DATABASE STORAGE AND TRANSPARENT MEMORY LOADING OF BIG SPATIAL DATASETS IMPLEMENTED WITH THE DUAL HALF-EDGE DATA STRUCTURE

Pawel Boguslawski[a,*], Patryk Balak[a], Chris Gold[b]

[a] Institute of Geodesy and Geoinformatics, Wroclaw University of Environmental and Life Sciences, Wroclaw, Poland
[b] Faculty of Computing, Engineering and Science, University of South Wales, Pontypridd, United Kingdom
(pawel.boguslawski, patryk.balak)@upwr.edu.pl, chris.gold@gmail.com

Commission IV, WG IV/I

KEY WORDS: data structures, 3D modelling, DBMS, big data, dual half-edge, city model, BIM

ABSTRACT:

3D spatial models covering big areas, such as cities, are widely developed in recent years. Loading of a whole model from a hard drive into a computer memory is often not possible due to big amount of data and memory size limitations. Optimisation techniques based on spatial indexing, such as tiling, are applied in order to load at least a part of a model as soon as possible, while the remaining parts are collected in the background. It is especially useful in visualisation of cities. A similar idea is proposed for a transparent loading of a model implemented with the dual half-edge (DHE) data structure and stored in a database. The existing DHE-based solutions require the whole model to be present in the memory, which is a considerable limitation in case of models covering big areas and including detailed representations of city objects, such as buildings and their interiors. The prototype mechanism developed in this work includes loading and unloading of model parts at a level of single edges as well as model tiling. This allows for spatial analysis without complete loading a big amount of data into memory.

1. INTRODUCTION

Spatial models of real objects are currently more often represented in three dimensions (Buyukdemircioglu and Kocaman, 2020). Modern computers provide computational power and big storage capacity, which is necessary for such a representation. Comparing to traditional 2D or 2.5D models, 3D approach offers direct relations between the real world and a model resulting in accurate visualisation as well as better spatial analysis results. It however requires a big amount of resources, e.g. computer memory. Computing centres may be able to deal with such big models, but it is always justified to use any methods to reduce the load and reduce the computational requirements.

Big spatial models are stored in files, e.g. a BIM model of a building stored using the IFC format (ISO 16739:2018, 2018), or databases, e.g. a city model stored in 3D CityDB implementing the CityGML standard (Yao et al., 2018). The selection of the storage form depends mostly on requirements for data exchange among users. However, in case of visualisation or spatial analysis, the model must be eventually loaded in a computer memory. Models covering big areas, for instant a city, consisted of many detailed objects are too big to fit into memory. On the other hand, solutions, which perform all spatial operations entirely in a database environment (Goudarzi et al., 2015) are not efficient in terms of computation time.

One of the solutions is to divide a model into small parts, called tiles, store them in a database or file, and load them when necessary. They form a mesh of tiles covering the whole area taken by the model (Campos et al., 2020; Oliveira and Rocha, 2013). For instance, in visualisation of a city model, selected tiles are loaded if they are in a close range from an observer (usually it is a camera in a rendering engine). This applies to models consisted of individual objects with a simple geometry, e.g. building envelopes in a city model. A vector tiling concept was proposed as a part of an open standard developed by Open Geospatial Consortium (OGC, 2018). It includes, among others, regular tiling and geoJSON format, which are utilised in this research. Vector tiling is implemented in the OGC format designed for streaming massive 3D data – 3D Tiles (OGC, 2019). Tile sets may include different formats organised using an internal data structure.

Another example might be a triangular irregular network (TIN) representing a digital terrain model, which is too big to be loaded at once. In this case, it is necessary to deal with one complex model instead of many unconnected objects. Individual edges or triangles can be assigned to tiles and displayed when necessary. It works well in case of visualisation, when the focus is put on the geometry but not topology, i.e. connections between individual elements. However, if the model is intended for further analysis, topology is essential and should be preserved in parts loaded into computer memory. The same applies to complex models, such as BIM models. In order to perform spatial analysis of a building interior, a logical network together with the geometry is expected to be present in computer memory. This is needed by various graph-based applications, e.g. indoor navigation and route planning.

Implementation of the aforementioned models requires an appropriate data structure, which is suitable for a boundary representation and is able to store the geometry and topology at the same time. There are several data structures available (Arroyo Ohori et al., 2015): the cell-tuple structure (Brisson, 1989), combinatorial maps or G-maps (Lienhardt, 1991) are widely recognised n-dimensional solutions. The dual half-edge (DHE) (Boguslawski and Gold, 2016) or Compact Abstract Cell

* corresponding author

Complexes (Ujang et al., 2019) are relatively new concepts, which can be also considered for the anticipated task. In this research, the DHE was selected, because it can be implemented in a database environment using a simple table structure. There are only two atomic construction elements, i.e. half-edges and nodes, which can be stored in two tables (Goudarzi et al., 2015). Thanks to the duality concept, other entity types are available without any additional tables: 3D cell, face, edge and vertex. Semantic information in a form of attributes is attached to an atomic element and can be considered as attached to a cell, face, edge or vertex. For instance, an attribute describing a room number can be attached to a dual vertex representing the room cell, which is a single record in a table. In Figure 1 the model consists of two cells, which are identified as individual dual vertices – attributes can be attached to these vertices. They are linked by a dual edge.

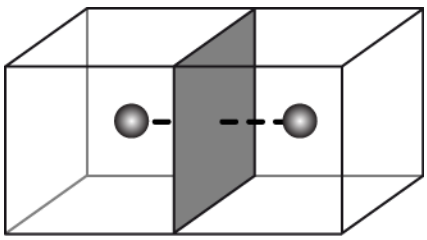

Figure 1. Two primal cells and their dual nodes linked by a dual edge (black dashed line).

DHE is a topological data structure. Each half-edge consists of five pointers: *V*, *S*, N_V, N_F and *D*, which point at other elements of the model. *V* points at the node assigned to the half-edge. Two half-edges forming an edge are linked by *S*. The next half-edge around a shared node is pointed by N_V, while the next half-edge in a loop forming a face is pointed by N_F. A couple of primal and dual half-edges are linked by *D*. The pointers are used to define basic navigation operators *Sym*, $Next_V$, $Next_F$, and *Dual*, which directly use *S*, N_V, N_F and *D* respectively. Basic operators are used to define compound operators: $Next_E$ and *Adjacent*. Selected operators are shown in Figure 2.

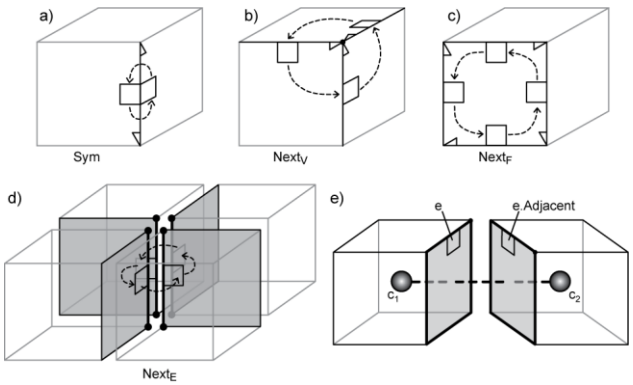

Figure 2. Navigation operators: a) Sym, b) $Next_V$, c) $Next_F$, d) $Next_E$, e) Adjacent (Boguslawski and Gold, 2016).

Another advantage of using the DHE is availability of construction operators providing an intuitive way to create and update the model. A set of operators conforms to Computer-Aided Design (CAD) modelling rules, which makes the construction process user friendly. The only limitation of the DHE is that it is capable of representing only 3D models. Further development would be necessary to extend it to higher dimensions. However, in the presented research it is not required.

In this paper, a simple method for a progressive loading of a 3D model geometry and topology stored in a database using the DHE data structure will be presented. A programming user interface allows for a transparent loading from a database to a computer memory and reverse operation of database updating, where modified elements are put back in the database. The proposed solution do not deal with individual basic elements, which in case of this research are half-edges and edges.

2. METHODOLOGY

There are two spatial models investigated in this research. The first one is a 2D TIN, which is used to present the principles of the described idea and test its efficiency. The second one is an indoor model of a building introduced to present a 3D approach. Pre-prepared models were imported to a PostgreSQL database. Time efficiency of data loading from the database to computer memory was tested for bulk and atomic single-edge queries. Comparative analysis provides a relative ratio indicator of the computation time. The developed prototype implementation is based on Python, which is a script programming language. Thus, absolute computation time, also provided in this paper, can be improved by reimplementation of the code in a compiled language, e.g. C++.

2.1 Input data

Three input datasets were used to develop spatial models: a list of points representing locations of Polish cities, a building mock dataset generated in Autodesk Revit and a city model in the CityGML format. All datasets were processed and implemented using the DHE data structure.

The list of cities and the city model are open datasets obtained from the Head Office of Geodesy and Cartography (GUGiK) via the national geoportal[1]. The dataset including all Polish cities and villages was filtered in order to retrieve information on location of cities. There are 124,799 items in total, including 954 cities (as of December 2021). Filtered data was used to compute TIN consisted of 2,845 edges and 1,892 triangles (see Figure 3).

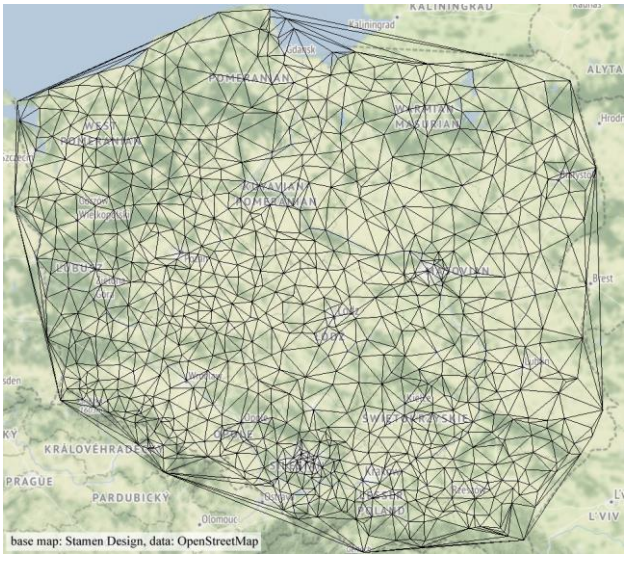

Figure 3. TIN model: triangulation of points representing locations of Polish cities.

[1] https://mapy.geoportal.gov.pl

The 3D mock dataset is a 11-storey building model generated in Autodesk Revit. It was exported to the gbXML format and processed in order to filter indoor spaces and openings (i.e. doors and windows) as well as to retrieve information about spatial relations among individual spaces based on methodology presented by Boguslawski et al. (2016). The result model (see Figure 4) is a cell complex consisted of a set of indoor spaces and zero-volume openings. Each cell is associated with a dual vertex and adjacent cells are connected by dual edges (dual structure is not shown in the picture). In total, there are 2,730 vertices and 19,224 edges in the model (including primal and dual edges). It should be noted that adjacent cells do not share edges, but each cell have their own set edges even the geometry of adjacent edges is the same. Thus, in order to display the building model there are 3,066 unique-geometry edges selected.

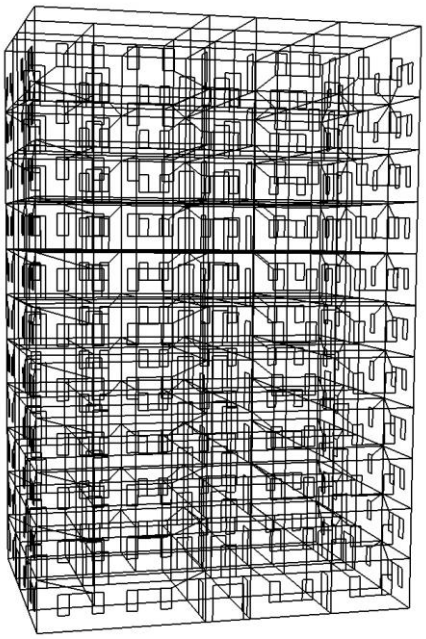

Figure 4. Mock dataset: a building model.

The third example, the city model shown in Figure 5, covers a certain area of interest (about 80x80 km) in the south part of Poland, in the Silesian Voivodship. It consists of 554,763 buildings in CityGML LoD2. The model converted to the DHE representation an stored in a database includes 63,594,268 primal and dual edges (stored in 127,188,536 rows in a database) and 11,047,740 vertices. In this case, there are 15,898,567 geometrically unique edges and 10,492,976 vertices in the primal, which are used for visualisation of the model.

In the conversion process the 'Cardboard and Tape' construction method is used to build a model (Boguslawski and Gold, 2010). The geometry of buildings in original CityGML files is represented as a set of faces, where a face is a list of consecutive vertices around the boundary. Individual faces of the DHE model are created based on the list of vertices and then joined by matching edges, which form a closed cell. Missing faces necessary to make a closed cell are detected and reported during the construction process. Thus, the final model consists of topologically valid cells representing buildings.

For the clarity of presentation, basic ideas related to programming interface for data loading will be illustrated using the TIN model, while the building and city models will be used as a proof of concept.

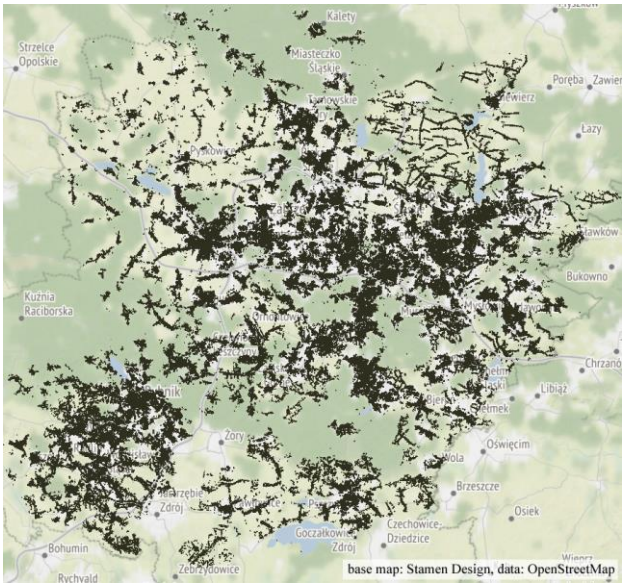

Figure 5. City model: 554,763 buildings in CityGML LoD2.

2.2 Database storage

Models implemented using DHE can be stored in two tables: vertex (see Table 1) and half-edge (see Table 2) storing the geometry and topology accordingly. Each entity, vertex and half-edge, is identified by *id*. *v* in the half-edge table is a foreign key from the vertex table, while *s*, *nv*, *nf* and *d* are identifiers of linked entities from the half-edge table. *Attrib* stores attributes attached to a vertex or edge in the JSON format.

Name	Type	Not NULL	Primary key
id	bigint	x	x
x	double	x	
y	double	x	
z	double	x	
attrib	varchar		

Table 1. Vertex table

Name	Type	Not NULL	Primary key
id	bigint	x	x
v	bigint	x	
s	bigint	x	
nv	bigint	x	
nf	bigint	x	
d	bigint	x	
attrib	varchar		

Table 2. Half-edge table.

2.3 DHE implementation

Transparent loading of a model from a database to computer memory requires new implementation of DHE navigation operators. The original version was designed to deal with a complete model present in computer memory. In the new concept, a model may be loaded only partially. Thus, in case of missing elements (usually on boundaries of the loaded model), they will be collected from a database when necessary. For the clarity of description it is assumed that all vertices are loaded

into the memory in advance, while provided examples will focus on operations related to half-edges.

Original navigation operators: *Sym*, *NextV*, *NextF* and *Dual*, are functions, which return one of the pointers of the DHE data structure: *s*, *nv*, *nf* and *d* respectively. In the new version they work in the same way, if a certain pointer points at a half-edge, which is already loaded. Otherwise, an attribute with the target half-edge *id* is added and a navigation operator is 'redirected' to a special function, which role is to collect the half-edge from a database. This means that collection of a half-edge from a database is triggered by a navigation operator. Once the half-edge is loaded, the navigation operator works in a standard way.

In this implementation, there is a list of loaded half-edges included (the list is implemented as a Python dictionary). This allows for a quick check up of available half-edges and incremental loading of missing ones if required. The following pseudo-code shows the idea by providing only one operator – *Sym*. The rest of operators: *NextV*, *NextF* and *Dual* may be readily implemented based on the provided example.

Input:
eList – a global key-value list of loaded half-edges (originally empty)
vList – a global key-value list of pre-loaded vertices from the vertex table

```
class dhe
  id: integer (identifier)
  v, s, nv, nf, d: dhe = NULL
  attrib: {} // a key-value list of attributes
  Sym = &getSFromStorage //assign the getSFromStorage
                         //method to the Sym operator

  method getSFromStorage()
      if attrib['s'] != NULL
        setSym(getEdgeFromDB(attrib['s']))
      return s

  method setSym(he)
    if he == NULL
      if s != NULL
        attrib['s'] =s.id
        s = NULL
        Sym =&getSFromStorage
    else
      s = he
      Sym =&getSym  //assign the getSym
                    //method to the Sym pointer
      attrib.remove('s')

  method getEdgeFromDB(edgeId)
      he = eList[id] //get a half-edge from eList
      if he == NULL
          rec = executeSQL("SELECT id, v, s, nv, nf, d, attrib
                FROM half-edge WHERE id="+edgeId)
          ID, IDV, IDSym, IDNextV, IDNextF, IDD, attrib=rec
          he = new dhe
          he.id = ID
          eList[ID] = he
          he.v = vList[IDV]
          he.attrib['s'] = IDSym
          he.attrib['nv'] = IDNextV
          he.attrib['nf'] = IDNextF
          he.attrib['d'] = IDD
      return he
```

dhe is a class representing an individual half-edge. When a new object is created, an identifier is assigned to the *id* property. The rest of properties are set to an empty value, i.e. NULL, and navigation operators are set to special methods, which read data from a database. In order to initiate the model, the root edge should be explicitly read by calling *getEdgeFromDB* (e.g. rootEdge = getEdgeFromDB(1)), which will fill the *attrib* property with identifiers of linked half-edges. Once an operator is called for a newly created edge *e*, e.g. *e.Sym()*, an identifier *e.attrib['s']* is used to collect a new half-edge from a database using the *getEdgeFromDB* function, which is then assigned to *e.s*. *e.Sym* is redirected to the *getSym* function, which sets the standard role of the *Sym* operator.

In any case, when *getEdgeFromDB* is called, the availability of the half-edge is checked in *eList*, which includes loaded half-edges. If it is not loaded then the database is queried. A similar idea of loading may be applied to *vList*, which is a list of vertices in the memory. In this example, it is assumed that all vertices are preloaded and put on the list.

The navigation operators query a database each time there is no required half-edge present in *eList*. Thus, the number of queries is the same as the number of half-edges to be collected. In order to reduce the number of queries, a modified version of *getEdgeFromDB* was tested. Instead of collecting and processing half-edge by half-edge, full edges, i.e. two half-edges linked by *s*, are collected in one query. Modifications affecting the code are related to the SQL query and assignment of *s*. A half-edge of a given *id* and the coupling half-edge *s* are loaded from the database and two *dhe* objects are created at the same time.

Another important aspect of the loading mechanism is transferring half-edges from the memory back to the database, which allows for releasing of memory resources. In this implementation, the focus will be put only on attributes. Changes of the model, its geometry or topology, and propagation of these changes to the database will not be considered. The following pseudo-code shows the *putEdgeToDB* method, which removes a half-edge from a list of loaded half-edges, updates the *attrib* field and removes references to the removed entity from other loaded half-edges. For the sake of clarity, the following code includes only operations related to the *s* pointer, while the rest: *nv*, *nf* and *d* can be implemented in a similar way.

```
method putEdgeToDB(he):
    eList.remove(he.id)
    executeSQL("UPDATE half-edge SET attrib='"+
        he.attrib+"' WHERE id"+ he.id)  //update edge attributes
                                        //in the database

    rec = executeSQL("SELECT id FROM half-edge
        WHERE s="+he.id+" OR nv="+he.id+" OR
        nf="+he.id+"  OR d="+he.id //collect all half-edges
                                   //linked to he
    for all in rec
        ID = rec
        he = eList[ID]
        if he != NULL and he.s != NULL and he.s.id == id:
            he.setSym(NULL) //do the same for nv, nf and d
```

2.4 Graph traversal functions

Models implemented using the DHE data structure form a graph, which can be traversed using graph algorithms. In order to test the new implementation, two graph-based functions were developed:

- getCellEdges – takes a root half-edge parameter and returns a list of edges forming a cell, i.e. polyhedron; breadth-first graph traversal algorithm is applied;
- getAllEdges – calls getCellEdges for each cell in a complex and returns all edges of a model in one space – primal or dual; dual graph is used to go from one cell to another.

A function for a bulk load was also provided. It collects all half-edges from a database in a single query and put them on the list – *eList*. It should be noted that the computation time of *getCellEdges* and *getAllEdges* will be affected if the bulk load is done before the functions are called, as no database query is made if a half-edge exists on the list. The computation time of graph-based functions will be referenced to the time of a bulk load.

3. RESULTS

The code was run on a computer with following properties: Intel Core i9-11900K, 32 GB RAM, Windows 10 (64-bit), Python 3.10, PostgreSQL 14.1.

3.1 Model loading

A computation time of a bulk load from a database was compared with model loading using graph-based functions presented in section 2.4. They were applied for TIN and building models (shown in Figure 3 and Figure 4). In case of the TIN model, *getCellEdges* was applied as there is only one cell in this 2D model. In case of the building model, *getAllEdges* is applied in order to collect edges from all the cells.

Computation time of the TIN model loading is provided in Table 3. It is an average value of five independent computation attempts for each function version. The bulk load results in collection of 5,690 half-edges from a database, while getCellEdges returns a list of 2,845 edges. It should be noted that existence of dual edges in the TIN model is not necessary, as the model consists of one cell. Therefore the dual structure is not included in the database. This also demonstrates that incomplete models can be also utilised in this new approach, which was not possible with the original DHE version.

Function version	Computation time [s]	Computation time ratio
bulk load	0.022	1
getCellEdges (preloaded)	0.026	1.17
getCellEdges (half-edge)	0.341	15.53
getCellEdges (edge)	0.304	13.85

Table 3. Loading time of the TIN model.

The bulk option is the fastest one comparing to any version of *getCellEdges*, because there is only one database query performed. However, *getCellEdges* is only 1.17 times slower when the model is preloaded. This shows that graph traversal is relatively fast comparing to data collection from a database.

The bigger difference is observed when the model is collected half-edge by half-edge (15.53 times slower) and edge by edge (13.85 times slower). There are respectively 5,690 and 2,845 separate database queries necessary to read the model.

The building model consists of 642 cells built of 38,448 primal and dual half-edges. There are about 6.75 times more-half edges than in the TIN model, while the bulk load time is 7.34 times higher. The execution time of *getAllEdges* for the preloaded model, half-edge by half-edge and edge by edge versions is 1.24, 12.86 and 11.80 times higher respectively.

Function version	Computation time [s]	Computation time ratio
bulk load	0.161	1
getAllEdges (preloaded)	0.203	1.24
getAllEdges (half-edge)	2.080	12.86
getAllEdges (edge)	1.895	11.80

Table 4. Loading time of the building model.

A time of model visualisation is not taken into consideration in above analysis.

3.2 Database updating

In the following experiment, edges from the original TIN model (see Figure 6a) were removed from the memory. 250 and 750 edges (see Figure 6b and c) that were loaded first by *getCellEdges* were put back to the database including their attributes using *putEdgeToDB* (see section 2.3). Next, some edges were reloaded (see Figure 6d). This shows that the model can be partially present in the memory, while the rest can be loaded when necessary. Also some edges may be unloaded in case the model takes too much space in the memory.

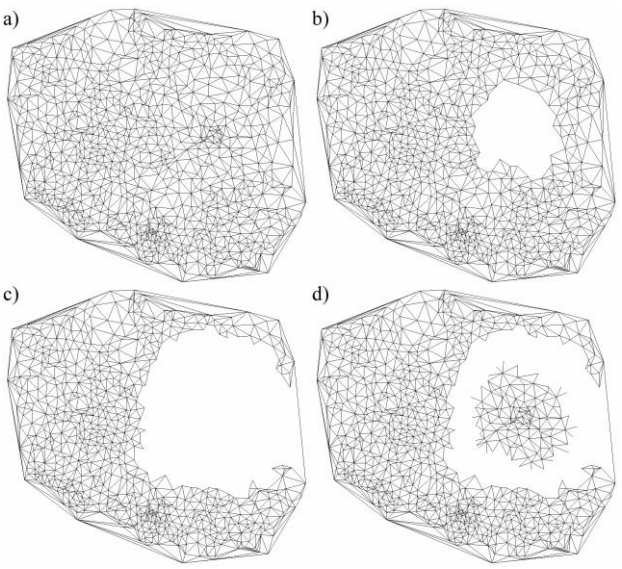

Figure 6. TIN model: a) original; b) 250 edges removed; c) 750 edges removed; d) new edges reloaded.

3.3 Model tiling

In order to present a big data application, a part of the city model (5x5 km area covering the city of Katowice shown in Figure 7) was used to prepare a set of regular tiles (1x1 km) in the geoJSON format. 10,301 buildings where divided into 25 tiles, with the number of buildings varying from 15 to 1,438. An average time of a database query and geoJSON file preparation

for a single tile is 170 s. A fragment of the 3D city model is shown in Figure 8.

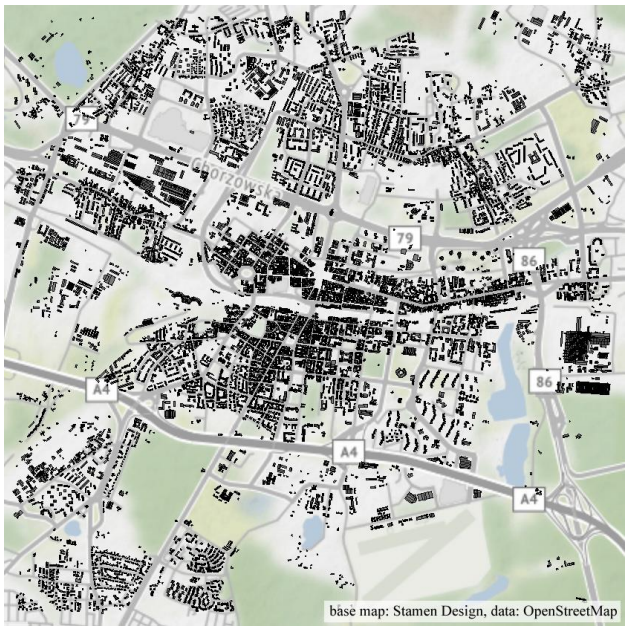

Figure 7. A fragment of the city model, 5x5 km including 10,301 buildings.

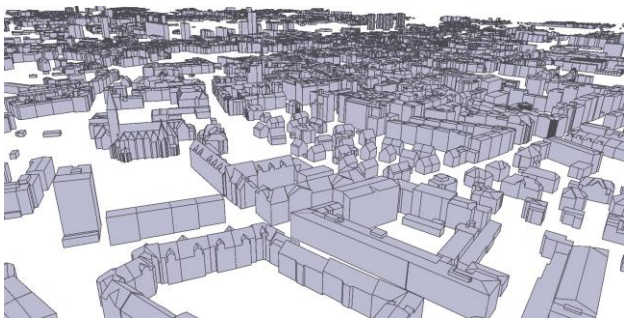

Figure 8. 3D city model in the geoJSON format.

4. CONCLUSIONS

In the paper, a database storage of big spatial models in boundary representation implemented with the DHE data structure as well as a transparent mechanism of loading the model to computer memory was shown. Parts of a model may be also put back in the database while preserving semantic information attached to atomic entities as attributes. The functionality of the proposed solution was tested in a prototype developed in Python in connection with PostgreSQL.

Loading of partial models is an important issue when dealing with big spatial datasets. They may represent an area of a city or country and include terrain model, buildings, transportation network and other infrastructures. They are too big to be put as a whole in computer memory. Excessive data is transferred to a virtual memory on a hard drive, which significantly extends the access time and influences efficiency of analysis. Also, the time necessary to load the model on edge-by-edge basis is high and needs an optimised solution. A mechanism based on regular tiling, where a tile covering a relatively small part of the model is loaded in bulk, was presented using a city model consisted of several thousands of buildings.

The city model retrieved from open datasets stored in the CityGML format was implemented with the DHE. This required 3D topology reconstruction in order to introduce a valid model with required topological connections among model entities, i.e. half-edges, which allowed testing topological validity of original data.

One of developments envisaged in a future is an automated mechanism for loading of optimal number of edges from a database. In this solution, tiles will be loaded and unloaded depending on utilisation of the memory. Edges, which were not accessed for a long period of time will be stored back in the database.

Future works also include propagation of geometry and topology changes done in a computer memory back to a database. This is an essential functionality in case of dynamic models.

The last but not least planned improvement is implementation of the proposed solutions using compiled programming languages, e.g. C++ or Delphi. This should significantly improve computation time efficiency.

ACKNOWLEDGEMENTS

This work was supported by the National Science Centre (NCN), Poland as part of the research program OPUS, project no.: UMO-2021/41/B/ST10/03178.

REFERENCES

Arroyo Ohori, K., Ledoux, H., Stoter, J., 2015. An evaluation and classification of nD topological data structures for the representation of objects in a higher-dimensional GIS. International Journal of Geographical Information Science 29, 825-849.

Boguslawski, P., Gold, C., 2010. Euler Operators and Navigation of Multi-shell Building Models, in: Neutens, T., Maeyer, P. (Eds.), Developments in 3D Geo-Information Sciences. Springer, pp. 1-16.

Boguslawski, P., Gold, C., 2016. The Dual Half-Edge—A Topological Primal/Dual Data Structure and Construction Operators for Modelling and Manipulating Cell Complexes. ISPRS International Journal of Geo-Information 5, 19.

Boguslawski, P., Mahdjoubi, L., Zverovich, V., Fadli, F., 2016. Automated Construction of Variable Density Navigable Networks in a 3D Indoor Environment for Emergency Response. Automation in Construction 72, 115-128.

Brisson, E., 1989. Representing geometric structures in d dimensions: topology and order, Proceedings of the fifth annual symposium on Computational geometry. ACM, Saarbruchen, West Germany.

Buyukdemircioglu, M., Kocaman, S., 2020. Reconstruction and Efficient Visualization of Heterogeneous 3D City Models. Remote Sensing 12, 2128.

Campos, R., Quintana, J., Garcia, R., Schmitt, T., Spoelstra, G., M. A. Schaap, D., 2020. 3D Simplification Methods and Large Scale Terrain Tiling. Remote Sensing 12, 437.

Goudarzi, M., Asghari, M., Boguslawski, P., Rahman, A.A., 2015. DUAL HALF EDGE DATA STRUCTURE IN DATABASE FOR BIG DATA IN GIS. ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci. II-2/W2, 41-45.

ISO 16739:2018, 2018. Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries.

Lienhardt, P., 1991. Topological models for boundary representation: a comparison with n-dimensional generalized maps. Computer Aided Design 23, 59-82.

OGC, 2018. OGC Testbed-13: Vector Tiles Engineering Report. Open Geospatial Consortium.

OGC, 2019. 3D Tiles Specification 1.0. Open Geospatial Consortium.

Oliveira, N., Rocha, J.G., 2013. Tiling 3D Terrain Models, in: Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O. (Eds.), Computational Science and Its Applications – ICCSA 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 550-561.

Ujang, U., Anton Castro, F., Azri, S., 2019. Abstract Topological Data Structure for 3D Spatial Objects. ISPRS International Journal of Geo-Information 8, 102.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., Kolbe, T.H., 2018. 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. Open Geospatial Data, Software and Standards 3, 5.