яОдеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування і економічної кібернетики

Дипломна робота

Бакалавра

на тему: «Алгоритми побудови оптимального маршруту на основі геоданих»

«Algorithms for constructing the optimal route based on geodata»

Виконав: студент денної форми навчання
спеціальності 113 Прикладна математика

Веремйов Кирил Валерійович

Керівник
Канд. техн. наук, доц. Мазурок І.Є._____

Рецензент
Канд. техн. наук, проф. Мороз В.В.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ р.

Завідувач кафедри

_____ _____
(підпис) (прізвище, ініціали)

Захищено на засіданні ЕК №_____

протокол № ____ від _____ р.

Оцінка_____/___ ___/_____
(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

_____ _____
(підпис) (прізвище, ініціали)

Одеса – 2022

SUMMARY

INTRODUCTION

There are many criteria for optimality. For the case when you are in a hurry, the most optimal route for you will probably be the fastest route. But for another occasion, for example when you want to take a walk, the most preferred route may be the most interesting and relaxing one. Let's look at this particular case. So, by optimization, we mean building to make the route more interesting.

In this paper, we consider the problem of constructing the most interesting route. The solution was implemented by obtaining geodata, analyzing them and, depending on the settings, choosing the best points to visit and laying the path between them. For implementation, the option with a mobile application was chosen, as it allows you to easily and conveniently use the application in situations that are closest to use cases.

As already mentioned, we are considering a situation where the most important criteria for a route are its attractions and interestingness. This can be in many cases: for example, you have free time in an unfamiliar city (or you would like to get to know your home area better) and want to take a walk (For example, waiting a transfer at the train station, etc.).

Alternatively, you need to go from one point to another, but you have free time and want to relax. Or you are a tourist and you are interested in the area, so you prefer to walk near beautiful places like parks, lakes, beautiful architecture, etc. not just to come as quickly as possible, but to relax a little and enjoy the surroundings. There should be a balance between travel time and interestingness so that you can enjoy the surroundings, look at the places of interest, and to reach your destination at time.

In addition, we have a look on vehicle trip. There are many cases for such a situation, for example, many people travel by car between cities and they may be interested in seeing (and maybe visiting) some interesting places by the way. Another case when you have free time (in a familiar or unfamiliar area), you have a bike or a car and you are interested in the surroundings. Trips by vehicle

compared to walking has its own features as directions of roads and road traffic, so we consider them too.

ВСТУП

Існує багато критеріїв оптимальності. У випадку, коли ви поспішаєте, найоптимальнішим маршрутом для вас буде, мабуть, найшвидший. Але для іншого випадку, наприклад, коли ви хочете прогулятися, найбільш вподобаним маршрутом може бути найцікавіший і розслабляючий. У цій праці й розглядається цей випадок. Отже, під оптимізацією ми маємо на увазі творення маршрута цікавішим.

У цій роботі ми розглядаємо проблему побудови найбільш цікавого маршруту. Рішення було реалізовано шляхом отримання геоданих, їх аналізу та, залежно від налаштувань, вибору кращих точок для відвідування та прокладки шляху між ними. Для реалізації був обраний варіант з мобільним додатком, оскільки він дозволяє легко та зручно використовувати додаток у ситуаціях, найближчих до випадків використання.

Як вже було сказано, ми розглядаємо ситуацію, коли найважливішими критеріями маршруту є його привабливість та цікавість. Таке може бути в багатьох випадках: наприклад, у вас є вільний час у незнайомому місті (або ви хочете ближче познайомитися з рідним місцем) і хочете прогулятися (наприклад, чекаєте на пересадку на вокзалі, тощо).

Як варіант, вам потрібно перейти від однієї точки до іншої, але у вас є вільний час і ви хочете відпочити. Або ви турист і вас цікавить місцевість, тому ви віддаєте перевагу ходити поблизу красивих місць, таких як парки, озера, прекрасна архітектура тощо, щоб не просто приїхати якомога швидше, а трохи відпочити та насолодитися околицями. Повинен бути баланс між часом подорожі та цікавим, щоб ви могли насолоджуватися околицями, розглядати визначні місця та вчасно дістатися до місця призначення.

Крім того, ми розглянемо поїздки транспортом. Випадків для такої ситуації багато, наприклад, багато людей їздять на автомобілі між містами і їм, до

речі, може бути цікаво побачити (а може, і відвідати) якісь цікаві місця. Інший випадок, коли у вас є вільний час (у знайомій чи незнайомій місцевості), у вас є велосипед чи машина і вас цікавить околиця. Поїздки на транспорті в порівнянні з пішохідними мають свої особливості як напрямки доріг і дорожнього руху, тому ми також їх розглядаємо.

OVERVIEW OF SYSTEMS AND ALGORITHMS FOR BUILDING TRIPS AND INTERESTING ROUTES

We are not the first with this issue; there are services with the same topic. One of the closest we found is "Site Safari" by Egor Smirnov and ITMO University.

It also provides choosing displayed categories, constructing routes from one point to another and a circular route. Directed route idea based on visibility polygons and tag analysis. The circular route idea is based on a genetic algorithm. Fig. 3.1 is an example of a directed route.

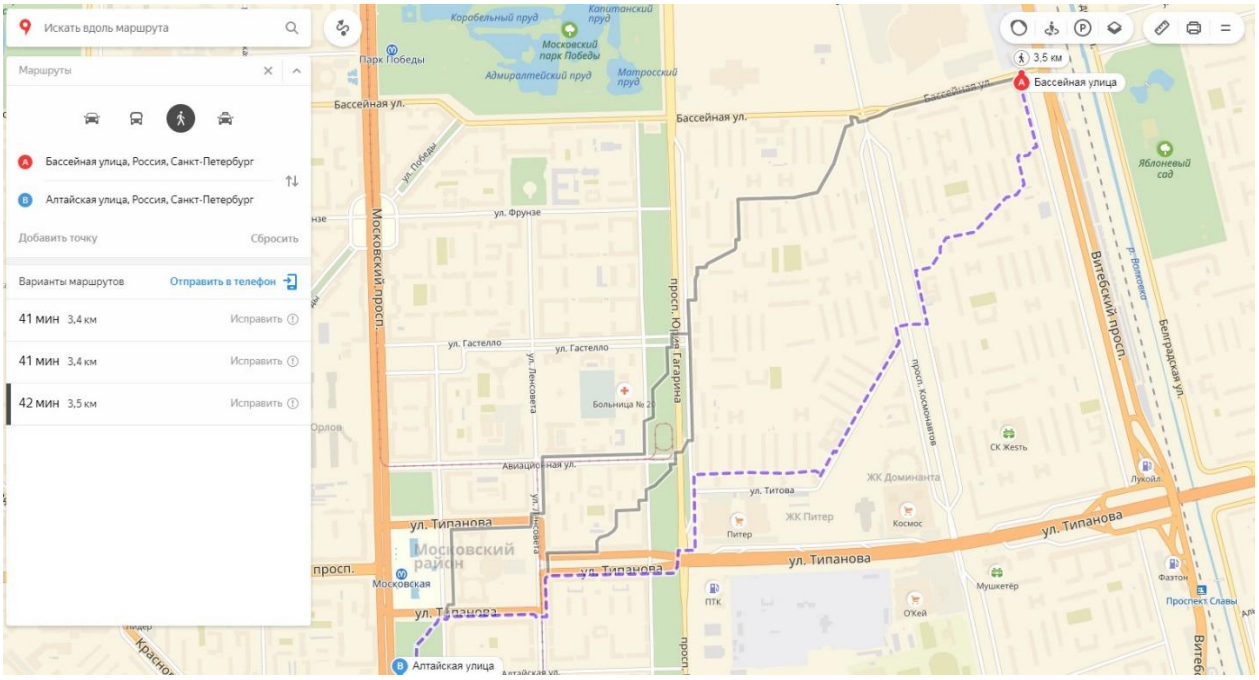


Figure 3.1

Another analogue with a different approach is the Inspirock service. It provides several options for places to visit over time.

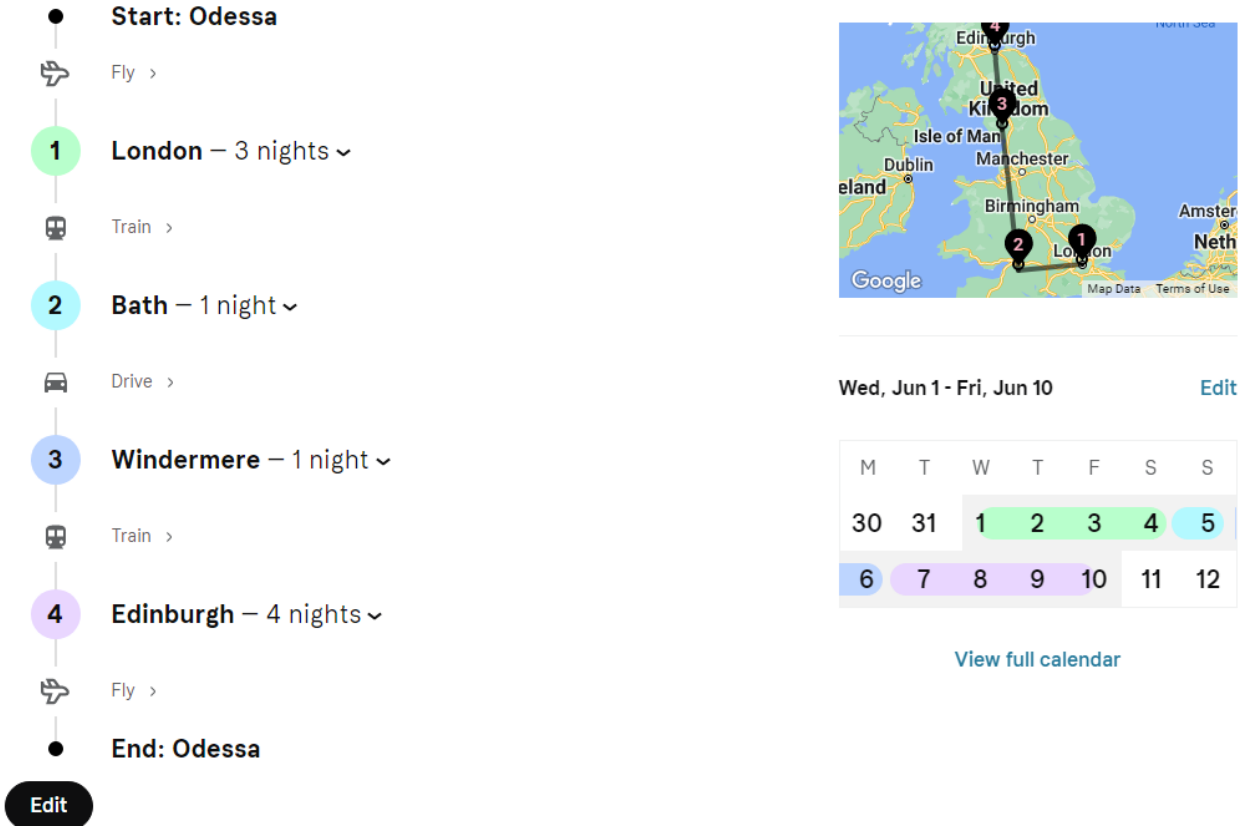


Figure 3.2

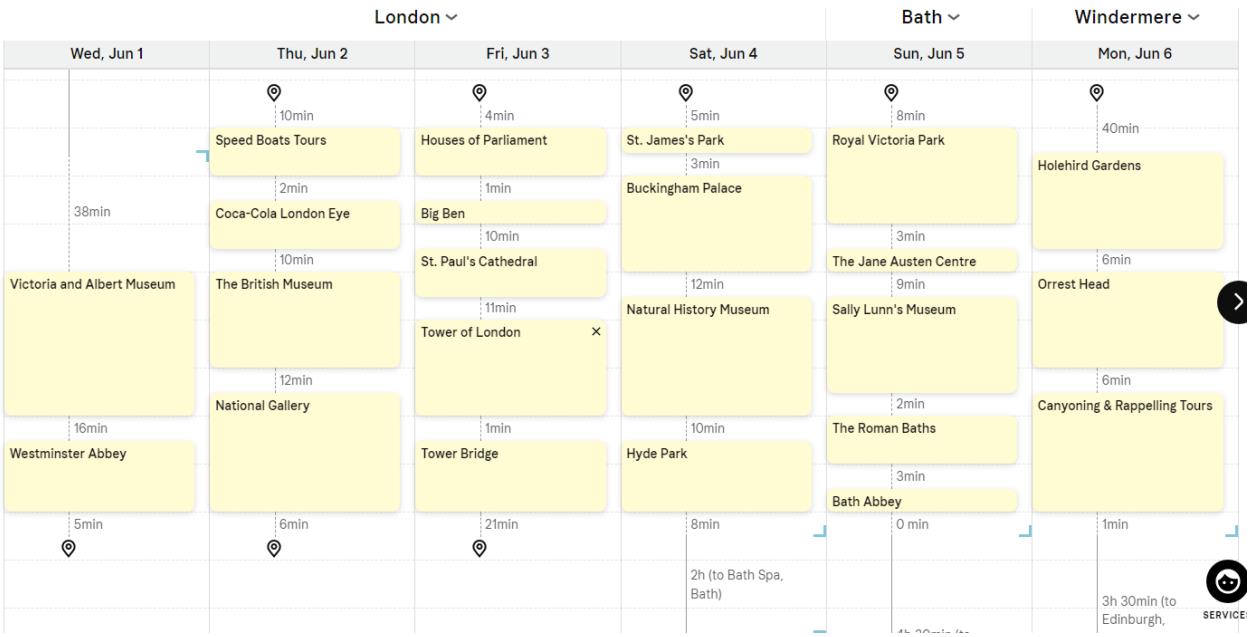


Figure 3.3

It offers a detailed plan for visiting places in the form of a timeline (fig. 3.2) or calendar (fig. 3.2) or map. Also, this application allows you to plan trips for several days or even weeks.

So, we see that there are several different approaches to quickly suggest an interesting route, and the task is quite empirical. Therefore, we decided to try some fairly simple solutions first, and focus on these use cases in this paper: a circular and directed route, and a walking and vehicle trip.

TASKING

The main objective of this paper is to offer a solution for building interesting routes. To do this, it was decided to develop an application that should provide the ability to plot routes on a map, taking into account the type of route, traffic options, user preferences, and so on.

IMPLEMENTATION

The implementation is built in Kotlin for Android using the MapBox SDK to get places (Open Street Map) and traffic data. MapFragment class is responsible for UI, and RouteLineGreedy, RouteCircleGreedy classes contain implementations of algorithms (look additions).

Case of a circular route

The first case, as already mentioned, you have free time and want to take a walk. For example, let's say you have free time at the train station or a free evening at the hotel. Thus, this is a case to get around the sights near you, the route around the area.

The idea of solution is quite simple: we can get some sights around the user, assign some score to each place, depending on information about it (description in tags, etc.), user preferences, and distance to the place. Then we can use a simple recursive greedy algorithm: for each place, we calculate a weight sum function that takes into account these details as terms with some weights and then we can just take the places with the highest score, while the time limit or the points end.

That was the idea, let's look at the details. Considering that we have relatively few points, we can count the $durationsMatrix$ for $O(n^2)$. It contains time to get from point i to point j (in seconds). $durationsMatrix$ is obtained by MapBox Matrix API.

For each point we can set a $score$, depending on the user's preferences, the point's interest, rating, etc. In this work we can just set $score = 1$ to all the points.

We then compute a $weightMatrix$ where the elements are marks if the user moves from point i to point j. We calculate this matrix in different ways. In this paper we calculate it as

$$weightMatrix[i][j] = score[j] * weightScore - weightTime * durationsMatrix[i][j]$$

Where $weightScore$ and $weightTime$ are the weights for scores and time (in this work were 10000 and 1 respectively). Thus, a point gets more points the higher its score and the less time it takes to get to it.

Therefore, for the point i where the user is, we can select the next point by doing the following:

1. Choose $\arg\max_j weightMatrix[i][j]$ as possible next point.
2. Check

a. If the point j was not visited before.

b. If $durationsMatrix[i][j] + time \leq timeLimit$,

Where time is the elapsed time and the time limit was previously set.

3. If this is true, then the user can visit point j, so we should

a. add $durationsMatrix[i][j]$ to $time$

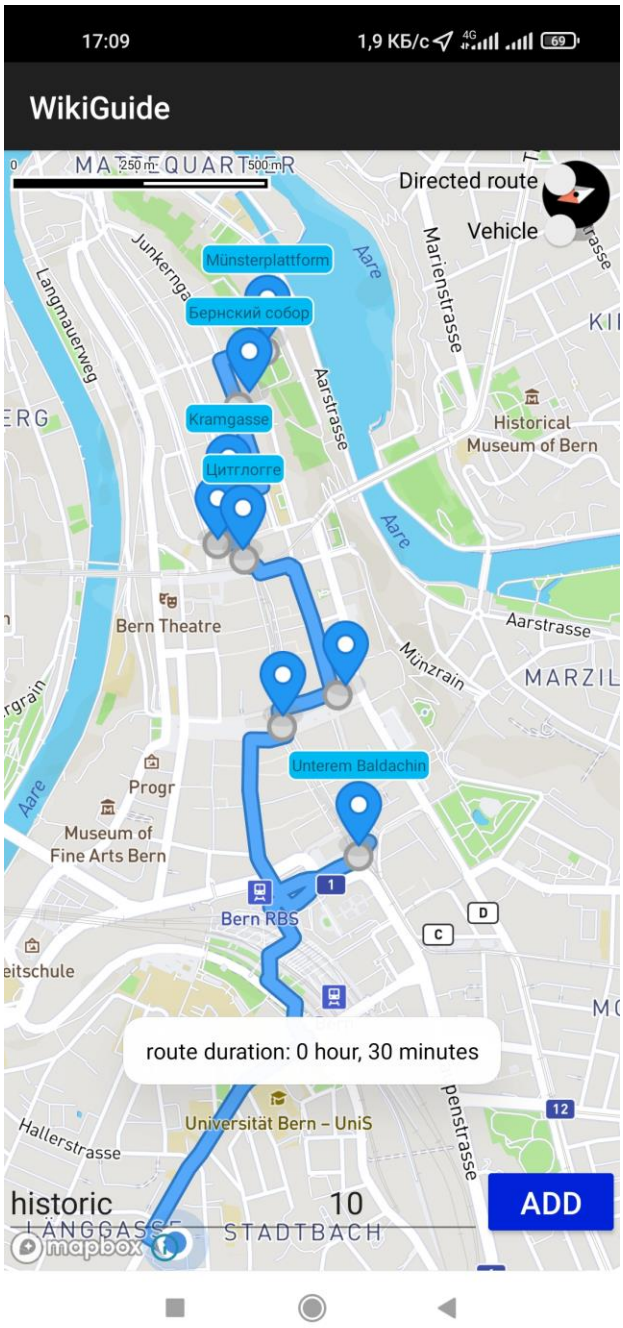
b. Mark j as visited point

c. change point i to point j.

Otherwise, we should exclude j from the possible points to visit for point i and return to point 1.

4. And we can try this as long as there are possible points to visit from point i.

Thus, we can get a list of points to visit in order. With MapBox Navigation API (which plots a short route between given points in their order) we can build a route for them (implemented).



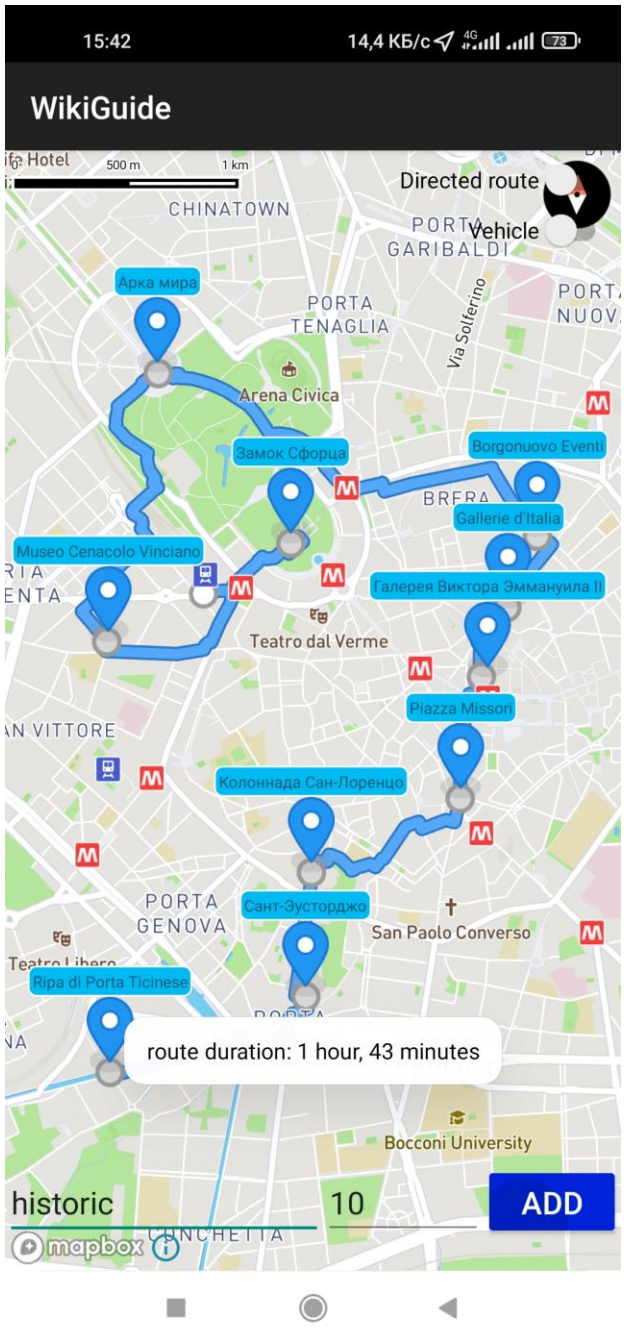


Figure 5.1.1 Figure 5.1.2

In figures 5.1.1 and 5.1.2 you can see examples of results. You can see a straight line-like route in Figure 5.1.2 and a circular-like route in Figure 5.1.2.

In this way, we can create rather interesting routes for nice walks even in this fairly simple way.

Case of a directed route

Another case is when you need to get from one point to another, but you have free time and would like to visit some attractive places and sights along the way.

The solution is based on the rather simple idea too. We can add points of interest as long as they are on our path and not too far from the shortest route. The shortest route between two points is a line. So, to approximate a route, we can select it. Of course, there are times when this is not entirely true, but for many situations, an angled line, similar to a straight one, can represent the real shortest route.

So, the algorithm is also quite simple. First, after getting a list of points of interest (places could be nice to visit, such as sights, nature such as parks, rivers, etc.) in the area, in order to determine which points we should include in the new route, we draw a line from the start point to the end point and calculate the distance from the point to the line. If the distance is less than a given threshold (and after other checks, such as whether the point is behind the destination point or it is before the start point, or whether the point is interesting enough for user), we can add the point.

It was about the idea. Let's look at the details.

To check if the point is interesting enough for the user, we can use the same idea as in the previous case: set the score in the same way:

$$score[i] = weightScore - weightDistance * distancesToLine[i]$$

and if it is greater than a $scoreThreshold$, we can select the point.

$weightDistance$ is the selected weight for distance, and $distancesToLine[i]$ is the distance to the line between the start and end points from point i.

To calculate distances, the coordinates of a point from the latitude-longitude form can be represented as coordinates in a two-dimensional coordinate system on a plane. The center of this system can be at $(0,0)°$, but we can move it to the user's current location - this way the coordinates of the places will be smaller numbers, which may be better for calculations.

To do this, we need to find out how many meters are in one degree of latitude and longitude.

For longitude: the circumference is constant, 40008548 meters. Dividing by 360° we get

$$111134.855556\ \frac{m}{deg} = MPD,$$

Where MPD is “meters per degree” for latitude.

For latitude the length is different: for equator it is 40075696 meters, for poles it is zero.

So we can calculate the length of one degree as the length of one degree along the equator multiplied by the cosine of the current latitude. One degree at the equator is

$$\frac{40075696\ m}{360°} = 111321.377778\frac{m}{deg} = MPDE,$$

Where $MPDE$ is “meters per degree at equator” for longitude.

Therefore, denoting

(O_{lat}, O_{lon}) −coordinates for current location in degrees.

(P_{lat}, P_{lon}) − coordinates for point coordinates in degrees.

(P_x, P_y) − coordinates for point in new coordinate system.

$degToRad(x) = x * \frac{\pi}{180}$ − function to convert degrees to radians.

We can write formulas for points in the new coordinate system:

$$P_x = (P_{lat} - O_{lat}) * MPL$$

$$P_y = (P_{lon} - O_{lon}) * \cos(\text{deg}ToRad(P_{lon})) * MPDE$$

Thus, we get new coordinates as an indicator of how far this point is from the origin in meters.

Denote

$A = (X_0, Y_0)$ −Origin in new coordinate system (user′s current location),

$B = (X_1, Y_1)$ − User destination coordinates.

In order not to lag behind the destination point and not to go in the opposite direction, we should discard the points (P_x, P_y) whose projection onto the line AB does not belong to the segment AB.

To do this, we can calculate the dot product of the vectors $\overrightarrow{AP}$, $\overrightarrow{AB}$,and of $\overrightarrow{PB}, \overrightarrow{AB}$.

As is known, the dot product of vectors $\vec{a}, \vec{b}$ can be calculated as $|\vec{a}|\left|\vec{b}\right| \cos(\vec{a}, \vec{b})$.

Hence, its sign depends only on the sign of $\cos(\vec{a}, \vec{b})$. And, considering angels between 0°to 180° it is positive if $(\widehat{\vec{a}, \vec{b}}) \in (0°, 90°)$ and it is negative if $(\widehat{\vec{a}, \vec{b}}) \in (90°, 180°)$.

If the projection of P onto AB is between A and B, then $(\widehat{\overrightarrow{AB}, \overrightarrow{AP}}) \in (0°, 90°)$ (this means P isn't behind A) and $(\widehat{\overrightarrow{AB}, \overrightarrow{BP}}) \in (0°, 90°)$ (this means P isn't behind B).

Here is an illustration. As we can see, when $\angle PAB$ is an acute angle (Figure 5.2.1), the projection to AB is between points A and B. Otherwise, when $\angle PAB$ is an obtuse angle (Figure 5.2.2), the projection is not.

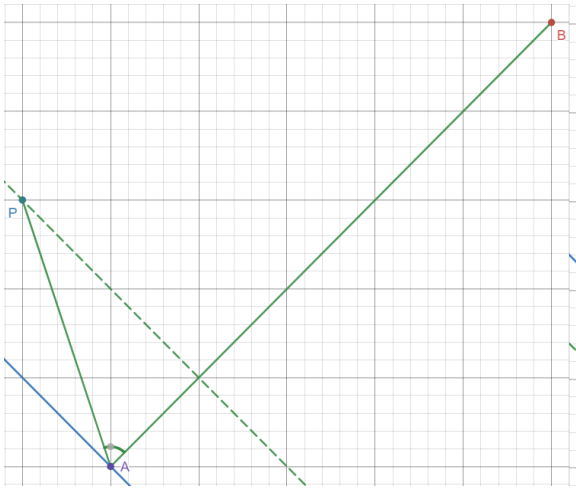


Figure 5.2.1

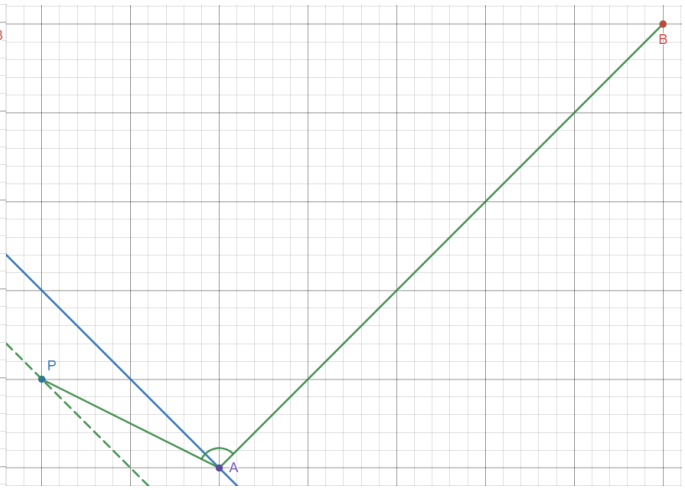


Figure 5.2.2

Therefore, to take correct points, we simply have to check that both of dot products $(\overrightarrow{AP}, \overrightarrow{AB})$ and $(\overrightarrow{PB}, \overrightarrow{AB})$ are positive. In this case, the degree between these vectors will be between $(0°, 90°)$, (because the cosine in the dot product will be positive) and the projection of point P onto line AB will be between points A and B, that is, it is possible to include it in user's route.

This way we can filter the points to only visit those that are in some area between the start and end points.

To check if a point is not too far from the shortest route (we are approximating it with a line), we calculate the distance between the point and the line.

The distance can be calculated as

$$d = \frac{\left|A \cdot P_x + B \cdot P_y + C\right|}{\sqrt{A^2 + B^2}},$$

Where

$$A = Y_1 - Y_0$$

$$B = -(X_1 - X_0)$$

$$C = -(A \cdot X_0 + B \cdot Y_0)$$

Thus, we can get points that are not far from the shortest route (we can check if d is less than some given $threshold$) and are on the user's path (the projection to the line is between A and B).

This not very intricate idea can give rather nice results for many situations.

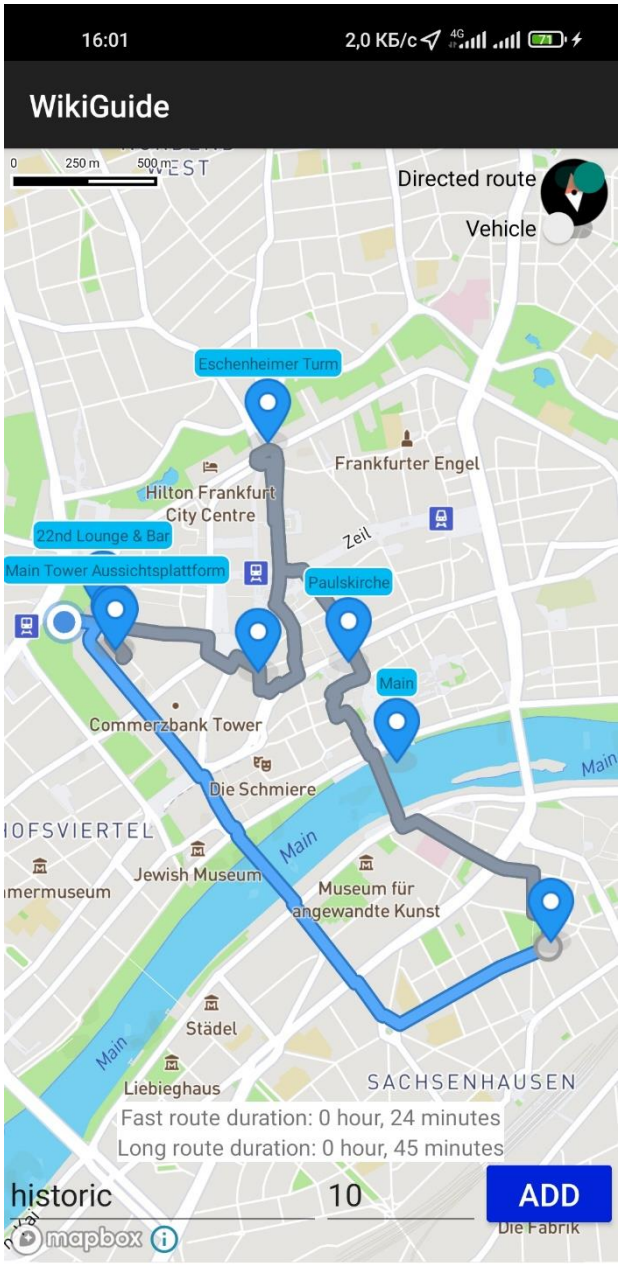


Figure 5.2.3

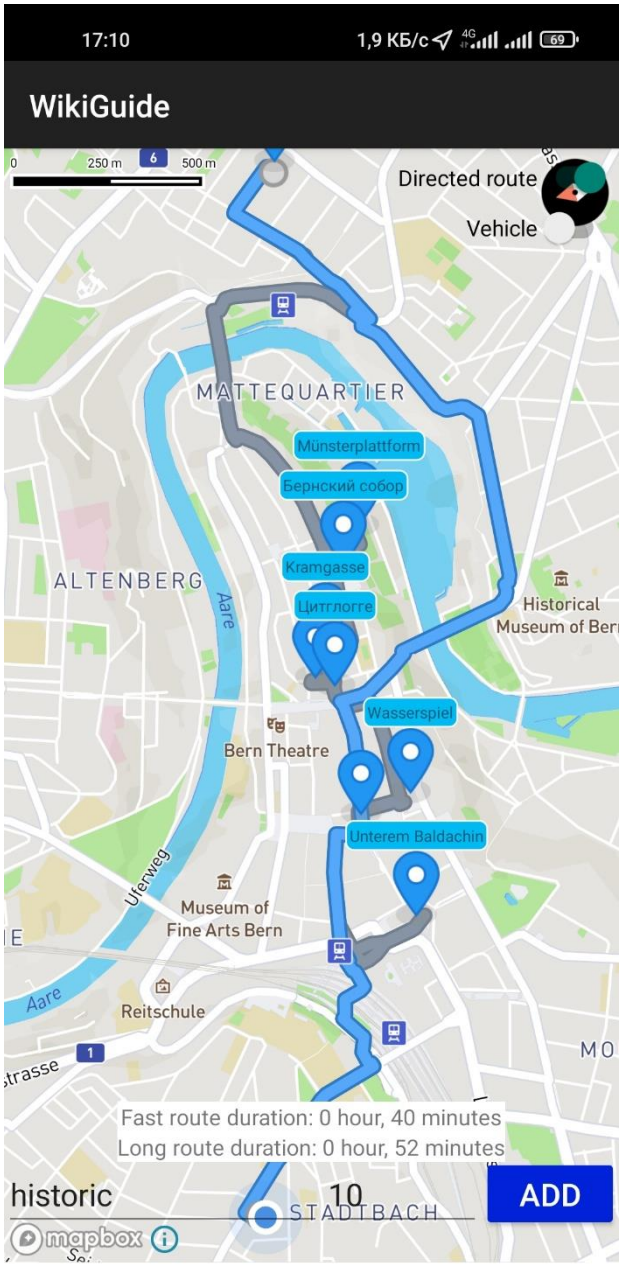


Figure 5.2.4

In Figure 5.2.3 we see an optional route that is slightly longer (and has an optional branch) than the faster one, but has some historical sites. Figure 5.2.4 shows several path options so that the user can select the places he would like to visit without straying too far from the shortest route. It is also slightly longer, but allows to visit some interesting places.

So, even in this rather simple and easy way we can make an ordinary reaching the destination more interesting and attractive.

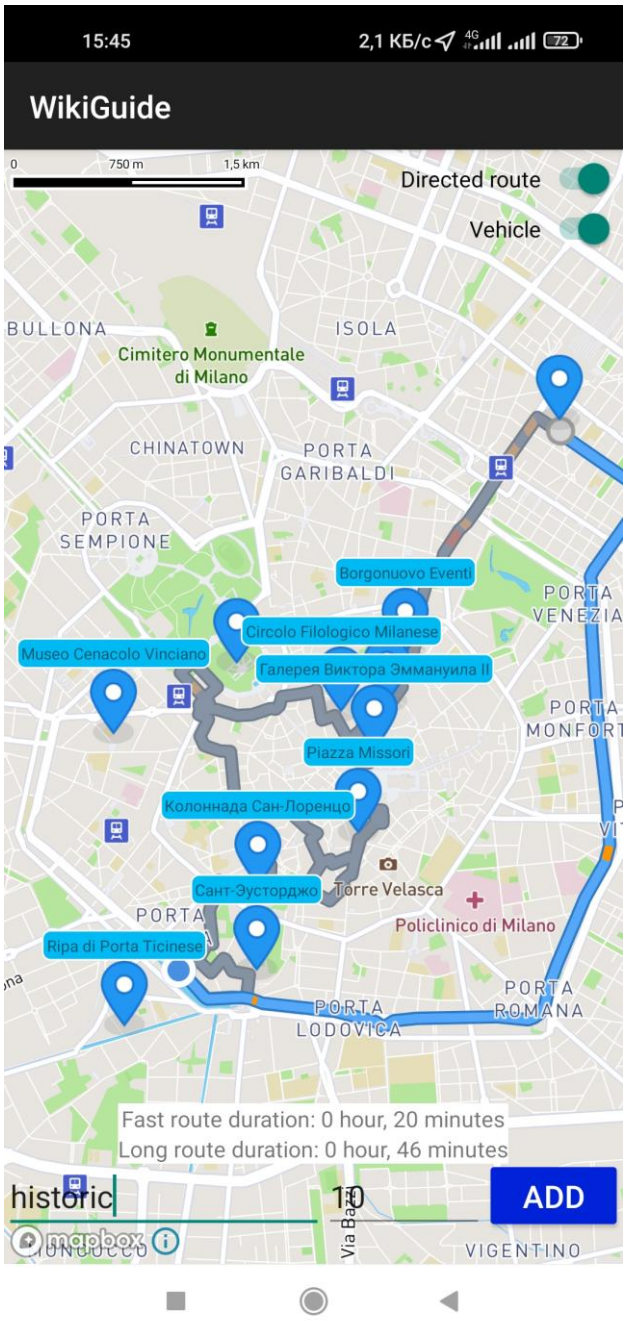
Vehicle case

We can extend the solutions of both cases to the vehicle case. For a better expansion, we took into account some features, such as traffic (to avoid traffic jams) and road directions. Moreover, in the new case, we can use both directed and circular trips solutions without even changing the algorithms (but we should calculate time and the route for a car, not for a pedestrian), so they can be quite universal.

For the case, when you are driving from one point to another, the solution is based on the same idea that if the interesting place is near the shortest route, the user can visit it. The decision function can be again a weighted sum of terms that take into account the distance from the point to the line, its attractiveness to the user, and so on.

The solution for the case where you want to tour the area can be the same as for the walking case (given a different way of calculating time and route from one point to another).

There are some examples.



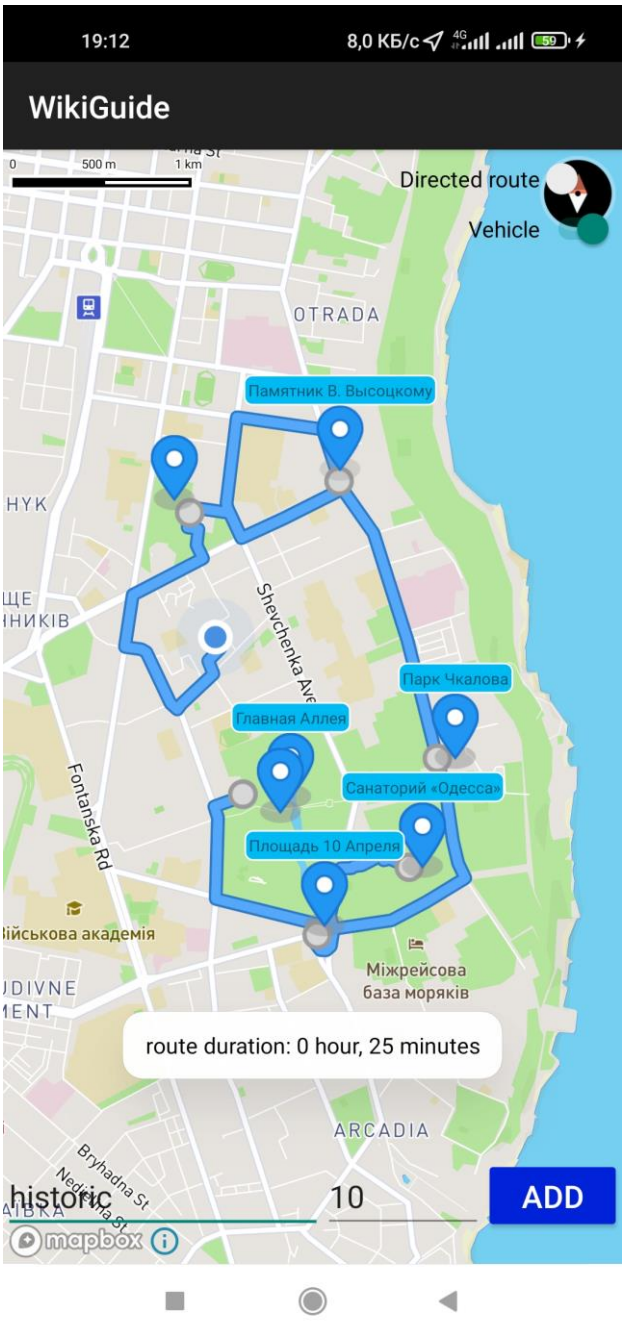


Figure 5.3.1 Figure 5.3.2

In Figure 5.3.1 you see a fast route and an additional route that leads to some interesting and historical sites. So, if you are passing through the city, you can visit the historical center. There are traffic jams on both routes but, as far as we can tell, but they don't seem too bad. In Figure 5.3.2 you can see the route around the current position, which can take you to several interesting places.

Therefore, as we can see, we can get pretty good results. And we didn’t even have to change the algorithms compared to the walking case, just take to attention some peculiarities.

Possible extensions

Of course, there may be better decision ideas (e.g. decide whether or not to include each point separately or consider them all at decision time), e.g. dynamic programming, graph algorithms (including solutions compatible to the current implementation. For example, for the Traveling Salesperson Problem there is MapBox Optimisation API) and so on. And another way to develop can be scenarios for long trips, different transport, take into account other details, such as the time spent at some points, weather, place rating, etc.

CONCLUSION

So, as mentioned, the task is rather empiric and there can be many solutions. But even quite simple ideas like this can give rather nice results.

In this work, we have considered several rather simple, but quite effective ideas that can solve the tasks: the circular route algorithm selects the nearest places, which can be good, because the user can see interesting places and benefit from the application in the shortest time of use. And for a directed route, the algorithm can suggest visiting places that are not far from the shortest route and suggest alternatives. Both algorithms can be used for a vehicle trip: it just had to take into account some features, for example, road directions, traffic. So the suggested ideas can solve this problem, but as mentioned, there may be other better solutions.

ВИСНОВОК

Отже, як зазначалося, завдання досить емпіричне, і рішень може бути багато. Але навіть такі прості ідеї можуть дати досить непогані результати.

У цій роботі ми розглянули кілька досить простих, але досить ефективних ідей, які можуть вирішити поставлені завдання: алгоритм кругового маршруту вибирає найближчі місця, що може бути добре, оскільки користувач може побачити цікаві місця і отримати вигоду від програми в найкоротший час використання. А для спрямованого маршруту алгоритм може запропонувати відвідування місць, які знаходяться неподалік від найкоротшого маршруту, і запропонувати альтернативи. Обидва алгоритми можна використовувати для поїздки на транспортному засобі: потрібно лише врахувати деякі особливості, наприклад, напрямки доріг, рух. Отже, запропоновані ідеї можуть вирішувати це питання, але, як згадувалося, можуть бути й інші кращі рішення.

LITERATURE

1. MapBox Matrix API documentation — 2022—Resource access mode: https://docs.mapbox.com/api/navigation/matrix/
2. MapBox Optimization API documentation — 2022—Resource access mode: https://docs.mapbox.com/api/navigation/optimization/
3. MapBox Android documentation —2022— Resource access mode: https://docs.mapbox.com/android/maps/guides/
4. Korlin for Android documentation —2022— Resource access mode: https://kotlinlang.org/docs/android-overview.html
5. Egor Smirnov, ITMO University — Walking around the city wisely: how I made a service for building interesting walking routes —2018— Resource access mode: https://habr.com/ru/post/414433/
6. Inspirock service — 2022— Resource access mode: https://www.inspirock.com/
7. Longitude, latitude degrees to kilometers — 2022— Resource access mode: https://v-ipc.ru/guides/coord#:~:text=Тут%20все%20просто%3A%20длина%20окружности,8707947531%20м
8. State, achievements and prospects of information systems and technologies: Collection of works of XXII All-Ukrainian scientific and technical conference of young scientists graduate students and students — 2022—pp 11-12— Resource access mode: https://www.onaft.edu.ua/download/konfi/2022/Conference_abstract-IT-21-22-04-22.pdf
9. Informatics, information systems and technologies: Collection of works of XIX all-ukrainian conference of students and young scientists —2022— pp 53-54.
10. Computer Intelligence Software: proceedings of Huaiyin Institute of Technology-Odessa I.I. Mechnikov National University conference— 2022.

11. Implementation link— 2022— Resource access mode: https://github.com/gorynalexander/WikiGuide_v2

ADDITION A

Code for MapFragment.kt:

```
package com.santo.wikiguide.UI.map

import android.annotation.SuppressLint
import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import android.widget.Toast
import androidx.asynclayoutinflater.view.AsyncLayoutInflater
import androidx.fragment.app.Fragment
import androidx.fragment.app.viewModels
import com.mapbox.android.gestures.MoveGestureDetector
import com.mapbox.api.directions.v5.DirectionsCriteria
import com.mapbox.api.directions.v5.models.DirectionsRoute
import com.mapbox.api.directions.v5.models.RouteOptions
import com.mapbox.bindgen.Expected
import com.mapbox.geojson.Feature
import com.mapbox.geojson.FeatureCollection
import com.mapbox.geojson.Point
import com.mapbox.maps.*
import com.mapbox.maps.extension.style.image.image
import com.mapbox.maps.extension.style.layers.generated.symbolLayer
import com.mapbox.maps.extension.style.layers.properties.generated.IconAnchor
import com.mapbox.maps.extension.style.sources.generated.GeoJsonSource
import com.mapbox.maps.extension.style.sources.generated.geoJsonSource
import com.mapbox.maps.extension.style.sources.getSourceAs
import com.mapbox.maps.extension.style.style
import com.mapbox.maps.plugin.gestures.*
import
com.mapbox.maps.plugin.locationcomponent.OnIndicatorPositionChangedListene
r
import com.mapbox.maps.plugin.locationcomponent.location
import com.mapbox.maps.viewannotation.ViewAnnotationManager
import com.mapbox.maps.viewannotation.viewAnnotationOptions
import com.mapbox.navigation.base.extensions.applyDefaultNavigationOptions
import
com.mapbox.navigation.base.extensions.applyLanguageAndVoiceUnitOptions
import com.mapbox.navigation.base.options.NavigationOptions
import com.mapbox.navigation.base.route.RouterCallback
```

```
import com.mapbox.navigation.base.route.RouterFailure
import com.mapbox.navigation.base.route.RouterOrigin
import com.mapbox.navigation.core.MapboxNavigation
import com.mapbox.navigation.core.MapboxNavigationProvider
import com.mapbox.navigation.core.directions.session.RoutesObserver
import com.mapbox.navigation.core.replay.MapboxReplayer
import com.mapbox.navigation.core.replay.ReplayLocationEngine
import com.mapbox.navigation.ui.maps.route.arrow.api.MapboxRouteArrowApi
import com.mapbox.navigation.ui.maps.route.arrow.api.MapboxRouteArrowView
import com.mapbox.navigation.ui.maps.route.arrow.model.RouteArrowOptions
import com.mapbox.navigation.ui.maps.route.line.api.MapboxRouteLineApi
import com.mapbox.navigation.ui.maps.route.line.api.MapboxRouteLineView
import
com.mapbox.navigation.ui.maps.route.line.model.MapboxRouteLineOptions
import com.mapbox.navigation.ui.maps.route.line.model.RouteLine
import com.mapbox.search.result.SearchResult
import com.santo.wikiguide.R
import com.santo.wikiguide.data.routerBuilder.RouteCircleGreedy
import com.santo.wikiguide.data.routerBuilder.RouteLineGreedy
import com.santo.wikiguide.databinding.FragmentMapBinding
import dagger.hilt.android.AndroidEntryPoint
import timber.log.Timber
import java.util.concurrent.CopyOnWriteArrayList
import kotlin.math.roundToInt


@MapboxExperimental
@AndroidEntryPoint
class MapFragment : Fragment(),OnMapClickListener  {

    private lateinit var binding: FragmentMapBinding
    private lateinit var mapView: MapView
    private lateinit var mapboxMap: MapboxMap
    private lateinit var viewAnnotationManager: ViewAnnotationManager


    private lateinit var mapboxNavigation: MapboxNavigation
    private val replayLocationEngine = ReplayLocationEngine(MapboxReplayer())
    private lateinit var
onIndicatorPositionChangedListener:OnIndicatorPositionChangedListener

    private val viewModel: MapFragmentViewModel by viewModels()
    private val points:ArrayList<Point> =ArrayList()
```

```
    private var markerWidth = 0
    private var markerHeight = 0
    private val asyncInflater by lazy { context?.let { AsyncLayoutInflater(it) } }

    private lateinit var route:List<Point>
    private var durationWalk=ArrayList<Double>()
    var I=0

    var firstUpdate=true

    private lateinit var destinationPoint: Point

    private var circleRoute=true
    private var walking=true

    private val durationLimit=7200.0
    private val distanceTreshold=1000.0

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        // Inflate the layout for this fragment
        binding = FragmentMapBinding.inflate(inflater)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        initNavigation()
        initMap()

        binding.inputCategoryName.setText("historic")
        binding.inputCategoryLimit.setText("10")

        binding.addButton.setOnClickListener {

viewModel.getPOIs(binding.inputCategoryName.text.toString(),binding.inputCate
goryLimit.text.toString().toInt())
        }
    }



    private fun initMap(){
```

```
        mapView = binding.mapView
        viewAnnotationManager = binding.mapView.viewAnnotationManager


        var bitmap = BitmapFactory.decodeResource(resources,
R.drawable.blue_marker)
        bitmap=
Bitmap.createScaledBitmap(bitmap,bitmap.width/2,bitmap.height/2,false)
        markerWidth = bitmap.width
        markerHeight = bitmap.height
        mapboxMap = binding.mapView.getMapboxMap().apply {
            loadStyle(
                styleExtension = prepareStyle(Style.MAPBOX_STREETS, bitmap)
            ) {

                mapView.location.updateSettings {
                    enabled = true
                    pulsingEnabled = true
                }
                addOnMapClickListener(this@MapFragment)
//                addOnMapLongClickListener(this@MapFragment)
//                Toast.makeText(this@MapFragment, STARTUP_TEXT,
Toast.LENGTH_LONG).show()
            }
            binding.directedSwitch.setOnCheckedChangeListener {
                view,isChecked->
                    circleRoute = !isChecked
            }
            binding.vehicleSwitch.setOnCheckedChangeListener{
                view,isChecked->
                    walking=!isChecked
            }
        }

//        addOnMapClickListener(this@MapFragment)
//        addOnMapLongClickListener(this@MapFragment)



        // Get the user's location as coordinates
//      Round's screen to the direction of user's look
//      Problem: always round back

//        val onIndicatorBearingChangedListener =
OnIndicatorBearingChangedListener {
```

```
//
mapView.getMapboxMap().setCamera(CameraOptions.Builder().bearing(it).build
())
//      }

        onIndicatorPositionChangedListener = OnIndicatorPositionChangedListener {

mapView.getMapboxMap().setCamera(CameraOptions.Builder().center(it).build())
            mapView.gestures.focalPoint =
mapView.getMapboxMap().pixelForCoordinate(it)
            viewModel.currentLocation=it
            if(firstUpdate){
                firstUpdate=false
                addObserver()
            }
        }

// Pass the user's location to camera

mapView.location.addOnIndicatorPositionChangedListener(onIndicatorPositionCh
angedListener)
//
mapView.location.addOnIndicatorBearingChangedListener(onIndicatorBearingC
hangedListener)


        mapView.gestures.addOnMoveListener(object : OnMoveListener {
            override fun onMoveBegin(detector: MoveGestureDetector) {
                mapView.location

.removeOnIndicatorPositionChangedListener(onIndicatorPositionChangedListener
)
            }

            override fun onMove(detector: MoveGestureDetector): Boolean {
                return false
            }

            override fun onMoveEnd(detector: MoveGestureDetector) {}
        })

        mapView.gestures.addOnMapLongClickListener{
            if(points.size>1) {
//                FOR CIRCLE ROUTE
//                val routeBuilder= RouteCircleGreedyBuilder()
```

```
//            routeBuilder.getRoute(7200.0, startPoint = viewModel.currentLocation
,points){
//                result->
//              route=result.first
//              durationWalk=result.second
//            }
//              #FOR_LINE_ROUTE
//   TODO Fix bug: need first initialize destination point, then add points
            if (circleRoute) {
                val routeBuilder = RouteCircleGreedy()
                routeBuilder.getRoute(
                    durationLimit,
                    viewModel.currentLocation,
                    points, getProfile()
                ) { result ->
                    route = result
                    drawRoute(route)
//                  durationWalk=result
                }
            } else {
                if (this::destinationPoint.isInitialized) {
                    val routeBuilder = RouteLineGreedy()
                    routeBuilder.getRoute(
                        distanceTreshold,
                        viewModel.currentLocation,
                        destinationPoint,
                        points
                    ) { result ->
                        route = result
                        drawRoute(listOf(viewModel.currentLocation,destinationPoint))
                        drawRoute(route)
//                      durationWalk=result
                    }
                } else {
                    Toast.makeText(
                        requireContext(),
                        "Add a destination point",
                        Toast.LENGTH_LONG
                    ).show()
                }
            }
        }
        true
    }
}
```

```
    fun addObserver(){
        viewModel.poiList.observe(viewLifecycleOwner) { poiList ->
            points.clear()
            for (item in poiList) {
//                item.coordinate?.let { addAnnotationToMap(it.longitude(), it.latitude())
}
                Timber.i("Item name: ${item.name}; and address: ${item.address}")
                val markerId = addMarkerAndReturnId(item.coordinate!!)
                addViewAnnotation(item, markerId)
                points.add(item.coordinate!!)
            }

//                Timber.i(points.toString())

        }
    }
    override fun onDestroyView() {

mapView.location.removeOnIndicatorPositionChangedListener(onIndicatorPositio
nChangedListener)
        super.onDestroyView()
    }


//    override fun onMapLongClick(point: Point): Boolean {
//        val markerId = addMarkerAndReturnId(point)
//        addViewAnnotation(point, markerId)
//        return true
//    }


// Markers and annotations section
    private val pointList = CopyOnWriteArrayList<Feature>()
    private var markerId = 0
    override fun onMapClick(point: Point): Boolean {
        destinationPoint=point
        addMarkerAndReturnId(destinationPoint)
//        TODO Remove marker
        mapboxMap.queryRenderedFeatures(
            RenderedQueryGeometry(mapboxMap.pixelForCoordinate(point)),
RenderedQueryOptions(listOf(LAYER_ID), null)
        ) {
            onFeatureClicked(it) { feature ->
                if (feature.id() != null) {
```

```
viewAnnotationManager.getViewAnnotationByFeatureId(feature.id()!!)?.toggleVi
ewVisibility()
                }
            }
        }
        return true
    }


    private fun prepareStyle(styleUri: String, bitmap: Bitmap) = style(styleUri) {
        +image(BLUE_ICON_ID) {
            bitmap(bitmap)
        }

        +geoJsonSource(SOURCE_ID) {
            featureCollection(FeatureCollection.fromFeatures(pointList))
        }

        +symbolLayer(LAYER_ID, SOURCE_ID) {
            iconImage(BLUE_ICON_ID)
            iconAnchor(IconAnchor.BOTTOM)
            iconAllowOverlap(true)
        }
    }


    private fun onFeatureClicked(
        expected: Expected<String, List<QueriedFeature>>,
        onFeatureClicked: (Feature) -> Unit
    ) {
        if (expected.isValue && expected.value?.size!! > 0) {
            expected.value?.get(0)?.feature?.let { feature ->
                onFeatureClicked.invoke(feature)
            }
        }
    }

    private fun View.toggleViewVisibility() {
        visibility = if (visibility == View.VISIBLE) View.GONE else View.VISIBLE
    }

    private fun addMarkerAndReturnId(point: Point): String {
        val currentId = "${MARKER_ID_PREFIX}${(markerId++)}"
        pointList.add(Feature.fromGeometry(point, null, currentId))
```

```
        val featureCollection = FeatureCollection.fromFeatures(pointList)
        mapboxMap.getStyle { style ->

style.getSourceAs<GeoJsonSource>(SOURCE_ID)?.featureCollection(featureColl
ection)
        }
        return currentId
    }

    @SuppressLint("SetTextI18n")
    private fun addViewAnnotation(searchResult: SearchResult, markerId: String) {
        asyncInflater?.let {
            viewAnnotationManager.addViewAnnotation(
                resId = R.layout.annotation_view,
                options = viewAnnotationOptions {
                    geometry(searchResult.coordinate)
                    associatedFeatureId(markerId)
                    anchor(ViewAnnotationAnchor.BOTTOM)
                    allowOverlap(false)
                },
                asyncInflater = it
            ) { viewAnnotation ->
                // calculate offsetY manually taking into account icon height only
because of bottom anchoring
                viewAnnotationManager.updateViewAnnotation(
                    viewAnnotation,
                    viewAnnotationOptions {
                        offsetY(markerHeight)
                    }
                )
                viewAnnotation.findViewById<TextView>(R.id.textNativeView).text
=when(searchResult.name){
                    "null"-
>"lat=%.2f\nlon=%.2f".format(searchResult.coordinate?.latitude(),
searchResult.coordinate?.longitude())
                    else-> searchResult.name
                }
            }
        }
    }

    private companion object {
        const val BLUE_ICON_ID = "blue"
        const val SOURCE_ID = "source_id"
        const val LAYER_ID = "layer_id"
```

```
        const val MARKER_ID_PREFIX = "view_annotation_"
    }




    private lateinit var routeLineApi: MapboxRouteLineApi
    private lateinit var routeLineView: MapboxRouteLineView
    private val routeArrowApi: MapboxRouteArrowApi =
MapboxRouteArrowApi()
    private lateinit var routeArrowView: MapboxRouteArrowView

    private fun initNavigation() {
        mapboxNavigation = if (MapboxNavigationProvider.isCreated()) {
            MapboxNavigationProvider.retrieve()
        } else {
            MapboxNavigationProvider.create(
                NavigationOptions.Builder(requireContext())
                    .accessToken(getString(R.string.mapbox_access_token))
                    // comment out the location engine setting block to disable simulation
                    .locationEngine(replayLocationEngine)
                    .build()
            )
        }

        val mapboxRouteLineOptions =
MapboxRouteLineOptions.Builder(requireContext())
            .withRouteLineBelowLayerId("road-label")
            .build()
        routeLineApi = MapboxRouteLineApi(mapboxRouteLineOptions)
        routeLineView = MapboxRouteLineView(mapboxRouteLineOptions)

        val routeArrowOptions =
RouteArrowOptions.Builder(requireContext()).build()
        routeArrowView = MapboxRouteArrowView(routeArrowOptions)
        mapboxNavigation.registerRoutesObserver(routesObserver)
    }


    private val routesObserver = RoutesObserver { routeUpdateResult ->
        if (routeUpdateResult.routes.isNotEmpty()) {
            // generate route geometries asynchronously and render them
            val routeLines = routeUpdateResult.routes.map { RouteLine(it, null) }
```

```
            routeLineApi.setRoutes(
                routeLines
            ) { value ->
                mapboxMap.getStyle()?.apply {
                    routeLineView.renderRouteDrawData(this, value)
                }
            }

            // update the camera position to account for the new route
        } else {
            // remove the route line and route arrow from the map
            val style = mapboxMap.getStyle()
            if (style != null) {
                routeLineApi.clearRouteLine { value ->
                    routeLineView.renderClearRouteLineValue(
                        style,
                        value
                    )
                }
                routeArrowView.render(style, routeArrowApi.clearArrows())
            }

            // remove the route reference from camera position evaluations
        }
    }



    private fun drawRoute(points: List<Point>) {

        // execute a route request
        // it's recommended to use the
        // applyDefaultNavigationOptions and applyLanguageAndVoiceUnitOptions
        // that make sure the route request is optimized
        // to allow for support of all of the Navigation SDK features

        val profile = getProfile()

        mapboxNavigation.requestRoutes(
            RouteOptions.builder()
                .applyDefaultNavigationOptions()
                .applyLanguageAndVoiceUnitOptions(requireContext())
                .coordinatesList(points)
                .profile(profile)
                .overview(DirectionsCriteria.OVERVIEW_FULL)
```

```
            .build(),
        object : RouterCallback {
            override fun onRoutesReady(
                routes: List<DirectionsRoute>,
                routerOrigin: RouterOrigin
            ) {
//                #FOR_LINE_ROUTE
//                durationWalk=0.0
                durationWalk.add(0.0)
                for(route in routes){
                    durationWalk[I]+=route.duration()
                }
//                setRouteAndStartNavigation(routes)
                mapboxNavigation.setRoutes(
                    mapboxNavigation.getRoutes().plus(routes))
//                Toast.makeText(requireContext(),
//                    "Route duration: ${durationWalk.roundToInt()/3600} hour," +
//                            " ${(durationWalk.roundToInt()/60)%60}
minutes",Toast.LENGTH_LONG).show()

                if(circleRoute){
                    Toast.makeText(requireContext(),

getRouteDurationString(durationWalk[0]),Toast.LENGTH_LONG).show()
                }

                else{

                    if(I==1) {

                        val inflater = layoutInflater
                        val layout = inflater.inflate(
                            R.layout.custom_toast,
                            view?.findViewById(R.id.toast_layout_root) as ViewGroup?
                        )
                        layout.findViewById<TextView>(R.id.text_duration_short).text
=
                            "Fast "+getRouteDurationString(durationWalk[0])
                        layout.findViewById<TextView>(R.id.text_duration_long).text
=
                            "Long "+getRouteDurationString(durationWalk[1])
                        val toast =  Toast(requireContext());
                        //                toast.setGravity(Gravity.CENTER_VERTICAL, 0,
0);
                        toast.duration = Toast.LENGTH_LONG;
```

```
                    toast.view = layout;
                    toast.show();
                }
                I=(I+1)%2
            }
            if(I==0)
                durationWalk=ArrayList<Double>()
        }

        override fun onFailure(
            reasons: List<RouterFailure>,
            routeOptions: RouteOptions
        ) {
            // no impl
        }

        override fun onCanceled(routeOptions: RouteOptions, routerOrigin:
RouterOrigin) {
            // no impl
        }
    }
  )
}

private fun getProfile(): String {
    return if(walking){
        DirectionsCriteria.PROFILE_WALKING
    } else{
        DirectionsCriteria.PROFILE_DRIVING_TRAFFIC
    }
}

fun getRouteDurationString(duration:Double):String{
    return "route duration: ${duration.roundToInt()/3600} hour," +
                    " ${(duration.roundToInt()/60)%60} minutes"
}
}
```

ADDITION B

Code for RouteCircleGreedy.kt:

```
package com.santo.wikiguide.data.routerBuilder

import com.mapbox.api.directions.v5.DirectionsCriteria
import com.mapbox.api.matrix.v1.MapboxMatrix
import com.mapbox.api.matrix.v1.models.MatrixResponse
import com.mapbox.geojson.Point
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response
import timber.log.Timber
import java.util.*
import kotlin.collections.ArrayList
import kotlin.collections.HashSet

class RouteCircleGreedy : RouteBuilder() {
    val weightScore=10000
    val weightTime=1

    lateinit var durationsMatrix: List<Array<Double>>

    override
    fun getRoute(timeLimit:Double, startPoint:Point, pointsToVisit:List<Point>,
profile:String, circleResultListener: routeResultListener){
        val points= pointsToVisit.plus(startPoint)
        Collections.reverse(points)
        getMatrix(points,profile){
            val result=getOrder(timeLimit,points)
            val route=ArrayList<Point>()
            for(i in 0 until result.first.size){
                route.add(points[result.first[i]])
            }
//            circleResultListener.onRouteResult(Pair(route,result.second))
            circleResultListener.onRouteResult(route)
        }
    }

    override
    fun getOrder(timeLimit:Double, points: List<Point>): Pair<ArrayList<Int>,
Double> {
        val N=points.size
        val scores=ArrayList<Double>()
//TODO change to POI
```

```
        for (point in points){
            scores.add(getScore(point))
        }
        val weightMatrix=ArrayList<ArrayList<Pair<Int,Double>>>()
        for (i in 0 until N){
            weightMatrix.add(ArrayList())
            for (j in 0 until N){
                if(i==j)
                    weightMatrix[i].add(Pair(j,-INF))
                else
                    weightMatrix[i].add(Pair(j,scores[j]*weightScore-
weightTime*durationsMatrix[i][j]))
            }
        }
//      Bubble sort
        for(i in 0 until N){
            for(j in 0 until N-1){
                for (k in 0 until N-j-1){
                    if(weightMatrix[i][k].second<weightMatrix[i][k+1].second) {
                        val tmp=weightMatrix[i][k]
                        weightMatrix[i][k]=weightMatrix[i][k+1]
                        weightMatrix[i][k+1]=tmp
                    }
                }
            }
        }
        val was=HashSet<Int>()
        val order=ArrayList<Int>()
        var time=0.0
        var I=0
        was.add(I)
        order.add(I)
        for(i in 0 until N){
            var j=0

while(j<N&&(was.contains(weightMatrix[I][j].first)||(durationsMatrix[I][weightM
atrix[I][j].first]+time>timeLimit)))
                j++
            if(j!=N){
                time += durationsMatrix[I][weightMatrix[I][j].first]
                order.add(weightMatrix[I][j].first)
                was.add(weightMatrix[I][j].first)
                I=weightMatrix[I][j].first
            }
            else
```

```
                break
            }
            return Pair(order,time)
        }
        fun getMatrix(points:List<Point>,profile:String, resultListener:
matrixResultListener){
//          TODO: change hardcode
            val matrixApiClient= MapboxMatrix.builder()

.accessToken("pk.eyJ1Ijoia3lyeWx2ZXJlbWlvdiIsImEiOiJjbDEyNGhvY2gwM25
1M2tzMDhlOHE1aXAxIn0.r_PHqSSTrolmC5HBShznQw")
                .coordinates(points)
                .profile(profile)
                .build()
            matrixApiClient.enqueueCall(object : Callback<MatrixResponse> {
                override fun onResponse(call: Call<MatrixResponse>, response:
Response<MatrixResponse>) {
                    Timber.i(response.body()?.toString())
                    durationsMatrix= response.body()?.durations()!!
                    Timber.i("Getting Matrix success")
                    resultListener.onMatrixResult(durationsMatrix)
                }

                override fun onFailure(call: Call<MatrixResponse>, throwable: Throwable)
{
                    Timber.i("Getting Matrix failed")
                }
            })
        }

        fun interface matrixResultListener{
            fun onMatrixResult(results: List<Array<Double>>)
        }

}
```

ADDITION C

Code for RouteLineGreedy.kt:

```
package com.santo.wikiguide.data.routerBuilder

import com.mapbox.geojson.Point
import java.util.*
import kotlin.collections.ArrayList
import kotlin.math.PI
import kotlin.math.abs
import kotlin.math.cos
import kotlin.math.sqrt

class RouteLineGreedy:RouteBuilder() {
    private val scoreTreshold=0
    private val weightScore = 10000
    private val weightDistance = 1
    private val mPerLongitudeDegree=111134.855556
    private val mPerLatitudeDegreeOnEquator=111321.377778

    var N=0

/*distanceThreshold in meters*/
    override
    fun getRoute(
    distanceThreshold: Double,
    startPoint: Point,
    endPoint: Point,
    points: List<Point>,
    resultListener: routeResultListener
    ) {
        N = points.size
        getDistances(startPoint,endPoint,points){
            distances->
            val resultOrder = getOrder(distanceThreshold,distances, points)
            val route = ArrayList<Point>()
            route.add(startPoint)
            for (i in 0 until resultOrder.size) {
                route.add(points[resultOrder[i]])
            }
            route.add(endPoint)
            resultListener.onRouteResult(route)
        }
    }
```

```
    override
    fun getOrder(distanceThreshold: Double,
distances:Pair<List<Double>,List<Double>>, points: List<Point>):
ArrayList<Int> {
        val scores = ArrayList<Double>()
//TODO change to POI
        for (point in points) {
            scores.add(getScore(point))
        }
        val distancesToLine=distances.first
        val distancesToStart=distances.second
        for(i in 0 until scores.size){
            if(distancesToLine[i]>distanceThreshold || distancesToLine[i]==INF){
                scores[i]=-INF
            }
            else{
                scores[i]=scores[i]*weightScore-weightDistance*distancesToLine[i]
            }
        }
        val chosenPoints = ArrayList<Int>()

        for(i in 0 until scores.size){
            val maxScore= Collections.max(scores)
            if(maxScore==-INF || maxScore<scoreTreshold)
                break
            val maxIndex=scores.indexOf(maxScore)
            chosenPoints.add(maxIndex)
            scores[maxIndex]=-INF
        }
//        optimize(chosenPoints){}
        val order = ArrayList<Int>()
        val mutableDistancesToStart=ArrayList<Double>()
        for(i in 0 until N) {
            mutableDistancesToStart.add(distancesToStart[i])
        }
        for(i in 0 until N) {
            val minDistance= Collections.min(mutableDistancesToStart)
            if(minDistance==INF)
                break
            val minIndex=mutableDistancesToStart.indexOf(minDistance)
            if(chosenPoints.contains(minIndex)){
                order.add(minIndex)
            }
            mutableDistancesToStart[minIndex]=INF
        }
```

```
        return order
    }

////      TODO: change hardcode
//    private fun optimize(points: List<Point>, optimizeListener:OptimizeListener) {
//        val optimizationApiClient= MapboxOptimization.builder()
//
.accessToken("pk.eyJ1Ijoia3lyeWx2ZXJlbWlvdiIsImEiOiJjbDEyNGhvY2gwM251
M2tzMDhlOHE1aXAxIn0.r_PHqSSTrolmC5HBShznQw")
//            .coordinates(points)
//            .profile(DirectionsCriteria.PROFILE_WALKING)
//            .build()
//        optimizationApiClient.enqueueCall(object :
Callback<OptimizationResponse>{
//            override fun onResponse(
//                call: Call<OptimizationResponse>,
//                response: Response<OptimizationResponse>
//            ) {
//
//                Timber.i("Getting optimized route success")
//                optimizeListener.onOptimizeResult(response.body()?.trips())
//            }
//
//            override fun onFailure(call: Call<OptimizationResponse>, t: Throwable)
{
//                Timber.i("Getting optimized route failed")
//            }
//
//        })
//    }

    fun getDistances(startPoint: Point,endPoint: Point,
        points: List<Point>,
        resultListener: DistancesResultListener
    ) {
        val distancesToLine=ArrayList<Double>()

        val Ox=startPoint.latitude()
        val Oy=startPoint.longitude()

        val X0=0
        val Y0=0
        val X1=(endPoint.latitude()-Ox)*mPerLongitudeDegree
        val Y1=(endPoint.longitude()-
Oy)*(cos(degreesToRad(endPoint.longitude()))*mPerLatitudeDegreeOnEquator)
```

```
        val A=Y1-Y0
        val B=-(X1-X0)
        val C=-(A*X0+B*Y0)
        val sqrtAB= sqrt(A*A+B*B)
        assert(sqrtAB!=0.0)
        val Mx=ArrayList<Double>()
        val My=ArrayList<Double>()
        for (i in 0 until N){
            Mx.add((points[i].latitude()-Ox)*mPerLongitudeDegree)
            My.add((points[i].longitude()-
Oy)*(cos(degreesToRad(points[i].longitude()))*mPerLatitudeDegreeOnEquator))
            if((((X1-X0)*(Mx.last()-X0)+(Y1-Y0)*(My.last()-Y0))>0)&&(((X0-
X1)*(Mx.last()-X1)+(Y0-Y1)*(My.last()-Y1))>0)){
                distancesToLine.add(abs(A*Mx.last()+B*My.last()+C)/sqrtAB)
            }
            else{
                distancesToLine.add(INF)
            }
        }
        val distancesToStart=ArrayList<Double>()
        for(i in 0 until N){
            distancesToStart.add(sqrt((X0-Mx[i])*(X0-Mx[i])+(Y0-My[i])*(Y0-
My[i])))
        }
        resultListener.onDistanceResult(Pair(distancesToLine,distancesToStart))
    }

    private fun degreesToRad(degree: Double): Double {
        return degree* PI/180
    }

    fun interface DistancesResultListener{
        fun onDistanceResult(results: Pair<List<Double>, List<Double>>)
    }

//    fun interface OptimizeListener{
//        fun onOptimizeResult(results: MutableList<DirectionsRoute>?)
//    }
}
```