Dynamic Programming
Master MOSIG - Algorithms and Program Design

Marc Tchiboukdjian - Denis Trystram
(from F. Wagner's lecture notes)

Course

Objectives: Efficiently compute a recursive formula

To Remember

Dynamic programming is achieved while following different steps.

1. Write down the recursive formula corresponding to your needs. The formula might contain some conditionals.
2. Check the computations are finishing when using this formula (proof by induction).
3. Compute the complexity for the raw recursive computation. Are any function calls taking place twice ?
4. Write the function in your favorite programming language.
5. Add a table storing results and modify your program to avoid duplicate computations.
6. Write the dependency graph of the calls. Find vectors describing these dependencies. Follow these vectors to write a *sequential* program computing your function.
7. Optimize your sequential program for space.

Example

Naive recursive version

We consider as example the following formula : $f(x, y) = f(x-1, y) + f(x, y-1)$ with $f(0, y) = 1$ and $f(x, 0) = 1$ which enables us to compute elements from the pascal triangle.

Clearly the corresponding C program can be written as follows:

```
int f (int x, int y) {
        int result;
        if ((x==0)||(y==0))      result = 1;
        else result = f(x-1, y) + f(x, y-1);
        return result;
}
```

Figure 1 displays the call tree of function f for parameters 3,3. We can clearly see that some calls are redundant and the computing cost of f is exponential.

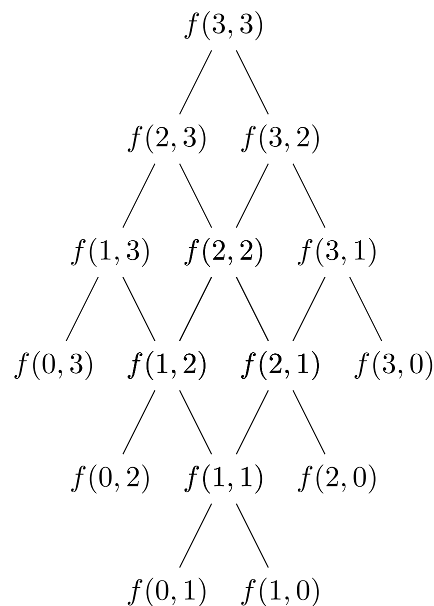

Figure 1: Call-tree for $f(3,3)$

Optimized recursive version

In order to improve things we are going to remove redundant calculations by caching temporary results. To achieve this we allocate an array (the cache) big enough to contain all results for all different inputs of the f function. We then modify the function f in two steps : first we ensure that results are stored in the cache before all return points of the function and then we ensure that no duplicate computations take place by checking the content of the cache at entry point of the function. Of course the cache needs to be initialized with values which will never be mistaken with real results. In this example we can simply initialize all cache entries with -1 since any result will always be positive.

```
1 int cache[N][N];
2 int f_optim (int x, int y) {
3         if (cache[x][y] != -1) return cache[x][y];
4         int result;
5         if ((x==0)||(y==0))      result = 1;
6         else result = f(x-1, y) + f(x, y-1);
7         cache[x][y] = result;
8         return result;
9 }
```

Figure 2: Recursive version with caching

Figure 2 shows the modified f function. We can see the 2 modifications, line 3 and line 7. At this point, we compute the cost of execution of our algorithm when computing $f(n_1, n_2)$. Clearly this cost C is equal to $\sum_{\text{reached values of } x} \sum_{\text{reached values of } y} c(x,y)$ where $c(x,y)$ is the cost for computing the cache entry associated to x, y. In our case, $c(x,y) = O(1)$ since f contains no loop. This means that $C = O(1) \times \sum_x \sum_y 1 \leq (n_1 + 1) \times (n_2 + 1) \times O(1) = O(n_1 n_2)$.

Sequential version

Current cost is rather nice but we would like to optimize a bit more by getting rid of recursive calls. We start by making a small drawing of the cache together with the dependencies between the different results.

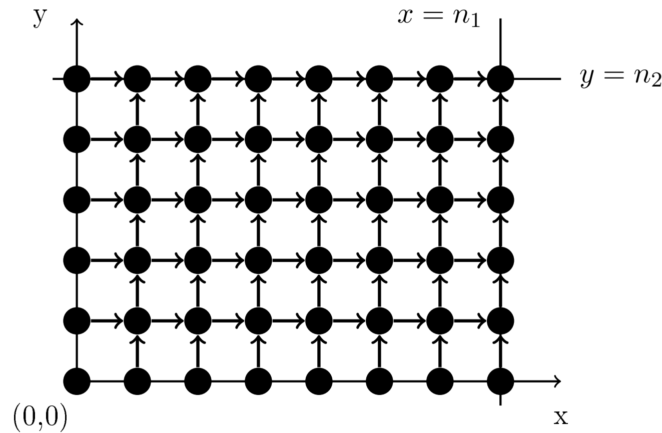

Figure 3: Dependence graph

As Figure 3 shows, dependencies follow two directions : we have vertical

vectors and horizontal vectors. These dependencies mean that cache[i][j] cannot be computed before cache[i−1][j] and cache[i][j−1]. We choose as a basis of our space the two vectors $(0, 1)$ and $(1, 0)$ which translates into two nested loops, one loop on x and one loop on y. We have two possibilities because we can choose which one is the inner-most and which one is the outer-most loop. We choose here to iterate on y in the inner-loop.

```
10 int f_seq(int n1, int n2) {
11         for(int x = 0 ; x < n1 ; x++)
12                 for(int y = 0 ; y <n2 ; y++) {
13                         int result;
14                         if ((x==0)||(y==0))       result = 1;
15                         else result = cache[x-1][y] + cache[x][y-1];
16                         cache[x][y] = result;
17                 }
18         return cache[n1][n2];
19 }
```

Figure 4: Sequential code

Figure 4 shows the sequential code obtained. You can see that the body of the two nested loops is almost identical to the body of the recursive version of Figure 2. The only difference is that functions calls have been replaced by reading in the cache. Computing the cost of the sequential algorithm is direct ($O(n_1 n_2)$).

Memory optimization

The last step of optimisation which can be achieved is by reducing the amount of memory used.

Figure 5 shows a decomposition of the cache into diagonal lines (12 in this particular example). It is clear than any cache value belonging to diagonal d_i can be computed if all values of diagonal d_{i-1} are available. Thus, if the outer-loop we choose iterates on the diagonals and the inner loop iterates inside each diagonal we can compute the final result while only storing two diagonals in memory. Thus, memory consumption can be reduced from $O(n^2)$ to $O(n)$. We can see from the figure that inner-loop vector will be $(-1, 1)$ while for outer loop vector it will be $(1, 1)$.

You can find below the source code for this last function. We decompose the iterations on the diagonals (on our example first from d_0 to d_7 and then from d_8 to d_12) into two loops. As you can see it is very difficult to write such a function without a drawing of the cache guiding your work.

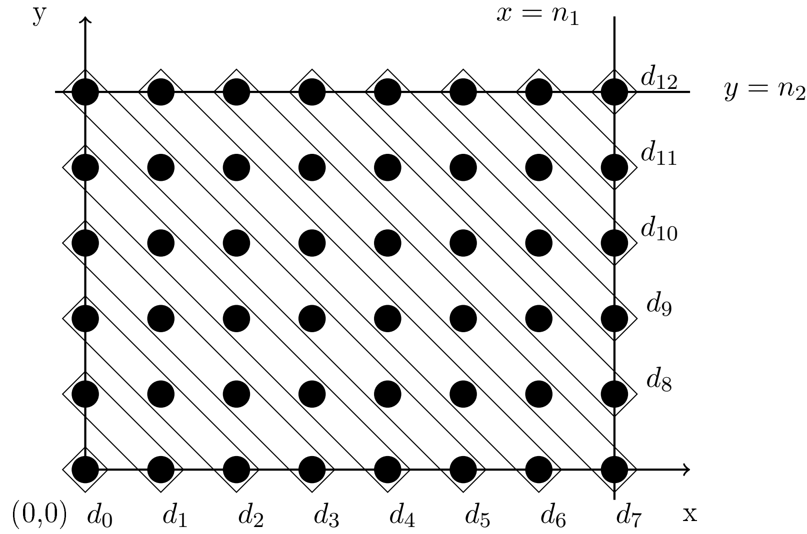

Figure 5: Iterating on diagonals

```
int diagonal1[N];
int diagonal2[N];
int f_horribilis(int n1, int n2) {
  int x, y;
  int current_cache = 0;
  for(int d = 0 ; d <= n1 ; d++) {
    y = 0;
    for(int x = d ; x >= 0 ; x--) {
      int result;
        if ((x==0)||(y==0)) result = 1;
      else {
        if (current_cache == 0) result = diagonal2[d-x-1] + diagonal2[d-x];
        else result = diagonal1[d-x-1] + diagonal1[d-x];
      }
      if (current_cache == 0) diagonal1[d-x] = result;
      else diagonal2[d-x] = result;
      y++;
    }
    current_cache = (current_cache + 1) % 2;
  }
  for(int d = 1 ; d <= n2 ; d++) {
    x = n1;
    for(int y = d ; y <= n2 ; y++) {
      int result;
        if ((x==0)||(y==0)) result = 1;
      else {
        if (current_cache == 0) result = diagonal2[y-d] + diagonal2[y-d+1];
```

```
        else result = diagonal1[y-d] + diagonal1[y-d+1];
      }
      if (current_cache == 0) diagonal1[y-d] = result;
      else diagonal2[y-d] = result;
      x--;
    }
    current_cache = (current_cache + 1) % 2;
  }
  if (current_cache == 0) return diagonal2[0];
  else return diagonal1[0];
}
```

Exercises

Distance between two strings

In order to compare strings efficiently, we define a notion of distance between two strings as the minimum number of modifications needed to go from string A to string B. 3 types of modifications are possible:

- adding a letter at beginning of string A
- removing a letter from beginning of string A
- changing the first letter of string A

To each operation is associated a different cost : $a(l)$ to add the letter l, $r(l)$ to remove the letter l and $c(l_1, l_2)$ to change l_1 into l_2.

We define the distance function as follows:

$$d(lX, lY) = d(X, Y)$$

$$d(lX, mY) = \min(a(m) + d(lX, Y), r(l) + d(X, mY), c(l, m) + d(X, Y))$$

$$d(\epsilon, \epsilon) = 0$$

1. write a recursive function computing d
2. optimize this recursive function by adding a cache
3. write a sequential version
4. is it possible to optimize the memory use ?

Floyd-Warshall All Pairs Shortest Path

We consider the problem of finding shortest paths between all pairs of vertices in a graph. We are given a weighted directed graph $G = (V, E)$ using the adjacency matrix representation and a weight function $w : E \rightarrow \mathbb{R}$ that maps edges to real-valued weights. We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. We want the output in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v. Negative-weight edges are allowed but we assume that the input graph contains no negative weight cycles.

Assume the vertices of G are $V = \{1, \ldots, n\}$. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, \ldots, k\}$. Consider a shortest path p from i to j with all intermediate vertices in $\{1, \ldots, k\}$. If k is not an intermediate vertex, then all intermediate vertices of p are in $\{1, \ldots, k-1\}$. If k is an intermediate vertex, p can be decomposed into two paths p_1 from i to k and p_2 from k to j where all intermediate vertices of p_1 are in $\{1, \ldots, k-1\}$ and all intermediate vertices of p_2 are in $\{1, \ldots, k-1\}$.

1. What are $d_{ij}^{(0)}$ and $d_{ij}^{(n)}$?
2. Write a recursive formula of $d_{ij}^{(k)}$ using $d^{(k-1)}$.
3. Write a sequential function that compute $d^{(n)}$.
4. What is the running time of this algorithm?
5. How much memory do you use?