Dynamic Programming II

Dynamic Programming in Trees

Trees provide another structure where we can frequently bound the number of subproblems. Consider a rooted tree T with n vertices. How many subtrees does T have? Since each subtree consists of all the descendants of a particular vertex in T, there are as many subtrees as number of vertices in T. i.e. n in all. This is the basis of the following dynamic programming algorithm:

Maximum Independent set: Given a graph $G(V, E)$, the maximum independent set in G is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G, and such that I is as large as possible. It is immensely difficult to solve this problem in general. Indeed, it is one of the NP-complete problems (a class of problems we will talk about later in the semester).

We will show that if the given graph $G(V, E)$ is a tree, then using dynamic programming, the maximum independent set problem can be solved in linear time. Here is how the algorithm proceeds:

Root the tree at an arbitrary vertex. Now each vertex defines a subtree (the one hanging from it). Dynamic programming proceeds, as always, from smaller to larger subproblems — that is to say, bootom-up in the rooted tree. Suppose that we know the size of the largest independent set of all subtrees below a node j. What is the maximum independent set in the subtree hanging from j? Two cases: either j is in the maximum independent set, or it is not. If it is not, then the maximum independent set is simply *the union of the maximum independent sets of the subtrees of the children of j*. If j is in the maximum independent set, then the maximum independent set consists of j, *plus the union of the maximum independent sets of the subtrees of the grandchildren of j*.

The recursive equation is now easy to write: Let $I(j)$ be the size of the maximum independent set in the subtree rooted at vertex j.

$$I(j) := \max\{\sum_{k \text{ child of } j} I(k), 1 + \sum_{k \text{ grandchild of } j} I(k)\}.$$

It might seem at first that the complexity of this algorithm is $O(n^2)$ — since there are n entries to be filled in, and the maximum time to compute an entry can be as large as $\Theta(n)$ (since a vertex j might have $\Theta(n)$ grandchildren). However, there is a clever argument showing that the total number of steps is only $O(n)$: For each vertex, the algorithm only looks at its children and its grandchildren; hence, each vertex j is looked at only three times: when the algorithm is processing vertex j, when it is processing $j's$ parent, and when it is processing $j's$ grandparent. Since each vertex is looked at only a constant number of times, the total number of steps is $O(n)$.

2. Travelling Salesman Problem:

The traveling salesman problem. Suppose that you are given n cities and the distances d_{ij} between any two cities; you wish to find the shortest tour that takes you from your home city to all cities and back.

Naturally, the TSP can be solved in time $O(n!)$, by enumerating all tours —but this is very impractical. Since the TSP is one of the NP-complete problems, we have little hope of developing a polynomial-time algorithm for it. Dynamic programming gives an algorithm of complexity $O(n^2 2^n)$ —exponential, but much faster than $n!$.

We define the following subproblem: Let S be a subset of the cities containing 1 and at least one other city, and let j be a city in S other than one. Define $C(S, j)$ to be *the shortest path that starts from* 1*, visits all nodes in S, and ends up in j*. The program now writes itself:

for all j do $C(\{1,j\},j) := d_{1j}$
for $s := 3$ to n do (the size of the subsets considered this round)
for all subsets S of $\{1,\ldots,n\}$ of size n and containing 1 do
for all $j \in S, j \neq 1$ do
$\{C(S,j) := \min_{i\neq j, i\in S}[C(S-\{j\},i) + d_{ij}]\}$
opt:= $\min_{j\neq 1}[C(\{1,2,\ldots,n\},j) + d_{j1}$.

As always, we can also recover the optimum tour by remembering the i's that achieve the minima. The complexity is $O(n^2 2^n)$: The table has $n2^n$ entries (one per set and city), and it takes about n time to fill each entry.

Chain matrix multiplication

Suppose next that you want to multiply four matrices $A \times B \times C \times D$ of dimensions 40×20, 20×300, 300×10, and 10×100, respectively. Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications (a good enough estimate of the running time).

To multiply these matrices as $(((A\times B)\times C)\times D)$ takes $40\cdot 20\cdot 300 + 40\cdot 300\cdot 10 + 40\cdot 10\cdot 100 = 380,000$. A more clever way would be to multiply them as $(A \times ((B \times C) \times D))$, with total cost $20\cdot 300\cdot 10 + 20\cdot 10\cdot 100 + 40\cdot 20\cdot 100 = 160,000$. An even better order would be $((A \times ((B \times C)) \times D)$ with total cost $20\cdot 300\cdot 10 + 40\cdot 20\cdot 10 + 40\cdot 10\cdot 100 = 108,000$. Among the five possible orders (the five possible binary trees with four leaves) this latter method is the best.

How can we automatically pick the best among all possible orders for multiplying n given matrices? Exhaustively examining all binary trees is impractical: There are $C(n) = \frac{1}{n}\binom{2n-2}{n-1} \approx \frac{4^n}{n\sqrt{n}}$ such trees ($C(n)$ is called the *Catalan* number of n). Naturally enough, dynamic programming is the answer.

Suppose that the matrices are $A_1 \times A_2 \times \cdots \times A_n$, with dimensions, respectively, $m_0 \times m_1, m_1 \times m_2, \ldots m_{n-1} \times m_n$. Define a *subproblem* (remember, this is the most crucial and nontrivial step in the design of a dynamic programming algorithm the rest is usually automatic) to be to multiply the matrices $A_i \times \cdots \times A_j$, and let $M(i,j)$ be the optimum number of multiplications for doing so. Naturally, $M(i,i) = 0$, since it takes no effort to multiply a chain consisting of one matrix. The recursive equation is

$$M(i,j) = \min_{i\leq k<j}[M(i,k) + M(k+1,j) + m_{i-1}\cdot m_k \cdot m_j].$$

Naturally, $M(i,i) = 0$, since it takes no effort to multiply a chain consisting of the ith matrix. This equation defines the program and its complexity —$O(n^3)$.

for $i := 1$ to n do $M(i,i) := 0$
for $d := 1$ to $n-1$ do
for $i := 1$ to $n-d$ do
$\{j = i + d, M(i,j) = \infty$, best$(i,j)$:=nil
for $k := i$ to $j-1$ do
if $M(i,j) > M(i,k) + M(k+1,j) + m_{i-1}\cdot m_k \cdot m_j$ then
$\{M(i,j) := M(i,k) + M(k+1,j) + m_{i-1}\cdot m_k\cdot m_j$, best$(i,j) := k\}\}$

As usual, improvements are possible (in this case, down to $O(n\log n)$).

Run this algorithm in the simple example of four matrices givem to verify that the claimed order is the best!

Optimum binary search trees. Suppose that you know the frequency with which keywords appear in programs in a language:

begin	5%
do	40%
else	8%
end	4%
if	10%
then	10%
while	23%

We want to organize them in a binary search tree, such that the keyword in the root is lexicographically bigger than all keywords in its left subtree and smaller than all keywords in its right subtree (and the same for all other nodes) as shown below:

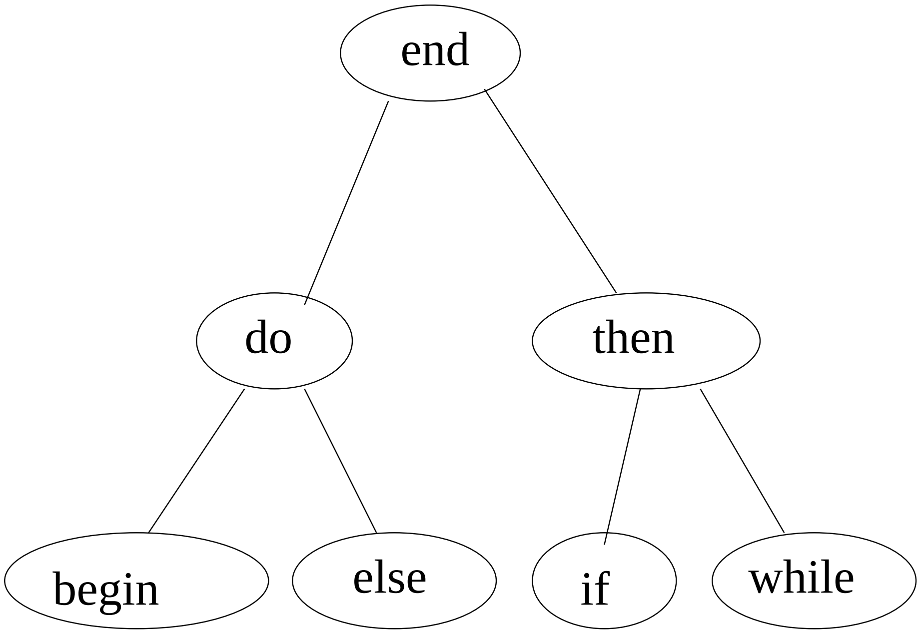

Figure 1: Binary search tree

The cost of this tree is 2.42 —that is to say, the average keyword finds its position after 2.42 comparisons (1 comparison with probability 4% for 'end', 2 comparisons with probability $40 + 10 = 50\%$ for 'do' and 'then', and so on. The optimum binary search tree is shown below; it uses 2.18 comparisons on the average, and can be found by dynamic programming.

Let p_i be the probability of the ith keyword, and define $P_{ij} = \sum_{k=i}^{j} p_k$. Once more, let $T(i, j)$ be the average number of comparisons in the optimum tree for keywords i through j. It is easy to see that

$$T(i, j) = \min_{i \leq r \leq j} [T(i, r-1) + T(r+1, j) + P_{ij}].$$

Here r is the root of the optimum tree (to be determined by minimization). The equation states that any keyword in the range $i \ldots j$ would cost us one for a comparison with the root, and, if it is not the root, $T(i, r-1) + T(r+1, j)$ for the comparisons performed in the left or right subtrees. As always, the program is easy to write now (the initialization here is $T(i, i-1) = 0$).

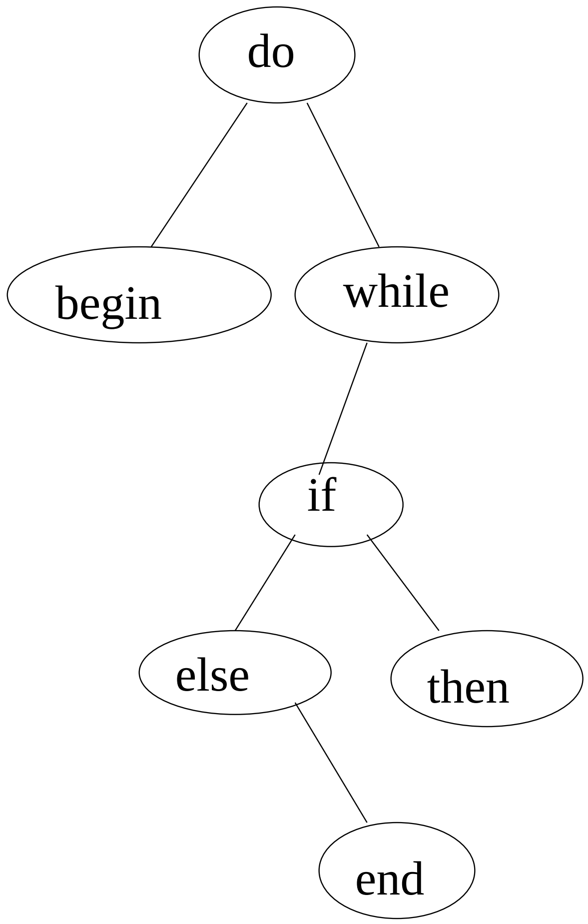

Figure 2: The optimum binary search tree