Impact Factor:					
ISRA (India)	= 3.117	SIS (USA)	= 0.912	ICV (Poland)	= 6.630
ISI (Dubai, UAE)	= 0.829	РИНЦ (Russia)	= 0.156	PIF (India)	= 1.940
GIF (Australia)	= 0.564	ESJI (KZ)	= 8.716	IBI (India)	= 4.260
JIF	= 1.500	SJIF (Morocco)	= 5.667	OAJI (USA)	= 0.350

SOI: 1.1/TAS DOI: 10.15863/TAS

International Scientific Journal

Theoretical & Applied Science

p-ISSN: 2308-4944 (print) **e-ISSN:** 2409-0085 (online)

Year: 2019 **Issue:** 05 **Volume:** 73

Published: 27.05.2019 **http://T-Science.org**

QR – Issue QR – Article

Vadim Andreevich Kozhevnikov
Senior Lecturer
Peter the Great St. Petersburg Polytechnic University
vadim.kozhevnikov@gmail.com

Nikita Dmitrievich Yatskovets
student
Peter the Great St. Petersburg Polytechnic University
nikita123456789012@yandex.ru

DEVELOPMENT OF A WEB SERVICE FOR BUILDING ROUTES ACCORDING TO USER INTERESTS

Abstract: *This article contains ideas and suggestions about how to create a service and an algorithm that recommends to users a route for a walk based on his chosen interests. And the article also contains base ideas about optimization of the suggested algorithm to achieve «production» quality of execution and response time.*

Key words: *recommender systems, routes building.*

Language: *English*

Citation: Kozhevnikov, V. A., & Yatskovets, N. D. (2019). Development of a web service for building routes according to user interests. *ISJ Theoretical & Applied Science, 05 (73),* 350-357.

Soi: http://s-o-i.org/1.1/TAS-05-73-51 ***Doi***: crossref https://dx.doi.org/10.15863/TAS.2019.05.73.51

Introduction

Imagine a situation, a person flies on a plane to Moscow or other big city for a few days on business, or just to rest. A person is not familiar with the city and does not know where to go for a walk, what to visit, etc. In the case of attempts to find out popular places in Moscow, a person will face the problem that there are too many options and to choose one of these options is difficult. Person need to know the time of work of each place, read reviews. If there was a service that would allow in a few clicks to build a ready route through several popular places, the user would be much easier. Yes, today we have Google. Trips [1], but this application doesn't allow to choose different options for your route, this application just build some route, you can't affect this process. The main goal of the developed service is to solve this problem and give user ability to build route by own interests.

The second situation, people who are in a familiar city, but do not know where else to go, because most places have already been visited, and they don't want to think where to go (often real situation, when you don't know where to go, and you choose to stay at home). Or a person just wants to wander around the city, not thinking about self-chosen route.

Thus, it was decided to create a service that in a few clicks would allow the user to get an answer to the question "where to go for a walk in this city?".

The aim of the article

In this article it was decided not to consider the creation of a mobile application because mobile application just should to print points on the map, nothing interesting. Main point of this article – is describe possible algorithmically solution of suggested problem and server architecture that can be used to run suggested algorithm.

The solution that will be described in the article was tested in St. Petersburg city and showed good results both in terms of the quality of the routes and the speed of work. You can find examples of work in the end of article.

Methods

As a main programming language was chosen Java programming language, because this langue de facto is standard for web development and a lot of framework and libraries exist for solving any web development tasks.

To store objects that will be used for testing was chosen MySQL DBMS as most popular free DBMS [2] for simple web projects [3].

To work with database was chosen Hibernate ORM instead of using in-code SQL queries. The main reason of this choice is simplify working with database objects like place, because each place will contain a lot of information like coordinates, name, description, full address, rating and so on. It is too hard to handle each filed separately. The easiest way – just use Hibernate ORM [4] and work with completed place objects without writing a lot of queries.

To receive user's requests and proxy these requests to the algorithm simple http-server is required. Netty framework [5] was chosen as base for handling http requests. This framework allows to implement simple http server with non-blocking handling of http request quite fast and simple.

So, wrapper (means parts of receiving and sending user's requests) of the algorithm that will be suggested in article has the following form:

1. Using Netty framework, server receiving user's request, handle requests and send handled request to the next stage.
2. Algorithm handle requests and return JSON object with suggested points.
3. Netty framework send JSON object to user.

As you can see, no stages related with database working. It is related with performance issue, because queries to database is too long and affect service performance. By this reason all the objects stored on the RAM. Details of this solution will be described later.

To solve suggested problem offline maps with meta information about geo-objects required. Was chosen OpenStreetMaps [6] as maps source, because these maps have open format and free license.

The algorithm proposed further will not use machine learning, as it might seem from the title of the article, since it is almost impossible to select for this task some training dataset. So, as the method of creating algorithm was chosen method of heuristics, when we creating algorithm using various approximations and assumptions.

Implementation of the algorithm

The first problem that had to be solved of algorithm development – it is the choice of the way to interact with offline maps that were obtained from OpenStreetMaps service. Maps from OpenStreetMaps service contains a huge amount of meta information to construct best route between two points and get estimates of time and distance for constructed route, so manually working with these maps is very difficult. By this reason the open-source library GraphHooper [7] was chosen to work with OpenStreetMaps maps.

GraphHooper library contains completed algorithms to work with maps from OpenStreetMaps [6] directly. This library also contains algorithms to estimate time and distance of the constructed algorithms. To build best route between two points chosen library implements two public algorithms: Dijkstra's algorithm [8] and A star algorithm [9] only in bidirectional implementation. Besides these two algorithms GraphHooper library also implements own heuristics to construct best route in the fastest way [10], but we will not to consider these implementations, because they are not related with main target of this work.

Constants description

Due to the computational complexity of the suggested problem, it was proposed to introduce some constants to reduce huge number of possible routes to build. To understand the scale of the problem, consider an example: suppose our service has about 1000 real points on the map, to build only one route with 5 points for one user service should consider 8250291250200 different routes. It is impossible, so by this reason we will consider following constants.

Were proposed and implemented following constants:

```
private final int
NEXT_STEP_POINTS_RANDOM_COUNT = 8;
private final double
MIN_DISTANCE_BETWEEN_TWO_POINTS =
500;
private double
MAX_DISTANCE_BETWEEN_TWO_POINTS =
1300;
private final double MAX_WALK_TIME = 240 *
1000 * 60;
private final int MAX_POINTS_PER_ONE_ROUTE
= 5;
private final int MIN_POINTS_PER_ONE_ROUTE
= 3;
private final int RADIUS_INCREMENT = 400;
private final int
MAX_DISTANCE_OF_INTERSECTION = 150;
private static final int
MAX_CATEGORIES_NUMBER = 5;
private static final int MAX_RESET_TIMES = 3;
```

Let's consider what each constant means separately.

NEXT_STEP_POINTS_RANDOM_COUNT – this constant determines the number of points that will be chosen as potential candidates for the next point in the route. The choice is made among the points that have passed the initial filtering by category and fall into the current search boundaries by the distance (constants that determines these distances will be proposed later) from the current point. Algorithm should try to choose NEXT_STEP_POINTS_RANDOM_COUNT points with the shortest distance from the current one for the next iteration of filtering. The next stage of the filtering does not necessarily get exactly NEXT_STEP_POINTS_RANDOM_COUNT points, this constant is the upper limit of the number of points at this stage of the algorithm. Default value decided to set 8.

MIN_DISTANCE_BETWEEN_TWO_POINTS – this constant determines the minimum distance between two points in the one route. After some tests of the completed algorithm it was decided to choose a value of 500 meters for this constant. It should be noted that this distance – not the distance between two points as route, it is distance between two points by «line». That is, the real distance that the user will have to go may be somewhat more than 500 meters.

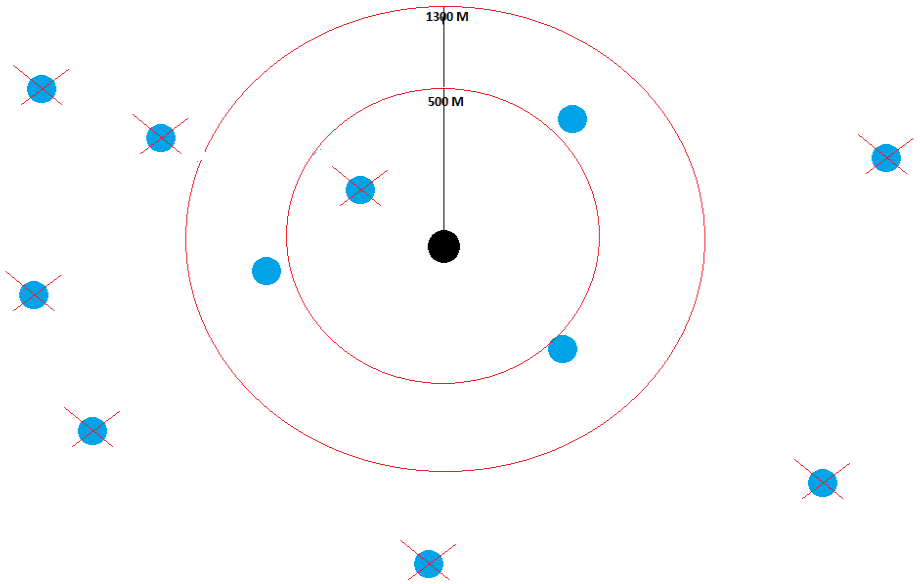

Figure 1 - Filter working demonstration

MAX_DISTANCE_BETWEEN_TWO_POINTS – this variable determines the upper limit of the distance between two points. This distance as the MIN_DISTANCE_BETWEEN_TWO_POINTS is not the distance between two points as route, it is distance between two points by «line». It should be noted, that it is not a constant, it is dynamic value that can be increased in some bad cases by RADIUS_INCREMENT constant (this constant will be proposed later). Main case when this value should be increased is when user want to build route far from city center. In this case points in the initial radius may be not enough to choose next point of the route and we should increase MAX_DISTANCE_BETWEEN_TWO_POINTS value to have more points for choice. Default value is 1300 meters.

Let's explain how filter works (Fig. 1). Black point – it is current point of the route. Blue points – are potential candidates for the next point in the route.

MAX_WALK_TIME – this constant determines the maximum time for which the proposed route can be walked without considering stops and visits to proposed places or establishments. This constant is introduced to handle cases where the variable MAX_DISTANCE_BETWEEN_TWO_POINTS grows to too large values, as well as for cases where the actual distance between points is very different from the estimated distance by «line». Default value is 4 hours should be enough for regular people.

MAX_POINTS_PER_ROUTE – this constant determines the maximum points in the one route. Default value for this constant was chosen as 5 points in the one route. This value was chosen because for some categories of places, such as parks, for example, the points become too far geographically separated and the route will not be interesting to the user (e.g. in one area of the city rarely more than 5 parks).

MIN_POINTS_PER_ROUTE – this constant determines the minimum number of points in the one route. The value set to 3 was chosen as the minimum adequate, since a route of less than 3 points does not have any informative meaning for the user.

RADIUS_INCREMENT – this constant determines the size of the increase in the maximum allowable distance between the current point and points - potential candidates for the next point in the route (variable MAX_DISTANCE_BETWEEN_TWO_POINTS) with an unfavorable outcome.

MAX_DISTANCE_OF_INTERSECTION - this constant determines the maximum value of the intersection of the already existing part of the route and the potential continuation of the route. This constant was introduced due to the fact that, as it was found out during the experiments, routes that are built without any restrictions on the places where the paths

between the points will pass, will be "porridge" from the intersection and overlapping of the paths between the points each other. Simply put, the user will go to different points around the same place, and in the worst case the route will be impossible to disassemble, or the user will go on the same roads, which is not at all interesting. A value of 150 meters was chosen as optimal for the speed of work and the quality of the received routes. With the increase of this number, the quality of routes significantly decreases, since there are many paths that follow the same road, but in different directions.

MAX_CATEGORIES_NUMBER – this constant determines current value of available categories to choose by user. Now, this constant is equal to 5.

MAX_RESET_TIMES - this constant defines the maximum number of times the application will try to rebuild the route in case of failure to build a route the first time. The introduction of this restriction was a side effect of the introduction of all the improvements and limitations described above, as now there are many options when a route cannot be built under certain conditions that will cause the entire service to hang. The most frequent case when there is a "reset" (which will be discussed in more detail later) of the constructed route is when there is only one road to any establishment and this road is longer than MAX_DISTANCE_OF_INTERSECTION. Thus, if this institution was chosen at the previous step, then it will be impossible to build the next point due to the restriction MAX_DISTANCE_OF_INTERSECTION, which will not allow to go back along the same road.

Full description of the algorithm

Consider, now, the final implementation of the algorithm.

Each user request creates its own object — an instance of the class with the constants defined above. This instance is initialized with two parameters — the location of the expected start of the walk and the user's interests, which were chosen from the available categories.

At the current stage of the algorithm we also need to get all the points in the city that fit the chosen user categories. Working with a database is a very long operation, so it was decided to create a common storage which contains all the points from database in memory of application instead call database on each request. The storage it is just standard Java List collection. We don't need to use multi-threading collections, since the storage will be read-only in its idea. The storage is initialized when the server starts and does not change during the service is working. This solution imposes some difficulties if it is necessary to update the points. However, this operation should not occur often, so if necessary, it will be possible to quickly re-initialize the storage during small down time of the service. Also, all points were ordered by categories, which automatically as quickly as possible solves the problem of filtering points by categories. These solutions have significantly accelerated the application.

After selecting all the points that fit the categories, it is necessary to build a "distribution" of categories along the route. Under the distribution in this case refers to the order of each category in the route. This is necessary so that each category chosen by the user at least once is guaranteed to meet on the route in case this condition is not fulfilled, it is considered that the route cannot be built from this starting point. Distribution is also needed to exclude situations when the user chooses the categories "parks" and "food", and the service will offer to go 4 times to eat and go in one park, which is most likely to be completely uninteresting to the user. In this regard, the conditional weight of each category is not the same.

Consider the algorithm for constructing the distribution of points by categories during the route. Three different situations were highlighted:

1. The user has chosen one category. In this situation, there is no need to calculate anything - just fill the resulting array with a value that corresponds to the chosen category.

2. The user has selected all five categories currently available. In this case, we also do not need to calculate anything, algorithm just fill in the result array with the values of the categories in accordance with the order in the source list of categories.

3. The user selected two categories and more, but less than 5. Let the number of selected categories be N. Then the first N points will have N selected categories in the appropriate order. Consider now how the remaining 5-N places are distributed. As noted earlier, categories have different priorities. It was decided that the categories "parks", and "culture" would have the highest priority. Thus, if among the categories selected by the user there is the category "parks" or the category "culture", then the remaining 5-N places in the distribution are given to one of the respective categories. If both categories have been selected, the remaining places are distributed only among these categories randomly (that is, the chance to choose each of these two categories for each place will be 50%). In case there is none of these two categories, then each category of the chosen ones has the same chance of getting into each of the remaining places.

At this stage of the algorithm for the current instance of the class has all the necessary data to start the basic algorithm for finding the route. The algorithm is based on a cycle that works until the maximum walk time is reached, or until the maximum number of points in the route is reached by the MAX_POINTS_PER_ROUTE constant, or until the algorithm send signal to main thread that route cannot be built for current input parameters.

At each iteration of the loop, an attempt is first made to select the NEXT_STEP_POINTS_RANDOM_COUNT nearest points to the current point. In this case, the distance between the potential new point and the current point should lie in the range from MIN_DISTANCE_BETWEEN_TWO_POINTS to MAX_DISTANCE_BETWEEN_TWO_POINTS. To do this, we need to calculate the distance to a very large number of points. Considering that this is one of the first steps of the algorithm and it is possible to reset the route later, which will result in having to recalculate everything again, the performance of this piece of code is very critical for the performance of the service. During the attempts to optimize this part of the code, it was found that it was too resource-intensive to calculate the distance between two points using the GraphHooper library. It was decided to sacrifice the possible accuracy to significantly speed up the algorithm. Instead of calculating the real distance, it was decided to calculate the distance by «line» as described before. That is why the description of all the constants appears exactly the distance by «line». As you know, the distance between the points by «line» can be easily and

accurately calculated if there is information about the coordinates of each point, information on which hemisphere points and the availability of some information about the globe [11]. The problem with the definition of the hemisphere in this case is not necessary to solve, since all points are obviously located in the northern hemisphere. The coordinates of both points are known. As a radius of the earth, a value of 6371 kilometers was taken. As a working formula, the Haversine formula was chosen - (1).

$$\Delta\sigma = 2 * \sin^{-1}\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos\phi_1 * \cos\phi_2 * \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right) \quad (1)$$

Next, we need to translate the obtained distance from radians into meters, for this purpose, the obtained value using the formula of Haversine must be multiplied by the radius of the earth in meters. Thus, the distance between the points will be obtained with good accuracy. This formula is quite complex in terms of computation, but the use of the spherical cosine theorem is not possible because the distances between points are usually very small, and the spherical cosine theorem gives a very large error in this case. To use an even more accurate modification of the Haversine formula – (2) for antipode points does not make any sense, since its computational complexity is even more, but antipode points are obviously impossible within the boundaries of one city (antipode points are points that lie opposite each other, but in different parts of the planet).

$$\Delta\sigma = \tan^{-1}\left(\frac{\sqrt{(\cos\phi_2 * \sin\Delta\lambda)^2 + (\cos\phi_1 * \sin\phi_2 - \sin\phi_1 * \cos\phi_2 * \cos\Delta\lambda)^2}}{\sin\phi_1 * \sin\phi_2 + \cos\phi_1 * \cos\phi_2 * \cos\Delta\lambda}\right) \quad (2)$$

After replacing the use of the GraphHooper library to calculate the exact distance to calculate the distance using the Haversine formula, the speed of the algorithm increased significantly.

However, as you can see, the Haversine formula contains many «heavy» CPU operations, such as various trigonometric operations, multiplications, and taking the root. In this regard, it was decided to use a non-standard library to calculate these operations. The choice fell on the library Jafama FastMath. According to the authors of the library, the acceleration should be up to 10 times due to various optimizations [12]. After examining the performance estimates, it was found that all trigonometric functions used in the Haversine formula are much faster than in the standard Java library (Table 1).

However, it was also found that the speed of taking the root is 2 times slower [12] than in the standard library (Table1), therefore, the standard library will be used to calculate the root. The application of these changes also greatly accelerated the process of calculating all distances. In addition to calculating distances, it is also checked that the user will have time to reach a potential new point, considering the entire previous route. The initial time of the walk, now, is considered as the time the user accesses the system, but in the future, it can be set by the user for more flexible construction of routes.

If no matching points were found for the current MAX_DISTANCE_BETWEEN_TWO_POINTS parameter, MAX_DISTANCE_BETWEEN_TWO_POINTS will be increased by RADIUS_INCREMENT and the search will continue with the new value of the parameter. Each point found is excluded from the search for this route in order not to come to a point that has already been visited at a later stage. No matter how many points were found, more than NEXT_STEP_POINTS_RANDOM_COUNT search iterations will not be performed and regardless of the result, the algorithm proceeds to the next step.

In the next step, we need to weed out the points from those that were obtained in the previous step, which have intersections of more than MAX_DISTANCE_OF_INTERSECTION meters with paths already built. For this it is necessary to resort to the help of the GraphHooper library and to build already real routes and distances between the current point and of each potential one. Of all the

remaining points, a random one is selected and added to the route. In addition to the point itself, we must also save all the points that make up the path between the current point and the selected one in order to check the dimensions of the intersection of the paths for the new points. Further, the cycle begins already for the selected point.

The result of the algorithm is a data structure that contains from 3 to 5 points of the route to visit, as well as data arrays that contain information about how to move between each of the neighboring points. We can send this data to mobile client using JSON format.

Table 1. Comparing Java.Math and FastMath.

Function	Math Mean	FastMath Mean	Times Faster
acos	58	16	3.6
asin	57	15	3.8
atan	94	15	6.2
atan2	145	23	6.3
cbrt	112	18	6.2
cos	74	13	5.7
sqrt	7	15	0.5

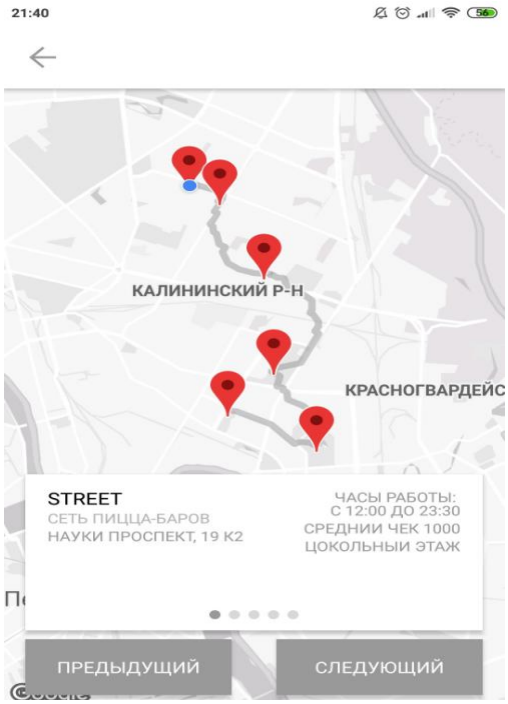

Figure 2 – «Eat» and «Parks» categories

Results and discussion

So, as said before, described algorithm already implemented completely and all the described constants were chosen after manually testing. To show examples of working of described algorithm was implemented Android application using React Native framework developed by Facebook company [13]. Implementation description of this application has been omitted, because it is just simple prototype developed only for illustrative examples.

First example is using «Eat» and «Parks» categories with starting point near university. As you can see on picture, application suggested to visit 4 parks and one pizza café (Fig. 2).

Second example is using «Culture» and «Drink» categories near center of Saint-Petersburg city. You can see suggested points on picture. Was suggested very interesting route near Neva river with beautiful landscape and famous points like

Kunstkamera museum (it is first museum in Russia) and Peter and Paul Fortress (Fig. 3).

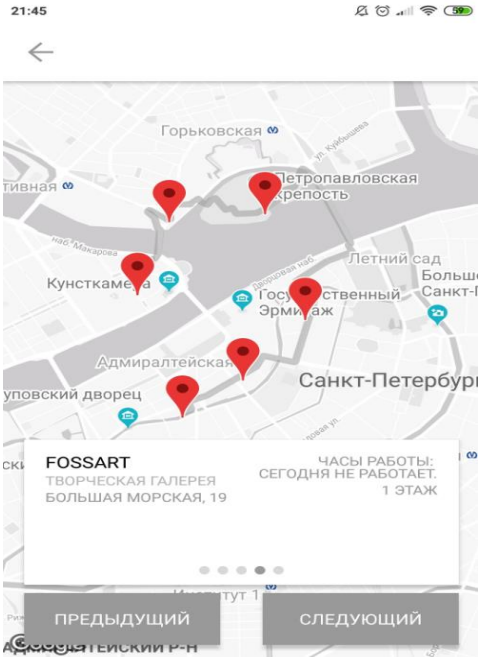

Figure 3 – «Culture» and «Drink» categories

Conclusion

A ready-to-use web service was created. In addition, was implemented simple mobile phone application to test created service. This service helps users to get an answer to the question "where to go for a walk in this city?" with completed route and points to visit related with chosen interests.

References:

1. (n.d.). Organize your trips with Google trips [online]. Retrieved May 24, 2019, from https://get.google.com/trips/
2. (n.d.). What is DBMS [online]. Retrieved May 23, 2019, from https://searchsqlserver.techtarget.com/definition/database-management-system
3. (n.d.). Top 10 DBMS [online]. Retrieved May 24, 2019, from https://mytechdecisions.com/it-infrastructure/10-best-database-software-systems-business-professionals/
4. (n.d.). Hiberante ORM for Java [online]. Retrieved May 24, 2019, from https://hibernate.org/orm/
5. (n.d.). Introduction to Netty [online]. Retrieved May 24, 2019, from https://www.baeldung.com/netty
6. (n.d.). About OSM [online]. Retrieved May 24, 2019, from https://wiki.openstreetmap.org/wiki/Main_Page
7. (n.d.). GraphHooper how to build route [online]. Retrieved May 23, 2019, from https://github.com/graphhopper/graphhopper/blob/0.7/docs/core/routing.md
8. (n.d.). Description and implementation of the Dijkstra's algorithm [online]. Retrieved May 15, 2019, from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
9. (n.d.). Description and implementation of the A start algorithm [online]. Retrieved May 15, 2019, from https://en.wikipedia.org/wiki/A*_search_algorithm

10. (n.d.). Now flexible routing is at least 15 times faster [online]. Retrieved May 23, 2019, from https://www.graphhopper.com/blog/2017/08/14/flexible-routing-15-times-faster/
11. (n.d.). Calculate distance, bearing and more between Latitude/Longitude points [online]. Retrieved May 19, 2019, from https://www.movable-type.co.uk/scripts/latlong.html
12. (n.d.). Improving Java Math Performance with Jafama [online]. Retrieved May 24, 2019, from https://www.element84.com/blog/improving-java-math-performance-with-jafama
13. (n.d.). Getting started React Native [online]. Retrieved May 24, 2019, from https://facebook.github.io/react-native/docs/getting-started.html