Programming Team Lecture: Dynamic Programming

Standard Algorithms to Know

Computing Binomial Coefficients (Brassard 8.1)
World Series Problem (Brassard 8.1)
Making Change (Brassard 8.2)
Knapsack (Brassard 8.4 Goodrich 5.3)
Subset Sum (special instance of knapsack where weights=values)
Floyd-Warshall's (Brassard 8.5 Cormen 26.2)
Chained Matrix Multiplication (Brassard 8.6, Cormen 16.1 Goodrich 5.3)
Longest Common Subsequence (Cormen 16.3)
Edit Distance (Skiena 11.2)
Polygon Triangulation (Cormen 16.4)

Example #1: The Matrix Chain Problem

Given a chain of matrices to multiply, determine the how the matrices should be parenthesized to minimize the number of single element multiplications involved.

First off, it should be noted that matrix multiplication is associative, but not commutative. But since it is associative, we always have:

((AB)(CD)) = (A(B(CD)))

or equality for any such grouping as long as the matrices in the product appear in the same order.

It may appear on the surface that the amount of work done won't change if you change the parenthesization of the expression, but we can prove that is not the case with the following example:

Let A be a 2x10 matrix
Let B be a 10x50 matrix
Let C be a 50x20 matrix

Note that any matrix multiplication between a matrix with dimensions ixj and another with dimensions jxk will perform ixjxk element multiplications creating an answer that is a matrix with dimensions ixk. Also note that the second dimension in the first matrix and the first dimension in the second matrix must be equal in order to allow matrix multiplication.

Consider computing A(BC):

multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix

multiplications for A(BC) = 2x10x20 = 400,

Total multiplications = 10000 + 400 = 10400.

Consider computing (AB)C:

multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix

multiplications for (AB)C = 2x50x20 = 2000,

Total multiplications = 1000 + 2000 = 3000, a significant difference.

Thus, the goal of the problem is given a chain of matrices to multiply, determine the fewest number of multiplications necessary to compute the product. We will formally define the problem below:
Let $A = A_0 \bullet A_1 \bullet \ldots A_{n-1}$

Let $N_{i,j}$ denote the minimal number of multiplications necessary to find the product $A_i \bullet A_{i+1} \bullet \ldots A_j$. And let $d_i x d_{i+1}$ denote the dimensions of matrix A_i.

We must attempt to determine the minimal number of multiplications necessary($N_{0,n-1}$) to find A, assuming that we simply do each single matrix multiplication in the standard method.

The key to solving this problem is noticing the sub-problem optimality condition:

If a particular parenthesization of the whole product is optimal, then any sub-parenthesization in that product is optimal as well. Consider the following illustration:

Assume that we are calculating ABCDEF and that the following parenthesization is optimal:

(A (B ((CD) (EF))))

Then it is necessarily the case that

(B ((CD) (EF)))

is the optimal parenthesization of BCDEF.

Why is this?

Because if it wasn't, and say (((BC) (DE)) F) was better, then it would also follow that

(A (((BC) (DE)) F)) was better than

(A (B ((CD) (EF)))), contradicting its optimality.

This line of reasoning is nearly identical to the reasoning we used when deriving Floyd-Warshall's algorithm.

Now, we must make one more KEY observation before we design our algorithm:

Our final multiplication will ALWAYS be of the form

$(A_0 \bullet A_1 \bullet \ldots A_k) \bullet (A_{k+1} \bullet A_{k+2} \bullet \ldots A_{n-1})$

In essence, there is exactly one value of k for which we should "split" our work into two separate cases so that we get an optimal result. Here is a list of the cases to choose from:

$(A_0) \bullet (A_1 \bullet A_{k+2} \bullet \ldots A_{n-1})$
$(A_0 \bullet A_1) \bullet (A_2 \bullet A_{k+2} \bullet \ldots A_{n-1})$
$(A_0 \bullet A_1 \bullet A_2) \bullet (A_3 \bullet A_{k+2} \bullet \ldots A_{n-1})$
...
$(A_0 \bullet A_1 \bullet \ldots A_{n-3}) \bullet (A_{n-2} \bullet A_{n-1})$
$(A_0 \bullet A_1 \bullet \ldots A_{n-2}) \bullet (A_{n-1})$

Basically, count the number of multiplications in each of these choices and pick the minimum. One other point to notice is that you have to account for the minimum number of multiplications in each of the two products.

Consider the case multiplying these 4 matrices:

A: 2x4
B: 4x2
C: 2x3
D: 3x1

1. (A)(BCD) - This is a 2x4 multiplied by a 4x1,
so 2x4x1 = 8 multiplications, plus whatever
work it will take to multiply (BCD).

2. (AB)(CD) - This is a 2x2 multiplied by a 2x1,
so 2x2x1 = 4 multiplications, plus whatever
work it will take to multiply (AB) and (CD).

3. (ABC)(D) - This is a 2x3 multiplied by a 3x1,
so 2x3x1 = 6 multiplications, plus whatever
work it will take to multiply (ABC).

Thus, we can state the following recursive formula:

$N_{i,j}$ = min value of $N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}$, over all
valid values of k.

One way we can think about turning this recursive formula into a dynamic programming solution is by deciding which sub-problems are necessary to solve first. Clearly it's necessary to solve the smaller problems before the larger ones. In particular, we need to know $N_{i,i+1}$, the number of multiplications to multiply any adjacent pair of matrices before we move onto larger tasks. Similarly, the next task we want to solve is finding all the values of the form $N_{i,i+2}$, then $N_{i,i+3}$, etc. So here is our algorithm:

1) Initialize N[i][i] = 0, and all other entries in N to ∞.
2) for i=1 to n-1 do the following
    2i) for j=0 to n-1-i do
        2ii) for k=j to j+i-1
            2iii) if (N[j][j+i-1] >
                N[j][k]+N[k+1][j+i-1]+$d_j d_{k+1} d_{i+j}$)

                N[j][j+i-1]=
                N[j][k]+N[k+1][j+i-1]+$d_j d_{k+1} d_{i+j}$

Here is the example we worked through in class:

Matrix	Dimensions
A	2x4
B	4x2
C	2x3
D	3x1
E	1x4

	A	B	C	D	E
A	0	16	28	22	30
B		0	24	14	30
C			0	6	14
D				0	12
E					0

First we determine the number of multiplications necessary for 2 matrices:

AxB uses 2x4x2 = 16 multiplications
BxC uses 4x2x3 = 24 multiplications
CxD uses 2x3x1 = 6 multiplications
DxE uses 3x1x4 = 12 multiplications

Now, let's determine the number of multiplications necessary for 3 matrices

(AxB)xC uses 16 + 0 + 2x2x3 = 28 multiplications
Ax(BxC) uses 0 + 24 + 2x4x3 = 48 multiplications, so 28 is min.

(BxC)xD uses 24 + 0 + 4x3x1 = 36 multiplications
Bx(CxD) uses 0 + 6 + 4x2x1 = 14 multiplications, is 14 is min.

(CxD)xE uses 6 + 0 + 2x1x4 = 14 multiplications
Cx(DxE) uses 0 + 12 + 2x3x4 = 36, so 14 is min.

Four matrices next:

Ax(BxCxD) uses 0 + 14 + 2x4x1 = 22 multiplications
(AxB)x(CxD) uses 16 + 6 + 2x2x1 = 26 multiplications
(AxBxC)xD uses 28 + 0 + 2x3x1 = 34 multiplications, 22 is min.

Bx(CxDxE) uses 0 + 14 + 4x2x4 = 46 multiplications
(BxC)x(DxE) uses 24 + 12 + 4x3x4 = 84 multiplications
(BxCxD)xE uses 14 + 0 + 4x1x4 = 30 multiplications, 30 is min.

For the answer:

Ax(BxCxDxE) uses 0 + 30 + 2x4x4 = 62 multiplications
(AxB)x(CxDxE) uses 16 + 14 + 2x2x4 = 46 multiplications
(AxBxC)x(DxE) uses 28 + 12 + 2x3x4 = 64 multiplications
(AxBxCxD)xE uses 22 + 0 + 2x1x4 = 30 multiplications

Answer = 30 multiplications

Dynamic Programming Notes: World Series Problem

The World Series involves two teams that play a best of 7 games contest. The first team to win 4 games wins the World Series. (In the most recent year, 2007, that team was the Boston Red Sox!!!)

To generalize the problem slightly, we might as the following question:

Given that in a single contest between team A and team B, the probability team A wins is p, (and the probability that team B wins is 1-p, so we are precluding ties), what is the probability in i+j games played, that team A will win i games and team B will win j games. Let this value be denoted by the function p(i,j).

Using some mathematics, we can answer the question with the following formula (from the binomial theorem):

$$p(i, j) = \binom{i+j}{i} p^i (1-p)^j$$

An alternative way to solve the problem involves dynamic programming. In order to obtain the dynamic programming solution, we must first develop a recursive formula for the function p(i,j). In order for team A to have won i games and team B to have won j games, before the last game, either A won i and B won j-1 OR A won i-1 and B won j. Here is a recursive formula that captures that reasoning:

$$p(i, j) = p(i, j-1) * (1-p) + p(i-1, j) * p$$

Now, instead of actually using recursion in our code to figure this out, we can convert our algorithm into dynamic programming by creating a two-dimensional array that stores the answers to all necessary recursive calls. Here's some (uncompiled) Java code to solve the problem:

```
public static int WS(int i, int j, double p) {

  double[][] prob = new double[i+1][j+1];

  prob[0][0] = 1;

  // Fill in probabilities for team A sweeping.
  for (int Awin=1; Awin<=i; Awin++)
    prob[Awin][0] = p*prob[Awin-1][0];

  // Probabilities for team B sweeping
  for (int Bwin=1; Bwin<=j; Bwin++)
    prob[0][Bwin] = (1-p)*prob[0][Bwin-1];

  // Here's where we fill in the table, using the recursive
  // formula.
  for (int Awin=1; Awin<=i; Awin++)
    for (int Bwin=1; Bwin<=j; Bwin++)
      prob[Awin][Bwin] = p*prob[Awin-1][Bwin] +
                         (1-p)*prob[Awin][Bwin-1];

  // Here's our answer.
  return prob[Awin][Bwin];
}
```

Typically, this one's not too fun to trace through by hand...

DP Algorithm for Traveling Salesman Problem

One version of the traveling salesman problem is as follows:

Given a graph of n vertices, determine the minimum cost path to start at a given vertex and travel to each other vertex exactly once, returning to the starting vertex.

In some versions, the starting and ending points are different and fixed, and all other points have to be visited exactly once from start to end.

A standard way to solve these problems is to try all possible orders of visiting the n points, which results in a runtime of O(n!). (If evaluation of a path takes O(n) time, then it would be O(n x n!).) this limits the input size to n = 10 on current (2010ish) machines.

We can solve this problems for a slightly larger input size by realizing that some of the paths we try are "redundant". For example, consider the two paths specified below:

1, 5, 2, 8, 4, 6, 3
1, 2, 8, 4, 5, 6, 3

You'll notice that both paths end in the edge $6 \rightarrow 3$ but visit the vertices 2, 4, 5 and 8 in between 1 and 6 in a different order. If $1 \rightarrow 5 \rightarrow 2 \rightarrow 8 \rightarrow 4 \rightarrow 6$ is less costly than $1 \rightarrow 2 \rightarrow 8 \rightarrow 4 \rightarrow 5 \rightarrow 6$, then we only want to consider building off the first path and have no need to evaluate the second path.

In particular, for each subset of vertices to visit and a fixed ending vertex, we only need to store the minimum cost for visiting that subset, ending at the fixed end vertex.

Let f(S, k) represent the minimum cost for visiting all the vertices in subset S, ending at vertex k. Using this definition, we could build answers to arbitrary queries of this form. Consider calculating, where we assume 1 is always the start vertex: f({1, 2, 3, 4, 5, 6, 8}, 3)

We know we need to end in vertex three, so that our previous subset visited must be:
{1, 2, 4, 5, 6, 8}

Now, with this subset, we could have five possible ending vertices: 2, 4, 5, 6, or 8. Thus, if we want to find the answer to our query, we must simply take the minimum answer over trying each of these five possibilities. Formally, we have:

f({1, 2, 3, 4, 5, 6, 8}, 3) = minimum (f({1, 2, 4, 5, 6, 8}, 2) + edge($2 \rightarrow 3$) ,
f({1, 2, 4, 5, 6, 8}, 4) + edge($4 \rightarrow 3$) ,
f({1, 2, 4, 5, 6, 8}, 5) + edge($5 \rightarrow 3$) ,
f({1, 2, 4, 5, 6, 8}, 6) + edge($6 \rightarrow 3$) ,
f({1, 2, 4, 5, 6, 8}, 8) + edge($8 \rightarrow 3$))

Note that all of the function evaluations are smaller problems of the same nature.

Thus, in code, we can either write this recursively with memoization or with dynamic programming. Either way, we need to declare a data structure to store all possible answers to recursive queries. We'll need an array with two dimensions. One dimension tells us the subset we are referring to while the other dimension tells us the last vertex.

The best way to store an arbitrary subset of n vertices is a bitmask of n bits. Thus, our array that stores all of our answers might look like:

```
int[][] dp = new int[1 << n][n];
```

where n stores the number of vertices in our problem. Note that some elements in the array go unused, since they are non-sensical. (You can't visit the subset {1, 3, 5} ending at vertex 4, for example.)

More specifically, for a given integer, if the i^{th} bit is 1, then element i is in the subset, and if the i^{th} bit is 0, then element i is not in the subset. For example, the value $25 = 11001_2$ would represent a subset of elements 0, 3 and 4. (Typically we store subsets of 0-based sets.)

We can check if element k is in the bitmask represented by the integer item in these two ways (and there are probably more):

```
if (((item >> k) & 1) == 0) …
if ((item & (1 << k)) > 0) …
```

We can check if the set a is a subset of the set b as follows:

```
if ((a & b) == a) …
```

We can check if two sets a and b are disjoint as follows:

```
if ((a & b) == 0) …
```

If we know that item k is in the set subset, then we can remove it from the set as follows:

```
int itemsLeft = subset - (1 << k);
```

In short, once we know how to use our bitwise operators, we can easily make manipulations to sets as necessary for algorithms where we are building answers from various subsets.

Note that when building an answer for a particular subset, you only need to look up answers for subsets of that subset. Also, for any bitmask, all of its subsets are strictly less than it. Thus, if we simply write our dynamic programming algorithm to cycle through each subset in numerical order of bitmask, all of our necessary subcases will be previously solved.

On the following page we'll have the rough structure of code to solve a traveling salesman like problem using the bit mask dynamic programming technique. For each specific problem, how the array is initialize may change as well as other small items.

Pseudocode structure of the bitmask dynamic programming solution to Traveling Salesman

```
int[][] dp = new int[1 << n][n];

// Some initialization of dp, possibly.

for (int mask = 1; mask <= (1<<n); mask++) {
  for (int last = 0; last<n; last++) {

    if (((mask >> last) & 1) == 0) continue;

    int prev = mask - (1 << last);
    dp[mask][last] = Integer.MAX_VALUE;

    // v is the last item visited in prev.
    for (int v=0; v<n; v++) {

      if (((prev >> v) & 1) == 0) continue;

      int curScore   = dp[prev][v] + cost(v, last);
      dp[mask][last] = Math.min(dp[mask][last], curScore);
   }
  }
}
```

Explanation of Pseudocode

1. The first two loops go through the total search space.

2. The first if statement screens out impossible sub-cases, where the last vertex isn't in the subset specified by mask.

3. prev represents the subset of vertices visited BEFORE arriving at vertex last.

4. v represents the possible last locations we could visit in visiting each vertex in perv.

5. The inner most if does the same task as the previous if, screening out impossible cases.

6. The relevant score for this path (curScore) is the sum of the best score of visiting everything in prev, ending in vertex v, added to the cost/edge from vertex v to vertex last, which is where we are trying to end up. I denote the edge weight simply by a cost function that the programmer is free to define. (In some questions, an edge isn't given and some cost has to be calculated from one location to the next.)

7. The last line in the inner-most loop simply updates our answer to be the best of all possibilities where we visit all the vertices in prev followed by traveling to vertex last.