Paper presented at the VIII Geostatistical Congress, Santiago, Chili December 1-5, 2008. To appear in the proceedings.

OPTIMISATION ISSUES IN 3D MULTIPLE–POINT STATISTICS SIMULATION

J. STRAUBHAAR[1], A. WALGENWITZ[2], R. FROIDEVAUX[2], P. RENARD[1] and O. BESSON[3]

[1]CHYN, University of Neuchâtel, rue Emile–Argand 11, 2009 Neuchâtel, Switzerland,

[2]FSS Consultants, 9, rue Boissonnas, CH – 1227 Geneva, Switzerland,

[3]Institute of Mathematics, University of Neuchâtel, rue Emile–Argand 11, 2009 Neuchâtel, Switzerland

ABSTRACT

Multiple–point statistics simulation has gained wide acceptance in recent years and is routinely used for simulating geological heterogeneity in hydrocarbon reservoirs and aquifers. In classical implementations, the multiple–point statistics inferred from the reference training image are stored in a dynamic data structure called search tree. The size of this search tree depends on the search template used to scan the training image and the number of facies to be simulated. In 3D applications this size can become prohibitive. One promissing avenue for drastically reducing the RAM requirements consists of using dynamically allocated lists instead of search trees to store and retrieve the multiple–point statistics. Each element of this list contains the identification of the data event together with occurence counters for each facies. First results show that implementing this list based approach results in reductions of RAM requirement by a factor 10 and more. The paper discusses in detail this novel list based approach, presents RAM and CPU performance comparisons with the (classical) tree based approach.

INTRODUCTION

Multiple–point statistics simulation has been introduced by Guardiano and Strivastava (1993) and was further developed in recent years (Strébelle, 2002; Journel and Zhang, 2006; Arpat and Caers, 2007). The method allows to simulate (2D or 3D) heterogeneous geological facies images, respecting structures at small and large scales of a given reference training image and conditioning data. Following the algorithm proposed by Strébelle (2002), the training image is scanned with a search template and the multiple–point statistics are stored in a dynamically allocated tree. From this basic idea, the performances of multiple–point simulation algorithms were improved by addressing some critical issues such as: the post–processing (Strébelle and Remy, 2005), the choice of the simulation path (Daly and Knudby, 2007; Suzuki and Strébelle, 2007), the treatment of non–stationarity (Strébelle and Zhang, 2005; Chugunova and Hu, 2008).

In this paper, we come back to the basis of the multiple–point simulation algorithm and propose an alternative to the search tree for storing the multiple–point statistics. The motivation behind this proposal is the amount of memory required to store the search tree for a large image. Indeed, in 3D the size of the search tree can quickly become prohibitive and this imposes the use of small template sizes. The consequence of using such small templates is that complex structures are not properly reproduced. This is why, instead of using a search tree, we propose to use a list. This novel approach allows to drastically reduce the RAM requirements, and hence to perform simulations that were unmanageable with the tree based approach. Tests are performed with 3D images and performances in terms of RAM requirements and CPU time are compared for the two storage methods. We also discuss about RAM limitation.

CLASSICAL IMPLEMENTATION OF MULTIPLE–POINT STATISTICS

Let us recall here the basic principles of multiple–point statistics for simulating a facies at each node (pixel) of a grid. A *search template* is defined as a set of relative node locations $h_1, \ldots, h_N$, where h_i is a 2D or 3D vector,$1 \leqslant i \leqslant N$. For a given reference node u, the search template at u is the set of nodes

$$\tau(u) = \{u + h_1, \ldots, u + h_N\}, \tag{1}$$

and, if $s(v)$ denotes the facies at node v, the vector

$$d(u) = \{s(u + h_1), \ldots, s(u + h_N)\}, \tag{2}$$

defines the *data event* at u. Note that we can consider a data event with undefined components (for nodes not yet simulated). To attribute a facies at a node u in the simulated image, we retain the nodes v in the training image where the data event $d(v)$ has the same components as those of $d(u)$. Then, the number of occurences of all the facies at the nodes v is counted. This provides a probability distribution function (pdf) that can be used to draw a facies at the node u randomly.

In addition, the multigrid approach (Tran, 1994; Strébelle, 2002) allows to capture structures within the training image that are defined at different scales. For a situation with m levels of multigrid, all the unsimulated nodes whose coordinates are a multiple of 2^j are simulated considering the scaled lag vectors $2^j \cdot h_i$ instead of h_i in the search template, with $j = m - 1$, then $j = m - 2$,..., and finally $j = 0$.

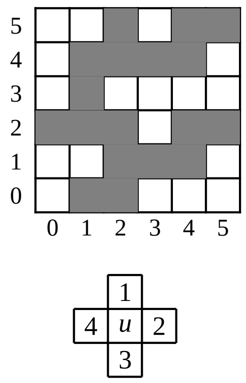
Search tree

Let us recall the definition of the search tree (Strébelle, 2002) for storing multiple–point statistics. The tree has a depth of N (size of the search template).

It is made up of cells divided in M (number of facies) subcells. The levels of the tree are numbered from 0 to N and the subcells in a cell from 0 to $M-1$. Each subcell allows to store a counter and can have a child cell (in the next level of the tree): the tree is an M–ary tree. The counter in a subcell is defined as follows. Let $\{i(0), i(1), ..., i(k)\}$ a path in the tree where $i(j)$ is the identification number of a subcell in a cell of level j. The counter in the last subcell of the path is the number of data events found in the training image with facies $i(j)$ at node $v+h_j$ of the the search template $\tau(v)$ for $0 \leqslant j \leqslant k-1$, and with facies $i(k)$ at the reference node v.

ALTERNATIVE TO SEARCH TREES: STORAGE IN LISTS

Instead of using the complete search tree, the multiple–point statistics inferred from the training image can be stored in a list. The idea is to store only the data events corresponding to the last level in the search tree. An element of the list is then a pair of vector (d, c) where $d = (s_1, \ldots, s_N)$ defines a data event and $c = (c_0, \ldots, c_{M-1})$ is a list of occurence counters for each facies: c_i is the number of data events $d(v)$ equal to d found in the training image with facies i at the reference node v.



Elements of the list	ref. nodes
$L_1 = ((0,0,1,1),(0,1))$	$\{\emptyset, \{(2,2)\}\}$
$L_2 = ((0,1,0,1),(0,2))$	$\{\emptyset, \{(3,1),(3,4)\}\}$
$L_3 = ((0,1,1,0),(0,2))$	$\{\emptyset, \{(1,4),(4,2)\}\}$
$L_4 = ((0,1,1,1),(1,0))$	$\{\{(3,2)\}, \emptyset\}$
$L_5 = ((1,0,0,0),(1,0))$	$\{\{(3,3)\}, \emptyset\}$
$L_6 = ((1,0,0,1),(0,2))$	$\{\emptyset, \{(4,1),(4,4)\}\}$
$L_7 = ((1,0,1,0),(1,1))$	$\{\{(4,3)\}, \{(1,3)\}\}$
$L_8 = ((1,0,1,1),(1,0))$	$\{\{(2,3)\}, \emptyset\}$
$L_9 = ((1,1,0,1),(0,2))$	$\{\emptyset, \{(1,2),(2,4)\}\}$
$L_{10} = ((1,1,1,0),(1,1))$	$\{\{(1,1)\}, \{(2,1)\}\}$

Figure 1: Training image (2 facies: white (0) and black (1)), search template and corresponding list.

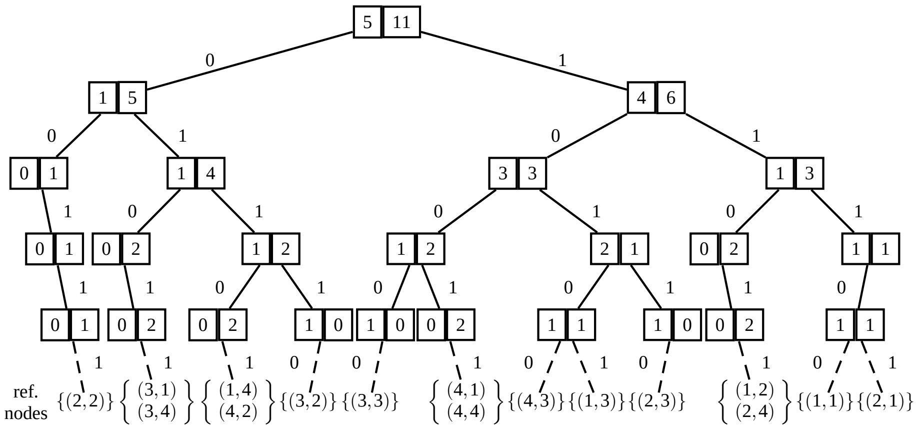


Figure 2: Search tree for the training image and the search template of figure 1.

Note that the total number of elements in the list is the number of different data events in the training image. To build the list (or the search tree), the training image is scanned such that the search template is always entirely included in the training image. At the end of the process, the statistics stored in the list and the search tree are identical. An important point to emphasize is that knowing the list allows to reconstruct exactly the search tree. See figures 1 and 2 for illustration.

RAM REQUIREMENTS

The size of the search tree and the list depends on the size N of the search template, the number M of facies and the entropy of the training image. Moreover, the size of the search tree can also depend on the order of the nodes numbering in the search template.

Let us compare the RAM requirements for the search tree and the list. Let s_{tree} be the size in octets of a tree cell (each subcell in a cell contains a counter and an address (child cell, eventually NULL)), s_{list} the size in octets of an element of the list, $r = s_{tree}/s_{list}$ their ratio and K_{tree} and K_{list} the number of cells in the tree and the number of elements in the list respectively. Then, the number

$$R = \frac{K_{tree}}{K_{list}} \cdot r \tag{3}$$

denotes the gain factor in terms of RAM usage for the list in comparison with the tree. In the lowest possible entropy case, there is only one data event in the training image and we have $K_{tree} = N+1$ (degenerated tree), $K_{list} = 1$ and $R = (N+1)\cdot r$. In the highest possible entropy case, all possible data events appear in the training image and we have $K_{tree} = 1 + M + M^2 + \ldots + M^N = (M^{N+1}-1)/(M-1)$ (full M–ary tree), $K_{list} = M^N$ and $R = (1-M^{-(N+1)})/(1-M^{-1})\cdot r$. It is clear that in all real cases, the gain factor is in the interval given by these two values of R. Moreover, assume that a counter is coded in a long integer of size 8, a facies of a data event is coded in a character of size 1 and an address is of size 8. Then, $s_{tree} = (8+8)\cdot M = 16\cdot M$ and $s_{list} = N + 8\cdot M$. Finally, we have

$$R \in \left[\frac{1-M^{-(N+1)}}{1-M^{-1}}\cdot r, (N+1)\cdot r\right], \text{ with } r = \frac{16\cdot M}{N+8\cdot M}. \tag{4}$$

For the previous example, $N = 4$, $M = 2$, $r = 1.6$ and $R = 3.84 \in [3.1, 8]$. For the more realistic 3D applications in next section, we have $N = 26$, $M = 4$, $r \approx 1.10$ and $R \in [1.47, 29.80]$ (first test) and $N = 124$, $M = 4$, $r \approx 0.41$ and $R \in [0.54, 51.29]$ (second test). Note that practically, the extreme cases are never reached.

NUMERICAL COMPARISONS

Let us consider a training image of dimensions $100\times 100\times 60$ ($600'000$ nodes) representing a fluviatile environment including 3D channels with $M = 4$ facies. The number of multigrid levels is set to $m = 4$. Two box–shaped search templates of dimensions $3 \times 3 \times 3$ (size $N = 26$) and $5 \times 5 \times 5$ (size $N = 124$) are used. The results for these two tests, presented in table 1, show that in terms of memory, the gain with the list is significant. Moreover, the lists for all multigrid levels can be stored in the RAM simultaneously, and hence several images can be simulated computing the lists once, whereas this is not always the case using trees.

Table 1: Ram usage.

Multigrid	K_{tree}	Tree size	K_{list}	List size	R
First test: $M = 4$, $N = 26$ ($r \approx 1.10$)					
1	941′537	57.47 MB	94′896	5.25 MB	10.95
2	923′839	56.39 MB	90′461	5.00 MB	11.27
3	534′732	32.64 MB	53′427	2.96 MB	11.04
4	246′204	15.03 MB	24′427	1.35 MB	11.12
Second test: $M = 4$, $N = 124$ ($r \approx 0.41$)					
1	10′721′840	654.41 MB	125′538	18.68 MB	35.04
2	18′522′678	1′130.53 MB	238′416	35.47 MB	31.87
3	15′432′211	941.91 MB	211′050	31.40 MB	30.00
4	10′075′311	614.95 MB	151′235	22.50 MB	27.33

Let us discuss in more details about RAM limitation. For this, we consider a training image and a search template that contains the N closest nodes to the reference node and we increase the size N. For each search template, we build the corresponding search tree and list, and we retrieve their size in gigabytes (GB). The results are presented on graphs in figure 3 for the previous 3D training image (a) and a training image of dimensions $101 \times 101 \times 1385$ with 6 facies (b). The gain factor for these two cases is drawn on graph (c). These tests show that a fixed amount of RAM is reached much faster for the tree than for the list: for example, in the case (a), 512 MB is reached with $N = 115$ and $N = 1962$ for the tree and the list respectively and, in the case (b), the same is true with 2 GB, $N = 30$ and $N = 169$.

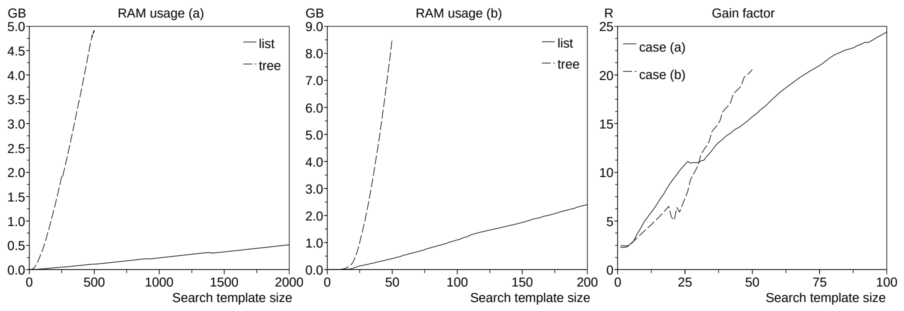


Figure 3: (a) RAM usage for a 3D training image having 600′000 nodes and 4 facies; (b) RAM usage for a 3D training image having 14′128′385 nodes and 6 facies; (c) Gain factor with the list in comparison with the search tree.

Let us also compare the CPU performances for the two storage methods. For the two tests in table 1, a single realization is generated and the used CPU time is retrieved. For the first test, 219 *s* and 290 *s* are needed using trees and lists respectively, and for the second test, we obtain 43 *min* and 109 *min*. The two methods lead to the same simulated images provided that the path and the random numbers used in both simulations are identical. Since less information is available in lists than in trees, a simulation requires more CPU time using lists than trees. However, the ratio of these CPU times is inferior to 3 (with the $9 \times 9 \times 5$ search template ($N = 404$), 2 *h* 30 *min* are needed using trees and 5 *h* 18 *min* using lists).

CONCLUSIONS

More statistics are stored in search trees than in lists, and hence, the tree method is less CPU demanding than the list method if the RAM capacity allows to store the search trees. For large (3D) training images however, the simulations become unmanageable due to the size of the search trees. The storage of statistics in lists drastically reduce the RAM requirement and allows to complete simulations in such cases. The classical implementations are quickly limited by the RAM capacity of computers. On the other hand, the list based approach is not very affected by RAM limitation and then the search templates can be chosen large enough. For large applications, the RAM requirement is deciding and only the list based approach allows to complete acceptable simulations.

ACKNOWLEDGEMENTS

We thank the Swiss Confederation's Innovation Promotion Agency (CTI project No. 8836.1 PFES–ES) and the Swiss National Science Foundation (grant No. PPOOP2–106557) for supporting this research.

REFERENCES

Arpat, G and Caers, J (2007). *Conditional simulation with patterns*. In Mathematical Geology, vol. 39, no. 2, pp. 177–203.

Chugunova, T and Hu, L (2008). *Multiple-point statistical simulations constrained by continuous auxiliary data*. In Mathematical Geosciences, vol. 40, no. 2, pp. 133–146.

Daly, C and Knudby, C (2007). *Multipoint statistics in reservoir modelling and in computer vision*. In *Petroleum Geostatistics 2007, Cascais, Portugal*.

Guardiano, F and Strivastava, R (1993). *Multivariate geostatistics: beyond bivariate moments*. In A Soares, ed., *Geostatistics Troia*, vol. 1. Kluwer Academic, Dordrecht, pp. 133–144.

Journel, A and Zhang, T (2006). *Necessity of a multiple-point prior model*. In Mathematical Geology, vol. 38, no. 5, pp. 591–610.

Strébelle, S (2002). *Conditional simulation of complex geological structures using multiple-points statistics*. In Mathematical Geology, vol. 34, no. 1, pp. 1–21.

Strébelle, S and Remy, N (2005). *Post-processing of multiple-point geostatistical models to improve reproduction of training patterns*. In CD O Leuangthong, ed., *Geostatistics Banff 2004*. Springer, pp. 979–988.

Strébelle, S and Zhang, T (2005). *Non-stationary multiple-point geostatistical models*. In CD O Leuangthong, ed., *Geostatistics Banff 2004*. Springer, pp. 235–244.

Suzuki, S and Strébelle, S (2007). *Real-time post-processing method to enhance multiple-point statistics simulation*. In *Petroleum Geostatistics 2007, Cascais, Portugal*.

Tran, TT (1994). *Improving variogram reproduction on dense simulation grids*. In Computers & Geosciences, vol. 20, no. 7, pp. 1161–1168.