ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/260011432

FGNG: A fast multi-dimensional growing neural gas implementation

Article *in* Neurocomputing · March 2014

DOI: 10.1016/j.neucom.2013.08.033

CITATION
1

READS
357

3 authors:

Carlos Mendes
Pontifícia Universidade Católica do Rio de Ja...
1 PUBLICATION **1** CITATION
SEE PROFILE

Marcelo Gattass
TECGRAF / PUC-Rio
212 PUBLICATIONS **1,050** CITATIONS
SEE PROFILE

Hélio Lopes
Pontifícia Universidade Católica do Rio de Ja...
100 PUBLICATIONS **898** CITATIONS
SEE PROFILE

All content following this page was uploaded by Marcelo Gattass on 04 September 2014.

The user has requested enhancement of the downloaded file. All in-text references underlined in blue are added to the original document and are linked to publications on ResearchGate, letting you access and read them immediately.

Neurocomputing 128 (2014) 328–340

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

FGNG: A fast multi-dimensional growing neural gas implementation

Carlos Augusto Teixeira Mendes *, Marcelo Gattass, Hélio Lopes

Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil

ARTICLE INFO

Article history:
Received 21 December 2012
Received in revised form
1 August 2013
Accepted 30 August 2013
Communicated by F. Rossi
Available online 18 October 2013

Keywords:
Growing neural gas
Multi-dimensional data
Spatial data structures
Algorithms

ABSTRACT

The Growing Neural Gas algorithm (GNG) is a well-known classification algorithm that is capable of capturing topological relationships that exist in the input data. Unfortunately, simple implementations of the GNG algorithm have time complexity $O(n^2)$, where n is the number of nodes in the graph. This fact makes these implementations impractical for use in production environments where large data sets are used. This paper aims to propose an optimized implementation that breaks the $O(n^2)$ barrier and that addresses data in high-dimensional spaces without changing the GNG semantics. The experimental results show speedups of over 50 times for graphs with 200,000 nodes.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

The Growing Neural Gas algorithm (GNG), which was proposed by Fritzke [10], is a well-known unsupervised classification algorithm; its main characteristic is the capability of capturing topological relationships that exist in the input data. GNG is used in several applications including reverse engineering [13], power systems [17] and seismic modeling [7], to cite only a few.

Unfortunately, simple implementations of the GNG algorithm require an amount of time that is proportional to $O(n^2)$, where n is the number of nodes in the graph, which makes them impractical for use in production environments where large data sets are required. As an example, Fig. 1 illustrates a graph (on the right) that was generated by the GNG algorithm while using a small volume ($64 \times 106 \times 99$ samples) of fault attributes as input with a probability distribution function (partially shown on the left) [7]. Such a graph is later processed by other means to automatically identify its geologic faults. In practice, complete volumes of seismic data are much larger than the data sets shown in Fig. 1, regularly reaching sizes on the order of $2000 \times 2000 \times 1000$ samples, which makes the execution times impractical for applying the GNG algorithm.

In addition, the dimension of the feature space that is used for grouping the input data into regions can substantially impact the GNG efficiency [14].

Contributions: In this study, we propose the Fast Growing Neural Gas (FGNG) algorithm, which breaks the $O(n^2)$ barrier with a generic and optimized implementation that performs an order of magnitude faster than the simple (previous) implementations of the GNG. It is important to note that the FGNG does not change the GNG semantics, which means that the FGNG algorithm is suitable to be used whenever the GNG algorithm is applicable.

In contrast to Fišer et al. [8], who proposed a solution that is based on a grid structure to optimize the nearest neighbor search, we adopt spatial data structures that better scale to higher dimensions where the grid strategy is prohibitive.

Paper outline: Section 2 provides an overview of the GNG algorithm and shows that simple implementations are in fact $O(n^2)$. Section 3 includes a brief review of related studies. Section 4 presents the FGNG algorithm. Section 5 elaborates on nearest neighbors search algorithms, a fundamental part of the FGNG. Section 6 presents some experiments and discusses their analysis. Finally, Section 7 concludes the paper and notes directions for future research.

2. Growing Neural Gas

The Growing Neural Gas algorithm (GNG), proposed by Fritzke [10], is a competitive learning algorithm that is capable of capturing topological relationships that are present in the input data; it combines the Competitive Hebbian Learning (CHL) edge creation rule with the Growing Cell Structures node creation strategy.

From a set of samples, the GNG builds a graph whose nodes are associated with a position in a k dimensional space ($\mathbb{R}^k$). Each of these nodes represents the subset of samples that are "close" to it.

* Corresponding author. Tel.: +55 2191460248.
E-mail addresses: cteixeira@inf.puc-rio.br (C.A.T. Mendes), mgattass@inf.puc-rio.br (M. Gattass), lopes@inf.puc-rio.br (H. Lopes).

0925-2312/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.neucom.2013.08.033

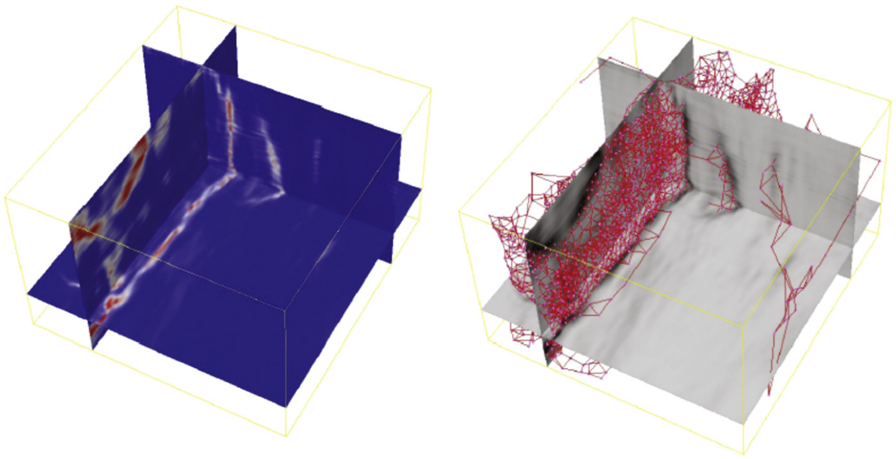


Fig. 1. A GNG-generated graph created from seismic fault attributes. (Adapted from [7].)

To assess the error that is associated with this representation, each node keeps a measure of the incurred error by representing several samples by a single node in the graph. Additionally, each edge on the graph saves age information that is used in the GNG algorithm, to later remove the "senile" edges.

The output graph generated by the GNG is built incrementally. From the input data set, a new sample is selected according to a given probability distribution, and a search is performed for the closest node to the new input sample. This node is called the *winning* or *activated* node, and it will suffer an adaptation process, moving toward the sample and having its error measure updated. Periodically, new nodes are added to the graph to minimize the representation error. The GNG algorithm is composed of 10 steps, which are described below. For a more in-depth explanation of the GNG algorithm specifically or the competitive learning processes in general, the reader is referred to [10].

Initiation: Create the graph by adding two samples as independent nodes. The edge set is initialized to be empty.

Step 1: Get another sample S. Find, in the graph, the closest node to S, which is N_a, and the second closest node, which is N_b.

Step 2: Update N_a's error measure

$$error(N_a) = error(N_a) + distance(N_a, S)^2.$$

In this step, the local error measure from the activated node is adjusted. The distance function is the Euclidean distance between the sample and N_a's position in $\mathbb{R}^k$. This local measure is used to determine where new nodes should be inserted. A high error value means that the node has been activated frequently and as a consequence we should refine our representation of the region around this node.

Step 3: Adjust the position of N_a and also of all of its neighbors:

$$position(N_a) = position(N_a) + \varepsilon_b \cdot (position(S) - position(N_a))$$

$$position(x) = position(x) + \varepsilon_n \cdot (position(S) - position(x)); \quad \forall \times \in neighbors(N_a)$$

This step corresponds to the adaptation step in which the activated node N_a and all of its neighbors are moved in the direction of sample S. Parameters ε_b and ε_n control the movement's magnitude.

Step 4: Increment the age of all of the edges between N_a and its neighbors.

Step 5: Connect N_a to N_b. If both of the nodes were already connected, then reset the age that is associated with this edge to zero.

This step implements the CHL rule: for each new sample, add a new edge that connects the two closest nodes from the sample.

Step 6: For each incident edge e on N_a, if $age(e) > a_{max}$, then remove the edge e. If either node N_a or its incident neighbor became isolated from the remainder of the graph, then remove that node from the graph.

The aim here is to remove from the graph edges that are no longer useful because they were replaced by younger and shorter edges that were created during the graph refinement in Step 7.

Step 7: If the current iteration is a multiple of λ, then do the following:

7.1 Find the node N_u with the largest error.
7.2 Find the node N_v with the largest error among N_u's neighbors.
7.3 Disconnect N_u from N_v.
7.4 Create a new node N_r between N_u and N_v with the following:

$$position(N_r) = (position(N_u) + position(N_v))/2.$$

7.5 Connect N_r to N_u and to N_v.
7.6 Adjust the error from N_u and N_v, multiplying both by a reducing factor α:

$$error(N_u) = error(N_u) \cdot \alpha$$

and

$$error(N_v) = error(N_v) \cdot \alpha.$$

This equation is the growing step for the graph. For every λ steps, the graph is refined by the inclusion of a new node in the vicinity of the node with the largest error; as a consequence, the algorithm puts effort into an area that requires additional refinement.

Step 8: Multiply the error of every node by $(1-\beta)$.

This step caters to the fact that errors due to recent steps should have a greater weight than older errors, which refer to a less refined graph.

Loop: If the stopping criterion has not been met, then repeat the outlined procedure from Step 1. The usual stopping criterion is to verify whether the graph already has a predetermined number of nodes.

Typical values for the parameters used in the algorithm are listed in Table 1.

For our preliminary complexity analysis of the GNG algorithm, we will consider a simple implementation that performs sequential searches to find the closest node to a sample (Step 1) and also to find the node with the greatest error (Step 7). In the following analysis, we will consider that n is the number of nodes in the graph, k is the space dimension, and also that

- The number of neighbors of each node is much smaller than n. Typically in the 2D case, it is very unusual to have nodes with more than 5 or 6 edges. For higher dimensions, the valency of a vertex is usually more than 6, but it is still small when compared to n. This finding is a result of the CHL rule, which guarantees that the generated graph corresponds to an induced Delaunay triangulation [10]. An immediate consequence of this assumption is that we can consider any operation on all of the neighbors of a node to be $O(1)$.
- The number of node removal operations is very small. This hypothesis, which has been verified in practice, allows us to simplify our analysis by considering that a graph with n nodes was generated by $\lambda \cdot n$ iterations of the GNG algorithm. Without that assumption, we would have to add an additional term that refers to the removed nodes and consider that a graph with n nodes was generated by $\lambda \cdot (n+r)$ iterations, where r is the number of removed nodes.

With these hypotheses in mind, Table 2 summarizes in a nutshell the complexity of each individual step of the GNG algorithm. Operations that compare two samples or update a sample have time proportional to k because all of the dimensions must be traversed. When working with a problem with a fixed dimension problem, k can be seen as a constant and, thus, it can be removed from the complexity expression.

Clearly, Steps 1 and 8 are executed $\lambda \cdot n$ times and Step 7 is executed n times. Therefore, the complete algorithm has a complexity given by $\lambda \cdot n \cdot O(k \cdot n) + n \cdot O(k+n) = O(k \cdot n^2)$.

3. Related studies

The idea of improving the GNG algorithm performance is not new. As seen in Section 2, Steps 1, 7 and 8 are the ideal candidates to optimize the time complexity. Several studies have explored modifications to these steps, either by proposing modifications to the core GNG algorithm or by adding access structures to improve efficiency.

To create the nodes in a faster way, independent of the λ parameter, the Incremental Growing Neural Gas algorithm (IGNG) [16] proposes a change over the original process to allow nodes to be created automatically when new samples are far away from all of the other sample. By taking this action, no error accounting is necessary, and Steps 7 and 8 are naturally eliminated from the original GNG algorithm. This idea is further developed in the Density-Based Growing Neural Gas method (DB-GNG) [15], where the same basic idea is used, but the nodes are now included in the graph only if the sampled region is sufficiently dense. To improve the nearest neighbor search that is conducted in Step 1, an R-Tree is used.

The use of R-Trees for GNG acceleration was proposed earlier in [6], but this work explores only the nearest neighbor aspect of GNG optimization and considers only spaces that have up to 3 dimensions; as a result, the algorithm remains $O(n^2)$ due to Steps 7 and 8.

Fišer et al. present recently [8] an interesting proposal that uses a grid-based structure to optimize the NN search. The proposed algorithm also uses a heap structure and a form of delayed error reduction that is very similar to our proposal. The main disadvantage of using a grid-based access method over tree access methods is that it scales poorly in terms of efficiency and memory use when working with higher dimensional spaces.

To the best of our knowledge, no previous work attempts to optimize an unmodified GNG algorithm by applying both the use of tree access methods and heap-based optimizations. Additionally, none of the previous articles analyzes the multidimensional problem or evaluates the effects of the different parameters that can greatly affect the R-Tree performance such as the maximum number of children per node and the applied nearest neighbor algorithm.

4. Fast Growing Neural Gas

4.1. Overview

Section 2 presented a traditional implementation of the GNG algorithm with an $O(k \cdot n^2)$ time complexity, which makes it unfeasible to apply it to large data sets. In this section, we propose the adoption of a set of techniques and suitable data structures to reduce the time complexity order, without changing the original GNG semantics. In our algorithm we modify Steps 1, 7 and 8 of the GNG algorithm as described in Table 2, because those steps are responsible for making that implementation of the original GNG algorithm to have a high time complexity.

The first column in Table 3 summarizes the steps for the FGNG algorithm: the differences from the GNG description in Table 2 are shown in bold typeface. Sections 4.2–4.4 describe in detail the key modifications performed in Steps 1, 7 and 8. Next, Section 4.5

Table 1
Typical values for the GNG parameters. (Adapted from [7].)

Parameter	Description	Value
λ	Cycle interval between node insertions	300
ε_b	Winning node adaptation factor	0.05
ε_n	Winning node, neighbor adaptation factor	0.0006
α	Error reduction factor upon insertion	0.05
β	Error reduction factor for each cycle	0.0005
a_{max}	Oldest age allowed for an edge	80

Table 2
The basic GNG implementation, with the complexity of the individual steps.

Repeat the following steps $\lambda \cdot n$ times	
(1) Get a new k-dimensional sample and find the two closest nodes to this sample (N_a and N_b);	$O(k \cdot n)$
(2) Update N_a's error;	$O(k)$
(3) Adjust the position of N_a and of all of its neighbors;	$O(k)$
(4) Increment the age of the edges between Na and all of its neighbors;	$O(1)$
(5) Connect N_a to N_b or clear the age on the edge between them if both nodes were already connected;	$O(1)$
(6) Remove older edges. Remove isolated nodes;	$O(1)$
(7) If the current iteration is a multiple of λ:	
Find the node N_u with the largest error;	$O(n)$
Find the node N_v, neighbor to N_u, with the largest error;	$O(1)$
Create a new node N_r between N_u and N_v. Connect it to both N_u and N_v;	$O(k)$
Adjust the error of N_u and N_v;	$O(1)$
(8) Reduce the error on every node, multiplying each one by $(1-\beta)$;	$O(n)$

presents a review of the new time complexity order, which is summarized in the second column of Table 3.

4.2. Step 1: search for the nearest neighbors

In the first step of the algorithm, the critical operation constitutes finding the two closest nodes in the graph to the sample S. To accelerate this operation, it is necessary to adopt a spatial index to improve the efficiency of the search for the nearest neighbor instead of using a bare $O(k \cdot n)$ sequential search. The price is that this index will have to be updated whenever a new node is inserted in the graph (Step 7), removed (Step 6) or has its position changed (Step 3).

There are several search structures that have been proposed in the literature. Because the graph is built incrementally, the chosen structure should be dynamically constructible. The algorithm should also allow for data insertion, removal and updating as well as searching for the two nearest neighbors (2NN, an instance of the KNN, K-Nearest Neighbors, problem) in a more efficient way than a sequential search (i.e., these operations should perform better than $O(k \cdot n)$).

Many search tree options were analyzed in the present study. Despite the fact that kd-Trees are highly efficient for search operations, they were ruled out because the structure should be created “a priori” and not dynamically.

Quadtrees and Octrees are interesting options, but they can only treat problems with fixed dimensionality for a node position. Because this work aims to contemplate higher dimensions than the traditional 2D and 3D cases, these approaches were also ruled out. Theoretically, it should be possible to work with a 2^k-Tree, generalizing the concept, but with more than 3 dimensions; the cost of having a fixed number of 2^k children per node quickly becomes prohibitive.

Grid based structures [8] can be an interesting solution for 2D or 3D problems because the structure can be updated in time proportional to $O(k)$; however, they suffer from the same memory problem observed for 2^k-Trees. A small grid with only 10 cells per dimension would need 10^8 cells in an 8-dimensional scenario.

For this work, we have chosen to use of an R-Tree [11], a type of balanced search tree that is heavily used for spatial indexing in data bases that can address any number of dimensions. An R-Tree is a structure that is very similar to the traditional B-Tree; it possesses two parameters, M and m, which define the maximum and minimum number of children for each tree node. Every indexed data point is found in a tree leaf, and all of the leaves are on the same level, which creates a guarantee that the tree is balanced with a height that is proportional to $\log(n)$, with n being equal to the number of points added to the tree.

Every tree node stores the minimum bounding box for each of its children, forming a hierarchy of bounding boxes in which every child is contained inside the bounding box of its father. One point to emphasize is that there can be overlaps between the bounding boxes of “brother” nodes, as seen in Fig. 2, which shows a sample tree organization on the left and the bounding boxes for the leaves and for each internal node on the right.

The insertion of points into an R-Tree with n points in a k dimensional space is an operation with order $O(k \cdot \log\ n)$ operations, where the k constant appears because it compares the positions of two nodes in $O(k)$ operations. The removal operation is of order $O(k \cdot \log\ n)$ in general, but it can have a worst case

Table 3
The FGNG implementation, with the complexity of the individual steps.

Repeat the following steps $\lambda \cdot n$ times	
(1) Get a new k-dimensional sample and find the two closest nodes to this sample (N_a and N_b), **using an R-Tree to speedup the search**;	$O(s(k,n))$
(2) Update N_a's error. **Update the error index heap;**	$O(k)+O(\log\ n)$ - binary heap or $O(k)$ + Amortized $O(1)$ - Fibonacci heap
(3) Adjust the position of Na and all of its neighbors. **Adjust changed positions on the R-Tree**;	$O(k \cdot \log\ n)$
(4) Increment the age of the edges between Na and all of its neighbors;	$O(1)$
(5) Connect N_a and N_b or clear the age on the edge between them if both nodes were already connected;	$O(1)$
(6) Remove older edges. Remove isolated nodes from the graph, **from the R-Tree and from the error index heap**;	$O(k \cdot \log\ n)$
(7) If the current iteration is a multiple of λ:	
Find the node N_u with the largest error **using the error heap**;	$O(1)$
Find the node N_v, neighbor to N_u, with the largest error;	$O(1)$
Create a new node N_r between N_u and N_v and connect it to both N_u and N_v. **Insert the node into the R-Tree and into the error index heap**;	$O(k \cdot \log\ n)$
Adjust the error of N_u and N_v. **Update the error index heap**;	$O(\log\ n)$ - binary heap or Amortized $O(\log\ n)$ - Fibonacci heap
(8) **Not needed.** See Section 4.4;	

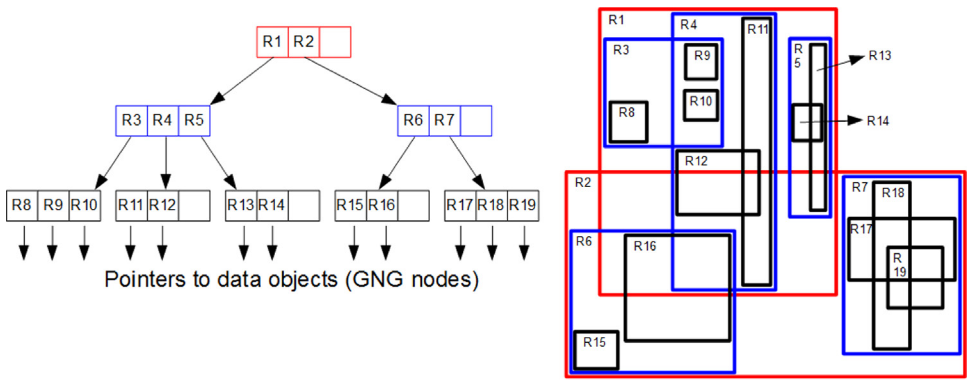


Fig. 2. An R-Tree data structure example. (Adapted from [11].)

scenario of $O(k \cdot \log^2 n)$. This worst scenario occurs when each node in the path between the leaf that stores the point to be removed and the tree root must have its number of children decreased below the minimum value m, which triggers a removal and subsequent reinsertion of that node. We must stress the fact that this problem is not a realistic problem in the GNG algorithm because the number of node removals is very small.

In [11], there is a suggestion that value updates should be performed through its removal and reinsertion; however, for the GNG case, updates of the nodes positions are always small changes and, therefore, changes in the bounding box of the tree node containing the point are always small or even nonexistent. With this assumption, the update process must update only the required bounding boxes in the path between the updated node and the root, an $O(k \cdot \log n)$ operation.

Algorithms for nearest neighbor searches will be discussed in detail in Section 5.

4.3. Step 7: search for the node with biggest error

In Step 7 of the GNG algorithm, the critical operation constitutes searching for the node with the largest error value. To accelerate this operation, a well-known choice is to maintain a priority queue stored in a binary heap [5]. Using a heap, the query for the node with the largest error becomes $O(1)$. On the other hand, the heap must be updated whenever a new node is inserted (Step 7), removed (Step 6) or has its error value updated (Steps 2 and 7). Fortunately, all of these operations can be implemented in $O(\log n)$ if some precautions are taken.

To avoid searches whenever we need to update the error value of a node, it is necessary that the graph stores a reference in each of its nodes to the position of the node in the heap. In practice, an even better approach is to make the graph and the heap share the same data vector.

An alternative to the use of a binary heap is the use of a Fibonacci heap [9]. This enhanced heap structure introduces the possibility of incrementing the error value of a node in amortized $O(1)$ time. Theoretically, for the GNG algorithm, this approach can be a large advantage over the traditional binary heap because this operation is performed at every iteration in Step 2. On the other hand, Fibonacci heaps are much more complex structures than binary heaps. Moreover, constants that are associated with its implementation, hidden behind the big-oh notation, are usually high, which potentially counteracts its theoretical advantage in practical applications. In Section 6, we will present the results that were obtained with both of the heap structures.

4.4. Step 8: decrementing the error value of every node

The 8th step in the GNG algorithm states that the error value of each node in the graph should be decremented by a β factor. Because this action should be performed for every node, this operation will always be $O(n)$ if executed directly. However, we can make a small change in the algorithm, without altering its result, to postpone the decrement operation until the error value is actually used/updated in Steps 2 and 7. To accomplish that change, we must store, in the node, together with the error information, the iteration number in which the error was last updated. With that, if the last complete iteration was iteration I, then the updated error value of any node can be given by the expression:

$$\text{Error} = \text{Stored error on node} \cdot (1-\beta)^{(I - \text{last error update})}.$$

Whenever it becomes necessary to update or retrieve the error information in Step 2 or 7, the above expression can be evaluated along with the new error value stored in the node together with the current iteration number. In practice, this rule must be integrated with the heap algorithm described in Section 4.3 because the correct error value must be calculated whenever the heap needs to update its structure.

4.5. The FGNG algorithm

Fig. 3 illustrates the data structures that are used in Sections 4.2 and 4.3 to optimize the search for the nearest neighbors and for the node with the largest error, respectively. For implementations that use a binary heap, the heap nodes and graph nodes can coalesce. The same arrangement is not true when using a Fibonacci heap.

Based on the proposed data structures, the expected time complexity can be reviewed, as seen in the second column of Table 3. It is important to observe that several operations that were originally executed in $O(1)$ or $O(k)$ are now being executed in $O(\log n)$ or $O(k \cdot \log n)$ because of the need for keeping the used data structures that are to be updated. On the other hand, there are no longer $O(n)$ operations anymore, and Step 8 is no longer executed.

In Step 1, the $O(s(k,n))$ notation represents the complexity of the selected 2NN (2-Nearest Neighbor) algorithm, as will be discussed in Section 5.

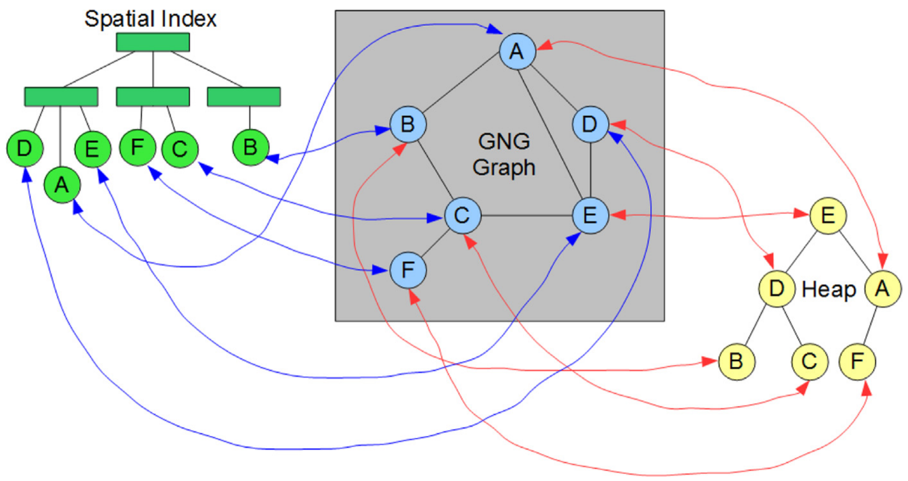


Fig. 3. Data structures and their relations with the generated graph.

A simple analysis of the table can show that Step 1 contributes with a time proportional to $\lambda \cdot n \cdot O(s(k,n))$. The contribution of Steps 2 through 6 is proportional to $\lambda \cdot n \cdot O(k \cdot \log n)$ and Step 7 contributes with $n \cdot O(k \cdot \log n)$. Summing all of the terms, one can see that the complete algorithm performs in an amount of time that is proportional to $O(k \cdot n \cdot \log n) + O(n \cdot s(k,n))$.

We show in Section 5 that there is not a suitable expression to represent the complexity of the proposed 2NN algorithm described above; however, in practice, as shown in Section 5.4, this approach is quite efficient, and the overall performance is much better than $O(k \cdot n)$, albeit worse than $O(k \cdot \log n)$ (see Section 6 for the experimental results).

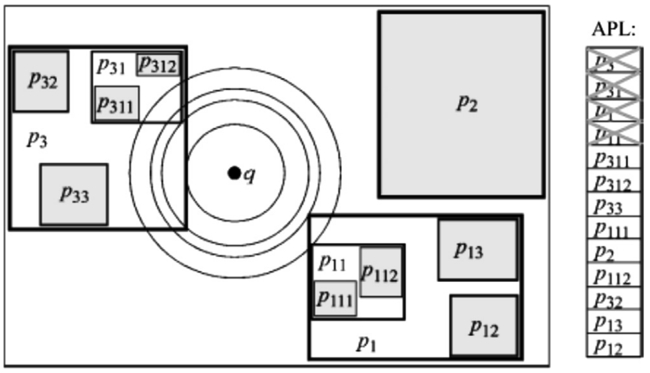


Fig. 4. The HS algorithm. (Adapted from [3].)

5. Nearest Neighbors

The efficiency of the nearest neighbor search is a fundamental part of improving the efficiency of the FGNG algorithm because it is used in every iteration to find the pair of closest nodes to the new sample *S*. Section 6 shows that the improvement in this algorithm can have an impact of more than 90% in the total time taken by the complete GNG implementation.

There are two commonly used algorithms for solving the NN (Nearest Neighbor) problem over R-Trees: the HS and the RKV algorithms. The HS algorithm, developed by Hjaltason and Samet [12], conducts a best-first search, i.e., the search is ordered by the best candidates. In contrast, the RKV algorithm, developed by Roussopoulos et al. [18], is a “branch and bound” algorithm that operates using a depth-first search.

In [3], Böhm et al. show that the RKV algorithm involves an additional memory cost that is proportional to $O(\log n)$, in its worst case, while the HS algorithm involves, also in the worst case, an additional memory cost that is proportional to $O(n)$, where n is the number of nodes in the R-Tree. The same survey also demonstrates that the HS algorithm is optimum regarding the number of “disk pages” that are accessed to find the nearest neighbor, but that finding does not mean that it is necessarily faster because the RKV algorithm takes better profit from page locality (minimizing the disk head movement for trees that keep their data in secondary storage). There are also a series of difficulties when extending the most efficient version of the RKV algorithm from the context of the NN search to the context of a KNN (K-Nearest Neighbors) search.

In [4], Cheung and Fu show that the use of MINMAXDIST, one of the three metrics used by the RKV algorithm, is not necessary for this algorithm. In [1], Adler and Heeringa present a new version of the RKV algorithm, called RKV-PP, that solves all of the difficulties presented in [3] in extending the basic algorithm to KNN and also “redeems” the MINMAXDIST metric in a new context called “promise pruning”, effectively enhancing the algorithm performance.

To test this scenario and also considering that our R-Tree implementation keeps all of its data in the main memory (a different environment from the environment studied in most of the literature), we have performed tests with three different implementations: the first implementation is of the HS algorithm, the second is a modified version of HS that uses the MINMAXDIST metric and the third implementation is the RKV-PP method. It is also important to remember that, for the FGNG algorithm, we do not need a full KNN algorithm; we only need an algorithm that is capable of calculating a 2NN problem.

Next, we present a brief explanation of each of the implemented algorithms. For more details, see [12,3,1]. Afterward, we show a brief experimental performance analysis.

5.1. The HS algorithm

The HS algorithm performs a search for the closest neighbor from a point q; the search is performed in a best-first order. To accomplish that goal, the algorithm maintains a priority queue called APL (active page list). Fig. 4 illustrates its behavior.

To initiate the computation, the algorithm includes the tree root in the queue and sets the value of a variable called *cpc* (closest point candidate) to an infinite value. Next, the process repeats Steps 1–4, which are presented below, until the queue becomes empty or the minimum distance has been found.

1. Remove from the APL the node with the smaller distance to q;
2. If the distance from the removed node to q is larger than the current value of *cpc*, then the nearest neighbor has already been found and the loop can finish;
3. If the node is a leaf, then calculate the distance from q to every point stored in this leaf, updating the value in *cpc* if the calculated distance is smaller then the stored value;
4. If the node is an internal node, then calculate, for each child of this node, the smallest distance between q and the minimum bounding box of the child, inserting the child node into the APL with a priority equal to the calculated distance.

To extend this NN algorithm for the desired 2NN case, it is sufficient to store and update two *cpc* values.

5.2. The modified HS algorithm

The modified version of the HS algorithm attempts to profit from the MIN-MAXDIST metric, which was originally proposed by the RKV algorithm. This metric attempts to explore the fact that, independent of the space dimension that is in use, each hyperplane that delimits the minimum bounding box must contain at least one point because if that was not true, the box could be reduced and therefore could not be a minimum. Given a point q and a minimum bounding box, this metric calculates the largest distance from q to each of the bounding box hyperplanes and selects the smallest of these measures.

Given the value of MINMAXDIST, one can be certain that at least one of the objects inside the bounding box will have a distance to q that is smaller than or equal to MINMAXDIST. Fig. 5 illustrates this relation and gives a formal definition to the metric, where *lbk* and *ubk* are the smaller and larger values of the bounding box in the k direction, respectively, q_k is the point value in the k direction, and d is the space dimension.

This metric can be explored in Step 4 in the HS algorithm to avoid including nodes in the APL when the smallest distance from q to the node is greater than the smallest value of MINMAXDIST found so far.

This optimisation is quite simple for calculating the NN, but for the KNN case greater care must be taken, as presented in [3]. This problem occurs because the metric guarantees that each hyper-plane of the bounding box contains at least one point, but nothing guarantees that it has K points. For the specific case of 2NN a simple solution can be found if care is taken while updating the smallest MINMAXDIST value that has been found so far. One can proceed by storing the values of the two smallest values of MINMAXDIST, but while updating the smallest value, its old value can only be transferred to the second smallest only if the new smallest node is not a children of the original smallest node (otherwise, we would be considering the same case twice for a single sub-tree).

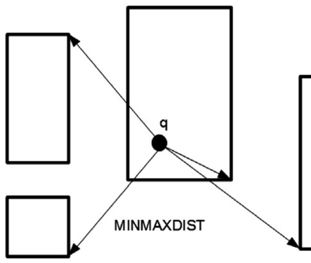
An important point to keep in mind is that calculating the value of MINMAXDIST is an $O(k^2)$ operation: as a result, for higher dimensions, the calculation of this metric can become slow, effectively negating its pruning benefits.

5.3. The RKV-PP algorithm

The RKV-PP algorithm is based on a depth-first search in the tree. For each node, a recursive search of the children is performed in the increasing order of the distance from the child to the search point q. Additionally, the MINDIST and MINMAXDIST metrics are used to prune the tree and eliminate some of the recursive paths. The algorithm operates through the following steps:

1. If the node is a leaf, then compute the distance from q to every point stored in this leaf updating the cpc (closest point candidate) value if the calculated distance is smaller than the stored value;

$$MINMAXDIST^2(q,box) = \min_{0\le k<d}\left(|q_k - rm_k|^2 + \sum_{0\le i<d,\, i\ne k} |q_i - rM_i|^2\right),$$



where

$$rm_k = \begin{cases} lb_k & \text{if } q_k \le \dfrac{lb_k + ub_k}{2} \\ ub_k & \text{otherwise} \end{cases}$$

$$rM_i = \begin{cases} lb_i & \text{if } q_i \ge \dfrac{lb_i + ub_i}{2} \\ ub_i & \text{otherwise} \end{cases}$$

Fig. 5. The MINMAXDIST metric. (Adapted from [3].)

2. If the node is an internal node:
   (a) Create an ABL (active branch list) that includes all the child nodes of the current node. Sort the list by increasing the distance between q and the child bounding box (MINDIST metric).
   (b) For each node in the ABL, if the MINMAXDIST between the node's bounding box and q is smaller than the current cpc value, then update the cpc. This rule is the rule that gives the name to the algorithm (promise pruning) because it immediately updates the cpc value using a distance that will be found in the future;
   (c) For each node in the ABL, if its MINDIST value is smaller than the current cpc, traverse that sub-tree recursively.

To extend this algorithm to the KNN case, we must maintain an ordered list with the smallest K cpc's and change Step 2(c) to verify, before the recursive call, if any promise relative to the sub-tree that will be traversed exists in the list. If so, it should be removed from the list and replaced by an infinite value. For the 2NN case, it is sufficient to have two cpc variables, instead of maintaining a full list. It is interesting to note that this solution is equivalent to the solution that was adopted in the modified HS algorithm. Additionally, the same caveat about MINMAXDIST and higher dimensions that was described for the modified HS algorithm applies.

5.4. 2NN performance analysis

None of the three studied algorithms can guarantee a search order that is better than $O(k \cdot n)$, and the researched literature does not present an estimation for the mean behavior of those algorithms. In practice, as seen below, their behavior is very good, accessing only a small percentage of the total number of nodes in the tree in every search.

To extract some of the experimental data, we have instrumented our R-Tree to count the number of tree nodes that were visited during the search for the two nearest neighbors of a random search point. Experiments were conducted for 2, 4, 6 and 8 dimensions.

Data from Figs. 6–9 were obtained by varying the number of points that were preloaded on the tree and calculating the time and average number of visited nodes per search after conducting 10,000 searches on the same tree. Each data point in those graphs represents an average of 10 such experiments over an R-Tree with optimized M and m values (see Section 6).

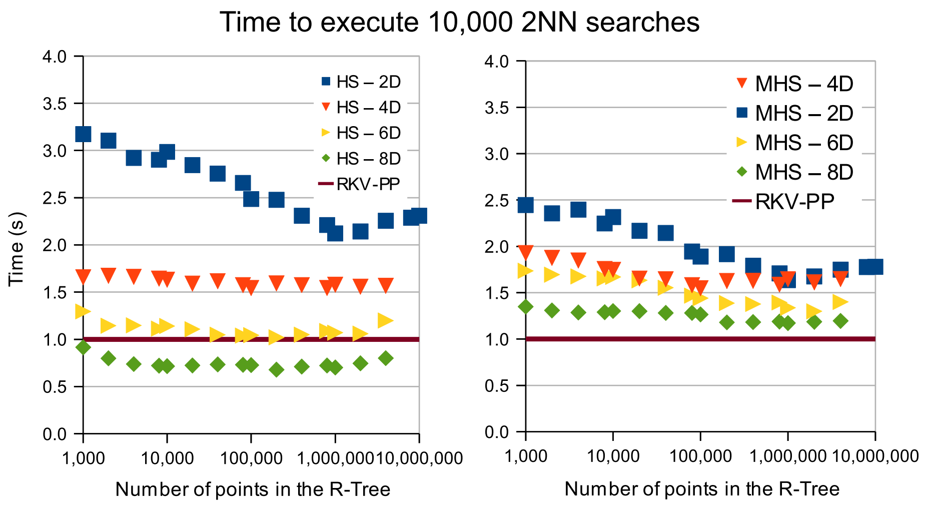
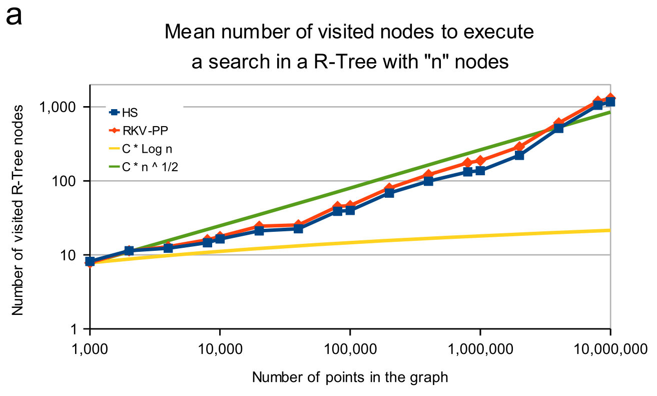


Fig. 6. Search time comparison, normalized by the RKV-PP time. Smaller is better.



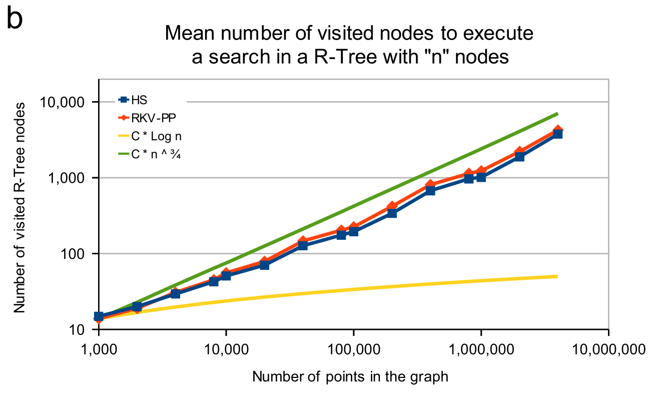
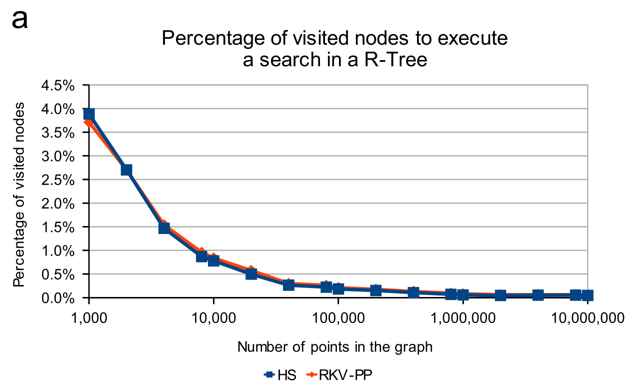


Fig. 7. Average measured number of traversed nodes × theoretical functions. (a) 2 dimensions and (b) 8 dimensions. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)



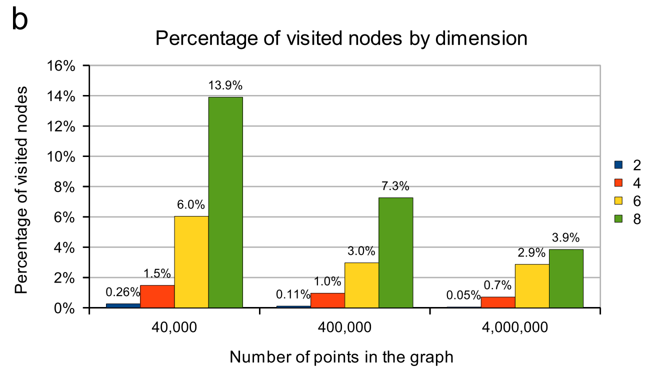
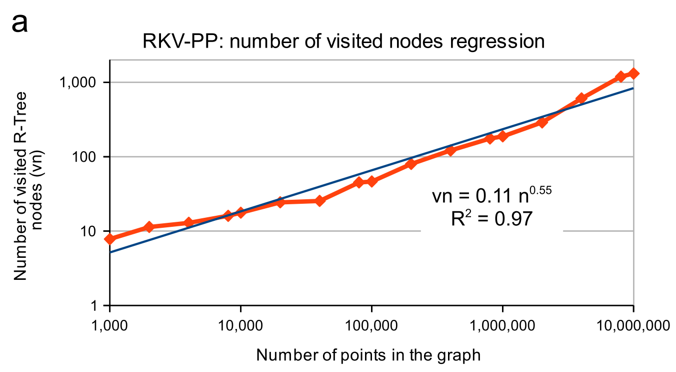


Fig. 8. Percentage of visited nodes while searching. (a) 2 dimensions and (b) visited nodes ratio comparison for 2, 4, 6 and 8 dimensions.



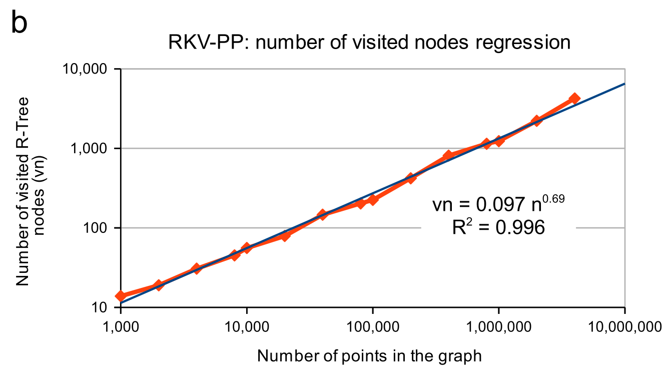


Fig. 9. Regression for sample values measured for the RKV-PP method. (a) 2 dimensions and (b) 8 dimensions.

Fig. 6 compares the time taken by each algorithm to conduct 10,000 random searches over an R-Tree that is preloaded with a set of 1000–10,000,000 random points and dimension varying between 2 and 8. The times are normalized by the time taken by the fastest algorithm, RKV-PP. The left graph of Fig. 6 shows the results for the HS algorithm, while the right graph shows results for the Modified HS algorithm.

It can be observed that for small dimensions, the RKV-PP algorithm is much better compared with the other options. As expected, increasing the problem dimension makes the HS algorithm, which does not use the $O(k^2)$ MINMAXDIST metric, a superior choice. Measured data shows that the turn over point is approximately $k=6$.

Fig. 7a shows the average number of visited nodes to execute a 2NN search in an R-Tree that is preloaded with 1000–10,000,000 points in a 2D scenario. Blue and red series refer to the HS and RKV-PP algorithms. The yellow curve shows the theoretical number of visited nodes if the algorithms were $O(\log n)$, with constants adjusted to make the first point matches the measured result for 1000 points and with n being the number of nodes in the tree (not points). The green curve does the same for an $O(n^{1/2})$ function. Fig. 7b shows the data for the 8D scenario. In this case, the green curve refers to the $O(n^{3/4})$ function.

The absence of the modified HS algorithm from Fig. 7 is deliberate because the number of visited nodes is the same as the number for the HS algorithm. For small dimensions, a modified algorithm is faster because it avoids inserting unnecessary nodes in the priority queue, not because it traverses fewer points. In fact, a closer look at the graph shows that the RKV-PP in fact traverses more points than the HS algorithm, but it is faster because it does not need to pay the price of maintaining the priority queue, which is implemented as a binary heap.

Another important point to observe is that, for a 2D tree loaded with 10,000,000 points, the search visits approximately 1300 nodes in the tree, or 0.05% of its total nodes, as seen in Fig. 8a. The observed curve shape is the same for higher dimensions, but the visited node ratio increases with k, growing to 3.85% for an 8D tree loaded with 4,000,000 points, as seen in Fig. 8b.

Finally, Fig. 9a and b presents regression curves over the measured values for the RKV-PP algorithm for the 2D and 8D scenarios, which show the non-linear behavior for the conducted experiments.

6. Results

6.1. Test suite

Because FGNG achieves the same results as GNG, their classification power is the same, and the FGNG is a suitable replacement for GNG whenever the GNG algorithm is applicable. In this way, this section aims to verify the efficiency of our FGNG proposal. To accomplish that goal, several implementations of the GNG algorithm were made, which ranged from a simple version that was implemented according to the algorithm presented in Section 2 using only sequential searches to the full optimization proposal in Section 4. In this paper, the results from the 3 most important implementation solutions are presented. Their main characteristics are the following:

1. *Simple*: Simple implementation using only sequential searches (see Section 2);
2. *FGNG with a binary heap*: Fully optimized implementation, including the R-Tree, a binary heap and removal of the explicit execution of Step 8 of the implementation in item 1. Searches for the nearest neighbors can be performed with the three different solutions (HS, modified HS, RKV-PP) as described in Section 4. From now on, these three solutions will be denoted by: FGNG–HS; FGNG–MHS and FGNG–RKV-PP.
3. *FGNG with a Fibonacci heap*: This approach replaces the binary heap in the previous implementation with a Fibonacci heap (see Section 4).

All of the performance measures to be presented on this section were run on a Dell Studio XPS8100 machine with an i5 750 @ 2.67 GHz processor, 4 cores, 4GB RAM and a Windows 7 64-bits operating system.

To guarantee fair time comparisons, all of the tests were run while feeding the random number generator with the same seed. For all of the tests, we use the typical parameter values that are presented in Section 2, and we sample the points using a uniform probability distribution function in each coordinate.

Several time measures were used for the implementations described above, with a varying number of graph nodes and dimensions. When appropriate, the KNN algorithm in use was also varied. In the next subsections, the results are presented for a 2-dimensional test case, which is followed by an analysis of the influence of R-Tree parameters in the results and by a k-dimensional test case.

The source code for our FGNG implementation and the code for the test application developed in this section are available as a Supplementary Material and also in http://www.tecgraf.puc-rio.br/fgng.

6.2. The 2-dimensional test case

Table 4 presents measured results for all 3 implemented versions, for graph sizes that vary from 100 to 2,000,000 nodes. For the FGNG implementation, it includes also measures for the 3 KNN algorithms. It is interesting to note that, for a small number of nodes (100 and 200), the optimized implementations are actually slower than the simple implementation. This behavior is expected because there is a cost to managing the required data structures. With over 10,000 nodes, the benefits of the optimization are quite clear, with speedups of almost $60\times$ for 200,000 nodes.

Fig. 10 presents a graphic view for the data in Table 4, together with regression curves for the simple and FGNG-RKV-PP cases. The general behavior follows the expected results, showing the benefits of the proposed optimizations and the clear superior behavior of the RKV-PP algorithm for the 2-dimensional test case, as previewed by the experiments shown in Section 5.4. Numerically, the speedup of the RKV-PP algorithm over the HS algorithm is 97% for a test case with 200,000 nodes.

To further investigate the scalability, the GNG implementation was instrumented to measure the time taken to add each 1000 nodes of the graph while constructing a 100,000-node graph. Fig. 11 shows the measured times on a logarithmic scale, together with two theoretical curves for an $O(n^2)$ and an $O(n \cdot \log n)$ algorithm, respectively, in green and yellow. Both of the curves had their constants adjusted to match the first point of the measured data. From the regression analysis and from that graph, it is clear that the performance of the optimized version is much better than $O(n^2)$ but is worse than $O(n \cdot \log n)$.

Fig. 12 shows the effect of replacing the simple binary heap for the more elaborate Fibonacci heap in the FGNG implementation. From the theory presented in Section 4.3, one can expect that, for large numbers of nodes, the Fibonacci heap might outperform the binary heap. The graph confirms this theory, but the improvement is so small that the proposed acceleration does not appear to pay for the additional implementation complexity.

6.3. Number of children per node of the R-Tree

All of the measures presented previously for the 2D case were performed with an R-Tree with parameters $M=8$ and $m=4$, i.e., the maximum number of children of each node is 8 and the minimum is 4. To complete the 2D analysis, some measures were made to verify whether the efficiency could be improved by increasing the M parameter (and thus decreasing the tree height). For all of these measures, the value of m was kept equal to $M/2$.

Table 4
The measured execution times for the GNG and FGNG algorithms. Values marked with an * are estimates based on growth rates.

Implementation	Time (s) per graph size									
	100	200	1000	2000	10,000	20,000	100,000	200,000	1,000,000	2,000,000
Simple	0.07	0.15	1.85	6.58	149.71	590.77	14,485	59,146	16 days*	2 months*
FGNG–HS	0.13	0.31	2.25	5.28	39.58	93.18	780	1975	15,290	31,071
FGNG–MHS	0.12	0.31	2.18	5.07	36.27	85.36	616	1410	11,025	25,358
FGNG–RKV-PP	0.09	0.20	1.40	3.26	23.39	54.95	418	1004	7596	18,109
FGNG–Fibonacci	0.09	0.21	1.44	3.29	23.45	54.94	416	990	7310	17,120

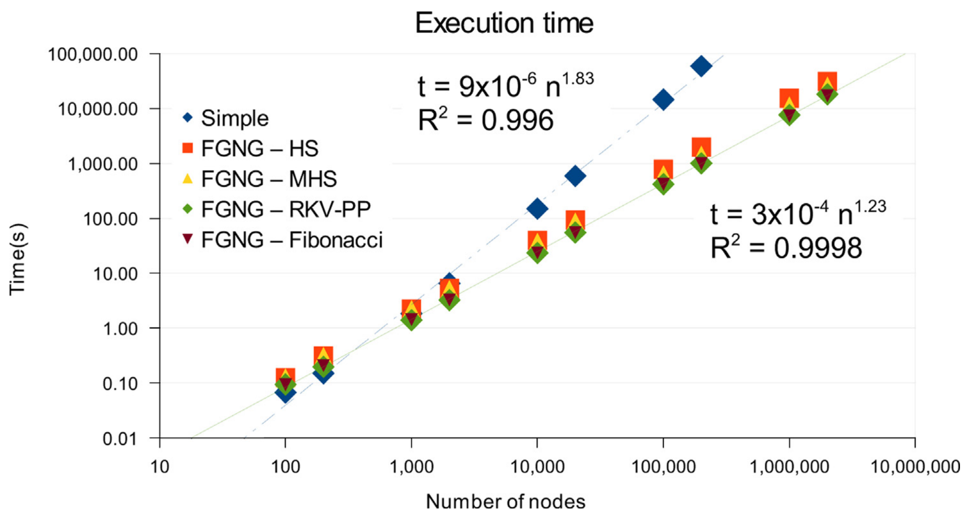


Fig. 10. Comparing the execution times by the algorithm and the number of nodes in the graph.

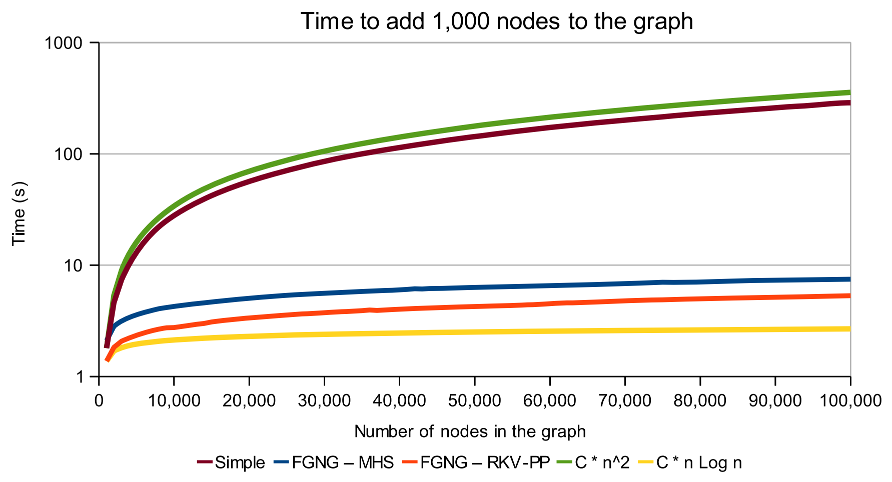


Fig. 11. Time taken to add each 1000 nodes in the graph, from 0 to 100,000 nodes. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

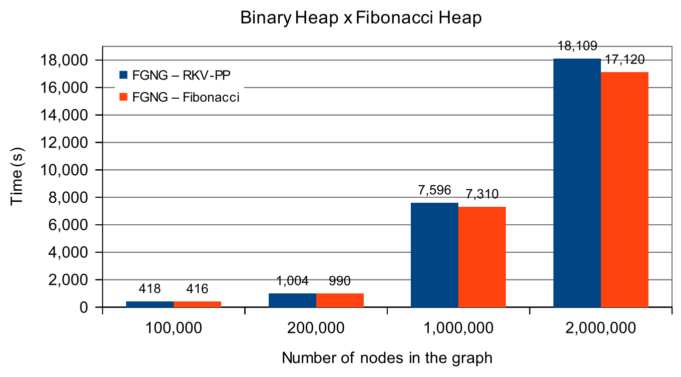


Fig. 12. Effect on the time of using a Fibonacci heap instead of a Binary Heap.

Experimental data shows that, for the modified HS method, the measured time is minimum when $M=20$, with an approximate difference of 10% from the base case with $M=8$. For the RKV-PP method, experiments have shown that the time is minimum for $M=8$. This difference between the methods can be explained by the nature of the RKV-PP algorithm, which includes the sorting of values inside the nodes. When M is increased, the weight of the sorting operation grows; as a result, the optimum balance between the sorting time and the tree height favors a smaller value of M.

For k-dimensional test cases, the effects of changing the values of M and m are much more important than for the 2D case. Imagine a scenario that has 8 dimensions and an R-Tree with $M=8$ and $m=4$. In that setup, each bounding box on the R-Tree has 16 hyperplanes; however, with a maximum of only 8 children per node, there can be hyperplanes that do not contain values (making the bounding box have a 0 volume), which leads to a serious underutilization of the R-Tree. As an example, let us consider a setup with 10,000 nodes, $k=8$, and the use of the RKV-PP method. With $M=8$ and $m=4$, the GNG algorithm is executed in 1010 s. With $M=64$ and $m=32$, the same test case is executed in 409 s, a speedup of nearly 2.5 times, despite the fact that now the sorting operation is executed over a larger data set.

To analyze this effect, some measurements were taken to determine the best values of M and m for 3, 4, 6 and 8 dimensions. Tests were performed with 1000, 2000 and 10,000 nodes and the RKV-PP method. It was also verified that, for higher dimensions, the optimum point is the same for the RKV-PP and for the HS algorithms. Table 5 summarizes the results.

6.4. The k-dimensional test case

Table 6 presents measured results for the simple FGNG-RKV-PP and FGNG-HS algorithms for graph sizes that vary from 100,000 to 200,000 nodes while k varies from 3 to 8 dimensions. All of the measures were performed with the optimum values of M and m analyzed in Section 6.3. Smaller graph sizes were not included because, for higher dimensions, a larger number of nodes are necessary for the optimizations to take effect. The same data can be seen in Fig. 13.

Table 7 is based on the same data presented in Table 6, showing the speedup of the FGNG implementations with respect to the simple implementation. With 200,000 nodes, it can be seen that the speedup that is approximately 60 times for the 2D case drops to 2.2 times for the 8D case. This effect is predictable by the behavior of the NN algorithm analyzed in Section 5.4.

Another important observation, which confirms the analysis in Section 5.4, is the comparison between the RKV-PP and the HS NN methods. For smaller dimensions, the former is better, but for higher dimensions, the latter is superior. Experimentally, the turning point is approximately 6 dimensions.

To measure the growth rate of the k-dimensional case, similar to the actions taken for the previous 2D tests, the implementation was instrumented to measure the time that was required to add each 1000 nodes of the graph while constructing a 200,000-node graph. Fig. 14 shows the results for 6 and 8 dimensions.

Analyzing the graph, we can see that the growth rate of the FGNG implementation is much smaller than for the simple implementation, which means that, for a higher number of nodes, we can expect greater speedups than those recorded in Table 7.

Another interesting point to note is the sudden slope change for the simple implementations of approximately 100,000 nodes. One possible explanation for that effect lies in the heavy dependence of today's hardware on cache hits. After some memory usage threshold, the whole GNG node vector does not fit entirely in the cache anymore. Because sequential NN traverses the entire node set, it is more susceptible to cache effects than the R-Tree, which accesses only a fraction of the total number of nodes. Further investigation shows that the slope change occurs when the size of the vector that contains the GNG nodes reaches approximately 7.4 Mb, while the test machine has an 8 Mb cache.

Table 5
Optimum values for M and m R-tree parameters.

Number of nodes	Optimum M/m per dimension			
	3	4	6	8
1000	14/7	32/8	36/18	36/14
2000	14/7	32/8	36/18	52/14
10,000	14/7	32/8	36/18	64/32

Table 6
The measured execution times for the GNG and the FGNG algorithms considering different dimensions.

Implementation	Time (s) per dimension			
	3	4	6	8
	100,000 nodes			
Simple	16,750	18,717	22,543	25,044
FGNG–RKV-PP	1390	3464	12,924	22,734
FGNG–HS	1990	3972	11,693	16,764
	200,000 nodes			
Simple	71,502	84,962	111,255	129,118
FGNG–RKV-PP	3624	9551	43,279	82,209
FGNG–HS	5252	10,968	38,962	58,857

7. Conclusion

The results discussed in Section 6 show that the proposed FGNG implementation is efficient and enhances the time response for the GNG algorithm when the generated graph has a large number of nodes in a k-dimensional space.

Table 7
Acceleration regarding the simple implementation for different dimensions.

Implementation	Speedup per dimension 200,000 nodes			
	3	4	6	8
Simple	1.0	1.0	1.0	1.0
FGNG–RKV-PP	19.7	8.9	2.6	1.6
FGNG–HS	13.6	7.7	2.9	2.2

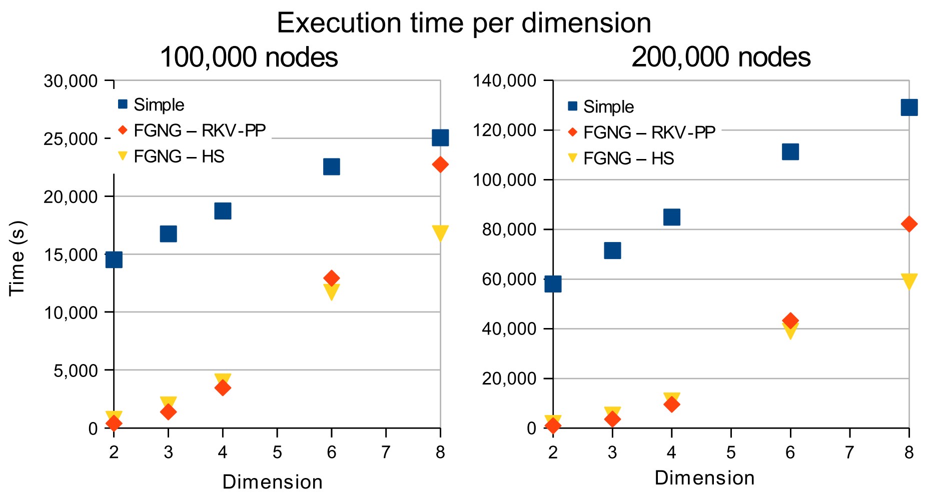


Fig. 13. Comparing the execution times by the algorithm and the number of dimensions.

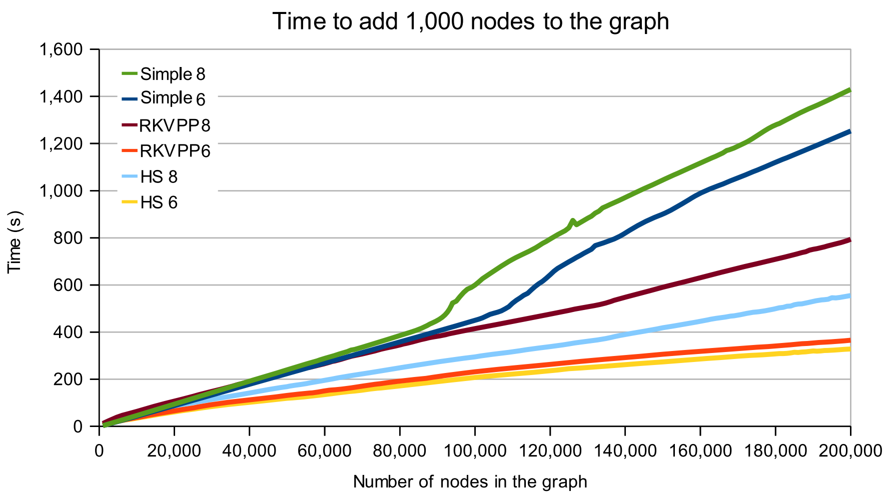


Fig. 14. Time taken to add each 1000 nodes in the graph, from 0 to 200,000 nodes.

Table 8
Time comparison with and without optimization.

Number of nodes	GNG	FGNG
10,000	2.5 min	23 s
20,000	10 min	55 s
100,000	4 h	7 min
200,000	16.4 h	17 min
1,000,000	16 days[a]	2.1 h
2,000,000	2 months[a]	5 h

[a] Estimated values.

Table 8 summarizes the achieved speedup for the 2-dimensional test case. Times for the non-optimized column refer to the simple implementation. For sizes that are larger than 200,000 nodes, the times shown are estimations that are based on the growth factor of this implementation. The times for the optimized column refer to the optimized implementation together with the RKV-PP algorithm.

The acceleration decreases with the space dimension but remains an interesting optimization strategy because the FGNG's growth rates for the k-dimensional case are much slower when compared to the simple GNG implementation. Further tests should be conducted with pyramid trees [2] to evaluate their effects with higher dimensional data.

Another important point to stress is that the memory consumption that is associated with the R-Tree does not explode as happens with grid structures [8]. For an R-Tree with a minimum number of children per node, m, and that stores n data points, the maximum number of nodes in the tree (mnn) can be given by the following expression:

$$mnn \le 1 + n \cdot \sum_{i=1}^{h} m^{-i}, \quad \text{where } h = \lfloor \log_m n \rfloor$$

and which can also be written as

$$mnn \le 1 + n \cdot \frac{1 - m^{-h}}{m - 1}.$$

Finally, the analysis conducted in Section 6 shows that the proposed algorithm has a growth factor that lies between the original $O(k \cdot n^2)$ and $O(k \cdot n \cdot \log\ n)$.

For even better speedups of the GNG implementation, the authors plan to investigate possibilities to parallelize the FGNG algorithm without changing its semantics.

Acknowledgements

The authors would like to thank the reviewers for their valuable comments and suggestions. Also, they would like to thank CNPq, FAPERJ and FINEP for the support of this research.

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version at http://dx.doi.org/10.1016/j.neucom.2013.08.033.

References

[1] M. Adler, B. Heeringa, Search Space Reduction in r-Trees, 2001.
[2] S. Berchtold, C. Böhm, H.-P. Kriegal, The pyramid-technique: towards breaking the curse of dimensionality, SIGMOD Rec. 27 (June (2)) (1998) 142–153.
[3] C. Böhm, S. Berchtold, D.A. Keim, Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases, ACM Comput. Surv. 33 (September (3)) (2001) 322–373.
[4] K.L. Cheung, A.W.-C. Fu, Enhanced nearest neighbour search on the r-tree, SIGMOD Rec. 27 (September (3)) (1998) 16–21.
[5] T.H. Cormen, C. Stein, R.L. Rivest, C.E. Leiserson, Introduction to Algorithms, 2nd edition, McGraw-Hill Higher Education, 2001.
[6] E. Cuadros-Vargas, R. Romero, A sam-som family: incorporating spatial access methods into constructive self-organizing maps, in: Proceedings of the International Joint Conference on Neural Networks 2002. IJCNN '02, vol. 2, 2002, pp. 1172–1177.
[7] M. de Carvalho Machado, Determinaç ao de malhas de falhas em dados sísmicos por aprendizado competitivo (Ph.D. thesis), Pontifical Catholic University of Rio de Janeiro, 2008.
[8] D. Fišer, J. Faigl, M. Kulich, Growing neural gas efficiently, Neurocomputing 104 (2013) 72–82, URL 〈http://www.sciencedirect.com/science/article/pii/S0925231212008351〉.
[9] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (July (3)) (1987) 596–615.
[10] B. Fritzke, A growing neural gas network learns topologies, in: Advances in Neural Information Processing Systems 7, MIT Press, 1995, pp. 625–632.
[11] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84, ACM, New York, NY, USA, 1984, pp. 47–57.
[12] G.R. Hjaltason, H. Samet, Ranking in spatial databases, in: Proceedings of the 4th International Symposium on Advances in Spatial Databases, SSD '95, Springer-Verlag, London, UK, UK, 1995, pp. 83–95.
[13] Y. Holdstein, A. Fischer, Three-dimensional surface reconstruction using meshing growing neural gas (MGNG), Vis. Comput. 24 (4) (2008) 295–302.
[14] S. Mitsyn, G. Ososkov, The growing neural gas and clustering of large amounts of data, Opt. Mem. Neural Netw. 20 (4) (2011) 260–270.
[15] A. Ocsa, C. Bedregal, E. Cuadros-Vargas, DB-GNG: a constructive self-organizing map based on densilty, in: Proceedings of the International Joint Conference on Neural Networks 2007, IJCNN '07, August 2007, pp. 1953–1958.
[16] Y. Prudent, A. Ennaji, An incremental growing neural gas learns topologies, in: Proceedings of the International Joint Conference on Neural Networks 2005, IJCNN '05, vol. 2, July–4 August 2005, 2005, pp. 1211–1216.

[17] C. Rehtanz, C. Leder, Stability assessment of electric power systems using growing neural gas and self-organizing maps, in: European Symposium on Artificial Neural Networks Bruges (Belgium), 2000, pp. 401–406.

[18] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD '95, ACM, New York, NY, USA, 1995, pp. 71–79.

Carlos Mendes is a D.Sc. student at PUC-Rio. He has a Masters degree in Informatics (1999) and a Bachelor's degree in Computer Engineering (1996), both from PUC-Rio. Since 1997, he has been a partner at K2 Sistemas and works on coordinating the development of engineering software for the oil industry and on GIS solutions.

Marcelo Gattass, Ph.D. Cornell 1982, has been a Full Professor at PUC-Rio since 1993 and has been the Director of Tecgraf since 1986. He has supervised over 50 M.Sc. dissertations and over 20 Ph.D. theses. He received many awards, including the Formation of Human Resources award of the SPE/IBP/ONIP (2010), member of the National Academy of Engineers (2009), Great Cross of the National Order of Ministry of Science and Technology (2007) and Comendador of the National Order of Scientific Merit (2000). He is a CNPq 1A researcher, and Tecgraf has developed more than 50 innovative technical software systems, which are widely used in the Oil Industry.

Hélio Lopes is an Associate Professor in the Department of Informatics at PUC-Rio. He has a Doctoral degree in Mathematics (1996), a Master's degree in Informatics (1992) and a Bachelor degree in Computer Engineering (1990). All of these degrees were obtained at PUC-Rio. His research is mainly in the fields of Computer Graphics and Computational Modeling. He likes to develop and to apply advanced mathematical and computational techniques to solve real-world problems that have engineering, scientific and industrial relevance.

View publication stats