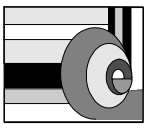

INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

No. 79 **Serie:** AZUL **Fecha:** Agosto 2000

Families of "any time" Algorithms and Methods to Compose Them

Adolfo Guzmán[1]

ABSTACT

An "any time" algorithm has to deliver its answer any time it is needed; it has to provide an answer or a solution in certain (often small) amount of time, which is not known in advance. Thus, the algorithm must have always "an (approximate) answer at hand", that may keep refining as time passes, until the demand for it arrives. Some ways to create any time (ant) algorithms from non ant ones is presented. A collection of ant algorithms is given. Several manners to form new ant algorithms out of old ones are exposed. It is also shown a creative way to use the *idle time* (the time the ant algorithm is doing nothing, waiting to be called) to improve the values that are required in a hurry.

Key words: Any time algorithm; approximate computation, predictor, preliminary results, successive approximation.

[1]Centro de Investigación en Computación del IPN

Families of “any time” algorithms and methods to compose them

1. INTRODUCTION AND OBJECTIVES

Normal algorithms provide their answer or solution when it is ready, and the time taken to compute it is not known or it is irrelevant.

Real time algorithms must provide their answer within a time interval, which is known at the beginning of execution. These algorithms have known constraints on the time they can use to compute their answer.

Incremental algorithms compute the answer using as basis the previous result(s). There is no constraint on how much time they take.

Iterative algorithms compute the answer repeating the same procedure over the successive values of the candidate answer, until some criteria on convergence or number of loops is met. They belong to the incremental algorithms. There is no constraint on how much time they take.

Any time (ant) algorithms must provide an answer “on demand”, but how much time they have to compute it, is not known in advance. Sometimes the answer is required (and must be provided) very soon after starting; in other executions, the same algorithm may have more time before the answer is required. Thus, they are designed to have an answer always ready, but they use the available time for improving it. A chapter of the book [Guzmán 94] provides practical ways to produce these algorithms.

Example: a program to play chess, where a search is done up to a certain level, and a result (play) is obtained. If there is more time, it looks at a deeper level, and obtains a better result. And so on, until it runs out of time.

Example: as time passes, a given algorithm has the values 3., 3.1, 3.14, 3.14, 3.1416, 3.14159, ... as its answer. Which one is given depends on the time of the request.

Example: an election prediction algorithm has, as time passes, the answers Cárdenas, Cárdenas, Labastida, Labastida, Fox, Labastida, Fox, Fox, Fox, ...

Example: a normal decompressing algorithm produces an image only when it finishes. A flexible decompressing algorithm produces quickly a (poor) image, which then becomes sharper as time passes.

Ant algorithms or computation are also known as “approximation” algorithms, or converging computations. They are useful because they provide an answer no matter how little or much time there is to obtain it. We can think of an ant algorithm as containing several ways (other algorithms, perhaps non ant) to predict the true value, executed in order of accuracy (closeness to the true value; see definition in §1.3).

Flexible algorithms (flexible computation) are a generalization of ant algorithms (which only have time restrictions); here, restrictions can be on space, bandwidth or other resource.

1.1 Objective

Our objective is to exploit and handle ant algorithms. Thus, §2 presents families of ant algorithms, and ways to obtain them out of normal algorithms. Also, §2.5 constructs *new* ant algo-

rithms out of *old* ones. §3 proposes a creative way to use the *idle time* to improve things, so that the computation of future values becomes faster. Finally, some thought is given to their parallelization (§3.2).

We consider an algorithm with several inputs *u*, *v*, *w*, *x*, ... and one output *y*. Since §3.4 tells us how to handle non numeric values (for the purposes of this paper) as if they were numeric, we can assume that all these values are numeric. We can think that there is a cell **y** where the value *y* is deposited several times, and read (by a an exogenous consumer) only once. See Figure 1. Compare with *single assignment languages*, where each variable is written only once, and read many times. It is convenient to assign a semaphore to this cell, with values **virgin** (no *y* value yet; cell is empty), **in use** (cell has a valid value), **read** (cell has been read, no sense to keep improving the answer), and **finished** (algorithm has finished, no further accuracy [Cf. §1.3] can be sought). The semaphore takes these values in the sequence **v, i, r** or **v, i, f.**

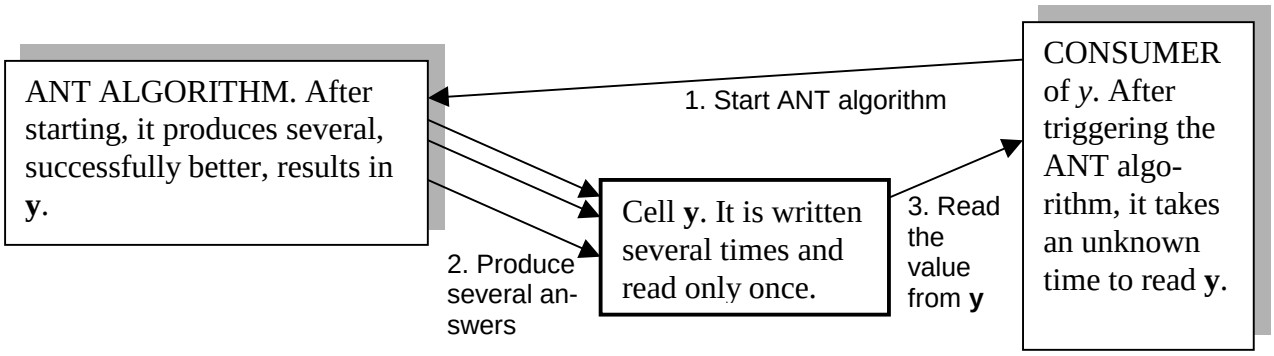

FIGURE 1. The *consumer* (1) starts the ANT algorithm, which then (2) produces several answers in cell **y**. Asynchronously, (3) the *consumer* reads cell **y** whenever it wishes, obtaining the best result obtained thus far.

1.2 Research areas for ant algorithms

Some are: a) how to provide best results with least resources (time); b) how to make gradual, instead of sharp, improvements of the result (ideally, a continuous improvement is sought); c) how to produce ant algorithms out of ordinary ones; d) how to mix ant algorithms; e) what to do while the algorithm is not computing, so as to improve its accuracy for *next* time; f) more general tradeoffs with flexible algorithms; g) how to parallelize them. This paper addresses (a, c-e).

1.3 More definitions

Algorithm. A formal procedure, a computer program, to perform something, such as a computation; to obtain some output values, to make some calculation, out of input values, out of measurements.

Exact algorithm. An algorithm whose output is “correct” (i. e., not an approximation, not an estimation).

Approximate algorithm, approximating algorithm. Algorithm A approximates algorithm E, if E is exact, and B gives a value near that of E. Compare with ant algorithm.

A chain ant algorithm is an ant algorithm that can be represented as a sequence of successively more accurate approximation (non ant) algorithms to compute value *y* (Cf. §1.1), the last one in the chain being the exact calculation. I. e., A(u, v, w, x, ..) = {A1, A2, A3, ...}(u, v, w, x, ..). Each Ai is an *element* of the chain, and produces a value for y, of increasing accuracy. In general, the Ai's are not ant algorithms. Example: J1 of §2.2.

Some definition of scalar quantities follow for Ai, an element of a chain. Since these quantities vary from one computation (one execution of Ai) to the next, we should consider their average value over many computations.

The *accuracy* of Ai is N_{i2}/N_T, a number between 0 and 1, where N_{i1} is the number of correct bits (when compared with N_T, the number of bits[1] of the correct answer) that the initial value of y (the input to Ai) has,[2] and N_{i2} is the number of correct bits that the final value of y produced by Ai has. Ai has increased the correct bits of y from N_{i1} to N_{i2}.

The *increase in accuracy* of Ai is $(N_{i2} - N_{i1})/N_T$.

The *additional number of correct bits computed* in Ai is $N_{i2} - N_{i1}$.

The *rank* obtained by Ai is the position of the least significant bit that Ai obtains correctly. Bits are counted from the most significant to the least; the count starts at 1. Example: If A3 produces values of y that are correct up to seven bits, the rank of A3 is 7, and we sometimes write it as a subindex, thus: $A3_7$.

The *compute time* of Ai is the time t it takes to execute, if it ran in a single dedicated processor. Sometimes write it as a superindex, thus: Ai^t.

The *efficiency* of Ai is the ratio *increase in number of additional correct bits obtained/compute time,* or $(N_{i2} - N_{i1})/t_i$.

An element that augments in three the number of correct bits of y in 0.1 second has an efficiency of 30 bits/sec, one that augments these bits in 1 second has an efficiency of 3 bits/sec. But: final (less significant) correct bits are usually slower to obtain than coarse (more significant) bits. This may be taken into account in a more elaborate definition of *efficiency*, which we shall not develop.

Idle time. The interval elapsed from the time an ant algorithm has delivered its value (**y** has been read) until the next time the algorithm is started.

An *element* of a chain ant algorithm is *independent* when it does not use values produced by some previous *element* (as initial value, for instance). Example: J1 of §2.2. Otherwise, the element is *dependent.*

To *ant* an algorithm E is to make an any time algorithm out of it. Making an ant algorithm out of a non-ant one, or *anting it,* is shown in §2.2.

X is *better than* Y, where X and Y are ant algorithms that ant the same function E if, given that both execute for the same amount of time Δt, then, for all Δt, X produces better results (with better or equal accuracy) than Y.

If algorithms X and Y do not ant the same function E, the comparison is meaningless.

---

[1] We assume fixed point format, in two's complement form. That is, the sign counts as one bit, is the most significant bit, and it is bit number 1. Other formats can be used, with simple variations in these definitions.

[2] If the initial value is a random value, we can say that $N_{i1}=0$. If the initial value is some previously computed value of y, we know already the value of N_{i1}.

2. GENERATING ANT ALGORITHMS

2.1 An initial family of algorithms approximating an exact algorithm E

Given an exact algorithm y = E (u, v, w, x, ..), which we think is slow, we would like to produce other (non ant) algorithms yielding results close to y, but faster. That is, we are willing to produce fast *approximating algorithms to E.* These are some, in increasing order of execution time and of average accuracy:

A. **Return last value**. The fastest algorithm: return the *previous* (last) value. It remembers the last value it produced, which is kept in cell **y** and can be read at any time. A takes no time to compute its answer, "it already knows it."

B. **Memoing.** Have a cache of tuples (y_1, u_1, v_1, w_1, x_1); (y_2, u_2, v_2, w_2, x_2), ...[a large table] and, depending on input (u, v, w, x), produce a suitable value for y. Today's main memories easily store one million answers.

C. **Interpolation.** Linearly interpolate the value y for point (u, v, w, x) from values of y for points (y_1, u_1, v_1, w_1, x_1); (y_2, u_2, v_2, w_2, x_2), ... (Figure 2). It uses the table of (B).

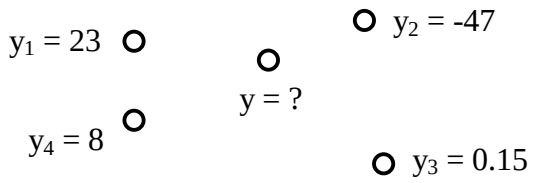

FIGURE 2. Value of y is deduced from interpolating from values from y_1, y_2, y_3, ...

D. **Non linear interpolation.** Similar to C, but give more weight to points closest to (u, v, w, x).

E. **Compute the bona fide value.** Execute E, that is, compute the correct value. It is assumed that this is the slowest of A through E.

Notice that all of these A through E are "outside approximations" to E, in the sense that they are obtained without examining the insides (the code, the inner works) of algorithm E, which is regarded as a black box. More approximating algorithms can be produced in §2.3 by opening E and applying A-D, and other methods, to its parts or subcomputations.

2.2 How to produce an ant algorithm from a non-ant algorithm in black box form

Remember that an ant algorithm always has its output y ready for consumption. Given an exact algorithm E, the most straightforward ant of E is

```
J1. y = A (u, v, w, x, ...);
    y = B (u, v, w, x, ...);
    y = C (u, v, w, x, ...);
    y = D (u, v, w, x, ...);
    y = E (u, v, w, x, ...);
    end;
```

where A through E are the (non ant) algorithms of §2.1. Algorithm J1 says: have ready the previous value *y* in **y**; then, read the cache of B to store in **y** the appropriate value; then, linearly interpolate (using C) from values in the cache; then, non-linearly interpolate (using D) from same values; then, compute the exact value. At some time during the execution of J1, the value y will be read. After this, there is no sense to continue the execution of J1.

Notice that we treated E as a black box, and A, B, C, and D were obtained without delving in the form or code of E.

J2. Given a chain ant algorithm J1, one can produce a better chain ant algorithm J2 by deleting all those elements (except one: the fastest) which have the same rank. Because it does not make sense to execute two elements to obtain several values with the same rank; one of them will do.

J3. Given a chain ant algorithm J2, one can produce a better chain ant algorithm J3 by sorting the elements of J2 in ascending rank order. Because it does not make sense to spend some time to produce an answer of rank r1, and then to spend an additional amount of time to produce a new answer with rank less than r1.

J4. When in a chain ant algorithm where its elements are in ascending rank order (like J3), one element Ai is not faster than another element Ai+k somewhere to its right (i. e., with higher rank), and no element of higher rank than Ai is dependent on Ai, a better algorithm J4 is obtained by deleting Ai from the chain, since Ai+k is faster and more accurate than Ai.

Remark: after the above simplifications, the elements of the chain are in strictly ascending rank (and in strictly ascending compute time, too, if there are no elements on which other elements depend). Next section delves more on improving ant algorithms.

2.3 Anting an algorithm by combining antings of its subcomputations

Given that we want to *ant* a non ant algorithm, we can get results better than §2.2, by *opening* the algorithm to expose its parts or subcomputations, anting *these* and combining the results. This section explains how.

Suppose the exact algorithm E(u, v, w, x) has the following form:

```
E.  a = f (u, v, w, x);
    b = g(u, v, w, x);
    c = h(u, v, w, x);
    y = a + b + c;
    end;
```

then an ant of E can be:

```
J5. a = ant of f;
    b = ant of g;
    c = ant of h;
    y = a + b + c;
    end;
```

J5 is not the best: it spends a good amount of time computing ant of f, and it may not have enough time to compute ant of g. The bell rings for J5; it needs to produce the value of y and it just finished computing (very accurately) ant of f. The solution to this problem, given below, is to observe the *rank* of the successive values (of *y*) produced.

2.3.1 General procedure to obtain the best ant of E

Let us say we are looking for M = best ant of E, where M is of the form {m1, m2, ...}. How many mi's? Of what rank each? What should they contain?

A. Decide how many elements the resulting chain {mi} should have. Let us assume that the accurate y has rank 12, so that it is reasonable to produce four elements m1, m2, m3 and m4, to render y's of rank 4, 8, 10 and 12, respectively. We need ant of f, ant of g and ant of h to have chains of four elements, too. But, of what rank each?
B. How are the partial results a, b and c combined (let us call this the *combination part*) in E? Well, they are just added: y = a+b+c (we will treat below other ways to combine partial results). We notice that rank y = max (rank of a, rank of b, rank of c). Thus, it makes sense to spend just enough time in each of the chains fi, gi and hi to obtain *the same rank* in each. The solution is: find four elements for each of chains {fi}, {gi}, {hi}, with ranks 4, 8, 10 and 12, respectively.
C. Do the anting f, g and h, then apply the simplifications of §2.2, in order to obtain the three chains of step B.
D. The solution is: M= {m1; m2; m3; m4} where mi = (a=fi; b=gi; c=hi; y=a+b+c), and the rank of mi = rank of fi = rank of gi = rank of hi = 4, 8, 10 and 12, respectively.

What if the combining part does not treat the partial results equally? Suppose the combination part were y= a + b×c. Since the product yields a rank which is the sum of the ranks of the factors, we need in the first iteration to have f1 of rank 4, but g1 and h1 need to yield an answer of rank 2 only. In this manner, the approximate answer y of m1, the first element, will be of rank 4. For the second element, m2, we have: $m2_8 = (a_8=f2_8; b_4=g2_4; c_4=h2_4; y_8=a_8+b_4\times c_4)$ where we have used subindices to indicate the ranks.

Similarly, if the combining expression were y = a + sqrt(b) + c, the second element would be $m2_8 = (a_8=f2_8; b_{16}=g2_{16}; c_8=h2_8; y_8 = a_8 + sqrt_8 (b_{16}) + c_8)$, since the rank of sqrt is half the rank of its input. Similar results can be obtained for other functions. In general, if y is a complicated function of its previous parts a, b and c, then you need to understand how the rank of y depends on the ranks of f, g and h. Moreover: you do not need to compute this complicated function accurately at each element; you can ant it, too: see §2.3.1.2.

2.3.1.1 If each part can be expressed as a continuous approximation

Sometimes, the ant of each subcomputation of an algorithm can be expressed as a successive refinement of the form of §2.3.2. In our example of §2.3, suppose ant f can be expressed as iterate (fk), and similarly for ant g = iterate (gk), and ant h = iterate (hk), where fk, gk and hk are some expressions that through iteration get the answers. It is tempting to say:

ant E = iterate (y = fk + gk, + hk); this will not work because we do not know when to quit. Let us try:

```
ant E =     y' = y0;                          /* some initial value for y  */
            do {    y = y';
                    y' = fk + gk + hk      }
            until | y – y' | < ε
```

This is still not right since each iteration produces, in general, different advances in the ranks of fk, gk and hk. In other words, gk may need three iterations to add one more "good" bit to its result, while gk needs ten iterations. The expression y' = fk + gk + hk mixes ranks, so that y gets only the smaller of the three ranks of fk, gk and hk. The solution is to have each mi of the answer possess the form

mi_j = (a_j= $iterate_j$ (fk);
b_j= $iterate_j$ (gk);
c_j=$iterate_j$ (hk);
$y_j=a_j+b_j+c_j$)

where the subindices indicate rank, and $iterate_j$ means: iterate until rank j is obtained. For m1, j=4; for m2, j=8; for m3, j=10, whereas for m4, j=12.

2.3.1.2 If the combining part is a slow function that needs to be anted, too

What if the combination is of the form y=k(a, b, c), where k is a complex (slow) computation, instead of the simple and fast y=a+b+c? It is desirable to ant k, too. For M = best ant of E of §2.3.1, ant k will have four elements, of ranks 4, 8, 10 and 12: ant k= {$k1_4$, $k2_8$, $k3_{10}$, $k4_{12}$}. Then, $k1_4$= ($y_4=k_4$(a, b, c)), where k_4 means: compute k with rank 4. By analyzing the form of k, it can be deduced what are the right ranks of a, b and c that will yield a value of k with rank 4. And the same for $k2_8$, $k3_{10}$ and $k4_{12}$. This will give the right ranks for {fi}, {gi} and {hi}.

2.3.1.3 When the value is a composition of functions y = f(g(h(u, v, w, x, ..)))

Here, there is a need to know how the rank of h depends on the ranks of u, v, w and x. And how the rank of g depends on the rank of h. And how the rank of f depends on the rank of g. An example illustrates the procedure. If E = sqrt (v + sin (u – tan (wx))), our goal is to obtain (following our example of §2.3.1) ant E = {$m1_4$, $m2_8$, $m3_{10}$, $m4_{12}$} where the subindices indicate the desired ranks. For instance, the solution for m2 is

$m2_8$ = (a_{20}= tan_{20} ($w_\infty \times x_\infty$);
b_{16}= sin_{16} ($u_\infty - a_{20}$);
y_8= $sqrt_8$ ($v_\infty + b_{16}$)).

The derivation of the ranks starts at the end, in y, and proceeds towards the beginning, at a. y needs to be rank 8, since m2 has rank 8. This means that b has to have range 16. This forces sin to be sin_{16}, which means: iterate or approximate the value of sine until its rank is 16. That requires the rank of a to be 20 (the rank of sin is a bit lower than the rank of its input). This forces tan to become tan_{20}, which means: iterate or approximate the tangent until its rank is 20. It is assumed (∞) that the inputs u, v, x, w have large enough ranks.

2.3.2 Anting an exact algorithm which is already a successive refinement towards the exact answer

In many cases, an exact algorithm E is *already* in ant form. This happens when it is formed by a loop or iteration that keeps improving the value of y. Generally, E takes the form

```
y' = y0;              /*  initial value of y  */
do {        y = y';
```

y' = f(u, v, w, x, y) } /* this f is called the *kernel* of E */

until | y – y' | < ε

In this case, ant of E = E.

Nevertheless, §2.3.1 wants the algorithm to be broken into four elements, each one of given rank (ranks sought in §2.3.1 were 4, 8, 10 and 12). Four suitable values (ε1, ε2, ε3,ε4) of ε to obtain those ranks should be selected. The solution is:

E(u, v, w, x) = {y'=$y0_0$; $E1_4$; $E2_8$; $E3_{10}$; $E4_{12}$} where Ei = do {y=y'; y'=f(u, v, w, x, y)}
until | y – y' | < εi

where each εi keeps the iteration going until the desired rank for y is obtained. Thus, ε1 is the correct value needed to keep the iteration of E1 going until the rank of y is 4, ε2 is the correct value needed to keep the iteration of E1 going until the rank of y is 8, and so on.

With some variation, it is useful to apply this anting to algorithms that, from a quad tree, reconstruct a given image.

2.3.3 Anting algorithms with conditionals

To obtain M = best ant of E, where E is the exact algorithm

E = {f; if p(u, v, w, x, ..) then g (..) else h (..) end-if},

we proceed as in §2.3.1, ignoring the predicate p(..), so that we obtain four chains, each of the form

mi = (fi; if p(..) then gi else hi end-if),

of ranks 4, 8, 10 and 12, respectively. That is: do not ant the predicate p, but keep it exact. Predicates have only one bit accuracy (they are true or false), so they can not be anted.

Figure 3 shows the result M = ant of E, where each Ei = if f(u, v, w, x) then gi(..) else hi(..) end-if, and we have three elements in M.

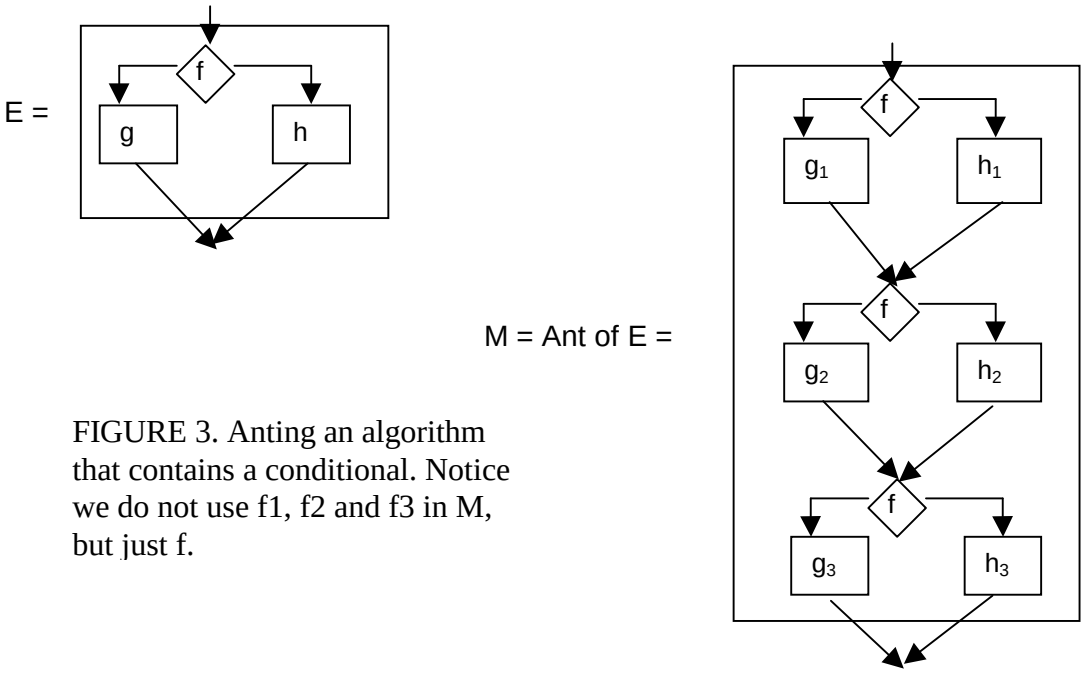

FIGURE 3. Anting an algorithm that contains a conditional. Notice we do not use f1, f2 and f3 in M, but just f.

2.3.4 Other cases of variable y

Some other cases and conditions of variable y are analyzed.

A. y is a value kept in a record in disk. For instance, y is the position an enemy submarine has. A data base with each submarine's position is kept in disk. Each day, a transaction file, with the transactions of the day (submarine movements) is obtained. At night, the E algorithm updates the disk data base with the daily transaction file. It is clear that the new value of y is a function of the value of y in the data base, plus the transaction in the transaction file. Probably E processes each y value sequentially. What does it mean to ant E? That we want to have a new E that delivers at any time the position of a submarine number s. y=E(s). Here, we do not have a good manner to approximate the (old) value of y lying on disk, except to memo it (§2.1.B). Thus, it makes sense to have a large table in random memory of part of the records in the data base, so that updating y will be much faster. Copying these values to disk and retrieving new ones could be done as time permits. The table should contain the most frequently asked submarines, perhaps. This solution is well known as virtual memory (to pretend that the whole data base is in memory, when only some part is) or caching, or memoing.

B. y is a global value that depends on many values on disk and on the transactions file. Let us give y the new meaning $y=(x_c, y_c, z_c)$ is the *center of mass* of the submarines: the position of the center of the submarine fleet. A new algorithm F now computes y at the end of the day by updating the data base with the transactions file, and then computing the center of mass. To ant F, you need to compute y "on the fly", as records are read and written. This ant of F reads every record of the data base, and updates some. Each record processed affects the value of y,

so that it slowly moves towards the "true" or "exact" center of mass. This is slow if the data base is large.

A better algorithm (which is exact, but you can ant it): keep the submarine file together (not fragmented), and don't read the submarines by their submarine number or key s, but *sequentially* as they appear on the file. Previous to processing it, *read into memory* (sequentially, not using s as key) the complete transaction file (it pays to have it not fragmented, too). You will be replacing 10,000 random access lectures (or whatever is the number of submarines, to be called S), each taking 10 milliseconds (total=100 seconds), by 10,000 sequential lectures, each taking 0.1 milliseconds (total=1 sec.), since the records are together in the disk track, so the Winchester head does not need to be jumping all over the disk (each jump takes on average the *seek time*, typically 10 ms).

A better ant of E is produced by keeping together with yesterday's value of **y**, some additional information, namely, the number of submarines (or the total mass, if they are not equally massive). While constructing the transaction file, likewise, keep track of the change of the center of mass of the submarines moving today; of course, this change is (0, 0, 0) at the beginning of the day. With this,

y = [S×old y + today's change of the center of mass]/S.

Moreover, you can iterate and do the above addition incrementally, as soon as submarine movements are detected:

```
y = yesterday's y;                          /* S = total number of submarines */
whenever    (submarine s changes position)
do          { y = [S×y + the change in position of s]/S;
              write the change in position of s to transaction file }
```

and in this case you do not even need to compute and keep the "change of the center of mass of the submarines moving today". This resembles *incremental data mining algorithms*, which avoid recomputing from a large amount of old data [Guzman 97], by keeping additional information (about that old data) to permit division of the calculation in two functions o(old) and n(new), so that the desired value y is y=h(o, n), and the value of o(old) is kept, to avoid recomputing it as new values appear. In our example, o(old)= S×old y; n(new)=change in position of s; and h(o, n) = [o+n]/S. In fact, these algorithms are very good to parallelize; witness our example: suppose you have three computers, one in Europe, one in Asia, and other in USA, where the change of position of submarines in those regions of the world is computed. Then, the kernel of the above iteration is

y = [S×y + the change in Europe + the change in Asia + the change in US]/S.

Since the new function o(new) can be parallelized (in 3 machines, in our example), the value of y is obtained faster.

2.4 Use of the semaphore of the y cell

The interval (**r, v**) or (**f, v**) where the semaphore (§1) goes from being read (**r**) or have finished (**f**) to the beginning (**v**) of the next execution of the ant algorithm, signals the *idle time.* This idle time will be exploited in useful ways in §3.1.

2.5 Combining several chain ant algorithms into newer ones

Assume we want to merge several chain ant algorithms, into a new chain ant algorithm that makes use of the *elements* of the merged chains in the most convenient way. An example will show the procedure. Let us say we want to form a new chain ant algorithm C by adequately merging A = {$A1_4^6$; $A2_8^{9-6}$; $A3_{10}^{12-9}$; $A4_{12}^{16-12}$}, B = {$B1_4^2$; $B2_8^9$; $B3_{10}^{22}$; $B4_{12}^{36}$} and C = {$C1_4^1$; $C2_8^9$; $C3_{10}^{14-9}$; $C4_{12}^{16-14}$}, where the exponents are the execute times (the notation 9-6 explains not only that the execute time of A2 is 3, but that it is a *dependent* element), and the subindices indicate rank. See Figure 4.

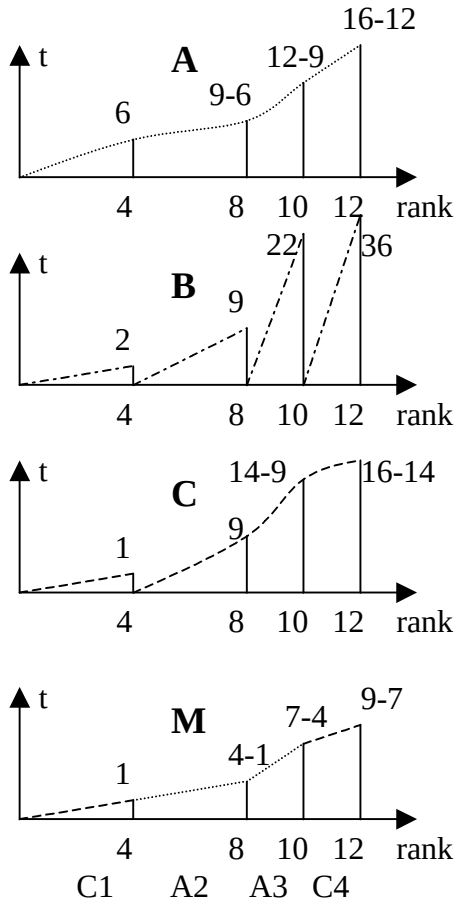

FIGURE 4. How to form a better chain ant algorithm M from three chain ant algorithms A, B and C.

For the first element of M, we see that, from $A1^6$, $B1^2$, $C1^1$, the fastest is $C1^1$, so C1 is chosen. For M2 we select the fastest from $A2^{9-6}$, $B2^9$, $C2^9$; thus, M2 = A2. For M3 we select A3 from $A3^{12-9}$, $B3^{22}$, $C3^{14-9}$. For the last element of M, we select C4 from $A4^{16-12}$, $B4^{36}$, $C4^{16-14}$; thus, M4 = C4.

The new algorithm M is formed of 4 chains. M = {C1; A2; A3; C4}. The ranks are 4, 8, 10 and 12. The compute times are 1, 4-1, 7-4 and 9-7, or 1, 4, 3 and 2.

The idea is to select, for each rank, the closest element of the given chains. Notice how the algorithm works with dependent and with independent elements. In fact, A2 takes its input value, a *y* of rank 4, not from A1, its usual feeder, but from C1, which also produces a *y* of rank 4, too (we selected C1 because it was faster than A1).

Note: make sure that each chain uses different sets of names of variables; otherwise, *aliasing* and wrong results will result.

2.5.1 When merging is no good

Try this example: you have chains {Ai} i =1, m and {Bj} j=1,n; {Ai} producing values with ranks 1, 3, 5, 7, ... and Bi values with even ranks. Each element takes 10 seconds to compute.

The chain Ai produces a value of rank 7 in 40 seconds, while the merged chain A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 (which will be produced by §2.3.1, and we thought it will be the best) in 130 seconds. The advantage of the merged chain is that it produces more intermediate values of y, but it is much slower than either {Ai} or {Bi}. A better solution will be to modify slightly one of the chains (and discard the other), so that at all times (and not only every other rank) the *y* value is available. See §3.3 for how to do this.

3. PARALLEL ANT ALGORITHMS

With the introduction of parallel processors, ant algorithms get even further attention and interest. We discuss in this section (a) the use of idle time, and (b) an initial approach to parallelization. We already exposed in §2.3.4 the parallelization of incremental data mining algorithms.

3.1 Using the idle time to improve its accuracy next time

The time an ant algorithm is doing nothing can be used to improve its accuracy *next time* the algorithm is called. It is "to computer for the future."

I shall use the following simile: A thunder signals "it will rain soon", and triggers the "prepare for rain" ant algorithm. Quickly, I cover with plastic sheets the most valuable merchandise laying in the open backyard. If rain has not started, I cover other less valuable merchandise. If there is still time, I take into the house some merchandise, sequentially. At some time, rain starts. The "prepare for rain" algorithm stops. This is how an ant merchant works. But now, *from the moment the rain starts until the thunder signal of next rain is heard,* is the time to execute the procedure "prepare for *next* rain." This procedure may or may not be ant. I may *make room* in the house, so that no merchandise needs to lay in the patio. I may pile the merchandise in some manner that will facilitate its covering. I may move the merchandise closer to the door, so that it will take less time to move it inside. All of these make either faster or more accurate execution of the "prepare for rain" ant algorithm *next time.* In fact, if we follow this line of reasoning, we will be inventing the *prepared* algorithms, which prepare themselves for next execution, or which precompute the answers to the expected questions. They are very useful because they *store computer time*; that is, they convert idle or wasted computer time into *useful results.*

Question: How can it use the idle time of an ant algorithm? Some answers:

(a) To compute more values of the function, to have ready for a future use. Perhaps soon the input (34, 0, 47, 2) will be given; let's compute its answer and memoize it (§2.1.B).
This is useful for filling the cache of §2.1.B, which is also used in §2.1.C and §2.1.D.

(b) To measure the *accuracy* of each element of the chain. To measure its *rank.*

(c) To ascertain the *compute time* t_i taken by each *element* A_i of an approximate algorithm.

The computation of above quantities is important, since most of the procedures in this paper rely on knowing these quantities.

3.2 Parallelizing an ant algorithm

If an ant algorithm can use several processors, a simple way to parallelize it is to assign each *element* of the chain of the algorithm to a processor. This will not be optimal, since element 1 will finish soon, and processor 1 will be idle most of the time. See Figure 5.

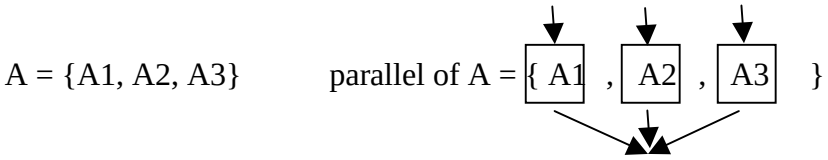

FIGURE 5. A (non optimal) way to parallelize ant algorithm A.

In general, to make best use of the processors available, *you have to divide each element Ai into the processors.* That is, the three processors need to be busy computing A1. Then, the three processors need all go to computer different parts of A2, and so on. For instance, for algorithm J5 of §2.3,

J5. a = ant of f;
b = ant of g;
c = ant of h;
y = a + b + c;
end;

a solution better than Figure 5 is to give a processor to compute a, another for b, another for c. See figure 6.

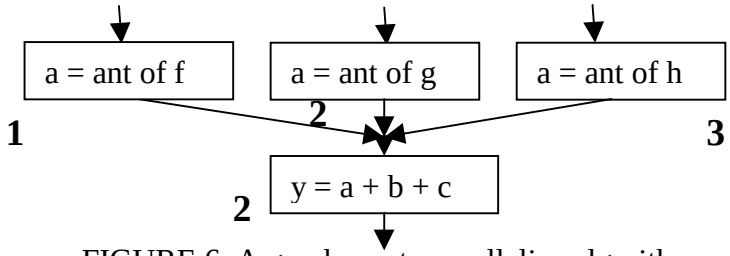

FIGURE 6. A good way to parallelize algorithm J5 of §2.3. The black number indicates the processor assigned to each computation.

To parallelize algorithm M of §2.3.1,

M= {m1; m2; m3; m4} where each mi is

mi = (a=fi; b=gi; c=hi; y=a+b+c),

give to processor 1 the computation of f1, f2, f3, ...; to processor 2 give g1, g2, ...; and to processor 3 give h1, h2, ... As in figure 6, processor 2 could handle the computation of y. Processor 2 has to take care to synchronize (wait for processors 1 and 3, if needed) so as not to add val-

ues with different ranks. By introducing a fourth processor, which only computes y = a+b+c, the load is distributed as follows:

PROCESSOR 1: compute ant of f, without interruption, going from low to high ranks, and have a as an external reference variable (§3.3), so that processor 4 can consume it at any time.

PROCESSOR 2. Same for ant of g.

PROCESSOR 3. Same for ant of h.

PROCESSOR 4. Do all the time y = a + b + c. Some values of y will come from adding values (of a, b, and c) with different ranks.

3.3 Making y an external reference

In general, it is more desirable to have ant algorithms with *dependent* elements (such as those in §2.3.2) than independent elements (like those in §2.2), because the independent elements of higher rank "also go through lower ranks", so that, in some sense, lower rank values get computed several times, but they are not "brought to the surface" (that is, they are not stored into y). Exception to this rule: when the independent elements are fast.

For this reason, in general, it is preferable to *expose* the y value, in an iteration such as §2.3.1.1, in order to have it available all the time. This avoids independent elements. This avoids the problem of §2.5.1, where one chain produced only values with even ranks, and the other, only with odd ranks. A way to do this is to declare **y** (i. e., cell y) to be an external reference, or a shared variable, or a global variable, or a common variable (depending on the language), so that any body can read it at any time (warning: have **y** tell you when it has been read, so that you should stop computing *y* and start using the available idle time, as §3.1 dictates).

3.4 How to handle non numeric values as if they were numeric

In many cases, algorithms yielding non-numeric values can be anted, too, if the values can be arranged in a tree. For algorithms that work on image decompressing, big squares (low resolution) can be reconstructed first, albeit with not so much accuracy. Later, smaller parts will be resolved. For algorithms that work on quad trees, first process the big squares, then the middle squares, etc. An example will illustrate how to map the labels (non numeric values) into integers, so that the rank of the value can be obtained, and this paper can apply to these values.. Let us say you have an algorithm that locates some submarine or somebody very precisely in the world. You want to arrange into a tree the non-numeric values that describe geographic places, as follows. First, your algorithm selects the country. Being 150 countries in the world, reserve the first 8 bits (of the integer tag we are seeking) for them. Then, the country with more states (or provinces) is U.S.A., with 50. Reserve the next (less significant) 6 bits of y for them. Now, reserve 9 bits for the up to 512 municipalities that can exist in a state (Oaxaca has 400). Reserve the next 7 bits for up to 128 cities that can exist in a given municipality. And so on.

Then, to construct a tag for each location in the world, first assign the most significant 8 bits: arrange the 150 countries alphabetically (or by some other ordering criteria), give the number 1 to Abysinia, the number 2 to Afganisthan, ... Then, assign the following 6 bits, by ordering the states of each country (starting with country 1=Abysinia) and give them the numbers 1, 2, ... up to 64 (countries have less than 64 states). Do the same for the following 9 bits, to be assigned to the

municipalities. And so on. The resulting tags will be 8+6+9+7=30 bits long. This gives you a mapping from words or geographic concepts to integer tags, where "rank" has a definition that works as if it were the definition of §1.3.

Thus, the normal course of the algorithm "go from the coarse to the fine" can be translated into "go from low to higher ranks".

Warning: do **not** add or otherwise numerically manipulate these integer tags. They are **not** integers, they are just tags. But: the *rank* of these values **is** an integer.

3.5 Conclusions and comments

- Given a normal algorithm considered as a black box, use of **memoing** or caching (§2.1.B) and other techniques allows us to provide fast but approximate answers to its result, thus making it an any time (ant) algorithm.
- By defining rank and other measures, several manners are introduced for forming ant algorithms out of non ant or normal algorithms.
- Also, the paper introduces ways to mix several ant algorithms into new ones.
- A manner to use the idle time to compute rank, accuracy, etc., is given (§3.1).
- Finally, the paper gives some advice as how to parallelize ant algorithms.

3.5.1 Recommendations for further work

- Write a compiler to ant algorithms when there is no access to its code, along the lines of §2.2. That is, given E, it generates the ants of E J1 through J4 of §2.2.
- Write a compiler to ant algorithms when there is access to its code, along the lines of §2.3. That is, given E, it generates ant of E.
- Invent the *prepared* algorithms of §3.1: you would have invented a way to *store* computer time, that is, to use today's time for *tomorrow's* computations.
- Write a compiler that, given an ant algorithm A, generates code that computes the useful values (a) to (c) of §3.1, during the idle time of A –so you do not have to write this code (a) through (c) manually.
- Proceed further in the parallelization of ant algorithms (§3.2).
- Continue work on anting algorithms with non numerical values (§3.4).

3.6 Acknowledgments

To Eduardo Morales and Enrique Súcar [Súcar 94], who introduced me to new material in the area.

These ideas came to mind (although they are not really about agents) as a result of working for project xxxxx "Interaction of purposeful agents that use different ontologies" [Guzman 2000] sponsored by Conacyt.

Part of this work was produced under COTEPABE (IPN) sponsorship.

3.7 References

Guzmán, A. *Técnicas Modernas de Programación.* (1994) Book in Spanish, 300 pages. © 1994, SoftwarePro International; printed by Centro Nacional de Cálculo (IPN); limited edition.

Guzmán A. Estado del Arte y de la Práctica en Minería de Datos, Análisis y Crítica. (1997) *Memorias del II Taller Iberoamericano de Reconocimiento de Patrones*, 367-376. La Habana, Cuba. Marzo 24-28.

Adolfo Guzmán, Jesús Olivares, Araceli Demetrio and Carmen Domínguez, Interaction of purposeful agents that use different ontologies. (2000) *Lecture Notes in Artificial Intelligence 1793, MICAI 2000: Advances in A. I.* Osvaldo Cairo, Enrique Sucar, Francisco J. Cantu (eds). Springer Verlag. Pages 557-573. Also: CIC Technical Report 46, Blue Series, January 2000. ISBN 970-18-4132-8

Enrique Súcar. La referencia falta.