Spatial algorithms in Neo4j
Internship Report

S.H.N. van der Linde
s.h.n.v.d.linde@student.tue.nl

July 5, 2019

Supervisors

Technical University of Eindhoven
Wouter Meulemans
w.meulemans@tue.nl

Neo4j
Craig Taverner
craig.taverner@neo4j.com

Abstract

Currently, the Neo4j graph database supports only one spatial geometry and a single algorithm out-of-the-box. This project discusses and implements spatial data structures and algorithms for Neo4j as a library. The library adds support for certain geometries, such as polylines and polygons, instead of only points to Neo4j. Moreover, it also implements various algorithms in both the Cartesian as well as WGS84 coordinate reference systems. The library also supports OpenStreetMap data imported into Neo4j. Lastly, a comparison between the different data structures as well as the algorithms for the different coordinate systems is made based on benchmarking results.

1 Introduction

Neo4j is one of the leaders in the world of graph databases. It differs to traditional relational databases in that it does not use tables, but nodes and relationships to store data. Its use cases range from fraud detection to recommendation engines. However, its spatial support is in its infancy. Natively, it can only store points, either solely or in an array. Furthermore, it contains only one spatial algorithm, computing distances between 2 points, out-of-the-box.

In this report, we discuss methods to improve the spatial support of Neo4j in the form of a library. This library can be loaded by the user to increase the

spatial functionalities of Neo4j. The user can then use their data on the mentioned algorithms, which support both Cartesian as well as WGS84 coordinate reference systems. We take the OpenStreetMap model as a use case and show how to modify the data to support our data structures and algorithms.

2 OpenStreetMap

OpenStreetMap [OpenStreetMap contributors, 2017] is a map database filled by volunteers. It stores both natural occurring structures as wall as man-made constructions. The data from OpenStreetMap can, for example, be laid over satellite photos to add a layer of information. The focus of this research is mainly on polygonal shapes. Hence, the methods can be applied on most polygons stored in OpenStreetMap, especially boundaries, both political as well as natural occurring ones.

In OpenStreetMap, one of the main building blocks are ways, which are strings of consecutive location points. Note that a way does not necessarily mean a road or a highway, it can represent anything which consists of multiple consecutive points, such as a river. These ways can be combined to create bigger shapes and represent larger structures using a relation in OpenStreetMap. For example, a river can consist of multiple ways and to represent the complete river, an OpenStreetMap relation is created and refers to all the ways which are part of the river. Ways and relations can also close themselves to create a closed area, such as a lake or a country's border.

2.1 OpenStreetMap in Neo4j

There exists a tool to import OpenStreetMap data into the Neo4j database [Taverner, 2019]. This tool creates a graph structure based on the XML data of the input. Because the essence of the input is based on relations and references, there exists a logical conversion to a graph. Each element, such as a relation or way, is represented as a node in the graph. When an element refers to smaller elements, relationships are formed. For example, a `Relation` node has relations to all of its `Way` nodes and so on.

On one place the graph differs from the input XML. In the original data, ways consist of nodes with a location. However, in the graph representation created by the tool, ways consist of `WayNodes` which do not contain any type of location data. They in turn have a relation to a `Node` vertex which stores the actual location data[1]. The advantage of this storage method is that a way node is

[1]Please note the difference between a node in Neo4j and a `Node` vertex of the OSM graph. The former is one of the main building blocks of the database, while the latter is part of the OSM importer hierarchy. Every vertex of this hierarchy is stored as a Neo4j node in the database.

specific to that way and can therefore easily be linked to its siblings, while the actual location data can be shared between multiple ways without interference. A visualization of the resulting structure in Neo4j is shown in Figure 1.

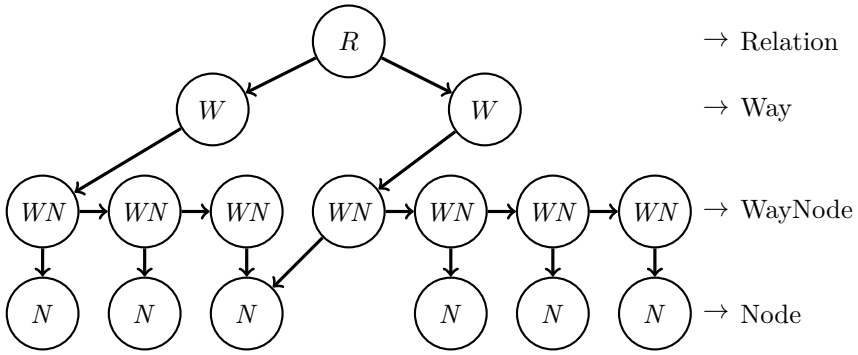


Figure 1: Example of the graph representation of the OpenStreetMap data.

3 Data structures

Geometries can be stored in multiple ways in the Neo4j database. In this section, we describe the various data structures which are used in the report.

3.1 Points

In geometry, a point is the most primitive data type. It does not have dimensions of itself and is used to build more complex geometries. Because this report considers only two dimensions, the point data type requires at least two components to describe a location in the space. Since Neo4j 3.4, the database natively supports a Point data type. This type also supports multiple coordinate reference systems of which the Cartesian and WGS84 variants are of interest for this report.

3.2 Simple polygons

A polygon is a geometry consisting of multiple consecutive line segments which all together define a region. Simple variants are polygons where no line segment crosses over another line segment of the polygon. If such a crossing would exist, the polygon is called complex. However, in this report, we do not consider the complex variants.

3.2.1 Array of points

A simple polygon can be stored as an array of Neo4j Point types in the property of a node. In this array, the order matters and two successive points in the array represent an edge of the polygon. This also holds for the first and last element of the array if these two points are not equal. We store this array of points as a property of a node in the graph database. This node will also contain metadata on which we can later search.

3.2.2 Sub-graph representation

Another way to store polygons is using the defining property of the Neo4j database, its graph structure. Simple polygons can be viewed as a planar graph with degree 2. Therefore, it is easy to model a polygon in the graph. Each point of the polygon is a separate node in the graph and when two points share an edge, a relationship is created between the two corresponding nodes. We do this between every point. Furthermore, we store one extra node to index the polygon on which also contains the metadata of the polygon and points to the first node of the actual polygon.

However, the nodes of an actual graph are not fixed to a location. You could move the nodes around and still have the same graph. This is not the case for a polygon, where moving points results in a different polygon. To remedy this, we give each node of our created graph a property which stores the location of the corresponding point.

The OSM hierarchy described in Figure 1 can be modified into this representation. To do this, we connect the last `WayNode` of each way to the first `WayNode` of the next relation with a special relationship identifying the polygon. To get the actual polygon, an algorithm has to traverse the relations between the `WayNodes` and collect the `Node` which contains the location data. When a way is shared between polygons, the algorithm knows which way to jump next to because of the previously mentioned special relationship, which uniquely identifies the coresponding polygon.

This method will take up more space than the property-based representation, but it will be easier to modify points of a polygon. This is especially true when points are shared between geometries.

3.3 Polylines

A polyline is a collection of successive line segments but which do not define a region. In this report, we consider only polylines that do not cross itself. Because of the many similarities between polygons and polylines, we can use the same two data structures used for polygons to store polylines. However,

we do have to specify which geometry is represented by the data structure. To do this, we use different key values for the property-based representation, respectively, `polygon` and `polyline`, and different meta node labels for the sub-graph representation.

3.4 Multipolygons

Multipolygons are a special type of polygons. Not only can they consist of multiple polygons, they can also have holes in them, which are also polygons. We call the polygons which are drawn *shells* and polygons which remove parts of a shell *holes*. In this report, we do require that none of the rings of a multipolygon may touch or intersect. A hole must be fully contained by a shell and an interior shell must be completely surrounded by a hole.

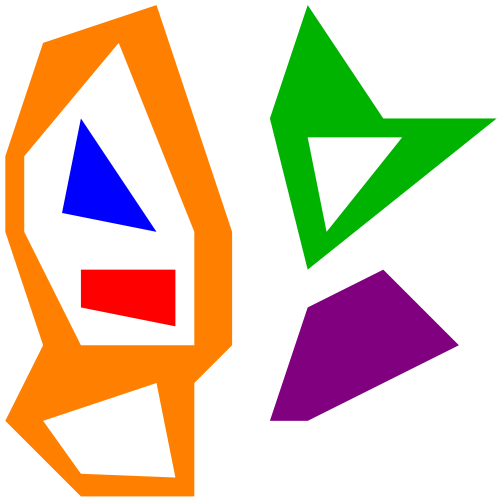


Figure 2: An example of a multipolygon consisting of 5 shells and 3 holes.

3.4.1 Sub-graph representation

You can view such a multipolygon as a tree-like structure, where a parent-child relation is created if a hole is contained in a shell, or vice versa, where the container is the parent and the contained is the child. If you apply this logic to multiple separate shells, you get a forest of shells, which each can have multiple holes. In their turn, the holes can have smaller shells in them.

Because a tree is in essence a graph, it is easy to convert this concept to the Neo4j database. We start by creating a main node to keep the forest connected. In the case of an OSM relation or way, we take that node as the root. The root can be seen as an (unbounded) hole in which the shells are placed. Next, we

create trees for every shell which is not contained in any other shell. Then, we create a children for every hole in that particular shell. We continue by again creating children for those children for every shell in these shells. This process is repeated until we have reached the lowest level. Note that the levels of the tree alternate between shells and holes. An example of such a tree is given in Figure 3.

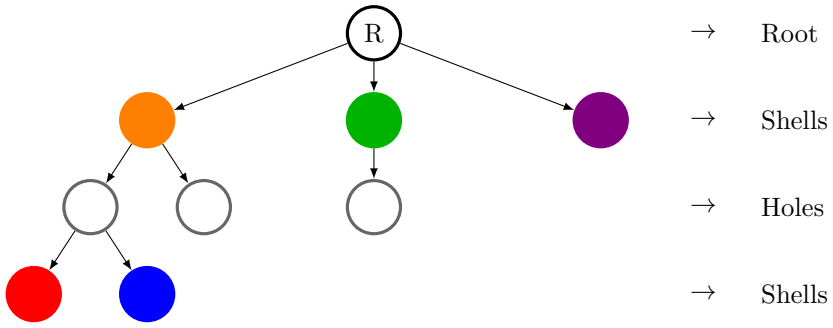


Figure 3: The sub-graph representation of the multipolygon in Figure 2.

The structure is built one polygon at a time. We start by creating the root node. The root will not refer to any specific polygon, but rather it groups all the highest level shells together. For every polygon, hole or shell, we insert in the root. The root than tries to insert the polygon into any of its children. It succeeds if the polygon is indeed contained by the child. However, if this is not the case, either because the polygon contains the child or they are both completely disjoint, then insertion fails. If a polygon cannot be inserted into any of the children of the root, the polygon will be seen as the highest level shell and is inserted as a child of the root node. The insertion logic of a polygon into a non-root node uses the logic as described in Algorithm 1.

For every polygon b one wants to insert, the algorithm is called in combination with every child a of the root. If for any given polygon, all the calls return FALSE, the polygon is a direct child of the root and is added as such. The algorithm to check containment are described in Sections 4.4 and 5.7.

Note that the test on line 4 will only return true when both a and b are the highest possible shells, which can only happen when the algorithm has not yet recursed. Furthermore, if the test on line 8 is true for at least one child, then the test on line 11 will always return false. A polygon can not be contained by one child while containing a different child, because the children are disjoint.

We have now only described the structure of the multipolygon, but not yet the actual points. Because a shell or hole is in essence a simple polygon, we can choose from the above mentioned implementations to describe the actual shape of the polygon. For example, we can give each node in the forest an array containing the points of its polygon or we can point the node to a sub-

Algorithm 1 PolygonInsertion

Input:
a Polygon to insert into
b Polygon to be inserted
Output:
TRUE iff a and b are not disjoint, otherwise FALSE
Ensures:
If a and b are not disjoint, one of the two is inserted in the other

```
1: if b contains a then                          ▷ b contains a → swap places
2:     Make a a child of b and make b a child of the previous parent of a
3:     return True
4: else if a does not contain b then    ▷ The two polygons are disjoint → fail
5:     return False
6: end if
7: for child in a.children do
8:     if child contains b then      ▷ Child contains b → recurse on child and b
9:         PolygonInsertion(child, b)
10:        return True
11:    else if b contains child then   ▷ b contains the child → move child to b
12:        Make child a child of b instead of a child of a
13:    end if
14: end for
15: Make b a child of a
16: return True
```

graph representation of the polygon, such as one of the `WayNodes` in the OSM hierarchy.

4 Cartesian algorithms

4.1 Linear referencing

Linear referencing can be used to reference points along a path. Instead of storing the 2 dimensions of the point, one can store just the distance and direction from the start point of the path and follow the path to find the point. This is for example used to mark points along a road or pipeline. Furthermore, it can be used to partition the path in segments. In case of a road, the changes in speed limit or the different surface materials can be saved.

Given a path, a direction and a goal distance distance d and returning the final position. This can be applied on a line segment, but also on a polyline or polygon. For every line segment $\overline{ab}$ we encounter along the path, we check its

length l. If $l > d$, then we update $d := d - l$ and go to the next segment. But if $l \leq d$, we compute the fraction $f = \frac{l}{d}$. Then, we interpolate to find the final point using the following formula:

$$x_i = x_a(1 - f) + x_b f$$
$$y_i = y_a(1 - f) + y_b f$$

4.2 Area of a polygon

The area of a polygon can be computed using the shoelace formula. The formula is as follows:

$$A = \frac{1}{2}|\sum_{i=1}^{n} x_i y_{i+1} - x_{i+1} y_i| = \frac{1}{2}|\sum_{i=1}^{n} x_i(y_{i+1} - y_{i-1})|$$

where the indices of x and y wrap around when they exceed the bounds, e.g. $x_{n+1} = x_1$.

For each edge $\overline{ab}$ of the polygon, we compute the area of the triangle consisting of $\overline{ab}$ and the origin O. We do this by taking the cross product and dividing it by 2, because the cross product gives us the parallelogram instead of the triangle. If we go along the polygon in a counterclockwise fashion and the edge is going from left to right from the perspective of O, the computed area is positive, and if it is going from right to left, the area will be negative. When we go around the polygon, the triangles will overlap and the positive and negative areas between the polygon and O will be negated. In the end, only the polygon will remain and we take the absolute value of the remaining value to get the area of the polygon. A visualization of this algorithm is shown in Figure 4.

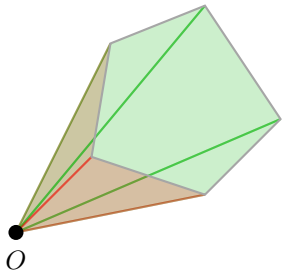


Figure 4: Visualization of the shoelace algorithm where we traverse around the polygon clockwise. Positive areas are colored green while negative areas are colored red.

4.3 Intersections

In this section we discuss two algorithms to compute the intersections of two geometries. We consider especially geometries made from multiple line segments such as polygons and polylines.

4.3.1 Naïve algorithm

The most simple solution to find out where two geometries intersect, is to check for intersections for every pair of edges of the two geometries. This simple method is quick to implement, but it is very inefficient. Its theoretical running time is $\mathcal{O}(n^2)$, where n is the total number of edges of the two geometries. This algorithm has such a high running time because it does not use any of the information, such as the location of the edges, to find which edges to compare.

4.3.2 Monotone chain sweep line algorithm

A smarter method to find the intersection points of two geometries is using a sweep line algorithm. Such an algorithm can be seen as a straight line which "sweeps" over the geometries and comparing only edges that currently are "sweeped" over. This method uses the fact that edges can intersect only if the intervals for their x-coordinates overlap. It reduces the number of intersection checks made by the algorithm.

The most well known sweep line algorithm is the Bentley–Ottmann algorithm [Bentley and Ottmann, 1979]. This algorithm runs in $\mathcal{O}((n + k) \log n)$, where n is the total number of edges of the two geometries and k is the number of intersections between those edges, which means that the algorithms is output-sensitive. When k is not too big, this algorithm performs better than our naive solution. This holds when $k = o(\frac{n^2}{\log n})$.

However, the algorithm does have a downside. It assumes that the input has general position, where no two input points or intersections have the same x-coordinate and no input point intersect a line segment. In reality, no data set will adhere to this. Hence, the algorithm is not the optimal candidate to implement.

An algorithm based on the Bentley-Ottmann algorithm is created by Sang Park, Hayong Shin, and Byoung Choi [Park et al., 2001]. It also uses a sweep line to check for intersections, but it is extended to use monotone chains. Monotone chains are strings of successive edges which are monotone with regards to the sweep direction. Every later point along the chain has a higher x-coordinate than all the previous points. The idea behind using these monotone chains is that no edge in the chain can intersect with another edge in the chain, because the x-coordinate always increases further along the chain and it never decreases to turn around and intersect a previous edge. However, vertical line segments

are not supported because the x-coordinate does not increase between the two points. To circumvent this, we can rotate all the points a fraction around the origin such that the vertical line segments are no longer vertical anymore and the x-coordinates differ between the two points. Please note that the found intersection points do have to be rotated back to get the actual intersections locations. An example of a x-monotone partitioned polygon is shown in Figure 5.

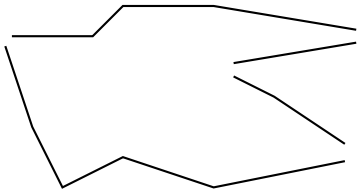


Figure 5: A polygon partitioned in x-monotone chains

The algorithm uses three lists: A, S, and O. A is known as the active chain list and contains all chains which we still have not fully covered in the algorithm. S is the sweeping chain list which contains all chains which are currently intersected by the sweep line. It is also known as the status of the sweep line. Lastly, O is the output list containing all the intersections between chains. Furthermore, every chain will keep a pointer to the vertex of its chain which is right in front of the sweep line. We call this vertex the front vertex. The chains in A are ordered according to the x-values of their front vertex such that the left-most chain will be first in A. In contrast, the chains in S are ordered by their y-value at the current sweep line position.

The line sweeps from negative x to positive x and keeps track of the currently intersected chains using its status S. Every vertex is considered an event. When the sweep line crosses a vertex, an event gets added to S. The situation can only change at an event and not in between two events. This means that the sweep line only has to jump from vertex to vertex and can ignore intermediate line segments. We have four different event types based on their corresponding vertex's location in its monotone chain. Let f be the currently left-most vertex that is not yet covered by the sweep line of monotone chain c.

Left-most vertex

These vertices represent chains we have not yet covered. We add it to our status S. Next, we check if c intersects with the chains just before or after it in S. In other words, we check if c intersects with the chains just above or below c.

Right-most vertex

When f is not the left-most but rather the right-most vertex in c, we have finished considering c. Therefore, we check if the chain before and after c in S intersect with each other. Furthermore, we remove c from both A as well as S because the sweep line has passed the whole chain c.

Internal vertex

If the vertex f is just an internal vertex of the chain, i.e. neither the left-most or right-most vertex, we just check if c intersects with the chain just above or just below it.

When we check for an intersection between two chains, c and c', we do not have to pairwise consider the line segments of the two chains. Because we know the location of the sweep line, we have to consider only the two line segments currently being intersected by the sweep line. If there exists an intersection of the two line segments, we do not yet report it. Rather, we add the point as a vertex to both the chains and we give it a new type: intersection vertex.

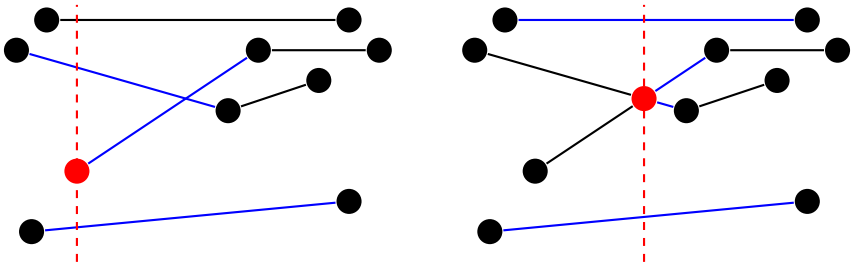
Intersection vertex

When the algorithm encounters an intersection vertex as the front vertex f, for example of c, it is not handled like a normal internal vertex. We first move the pointer of the intersecting chain c' to the next vertex. We know that c and c' intersect at this location, so they have to switch places in S, where the order is based on the y-value at the sweep line. Next, it looks for intersections between c and c' and the chain just above and below them. Finally, we add f to the output O.

When there are no chains in A left, the sweep line has passed all the chains and we have considered all of them. Lastly, we report the points in O as intersections.

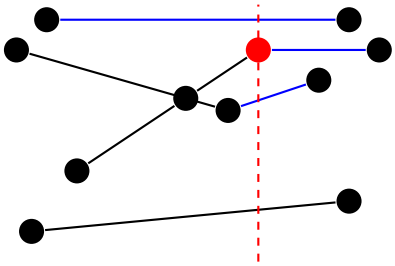
The running time of this algorithm is $\mathcal{O}((n + k)m \log m)$, where n and k are the same as above and m is the number of monotone chains. When the number of chains is just a fraction of the number of points or the maximum number of chains at the same x-coordinate is low, this algorithm will perform better than Bentley-Ottmann.

The algorithm also resolves the problems regarding general position of Bentley-Ottmann. The algorithm supports intersections of input points on line segments out-of-the-box. The problem of line segments sharing an end point is partially supported by the algorithm. The algorithm will not have any problems when two line segments share an end point, but it does not report an intersection. In most cases, this behaviour is fine, such as with a chain, where each edge is successive. In this situation, no intersection should be reported. However, when two geometries meet in a single point, the intersection should be reported. We have modified the algorithm to allow the report of such intersections. When a shared point is encountered, the modified algorithm will check if the two chains belong to two different polygons. Only if this is the case, the algorithm will add the point to O, but it will not create an intersection vertex, because the vertex already exists in both chains.

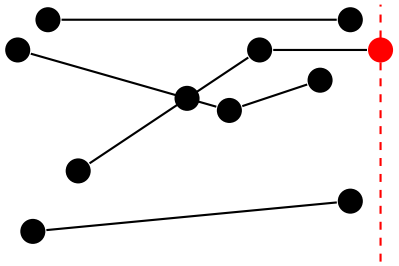


(a) A left-most vertex

(b) An intersection vertex



(c) An internal vertex



(d) A right-most vertex

Figure 6: Processing the different vertex types. Red: the currently considered vertex and sweep line. Blue: the line segments which will be checked for intersections.

4.4 Containment

To find out if a point p is contained in a polygon, we use the crossing number algorithm. This algorithm creates a horizontal ray from p to positive infinity. Then, we count the number of intersections (or crossings) we encounter with the polygon. If the number of crossings is odd, point p is inside the polygon, but if the number is even, p is not contained by the polygon. An example of this algorithm can be found at Figure 7.

4.5 Convex hull

A convex hull of a set of points is a polygon containing all the points of the input while also being convex. This can for example be used to quickly check if a point lies maybe in the original polygon itself. The convex hull is often a smaller than the original polygon with regards to points, therefore it is quicker to check if a point lies inside the convex hull than in the original polygon. Only if the point is inside the convex hull, you check if the point is also in the original polygon. If it is not inside the convex hull, the point will never be inside the original polygon.

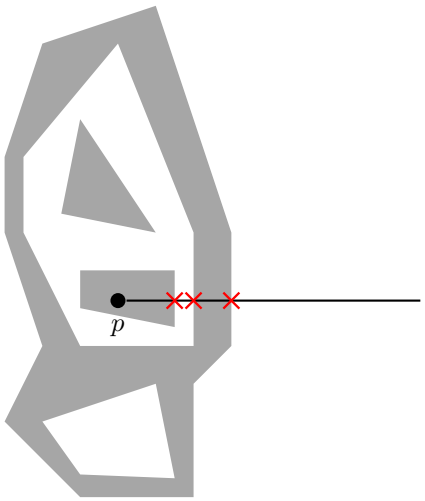


Figure 7: Example of the crossing number algorithm. In this scenario, the crossing number is 3, which means that point p is inside the polygon.

A well known algorithm for finding the convex hull of a polygon, or a set of points in general, is the Graham scan. This algorithm is simple yet quite fast. It first picks an extreme point by selecting the point with the lowest y-value. If multiple points satisfy this condition, we pick from this set the point with the lowest x-value. This is an extreme point and, therefore, we know that this point is always present in the final convex hull. We call this point p.

The next step is to sort the remaining points in increasing order based on their polar angle with p as the origin. However, we do not have to compute the actual polar angle to sort the points. We can use a different function, as long as this function is monotonic in the interval $[0, \pi]$, such as cosine. The interval is fixed, because we know that none of the points can have an angle greater than π. This would place a point below p, which is impossible, because p has the lowest y-value of all points. We use the dot product to compute the cosine.

After sorting the points, we look at every point of the list in turn. For each point c not yet in the convex hull, we check if going from the currently last two points, a and b, of the convex hull result in a left or right turn. In other words, we want to find out the sign of the z-coordinate of the cross product between $\overrightarrow{ab}$ and $\overrightarrow{ac}$. If this sign is positive, then the path abc makes a left turn. If it is negative, it makes a right turn. When the value of the z-coordinate is equal to 0, the three points are collinear. When we make a left turn, we add c to our convex hull. If the three points are collinear and we make a straight line, we can remove b from the convex hull. When we make a right turn, we know that point b does not belong to the convex hull. We can therefore remove b from the convex hull, but we also have to repeat the current step again with the new last

two points of the convex hull to check if there are new right turns. We keep removing the middle point until a left turn is encountered. We then know that these three points are on the convex hull up until now and we continue with the next point of the list.

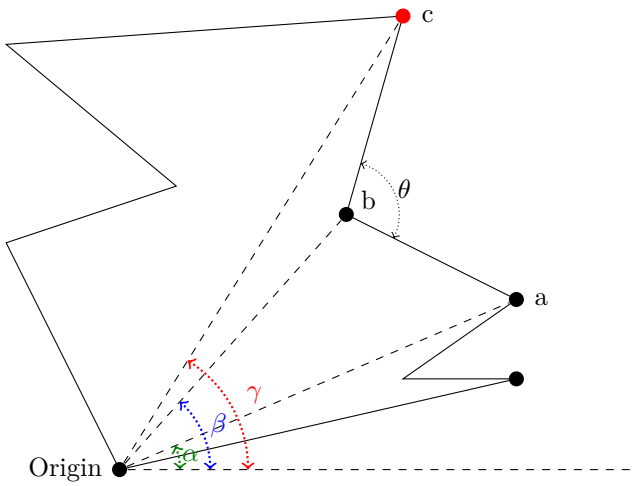


Figure 8: A step in the Graham scan. The black nodes are currently on the stack and the red node is being considered.

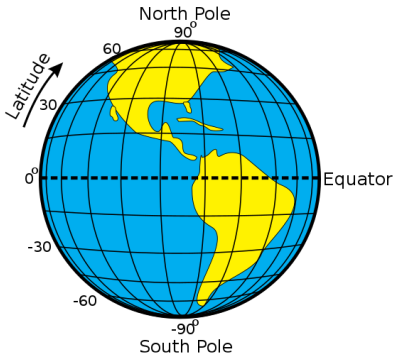
In Figure 8, a step of the Graham scan is visualized. Because the angle θ makes a right-hand turn, point b will be remove from the stack.

When all points are considered, we are back at point p. At this time, we have checked the turns of the convex hull. Because all of them are leftward, the hull is convex.

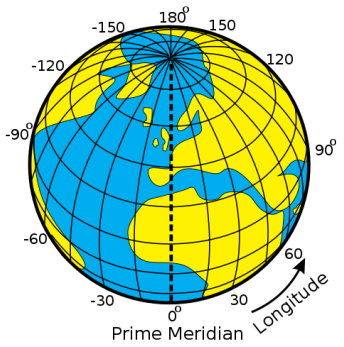
This algorithm runs in $\mathcal{O}(n \log n)$ time. The last step takes only linear time, but the total complexity is dominated by the sorting step.

5 Geodetic algorithms

The algorithms described above all operate on Cartesian coordinate system. They use an infinite 2D plane to work on and positions are described with (x, y)-coordinates. However, this does not translate to geographic locations, because the earth is spherical rather than planar. Therefore, different coordinate systems are developed to describe locations on the globe. One of the most well-known systems is the World Geodetic System, of which WGS84 is the latest revision. This system is for example used in GPS and it uses latitudes and longitudes to describe a point on the earth. In this report, we approximate the



(a) The parallels describing the latitude.



(b) The meridians describing the longitude.

Figure 9

earth as a sphere. While in reality, the earth is more similar to a spheroid where the radius on the plane of the equator is greater than the radius on the plane connecting the North and South Pole.

Latitude defines the position of the point with regards to the poles. When the point is exactly between the two poles, the latitude is 0°. When the point moves towards the North Pole, the latitude increases until the point reaches the North Pole, at which the latitude has a value of 90°. When going to the South Pole, the latitude decreases until at most −90°. Longitude describes the distance from the Prime Meridian, which is a straight line from the North Pole to the South Pole. In WGS84, the Prime Meridian passes Greenwich around 100 meters east, instead of actually crossing Greenwich. At the Prime Meridian, the longitudes is 0°. The longitude increases when a point moves eastwards from the Prime Meridian and it decreases when a point moves westwards until it is exactly on the other side of the globe from the Prime Meridian at which the longitude is $180°/-180°$, this is the International Date Line.

In Figure 9, one can see that the cells created by the parallels and meridians are not equal of size. This also contributes to the fact that most Cartesian algorithms will not work on this coordinate system. Even though the cells differ in size, the range of degrees they cover with regards to the latitude and longitude is equal in size. If we increase the longitude by one degree, the change in distance in actual meters will differ.

5.1 *n*-vectors

When one wants to do computations with this coordinate system, often a lot of spherical trigonometry is required. However, this may lead to several issues, such

as the singularities at the Poles and the discontinuity at the International Date Line. [Gade, 2010] proposes a new and different method representation which reduces the amount of trigonometry used in the calculations. The n-vector is a unit vector which is perpendicular to the surface of the sphere at the location of the point it represents.

$$\vec{\boldsymbol{v}} = \begin{bmatrix} \cos\varphi \cdot \cos\lambda \\ \cos\varphi \cdot \sin\lambda \\ \sin\varphi \end{bmatrix}$$

Conversely, one can extract the latitude and longitude from the n-vector using the following structure:

$$\varphi = \arctan2(v_3, \sqrt{v_1^2 + v_2^2})$$
$$\lambda = \arctan2(v_2, v_1)$$

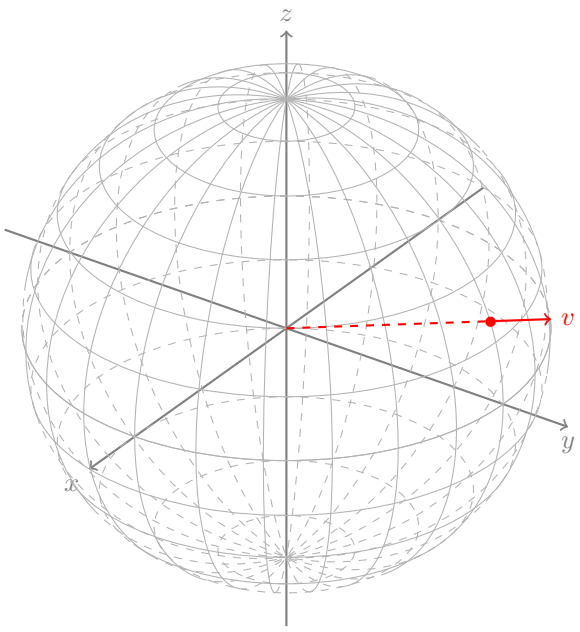


Figure 10: n-vector representation of the point 20° N, 90° E.

5.2 Great circle

A great circle is defined as the intersection of a plane through the Earth's center and the surface of the Earth. For example, the Prime Meridian and

the International Date Line together form a great circle. It also describe the shortest path between any two points on the circle.

Using n-vectors, the great circle is represented as the normal on the plane and where the direction of the normal is the direction of travel along the circle. To compute the normal given two points on the great circle, we use the cross product, $\vec{\boldsymbol{c}} = \vec{\boldsymbol{a}} \times \vec{\boldsymbol{b}}$, or $\vec{\boldsymbol{c}} = \vec{\boldsymbol{b}} \times \vec{\boldsymbol{a}}$ for the other direction. Figure 11 shows an example.

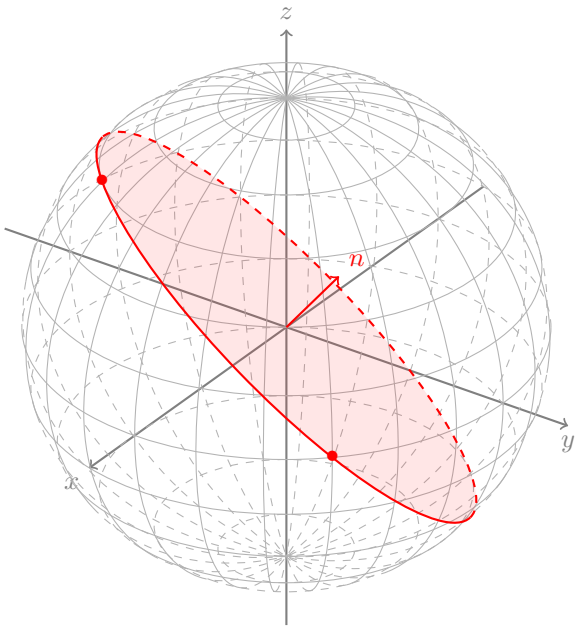


Figure 11: Great circle and its normal formed by two points.

5.3 Distance between two points

Using the n-vector model, the distance between two points, a and b is the same as the product of the angle between their vectors, $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{b}}$, and the radius r of the sphere, in the case of the earth $r \approx 6371 * 10^3$ meters. This gives us the following formula:

$$d(a, b) = r \cdot \arctan2(|\vec{\boldsymbol{a}} \times \vec{\boldsymbol{b}}|, \vec{\boldsymbol{a}}' \cdot \vec{\boldsymbol{b}})$$

5.4 Linear referencing

The basics of linear referencing in the n-vector model is equal to the Cartesian variant. One traverses over the line segments and reduce the goal distance until

one encounters a line segment which length is greater than the remaining goal distance. We then interpolate the remaining distance on this line segment to find our final point. In our spherical case, only the interpolation step differs. We first compute the angular distance δ between the two end points, u and v, of the line segment. Next, we interpolate this angular distance based on the fraction of the goal distance and the length of the line segment, which gives us the interpolated angular distance δ_i. We also find the direction vector $\vec{\boldsymbol{d}} = unit(\vec{\boldsymbol{u}} \times \vec{\boldsymbol{v}}) \times \vec{\boldsymbol{u}}$. Lastly, we apply the following formula to find the n-vector of the interpolated point: $\vec{\boldsymbol{p}} = \vec{\boldsymbol{u}} \cdot \cos(\delta_i) + \vec{\boldsymbol{d}} \cdot \sin(\delta_i)$. This last step moves along the point along the surface of the sphere from u towards v. Now, we have point p which is exactly d distance from u towards v.

5.5 Area of a polygon

Suppose we have a spherical triangle T with interior angles a, b, c on a unit sphere. One can compute the area of this triangle using Girard's theorem. This theorem uses lunes which are made up partly from the given triangle and sums these lunes. A lune is the area between two half great circles. This sum gives us the area of the sphere plus 4 times the area of the triangle, because it was used in multiple lunes. From there, one can compute the area A of the single triangle. An interactive visualization of this proof can be found at [Vanderbei, 2013]. This theorem is captured in the formula:

$$a + b + c = \pi + A$$

However, we do need to compute the area A of an arbitrary polygon. If we would partition the spherical polygon, we get $n - 2$ triangles. If we apply Girard's theorem to all of them, we get the following equation [Todhunter, 1871]:

$$\sum_{i=1}^{n-2}(a_i + b_i + c_i) = (n-2)\pi \sum_{i=1}^{n-2} A_i$$

The left-hand side of the equation is the sum of all angles of the triangles and the right-hand side is the sum of the triangles' areas. The latter is the same as the area of the whole polygon. The sum of interior angles of these triangles is equal to the sum of interior angles of the polygon. When we rewrite the equation, we get the following formula for the area of the polygon based on the interior angles α of the polygon:

$$A = (\sum_{i=1}^{n} \alpha_i) - (n-2)\pi$$

Hence, one only needs to sum its interior angles and subtract $(n-2)\pi$, to get the area of the polygon. Note that all of these computations are on the unit sphere. To get the final area of the triangle or polygon, we only have to do

5.6 Intersection

5.6.1 Intersection between two arcs

An arc is a path between two points along the great circle. However, such a path can go either side of the sphere. To remedy this, we define an arc as the shortest path between two points and its length is less than π on the unit sphere. This means it will never go the long way round. If one wants to go the long way round the sphere, an extra point has to be added to cut the arc in two parts.

To compute the intersection of two arcs on the sphere, $\overline{ab}$ and $\overline{cd}$, we first compute the great circle for each of them. This gives us two vectors $\vec{\boldsymbol{c_1}}$ and $\vec{\boldsymbol{c_2}}$. Then, we find the intersections between these two great circles, which is as easy as applying the cross product on the two normals. Because the great circles intersect twice, we get two intersection points based on the order of the cross products. We now have two candidate intersection points: $\vec{\boldsymbol{i_1}} = \vec{\boldsymbol{c_1}} \times \vec{\boldsymbol{c_2}}$ and $\vec{\boldsymbol{i_2}} = \vec{\boldsymbol{c_2}} \times \vec{\boldsymbol{c_1}}$. Lastly, we have to check whether any of the two candidates are actually on the two arcs. Only when a candidate lies on both arcs, did we find the intersection between the two arcs. In any other case, the two arcs do not intersect. Because the length of the arc is less than π, it is impossible for both i_1 and i_2 to lie on the arc at the same time.

5.6.2 Intersection between poly-geometries

Just as in the Cartesian coordinate system, we want to find an efficient method to compute the intersections of larger geometries, such as polygons and polylines. In particular, it would be preferable if we could use the monotonic chain sweep line algorithm. However, that is not possible in all cases. The algorithm uses the x-monotonic property of the chains to efficiently look for intersections. This property does not translate to spherical coordinate systems, such as WGS84. At the International Date Line, the sign of the longitude value flips. If we would apply the monotonic chain partitioner on this situation, it would process the arc as if it would go the long way around the sphere, while in reality it just crosses the International Date Line.

Nonetheless, in cases where there are no arcs across the International Date Line, we can apply our x-monotonic sweep line algorithm. It still holds that arcs can only intersect if their end points have a different y-ordering compared to their starting points and that the x-ranges of the arcs overlap.

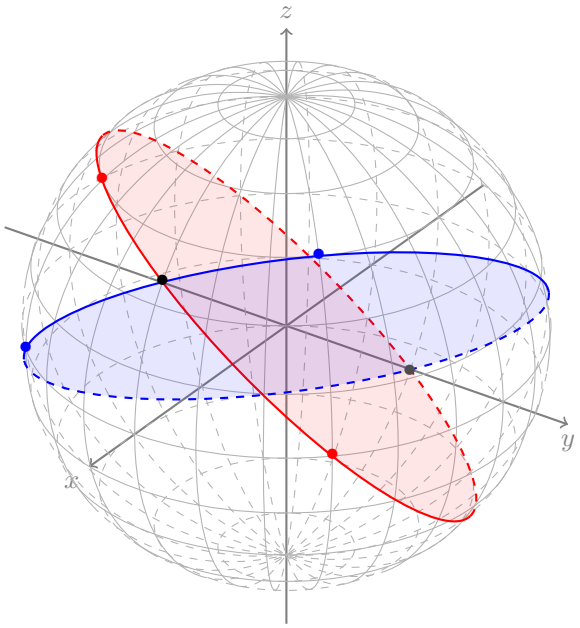


Figure 12: Two intersection points of two great circles.

5.7 Containment

In the Cartesian coordinate system, one of the most common methods to check for containment inside a polygon is using the crossing number. This algorithm is described in Section 4.4. Unfortunately, this does not convert directly to our geodetic coordinate system. It uses a line going to infinity to check for intersections. Such a line is impossible on a sphere, because it hits itself from the back. However, we can replace this by drawing the line to a point we know for sure is not in the polygon. This looks like a recursive problem, because how do we know a point is or is not in the polygon. Fortunately, there exists a method to check if the polygon contains one of the poles. If we know that the North Pole is not in the polygon, we can draw our line to the North Pole and use for the remainder the crossing number algorithm.

To check if one of the poles is inside the polygon, we use the bearing of the line segments [Gilman, 2012]. The bearing is the horizontal angle between the direction of the path and the North Pole. If you follow a path along a great circle, your bearing changes constantly, as visualized in Figure 13. To check if a pole is inside the polygon, we take the difference in initial bearing and final bearing of each line segment as well as the difference between the final bearing of a line segment and the initial bearing of the successive bearing. If we sum all these differences, we get five possible results:

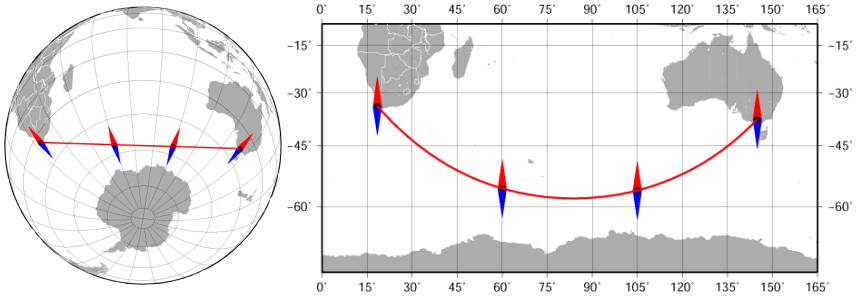


Figure 13: The bearing changes along a great circle path. Image by [Darekk2, 2015] is licensed under CC BY-SA 4.0.

$sum = 360°$:	None of the poles lie inside the polygon.
$sum = 180°$:	The polygon's boundary coincides a pole, but does not contain the opposite pole.
$sum = 0°$:	One of the poles lie inside the polygon or the polygon crosses both poles.
$sum = -180°$:	The polygon's boundary coincides a pole and contains the opposite pole.
$sum = -360°$:	Both poles lie inside the polygon.

Unfortunately, we did not find a way to discover which of the poles are contained. However, if one would have such a method, one could rotate the points such that the input point which is to be tested for containment is one of the poles and apply the above computations to find if the point is contained in the polygon. Therefore, our code considers only cases where both poles are not contained in the polygon. When we know that the North Pole is not contained, we apply the crossing number algorithm on a line from the input point to the North Pole. In every other case, the algorithm returns nothing.

5.8 Convex hull

In this report, we use the definition of a convex polygon as a hull which contains all shortest paths between two points [Higgins, 2002]. The convex hull is then the smallest of such convex polygons. In this definition, smallest is with regards to the area of the hull. However, if the points cover more than one hemisphere, the set of shortest path covers the whole sphere and, therefore, the convex hull does too. As our algorithm can not create a polygon representing the whole sphere, we do not consider inputs where the points cover more than one hemisphere.

For the convex hull in the Cartesian coordinate system, we used Graham's scan to create the convex hull of a set of points. However, this approach does not work for our spherical case directly. You can not pick the left-most refer-

ence point, because the sphere wraps around itself. However, we use the same principle as the Graham scan and adopt it to our spherical coordinate system [Grima and Marquez, 2001]. We again scan all the points to build the convex hull and when we encounter a point which makes the hull concave, we drop it and continue our scan.

Firstly, we check if all the input points reside on one hemisphere and, if true, find the pole of this hemisphere. One can see any point on the sphere as a pole of its own hemisphere. Every other point should be in this hemisphere for them to be in the same hemisphere. More generally, we want to find the intersection I of all these hemispheres. Any point in I would be a valid pole for the hemisphere containing all the input points.

We can find this using the algorithm from [Davis, 2014]. As stated above, every point can be seen as a pole of its own hemisphere, but we can also see it as the normal of a great circle. This algorithm uses the fact that the vertices of the polygon I are made of intersections between these great circles of the points. For every pair of points, we find the two places, i_1 and i_2, where their great circles intersect. We then have to find out which of these (or both) intersections are part of I by checking whether their hemisphere contains all other points. This is done by checking the angle between i_x and all other points. As long as this angle does not exceed $90°$, then i_x is a valid point and also a vertex of the polygon I. Lastly, we take the centroid of all these vertices of I to get a pole p_{pole} of the hemisphere which contains all the input points. If the input points are not all collinear, P lies in between at least 3 points. Therefore, p_{pole} is not part of the convex hull. Note that at least one of the two intersections is part of I if all the points lie on the same hemisphere. For the intersection of the first 2 points, this is obvious. For every next point, at least one point lies on the boundary of I, because the edges of I are previously considered points' segments of the great circles. Because each point contributed an edge (with duplicates) and all the points lie on the same hemisphere, at least one intersection point between the new point and every considered point lies on the boundary of I.

As with the original Graham scan, we first have to pick a reference point for which we know it lies on the convex hull. We pick this as the point farthest from the newly created North Pole. Next, we sort the points based on the angle between the point and the reference point along the xy-plane. Then, we start at our reference point and scan around the globe based on the ordering. The 2D version of the scan checks for right turns for three consecutive points. If this is the case, the middle point makes the hull concave and is not needed. The same holds for our modified algorithm, but now we check for intersections.

Let p_{pole} be the pole of the hemisphere, points p_{i-1}, p_i currently top items on the stack representing the current convex hull, and point q as the newly scanned point. Remember that the convex hull contains the shortest paths between all points. Therefore, if for a point all of its shortest paths may not cross the convex hull. If the arc $\overline{p_{i-1}q}$ does not intersect with the arc $\overline{p_i p_{pole}}$, as in Figure 14a, point p_i is not needed for the convex hull. All of its shortest paths to other

points will also not cross the arc $\overline{p_{i-1}q}$. However, if the two arcs do intersect, as in Figure 14b, point p_i is part of the convex hull. Remember that p_{pole} is the centroid of the input points. This means that at least 1 point lies on the other side of p_{pole} and that if $\overline{p_i p_{pole}}$ intersects, the arc to the other point also intersects.

So, when there is an intersection, we keep point p_i on the stack. Otherwise, we remove point p_i from the stack and do the same test on the new top two items of the stack p_{i-2}, p_{i-1}. This continues until no new intersection is found. In both cases point q is added to the stack and the scan continues to the next point until all points are considered. Note that the reference point is considered twice. Once at the start of the algorithm and the second time at the end of the scan as point q. Afterwards, we have to rotate the points on the stack back to their original positions to get the final convex hull.

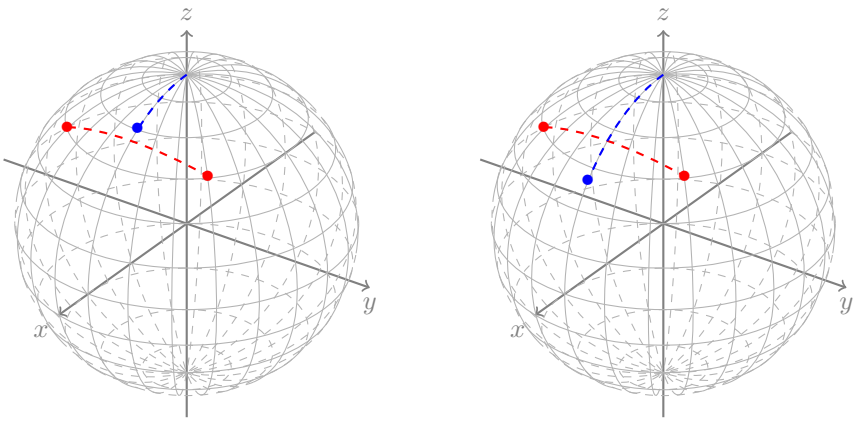
The running time of this algorithm is dominated by the first phase. For every pair of points, we compute the intersections of their great circles. For every such intersection, we have to check the remaining points and see whether they are at most orthogonal to the intersection point. In total, these steps takes $\mathcal{O}(n^3)$ time. Afterwards, the variant on the Graham scan just takes $\mathcal{O}(n \log(n))$ time. The total running time of the algorithm is therefore $\mathcal{O}(n^3)$.

However, this can be improved to get a running time of $\mathcal{O}(n \log(n))$ according to [Grima and Marquez, 2001]. To check if all the points lie on the same hemisphere, we want to know if all the points on the sphere can be separated from the center of the sphere in $\mathbb{R}^3$. This can be done using an algorithm for smallest bounding sphere problem, which is solvable in linear time. Only when the center of the result and the center of the actual sphere do not match, do the points lie on one hemisphere. Afterwards, one can create a pole by taking three non-collinear input points and computing their centroid. Unfortunately, there was no time to implement these improvements.

6 Implementation

The algorithms described in the above sections are implemented for Neo4j as user-defined functions and procedures in a library. The user can load this library into their Neo4j instance and call the functions on their graph data as long as their data is stored in the same way as described in Section 3.

In this section we describe the decisions we made and issues we encountered during the implementation of the theoretical components.



(a) p_{i-1} is removed from the stack, because there is no intersection.

(b) p_{i-1} stays on the stack, because there is an intersection.

Figure 14: Arc $\overline{p_i p_{pole}}$ (in blue) and arc $\overline{p_{i-1} q}$ (in red)

6.1 OpenStreetMap

Unfortunately, the main advantage of OpenStreetMap can also be a disadvantage. The fact that the data is being crowdsourced means that huge amounts of data can be collected, but it also means that it is hard to enforce consistency.

Let us consider the example of connecting the ways of a relation to create a single geometry. The order the ways are stored in the relation are sometimes used to also describe the order the ways succeed each other, however this is no guarantee. The same holds for the extreme points of ways. In some cases, the last point of a way is equal to the first point of the next way, but again, this is not guaranteed. We also cannot rely on the direction of ways, but this can be easily be explained by the reusing aspect of ways. This makes it impossible for every relation to have every way go in the same direction.

All these inconsistencies make creating geometries from relations non-trivial. To cope with this, we follow the following steps. Firstly, we try to create long chains by finding ways which overlap at their extreme points. Next, we connect ways by finding the nearest neighboring way. However, if the nearest neighbor is too far away, we do not connect the ways, because it is likely that the two ways are separate. This happens, for example, in multipolygons, where two neighboring polygons are still two separate geometries and should not be connected.

When we connect two ways, we create a relationship between the two neighboring points. This relationship refers back to the `Relation` which this polygon describes. When we later traverse through polygon and a `Way` ends, we can look for the previously created relationship to easily jump to the next way.

Another possible issue can be the roles given to ways in a relation. Every way has either the role `outer` or `inner`, for respectively, shells and holes. However, the wiki from OpenStreetMap advises to ignore those roles when making a polygon, because erroneous labeling can happen. Luckily, the multipolygon building algorithm does not rely on these labels. It has to check which holes reside in which shells anyway, because this information is not given by the roles at all.

6.2 Precision

All the numerical computations with regards to the coordinates of the points are done using Java's `double` primitive data type. This data type uses 8 bytes compared to `float`'s 4 bytes, but it offers greater precision. However, even this higher precision is not always accurate.

Because rounding errors can occur during computations and especially compound computations, which work on previously computed values, values may differ from what one would expect. For example, if we want to compare to the angles of two lines, the values could be virtually the same but differ in the 14th decimal place. Even though the actual angles are the same, the computations add some error to these values. If we would try to compare these two angles using Java's `==` operator, it would return `false`. To solve this issue, we have implemented our own equality function for `double`'s. This function does not compare the two actual values, but the absolute difference between the two. If this difference is small enough, for example, they differ after the 14th decimal place, it will return that the two values are equal. Hence, this equality function negates the rounding error of the computations.

6.3 Geodetic convex hull

While implementing the geodetic convex hull algorithm, a problem arose when applied to small polygons. When they are transformed to be around the North Pole, the range of values was greater and numerical imprecision had less of an impact. This expressed itself in the fact intersection which should be detected were missed. To solve this, we apply the cartesian convex hull algorithm on such small polygons. The great circle arcs not that long and, therefore, interfere less with the cartesian algorithm.

The described geodetic algorithm does work on polygons on a larger scale, such as across the United States of America. In those cases, the points lie further from the new North Pole, which helps to reduce the errors from numerical imprecision.

7 Benchmarks

We have discussed multiple data structures as well as comparable algorithms for two coordinate reference systems. In this section we show the results of benchmarking done on these variables to show how the data structures compare to each other as well as analyze how algorithms for the different coordinate reference systems perform.

7.1 Hypotheses

7.1.1 Data structures

As stated in Section 3, we have implemented both a property-based as well as an sub-graph representation of poly geometries. We have used them while running several algorithms. These algorithms can be divided into two categories, one is based on sequential access of the points and the second requires random access. The area computations and the linear reference algorithms for both coordinate systems fall in the first category. For both, we traverse through the polygon in order. The second category contains the remainder of the algorithms. Here we cannot guarantee that the points are accessed sequentially. An example of this is the Graham scan. Because it sorts the points, there is a large possibility that the original order is lost.

We suspect that the property-based data structure will perform better, because all of the points can be read quickly while the sub-graph representation has to first traverse the whole sub-graph which adds a lot of overhead. However, we made the distinction above between algorithms because we hypothesize that the sub-graph representation will perform relatively better on sequential algorithms compared to the random access algorithms with regards to memory usage. This is because only two points have to be stored in RAM at any given time compared to storing the whole polygon as with the property-based representation.

7.1.2 Algorithms

We also tested and compared the performance of the algorithms for their respective coordinate reference systems. For example, we compare the performance of the area calculations for both the Cartesian as well as the WGS84 coordinate reference system.

Our hypothesis is that the Cartesian algorithms will perform significantly better than the geodetic algorithms. This will even hold for algorithms where they share main part of the algorithm. The suspected cause of this are the many trigonometry functions used by the geodetic algorithms, such as transforming to and from the n-vector model.

7.2 Results

In the Tables 1, 2, and 3, the results of the benchmarking can be found. In the next sections we discuss the actual results and how they compare to our previously stated hypotheses.

7.2.1 Data structures

From the memory usage results, we cannot draw any definite conclusions. In two cases, the sub-graph representation uses more memory than the property-based representation, even though the algorithms in case, area en linear referencing, use sequential access of the points. In other cases, the memory usage of the sub-graph representation is less than the property-based version. However, this also holds for the convex hull algorithms, for which we hypothesized would take at least the same amount of memory because all points have to be in memory to be sorted. We do see worse performance across the board for the sub-graph represenation, which is in line with our expectations of the data structure.

A possible cause of the weird memory usage is the way that the JVM and Neo4j handle the memory and use caching may interfere with our measurements. Either more extensive measurements have to be taken or a functional test on large data sets on a low memory machine should be ran to see if our original hypothesis holds.

7.2.2 Algorithms

As expected, the geodetic algorithms perform worse than the Cartesian algorithms for all algorithms. The algorithms were often more complex or requires more mathematical functions to work.

We do see a difference between the intersection algorithms both on synthetic and actual data. The former was created by circular sweeping around a point and adding points to the polygon at a random distance from the reference point along the sweep line. This method gave fuzzy or spiky polygons. However, these polygons had a large number of x-monotonic chains of a small size. Therefore, the sweep line algorithm did not perform much better than the naïve algorithm. However, when using actual data, the sweep line algorithm generally performed better than the naïve implementation. We suspect that this improvement originates from the more natural data containing fewer x-monotonic chains in general.

Algorithm	CRS	Data structure	Time (s)	Memory (MiB)
Area	Cartesian	Graph	0.134	3317
	Cartesian	Array	0.026	827
	Geographic	Graph	0.167	324
	Geographic	Array	0.055	502
LinearReference	Cartesian	Graph	0.774	734
	Cartesian	Array	0.086	467
	Geographic	Graph	0.88	323
	Geographic	Array	0.115	496
ConvexHull	Cartesian	Graph	4.931	306
	Cartesian	Array	0.078	315
	Geographic	Graph	811.609	286
	Geographic	Array	577.039	315

Table 1: Comparison of algorithms and data structures on real OpenStreetMap data

Algorithm	CRS	Time (s)	Memory (MiB)
Area	Geographic	0.779	3317
	Cartesian	0.04	827
Distance	Geographic	2.323	324
	Cartesian	0.048	502
IntersectionLineSegments	Geographic	18.785	734
	Cartesian	0.389	467
IntersectionPolygonNaive	Geographic	1360.109	323
	Cartesian	11.559	496
IntersectionPolygonSweep	Geographic	1104.998	306
	Cartesian	80.476	315
LinearReference	Geographic	2.314	286
	Cartesian	1.539	315

Table 2: Comparison of impact of coordinate reference systems on synthetic data

Variant	CRS	Data structrure	Time (s)
Naive	Geographic	Graph	144.983
		Array	77.365
Sweep	Geographic	Graph	10.883
		Array	1.024
Naive	Cartesian	Graph	411.872
		Array	1.544
Sweep	Cartesian	Graph	32.389
		Array	1.608

Table 3: Comparison of the intersection algorithms on real OpenStreetMap data

8 Further work

As stated in Section 5.8, a method to create the pole of the hemisphere containing all the points can be created in $\mathcal{O}(n)$ time instead of $\mathcal{O}(n^3)$ time using a bounding sphere algorithm and three non-collinear points.

The containment and intersection algorithms can also be improved by comparing bounding boxes before the actual algorithms themselves. When the bounding boxes of the two geometries do not overlap, there will be no containment or intersections. These bounding boxes can be pre-computed and saved together with the geometries. Another potential improvement to the geodetic naive intersection algorithm is to precompute the n-vectors for the given points. While doubling the space requirement, it reduces the number of conversions from WGS84.

One way to possible improve the sub-graph representation with regards to memory usage is to make use of the Cursor APIs of Neo4j instead of the currently used Core API. The latter is more convenient to work with, but, because of that, it also has more overhead. The former works more closely to the underlying kernel to avoid adding overhead, but the developer has to use different cursors for different sources. The overhead is both computational (a new object is created for every node) and storage-wise (cursor uses less space).

Due to lack of time, the two data structures are not tested on extremely large (1M+ points) data sets. One could run them on these data sets and see if there is an improvement in memory usage for the sub-graph representation compared to the property-based representation. Using such large data sets, could also makes it possible for the property-based to not be able to run at all. Because the whole data set may not be able to fit in the memory at the same time.

9 Conclusion

In this report, two methods to store poly geometries in the Neo4j database as well as a suite of algorithms are discussed. The two different representations of poly geometries are property-based, in which the points are stored as an array in a property, and a sub-graph, where the points are stored as separate nodes in Neo4j. The property-based representation of poly geometries outperforms the sub-graph version with regards to running time. However, we could not draw any definite conclusions with regards to the memory usage of the two data structures.

The set of algorithms is developed for both the Cartesian and WGS84 coordinate reference system. For the WGS84 algorithms, a spherical approximation is used with the n-vector model. We did see worse performance for these algorithms, but those can be easily explained by the need of more mathematical computations.

We have also shown a method to import and use OpenStreetMap data in Neo4j. The above mentioned data structures and functions can also be applied to the imported OpenStreetMap data set while still using the original structure created by the importer.

References

[Bentley and Ottmann, 1979] Bentley, J. and Ottmann, T. (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647.

[Darekk2, 2015] Darekk2 (2015). Bearing and azimuth along the geodesic. `https://en.wikipedia.org/wiki/File:Bearing_and_azimuth_along_the_geodesic.png`.

[Davis, 2014] Davis, V. (2014). Same hemisphere test. `http://rstudio-pubs-static.s3.amazonaws.com/27121_a22e51b47c544980bad594d5e0bb2d04.html`.

[Gade, 2010] Gade, K. (2010). A non-singular horizontal position representation. *Journal of Navigation*, 63(3):395–417.

[Gilman, 2012] Gilman, J. (2012). Follow up to determining if a spherical polygon contains a pole. `https://www.element84.com/blog/follow-up-to-determining-if-a-spherical-polygon-contains-a-pole`.

[Grima and Marquez, 2001] Grima, C. and Marquez, A. (2001). *Computational Geometry on Surfaces: Performing Computational Geometry on the Cylinder, the Sphere, the Torus, and the Cone*. Springer Netherlands.

[Higgins, 2002] Higgins, P. (2002). *Mathematics for the Imagination*. OUP Oxford.

[OpenStreetMap contributors, 2017] OpenStreetMap contributors (2017). Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`.

[Park et al., 2001] Park, S. C., Shin, H., and Choi, B. K. (2001). A sweep line algorithm for polygonal chain intersection and its applications. In *Geometric Modelling: Theoretical and Computational Basis towards Advanced CAD Applications. IFIP TC5/WG5.2 Sixth International Workshop on Geometric Modelling December 7–9, 1998, Tokyo, Japan*, pages 309–321. Springer US, Boston, MA.

[Taverner, 2019] Taverner, C. (2019). OSM for Neo4j. `https://github.com/neo4j-contrib/osm/`.

[Todhunter, 1871] Todhunter, I. (1871). *Spherical Trigonometry, for the Use of Colleges and Schools*. Macmillan and Company.

[Vanderbei, 2013] Vanderbei, R. (2013). Girard's theorem. https://vanderbei.princeton.edu/WebGL/GirardThmProof.html.