SPATIO-SEMANTIC COMPARISON OF LARGE 3D CITY MODELS IN CITYGML USING A GRAPH DATABASE

S. H. Nguyen, Z. Yao, T. H. Kolbe

Technical University of Munich (TUM), Dept. of Civil, Geo and Environmental Engineering, D-80333 Munich, Germany - (son.nguyen, zhihang.yao, thomas.kolbe)@tum.de

KEY WORDS: CityGML, Spatio-semantic Comparison, Change Detection, Graph Database, Web Feature Service

ABSTRACT:

A city may have multiple CityGML documents recorded at different times or surveyed by different users. To analyse the city's evolution over a given period of time, as well as to update or edit the city model without negating modifications made by other users, it is of utmost importance to first compare, detect and locate spatio-semantic changes between CityGML datasets. This is however difficult due to the fact that CityGML elements belong to a complex hierarchical structure containing multi-level deep associations, which can basically be considered as a graph. Moreover, CityGML allows multiple syntactic ways to define an object leading to syntactic ambiguities in the exchange format. Furthermore, CityGML is capable of including not only 3D urban objects' graphical appearances but also their semantic properties. Since to date, no known algorithm is capable of detecting spatio-semantic changes in CityGML documents, a frequent approach is to replace the older models completely with the newer ones, which not only costs computational resources, but also loses track of collaborative and chronological changes. Thus, this research proposes an approach capable of comparing two arbitrarily large-sized CityGML documents on both semantic and geometric level. Detected deviations are then attached to their respective sources and can easily be retrieved on demand. As a result, updating a 3D city model using this approach is much more efficient as only real changes are committed. To achieve this, the research employs a graph database as the main data structure for storing and processing CityGML datasets in three major steps: mapping, matching and updating. The mapping process transforms input CityGML documents into respective graph representations. The matching process compares these graphs and attaches edit operations on the fly. Found changes can then be executed using the Web Feature Service (WFS), the standard interface for updating geographical features across the web.

1. INTRODUCTION

As an official OGC standard for encoding virtual 3D city models, CityGML opens up opportunities for applications in a broad range of areas such as urban planning, facility management, environmental simulations and thematic inquiries. One of the main factors contributing to this success is CityGML's capability of including not only 3D urban objects' graphical appearances, but also their semantic properties (e.g. relationships between objects). This ensures CityGML documents can be shared over various applications that make use of the model's common semantic information, which is "especially important with respect to the cost-effective sustainable maintenance of 3D city models" (Gröger et al., 2012).

However, although the increasing number of CityGML datasets in recent years indicates a positive sign of the open standard's steady growth, it has also been a great challenge to maintain sustainable 3D city models. One prominent example is the difficulty of handling undocumented collaborative as well as chronological changes of an existing city model, which is currently unavoidable due to the fact that as cities grow over time, so does the need to adjust their models accordingly (Navratil et al., 2010). Furthermore, because the current state of CityGML does not store these changes in its instances, multiple model documents of the same city may accumulate over time. As a result, during the maintenance phase, old city datasets are overwritten completely by newer ones, which not only costs a large amount of time and computational resources, but also loses track of collaborative changes and neglects the city's progress recorded during the given time period. Moreover, replacing entire large datasets due to only some small changes would cause an unnecessarily huge volume of transactions, especially if the database is managed via the Web Feature Service (WFS).

Therefore, instead of replacing older records, an ideal alternative should first compare the models, then attach edit operations to detected deviations, based on which only real changes are committed. This way, older datasets can both be updated and still retain their respective core structure, including own rules of syntax and internal object references. This plays a key role in enabling a version control system for collaborative work in modelling and storing digital 3D city models (Chaturvedi et al., 2015). Moreover, the number of transactions required for a WFS-enabled database is also reduced significantly.

In order to achieve this, the first task is to determine key aspects, based on which CityGML models should be compared. Since "one of the most important design principles for CityGML is the coherent modelling of semantics and geometrical/topological properties" (Gröger et al., 2012), a comparison between two CityGML instances should take into account both their geometrical and semantic aspects to ensure reliable results. For example, wall surfaces can be defined in-line or referenced to other existing walls of surrounding buildings via the XML Linking Language (XLink). Thus, the further question arises as to how geometrical and semantic information of CityGML documents can be compared. Considering the facts that CityGML elements belong to a complex graph-like structure, syntactic ambiguities (such as between XLink and in-line objects) can be disambiguated using a graph. Since CityGML documents can be very large in size, an approach to detect spatio-semantic changes in arbitrarily large-sized CityGML datasets utilizing a graph database is proposed.

In Section 2, some related tools and algorithms are discussed. Then, the implementation of this research is explained, which consists of three main parts: mapping CityGML datasets onto a graph database (Section 3), matching mapped graphs (Sections 4

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-IV-4-W5-99-2017 | © Authors 2017. CC BY 4.0 License.

and 5) and updating an existing CityGML database based on detected deviations (Section 6). An example of the proposed approach is introduced in Section 7. Section 8 further presents the results of some experiments conducted in this research. At the end, Section 9 summarizes and concludes the paper.

2. RELATED WORK

Conventional `diff` tools, such as the Hunt–McIlroy algorithm (Hunt and McIlroy, 1976), are only able to find deviations in pure texts and thus not compatible with highly structured data models employed in Geographic Information System (GIS). Bakillah et al., 2009 proposed a conceptual basis for a semantic similarity model (Sim-Net) for ad hoc network based on the multi-view paradigm. Olteanu et al., 2006 addressed the automatic matching of imperfect geospatial data during database integration. However, since each of these researches mainly focused on either the semantic or geometrical aspect of city objects, they are not fully applicable to CityGML, which provides an integrated view of both aspects.

Later, Redweik and Becker, 2015 presented a concept for detecting semantic and geometrical changes in CityGML documents. Since CityGML is an application schema of XML, which is a tree data structure, by assuming that CityGML instances can also be represented as trees, they extended the algorithm "X-Diff" (Wang et al., 2003) that considers tree equivalence as isomorphism. However, in contrast to XML, CityGML is not a tree but a graph data structure by definition, as it may contain cycles and nodes linked by multiple parents (e.g. due to XLinks). Therefore, this approach is generally not expressive enough regarding CityGML's graph data structure. Moreover, the methods proposed by Redweik and Becker, 2015 are not yet evaluated against massive input datasets.

To deal with large input datasets, the ability to efficiently preselect potential matching candidates based on their spatial properties is of advantage. Topologically relative allocations of objects can be expressed by the "4" or "9-intersection model" ("4-IM" or "9-IM") (Egenhofer and Franzosa, 1991; Egenhofer and Herring, 1991). Moreover, an object can be localized by recursively dividing its parent graph into quadtrees (2D) or octrees (3D) and colouring their interior as well as exterior (Berg et al., 2008). Alternatively, an R-tree can be applied to spatial objects grouped in regions based on their topological properties (Guttman, 1984). Since R-trees are balanced, their query response time in logarithmic time complexity $\mathcal{O}(\log_M n)$ is very efficient in large databases, where M is the maximum number of entries allowed per internal node.

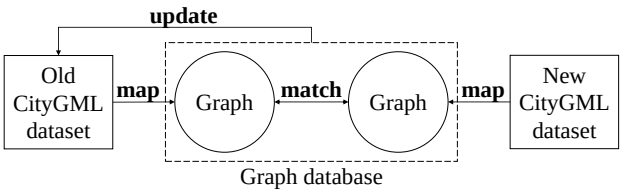


Figure 1. An overview of three major steps mapping, matching and updating of 3D city models using a graph database.

3. MAPPING 3D CITY MODELS ONTO A GRAPH DATABASE

The overall implementation consists of three major parts as shown in Figure 1, in all of which the graph database **Neo4j** is employed. In Neo4j, each city object is stored as a graph node, while the relationships between these objects are represented as edges between nodes. In other words, nodes are connected directly to each other. This is particularly useful in data models that have a complex and multi-level deep hierarchical structure like CityGML. In this section, the mapping of city objects onto Neo4j graphs shall be explained in several smaller steps.

3.1 Reading CityGML Datasets in Java

CityGML documents can be processed with the help of various XML parsing APIs in Java such as the Document Object Model (DOM), Simple API for XML (SAX), Streaming API for XML (StAX) or Java Architecture for XML Binding (JAXB). Each API comes with their own advantages and disadvantages depending on the application domain. Considering the fact that CityGML datasets can grow quickly in size, the library **citygml4j** employs a combination of JAXB and SAX (Nagel, 2017), which allows partial unmarshalling (or deserialization) of CityGML elements into Java objects with efficient memory consumption (see Figures 2 and 3a). Moreover, this approach provides an object-oriented view of read CityGML data, which facilitates the transformation of unmarshalled Java objects to graph entities in the next step.

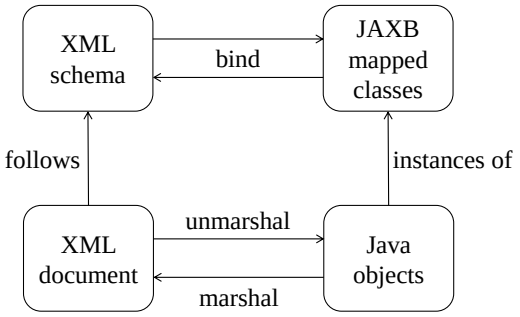
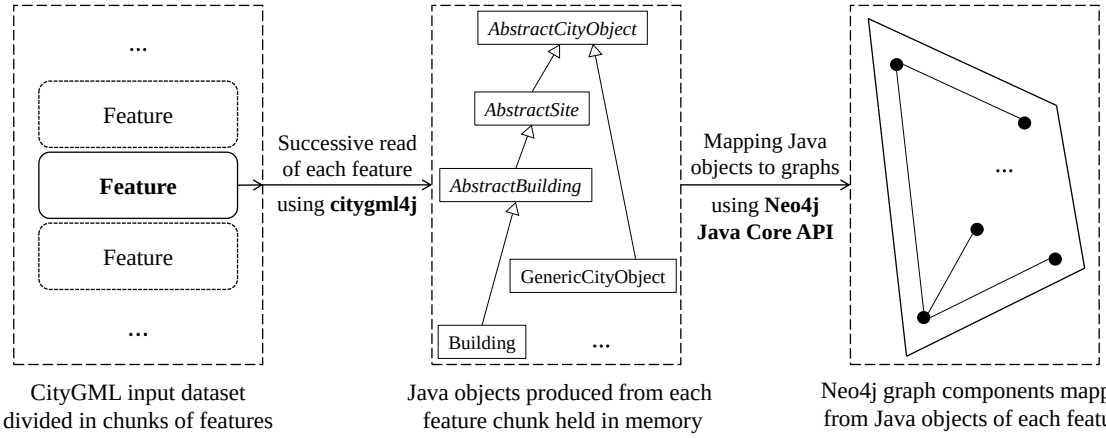


Figure 2. The JAXB binding process. Adapted from Oracle Corporation, 2015.

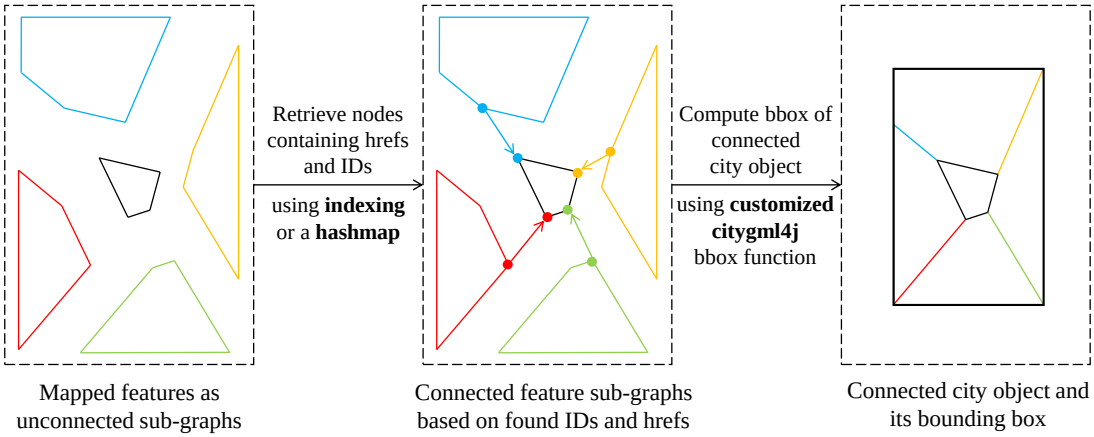
3.2 Converting Java Objects to Graph Entities

To transform previously unmarshalled Java objects to corresponding graph entities in Neo4j conserving as much data as possible (see Figure 3a), the **Neo4j Java Core API** is employed. Conceptually however, two major challenges arise. Firstly, unmarshalled Java instances belong to a complex class hierarchy defined by the XML schema of CityGML. This poses the difficulty in designing suitable graph structures, such as how to efficiently represent different instances of the same class. Secondly, Neo4j is a value-based graph database, which means that no explicit schema modelling is possible. As a result, compared to XML-structure, it is difficult to map Java objects onto graph entities without losing any information, especially their hierarchical inheritance relations.

To resolve these challenges, an approach capable of creating graph representations of given Java objects using their hierarchical information is proposed, which is described in Algorithm 1. The key concept is the use of a central expandable `container` node, where all (i.e. own and inherited) attributes and references of the respective Java object can be stored successively for each superclass. Algorithm 1 produces compact but expressive graphs while still maintaining a robust and efficient implementation, as most functions can be recycled due to the use of hierarchical modelling. Moreover, this method is capable of capturing almost all information available in given Java objects with the only exception of explicit hierarchical relations, since the graph database is value-based. This however does not play a role in the matching process between mapped graphs. An illustration can be found in Figure 4.



(a) Unmarshalling CityGML documents and mapping Java objects onto graphs.



(b) Resolving XLinks and computing minimum bounding boxes (as a preparation for the matching process).

Figure 3. An overview of the mapping process.

Algorithm 1: `map(instance, container)`

Input : A Java *instance*
Output : Created node in a graph database

```
if container is null then
    create a node in graph database;
    set node.label equal to instance.className;
    set container equal to this node;
end
initialize attributes as a set of local attributes and
  references of instance;
foreach attribute of attributes do
    if attribute can be stored as simple texts then
        store attribute as a property in container;
    else
        create a child node via map(attribute, null);
        create a relationship from container to child;
    end
end
if instance inherits SuperClass then
    call map((SuperClass) instance, container);
end
return container;
```

3.3 Connecting Mapped City Objects using XLinks

Section 3.1 addresses the problems of parsing large CityGML documents concerning memory consumption and proposes an approach dividing them into smaller feature chunks. Each chunk is then separately unmarshalled and consequently mapped onto sub-graphs in Neo4j as described in Section 3.2. As a result, the explicit connections between split features and their respective parent elements are lost during the process (see Figure 3b). To allow the subsequent recovery of severed connections, before splitting, the library citygml4j stores IDs of affected features in XLinks (or `hrefs`) and attaches them to respective parent elements. XLink is a simple yet practical means to referencing or reusing existing elements without having to define them "in-line" repeatedly and thus essentially reduces redundancies in XML documents (Bray et al., 2008; DeRose et al., 2010). However, despite their syntactic differences, both XLink and in-line declaration are often used to effectively define the same objects.

Therefore, the reconstruction of lost connections between features and their respective parents as well as the syntactic disambiguation between objects defined in-line or by hyperlinks can be achieved by resolving XLink nodes in a graph database. This can be realized in two different approaches using internal hash maps held in memory or Neo4j's indices stored on disk. Each time a node containing an ID or `href` is encountered during the mapping process, a corresponding entry is stored in the respective index structure on

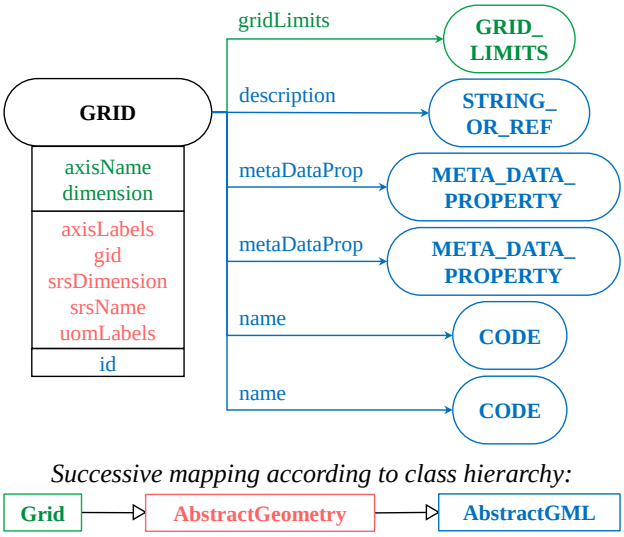


Figure 4. An example of a graph representing a `Grid` object. Rounded rectangles represent nodes. Node properties are displayed below nodes' label. The colours indicate the originating classes, in which nodes and properties are defined. The container `GRID` node is expanded successively for each superclass.

the fly. Then, XLinks can be resolved by linking indexed `hrefs` and IDs. Internal hash maps offer fast response time but come at the cost of memory consumption. On the contrary, Neo4j indices require less memory but may slow down the mapping process due to costly disk read and write operations. Moreover, sufficient additional storage space must also be reserved beforehand.

3.4 Calculating Minimum Bounding Boxes of Buildings

Ideally, in citygml4j, the minimum bounding box of a spatial Java city object (e.g. `Building`) can be computed by a built-in function that considers all of its geometric contents (e.g. boundary surfaces). However, this method has some limitations. Firstly, input Java objects must be completely available in memory as a whole, which is not always the case, since Section 3.1 shows that large city objects are to be split into features. Secondly, if Java objects have unresolvable XLinks (such as those contained in not yet loaded features), the function may fail. Thus, to overcome these limitations, graph representations, which are now connected and syntactically disambiguated as a result of Section 3.3, are reversely transformed to Java objects, from which respective minimum bounding boxes are calculated using the built-in function.

4. MATCHING 3D CITY MODELS USING GRAPH DATABASE

Since nodes play a central role in graphs, the matching process is based around the concept of their structure. In other words, two graphs can be matched by recursively comparing the properties and relationships of all of their respective nodes.

4.1 Comparing Node Properties

Actual data are mostly stored in node properties. Thus, differences found in node properties indicate possible deviations of respective data sources. Values from the same unique property name are compared with one another potentially followed by an update operation. Unmatched properties from the older and newer city model shall be attached with delete and insert operations respectively. An overview of all edit operations is shown in Section 6.1.

4.2 Matching Node Relationships

Matching relationships between two given nodes is complex considering the fact that relationships in Neo4j can be processed in both directions, namely `OUTGOING` and `INCOMING`. The matching process must however remain consistent in one specific traversing direction, so that no node is processed twice. The chosen direction is `OUTGOING`, as the matching process starts with root nodes. In contrast to node properties, a relationship may occur multiple times for a given node (i.e. 1 to n, n to 1 and n to n relationships).

Taking these into account, Algorithm 2 describes the main concept of matching relationships of two given nodes, where the function `find_candidate` in Line 5 plays a decisive role in terms of both efficiency and correctness of the whole matching process, as it dictates which object pairs should be compared to one another based on their specific characteristics. In CityGML, the most important aspect that can be used as a matching pattern among objects is their geometrical properties as well as spatial extents.

Algorithm 2: `match_relationships(node1, node2)`

Input : *node1* and *node2* of graphs representing nodes of old and new city model respectively

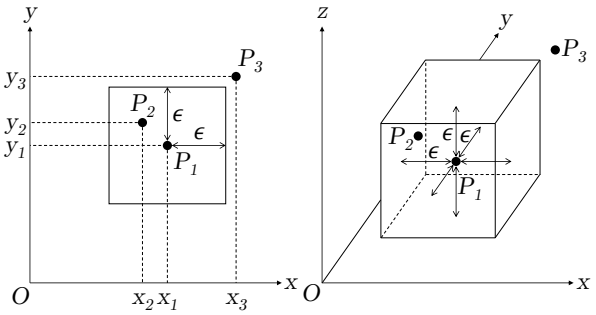
```
1  foreach matched rel_type of node1 and node2 do
2      chdr1 ← node1.get_children(rel_type);
3      chdr2 ← node2.get_children(rel_type);
4      foreach child1 of chdr1 do
5          child2 ← chdr2.find_candidate(child1);
6          if child2 is not empty then
7              match_node(child1, child2);
8          end
9      end
10     foreach unmatched child1 of chdr1 do
11         create a DELETE operation;
12     end
13     foreach unmatched child2 of chdr2 do
14         create an INSERT operation;
15     end
16 end
17 foreach unprocessed rel_type of node1 do
18     chdr1 ← node1.get_children(rel_type);
19     foreach child1 of chdr1 do
20         create a DELETE operation;
21     end
22 end
23 foreach unprocessed rel_type of node2 do
24     chdr2 ← node2.get_children(rel_type);
25     foreach child2 of chdr2 do
26         create an INSERT operation;
27     end
28 end
```

4.2.1 Matching Geometry of Points

Points are a primitive notion, upon which all other geometric objects are built. Since points do not have length, area or volume, the only property employed to distinguish them from others is their coordinates. In practice, however, real-world coordinates of the same point location may differ if they are given in different Spatial Reference Systems (SRS). Therefore, input CityGML instance documents should first be provided in the same spatial reference system.

On the other hand, even provided in one reference system, coordinates of two representations of the same point may still differ due

to numerical (such as rounding) and instrument errors. Such minor deviations should be tolerated. Thus, for a reference point P_1 as centre, depending on the chosen distance indicator, a neighbourhood $N(\epsilon)$ is constructed, where ϵ is the maximum empirically predetermined allowed distance tolerance. For example, if the Euclidean distance indicator is chosen, $N(\epsilon)$ shall be a circle (2D) or a sphere (3D). However, to calculate this distance, expensive operations such as square roots and multiplications are required. Since the research focuses on matching 3D objects of massive datasets, for a small ϵ, it is often sufficient to compare coordinates in each dimension, which requires only subtractions. In this case, $N(\epsilon)$ shall be a square (2D) or a cube (3D) (see Figure 5). A point P_2 is geometrically matched with point P_1 if, and only if, P_2 is located inside of $N(\epsilon)$ of P_1. Two geometrically matched points are equal and thus no further comparison is needed.



(a) $N(\epsilon)$ as a square in 2D. (b) $N(\epsilon)$ as a cube in 3D.

Figure 5. An illustration of the neighbourhood $N(\epsilon)$ of point P_1 in 2D (5a) and 3D (5b). P_1 is matched with P_2.

4.2.2 Matching Geometry of Line Segments Since line segments (or `LineStrings`) are composed of points, they can be geometrically matched by iterating over all control points and examining their spatial similarities successively with error tolerance ϵ taken into account (see Figure 6). Consecutive collinear line segments (given an empirically predetermined distance tolerance) can be merged together and thus treated as a single segment during matching. Alternatively, two `LineStrings` can be matched using the Buffer Overlay Statistics (BOS) methods (Tveite, 1999). Geometrically matched `LineStrings` are considered equal.

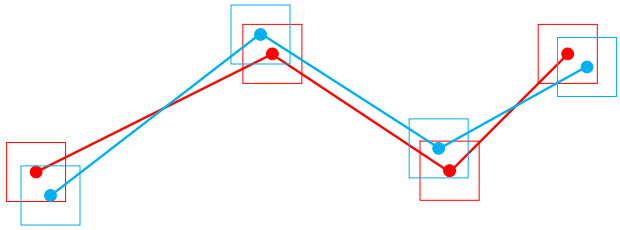


Figure 6. An example of two geometrically matched `LineStrings` considering error tolerances.

A more general concept of `LineStrings` is curves, each of whose curve segments may have a different interpolation method. A curve has a positive orientation. However, as long as such curves are composed of points, the same approach can be applied.

4.2.3 Matching Geometry of 3D Rings A ring in CityGML can be thought of as a closed `LineString` described in Section 4.2.2. Buildings in CityGML make extensive use of polygons (Section 4.2.4), whose boundaries are often represented as `LinearRings` (Cox et al., 2004; Gröger et al., 2012; Gröger, 2010). Although a `LinearRing` can theoretically consist of non-planar points, only planar `LinearRings` are considered.

The geometric comparisons of rings can be performed with the help of the libraries Abstract Window Toolkit (AWT) or Java Topology Suite (JTS). However, both of them are only applicable in two-dimensional space. Therefore, normal vectors (or orientations) of 3D rings are first computed and compared. Only rings that have similar orientations (given an empirically predetermined angle tolerance) are rotated to a plane parallel to a predefined reference one using a rotation matrix as illustrated in Figure 7. Then, their shapes are compared regardless of numbers or orders of contained points. Two shapes are geometrically equal if they "fuzzily" contain each other's points considering error tolerance ϵ.

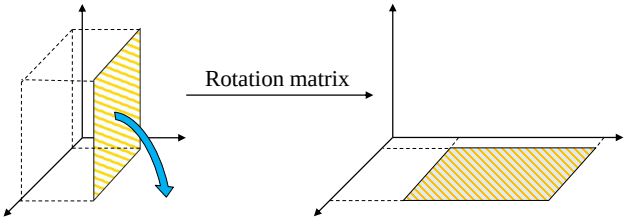


Figure 7. Rotating 3D (planar) ring.

4.2.4 Matching Geometry of 3D Polygons Polygons are extensively used in CityGML as a means to describe surfaces of buildings and building parts. A polygon consists of exactly one exterior and an arbitrary number of interiors, all of which are rings and must be on the same plane. While an exterior defines the outline, interiors define holes in a polygon (Cox et al., 2004; Gröger et al., 2012; Gröger, 2010).

Therefore, a polygon can be thought of as a shape bounded by an exterior with all interiors subtracted from its inner area. The geometric comparison of two polygons is then performed in the same manner as in Section 4.2.3. This can also be extended for the comparison of `Surface`, `OrientableSurface`, `MultiSurface` and `CompositeSurface` objects.

4.2.5 Matching Geometry of Solids A solid is bounded by a set of connected polygons, whose intersections are either empty or an edge shared by both respective polygons. A matching candidate of a given solid can be determined by using its footprint (as a polygon) or minimum bounding box (see Section 4.2.6). However, in contrast to previously discussed geometric entities, matched solid candidates may still be unequal since different solids can have the same footprint or minimum bounding box. Therefore, found candidates are further compared by successively matching their boundary polygons as described in Section 4.2.4.

4.2.6 Matching Geometry of Minimum Bounding Boxes The minimum bounding box of a building is calculated by all its contained geometries, such as ground, wall and roof surfaces (Section 3.4). To make full use of information available in all dimensions and thus increase the probability of finding correct matching candidates, minimum bounding boxes are compared based on their shared volume.

Given two arbitrary minimum bounding boxes represented by lower corner points P, R and upper corner points Q, S respectively, their own and shared volume are denoted by V_{PQ}, V_{RS} and V_{shared} respectively. For a given threshold $H \in (0,1]$, the following applies:

$$\text{Minimum bounding boxes } (P,Q) \text{ and } (R,S) \text{ are matched}$$
$$\iff \frac{V_{shared}}{V_{PQ}} \geq H \wedge \frac{V_{shared}}{V_{RS}} \geq H.$$

5. SPATIAL MATCHING USING AN R-TREE

Section 4.2, particularly Section 4.2.6, determines whether two geometric entities are equivalent and thus can be matched. However, repeatedly comparing all possible pairs of said objects results in a quadratic time complexity $\mathcal{O}(n^2)$. Thus, to enable more efficient object retrieval and querying based on their spatial properties, two matching strategies organizing buildings in an R-tree and a grid layout are employed in the course of this research, the former of which shall be explained in the following sections. For more details on the grid layout, please refer to (Nguyen, 2017).

5.1 Constructing the R-tree

R-trees are constructed using the plug-in **Neo4j Spatial**. Coordinates of lower and upper corner point of city models are not needed beforehand, since an R-tree automatically expands its envelope on the fly. Building footprints (or minimum bounding rectangles) are however required. While iterating, each building footprint is extracted or computed if not available (as described in Section 3.4). Splitting and merging nodes in an R-tree are handled by Neo4j Spatial, which ensures the tree structure is always balanced.

5.2 Assigning Buildings to the R-tree

The most important task while expanding an R-tree is to link spatial information to data sources it represents. For this purpose, a suitable adapter is needed (see Figure 8), where a connection between an R-tree node and the minimum bounding box of the respective building is established. Buildings are assigned to an R-tree on the fly. Note that a building is assigned to exactly one R-tree node.

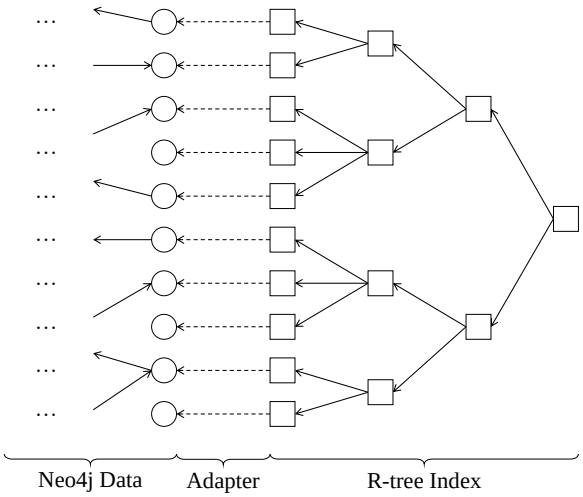


Figure 8. Illustration of an adapter (middle) connecting spatial indices in Neo4j Spatial (right) with data stored in Neo4j (left).

5.3 Matching Buildings using the R-tree

For each building of the older city model, a query containing its footprint is sent to the R-tree index layer stored in Neo4j Spatial. There, from the R-tree's root node, a corresponding internal node is reached, where the intersection tests take place. Leaf nodes, whose rectangles intersect with the queried footprint, are then returned together with respective linked buildings using constructed adapter. If no candidate is found, a delete operation shall be created for the current building. Otherwise, the best candidate among returned buildings is determined as described in Section 4.2.6. Finally, an insert operation is created for each remaining unmatched building in the newer city model.

The most important advantage of using an R-tree is the logarithmic time complexity $\mathcal{O}(\log_M n)$ on search operations. Moreover, with the help of Neo4j Spatial, employing an R-tree while matching is simple and straightforward.

6. UPDATING 3D CITY MODELS USING GRAPH DATABASE

The matching process in Sections 4 and 5 attaches edit operations to deviation sources on the fly while keeping the actual data untouched. These edit operations can then be executed accordingly in the updating process in this section.

6.1 Edit Operations

The general model of edit operations is shown in Figure 9. `EditOperation` is the superclass of all edit operations. It defines a `targetNode`, to which the edit operation is attached, and a flag `isOptional` indicating whether said operation should be executed. Such flag is mainly used for geometrically matched objects that are defined by different syntactic methods. The class `EditPropertyOperation` defines all edit operations created on node properties (i.e. object attributes), while `EditNodeOperation` defines edit operations on the node level (i.e. geo-objects).

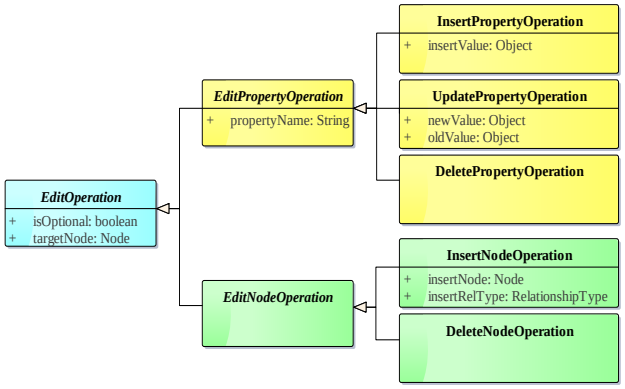


Figure 9. A UML class diagram of edit operations.

6.2 Updating Buildings using Web Feature Service (WFS)

`EditPropertyOperation` objects can be transformed to corresponding WFS transactions using their respective stored information. The same applies for `EditNodeOperation` with the only exception of `InsertNodeOperation`, which requires a payload (or content) encoded in XML (Vretanos, 2014). XML contents of affected entities can be retrieved by using a Graph-to-CityGML parser, which basically is the reverse of the mapping process introduced in Section 3 as described in Figure 10. For a more comprehensive look at the specifications of WFS requests and responses, please refer to (Vretanos, 2014) and (Nguyen, 2017).

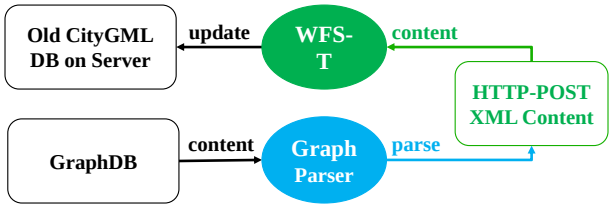


Figure 10. Retrieving XML contents of a CityGML object using a Graph-to-CityGML parser.

7. EXAMPLE

In this section, two syntactically different but geometrically equivalent polygons shall be compared (Figure 11). Both polygons contain one exterior `LinearRing` $ABCD$. However, the hole in the first polygon is represented by an interior `LinearRing` $E_1F_1G_1H_1I_1J_1$, while that of the second polygon is composed of two adjacent interiors, namely $E_2F_2G_2K_2$ and $K_2H_2I_2J_2$. Each `LinearRing` can be defined by either a `gml:posList` or a set of `gml:pos` or `gml:pointProperty` elements. The declared order and number of vertices (e.g. $E_2F_2G_2K_2$ or $G_2H_2K_2E_2F_2$) are insignificant. In addition, distance deviations within allowed error tolerances are detected between the interiors of both polygons.

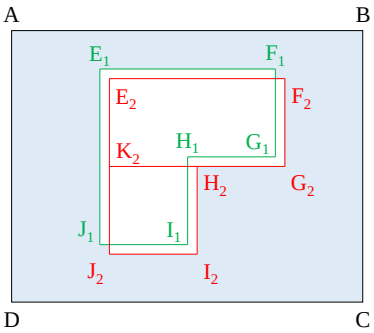


Figure 11. An example of two geometrically matched polygons (considering error tolerances) that are syntactically different.

The application returns a list of edit operations, which can be executed to make the first polygon identical to the second one. However, all of these operations are flagged as optional indicating that both polygons are correctly matched considering their equivalent geometry and error tolerances among coordinates.

8. APPLICATION RESULTS

8.1 Test Setup

All experiments in this research are performed on a dedicated server-class machine running SUSE Linux Enterprise Server 12 SP1 (64 bit) and equipped with Intel® Xeon® CPU E5-2667 v3 @3.20GHz (16 CPUs + Hyper-threading), a Solid-state Drive Array (SSD) connected via PCIe as well as 1 TB of main memory.

The tests use an input dataset of the 3D city model of Berlin. The dataset is encoded in CityGML v2.0.0, contains LOD2 information of 539,182 buildings and occupies 15.5 GB in physical storage.[1] The new dataset contains changes added manually to the old one.

8.2 Test Configurations

Both the testing system and Neo4j share the same Java Virtual Machine (JVM), which is provided with an initial and maximum heap space of 30 GB. The concurrent garbage collector `G1GC` is employed. The application configurations for the mapping and matching process are empirically determined to ensure a stable testing environment and optimum throughput. Namely, by default (unless specified otherwise), the following configurations are applied: multi-threading with 1 producer and 15 consumers, splitting CityGML elements per collection member (top-level feature), indexing using hash maps stored in memory while mapping, matching buildings using an R-tree with $M = 10$, batch size of 10 buildings and 5000 operations per database transaction.

[1]The CityGML datasets of Berlin are available under `http://www.businesslocationcenter.de/en/downloadportal`.

8.3 Experiment Results

8.3.1 Statistics of Mapped Graph Database A total number of 321,142,046 nodes representing 1,078,364 buildings, including e.g. 12,928,580 polygons and 5,864,792 boundary surfaces are created after the mapping process of two 3D city models of Berlin is complete. The graph database allocates 126 GB of disk storage in total (excluding test and indexed data).

8.3.2 Multi-threading Performance The multi-threading implementation of the mapping and matching process employs the well-known producer-consumer design pattern. The differences in performance between some combinations of numbers of producers and consumers are shown in Figure 12. The results show that the matching process generally benefits more from the number of concurrent threads than mapping. However, diminishing returns are observed where the total number of producers and consumers exceeds that of the testing system's physical CPU cores.

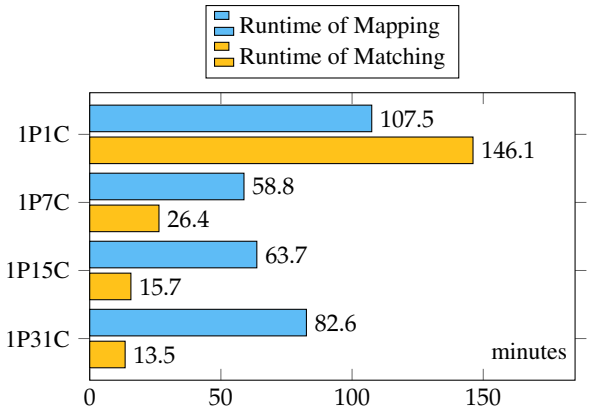


Figure 12. Differences in multi-threading performance. P and C denote the number of producers and consumers respectively.

8.3.3 Indexing Performance while Resolving XLinks The impact of storing indices on disk (using Neo4j legacy indices) and in memory (using self-developed internal hash maps) on performance is shown in Figure 13. Indexing using internal hash maps results in a much better overall performance but requires a large amount of memory. On the other hand, Neo4j indices stored on disk do not require as much main memory but are significantly slower due to expensive disk read and write operations.

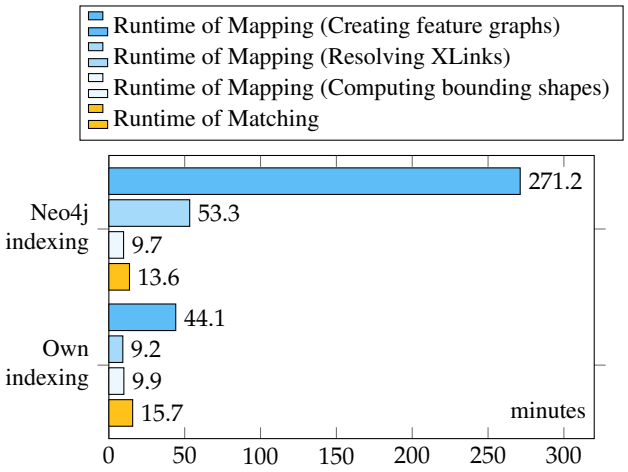


Figure 13. The performance of storing indices on disk (using Neo4j built-in indices) and in memory (using internal hash maps).

9. CONCLUSION AND FUTURE WORK

Overall, the mapping process developed in this research is capable of handling arbitrarily large-sized CityGML documents given a reasonable amount of memory and storage allocation. It facilitates the seamless interaction between unmarshalling CityGML elements to Java objects with the help of citygml4j and mapping Java objects to graph entities using Neo4j's Java Core API.

The matching process can disambiguate the common syntactic ambiguities existing in GML between XLink and in-line object declarations. All changes made to the old city model in Section 8 were identified correctly. In addition, although LOD2 data were used in the test scenarios, changes in other LODs can also be detected. Moreover, geometric objects such as points, line segments, polygons, surfaces, etc. can be matched correctly even with altered identifiers. Furthermore, buildings can be organized in a grid layout or an R-tree based on their spatial allocations. These strategies offer a noticeable boost in overall performance.

Found deviations are attached to their respective sources in the graph database and can be transformed to WFS requests complying with the official OGC standards. In case of complex XML properties, such as CityGML generic attributes and external references, although the update procedures can be formally represented by graphs, the ordinary WFS is not expressive enough in such scenarios. Thus, vendor-specific extensions allowed by the WFS standard, such as defined by the virtualcityWFS, can be employed.

Some improvements and extensions are possible in the near future. For instance, momentarily, only the modules `Building` and `Appearance` are implemented. Other CityGML modules, like `CityFurniture`, `Transportation`, `Bridge`, `Tunnel`, etc. can be included in the future. Moreover, it is previously assumed that both CityGML input documents are provided in the same spatial reference system, which is not always the case in practice. Therefore, one future task is to integrate the transformation between different spatial reference systems in the implementation. In addition, more thorough tests are required to evaluate application outputs against all different types of geometrical deviations. Finally, the methods and algorithms proposed in this research can be extended and applied to enable a version control system for collaborative work in modelling and storing digital 3D city models in the future (Chaturvedi et al., 2015).

References

[1] M. Bakillah, Y. Bédard, M. A. Mostafavi, and J. Brodeur. "SIM-NET: A View-Based Semantic Similarity Model for Ad Hoc Networks of Geospatial Databases." In: *T. GIS* 13.5-6 (2009).

[2] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Springer, 2008.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0*. Fifth edition. W3C. Nov. 2008.

[4] K. Chaturvedi, C. S. Smyth, G. Gesquière, T. Kutzner, and T. H. Kolbe. "Managing versions and history within semantic 3D city models for the next generation of CityGML." In: *Selected papers from the 3D GeoInfo 2015 Conference*. Ed. by A. A. Rahman. Springer, 2015.

[5] S. Cox, P. Daisey, R. Lake, C. Portele, and A. Whiteside. *OpenGIS Geography Markup Language (GML) Implementation Specification*. Specification OGC 03-105r1. Version 3.1.1. Open Geospatial Consortium, 2004.

[6] S. DeRose, E. Maler, D. Orchard, and N. Walsh. *XML Linking Language (XLink) Version 1.1*. W3C. May 2010.

[7] M. J. Egenhofer and R. D. Franzosa. "Point-set topological spatial relations." In: *International Journal of Geographical Information Systems* 5.2 (1991).

[8] M. J. Egenhofer and J. Herring. *Categorizing binary topological relations between regions, lines, and points in geographic databases*. Technical report. Department of Surveying Engineering, University of Maine, 1991.

[9] G. Gröger. *Modeling Guide for 3D Objects - Part 1: Basics (Rules for Validating GML Geometries in CityGML)*. `http://en.wiki.quality.sig3d.org/index.php/Modeling`. Version 0.6.0. Accessed: 2017-03-01. SIG3D - Special Interest Group 3D, Dec. 15, 2010.

[10] G. Gröger, T. H. Kolbe, C. Nagel, and K.-H. Häfele. *OpenGIS(R) City Geography Markup Language (CityGML) Encoding Standard*. Version: 2.0.0. OGC. Apr. 2012.

[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, 1984.

[12] J. W. Hunt and M. D. McIlroy. *An Algorithm for Differential File Comparison*. Computing Science Technical Report. Bell Laboratories, June 1976.

[13] C. Nagel. *citygml4j - The Open Source Java API for CityGML*. `https://github.com/citygml4j/citygml4j`. Version 2.4.3. Accessed: 2017-03-01. 2017.

[14] G. Navratil, R. Bulbul, and A. U. Frank. "Maintainable 3D Models of Cities." In: *Proceedings of the 15 International Conference on Urban Planning, Regional Development and Information Society*. Real CORP, 2010.

[15] S. H. Nguyen. "Spatio-semantic Comparison of 3D City Models in CityGML using a Graph Database." Master's thesis. Department of Informatics, Technical University of Munich, May 2017.

[16] A. Olteanu, S. Mustière, and A. Ruas. "Matching imperfect spatial data." In: *Caetano, M., Painho, M.(Es.), Proceedings of 7th International Symposium on Spatial Accuracy Assessment in Natural Resources and Environmental Sciences. Lisbon*. 2006.

[17] Oracle Corporation. *Java Architecture for XML Binding (JAXB)*. `http://docs.oracle.com/javase/tutorial/jaxb/intro/arch.html`. Tutorial. Accessed: 2017-03-01. 2015.

[18] R. Redweik and T. Becker. "Change Detection in CityGML Documents." In: *3D Geoinformation Science: The Selected Papers of the 3D GeoInfo 2014*. Springer, 2015.

[19] H. Tveite. "An accuracy assessment method for geographical line data sets based on buffering." In: *International journal of geographical information science* 13.1 (1999).

[20] P. A. Vretanos. *OGC® Web Feature Service 2.0 Interface Standard*. OGC® Standard 09-025r2. Version 2.0.2. Open Geospatial Consortium, July 2014.

[21] Y. Wang, D. J. DeWitt, and J. Y. Cai. "X-Diff: an effective change detection algorithm for XML documents." In: *Data Engineering, 2003. Proceedings. 19th International Conference*. Mar. 2003.