Lecture 24 — Dynamic Programming II (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Margaret Reid-Miller — 12 April 2012

Today:

- Coding Dynamic Programming
- Optimal Binary Search Trees

1 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 does this hash consing automatically, 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 as is, generating implicitly the recursion structure from the root of the DAG down to the leaves. Each time a solution to a smaller instance is found for the first time it generates a mapping from the input argument to its solution. This way when we come across the same argument a second time we can just look up the solution. This process 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, and even though the values are equal, there could be two copies in different locations. 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 1-to-1 correspondence.

Consider how we might use memoization 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

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[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.

do it in a purely functional way, which requires that we "thread" the table that maps arguments to results through the computation. Although this threading requires a few extra characters of code, it is safer for parallelism.

Recall that MED takes two sequences and on each recursive call, it uses suffixes of the two original sequences. To create integer surrogates we can 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 so instead of working front to back and using suffix lengths, it could work back to front and use prefix lengths—we make this switch 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.

To implement the memoization 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.

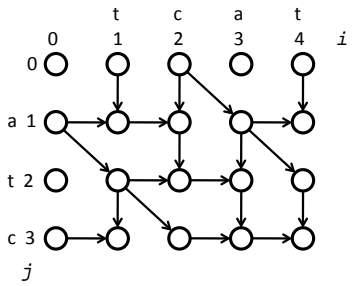
```
1  fun MED(S, T) = let
2    fun MED'(M, (i, 0)) = (M, i)
3      | MED'(M, (0, j)) = (M, j)
4      | MED'(M, (i, j)) =case (S_i = T_j) of
5                            true ⇒ MED''(M, (i − 1, j − 1))
6                          | false ⇒ let
7                              val (M', v_1) = MED''(M, (i, 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
```

Note that the memo table M is threaded throughout the computation. In particular every call to MED not only takes a memo table as an argument, it 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. In this case calls to MED can be made in parallel. However, then one has to be very careful since there can be race conditions (concurrent threads modifying the same cells). Furthermore if two concurrent threads make a call on MED with the same arguments, they can and will often both end up doing the same work. There are ways around this issue 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 the alternate 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 careful scheduling, 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. It is therefore useful to understand the structure of the DAG. For example, consider the structure of the DAG for minimum edit distance. In particular let's consider the two strings $S =$ `tcat` and $T =$ `atc`. We can draw the DAG as follows where all the edges go down and to the right:

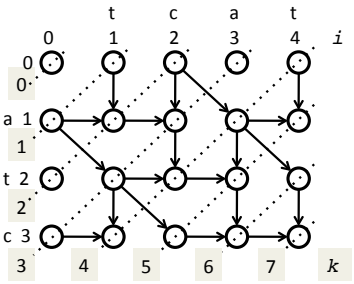


The numbers represent the i and the j for that position in the string. We draw the DAG with the root at the bottom right, so that the vertices are structured the same way we might fill an array indexed by i and j. We Consider $\texttt{MED}(4,3)$. The characters S_4 and T_3 are not equal so the recursive calls are to $\texttt{MED}(3,3)$ and $\texttt{MED}(4,2)$. This corresponds to the vertex to the left and the one above. Now if we consider $\texttt{MED}(4,2)$ the characters S_4 and T_2 are equal so the recursive call is to $\texttt{MED}(3,1)$. 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 column 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 processed in parallel.

In general when applying $\texttt{MED}(S,T)$ we can use an $|T| \times |S|$ array to store all the partial results. We can then fill the array either by row, column, or diagonal. Using diagonals 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 − |T|)
11        val e = min(k, |S|)
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, starting at the leaves at top left. The figure below shows these diagonals indexed by k on the left side and at the bottom. We note that the index calculations are a bit tricky (hopefully we got them right). Notice that the size of the diagonals grows and then shrinks.



2 Optimal Binary Search Trees

We have talked about using BSTs for storing an ordered set or table. The cost of finding an key is proportional to the depth of the key in the tree. In a fully balanced BST of size n the average depth of each key is about $\log n$. Now suppose you have a dictionary where you know probability (or frequency) that each key will be accessed—perhaps the word "of" is accessed much more often than "epistemology". The goal is find a static BST with the lowest overall access cost. That is, make a BST so that the more likely keys are closer to the root and hence the average access cost is reduced. This line of reasoning leads to the following problem:

Definition 2.1. The *optimal binary search tree* problem is given an ordered set of keys S and a probability function $p : S \to [0 : 1]$:

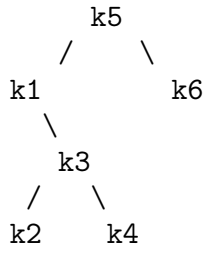
$$\min_{T \in \texttt{Trees}(S)} \left(\sum_{s \in S} d(s, T) \cdot p(s) \right)$$

where $\texttt{Trees}(S)$ is the set of all BSTs on S, and $d(s, T)$ is the depth of the key s in the tree T, where the root has depth 1.

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(key)	1/8	1/32	1/16	1/32	1/4	1/2

Then the tree



has cost 31/16, which is optimal. 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.

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

The brute force solution would be to generate every possible binary search tree, compute their cost, and pick the one with the lowest costs. But the number of such trees is $O(4^n)$ which is prohibitive.

Exercise 2. *Write a recurrence for the total number of distinct binary search trees with n keys.*

Since we finding a binary search tree, one of the keys must be the root of the optimal tree. Suppose S_r is that root. A key observation is that both of its subtrees must be optimal, which is a common property of optimization problems: The optimal solution to a problem contains optimal solutions to subproblems. This *optimal substructure* property is often a clue that either a greedy or dynamic programming algorithm might apply.

Which key should be the root of the optimal tree? 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 let's consider a dynamic programming solution.

Since we cannot know in advance which key should be the root, let's try all of them, recursively finding their optimal subtrees, and then pick the best of the $|S|$ possibilities. As typical, we will not try to find the optimal tree itself initially, but consider only the cost of the tree. Later we will consider how to reconstruct the tree.

With this inductive framework, how should we define the subproblems? For convenience, let's assume the keys in sequence S are in increasing order: $S_1 < S_2 < S_3 < \cdots < S_n$. Now, each recursive call will divide the keys into two contiguous subsequences and eventually we might consider any contiguous subsequence. So the subproblems will be to find the minimum cost BST for a contiguous subsequence $S_i, S_{i+1}, S_{i+2}, \ldots, S_j$ which we denote as $S_{i,j}$.

Let's consider how to calculate the cost give the solution to two subproblems. For subproblem $S_{i,j}$, assume we pick key S_r $(i \le r \le j)$ as a the root. We can now solve the OSBT problem on the prefix $S_{i,r-1}$ and suffix $S_{r+1,i}$. We therefore might consider adding these two solutions and the cost of the root $(p(S_r))$ 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_r we have increased the depth of each of their keys by 1. Let T be the tree on the keys $S_{i,j}$ with root S_r, and T_L, T_R be its left and right subtrees. We therefore have:

$$\begin{aligned}
\texttt{Cost}(T) &= \sum_{s \in T} d(s,T) \cdot p(s) \\
&= p(S_r) + \sum_{s \in T_L} (d(s,T_L)+1) \cdot p(s) + \sum_{s \in T_R} (d(s,T_R)+1) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \sum_{s \in T_L} d(s,T_L) \cdot p(s) + \sum_{s \in T_R} d(s,T_R) \cdot p(s) \\
&= \sum_{s \in T} p(s) + \texttt{Cost}(T_L) + \texttt{Cost}(T_R)
\end{aligned}$$

That is, the cost of a subtree T the probability of accessing the root (i.e., the total probability of accessing the keys in the subtree) plus the cost of searching its left subtree and the cost of searching its right subtree. 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 3. *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 i^{th} 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.

2.1 Coding optimal BST

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.