

## Counting step-lines with area restriction

We have a graph with nodes at points  $(i, j)$  such that  $i, j$  integer,  $0 \leq i \leq m$ ,  $0 \leq j \leq n$ . We have edges of the form  $(i, j) \rightarrow (i + 1, j)$  and  $(i, j) \rightarrow (i, j + 1)$ .

We want to find the number of paths from  $(0, 0)$  to  $(m, n)$  such that the area under the path is equal to  $A$ . We define this quantity to be  $N(m, n, A)$  and we can easily find rules that allow to compute it:

for  $A < 0$  and  $A > mn$  we have  $N(m, n, A) = 0$  (no such paths);

for  $A = 0$  and  $A = mn$  we have  $N(m, n, A) = 1$  (exactly one such path);

this covers the cases when  $m = 0$  or  $n = 0$ ;

when we want to find  $N(m + 1, n + 1, A)$ , we have two cases:

the last step is vertical,  $(m + 1, n) \rightarrow (m + 1, n + 1)$ , and the area under the path did not change,

this contributes  $N(m + 1, n, A)$  possible paths;

the last step is horizontal,  $(m, n + 1) \rightarrow (m + 1, n + 1)$ , and the area under the path increased by  $n$ ,

this contributes  $N(m, n + 1, A - n)$  possible paths;

so we get  $N(m + 1, n + 1, A) = N(m + 1, n, A) + N(m, n + 1, A - n)$

These rules do not define an efficient recursive program, but we the number of distinct recursive calls is limited: there are three arguments, and when we make a call, the arguments in the call are the same or smaller. Thus the number of calls made by  $N(m, n, A)$  is  $O(mnA)$ . If  $n \geq m$ , only  $A < n^2$  require computation so we have  $O(n^4)$  subproblems.

In practice, this leads to a running time that can be reasonable for  $n$  in the range of several hundreds. If we can perform  $10^9$  operations per second, and we can operate for  $10^3$  seconds we can have  $(10^3)^4$  operations. The operations are not so elementary here, but we just change our estimate of the running time from 1000 seconds to, say, several days. But we cannot use  $10^{12}$  memory, so we need to think how to use less.

## Transitive closure and Warshall algorithm

We have an  $n \times n$  zero-one matrix  $A$  such that  $(i, j)$  is an edge in our graph if and only if  $A_{ij} = 1$  — the adjacency matrix of our graph.

We want to compute the transitive closure, which is zero-one matrix  $R$  such that  $R_{ij} = 1$  if and only if there is a path from  $i$  to  $j$ .

A recursive solution is obtained if we define extra matrices, such that  $R_{ij}^k = 1$  if and only if there is a path from  $i$  to  $j$  in which intermediate nodes are in the set  $\{0, \dots, k-1\}$ . In this way,  $R_{ij}^0 = A_{ij} \parallel [i \equiv j]$ , and  $R_{ij}^n = R_{ij}$ . So we know how to obtain the initial matrix  $R^0$ , and  $R^n$  is the matrix we need.

The recurrence is:

$$R_{ij}^{k+1} = R_{ij}^k \parallel R_{ik}^k \ \&\& \ R_{kj}^k$$

This gives  $O(n^3)$  algorithm: compute  $R^0$  in  $O(n^2)$  steps, and then for  $k = 1, \dots, n$  compute  $R^k$  using the recurrence.

We can save memory with a very simple observation: when we compute  $R^{k+1}$  from  $R^k$ , we can tolerate if entries of  $R^k$  are larger if we know that they are not larger than the entries of  $R$ . How is the recurrence working: a path from  $i$  to  $j$  with intermediate nodes in  $\{0, \dots, k\}$  exists if either there is a path that does not use  $k$  as an intermediate node, or there is a path from  $i$  to  $k$  and a path from  $k$  to  $j$ , and these two paths do not need  $k$  as an intermediate node.

If  $R^k$  is “too large”, we allow, perhaps, more intermediate nodes on paths from  $i$  to  $k$  and from  $k$  to  $j$ , but such a pair of paths still shows a valid path from  $i$  to  $j$ , so no harm is done. This idea translates to the following pseudo-code:

```
// initial matrix  $R^0$ 
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        R[i][j] = A[i][j];
for (i = 0; i < n; i++)
    R[i][i] = 1;

for (k = 0; k < n; k++)
    // compute matrix that is at least  $R^{k+1}$ 
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            R[i][j] |= R[i][k] && R[k][j];
```

## All-pairs shortest paths and Floyd algorithm

In transitive closure we learn whether a path exists or not. We can ask for more: if a path exists, what is the length of the shortest path.

We will assume that  $C_{ij}$  is the length of the edge  $(i, j)$ , and if there is no such edge,  $C_{ij} = \infty$  where  $\infty$  is a very large number.

Identical ideas apply, except for a new convention: we need some large number  $\infty$  so  $R_{ij} = \infty$  means we have no path from  $i$  to  $j$ . Then it is the same operation to find a shorter path than before and to find a path when we did not know any.

We can restrict paths to have intermediate nodes  $\{0, \dots, k-1\}$ , or more precisely, compute paths which are at least as good as the best paths that have that restriction.

We can easily find best paths that have no intermediate nodes.

We can easily check if a new intermediate node  $k$  allows to create a path that is shorter than before: we check  $i \rightarrow k \rightarrow j$ . This ideas translate to the following pseudo-code:

```
// initial matrix  $R^0$ 
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        R[i][j] = C[i][j];
for (i = 0; i < n; i++)
    R[i][i] = 0;

for (k = 0; k < n; k++)
    // compute matrix that is at most  $R^{k+1}$ 
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (d = R[i][k]+R[k][j], d < R[i][j])
                R[i][j] = d;
```

## Optimal binary search trees

We have an ordered set of keys  $a_0, \dots, a_{n-1}$  and we want to store a set of objects with those keys. The set is fixed and we will have a stream of requests “find the object with the least key above  $x$ ” (or the largest key above  $x$ ), and key  $a_i$  provides the answer with probability  $p_i$ .

We want to construct a binary search tree that will minimize the average length of the search. A search terminates at a node of key  $a_i$  with probability  $p_i$ , and under that condition, it has the length  $d(i)$ , which is the number of nodes on the path from the root to the node that stores  $a_i$ . Thus the average cost is

$$\sum_{i=0}^{n-1} d(i)p_i = \text{cost}(0, n)$$

Designing the binary search tree for  $a_0, \dots, a_{n-1}$  can be viewed recursively:

pick the root  $a_r$

design a tree for  $a_0, \dots, a_{r-1}$

if  $d'(i)$  is defined in that tree, the true value is  $d(i) = d'(i) + 1$

design a tree for  $a_{r-1}, \dots, a_r$

if  $d'(i)$  is defined in that tree, the true value is  $d(i) = d'(i) + 1$

What do the subtrees contribute to  $\text{cost}(0, n)$ ?

The left subtree contributes  $\text{cost}(0, r) + \sum_{i=0}^{r-1} p_i$ . The right subtree contributes  $\text{cost}(r+1, n) + \sum_{i=r+1}^{n-1} p_i$ . The root contributes  $p_r$ .

Together, we get  $\text{cost}(0, r) + \text{cost}(r+1, n) + \sum_{i=0}^{n-1} p_i$ .

Crucial observation: the design of the left and right subtree is independent so we can try to compute it before. General conditional recurrence:

$\text{cost}(a, b) = \text{cost}(a, r) + \text{cost}(r+1, b) + \sum_{i=a}^{b-1} p_i$ . But how to choose  $r$ ?

Choose the best one!

Correct recurrence:

$$cost(a, b) = \begin{cases} 0 & \text{if } a = b \\ \sum_{i=a}^{b-1} p_i + \min_{r=a}^{b-1} cost(a, r) + cost(r + 1, b) & \text{otherwise} \end{cases}$$