A Method of Multi-Level Uniform Grids for Spatial Searching

V.V. Davydov

Silvaco Inc., vitality.davydov@silvaco.com

Annotation — This paper introduces a method for spatial indexing of a fixed set of geometrical data. The method is based on spatial decomposition using multi-level uniform grids. The presented implementation of the method demonstrates its applicability to EDA problems. It demonstrates low memory usage and fast query performance for not very clustered data. The method has been successfully implemented and applied for power networks connectivity extraction for analog and mixed designs. The introduced method implementation has been compared with BOOST R-tree in terms of performance and memory consumption.

Keywords — grid method, multi-level uniform grid, spatial data indexing.

I. INTRODUCTION

Geometry search algorithms are of great practical importance in the problems being solved by electronic design automation tools (topological design, physical verification, data visualization). The technology development leads to a substantial increase of the element pack density. As a result, the requirements for geometry search (indexing) algorithms increase in terms of performance and memory consumption.

There are many approaches which can be divided into two basic types: approaches based on clustering (R-trees, adapted B-trees) and approaches based on spatial decomposition (grids, quadrees, BSP-trees) [1]. In practice, the uniform grid method is often used for geometry search because of its simplicity and good performance [2], [3]. The uniform grid method consists in decomposing the whole region containing geometries into cells with a regular mesh. Each cell of the grid contains the list of references to geometries intersecting with this cell. In practice, the uniform grid method often demonstrates better performance in comparison with alternative approaches based on trees of different types. This is because the most designs often have a structure that is close to regular, and geometries (metal wires or cell bounding boxes, for example) are not highly clustered.

One of the drawbacks of the uniform grid method is the substantial query performance degradation in case of highly clustered data. Another drawback is that the choice of grid cell size significantly affects the performance of search operation and the amount of memory used. The significant part of geometries can intersect more than one cell on a grid with small size cells. In some cases, it is required to remove duplicates in search operations. If cell sizes are too big the query performance will degrade. There are several adaptive grid methods in which cell sizes are calculated based on incoming data (for example, [4]). De-

spite it, they do not solve the problem of long and narrow geometries that are often found in real designs.

The method proposed in the article solves the problems of selecting grid cell sizes and eliminating duplications in the result in the case of a fixed set of geometries, i.e. a set that doesn't imply additions, change or deletion of existing geometries in the search structure. The method allows to apply fast parallel algorithms to build a search structure. Two-dimensional rectangles with integer coordinates (32 bits) are considered as geometries. In the case of complex geometries, such as polygons (including polygons with holes), their bounding boxes can be used instead. The method allows generalization to multidimensional space but there are several optimization solutions for two-dimensional space that allow to achieve a good result in terms of query performance and memory consumption.

The main idea of the method is to classify all geometries by size groups and create a separate spatial uniform grid for each size group. The sizes of grid cells are chosen to exceed the sizes of any geometry in the size group. There are only a few articles found related to the grid method. Sometimes, the grid method is combined with hierarchical structures ([5], [6]). The proposed method differs from approaches using hierarchical structures in that the set of spatial uniform grids doesn't form a hierarchy.

II. METHOD

Consider the case of two-dimensional geometries on the plane. Define search area as the minimal rectangular area containing all bounding boxes of all the geometries. Define query window as the rectangular area where a search operation is performed. Classify all the geometries by their sizes, group the geometries with common size properties into size groups. For that purpose, divide all the geometry widths and heights using a logarithmic grid (size grid). For each geometry assign the coordinates on the size grid using the following formulas:

$$= \text{FLOOR}(\log_K W)$$

$$h = \text{FLOOR}(\log_K H),$$

where W , H are width and height of a geometry (its bounding box), w_g , h_g are the coordinates of size grid cell (size group), K is heuristically given log base, FLOOR is the rounding down operator (to an integer value). Each cell of the size grid defines the subset of geometries which widths and heights are in the range. Let's call such subset of geometries belonging to the same size grid cell as size group.

The base of the logarithm is selected considering the need to reduce the number of cells in the size grid (size groups). In case of 32-bit integer coordinates and log base $K = 8$ the number of size groups will be 121 (size grid with 11×11 cells).

For each size group a regular spatial grid is created. The bounding boxes of spatial grids are limited by the search area. Spatial grid cell sizes are defined as the maximal widths and heights of geometries of the size group it is related to. The following formulas show the maximum sizes of geometries for size group defined by w_g, h_g :

$$W_{cell} = K^{w_g+1}$$

$$H_{cell} = K^{h_g+1}.$$

Thus, each geometry can be associated with one and the only one cell in the corresponding spatial grid using an anchor point. The search algorithms are based on the iteration over all spatial grid cells intersecting the query window independently for each size group.

For practical application of the method it is necessary to reduce the number of size groups. First, define the minimum and maximum allowable number of columns and rows of spatial uniform grids. This can be done considering the properties of the geometries or by defining the maximum number of rows and columns in advance (for example, 5000×5000). It is analogous to selecting a lower-left cell on the size grid. The minimal numbers of columns and rows are 1 in case of 1×1 spatial grid.

Introduce the following order of traversing size grid cells (Fig. 1):

1. Start the traversing from lower left cell of the size grid with coordinates (w_g, h_g) .
2. Fix h_g and traverse all the cells in the ascending order of w_g .
3. Fix w_g for the same lower left cell and traverse all the cells in the ascending order of h_g .
4. Continue with the next cell at diagonal direction incrementing both coordinates.
5. Repeat steps 1-4 until all the cells of the size grid will be traversed.

h_g (height) ↕	7	12	15	16
	6	11	13	14
	5	8	9	10
	1	2	3	4
	w_g (width) →			

Fig. 1. The traversing order of size grid cells

The defined traversing order of size grid cells allows to choose only those cells (size groups) for which a certain heuristic criterion is met, for example, the fill factor of the corresponding spatial grid. If this criterion is not met for a size group and related spatial grid, then this size group is

ignored but the next size group is iteratively chosen in the defined traversing order until the criterion is met.

Assume that all the geometries are stored in the form of a flat array. Assign a size group on each geometry. Sort all the geometries in the array by size groups (by coordinates of size grid cells). Define start and end offsets of each size group by iterating over all the array. Considering the size group boundaries in the array, one can apply a heuristic criterion and decide whether to keep or ignore the size group. If the current size group is ignored, then its geometries will be assigned to the next size group. A criterion example – the number of geometries in the size group is greater than the number of cells in the corresponding spatial grid.

There is an alternative approach to fix the subset of size groups in advance, classify all the geometries and then reduce them. This approach will be used in the proposed implementation.

III. IMPLEMENTATION

This chapter introduces the method implementation in C++ language for a fixed set of two-dimensional geometries. To store geometric data C++ STL *vector* container or another array-like container with random access to elements can be used. Every container element can contain either indexable geometric data or references to them. Container elements should have a small size in memory and a fast copy or move constructor. In the opposite case the preparation of the search structure will take longer time because of container elements sorting. The decision to store references or geometrical data is determined by the nature of the problem being solved. Storing geometric data in the container decreases the amount of consumed memory and can serve both purposes: storing and indexing the data.

A. Pre-processing

The maximum subset of size groups is fixed in our implementation. For each size group a spatial uniform grid is created. The spatial grid slices the search area to several rows and columns. Used spatial grids (grayed) are shown in the table 1.

Table 1

The used subset of spatial grids

		Row Number					
		8192	4096	512	64	8	1
Column Number	8192						
	4096						
	512						
	64						
	8						
	1						

The table shows the initial numbers of rows and columns. These numbers will be corrected to make widths and heights of grid cells rounded to the powers of 2. It allows to replace division operations by much faster bit shift operations when calculating grid coordinates. The size of the search area containing all the geometries is calculated either by traversing all the geometries or by using parallel algorithms like “reduce” from Intel Threading Blocks library.

Each cell of a spatial grid is identified by grid coordinates X_g и Y_g which define the cell position on the grid:

$$X_g = FLOOR \frac{(X - X_0)}{W_{cell}}$$

$$Y_g = FLOOR \frac{(Y - Y_0)}{H_{cell}}$$

where, W_{cell} , H_{cell} are width and height of grid cell accordingly; X_0 , Y_0 are minimal coordinates of the search area. Grid coordinates are rounded down to the nearest integer. If the calculated grid coordinate is greater than the maximal grid coordinate, then the maximal coordinate is used. Each cell can be also identified by the cell index using the following formula:

$$I_{cell} = X_g + Y_g * N_x$$

where I_{cell} is the cell index (number), N_x is the number of columns. The division operations in the formulas above are replaced by bit shift operations because W_{cell} , H_{cell} were rounded to the powers of 2.

The next step is to split the whole vector with geometric data into partitions. Each partition corresponds to a size group. The vector partitioning is executed by the defined size group traversing order. Apply STL *partition* algorithm iteratively for each size group. The partition algorithm moves the elements inside the vector (without memory reallocation) with the purpose to split it into two parts – the first part contains the elements which satisfy some criteria. Another part contains the rest elements. If the number of elements in size group doesn’t satisfy some criteria, then the size group is ignored, and the partitioning is performed with the next size group in the defined order. One of the possible criteria – the number of elements in the group is greater than the number of cells in the corresponding spatial grid. Start and end positions of size group element range are stored in some auxiliary structures for fast access.

After partitioning each vector partition is sorted by spatial grid coordinates (cell indexes). The elements (geometries) with the same grid coordinates are sorted by its cartesian coordinates X or Y depending on their width/height ratio. The lower left coordinate can be used for that. Thus, all elements of each partition are ordered by spatial grid rows and columns. The important feature of such order is that the vector elements belonging to the same grid rows are placed next to each other. The sorting takes the signifi-

cant part of the pre-processing time. Several parallel algorithms can be applied to sort the vector elements. They can sufficiently decrease the sorting time on modern processors with multiple cores. One of such algorithms is the parallel sort from Intel Threading Building Blocks library.

The final pre-processing step is the construction of the array with offsets of vector element ranges of each cell for each spatial grid to speed up the search of cell elements by grid coordinates. The array with offsets can consume the significant part of used memory depending on the number of cells in spatial grids. It can consume about ~200M bytes of memory for 5000x5000 grid considering that the offset is represented by 8-bytes type (size_t) in case of 64-bit systems. The storing of cell offsets allows to sufficiently speed up the query operation, but it is not necessary. The binary search may be used as an alternative approach. We can store offsets for each grid row but use the binary search to find column offsets of each row.

Finally, all the vector elements are partitioned. Each partition belongs to the corresponding size group. The elements of each partition are sorted by spatial grid rows and columns as shown in Fig. 2. The elements that belong to the same spatial grid cell are sorted by cartesian coordinates.

Fig. 2. The order of spatial grid cells placement in the array with geometries

All the memory consumed is used to store geometry data or references to it as well as the offsets of spatial grids cells. The memory used to store the cell offsets can consume significant part of the used memory. The memory consumed for storing auxiliary data is negligible.

B. Searching

The search operation is executed for each size group (spatial grid) independently. The query results are united. The following steps are performed for each spatial grid:

- 1) Determine row and column ranges in grid coordinates by minimal and maximal cartesian coordinates of the query window. If lower-left point is used as the anchor point for each geometry, then grid coordinates of lower left point of the query window should be decremented by 1. This is because some geometries assigned to the adjacent cells located to the left and below the query window can still intersect the query window.
- 2) The search is performed for each grid row in the determined range. For each grid row, determine start and end positions of array elements (geometries) of row cells that are in the defined range. Use the array with cell offsets to calculate the positions of first and last row cells in the range. Iterate over all the geometries from start to end position and check it for intersection with the query window.

The performance of the search operation depends on some number of factors: the nature of the size groups and related spatial grids, spatial grid fill factors, geometry of the query window. The search in spatial grids with the biggest number of rows and columns has a major impact on the overall search time. This is because the number of cells to be traversed is much greater than the number of cells of grids with smaller number of rows and columns.

The search time in a spatial grid can be logically divided into two components: time of iteration over grid rows and time of iteration over row cells that are in the query window. Since all the row elements (geometries) are located sequentially in the memory we can simply iterate over all geometries of the row and do not calculate element ranges for each cell. The boundaries of the row element range are calculated in constant time if the array with offsets is used or for the binary search time if the binary search is used instead.

The first time component depends on the geometry of the query window and does not depend on the presence of the geometries in cells. The second time component depends on the presence of the geometries in the grid row. If the grid row has no geometries, then the time will not be spent for iteration over the geometries. If the second time component depends on the number of geometries in the result and can't be optimized, then the first time component is undesirable, since even in the absence of the geometries, some time will be spent on the iteration over all rows in the query window.

Introduce the following quantities: N_{gx} , N_{gy} – the number of cells intersecting with the query window, N_{cell} – the average number of geometries in a cell. Then the search operation complexity for a spatial grid can be described by the formula $O(N_{gx}N_{gy}N_{cell})$ if to use the array with cell offsets.

IV. TESTING

To test the applicability of the method in practice, data from a real design were used. Testing was performed on a computer with 512GB of RAM and 2 Intel Xeon E5-2699 v3 processors (18 cores, 36 worker threads), 72 working threads in total. The BOOST v. 1.64.0 library R-tree implementation was chosen as an alternative for comparison ([7], [8]). Bounding boxes of design components were used as test rectangles in amount about 1 billion rectangles.

One of the design features is the absence of components in a large area in the central part of the whole layout (~25% of the total area). It allows to test the case when a significant part of the spatial grid cells does not contain geometries.

The test consisted in filling the search structure with rectangles and then performing search operations in each rectangle from the test array. By author's opinion such test can emulate connectivity extraction to a large extent. Table 2 shows the structure of spatial grids for the test data. It can be seen that most rectangles were added to the spatial grid with smallest cells (5500x7063 cells).

The test results are shown in the table 3. To fill R-tree a packing algorithm was used ([9]) which is much faster alternative to the regular tree insertion operation. Using the packing algorithm involves the preliminary preparation of data in a specific format which may require additional memory and time.

Table 2

The structure of spatial grids

№	NX	NY	Total number of geometries	Average number of geometries per cell	Maximal number of geometries in a cell
1	5500	7063	1091292530	28.1	369
2	5500	442	3221776	1.3	187
3	344	7063	21865478	9.0	370
4	2750	442	5485955	4.5	54
5	2750	56	448955	2.9	21
6	2750	7	158760	8.2	26
7	43	3532	181687	1.2	114
8	344	56	126727	6.6	330
9	344	7	9393	3.9	15
10	43	442	22913	1.2	54
11	43	56	6147	2.6	58
12	43	7	584	1.9	11
13	43	1	67	1.6	28
14	6	56	444	1.3	47
15	1	56	134	2.4	53
16	6	7	74	1.8	9
17	6	1	15	2.5	6
18	1	7	50	7.1	19
19	1	1	28	28.0	28

Table 3

The test results comparison table

Step	Memory (GB)	Time (sec)
The grid method		
Filling in	17.5	19
Preprocessing (1 thread)	0.4	327
Preprocessing (~72 threads)	0.4	30
Searching	-	2300
R-tree (linear split, 16 nodes)		
Filling in	31.5	273
Searching	-	6981
R-tree (quadratic split, 16 nodes)		
Filling in	31.5	272
Searching	-	6873
R*-tree (16 nodes)		
Filling in	31.5	268
Searching	-	6895

The preparation of the search structure for performing queries consists of two stages: filling in and preprocessing. The filling consists of memory allocation and addition of elements into the array. The preprocessing stage consists in the preparation (sorting) of the array with data for performing queries. For comparison purposes 3 types of R-trees from BOOST library were used. The testing was performed with different values of R-tree parameter defining the maximum number of child nodes in a node: 4, 16, 64 child nodes. The case with 16 child nodes was chosen for final comparison as the best case. The testing was performed several times, the memory consumed, and perfor-

mance were averaged. The stages such as preprocessing and filling in are additionally shown for the introduced method. The preprocessing stage is introduced in both variants: single-threaded and multi-threaded. For R-tree, only the filling step is indicated since there is no separate preprocessing step.

The query time in case of the grid method is 3 times less compared with R-tree. The gain in the use of memory is about 2 times. The time spent to fill in R-tree is slightly less than the time spent on preprocessing for the grid method in single-threaded mode. In case of multi-threaded preprocessing the time spent is about 30 seconds instead of 327 seconds (~72 threads).

V. CONCLUSION

The proposed method of multi-level uniform grids demonstrates better search performance with significantly lesser memory consumption. The method was implemented and successfully applied in practice on designs with a huge number of geometries. Despite the fundamental drawback of the grid method – significant degradation of query performance with high data clustering – the method often shows better performance and memory consumption on real designs if to compare with variety of R-tree implementations and other search structures. The introduced implementation of the grid method is applicable to a fixed set of geometries which allows to use some software development techniques to improve the query performance. The method can be generalized to non-fixed and multidimensional data.

ACKNOWLEDGEMENT

The author is grateful to the developers of Silvaco K.K. Malinauskas and A.F. Samoilov for consultations on

mathematical and engineering topics, as well as Silvaco for the opportunity to prepare the article.

REFERENCES

- [1] Zolotov V.A., Semenov V.A. *Sovremenie metodi poiska i indeksirovaniya mnogomernykh danniy v prilozheniyah modelirovaniya bolshih dinamicheskikh tsen.* // Trudi Instituta sistemnogo programmirovaniya RAN. 2013. Vol 24. C. 381-416 (in Russian).
- [2] Bentley J.L., Friedman J. H. Data structures for range searching // *ACM Comput. Surv.* 1979. 11(4). P. 397–409.
- [3] Lazslo M. *Vychislitel'naya geometriya i komputernaya grafika na C++.* — M.: BINOM, 1997 (in Russian).
- [4] Akman V., Franklin W.R., Kankanhalli M., Narayanaswami C. Geometric computing and uniform grid technique // *Comput. Aided Des.* 21, 7 (August 1989), P. 410-420
- [5] Cazals F., Puech C. Bucket-like space partitioning data structures with applications to ray-tracing // *In Proceedings of the thirteenth annual symposium on Computational geometry (SCG '97).* ACM, New York, NY, USA, P. 11-20.
- [6] Nievergelt J., Hinterberger H., Sevcik K. C. The grid file: An adaptable, symmetric multikey file structure // *ACM Trans. Database Syst.* 1984. 9(1). P. 38–71.
- [7] Guttman A. R-trees: a dynamic index structure for spatial searching // *SIGMOD Rec.* June 1984. V. 14. Issue 2. P. 47-57.
- [8] Beckmann N., Kriegel H.P., Schneider R., Seeger B. The R*-tree: an Efficient and robust Access Method for Points and Rectangles // *Proc. ACM SIGMOD.* May 1990. P. 323-331.
- [9] Leutenegger S.T., Lopez M.A., Edgington J. STR: a simple and efficient algorithm for R-tree packing // *Proceedings 13th International Conference on Data Engineering.* 7-11 April. 1997.