Mixing Tree and List Structures for Multiple-Point Statistics Simulations

Julien Straubhaar and Alexandre Walgenwitz

Abstract Multiple-point statistics are widely used to simulate complex heterogeneous fields. The technique consists in inferring multiple-point statistics of a categorical variable from a training image. For 3D problems with numerous facies, large templates should be used for reproducing complex structures properly. Tree structures used in classical implementations for storing the multiple-point statistics of the training image are very RAM demanding and then imply limitations on the size of the template. On the contrary, using a list structure allows to obtain a straightforwardly parallelizable algorithm requiring a small amount of RAM. Nevertheless, retrieving statistics from a search tree is more efficient in terms of CPU time, because the shortcuts given by the branches of the tree are not present in the list. In this paper, we propose a new technique mixing list and tree structures for storing multiple-point statistics inferred from the training image. The idea is to build a tree of reduced size whose leaves are sub-lists that constitute the entire list when gathering them. This approach benefits from the advantages of both storage techniques: low RAM requirements are guaranteed by the list structure, while improved efficiency in terms of CPU time is provided by the tree structure and the parallelization. Numerical tests are performed for comparing the different methods and presented in this paper.

Julien Straubhaar
The Centre for Hydrogeology and Geothermics (CHYN), University of Neuchâtel, Emile-Argand 11, CH-2000 Neuchâtel, Switzerland, e-mail: julien.straubhaar@unine.ch

Alexandre Walgenwitz
Ephesia Consult, Boissonnas 9, CH-1227 Geneva, Switzerland, e-mail: alexandre.walgenwitz@ephesia-consult.com

1 Introduction

Multiple-point statistics (MPS) allows to stochastically generate categorical variables whose spatial structures are provided by a training image. This method is very interesting because the spatial characteristics of the structures such their size, shape and connectivity have an important impact on flow and transport processes [2, 3].

Multiple-point statistics was introduced by Guardiano (1993) [1] and the first efficient algorithm called *snesim* was developed by Strebelle (2002) [5] who proposed to store the multiple-point statistics inferred from the training image in a search tree. Indeed, each pattern within a given template found in the training image is stored along a path in the tree, whose the number of levels is the size of the template. Hence, the RAM requirements for this method grow very fast in function of the size of the template. Using lists instead of search trees, *i.e.* storing only the leaves of the tree, allows to drastically reduce the RAM usage [4]. Moreover the list-based algorithm is easily parallelizable since a list can be divided in several part in an obvious way. However, since the branches of the tree are very useful for retrieving statistics, the algorithm using lists in this serial version is generally slower.

In this paper, we propose to mix list and tree structures for storing multiple-point statistics. This new method consists to build the list of all pattern found in the training image for a given template, and then to build a tree indexing the entries in the list, according to the nodes of the template. The size of the tree is controlled by two parameters and then the RAM requirements are still low. The resulting algorithm is still parallelizable: if a list is divided into several parts, a tree is built for each part. Hence, this new approach gives an improved algorithm in terms of CPU time demanding low RAM requirements.

2 Storing multiple-point statistics in lists

The multiple-point statistics used for the simulations are provided by a training image and a search template. Assume that M is the number of facies present in the training image and that the facies codes 0 to $M-1$ are used; let N be the size of the search template, and $\tau(u) = \{u+h_1, \ldots, u+h_N\}$ be the search template centered at a node u, *i.e.* the search template is defined by the lag vectors $h_1, \ldots, h_N$ in 2D or 3D. The list consists in a catalogue containing every distinct pattern found in the training image. With the notations above and if $s(v)$ denotes the facies code at a node v, an element of the list is a pair of vector $(d = (s_1, \ldots, s_N), c = (c_0, \ldots, c_{M-1}))$, where c_i is the number of occurrences of the data event $\{s(v+h_1) = s_1, \ldots, s(v+h_N) = s_N\}$ found in the training image with the facies code $s(v) = i$ at the central node v [4]. An illustration of the list storage technique is given in the Figs. 1 and 2. In this example, only the locations of the central node v for which the search template $\tau(v)$ is entirely inside the training image are taken into account. Note that the patterns centered at a node in the boundary of the training image can be considered by introducing a new facies code for nodes that are outside of the training image. Then, the list is used

to compute the conditional probability distribution function required by the simulation process, *i.e.* the probability to draw each facies at a node u conditioned by the simulated nodes in the search template centered at u.

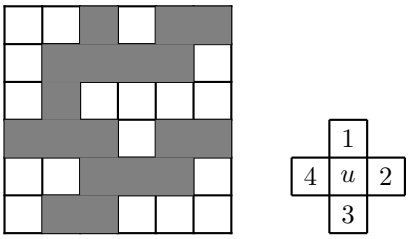

Fig. 1 Training image and search template

List elements
$L_0 = ((0,0,1,1),(0,1))$
$L_1 = ((0,1,0,1),(0,2))$
$L_2 = ((0,1,1,0),(0,2))$
$L_3 = ((0,1,1,1),(1,0))$
$L_4 = ((1,0,0,0),(1,0))$
$L_5 = ((1,0,0,1),(0,2))$
$L_6 = ((1,0,1,0),(1,1))$
$L_7 = ((1,0,1,1),(1,0))$
$L_8 = ((1,1,0,1),(0,2))$
$L_9 = ((1,1,1,0),(1,1))$

Fig. 2 List for the training image and the search template of the Fig. 1, with the search template always entirely inside the training image, facies code 0 for white node and facies code 1 for gray nodes

3 Indexing the list by a tree

The simulation of a node u requires to retrieve the counters c of each element of the list for which the data event d is compatible with the already simulated nodes in the search template $\tau(u)$. To do this, the list is scanned entirely. To avoid an exhaustive scan of the list, the idea is to index the list by a tree. The list is sorted lexicographically according to the data events and the branches of the tree are defined according to the facies code of the data event as for the usual search tree in *snesim* [5], and the cells of the tree contains the range of the corresponding elements in the lists. For a situation with M facies, the tree is an M-ary tree made up of cells divided in M sub-cells which can have a child cell. If the levels in the tree are numbered from 1 and the sub-cells in a cell from 0 to $M-1$, one can locate a sub-cell by a path $\{i(1), i(2), \ldots, i(k)\}$ in the tree, $i(j)$ being the identification number of a

sub–cell in a cell of level j. At the location given by the above path, the sub-cell contains the index of the first element in the list for which the data event d begins by $i(1), i(2), \ldots, i(k)$, and one plus the index of the last of those elements. Such a tree can be cut everywhere and is useful for reducing the range of the elements of the list that must be scanned to retrieve the counters according to a given data event partially informed. An example for the list of the Fig. 2 is given in the Fig. 3.

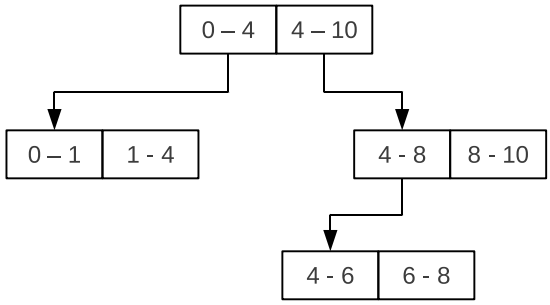

Fig. 3 Tree indexing the list of the Fig. 2

3.1 Controlling the size of the tree

Two parameters allows to control the size of the tree:

- d_{max}: maximal depth of the tree,
- s_{max}: maximal size for the sub-lists given by the leaves of the tree.

Note that the depth of the tree is defined as the number of levels minus one, and that the root of the tree is on level 1. Then, the tree is built by respecting the following rule.

A sub-cell in the tree has a child cell if and only if it is in a level $l < 1 + d_{max}$ and the size of the sub-list given by its indices is greater than s_{max}.

Note that if $d_{max} \geqslant N - 1$, where N is the size of the search template, then there is no constraint on the depth of the tree. The parameter s_{max} must be greater or equal to 1. For the tree of the Fig. 3, the parameter s_{max} is set to 3 and the parameter d_{max} to a number greater or equal to 2.

4 Numerical tests

In this section, the new approach is tested (with serial code) on two examples involving two training images. The parameter d_{max} is set to a big number so that only the parameter s_{max} is relevant in the previous rule for building the indexing tree. Then, we retrieve the real time spent for 10 realizations and the memory load when using the new algorithm for varying values of s_{max} and when using the pure list-based algorithm. Note that the realizations obtained are the same for all tests.

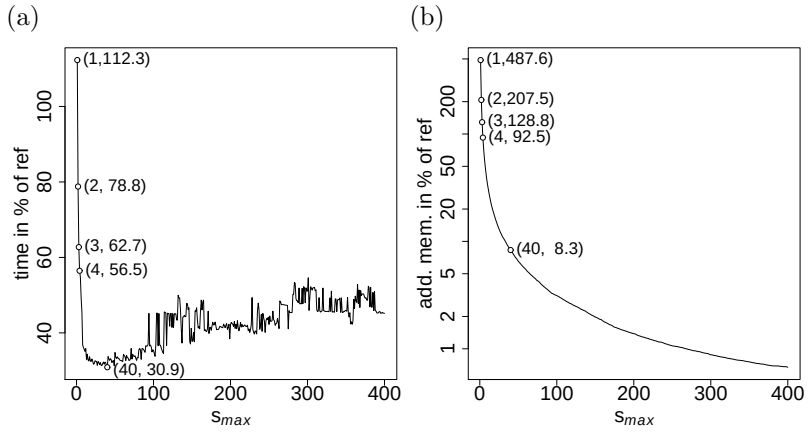

Fig. 4 Time and memory requirements for the new algorithm (with $d_{max} = 99$) compared to the pure list-based algorithm, for example (I); (a) real time in percent of the reference time (list-based algorithm), as a function of s_{max}; (b) additional amount of memory used for storing the indexing trees in percent of the amount of memory needed for the lists, as a function of s_{max} (logarithm-scale for vertical axis); the points for first values and for the minimum real time are shown in (a) and (b)

We consider the two examples:

(I) 2D training image of size 250×250 with 2 facies, simulation grid of size 300×300, 3 multi-grids, and disc-shape search templates containing 100, 60 and 20 nodes from the coarse multi-grid to the fine one;
(II) 3D training image of size $100 \times 100 \times 60$ with 4 facies, simulation grid of size $50 \times 50 \times 40$, 3 multi-grids, and spherical search templates containing 514, 256 and 32 nodes from the coarse multi-grid to the fine one.

The pure list-based algorithm is considered as the reference. The real time obtained when using the new algorithm is expressed in percent of the reference time (list-based algorithm) and the additional amount of memory used for storing the indexing trees in percent of the amount of memory needed for the lists. These results are plotted as a function of the parameter s_{max} in the Figs. 4 and 5 for the examples (I) and (II) respectively. The values for time and for memory usage are plotted on a same graph in the Fig. 6. For the two examples, a similar behaviour is observed

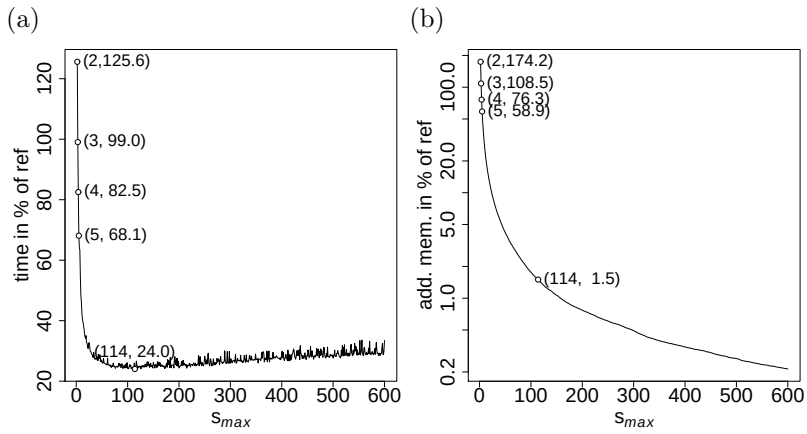

Fig. 5 Time and memory requirements for the new algorithm (with $d_{max} = 513$) compared to the pure list-based algorithm, for example (II); (a) real time in percent of the reference time (list-based algorithm), as a function of s_{max}; (b) additional amount of memory used for storing the indexing trees in percent of the amount of memory needed for the lists, as a function of s_{max} (logarithm-scale for vertical axis); the points for first values and for the minimum real time are shown in (a) and (b)

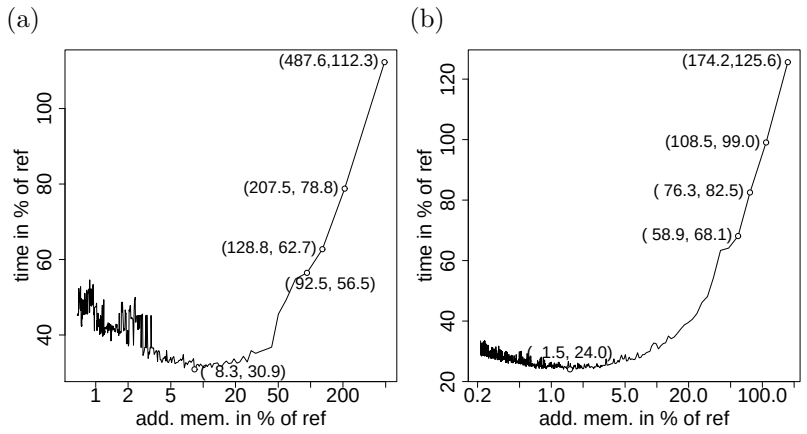

Fig. 6 Time VS memory requirements for the new algorithm (logarithm-scale for horizontal axis); (a) results for example (I) (Fig. 4); (b) results for example (II) (Fig. 5); the points for first values and for the minimum real time are shown in (a) and (b)

(similar curve shapes). In the two cases, the Fig. 6 shows that the spent time can be divided by 3 for an additional amount of memory of less than 5% related to the pure list-base algorithm.

Note that when using classical search trees (as in *snesim* [5]), we approximately obtain a real time of 140% and 90% of the reference real time for the examples (I) and (II) respectively, whereas the memory load is multiplied by a factor of more that 10 and 40 respectively.

5 Conclusions

The tests presented in this paper show that mixing list and tree structures for storing multiple point statistics allows to significantly improve multiple-point statistics algorithms. The tree-based algorithm is limited because of the important RAM usage. The list-based approach overcomes this RAM limitation but its serial version is generally slower. Indexing the lists by trees allows for a substantial gain of time resulting in better performances than tree-based algorithm, while the RAM usage remains low. Moreover, this new technique benefits from the parallelization that can be applied to the lists.

References

1. F. Guardiano and R. Strivastava. Multivariate geostatistics: beyond bivariate moments. In A. Soares, editor, *Geostatistics Troia*, volume 1, pages 133–144. Kluwer Academic, Dordrecht, 1993.
2. A. Journel and T. Zhang. Necessity of a multiple-point prior model. *Math Geol*, 38(5):591–610, 2006.
3. P. Renard. Stochastic hydrogeology: what professionals really need? *Ground Water*, 45(5):531–541, 2007.
4. J. Straubhaar, P. Renard, G. Mariethoz, R. Froidevaux, and O. Besson. An improved parallel multiple-point algorithm using a list approach. *Math Geosci*, 43:305–328, 2011. DOI: 10.1007/s11004-011-9328-7.
5. S. Strebelle. Conditional simulation of complex geological structures using multiple-points statistics. *Math Geol*, 34(1):1–21, 2002.