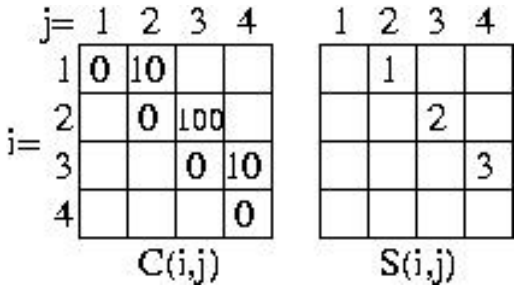

Figure 5.8: After the $l = 1$ iteration of the outer for loop.

Thus we have found the minimum cost. Again, since there are three nested for loops, this algorithm is $O(n^3)$.

One may be wondering how to actually obtain the correct ordering from the S table. Basically, one keeps dividing the chain into two. Start with $S(1, k)$: this value i means the optimal solution was obtained by performing the multiplication $M_{1..i}M_{i+1..j}$ last. Then look up $S(1, i)$ and $S(i + 1, j)$ to find where to divide these two chains, and so on.

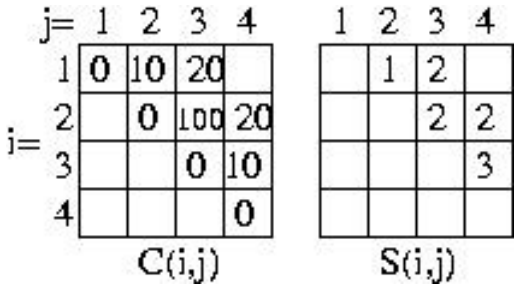

Figure 5.9: After the $l = 2$ iteration of the outer for loop.

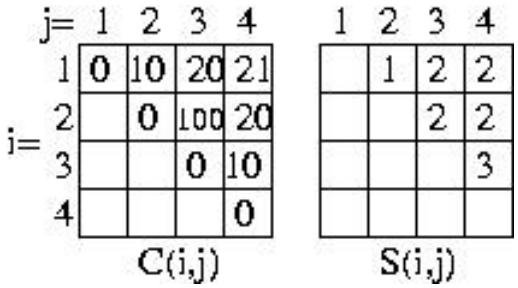

Figure 5.10: After the $l = 3$ iteration of the outer for loop.

5.4 Bibliographic notes

Most of the material covered in this lecture can be found in

- Cormen, T., Leiserson, C. and Stein, C., *Introduction to Algorithms*, MIT Press.
- Kleinberg, J. and Tardos, E., *Introduction to Algorithms*, unpublished manuscript.