Lecture 27 — Dynamic Programming II (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Dec 1, 2011

Today:
  - Dynamic Programming

1   Dynamic Programming Continued

1.1   Optimal Binary Search Trees

We have talked about using BSTs for storing an ordered set or table. The cost of finding an element is proportional to the depth of the element in the tree. In a fully balanced BST of size n the average depth of each element is about $\log n$. Now lets say you are given values associated with each element that specify the probability that the element will be accessed—perhaps the word "of" is accessed much more often than "epistemology". The probabilities across the elements must add to 1. The goal is to make it so that the more likely elements are closer to the root and hence the average access cost is reduced. This line of reasoning leads to the following problem:

Definition 1.1. The *optimal binary search tree* problem is given an ordered set of keys S and a probability function $p : S \rightarrow [0 : 1]$, determine:
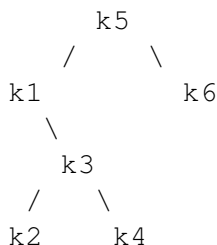
$$\min_{T \in \mathit{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\mathit{Trees}(S)$ is the set of all BSTs on S, and $d(s, T)$ is the depth of the key s in the tree T.

For example we might have the following keys and associated probabilities

key	k_1	k_2	k_3	k_4	k_5	k_6
$p(\text{key})$	1/8	1/32	1/16	1/32	1/4	1/2

Then the tree

```
        k5
      /     \
   k1         k6
     \
       k3
     /    \
   k2      k4
```

has cost $31/16$, which is optimal.

Exercise 1. *Find another tree with equal cost.*

As usual we are interested in solving the problem efficiently. A greedy approach might be to pick the key k with highest probability and put it at the root and then recurse on the two sets less and greater than k. You should convince yourself that this does not work. Instead lets consider a recursive solution. In particular lets try placing every element at the root and then recurse on the subproblems and pick the best of the $|S|$ possibilities. Lets consider how to calculate the cost give the solution to two subproblems. Assume S is a sequence ordered by the keys and we pick location i ($1 \geq i \leq |S|$) as a the root. We can now solve the OSBT problem on the prefix $S_{1,i-1}$ and suffix $S_{i+1,n}$ (the notation $S_{i,j}$ means the locations of S from i to j). Creating a tree with these two solutions as the left and right children of S_i, respectively, leads to the optimal solution given S_i as a root. We therefore might consider adding these two solutions and the cost of the root ($p(S_i)$) to get the cost of this solution. This, however, is wrong. The problem is that by placing the solutions to the prefix and suffix as children of S_i we have increased the depth of each of their keys by 1. However we can adjust for this. In particular for each key s in each subtree, its cost has gone up by $p(s) \cdot 1$. We therefore have:

$$
\begin{aligned}
OBST(S) &= \min_{i \in \langle 1...|S| \rangle} \left(OBST(S_{1,i-1}) + OBST(S_{i+1,|S|}) + p(S_i) + \sum_{s \in S_{i,i-1}} p(s) + \sum_{s \in S_{i+1,|S|}} p(s) \right) \\
&= \sum_{s \in S} p(s) + \min_{i \in \langle 1...|S| \rangle} \left(OBST(S_{1,i-1}) + OBST(S_{i+1,|S|}) \right)
\end{aligned}
$$

When we add the base case this leads to the following recursive definition:

```
1  fun OBST(S) =
2      if |S| = 0 then 0
3      else ∑_{s∈S} p(s) + min_{i∈⟨1...|S|⟩} (OBST(S_{1,i-1}) + OBST(S_{i+1,|S|}))
```

Exercise 2. *How would you return the optimal tree in addition to the cost of the tree?*

As in the examples of subset sum and minimum edit distance, if we execute the recursive program directly $OBST$ it will require exponential work. Again, however, we can take advantage of sharing among the calls to $OBST$. To bound the number of vertices in the corresponding DAG we need to count the number of possible arguments to $OBST$. Note that every argument is a contiguous subsequence from the original sequence S. A sequence of length n has only $n(n+1)/2$ contiguous subsequences since there are n possible ending positions and for the ith end position there are i possible starting positions ($\sum_{i=1}^{n} i = n(n+1)/2$). Therefore the number of possible arguments is at most $O(n^2)$. Furthermore the longest path of vertices in the DAG is at most $O(n)$ since recursion can at most go n levels (each level removes at least one key).

Unlike our previous examples, however, the cost of each vertex in the DAG (each recursive in our code not including the subcalls) is no longer constant. The subsequence computations $S_{i,j}$ can be done in $O(1)$ work each (think about how) but there are $O(|S|)$ of them. Similarly the sum of the $p(s)$ will take $O(|S|)$ work. To determine the span of a vertex we note that the min and sum can be done with a reduce in $O(\log |S|)$ span. Therefore the work of a vertex is $O(|S|) = O(n)$ and the span is $O(\log n)$. Now we simply multiply the number of vertices by the work of each to get the total work, and the longest path of vertices by the span of each vertex to get the span. This give $O(n^3)$ work and $O(n \log n)$ span.

This example of the optimal BST is one of several applications of dynamic programming to what are effectively trying all binary trees and determining an optimal tree given some cost criteria. Another such problem is the matrix chain product problem. In this problem one is given a chain of matrices to be multiplied $(A_1 \times A_2 \times \cdots A_n)$ and wants to determine the cheapest order to execute the multiplies. For example given the sequence of matrices $A \times B \times C$ it can either be ordered as $(A \times B) \times C$ or as $A \times (B \times C)$. If the matrices have sizes $2 \times 10, 10 \times 2$, and 2×10, respectively, it is much cheaper to calculate $(A \times B) \times C$ than $a \times (B \times C)$. The matrix chain product problem can be solved in a very similar structure as the OBST algorithm and with the same cost bounds.

2   Coding Dynamic Programs

So far we have assumed some sort of magic recognized shared subproblems in our recursive codes and avoided recomputation. This sort of magic could actually be fully automated in the functional setting using a technique called hash consing [1], but no languages do this so we are left to our own means. As mentioned in the last lecture there are basically two techniques to code up dynamic programming techniques: the top-down approach and the bottom-up approach.

Top-Down Dynamic Programming

The top-down approach is based on running the recursive code basically as is but generating a mapping from input argument to solution as we proceed. This way when we come across the same argument a second time we can just look up the solution. This is called *memoization*, and the table used to map the arguments to solutions is called a *memo table*. The tricky part of memoization is checking for equality of arguments since the arguments might not be simple values such as integers. Indeed in our examples so far the arguments have all involved sequences. We could compare the whole sequence element by element, but that would be too expensive. You might think that we can do it by comparing "pointers" to the values, but this does not work since the sequences can be created separately so even though the values are equal there could be two copies of the same value in different locations and comparing pointers would say they are not equal and we would fail to recognize that we have already solved this instance.

To get around this problem the top-down approach typically requires that the user create integer surrogates that represent the input values[2]. The property of these integers is that there has to be a 1-to-1 correspondence between the integers and the argument values—therefore if the integers match, the arguments match. The user is responsible for guaranteeing this.

We now cover how this can be done for dynamic program we described for minimum edit distance (MED). In 15-150 you covered memoization but you did it using side effects. Here we will do it in a purely functional way. This requires that we "thread" the table that maps arguments to results through the computation. Although this requires a few extra characters of code, it is safer for parallelism.

Recall that MED takes two sequences and on each recursive call only uses suffixes of the two original sequences. To create integer surrogates we can therefore simply use the length of each suffix. There is clearly a 1-to-1 mapping from these integers to the suffixes. MED can work from either end of the string

---

[1]Basically every distinct value is given a unique ID so that testing for equality even for sequences, sets or other complex types can be done very cheaply even if constructed separately.

[2]other simple types would also work.

Version 0.1

so instead of working front to back and using suffix lengths it could work back to front and use prefix lengths—we do this since it simplifies the indexing. This leads to the following variant of our MED code.

```
1    fun  MED(S, T) = let
2        fun  MED'(i, 0) = i
3            |  MED'(0, j) = j
4            |  MED'(i, j) = case  (S_i = T_j)  of
5                                true ⇒ MED'(i − 1, j − 1)
6                            |  false ⇒ 1 + min(MED'(i, j − 1), MED'(i − 1, j))
7    in
8        MED'(|S|, |T|)
9    end
```

We can now find solutions in our memo table quickly since it can be indexed on i and j. In fact since the arguments range from 0 to the length of the sequence we can actually use an array to store the table values.

Now we can actually implement the memoization. To do this we define a memoization function:

```
1    fun  memo  f  (M, a) =
2        case  find(T, a)  of
3            SOME(v) ⇒ v
4        |  NONE ⇒ let
5                val  (M', v) = f(M, a)
6            in
7                (update(M', a, v), v)
8            end
```

In this function f is the function that is being memoized, M is the memo table, and a is the argument to f. This function simply looks up the value a in the memo table. If it exists, then it returns the corresponding result, otherwise it evaluates the function on the argument and stores the result in the memo table. We can now write MED using memoization.

```
1    fun  MED(S, T) = let
2        fun  MED'(M, (i, 0)) = (M, i)
3            |  MED'(M, (0, j)) = (M, j)
4            |  MED'(M, (i, j)) = case  (S_i = T_j)  of
5                                true ⇒ MED''(M, (i − 1, j − 1))
6                            |  false ⇒ let
7                                    val  (M', v_1) = MED''(M, (i, j − 1))
8                                    val  (M'', v_2) = MED''(M', (i − 1, j))
9                                in  (M'', 1 + min(v_1, v_2))  end
10       and  MED''(M, (i, j)) = memo  MED'  (M, (i, j))
11   in
12       MED'({} , (|S|, |T|))
13   end
```
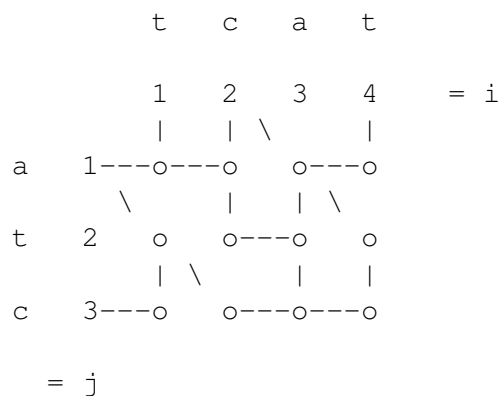
Version 0.1

Note that the memo table M is threaded throughout the computation. In particular every call toe MED not only takes a memo table as an argument but also returns a possibly different memo table as a result. Because of this passing the code is purely functional. The problem with the top-down approach as described, however, is that it is inherently sequential. By threading the memo state we force a total ordering on all calls to MED. It is easy to create a version that uses side effects, as you did in 15-150 or as is typically done in imperative languages. Now the calls to MED can be made in parallel. However, then one has to be very careful since there can be race conditions (parallel threads modifying the same cells). Furthermore if two parallel threads make a call on MED on the same argument then they can and will often both end up doing the work. There are ways around this which are also fully safe—i.e. from the users point of view all calls look completely functional, but they are beyond the scope of this course.

Bottom-Up Dynamic Programming

We will now consider a different technique for implementing dynamic programs. Instead of simulating the recursive structure, which starts at the root of the DAG, it starts at the leaves of the DAG and fills in the results in some order that is consistent with the DAG–i.e. for all edges (u, v) it always calculates the value at a vertex u before working on v. Because of this all values will be already calculated when they are needed.

The simplest way to implement bottom-up dynamic programming is to do some form of systematic traversal of a DAG and it is therefore useful to understand the structure of the DAG. For example lets consider the structure of the DAG for minimum edit distance. In particular lets consider the two strings $S = \mathtt{tcat}$ and $T = \mathtt{atc}$. We can draw the DAG as follows where all the edges go down and to the right.

```
            t   c   a   t

            1   2   3   4      = i
            |   | \     |
    a    1---o---o   o---o
          \      |   | \
    t    2   o   o---o   o
            | \      |   |
    c    3---o   o---o---o

        = j
```

The numbers represent the i and the j for that position in the string. Consider $MED(4, 3)$. The characters S_4 and T_3 are not equal so the recursive calls are to $MED(3, 3)$ and $MED(4, 2)$. This corresponds to the vertex to the left and the one above. Now if we consider $MED(4, 2)$ the characters S_4 and T_2 are equal so the recursive call is to $MED(3, 1)$. This corresponds to the vertex diagonally above and to the left. In fact whenever the characters S_i and T_j are not equal we have edges from directly above and directly to the left, and whenever they are equal we have an edge from the diagonal to the left and above. This tells us quite a bit about the DAG. In particular it tells us that it is safe to process the vertices by first traversing the first row from left to right, and then the second row, and so on. It is also safe to traverse the first column from top to bottom and then the second and so on. In fact it is safe to process the diagonals in the / direction from top left moving to the bottom right. In this case each diagonal can be done in parallel.

In general when applying $MED(S, T)$ we can use an $|T| \times |S|$ array to store all the partial results. We can then process the array either by row, column, or diagonal. This can be coded as follows.

```
1    fun  MED(S, T) = let
2        fun  MED'(M, (i, 0)) = i
3          |  MED'(M, (0, j)) = j
4          |  MED'(M, (i, j)) = case  (S_i = T_j)  of
5                                   true ⇒ M_{i-1,j-1}
6                                 | false ⇒ 1 + min(M_{i,j-1}, M_{i-1,j})

7        fun  diagonals(M, k) =
8          if  (k > |S| + |T|)  then  M
9          else  let
10              val  s = max(0, k - n)
11              val  e = min(k, m)
12              val  M' = M ∪ {(i, k - i) ↦ MED(M, (i, k - i)) : i ∈ {s, ..., e}}
13          in
14              diagonals(M', k + 1)
15          end

16      in
17          diagonals({}, 0)
18      end
19
```

The code uses a table M to store the array entries. In practice an array might do better. Each round of `diagonals` processes one diagonal and updates the table M. We note that the index calculations are a bit tricky (hopefully we got them right). The size of the diagonal grows and then shrinks.

OBST Revisited. We now return to the OBST problem. As with the MED problem we first replace the sequences in the arguments with integers. In particular we describe any subsequence of the original sorted sequence of keys S to be put in the BST by its offset from the start (i, 1-based) and its length l. We then get the following recursive routine.

```
1    fun  OBST(S) = let
2        fun  OBST'(i, l) =
3            if  l = 0  then  0
4            else ∑_{k=0}^{l-1} p(S_{i+k}) + min_{k=0}^{l-1} (OBST'(i, k) + OBST'(i + k + 1, l - k - 1))
5      in
6          OBST(1, |S|)
7      end
```

This modified version can now more easily be used for either the top-down solution using memoization or the bottom-up solution. In the bottom-up solution we note that we can build a table with the columns corresponding to the i and the rows corresponding to the l. Each of them range from 1 to n ($n = |S|$). It would as follows:

```
  1 2 ... n
1            /
2         /
.       /
.     /
n /
```

The table is triangular since as l increases the number of subsequences of that length decreases. This table can be filled up row by row since every row only depends on elements in rows above it. Each row can be done in parallel.