CSE 101 Class Notes

Dynamic Programming

November 20, 2006

General Problem

Dynamic programming (DP) is an efficient approach to solving problems where a recursive, backtracking solution ends up repeatedly solving many common subproblems. Recognizing this, DP systematically solves every possible subproblem, but comes out ahead of backtracking by only solving these problems once. Finding a DP algorithm usually involves the following steps:

1. Start with a simple recursive (backtracking) algorithm.
2. Identify repeated subproblems.
3. Name these subproblems, and store their solutions by name in an array.
4. Find a top-down order of dependencies between these subproblems.
5. Invert this order to get a bottom-up order.

The resulting algorithm will typically have the following form:

1. Initialize array.
2. Fill in base cases.
3. In bottom-up order, solve all subproblems.
4. Return the solution to the final subproblem (i.e. the problem).

Example: Bench Spacing

Input: An array of costs $C[1..n]$ and a maximum separation k.

Output: A set of indices $i_1 < \ldots < i_m$ such that $i_{j-1} + k \geq i_j$.

Goal: Minimize the total cost $\sum_j C[i_j]$.

A backtracking algorithm for this is:

```
BTBC(C[1..n], k)
    if n < k
        return 0
    else
        Cmin <- Inf
        for i = 1 .. k
            Cmin <- min(Cmin, C[i] + BTBC(C[i+k .. n], k))
        return Cmin
```

The running time for this algorithm is

$$T(n) = T(n-1) + \ldots + T(n-k-1) \leq kT(n-1) \leq k^n$$

which is bad. However, by looking at what the recursive algorithm does, we might guess that we could do much better. Specifically, backtracking solves the following subproblems:

```
C[1,n] -> C[2,n] -> C[3,n] ----> C[4,n]
       |  ...    |  ...        |  ...
       |         -> C[k+1,n]  -> C[k+2, n]
       |
       -> C[k,n] -> C[k+1,n]  -> ...
                 |  ...
                 -> C[2k+1,n] ->
```

Clearly, many of them are repeated. Therefore we should try applying DP:

Repeated subproblems: `C[j,n]`.

Names: `A[j] = C[j,n]`.

Base cases: if $n - j + 1 < k$, the cost is 0.

Case order: `C[j,n]` depends on `C[j+1,n] ... C[j+k,n]`. In other words, the subproblems are needed in order of increasing length.

In these terms, our backtracking algorithm is:

```
BTBC(C[j..n], k)
    if n - j + 1 < k
        return 0
    else
        Cmin <- Inf
        for i = j .. n - k
            Cmin <- min(Cmin, C[i] + BTBC(C[i+k .. n], k))
        return Cmin
```

Inverting this to compute the solutions from shortest to longest, the dynamic programming algorithm is

```
DPBC(C[1..n], k)
    A[1 .. n+1]           // array of costs
    for j = n-k .. n+1    // Initialize base cases
        A[j] <- 0
    for j = n-k-1 .. 1
        A[j] <- Inf
        for i = j .. j + k
            A[j] <- min(A[j], A[i] + C[i])
    return A[1]
```

This algorithm runs in time $O(kn)$ and space $O(n)$ (but it can be made to require space $O(k)$ – how?). Note that the algorithm does not currently find the optimal placements, but only their total cost. For a hint on how it can be extended to find the path, see the following example.

Example problem

k = 3,

Posn	1	2	3	4	5	6	7	8	9	10	11	
C	2	7	3	4	3	2	1	4	9	3	5	
A	9	8	7	6	4	4	4	3	3	0	0	0
Next	3	4	6	6	7	7	10	10	10	x	x	

Here `Next[i]` is an array holding the indices of the next item in the best placement starting at `i`. For example, `Next[9] = 10` because the best placement for positions 9 through 11 is to place a bench at 10, which has cost 3.