Internal Use Only (非公開)

TR-I-0295

Easier C programming
Dynamic programming

Yves Lepage

November 1992

Abstract

This report describes the dynamic programming facility developed for our programming work on distances among computational linguistics objects. We propose a set of functions to implement a simple database. The search and retrieve method uses AVL trees. Using this simple database structure, any function with two arguments of any type can be declared as a "dynamic" function. This is done using predefined macros. As an example, we applied the dynamic programming facility to the implementation of a general distance.

Keywords

Programming in C, dynamic programming, distance calculation.

©ATR Interpreting Telephony Research Laboratories

Contents

List of Figures

Introduction

This report describes the implementation of a dynamic programming facility. Dynamic programming is a technique with a very simple principle: *storing intermediate results avoids recomputing them several times.* Implementation requires a definite data structure for the storage of the results and good retrieval techniques.

For our work on distances among objects for natural language processing, we have found it possible (and elegant) to implement a general algorithm for distance calculation using this technique.

1 Dynamic programming

This section explains the principle of dynamic programming. It is illustrated on the computation of a mathematical function, and then on the computation of distances, which constituted the reason for this implementation.

1.1 Principle

The goal of the dynamic programming method is to reduce the computation time of some functions. Of course, this advantage is balanced by an increased use of space. The basic idea is to store intermediate results in a database. These intermediate results are solutions of subproblems similar to the general problem[1]. When a new computation is required, the database is first consulted. If the computation has already been performed and its result has been stored in the database, no further computation is required; else the computation is performed and its result is stored in the database.

Applying dynamic programming is interesting if the following condition is verified on the average:

> *the retrieval time from the database is less than the computation time of the function.*

[1] A description of dynamic programming was announced in [Knuth 73, p. 435] for Chapter 7. Unfortunately Volume 4, which would have included Chapter 7, seems to have never been published!

1.2 First example: combinations

In order to understand the dynamic programming technique, suppose we want to compute the number of combinations[2] for two non-negative integers. This number is given by this well-known following formula (for $n > m$):

$$C_n^m = \frac{n!}{m!(n-m)!} \tag{1}$$

Suppose the computation of $n!$ is performed using the recursive definition of *factorial*:

$$n! = n.(n-1)! \tag{2}$$

With this definition, since $n > m$ and $n > (n-m)$, computing $n!$ entails computing $m!$ and $(n-m)!$. Thus, in the computation of C_n^m, these values will be computed twice and, as a consequence, the number of calls to the function *factorial* will be $2n+3$.

Now, if intermediate results are stored, $m!$ and $(n-m)!$ are computed only once and the number of calls to the function *factorial* is $n+1$.

If retrieving intermediate results is of less cost than computing them again, then this technique decreases the computation time.

[2]The number of combinations of two integers n and m is the number of possible selections of m objects among n, without any regard to the order.

1.3 Second example: string distances

In this section, we show the advantage of the dynamic programming method for our intended application. This is the computation of distances between strings. We compare the asymptotic behaviour of two algorithms for the calculation of the distance: the first one is a direct application of the recursive definition, and the second one is the application of dynamic programming to this definition. This comparison shows that dynamic programming should drastically reduce computation time.

1.3.1 Definition

The dynamic programming technique is typically used for functions computed on arrays. For such functions, the final value of the function is the value of the rightmost lowermost cell in the array, the value of a cell in the array being obtained from the values of the preceding cells. Let A be the name of the array. Then for all i, j Equation 3 holds for cell $A[i,j]$ where f is some function:

$$A[i,j] = f(A[i-1,j], A[i-1,j-1], A[i,j-1]) \tag{3}$$

The Wagner and Fischer distance [Wagner & Fischer 74] between two strings c and c' is a function computed on an array. Each cell $A[i,j]$ contains the value of the distance between the substrings $c[1--i]$ and $c'[1--j]$. If we note the length of the string c as $\text{len}(c)$, the distance between strings c and c' is given by $A[\text{len}(c), \text{len}(c')]$, *i.e.* the bottom right cell in the array. For the Wagner and Fischer distance, the following formula gives the value of each cell:

$$\begin{array}{lll} A[i,j] = \min(& A[i-1,j-1] & + \text{dist}(c[i], c'[j]), \\ & A[i-1,j] & + \text{weight}(c[i]), \\ & A[i,j-1] & + \text{weight}(c'[j])) \end{array} \tag{4}$$

1.3.2 Behaviour of a direct algorithm

An implementation of the Wagner and Fischer distance making direct use of this recusive definition will compute the value of each cell many times. More precisely, $A[i,j]$ is recomputed for all $i' \geq i$ and $j' \geq j$ when computing $A[i',j']$. Let us call $f(n,m)$ the number of accesses to cells needed to compute $A[n,m]$.

When either n or m is zero, the algorithm is linear in the length of the non-zero argument. But this case has no significance.

For $m,n \geq 1$, we have shown that there exist a lower bound and an upper bound for the function f. More precisely, f can be framed in the following way for $m,n \geq 1$ (see Annex A for a proof):

$$\forall(n,m),\quad mn\left(\frac{1+\sqrt{2}}{2}\right)^{m+n} \leq f(m,n) \leq mn\left(\sqrt{6}\right)^{m+n}$$

This result shows that, in the general case, the asymptotic behaviour of this algorithm is exponential in the sum of the lengths of the strings.

1.3.3 Behaviour of a dynamic programming algorithm

If dynamic programming is used, the value of each cell is computed only once and is accessed twice[3]. This is explained by the fact that each cell $A[i,j]$ enters in the computation of its three neighbours $A[i+1,j]$, $A[i+1,j+1]$, $A[i,j+1]$. The first time it is needed, it is computed and stored; the two next times, it will be retrieved from the database, according to the principle of dynamic programming. Hence, if we note as f' the function denoting the number of computations really needed in the dynamic programming algorithm, we have:

$$Amn \leq f'(m,n)$$

where A is a constant. If we compare the two functions f and f', we see that the ratio:

$$\frac{A}{\left(\sqrt{6}\right)^{m+n}} \leq \frac{f'(m,n)}{f(m,n)}$$

decreases as an exponential function. Clearly, the dynamic programming method entails a dramatic reduction of time during computation.

[3]For simplification, we neglect the case of the cells on the rightmost and lowermost edges of the array, which are computed once and not accessed again.

1.3.4 Discussion

One could argue that the original algorithm given by Wagner and Fischer for the computation of their distance is equivalent to the algorithm with dynamic programming: its asymptotic behaviour is $\mathcal{O}(\text{len}(c) \times \text{len}(c'))$ and, as the whole array is stored, the space required is $\text{len}(c) \times \text{len}(c')$. Moreover, we were able to improve this algorithm by reducing the space required to $\min(\text{len}(c), \text{len}(c')) + 1$. In fact, this implementation was the first one we used.

But there are good arguments for preferring dynamic programming. They are twofold:

- simplicity. The dynamic programming facility allows one to keep the elegance of recursive definitions. By contrast, dedicated algorithms often depart considerably from the formal definitions of the functions they are supposed to compute;
- efficiency of implementation. It is not always obvious to design an algorithm like the one for the Wagner and Fischer distance, and to improve it. The use of the dynamic programming facility delivers an efficient program, without any effort.

These qualities are ideal qualities from the programmer's point of view: the source code is easily readible and the compiled code is efficient.

2 The dynamic programming facility

The facility we have implemented allows a C programmer to declare a two-argument function to be a *dynamic function*. This means that the function will have a dynamic behaviour when called: each new value is stored in a database, and a new call to the function first checks the database for the result. This is made possible by a set of macros and by the use of two separate modules. The first one defines the basic functions for comparing relations, and the second one implements a good store-and-retrieve algorithm using AVL trees.

2.1 An example

Le us reconsider the previous example concerning distances. We will define a function `dist` returning an `int` to implement the computation of the distance. Its arguments have a predefined type `STR *`, which is a pointer to a string. The normal definition of this function in C would be:

```
int dist(STR *arg1, STR *arg2)
{
...
}
```

Using the dynamic programming facility, the definition of the function will read as follows:

```
#define dist(arg1,arg2)  (int) _reln_(dist,arg1,arg2)

int dynamic(dist)(STR *arg1, STR *arg2)
{
...
}
```

The `#define` line can be seen as the declaration of a relation composed of the name of the function and its two arguments (relations are defined throughout the next section). The `dynamic` keyword can be seen as a functional: it applies to a function and creates a new function from its argument. This is explained in the next subsection.

2.2 The dynamic programming facility macro

The keyword dynamic is in fact a macro. It takes a function name as an argument and creates the name of another function, the use of which will be explained shortly.

The definition of this macro uses the ## operator of the ANSI-C preprocessor. It is as follows:

```
#define dynamic_prime(fct)      _static_ ## fct
#define dynamic(fct)            dynamic_prime(fct)
```

For our example, the name of the new function is _static_dist. This name (_static_) is used because this function performs the actual computation and does not access the database.

The macro definition for the relation hides a call to a specific function with specific objects. _reln_ itself is defined as a macro. It implements the search in the database before any actual call to the function. It is defined as follows:

```
#define _reln_(fct,gen1,gen2)                \
   ( ((RELN *) args2Reln(dynamic(fct),       \
                         gen1,               \
                         gen2,               \
                         NULL)           )->arg3 )
```

Keeping in mind that RELN and args2Reln are respectively a type and a function defined in the reln module, the interpretation is as follows:

- construct a relation with four elements: the name of the function, its two arguments, and a NULL in the fourth position, for the still unknown result;
- look for the database of relations for this uncomplete relation and return the value of the last position. The function args2Reln
  - either finds the relation in the database and can return the result;
  - or performs an actual call to the function, stores the complete relation, and returns the result.

This macro replaces any call dist(arg1,arg2) by a corresponding call to args2Reln throughout the source code.

2.3 Relations

The `reln` module defines a class of objects representing relations. Relations can be used for any purpose. However, for the dynamic programming facility, a particular interpretation has been retained.

A relation has four positions. In the interpretation for the dynamic programming facility, the first position stores a pointer to the function, the two following positions point to the two arguments of the function, and the last position represents the result of the application of the function to the arguments.

The set of all relations can be seen as a simple database into which it is possible to store new relations and from which one can retrieve stored relations.

2.4 AVL trees

Access to the relations stored in the database must be fast. In our implementation, we have chosen the AVL-tree method, because it meets the speed requirement, and it is elegant and compatible with our inclination toward tree structures.

In our general implementation work, this technique is not reserved for relations. All other stored objects (`ATOM`s, `LIST`s, `TREE`s) are stored and accessed through the same technique. Consequently, the implementation of this technique has been carried out in a separate module.

2.4.1 Definition

AVL-trees have been defined by [Adel'son-Velskiĭ & Landis 62][4]. Descriptions of the algorithms for storing and retrieving on this data structure can be found in [Aho & al. 74, pp. 166-167] and [Knuth 73, pp. 451-461]. Another simple presentation is to be found in [Alagar 89, p. 494-502]

AVL-trees are balanced binary trees: *for any node in the tree, the difference between the heights of the left and the right subtrees is at most* 1. See Figure 1 for examples of AVL trees, and Figure 2 for counter-examples.

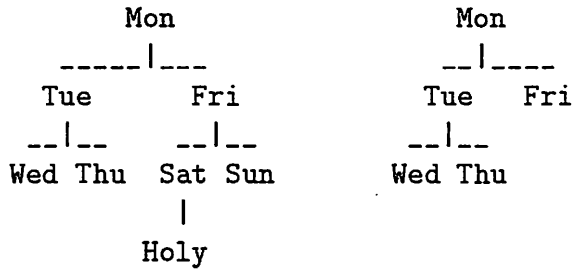

Figure 1: Examples of AVL-trees

[4]This reference renders unto Caesar the things which are Caesar's, but I must confess I have never read this original paper.

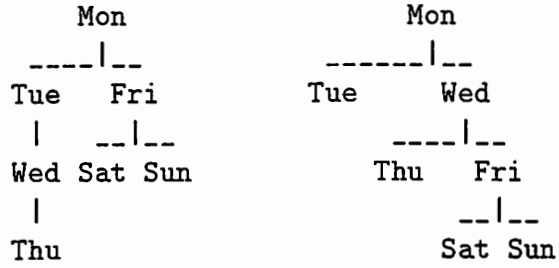

Figure 2: Counter-examples for AVL-trees

When used for ordering purposes, the nodes in an AVL trees are positioned in such a way that their projection renders the ordering (as in Figure 3).

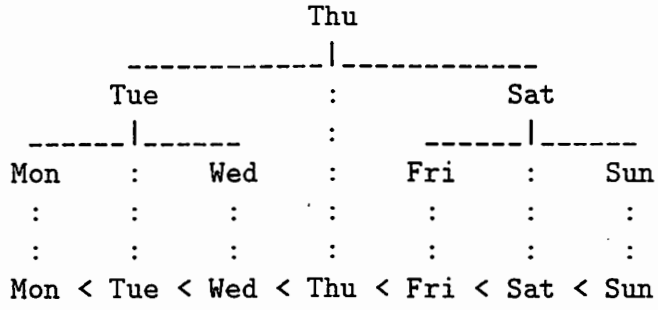

Figure 3: Ordering in AVL trees

An interesting property of this storage data is that the search for an element is in $log(n)$, n being the number of objects stored. This property is a direct consequence of the binary structure.

2.4.2 Rotations

When we insert a new node new-node in the tree, the tree may become unbalanced. In that case, a transformation has to be performed in order to rebalance it. It has been shown that, in all cases, one of two transformations, called rotations, suffice. They restore to the tree a balanced binary structure, and conserve the ordering which the tree accounts for. This remarkable property constitutes the elegance of AVL

trees. In the following, we just describe the rotations, without giving any formal justification.

Single rotation The simplest rotation is applied when a situation as shown in Figure 4 is encountered. In this configuration, according to the comparison function, the new-node has to be added under the tree rooted in :3. This would unbalance the tree, because the difference in height between the leftmost and the righmost subtrees becomes 2. The whole tree would then be unbalanced. The transformation shown in Figure 4 ensures the conservation of balance.

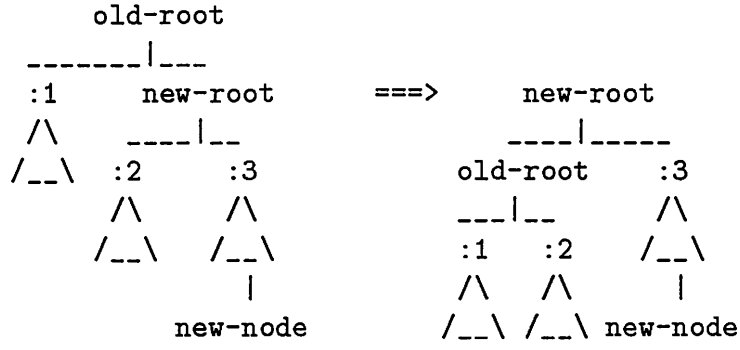

Figure 4: Single rotation

Double rotation This transformation must be applied when the node to be added provokes an imbalance as illustrated in Figure 5. We elevate a central node (here called new-root) as the root of all the transformed subtree; and link its right subtree to its new left daughter and its left subtree to its new right daughter.

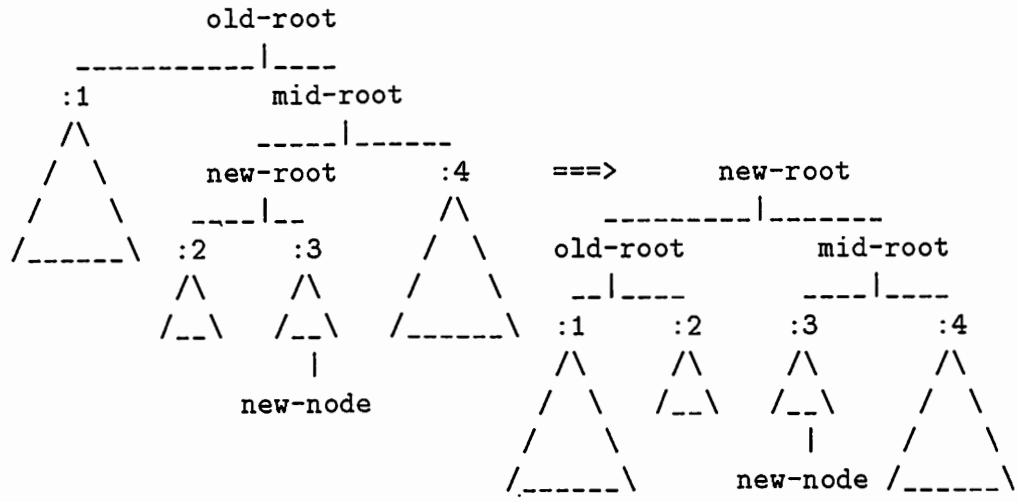

Figure 5: Double rotation

In order to know where in the tree a transformation has to be applied and which one, the nodes in the AVL tree must retain a balance factor which is the difference in height between the left and right subtrees.

2.4.3 The AVL module

The interface of the AVL tree module we have implemented proposes two functions, one for lookup, and the other one for storing:

```
void *findinAvl(void *, AVLCMP, AVL **) ;
void *addinAvl (void *, AVLCMP, AVL **) ;
```

These two functions take three arguments

- the object to be looked up or added to the AVL tree;
- the comparison method for the object type. It must yield -1 or 1 if two compared objects are not equal and 0 otherwise;
- the AVL tree for the object type.

As output, the lookup function returns the object if it was found and `NULL` if the object was not found. The storing functions returns the address of the object stored.

These two functions are general. Hence, in the framework of our studies on distances and pattern-matching on strings and trees, we use them for a range of different objects: atoms, strings, trees, boards and relations [Lepage 92b].

Conclusion

This report has presented a facility which eases programming in C: a set of macros transforming a two argument function into a dynamic function.

We have shown using examples that this facility allows the programmer to keep the simplicity of a recursive definition, and, at the same time, that the computation times are drastically reduced.

For our application to string distances, use of the facility divides the time by an exponential in the sum of the data limits! Thus, it yields performances which are comparable to the ones given by a dedicated algorithm.

The implementation of the dynamic programming facility entailed the definition of a special object called a relation. We use it to store the results of a computation in the database.

Storing into and retrieving from this database are realised through a well-known technique: using AVL trees. Our implementation is general and is consequently used to store all of the objects we have defined in our studies on distances and pattern-matching on linguistic objects.

A Annex: Asymptotic behavior of the recursive algorithm for the Wagner and Fischer distance

We consider an array A of size $M \times N$. Each cell in the array contains a value which is a function of the three previous cells:

$$A[m+1, n+1] = g(A[m, n+1], A[m+1, n], A[m, n])$$

where g is some function. A cell on the edges is a function of the previous cell on the same edge:

$$A[0, n+1] = g'(A[0, n]) \quad \text{and} \quad A[m+1, 0] = g'(A[m, 0])$$

Moreover, the first cell $A[0, 0]$ is given.

This definition of functions on an array is a generalisation of the recursive function defining the Wagner and Fischer distance between strings.

A.1 The problem

We want to compute, or at least to estimate, the number of accesses to cells when computing cell $A[m, n]$ through a direct application of the recursive definition. We call this number $f(m, n)$.

Commutativity First, we observe that the function f is commutative, because the problem is symmetrical relative to the first diagonal:

$$\forall (m, n),\ f(m, n) = f(n, m)$$

Value of $f(0,0)$ As the first cell A[0,0] is a given, its "computation" consists in just one read operation. Hence:

$$f(0, 0) = 1$$

First row On the edges, we have:

$$\forall n,\ f(0, n+1) = f(0, n) + 1 \quad \text{and} \quad f(n+1, 0) = f(n, 0) + 1$$

With the value of $f(0, 0)$ known, we get:

$$\forall n,\ f(0, n) = f(n, 0) = n + 1$$

Second row Using the previous result, we can compute the formula for the cases where $m = 1$ or $n = 1$. We consider $m = 1$. The value of $f(1,n)$ is computed from the preceding cell on the same row, plus the two preceding cells on the first row. Moreover the cell itself is accessed once. This yields the formula:

$$f(1,n) = f(1,n-1) + f(0,n) + f(0,n-1) + 1$$

Using the result obtained for the first row, we get:

$$f(1,n) = f(1,n-1) + n + 1 + n + 1 = f(1,n-1) + 2(n+1)$$

This formula can apply from n down to 1. By summing up the lines, and by replacing $f(1,0)$ with its value, we get:

$$\begin{array}{lcll} f(1,n) & = & f(1,n-1) & + \ 2(n+1) \\ f(1,n-1) & = & f(1,n-2) & + \ 2n \\ \vdots & & & \vdots \\ f(1,1) & = & f(1,0) & + \ 2 \times 2 \end{array}$$

$$\begin{array}{lcll} f(1,n) & = & f(1,0) & + \ 2\sum_{i=1}^{n}(n+1) \\ & = & 2 & + \ n(n+1) + 2n \\ & = & (n+1)(n+2) & \end{array}$$

As a result, we have:

$$\forall n,\ f(1,n) = f(n,1) = (n+1)(n+2)$$

General case When computing cell $A[m+1, n+1]$ for $m > 0$ and $n > 0$, we access this cell once and we must count the number of accesses in the computation of all three previous cells. Noting this, we write the following general formula:

$$f(m+1,n+1) = 1 + f(m,n+1) + f(m+1,n) + f(m,n)$$

Conjecture on the asymptotic behaviour In order to estimate the asymptotic behaviour of the function f, we did some experiments which led us to conjecture that there must exist some α, $\beta > 1$ such that, for all $m, n > 0$,

$$nm\alpha^{n+m} \leq f(m,n) \leq mn\beta^{m+n}$$

The remainder of this appendix proves that α and β exist.

A.2 A lower bound for f

We first try to determine the value of α. We will use noetherian induction to prove the following proposition, for all m, n:

Proposition 1

$$mn\alpha^{m+n} \leq f(m,n)$$

This proposition can be proved for all $m, n \geq 0$.

Ordering on pairs As noetherian order on the pairs (m,n), we take the lexicographic order defined as follows:

$$(n,m) < (m',n') \stackrel{\text{def}}{\Longleftrightarrow} (m < m') \vee (m = m' \wedge n < n')$$

With this order, the following implications are verified:

Proposition 2

$$\begin{array}{lrrcl} m < m+1 & \Rightarrow & (m,n+1) & < & (m+1,n+1) \\ m+1 = m+1 \ \wedge \ n < n+1 & \Rightarrow & (m+1,n) & < & (m+1,n+1) \\ m < m+1 & \Rightarrow & (m,n) & < & (m+1,n+1) \end{array}$$

Induction Suppose that Proposition 1 is true for all $(p,q) < (m',n')$. We will prove that it holds for (m',n').

First row If $m' = 0$ or $n' = 0$, we use the first row formula. As commutativity holds, we consider only the case $m' = 0$. Obviously,

$$0 \times n' \times \alpha^{0+n'} = 0 \ \leq \ f(0,n') = n'+1$$

is true for all n'.

General case If neither m' nor n' is zero, then $m' = m+1$ and $n' = n+1$ with $m \geq 0$ and $n \geq 0$. Suppose that Proposition 1 is true for all ranks lower that $(m+1, n+1)$. The orderings results given in Proposition 2 imply that Proposition 1 holds for ranks $(m, n+1)$, $(m+1, n)$ and (m, n). Then, by summing up the first four following inequations and by definition of f, we get:

$$\begin{aligned} 1 &\leq 1 \\ m(n+1)\alpha^{m+n+1} &\leq f(m, n+1) \\ (m+1)n\alpha^{m+n+1} &\leq f(m+1, n) \\ mn\alpha^{m+n} &\leq f(m, n) \end{aligned}$$

$$1 + ((m(n+1) + (m+1)n)\alpha + mn)\alpha^{m+n} \leq f(m+1, n+1)$$

For Proposition 1 to be true at rank $(n+1, m+1)$, it is sufficient that:

$$(m+1)(n+1)\alpha^{m+n+2} \leq 1 + ((m(n+1) + (m+1)n)\alpha + mn)\alpha^{m+n}$$

which is equivalent to

$$0 \leq 1 + \left(-(m+1)(n+1)\alpha^2 + (m(n+1) + (m+1)n)\,\alpha + mn\right)\alpha^{n+m}$$

If we posit that $\alpha > 0$, then $\alpha^{n+m} > 0$, and we can divide by this quantity without changing the sense of the inequality:

$$0 \leq \frac{1}{\alpha^{n+m}} - (m+1)(n+1)\alpha^2 + (m(n+1) + (m+1)n)\alpha + mn$$

We put the polynomial in α on the left side of the inequality to get:

$$(m+1)(n+1)\alpha^2 - (m(n+1) + (m+1)n)\alpha - mn \leq \frac{1}{\alpha^{n+m}}$$

It suffices that this proposition hold for all m, n, *i.e.* for the lower bound of $\frac{1}{\alpha^{n+m}}$. If we posit $\alpha > 1$, 0 is the lower bound and we have:

$$(m+1)(n+1)\alpha^2 - (m(n+1) + (m+1)n)\alpha - mn \leq 0$$

As the coefficient of α^2 is positive, this is true for $\alpha_1 \leq \alpha \leq \alpha_2$ where α_1 and α_2 are the two solutions of the equation

$$(m+1)(n+1)\alpha^2 - (m(n+1) + (m+1)n)\alpha - mn = 0$$

Their values are

$$\alpha_1 = \frac{1}{2} \times \left[\frac{m}{m+1} + \frac{n}{n+1} - \sqrt{\left(\frac{m}{m+1} + \frac{n}{n+1}\right)^2 + 4\frac{m}{m+1}\frac{n}{n+1}} \right]$$

and

$$\alpha_2 = \frac{1}{2} \times \left[\frac{m}{m+1} + \frac{n}{n+1} + \sqrt{\left(\frac{m}{m+1} + \frac{n}{n+1}\right)^2 + 4\frac{m}{m+1}\frac{n}{n+1}} \right]$$

We posited $\alpha > 1$. As α_1 is negative, it is sufficient to find the minimum value of α_2 for m and n such that $\alpha_2 > 1$, but we have to verify the property for those m, n for which α_2 would be inferior to that minimum. α_2 is a non-decreasing function of n (and of m), so we explore starting from $(0,0)$.

$$\begin{array}{rcl} \alpha_2(0,0) = 0 & \leq & 1 \\ \alpha_2(0,1) = \frac{1}{2} & \leq & 1 \\ \alpha_2(1,1) = \frac{1+\sqrt{2}}{2} & > & 1 \end{array}$$

For $(0,0)$ and $(0,1)$, Proposition 1 holds according to the first row case. So we can choose $\alpha = \frac{1+\sqrt{2}}{2}$. Hence, as a first partial result, we have proved that:

$$\forall (n,m),\ mn\left(\frac{1+\sqrt{2}}{2}\right)^{m+n} \leq f(m,n)$$

A.3 An upper bound for f

In exactly the same way, we can look for an upper bound for f, that is for a β such that

Proposition 3

$$f(m,n) \leq mn\beta^{m+n}$$

We will prove this proposition only for $m, n \geq 1$.

Second row We want to find β such that:

$$f(1,n) = (n+1)(n+2) \leq n\beta^{1+n}$$

or

$$\frac{n+2}{n} = 1 + \frac{2}{n} \leq \frac{\beta^{1+n}}{n+1}$$

An upper bound for $1+\frac{2}{n}$ is given with $n=1$. It is 3. Hence, it suffices to find a β such that $3 \leq \frac{\beta^{1+n}}{n+1}$. $\frac{\beta^{1+n}}{n+1}$ is a non-decreasing function for $n > 1$ if $\beta \geq 3/2$ because:

$$\frac{3}{2} \leq \beta \Rightarrow \forall n > 1,\ \frac{m+2}{m+1} \leq \frac{3}{2} \leq \beta = \frac{\beta^{m+2}}{\beta^{m+1}}$$

Hence:

$$\forall n > 1,\ \frac{\beta^{m+1}}{m+1} \leq \frac{\beta^{m+2}}{m+2}$$

Hence, taking the minimum value of $\frac{\beta^{m+1}}{m+1}$, which is obtained for $m=1$, we get:

$$3 \leq \frac{\beta^2}{2} \Rightarrow \sqrt{6} \leq \beta$$

As $\sqrt{6} \geq 3/2$, we can now look for a $\beta \geq \sqrt{6}$ for the general case.

General case If we follow the same rationale as for α, and considering that $\beta \geq 1$ since we want $\beta \geq \sqrt{6}$, we end up with the inequality:

$$\frac{1}{\beta^{n+m}} \leq (m+1)(n+1)\beta^2 - (m(n+1) + (m+1)n)\,\beta + mn$$

If it is true for all $m,n \geq 1$, then it must be true for an upper bound of $\frac{1}{\beta^{m+n}}$, 1 for example. Hence:

$$1 \leq (m+1)(n+1)\beta^2 - (m(n+1) + (m+1)n)\,\beta - mn$$

which yields

$$0 \leq (m+1)(n+1)\beta^2 - (m(n+1) + (m+1)n)\,\beta - mn - 1$$

This inequation is verified for

$$\beta \leq \beta_1 \text{ or } \beta_2 \leq \beta$$

where β_1 and β_2 are the two solutions of the equation

$$(m+1)(n+1)\beta^2 - (m(n+1)+(m+1)n)\,\beta - mn - 1 \;=\; 0$$

Their values are:

$$\beta_1 \;=\; \frac{1}{2} \times \left[\frac{m}{m+1} + \frac{n}{n+1} - \sqrt{\left(\frac{m}{m+1} + \frac{n}{n+1}\right)^2 + 4\frac{mn+1}{(m+1)(n+1)}} \right]$$

and

$$\beta_2 \;=\; \frac{1}{2} \times \left[\frac{m}{m+1} + \frac{n}{n+1} + \sqrt{\left(\frac{m}{m+1} + \frac{n}{n+1}\right)^2 + 4\frac{mn+1}{(m+1)(n+1)}} \right]$$

We want $\beta \geq \sqrt{6}$. As β_1 is negative, we must only consider $\beta_2 \leq \beta$. It can be easily proved that β_2 is a non-decreasing function of m (and of n). So, we explore the possible values from $(1,1)$:

$$\beta_2(1,1) = \tfrac{1+\sqrt{3}}{2} < \sqrt{6}$$

As $\beta_2(1,1)$ is already less than $\sqrt{6}$, we must take $\beta = \sqrt{6}$. Hence, as a second partial result, we have proved that:

$$\forall (n,m),\;\; f(m,n) \leq mn\left(\sqrt{6}\right)^{m+n}$$

A.4 Final result

By combining the two previous partial results, we have shown that f can be framed between two exponential functions:

$$\forall (n,m),\;\; mn\left(\frac{1+\sqrt{2}}{2}\right)^{m+n} \leq f(m,n) \leq mn\left(\sqrt{6}\right)^{m+n}$$

To conclude, the asymptotic behaviour of the function f is exponential in the sum of the size of the givens of the problem.

References

[Adel'son-Velskiĭ & Landis 62] Georgiĭ Maksimovich Adel'son-Velskiĭ and Evgeniĭ Mikhaĭlovich Landis
An algorithm for the organization of information
Dokl. Akad. Nauk SSSR, **146**, 263-266, 1962.
English translation in *Soviet Math.* **3**, 1259-1263.

[Aho & al. 74] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman
The Design and Analysis of Computer Algorithms
Addison-Wesley Publishing Company, 1974.

[Alagar 89] Vangalur S. Alagar
Fundamentals of computing
Prentice-Hall International Editions, 1989.

[Kernighan & Ritchie 88] Brian W. Kernighan and Dennis M. Ritchie
The C programming language, 2nd Ed.
Prentice Hall, Inc, 1988.

[Knuth 73] Donald E. Knuth
The art of computer programming
Volume 3/Sorting and Searching
Addison-Wesley Publishing Company, 1973.

[Lepage 92b] Yves Lepage
Easier C programming
Some useful objects
ATR report TR-I-0294, Kyoto, November 1992.

[Lowrance & Wagner 75] Roy Lowrance and Robert A. Wagner
An Extension of the String-to-String Correction Problem
Journal for the Association of Computing Machinery, Vol. 22, No. 2, April 1975, pp. 177-183.

[Selkow 77] Stanley M. Selkow
The Tree-to-Tree Editing Problem
Information Processing Letters, Vol. 6, No. 6, December 1977, pp. 184-186.

[Tai 77] Kuo-Chung Tai
The Tree-to-Tree Correction Problem

Journal for the Association of Computing Machinery, Vol. 26, No. 3, July 1979, pp. 422-433.

[Wagner & Fischer 74] Robert A. Wagner and Michael J. Fischer
The String-to-String Correction Problem
Journal for the Association of Computing Machinery, Vol. 21, No. 1, January 1974, pp. 168-173.

Index