Universitatea ”Politehnica” din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Aplicație mobilă de tip e-evironment pentru determinarea traseului optim

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de
Inginer în domeniul *Calculatoare și Tehnologia Informației*
programul de studii de licență *Ingineria Informației*

Conducător științific
Prof. Dr. Ing. Radu RĂDESCU

Absolvent
Mihai-Florin NEACȘU

Anul 2020

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul **EAII**

Anexa 1

TEMA PROIECTULUI DE DIPLOMĂ

a studentului **NEACŞU A. Mihai-Florin , 443A**

1. Titlul temei: Aplicație mobilă de tip e-evironment pentru determinarea traseului optim

2. Descrierea contribuţiei originale a studentului (în afara părţii de documentare) și specificații deproiectare:

Pornind de la API-ul OpenStreetMap (care conține informațiile necesare creării unei hărți) se va genera o hartă a unui oraș (București) sau a unei țări (în cazul rutelor mari), interfața grafică/harta propriu-zisa fiind realizată cu ajutorul librăriilor Osmdroid si Mapsforge.
După crearea hărții, vor rezulta mai multe liste de vectori (câte una pentru fiecare metodă de transport prin care se poate circula), iar pentru fiecare intersecție a străzilor se va construi un nod (acesta va reține latitudinea și longitudinea în care se realizeaza intersecția). Fiecare nod va fi pus intr-una din listele mai sus menționate. Construirea drumului/drumurilor se va face prin algoritmul Dijkstra (algoritm de găsire a drumului optim). După ce drumurile optime au fost găsite, pentru fiecare dintre acestea se va obține nivelul de poluare din aer și condițiile meteo cu ajutorul unor API-uri, dar și aglomerația în trafic din fiecare zona (în funcție de numărul de utilizatori ai aplicației).
Cu ajutorul algoritmului Minimax, aplicația va recomanda drumurile în funcție de preferințele utilizatorului (dacă utilizatorul dorește un drum cu aerul cat mai curat, acest drum va avea prioritate, deși nu este cel mai scurt).

3. Resurse folosite la dezvoltarea proiectului:
Android Studio, Genymotion, Java, osmdroid, mapsforge, OpenStreetMap - API, Air Quality Programmatic - API, OpenWeather - API

4. Proiectul se bazează pe cunoştinţe dobîndite în principal la următoarele 3-4 discipline:
PC, SDA, PPA, POO

5. Proprietatea intelectuală asupra proiectului aparţine: U.P.B.

6. Data înregistrării temei: 2019-11-29 14:15:50

Conducător(i) lucrare,
Prof. dr. ing. Radu RĂDESCU

Student,

Director departament,
Prof. dr. ing Sever PAȘCA

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **ebc51d525a**

Anexa 5

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "*Aplicație mobilă de tip e-evironment pentru determinarea traseului optim*", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Calculatoare și Tehnologia Informației*, programul de studii *Ingineria Informației* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, *23.06.20*

Absolvent *Mihai-Florin NEACȘU*

(semnătura în original)

Cuprins

Lista figurilor

Lista tabelelor

Lista acronimelor

AMD – Advanced Micro Devices
ANPM – Agenției Naționale pentru Protecția Mediului
API – Application Programming Interface/ Interfață de Programare a Aplicațiilor
BSD – Berkeley Software Distribution
EE – Enterprise Edition
ETC – Etcetera
GPS – Global Positioning System/ Sistem de Poziționare Globală
HAXM – Hardware Accelerated Execution/ Execuție Accelerată Hardware
HTTPS – Hypertext Transfer Protocol Secure/ Protocol de Transfer de Hipertext Securizat
ID – Identification/ Identificare
IDE – Integrated Development Environment/ Mediu de Dezvoltare Integrat
IMEI – International Mobile Equipment Identity/ Identitatea Internațională a Echipamentelor Mobile
IO – Input-Output/ Intrare-Ieșire
IOT – Internet Of Things
IT – Information Technology/ Tehnologia informației
JDK – Java Developer Kit
JRE – Java Runtime Environment
JSON – JavaScript Object Notation
LBS – Location-Based Service/ Serviciu Bazat pe Locație
ME – Micro Edition
MEID – Mobile Equipment Identifier/ Identificator de Echipamente Mobile
NBA-Star – New Bidirectional A-Star
NDK - Native Development Kit
OS – Operating System/ Sistem de operare
PC – Personal Computer/ Calculator Personal
REST – Representational State Transfer
SE – Standard Edition
SMS – Short Message Service/ Serviciu de Mesaje Scurte
URL – Uniform Resource Locator
VLC – Video Lan Client
WOT – Web Of Things
WWW – World Wide Web
Wi-fi – Wireless Fidelirt
XML – Extensible Makup Language

Introducere

Scopul proiectului

Dezvoltarea pe platforma android este fără îndoială unul dintre sectoarele care se extind cel mai rapid. În jur de 179 miliarde de aplicații sunt descărcate anual. În general, piața de aplicații este cucerită de aplicații Google, jocuri, social media, aplicații cu hărți etc..[1]

În prezent, majoritatea aplicațiilor cu hărți (google maps, waze, etc.) au la bază un algoritm pentru găsirea drumului optim care nu ia în calcul foarte multe date legate de mediul înconjurător, singurul factor care este luat în considerare în alegerea drumul optim este aglomerația din trafic.

Proiectul acesta are rolul de a crea o platformă ce îi va ajuta pe utilizatori să aleagă drumul cel mai optim în funcție de informațiile mediului înconjurător (poluare, aglomerație etc.), dar și în funcție de mijloacele de transport disponibile, de exemplu mașina, bicicleta sau pe jos.

Descrierea proiectului

Principalele API-uri pe care le vom folosi în acest proiect vor fi: *OpenStreetMap, Overpass API, Air Quality Programmatic, TomTom Traffic API* și *OpenWeatherMap*. Acestea oferă informații importante în crearea aplicației. Există mai multe modalități prin care putem folosi datele brute ale acestor API-uri; în cadrul acestei aplicații se vor transmite cereri HTTPS pentru care vom primi un răspuns de tip JSON sau XML. Pe lângă acest mod de a manipula informația dată de API-uri, mai putem folosi și biblioteci ce au fost dezvoltate special pentru manipularea acestor date, o astfel de biblioteca este *osmdroid.*

Ca prim pas, se va folosi biblioteca *osmdroid* pentru a se crea obiectul de tip *MapView*; acesta conține toate informațiile necesare pentru a crea harta, însă pentru a putea vizualiza/randa harta este nevoie și de biblioteca *mapsforge.* Biblioteca *osmdroid* și complementul său *osmBonusPack* oferă și posibilitatea de a adăuga alte funcționalități hărții, cum ar fi posibilitatea de a lucra mai multe straturi. Pe un strat putem afișa anumite obiecte pe hartă, dar fără a o modifica în mod direct, de exemplu, pe un astfel de strat se poate afișa utilizatorul pe hartă, se pot afișa obiectele de tip marcatori, se pot adăuga și alte caracteristici, cum ar fi folosirea atingerii multiple, rotirea hărții etc. În acest stagiu, aplicația va avea toată funcționalitatea de bază pe care o necesită o aplicație cu hărți, urmând să introducem toate elementele grafice ale interfeței.

Pentru găsirea drumului optim se vor implementa următorii algoritmi: *Dijkstra, A-Star, Dijkstra Bidirecțional, A-Star Bidirecțional* și *New Bidirectional A-Star.* Acești algoritmi vor fi aplicați asupra unui graf orientat creat cu ajutorul datelor brute extrase din baza de date de la *OpenStreetMap.* În timpul căutării, grafului i se va calcula și greutatea laturii ținând cont de datele mediului înconjurător, cum ar fi poluarea aerului și viteza traficului.

Datele extrase din API-ul *OpenWeatherMap,* nu vor fi folosite pentru a calcula greutatea laturilor, ci pentru a face o sugestie cu privire la modul de transport pe care utilizatorii ar trebui să îl folosească în ziua respectivă.

Am renunțat la algoritmul Minimax, deoarece acesta ar fi îngreunat timpul de procesare, funcționalitatea acestuia fiind înlocuită de anumite calcule matematice realizate în timpul analizei greutății laturi dintre două noduri succesive.

Capitolul 1 – Tehnologii folosite

În acest capitol se vor prezenta tehnologiile folosite pe parcursul realizării aplicației, cum ar fi: sistemul de operare pentru care a fost realizată aplicația, limbajul de programare folosit, mediul de dezvoltare integrat și emulatorul utilizate. Pentru fiecare dintre acestea vor fi prezentate caracteristici importante ce au dus la alegerea lor în realizarea proiectului.

1.1. Sistemul de operare Android

Sistemul de operare Android este o platformă care oferă dispozitivelor portabile și mobile (chiar și netbook-urilor) puterea și portabilitatea sistemului de operare Linux, dar și fiabilitatea și portabilitatea unui limbaj de programare standard de nivel înalt, dar și API-uri. Aplicațiile pe Android sunt scrise în limbajul de programare Java, folosind unelte precum Eclipse, compilație cu API-ul Android și traduse în byte code pentru *Dalvik VM*. [2]

Android Inc. a fost fondat in *Palo Alto, California*, in Octombrie 2003 de către Andy Rubin, *Rich Miner, Nick Sears*, și *Chris White*. [3]

Intențiile de început ale companiei au fost de a crea un sistem de operare pentru camerele digitale, acesta fiind poziția sa către investitori în Aprilie 2004. [4]

In Iulie 2005, Google a achiziționat Android Inc. pentru cel puțin 50 milioane de dolari.[5]

Figura 1.1. Primul telefon cu android. Sursă: [6]

HTC Dream este primul telefon care utilizează platforma Android. Acesta a fost lansat pentru prima data pe data de 23 Septembrie 2008, și a avut ca și scop să provoace *iPhone*-ul, ceea ce a și reușit. Acesta avea suport pentru rețeaua *3G*, avea integrat *Amazon MP3* (*Google Play Music* a apărut ulterior), și putea folosi aplicații precum *Google Talk*, *AOL*, *Yahoo Messenger* și *Windows* Live *Messenger* pentru mesaje, dar și un browser asemănător cu *Chrome*, dar denumit *Browser*. [6]

Android a avut niște schimbări majore în primele revizii, iar când acesta a primit primul său nume de cod (android „Cupcake”) se îndrepta spre a deveni un sistem de operare bazat în întregime pe atingerea ecranului.[6]

Google a continuat să folosească acest tipar pentru a-și denumi și următoarele versiuni de android (deserturi, în ordine alfabetică). Câteva exemple de denumiri pe care Google le-a folosit pentru versiunile de android: [7]

- Android 1.5: Android „Cupcake” [7]
- Android 1.6: Android „Donut” [7]
- Android 2.0: Android „Éclair” [7]
- Android 2.2: Android „Froyo” [7]
- Android 2.3: Android „Gingerbread” [7]

În Ianuarie 2010, Google a lansat seria de telefoane numita Nexus. Aceasta serie de telefoane a fost făcută de Google în parteneriat cu mai mulți producători de telefoane cum ar fi HTC, Samsung etc..[8]

După câteva update-uri majore ale sistemului de operare, Google introduce si seria de telefoane Pixel, în Octombrie 2016, primele telefoane produse de Google în totalitate, acestea înlocuind seria Nexus.[9]

Figura 1.2. Telefonul Google Pixel 2. Sursă: [10]

În prezent, dispozitivele cu sistemul de operare Android au ajuns să înlocuiască o mare parte din obiectele obișnuite pe care le vedeam cândva în toate casele, acest lucru se întâmplă datorita hardware-ului avansat (cum ar fi camera dispozitivului, puterea de procesare, senzorii acestuia), dar și datorită software-ului (aplicațiile, care se folosesc de hardware, pentru a realiza anumite sarcini). Câteva lucruri ce au fost **înlocuite** de telefoanele inteligente sunt: dispozitivele GPS, ziarele, telefoanele publice, camerele de filmat/fotografiat, scannerele, albumul de fotografii, aparatele de înregistrat vocea etc.. [11]

1.2. Java

Java este un limbaj de programare orientat pe obiect, puternic tipizat, conceput de către *James Gosling* la *Sun Microsystems* (acum filială *Oracle*) la începutul anilor ‘90, fiind lansat în 1995.[12]

Caracteristicile limbajului Java sunt:

- Limbaj interpretat, dar și compilat, acest lucru face ca limbajul de programare Java să fie independent de platformă, folosind o mașina virtuală Java care traduce instrucțiunile din „byte code” în niște instrucțiuni ce pot fi citite de către platforma curentă; [12]
- Limbaj orientat pe obiect, acesta pune în evidență următoarele aspecte: obiecte, trimitere de parametrii, încapsulare, clase, biblioteci, moștenire și modificatori de acces; [12]
- Limbaj concurent, deoarece oferă posibilitatea de a rula mai multe secvențe de cod în același timp (engleza: multithreading). [12]
- Limbaj distribuit, deoarece permite utilizarea obiectelor locale și de la distanță. [12]

Limbajul de programare Java este unul de nivel înalt folosit în mai multe domenii (jocuri video, aplicații de mesagerie etc.). Printre aplicațiile ce au fost dezvoltate în limbajul de programare Java se numără:

- *K-9 Mail* – o soluție avansată pentru persoanele ce folosesc e-mailul 24/7; [13]
- *Pixel-Dungeon* – acesta este o dovadă a naturii aproape universale ale tehnologiei. Deși Java este poziționat ca fiind o unealtă pentru aplicațiile de afaceri, acest joc arată un mod interesant de a utiliza limbajul de programare; [13]
- *VLC media player* – aplicație ce îi permite utilizatorului să asculte muzică sau să se uite la videoclipuri la viteze mari; [13]
- *Telegram* – aplicație realizată pentru platforma Android folosită pentru transmiterea instantanee a mesajelor; [13]
- *Bitcoin Wallet* – aplicație realizată pentru platforma Android, aceasta este o soluție inovativă ce îi permite utilizatorului să gestioneze cripto-monedele de pe orice dispozitiv cu sistemul de operare Android. [13]

Astfel, se poate observa ca cele mai multe aplicații distribuite sunt scrise în Java, iar noile evoluții tehnologice permit utilizarea sa și pe dispozitive mobile gen telefon, agenda electronică, palmtop etc. În felul acesta se creează o platformă unică, la nivelul programatorului, deasupra unui mediu eterogen extrem de diversificat. Acesta este utilizat în prezent cu succes și pentru programarea aplicațiilor destinate intranet-urilor.[2]

Există 4 platforme Java furnizate de Oracle:

- *Java Platform, Micro Edition (**Java ME**)* — pentru hardware cu resurse limitate (telefoanele mobile); [14]
- *Java Platform, Standard Edition (**Java SE**)* — pentru sisteme PC; [14]
- *Java Platform, Enterprise Edition (**Java EE**)* — pentru sisteme de calcul mari (de exemplu, pentru servere), eventual distribuite, este folosit in aplicații de nivel înalt. [14]

Pentru a putea scrie și rula programe Java în mod eficient, programatorul are nevoie de următoarele resurse:

- *Java Runtime Environment (**JRE**)* - acesta reprezintă totalitatea resurselor necesare pentru a putea rula programe Java. Printre ele se numără mașina virtuala și un set minimal de clase predefinite Java, corespunzătoare operațiilor des întâlnite in programare; [15]
- *Java Development Kit (**JDK**)* - acest pachet conține resursele necesare creării de programe Java: compilator, generator de documentație, arhivator și o întreaga serie de alte utilitare implicate direct sau indirect in procesul de dezvoltare. În mod normal acesta conține și JRE; [15]
- *Un mediu de dezvoltare Java (**IDE**)* - acesta este o interfață cu unelte puse la dispoziție de JDK; IDE-ul se folosește de ele și nu poate funcționa în lipsa lor. În mod normal codul Java

poate fi scris în orice fișier text, dar se prefera folosirea unui mediu de dezvoltare datorită facilităților pe care le oferă.

Un IDE se alege de programator in funcție de caracteristici și nevoi. Cele mai bune IDE-uri pentru Java în acest moment sunt: *Eclipse, BlueJ, IntelliJ IDEA, jGRASP, NetBeans* și *Android Studio*. [16]

În cazul proiectului, deoarece aplicația este dezvoltata pentru sistemul de operare Android (specific dispozitivelor mobile), pentru realizarea aplicației se va folosi platforma *Java ME*, iar ca mediu de dezvoltare integrat se va folosi *Android Studio*. Acest IDE va fi prezentat in detaliu in subcapitolul următor.

1.3. Android Studio

Deoarece limbajul de programare folosit pe parcursul proiectului este Java, am putea folosi orice mediu de dezvoltare ce îl suportă, dar sistemul de operare pentru care dezvoltăm aplicația este Android, deci vom folosi Android Studio.

Android Studio este mediul de dezvoltare integrat (*IDE*) oficial pentru sistemul de operare Android. Acesta a fost creat pe baza programului *JetBrains' IntelliJ IDEA* și proiectat special pentru dezvoltarea aplicațiilor Android. Acesta a fost dezvoltat de către Google în parteneriat cu JetBrains și a fost scris în *Java, Kotlin* și *C++*. [17]

Din 7 Mai 2019, limbajul de programare preferat de Google pentru dezvoltarea aplicațiilor android este *Kotlin*, dar rămân și alte limbaje de programare valabile, cum ar fi *Java* sau *C++*. [17]

Kotlin este proiectat pentru a opera împreuna cu Java și versiunea Java Virtual Machine (JVM) a bibliotecii sale standard, care depinde de *Java Class Library* (un set de biblioteci încărcate dinamic pe care aplicațiile Java le pot apela la rulare). [18]

Deși *Kotlin* este, în acest moment, superior limbajului Java când vine vorba de dezvoltarea aplicațiilor pe platforma Android, nu îl vom folosi, deoarece Java este un limbaj mult mai cunoscut și mai ușor de înțeles.

Android Studio are următoarele caracteristici, care îl favorizează pentru a fi folosit drept IDE in dezvoltarea aplicațiilor pe platforma Android:

- Suport de construire bazat pe *Gradle*; [17]
- Refactorizare specifică Android și remedieri rapide; [17]
- Instrumente *Lint* pentru a capta performanță, capacitatea de utilizare, compatibilitatea versiunii și alte probleme; [17]
- Unealtă pentru a crea componente și design-uri Android comune; [17]
- Un editor al layout-ului bogat, care permite utilizatorilor să tragă și să arunce componente UI (butoane, casete text, etc.), opțiunea de a pre-vizualiza layoutul pe mai multe configurații de ecran; [17]
- Asistență pentru crearea aplicațiilor *Android Wear*; [17]
- Suport integrat pentru *Google Cloud Platform*, care permite integrarea cu *Firebase Cloud Messaging* (anterior „*Google Cloud Messaging*") și *Google App Engine*; [17]
- Depozitul *Maven*; [17]
- Dispozitiv virtual Android (emulator) pentru a rula și depana aplicații în Android Studio; [17]
- *NDK* (*Native Development Kit*), ce reprezintă un set de unelte ce îi permite programatorului să folosească C și C++ în dezvoltarea aplicațiilor. [17]

Crearea unui proiect în Android Studio este foarte simpla și se poate realiza în doar doi pași:

Pasul 1. Se alege tipul de proiect ce urmează a fi creat, in Android Studio putem realiza aplicații pentru mai multe tipuri de dispozitive: *Telefoane* și *tablete*, *'Wear OS'*, *televizoare*, și așa mai departe; totodată punând la dispoziție anumite șabloane foarte utile pentru programatorii începători.

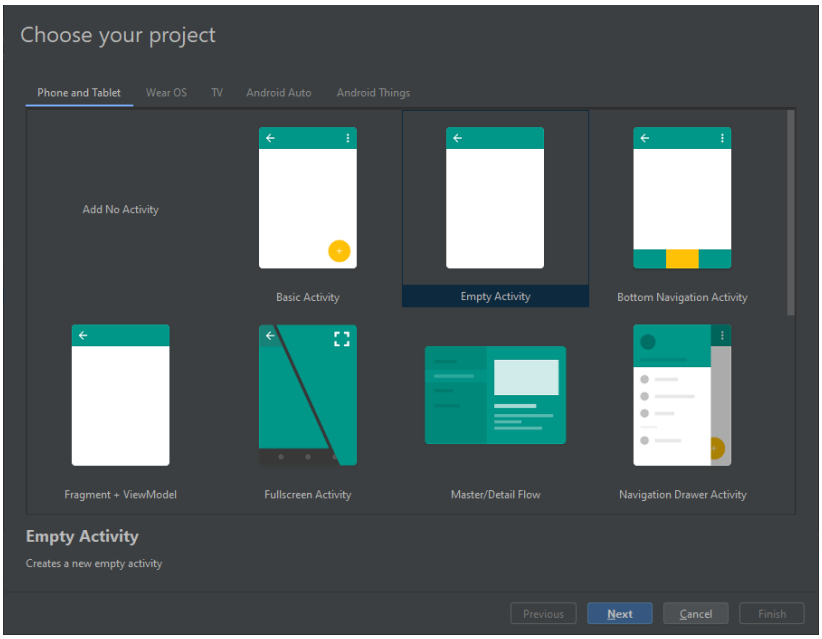


Figura 1.3. Fereastra ce apare la crearea unui proiect nou. Alegerea tipului de proiect.

Pasul 2. Denumirea proiectului, a limbajului de programare ce urmează a fi folosit (acest IDE suporta *Java* și *Kotlin*), dar și nivelul minim pentru API al aplicației (acesta corespunde cu versiunea de android, de exemplu: *API 28* corespunde versiunii *Android 9.0 – Pie*).

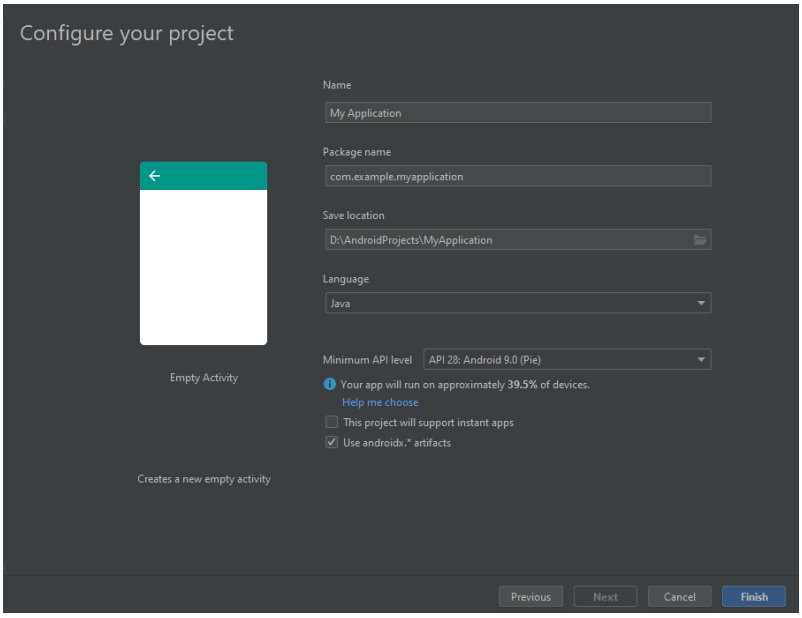


Figura 1.4. Fereastra in care se alege denumirea proiectului, limbajul de programare folosit si nivelul de API.

Modul în care arată activitatea principala (selecția făcută la primul pas) se poate modifica ulterior de către programator prin editorul de layout al IDE-ului, fie prin a trage componentele grafice,

fie prin cod. Pentru construirea proiectului am ales o activitate goala, adăugând alte componente grafice pe parcurs, în funcție de nevoile proiectului. Limbajul de programare ales este Java, iar nivelul pentru API este 29, asociat unei versiuni de android 10.0 (Q).

In continuare se va prezenta interfața grafică a mediului de dezvoltare ales.

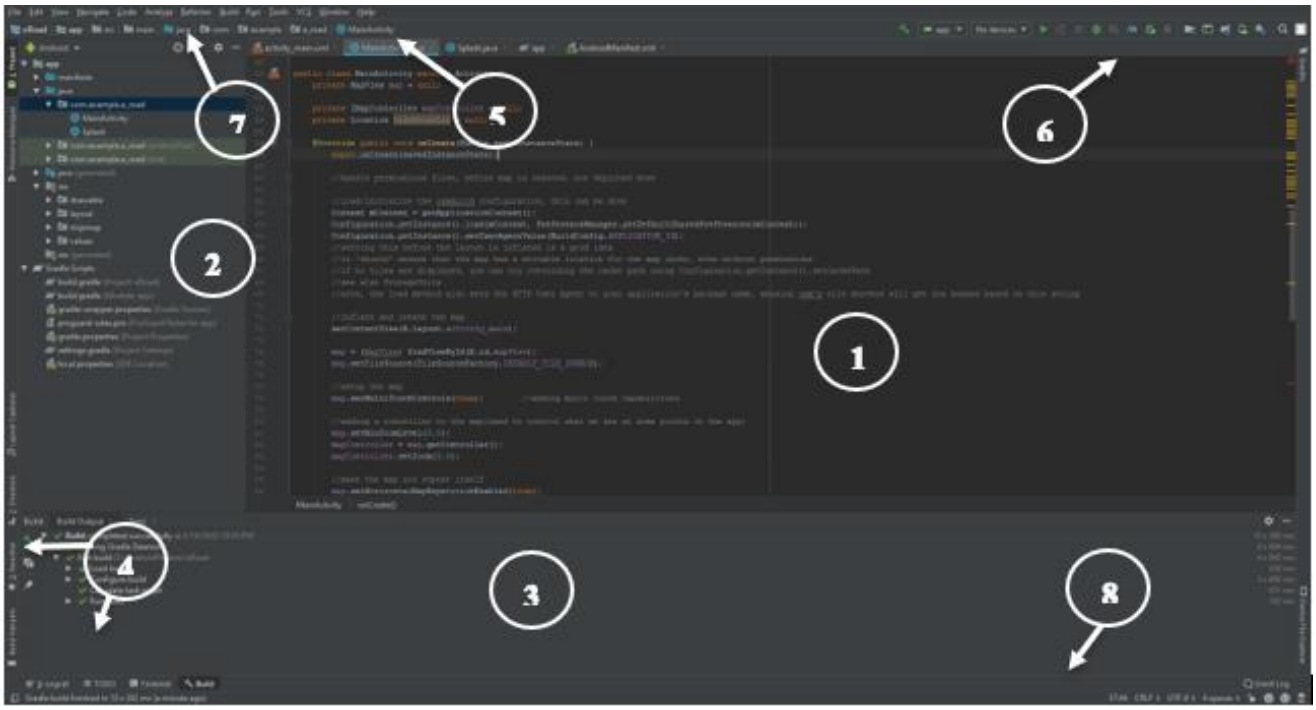


Figura 1.5. Interfața grafică a programului Android Studio.

- 1 – **Fereastra editorului**, în acest loc se creează si modifica codul. În funcție de tipul fișierului deschis, acesta își poate schimba înfățișarea. [17]
- 2,3 – **Fereastra cu unelte**, permite accesul la anumite sarcini specifice cum ar fi managementul proiectului, căutare, controlul versiunii, și multe altele. [17]
- 4 – **Bara pentru fereastra de unelte**, este așezată pe marginea ferestrei IDE-ului și conține butoane care permit extinderea sau strângerea individuală a ferestrelor pentru diferite unelte. [17]
- 5 – **Meniul principal**, din acest meniu se pot redeschide diverse unelte ce au fost închise la un anumit moment de timp, se poate crea un nou proiect etc.. [17]
- 6 – **Toolbar/bara de instrumente,** are o varietate mare de acțiuni, incluzând rularea aplicației. [17]
- 7 – **Bara de navigatie**, aceasta te ajuta sa navighezi prin proiect și să deschizi fișiere pentru editare. Oferă o vizualizare mai compactă a structurilor vizibile in fereastra 2. [17]
- 8 – **Bara de status**, afișează statusul proiectului și al IDE-ului, dar și orice mesaje sau avertizări. [17]

Fereastra editorului o sa difere in funcție de fișierul pe care programatorul îl editează la un anumit moment de timp; în **Figura 1.5.** se poate observa editorul standard, ce apare atunci când deschidem fișierele de tip .java. Pe lângă acest tip de editor, mai avem și editorul de layout, care apare în timpul editării fișierelor de tip .xml. Acesta poate avea doua forme, o formă de text (programatorul scrie cod xml pentru a crea obiectele din activitate) și o formă de design (în care programatorul poate adăuga butoane direct într-o interfața grafica).

În continuare vor fi prezentate cele doua tipuri de editor de layout si caracteristicile lor.

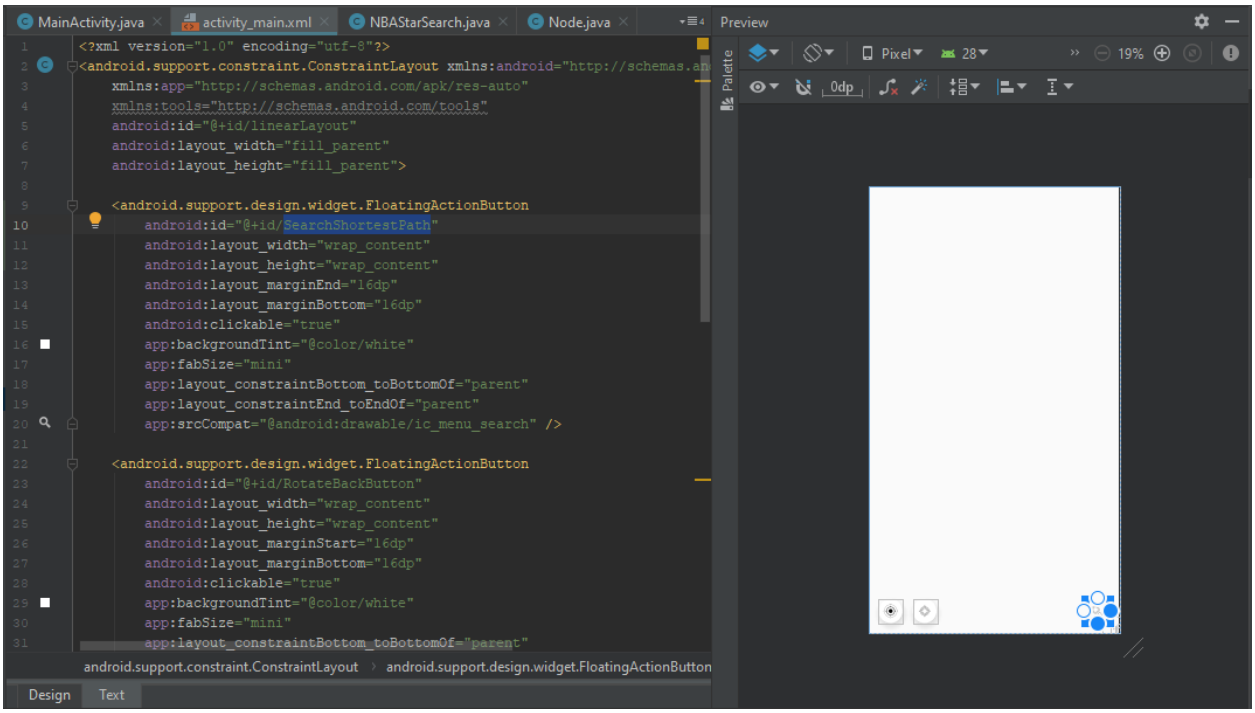


Figura 1.6. Editor de layout de tip text.

Putem observa cum, acest editor oferă posibilitatea de a scrie cod pentru a implementa interfața grafica, dar oferă și posibilitatea de a avea o reprezentare în timp real al codului ce este scris. Acest tip de editor este preferat de către programatorii profesioniști.

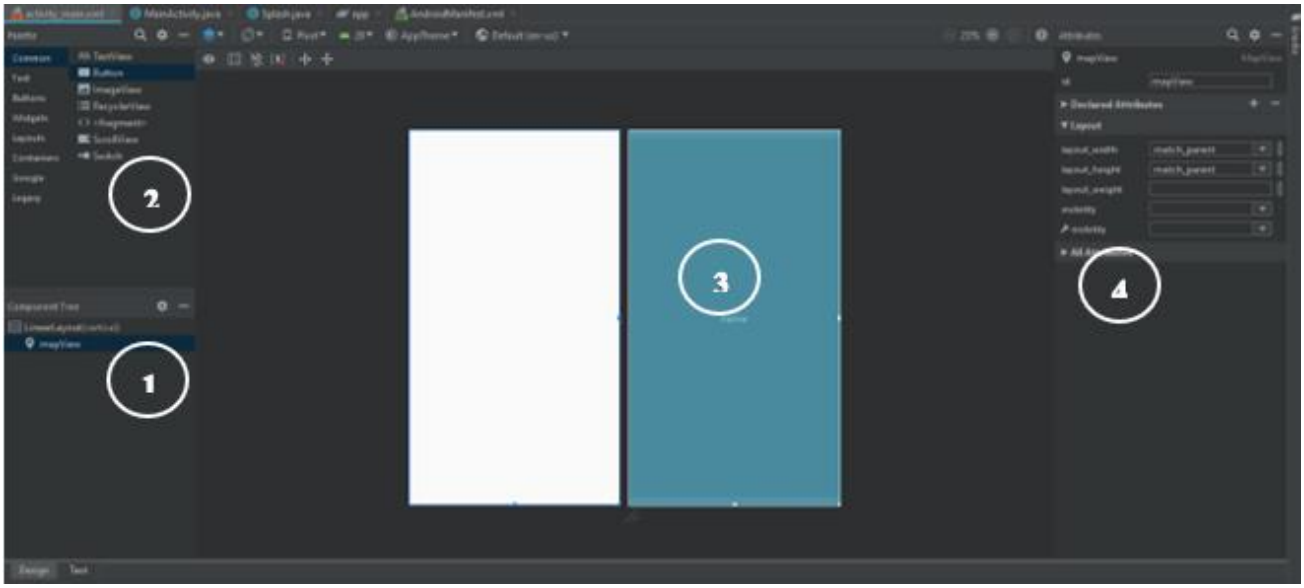


Figura 1.7. Editorul de layout de tip design.

- 1 – **View Tree**, în aceasta fereastra putem vedea ierarhia View-urilor din layout. Un View este o clasă ce reprezintă un bloc de construire de baza pentru componentele interfeței cu utilizatorul. [19]
- 2 – **Palette**, fereastra din care putem selecta diferite obiecte pe care le putem plasa în interiorul layout-ului. [19]

- 3 – **Screenshot**, reprezintă fereastra prin care putem vizualiza layoutul așa cum acesta o sa apară pe dispozitiv. [19]
- 4 – **Properties Table**, Proprietățile layout-ului pentru obiectul selectat. [19]

În **Figura 1.6.** (unde este selectat pentru editare un fișier *.xml*) se poate observa cum editorul de text se modifică dintr-un editor special pentru Java (vezi **Figura 1.5.**), într-un editor special pentru layout, în care putem adăuga butoane, ferestre speciale pentru text, și multe altele.

1.4. Emulator

În IT, un emulator este o componentă hardware sau software care permite unui sistem informatic (numit gazdă) să se comporte ca un alt sistem informatic (numit invitat). De obicei, un emulator permite sistemului gazdă să ruleze software sau să utilizeze dispozitivele periferice proiectate pentru sistemul oaspete. [20]

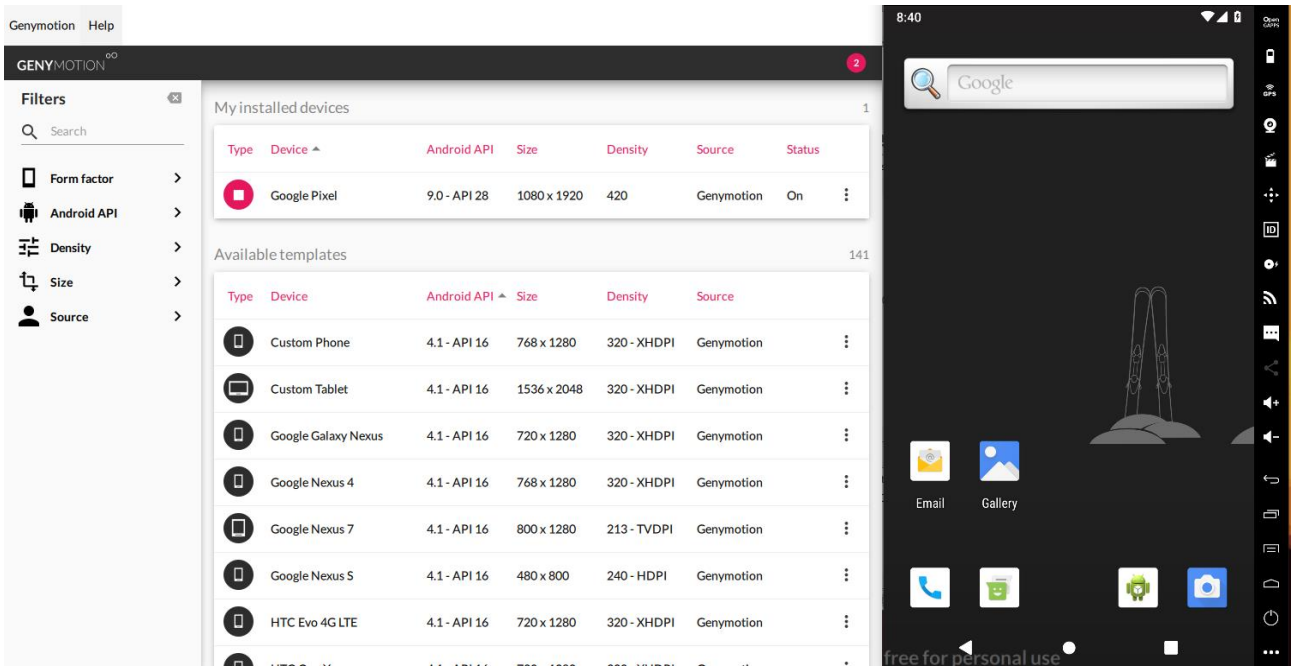


Figura 1.8. Interfața grafică a emulatorului Genymotion, și emularea unui dispozitiv Google Pixel cu versiunea de android 9.0.

Deoarece aplicația dezvoltată este specifica unui sistem mobil, pe platforma Android, nu vom putea folosi un calculator pentru a testa codul scris. Pentru a face acest lucru exista doua metode posibile: folosirea telefonului personal sau folosirea unui emulator ce va crea un dispozitiv cu sistemul de operare Android virtual.

Emulatorul standard, oferit de Android Studio, la început era foarte încet, chiar și cu un hardware capabil. Google a îmbunătățit drastic performanța în ultimii ani, dar câteva probleme încă rămân. Versiunea de Windows a emulatorului folosește HAXM (Intel Hardware Accelerated Execution), ceea ce făcea ca emulatorul sa fie compatibil doar pentru procesoarele Intel, într-un update introdus in Iulie 2018, acesta a devenit compatibil si cu procesoarele AMD. [21]

Astfel, în cazul de fata am ales emulatorul **Genymotion**, acesta este gratuit și ușor de folosit, creat pentru a ajuta dezvoltatorii de aplicații pentru Android să-și testeze produsele într-un mediu sigur și virtual. Acesta poate emula marea majoritate a versiunilor Android pe care le putem găsi pe piața.

În continuare vor fi prezentate caracteristicile cele mai importante ale emulatorului, caracteristici ce îl fac ideal pentru dezvoltarea aplicațiilor pe platforma Android, și un înlocuitor foarte bun al emulatorului standard, oferit de Android Studio.

Caracteristica cea mai importantă pentru aplicația propusă este cea a GPS-ului, aplicația având nevoie de permisiuni pentru locație (pentru a putea afla punctul în care utilizatorul se afla în momentul curent).

În **Figura 1.9.** este prezentat meniul prin care se pot manipula parametrii GPS-ului pentru telefonul emulat. Printre acești parametrii se numără: orientarea, latitudinea, longitudinea, acuratețea, dar și altitudinea, având și un buton special ce-ți afișează locația setată pe hartă (foarte important, deoarece dorim sa testam faptul că aplicația noastră afișează aceeași locație).

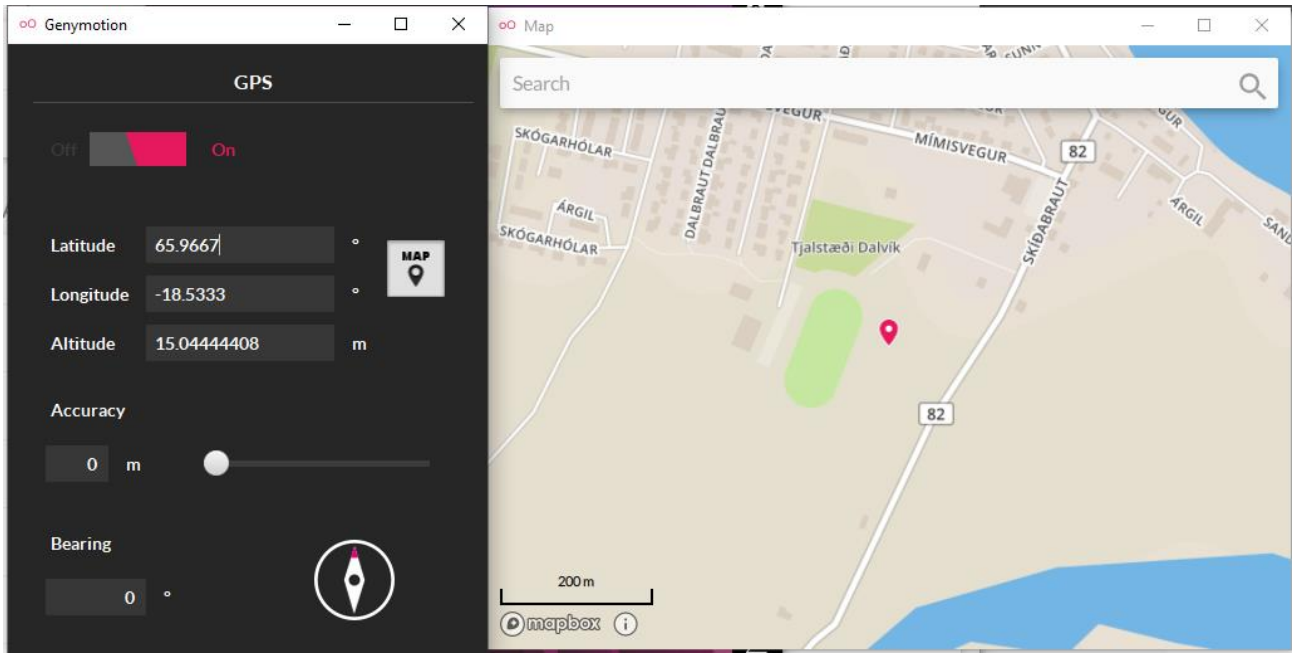


Figura 1.9. Meniul de setare al GPS-ului, pentru emulatorul ales.

O altă caracteristică importantă a acestui emulator este meniul din dreapta dispozitivului emulat, acesta ne permite manipularea anumitor resurse ale telefonului printre care:

- **Bateria**, în cazul în care aplicația trebuie să reacționeze diferit în funcție de nivelul la care se află bateria. [22]
- **Camera și captura**, oferă posibilitatea de a folosi camera web a calculatorului pentru a realiza poze și filme. [22]
- **Internet și Wi-Fi**, foarte important, dacă se dorește testarea conectivității prin deconectări (cum ar fi cele ce pot apărea la metrou). [22]
- **Sufocarea Disk IO**, emulează dispozitive cu stocare internă lentă. [22]
- **SMS si apeluri**, setează întreruperi realizate de sms-uri sau apeluri și se poate vedea dacă aplicația reacționează corect. [22]
- **Accelerometru și multitouch** (atingeri multiple), transmite evenimente ale giroscopului sau de *multi-touch* din orice dispozitiv Android conectat la calculator. [22]
- **Pixel perfect**, afișează aplicația pe ecranul calculatorului la dimensiunile reale. [22]
- **Open Gapps**, reprezintă un widget ce îi permite utilizatorului să instaleze pe emulator *Google Play Services.* [22]

- **Android ID, IMEI**, permite modificarea valorilor Android ID/IMEI/MEID pentru dispozitivul emulat. Indiferent dacă se testează sistemul de urmărire a instalării sau se clasifica dispozitivele, acesta este un instrument extraordinar și practic. [22]

Capitolul 2 – Biblioteci și API-uri

Un API (*Application Programming Interface*) reprezintă un set de unelte pe care programatorii le pot folosi pentru a crea software/aplicații. Un API bun o sa aibă comenzi concise, clare și simple pe care programatorul le poate folosi și refolosi, astfel încât sa nu fie nevoiți sa reconstruiască totul din nou. [23]

O bibliotecă este o colecție de implementări ale comportamentului, scrise în termeni de limbaj de programare, care are o interfață bine definită prin care este invocat comportamentul implementat de un programator. De exemplu, persoanele care doresc sa scrie un program de nivel superior pot utiliza o biblioteca pentru a efectua apeluri de sistem în loc sa implementeze acele apeluri de sistem încă o data. În plus, comportamentul poate fi reutilizat de mai multe programe independente.

Bibliotecile, in general, folosesc datele brute colectate de la un anumit API și îi dau o anumita funcționalitate.

Ca și API-uri folosite, proiectul are la baza: *OpenStreetMap API* și *Overpass API*. Pe lângă acesta, vom adăuga și API-urile: *OpenWeatherMap, TomTom Traffic API* și *Air Quality Programmatic*, pentru a prelua datele din mediul înconjurător.

2.1. OpenStreetMap (OSM)

OSM este un proiect care creează și distribuie date geografice gratuite pentru întreaga lume. Acest proiect a luat naștere, deoarece majoritatea hărților despre care se zice ca sunt gratuite au, de fapt, restricții legale sau tehnice privind utilizarea acestora, împiedicând oamenii sa le folosească în mod creativ, productiv sau neașteptat.[24]

Cu alte cuvinte, *OSM* este o harta a lumii, de tipul *open source*, creata de către o comunitate de cartografi care contribuie și menține datele despre drumuri, cafenele, și multe altele, pentru tot globul. Acest API pune accent pe cunoștințele locale. Cartografii utilizează imagini aeriene, dispozitive GPS și hărți de câmp cu tehnologie scăzută pentru a verifica dacă *OSM* este corect și actualizat. [24]

O varietate de servicii populare încorporează un fel de geo-localizare sau componenta bazată pe harta. Serviciile notabile care utilizează *OSM* sunt următoarele:

- *MAPS.ME, Guru Maps, INRIX Traffic, Navit, Cruiser* – toate aceste aplicații oferă navigare pas cu pas, chiar și ghidare prin voce. [25]
- *Vespucci, StreetComplete, OSM Contributor, OSMBugs* – aceste aplicații oferă anumite funcții ce îi permit utilizatorului să prelucreze/editeze informațiile date de *OSM*. [25]
- *Climb The World, Locus Map, Layar Reality Browser* – aceste aplicații suprapun peste datele *OSM* o vizualizare în timp real al camerei din dispozitiv (*Augumented Reality*). [25]
- *Alminav, AlpineQuest GPS Hiking, Androzic* – aceste aplicații sunt folosite pentru a reține drumul pe care utilizatorul îl urmează. [25]
- *Pokemon Go, BucketMan* – jocuri realizate cu ajutorul datelor OSM. [25]

Putem menționa și firma *Wayfinder System*, o firma deținută în întregime de un subsidiar al firmei *Vodafone*. Pe 11 martie 2010 a fost anunțat că *Vodafone* va închide compania, iar toți angajații vor fi dați afara, urmând ca pe 13 iulie 2010 să se anunțe de către *Wayfinder* că o să-și facă software-ul open source sub licența BSD cu 3 clauze. [26]

Codul sursă pentru server (cu unelte de importat pentru datele de harta din *OSM*), dar și pentru client sunt valabile pe *GitHub*.[27]

Wayfinder a asigurat navigarea pentru telefoanele inteligente pentru milioane de utilizatori cu o soluție end-to-end și o platforma *Location-Based Service* (LBS) extrem de scalabilă care rulează pe grupuri de servere *Linux*, cat și clienții care rulează pe platformele *Android, Windows Mobile* si *iOS*.[28]

Astfel *Wayfinder* poate fi văzută ca piatra de temelie a aplicațiilor cu hărți ce se folosesc de GPS.

2.2. Overpass

Atât *OpenStreetMap* API, cat și *Overpass* API își construiesc răspunsul folosind aceleași date (cele oferite de *OSM*), având aceeași funcționalitate, dar *OpenStreetMap* API are o limitare a numărului de noduri pe care le putem cere (aproximativ 5000 de noduri), acest lucru limitând căutările la o distanță de maxim 2 km; *Overpass* API elimină aceste limitări și ne oferă posibilitatea de a face cereri pentru hărți cât mai mari.

Deși API-ul acesta nu are limitări, trebuie să fim atenți cu dimensiunea hărții cerute, deoarece cu cat aceasta este mai mare, cu atât răspunsul o sa fie recepționat mai greu, având și un timp de prelucrare al hărții mai ridicat.

2.3. OpenWeatherMap (OWM)

OWM este un serviciu online care furnizează date despre vreme. Acesta este deținută de *OpenWeather Ltd*, cu sediul în *Londra, Marea Britanie*. [29]

Au fost dezvoltate peste 6,000 de API-uri pentru vreme, acestea pot fi găsite pe *GitHub*. *OWM* API furnizează date despre vreme, prognoze meteo, dar și date istorice pentru mai mult de 2 milioane de utilizatori. [29]

Acest API suportă mai multe limbi, unități de măsură și formate de date. În plus, serviciul *OWM* îi permite oricărui utilizator sa preia datele pentru vreme de pe website-ul companiei. [30]

Unul dintre proiectele cele mai cunoscute la care au fost folosite datele de vreme curente este realizat de *Mozilla*, și este numit *„The WebThings Gateway”* și are ca scop automatizarea caselor. [31]

Proiectul *The WebThings* are scopul de a conecta obiecte din lumea reală la *World Wide Web* (WWW). Ideea de *Web of Things* (*WoT*) este de a crea un *Internet of Things* (*IoT*) decentralizat, prin oferirea de adrese URL pe web pentru a le face conectabile și descoperibile, și definind un model de date standard și API-uri pentru a le face interoperabile. [31]

2.4. Air Quality Programmatic

Acest API oferă date despre poluarea aerului de pe întreg globul, are opțiunea de prognoza, dar și date istorice asupra locațiilor (din punct de vedere al poluării). [32]

Acesta are pana la 6 nivele de poluare:

- 0 – 50 : *Bun* – calitatea aerului este considerată satisfăcătoare, iar poluarea aerului este mica sau fără risc. [32]
- 51 – 100 : *Moderat* – Calitatea aerului este acceptabilă; în orice caz, din cauza anumitor poluanți s-ar putea să fie o anumită îngrijorare pentru sănătatea anumitor persoane sensibile la poluare. [32]
- 101 – 150 : *Nesănătos pentru grupuri sensibile* – Membrii grupurilor sensibile prezinta efecte asupra sănătății. Publicul general este puțin probabil sa fie afectat. [32]

- 151 – 200 : *Nesănătos* – Marea majoritate a lumii poate începe să vadă efecte asupra sănătății. [32]
- 201 – 300 : *Foarte nesănătos* – Toata populația este afectată. [32]
- 300+ : *Riscant.*

Cele mai poluate țări, din întreaga lume, sunt: India (poluarea aerului este 701), Africa de Sud (poluarea aerului este 603) și China (poluarea aerului este 590). Printre cele mai puțin poluate tari se numără: Myanmar, Moldova, Monaco, Costa Rica, toate acestea având nivelul de poluare sub 10. [33]

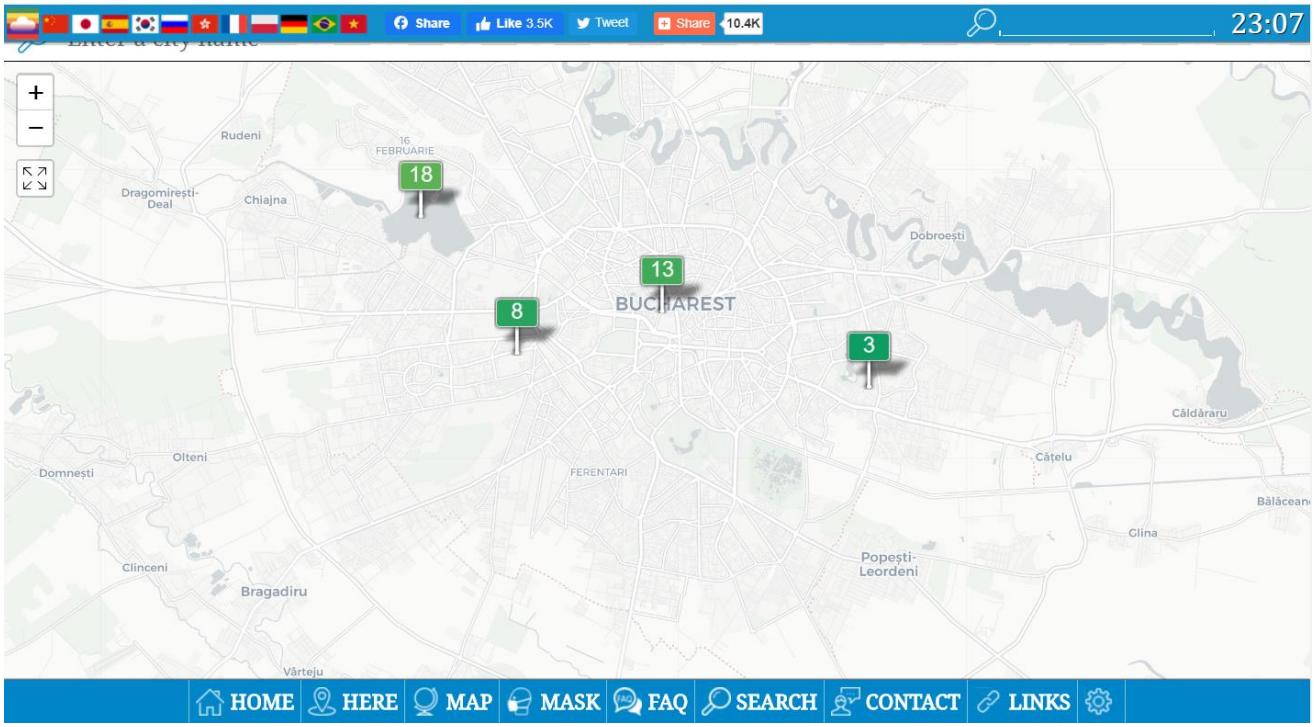


Figura 2.1. Nivelul de poluare din București. Sursă: [33]

Din câte se poate observa, deși acest API are o baza de date destul de mare, aceasta nu o sa influențeze drumul optim pe care noi îl vom crea, deoarece acesta măsoară nivelul de poluare doar în puncte cheie ale orașelor mari, deci vom avea un nivel de poluare asemănător pentru a calcula drumul optim. Acest API are mai mult sens pentru rutele mari, în care algoritmul o sa sugereze ocolirea orașelor (ceea ce ar presupune un drum mai lung, dar mai puțin poluat).

2.5. TomTom Traffic API

Acest API este realizat de cei de la *TomTom* și ne oferă date legate de drumurile ce au trafic [34], algoritmul de găsire al drumului optim trebuie sa detecteze dacă este trafic pe un anumit drum și să sa își dea seama dacă exista și alta cale mai rapidă.

Acest API reprezintă o suită de servicii web, concepute pentru dezvoltatori, pentru a crea aplicații web și mobile în jurul traficului în timp real. Aceste servicii web pot fi utilizate prin intermediul API-urilor *RESTful*.[34]

TomTom Traffic API poate oferii următoarele caracteristici pentru o anumită locație:

- *Incidente in trafic* – acesta oferă o detalii exacte despre blocaje și incidente în jurul unei rețele rutiere. [34]

- *Fluxul de trafic* – acesta oferă vitezele observate in timp real și timpul de deplasare pentru toate drumurile cheie dintr-o rețea. [34]

2.6. Bibliotecile folosite

Principalele bibliotecili folosite în acest proiect sunt:

- *OSMdroid* – Este un vizualizator de hartă (*map viewer*) și extrage datele din *OSM*. [35]
- *OsmBonusPack* – Oferă anumite funcționalități precum marcatori, suport pentru rute și direcții [36] (vor fi folosite pentru afișarea drumului optim găsit).
- *MapsForge* – Este folosit pentru a randa ‚plăcile' (*tile renderer*), mai exact, acesta folosește datele extrase de *OSMdroid* și le folosește pentru a afișa harta în aplicație. [37]
- *JSONObject* – Bibliotecă folosită, dacă dorești sa lucrezi cu JSON în aplicație. [38] Folosim aceasta bibliotecă pentru a analiza informațiile brute extrase din API-uri.
- *Xml* și *XmlPullParser* – Biblioteci folosite atunci când se dorește lucrul cu fișiere de tip XML (*Extensible Markup Language*).

Capitolul 3 – Algoritmi pentru găsirea drumului optim

În general algoritmii au pași care se repetă (iterează) sau au nevoie de decizii, cum ar fi logică sau comparații. [39]

Algoritmii sunt esențiali pentru modul în care calculatoarele procesează informația, deoarece programele de pe calculatoare, în esență, sunt algoritmi care îi spun calculatorului ce pași specifici să realizeze, astfel încât să se întâmple o anumită sarcină. [39]

Algoritmii sunt întotdeauna lipsiți de ambiguitate și sunt folosiți ca specificații pentru efectuarea calculelor, procesării datelor, raționamentului automat și altor sarcini. Algoritmii aleși pentru realizarea aplicației propuse sunt folosiți pentru găsirea drumului cel mai optim între doua noduri dintr-un graf orientat.

3.1. Dijkstra

Acest algoritm găsește drumul cel mai scurt între doua noduri dintr-un graf, care în cazul nostru, reprezintă o rețea de drumuri.

Algoritmul *Dijkstra* a fost conceput de către *Edsger W. Dijkstra* și a fost publicat în 1959. Algoritmul este valid și pentru grafuri nenegative având o eficiență $O(N^2)$, unde O reprezintă complexitatea algoritmului. [40]

În acest moment exista mai multe variații ale algoritmului; în varianta originală acesta găsea drumul cel mai scurt dintre doua noduri date [41], dar cea mai comuna varianta fixează un singur nod ca și nod "sursa" și găsește calea cea mai scurta de la sursa la toate celelalte noduri din graf, creându-se astfel un copac cu drumurile cele mai scurte. [42]

Fiecare nod va avea o variabilă G în care se reține distanța totală parcursă până la nodul curent.

Pentru a găsi drumul optim, prin algoritmul *Dijkstra*, trebuie sa urmam pașii:

- Se vor marca toate distanțele nodurilor ca fiind temporare, cu excepția nodului de start, care este permanentă. Un nod devine permanent atunci când variabila G nu are valoare infinită;
- Inițializăm variabila G pentru toate nodurile existente în graf cu "infinit";
- Inițializăm variabila G a nodului de start cu 0;
- Se va seta nodul de start ca fiind activ;
- Calculăm distanțele permanente ale tuturor vecinilor nodului activ, facem acest lucru prin adunarea tuturor distanțelor (luate ca valori ale laturilor din graf) de la nodul de start până ajungem la nodul activ. Dacă aceasta distanța calculata este mai mica decât cea curentă, distanța va fi actualizată și se va seta nodul curent ca fiind noul părinte al nodului vecin. Acest pas se mai numește și actualizare și este ideea centrala din spatele algoritmului *Dijkstra*;
- Se va seta nodul cu cea mai mica distanță permanentă ca fiind nodul activ. Se va nota distanța lui ca parametru;
- Se vor repeta ultimii 2 pași până când nu mai există noduri rămase cu distanța temporară în graf, sau daca am ajuns la destinație;
- Daca am ajuns la destinație se construiește drumul optim, altfel vom afișa un mesaj de eroare.

Pentru a exemplifica acest algoritm, vom considera un graf orientat cu 19 noduri, care are următoarele relații între noduri:

- Nodul 1 are ca și vecini nodurile 2, 5 si 9, având valorile pe laturi 4, 2 respectiv 7.
- Nodul 2 are ca și vecini nodurile 1 si 10, având valorile pe fiecare latură 4.

- Nodul 3 are ca și vecini nodurile 6 si 7, având valorile pe laturi 3 respectiv 4.
- Nodul 4 are ca și vecin nodul 7, având valoarea latura 6.
- Nodul 5 are ca și vecini nodurile 1 si 10, având valorile pe laturi 2 respectiv 3.
- Nodul 6 are ca și vecini nodurile 3, 10 si 11, având valorile pe laturi 3, 6 si respectiv 6.
- Nodul 7 are ca și vecini nodurile 3 si 4, având valorile pe laturi 4 respectiv 6.
- Nodul 8 are ca și vecin nodul 14, având valoarea pe latură 7.
- Nodul 9 are ca și vecini nodurile 1, 10 si 12, având valorile pe laturi 7, 6 respectiv 4.
- Nodul 10 are ca și vecini nodurile 2, 5, 6, 9, 12 si 13, având valorile pe laturi 4, 3, 6, 6, 5 respectiv 6.
- Nodul 11 are ca și vecini nodurile 6 si 14, având valorile pe laturi 3 respectiv 7.
- Nodul 12 are ca și vecini nodurile 9, 10 si 18, având valorile pe laturi 4, 5 respectiv 4.
- Nodul 13 are ca și vecini nodurile 10, 15 si 17, având valorile pe laturi 6, 3 respectiv 8.
- Nodul 14 are ca și vecini nodurile 8, 11, 16 si 17, având valorile pe laturi 8, 7, 4 respectiv 5.
- Nodul 15 are ca și vecini nodurile 13 si 19, având valorile pe laturi 3 respectiv 4.
- Nodul 16 are ca și vecini nodurile 13, 14 si 17, având valorile pe laturi 8, 4 respectiv 6.
- Nodul 17 are ca și vecini nodurile 14 si 16, având valorile pe laturi 5 respectiv 6.
- Nodul 18 are ca și vecin nodul 12, având valoarea pe latură 4.
- Nodul 19 are ca și vecin nodul 15, având valoarea pe latură 4.
- Vom considera ca și nod de start nodul 10, iar ca și nod destinație/țintă, nodul 14.

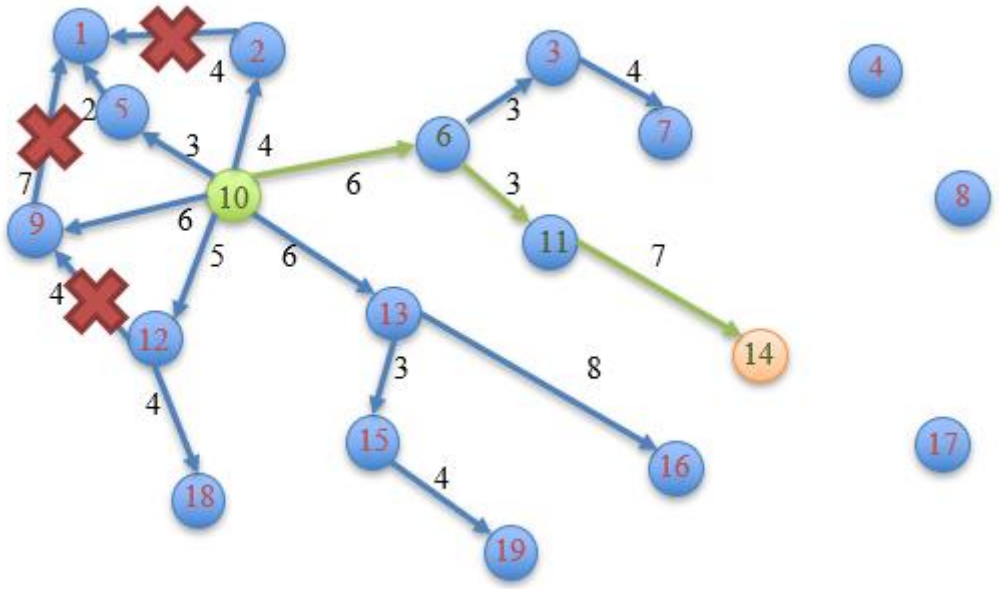


Figura 3.1. Exemplu de drum optim într-un graf, găsit prin algoritmul *Dijkstra*.

În **Figura 3.1.** se poate observa cum funcționează algoritmul *Dijkstra*, pornind de la un nod de start (10) vom vedea ce vecini are, dacă niciunul dintre acești vecini nu este destinația, algoritmul continua, schimbându-se punctul pentru care facem verificarea vecinilor, selectând nodul cu cea mai mică distanță permanentă. Se vor elimina vecinii care au un drum mai lung decât ceea ce a fost verificat până într-un anumit moment; Se poate observa cum laturile dintre nodurile 12 – 9, 9 – 1, 2 – 4 sunt eliminate, deoarece acestea constituie un drum mai lung pentru a ajunge la nodul 1, cel mai scurt drum fiind de la nodul 10 prin modul 5, având un cost total al drumului de 3+2=5.

Când se găsește drumul cel mai optim algoritmul se oprește (nodurile 4, 8, 17 nu au mai fost verificate) și se creează drumul optim.

În cazul nostru, graful este mult mai complex, și o sa fie creat folosind hartă dată de API-ul *OpenStreetMap*, astfel încât fiecare intersecție de străzi o sa fie considerată ca fiind un nod, iar doua noduri sunt legate între ele printr-o latura. ‚Greutatea'/ valoarea de pe o latura va consta în lungimea drumului, dar și de nivelul de poluare, trafic etc.. De exemplu, pentru doua drumuri, cu distanțe aproximativ egale (sa zicem 10 km fiecare), dacă am avea pentru unul un nivel de poluare 60, iar pentru celălalt nivelul de poluare este 30, prin adunarea acestor valori la greutatea laturii, algoritmul nostru o sa aleagă drumul cu mai puțină poluare, astfel dacă diferența de distanța este mult mai semnificativă algoritmul o sa facă un ‚compromis'.

Acest algoritm nu este foarte eficient pentru hărțile de dimensiuni mari (cum ar fi în cazul nostru), deoarece acesta verifica fiecare nod, pana când va ajunge să verifice și nodul țintă.

Variații ale acestui algoritm sunt: *Modified Dijkstra, A-Star*.

Datorită ineficienței acestui algoritm, se va realiza și implementarea algoritmilor: *A-Star* și *NBA-Star*, acești algoritmi fiind cel puțin la fel de eficienți ca și algoritmul *Dijkstra*.

3.2. A-Star

Acest algoritm a fost introdus în 1964 de către *Nils Nilsson* și reprezintă o abordare bazată pe euristică, ce a avut ca scop îmbunătățirea vitezei algoritmului *Dijkstra*, inițial numele acestui algoritm a fost A1, în 1967 *Bertram Raphael* a îmbunătățit algoritmul, dar nu a reușit să îl facă sa găsească optimul, acest algoritm a fost denumit *A2*. În 1968 *Peter Hart* a introdus argumentul că algoritmul *A2* poate găsi optimul daca se folosește o euristică constantă, acesta a adăugat și alte mici schimbări, acesta a numit algoritmul *A*/A-Star*. [43]

A-Star este unul dintre cei mai populari și puternici algoritmi de rezolvare a problemelor. Acesta este un algoritm de căutare în spațiu la nivel global, care poate fi folosit pentru a găsi soluții la multe probleme, găsirea drumului optim fiind doar unul dintre întrebuințările acestuia. [44]

Deci, pentru a putea găsi drumul optim într-un graf orientat, trebuie sa ținem cont și de aceasta euristică. Funcția euristica o sa calculeze distanța dintre doua puncte; astfel, fiecare nod, pe lângă distanța de la punctul de start la punctul curent, o să se rețină și valoarea aproximării drumului (calculata ca fiind distanța drumului deja parcurs + distanța aproximativa calculată prin funcția euristică de la punctul curent până la destinație).

Pentru a găsi drumul optim, prin algoritmul A-Star, trebuie sa urmam pașii:

- Se vor marca toate distanțele nodurilor ca fiind temporare, cu excepția nodului de start, care este permanentă; Un nod devine permanent atunci când variabila G nu are valoare infinită;
- Inițializăm variabila G pentru toate nodurile existente în graf cu "infinit";
- Inițializăm variabila G a nodului de start cu 0, iar variabila F cu distanța de la punctul de start până la destinație;
- Se va seta nodul de start ca fiind nodul activ;
- Calculăm distanțele permanente, pe care le vom nota cu G(x), unde x reprezintă nodul curent, ale tuturor vecinilor nodului activ, facem acest lucru prin adunarea tuturor distanțelor (luate ca valori ale laturilor din graf) de la nodul de start până ajungem la nodul activ. Dacă aceasta distanța calculata este mai mica decât cea curentă, distanța va fi actualizată și se va seta nodul curent ca fiind noul părinte al nodului (aceeași logica ca și la algoritmul Dijkstra);

- Calculăm distanțele totale aproximative, pe care le vom nota cu F(x), unde x reprezintă nodul curent, ale tuturor vecinilor nodului activ, facem acest lucru prin adunarea la distanța permanentă (calculata la pasul anterior), distanța calculată prin funcția euristică;
- Următorul nod care are cea mai mică distanță totală aproximată și nu a fost încă verificat devine nodul activ;
- Se vor repeta ultimii 3 pași până când nu mai sunt noduri rămase cu distanța permanentă în graf, sau daca următorul nod ce urmează a fi verificat este de fapt nodul destinație;
- Daca am ajuns la destinație se construiește drumul optim, altfel vom afișa un mesaj de eroare.

Se poate observa că prin adăugarea acestei funcții euristice, nodurile pe care algoritmul le caută ar trebui sa conveargă spre nodul destinație. Funcția euristică folosită va calcula distanța Euclidiană dintre două puncte, vom folosi acest tip de distanță deoarece nu consumă multă memorie, și este suficient de bună pentru a aproxima distanța dintre doua puncte.

Pentru a exemplifica acest algoritm, vom folosi același graf pe care l-am definit în **3.1. Dijkstra**, la care vom adăuga rezultatul funcției euristice pentru fiecare nod din graf și le vom nota H(x,y), unde x reprezintă nodul curent, iar y reprezintă nodul țintă. Vom lua ca si punct de start nodul 10, iar ca punct pentru destinație, nodul 14, astfel vom avea:

- H(1,14) = 15;
- H(2,14) = 11;
- H(3,14) = 7;
- H(4,14) = 7;
- H(5,14) = 13;
- H(6,14) = 7;
- H(7,14) = 6;
- H(8,14) = 5;
- H(9,14) = 14;
- H(10,14) = 10, nu este obligatoriu să îi știm valoarea dată de funcția euristică;
- H(11,14) = 6;
- H(12,14) = 12;
- H(13,14) = 8;
- H(14,14) = 0, deoarece nodul 14 este și nodul destinație/țința;
- H(15,14) = 9;
- H(16,14) = 3;
- H(17,14) = 4;
- H(18,14) = 12;
- H(19,14) = 7;

După prima iterație a algoritmului, o să avem toți vecinii nodului de start (10) într-o coadă de priorități în funcție de distanța totală aproximată, astfel vom avea următoarea coadă:

- Poziția 1 : F(6) = H(6,14)+G(6) = 7+6 = 13;
- Poziția 2 : F(13) = H(13,14)+G(13) = 8+6 = 14;
- Poziția 3 : F(2) = H(2,14)+G(2) = 11+4 = 15;
- Poziția 4 : F(5) = H(5,14)+G(5) = 13+3 = 16;
- Poziția 5 : F(12) = H(12,14)+G(12) = 12+5 = 17;

- Poziția 6 : F(9) = H(9,14)+G(9) = 14+6 = 20;

Din această lista vom scoate nodul cu F(x) cel mai mic, adică nodul 6. Vom verifica și pentru acest nod vecinii și vom calcula pentru ei distanța totală aproximată, urmând să le adăugăm în lista cu noduri ce urmează a fi verificate.

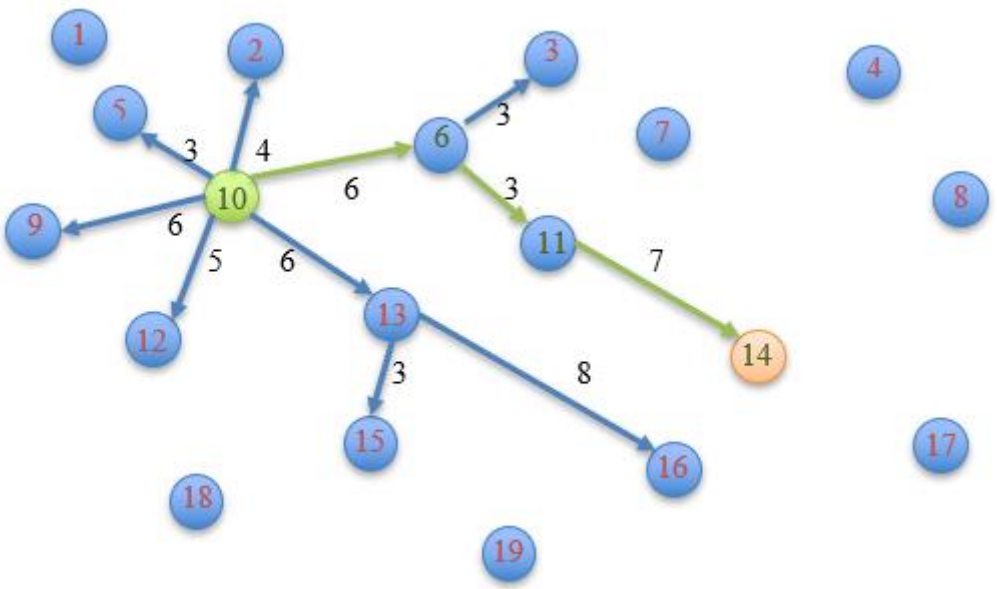
- F(11) = H(11,14)+G(6)+G(11) = 6+6+3 = 15;
- F(3) = H(3,14)+G(6)+ G(3) = 7+6+3 = 16;

Vom adăuga nodul 11 pe poziția 2 in listă (deoarece nodul 13 are distanța aproximată totala mai mica), iar nodul 3 pe poziția 5. De data aceasta nodul ce urmează a fi verificat este nodul 13, rezultând următoarele distanțe totale aproximate:

- F(16) = H(16,14)+G(13)+G(16) = 3+6+8 = 17;
- F(15) = H(15,14)+G(13)+G(15) = 9+8+9 = 26;

Vom adăuga nodurile în lista, urmând sa verificăm vecinii următorului nod cu F(x) cel mai mic, nodul ales este 11, vecinul acestui nod este destinația, deci algoritmul se oprește.

Se poate observa ca daca în acest punct nu am fi găsit nodul țintă, următorul nod verificat ar fi fost nodul 2, care se afla destul de departe de nodul destinație, deci nu întotdeauna vom verifica nodurile care se apropie de destinație.



.

Figura 3.2. Exemplu de drum optim într-un graf, găsit prin algoritmul *A-Star*.

Se poate observa că algoritmul *A-Star* se poate comporta ca și algoritmul *Dijkstra*, dacă funcția euristică ar returna 0 tot timpul, astfel vom verifica absolut toate nodurile de la nodul de start către nodul destinație ținând cont doar de distanța drumului pana la momentul respectiv.

3.3. Algoritmi bidirecționali

Atât *Dijkstra*, cât și *A-Star* pot fi particularizați astfel încât să devina algoritmi bidirecționali. Se numesc algoritmi bidirecționali, dacă aceștia caută drumul optim de la nodul de start către nodul destinație, dar și de la nodul destinație către nodul de start.

Acest mod de a căuta drumul optim poate fi mai eficient decât algoritmii clasici, deoarece atunci când avem de a face cu grafuri foarte complexe și noduri foarte îndepărtate, putem ajunge sa avem foarte multe noduri pe care trebuie să le verificăm, avansând din ce în ce mai lent către nodul țintă.

Logica din spatele algoritmilor bidirecționali este relativ aceeași, ca și a celor clasici, singura diferență este faptul că de data aceasta algoritmul se oprește când una dintre căutări ajunge să verifice un nod deja verificat de căutarea făcută în sens opus.

Daca am aplica algoritmul A-Star bidirecțional pe graful prezentat în **3.1. Dijkstra**, drumul optim va arăta ca în **Figura 3.3.**.

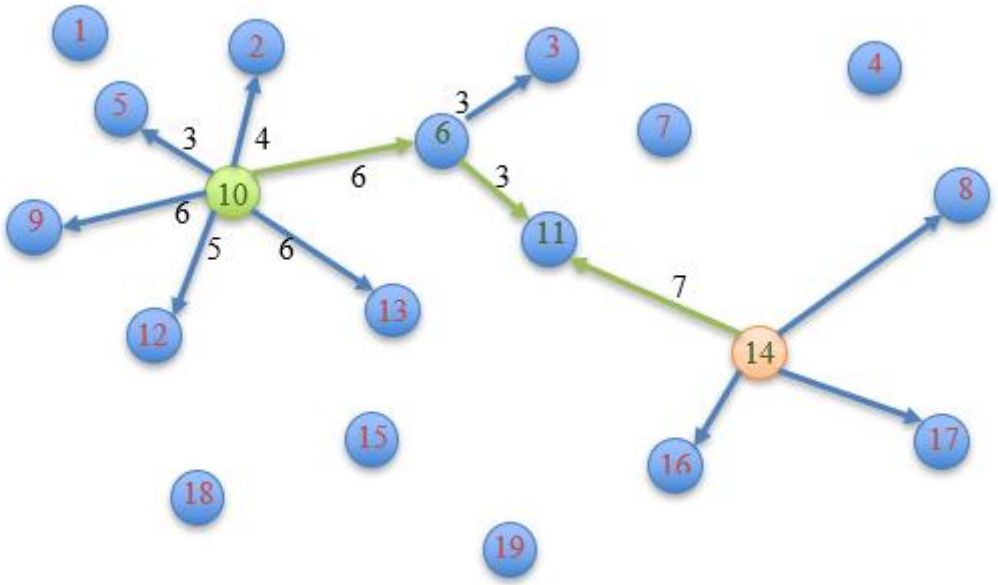


Figura 3.3. Exemplu de drum optim într-un graf, găsit prin algoritmul *A-Star Bidirecțional*.

Putem face următoarele observații:

- Pentru distanțe foarte scurte, oricare dintre algoritmii prezentați este suficient de performant (nu ar trebui sa existe timpi mari de rulare), acest lucru se poate observa în numărul de noduri pe care algoritmii îi verifica în graful dat ca exemplu: *Dijkstra* – 16 noduri, *A-Star* – 12 noduri, *A-Star Bidirecțional*– 12 noduri.
- Pentru algoritmul *A-Star*, deși teoretic nodurile pe care le căutăm ar trebui să conveargă spre nodul destinație, la distanțe lungi acesta va începe să verifice și noduri destul de îndepărtate (aproape de nodul de start).
- Prin folosirea unui algoritm bidirecțional, vom înjumătăți mărimea grafului, fiind necesară găsirea mijlocului drumului, ceea ce face mult mai rapidă găsirea drumului optim pentru distanțe mai lungi. De exemplu, în **Figura 3.3**. se poate observa ca nodurile 15 și 16 nu au mai fost verificate.
- Algoritmul *A-Star* poate avea rezultate de performanță asemănătoare cu algoritmul *Dijkstra*, dacă graful este destul de complex, deci nu este foarte eficient pe distanțe foarte mari.

3.4. New Bidirectional A-Star

Acest algoritm are ca scop îmbunătățirea algoritmului *A-Star*, și reprezintă o modificare minoră asupra algoritmului propus de *W. Pijls* și *H. Post* în lucrarea „*A new bidirectional search algorithm with shortened postprocessing*” publicat în „*European Journal of Operational Research*”. [45]

NBA-Star a fost propus ca un nou algoritm bidirecțional pentru găsirea drumului cel mai scurt de către *W. Pijls* și *H. Post* în 2008. Acesta se poate comporta ca și *A-Star* clasic dacă variabila *L* rămâne cu valoare „Infinită” și *R* rămâne o listă goala. Daca euristica rămâne 0, algoritmul devine *Dijkstra*. [45]

Variabila *L* reprezintă de fapt cea mai bună cale găsită până la un moment dat, astfel, după ce se găsește un anumit drum optim, următoarele noduri vor fi verificate daca și numai dacă sunt îndeplinite una din următoarele condiții: $G(x) + H(x, F) < L$ sau $G(x) + G(y) – H(x, S) < L$, unde *x* este nodul curent pentru căutarea curentă, *y* reprezintă nodul opus, *F* reprezintă nodul destinație, iar *S* reprezintă nodul de start. Acest lucru înseamnă că vom verifica doar nodurile cu distanța totală aproximată este mai mică decât distanța cea mai mică găsită de algoritm până în acel moment, vom avea un posibil drum mai optim.

Motivul pentru care am realizat acest algoritm este performanța susținută de lucrarea din [45], acest lucru se reflectă și în implementarea realizată în această lucrare.

Capitolul 4 – Implementarea aplicației

În acest capitol se vor prezenta pașii/etapele ce au fost urmate pentru realizarea aplicației. Acest proiect a fost realizat in 3 etape:

- **Etapa 1** – Este constituita din introducerea hărții în aplicație și adăugarea diferitelor funcționalități de bază pe care orice aplicație de acest gen ar trebui sa le aibă. Pentru această etapă ne vom folosi de bibliotecile *osmdroid* (folosită pentru a ne crea obiectul de tip *MapViewer*), *Mapsforge* (folosită pentru a ne randa harta propriu-zisă în aplicație), dar și *OsmBonusPack* (ce conține marcatorii și alte unelte necesare pentru a randa drumul optim).
- **Etapa 2** – Introducerea diverselor API-uri din care vom extrage datele de care avem nevoie. Printre acestea se numără: *OpenStreetMap* (extrage datele brute pentru o anumita arie a hărții în funcție de coordonate, este limitat la 5000 de noduri), *Overpass* (are aceeași funcționalitate ca și API-ul *OpenStreetMap*, dar nu are limitare de noduri), *OpenWeatherMap* (o sa fie folosit pentru a extrage datele de vreme), *Air Quality Programmatic* (oferă posibilitatea de a extrage, în funcție de locație, poluarea aerului), *TomTom Traffic* (o să fie folosit pentru a extrage viteza de deplasare a traficului pentru o anumita coordonata).
- **Etapa 3** – Scrierea algoritmilor de găsire al drumului optim: *Dijkstra, A-Star, Bidirectional Dijkstra, Bidirectional A-Star* și *New Bidirectional A-Star.*
- **Etapa 4** – Finalizarea aplicației prin introducerea datelor luate de la API-urile din **Etapa 2** în algoritmii de găsirea drumului optim realizați în **Etapa 3**. Îmbogățirea interfeței grafice a utilizatorului prin adăugarea unor butoane de tip „CheckBox” prin care îi vom permite utilizatorului să aleagă: Algoritmul pe care dorește să îl folosească, ce factori de mediu dorește ca algoritmul să ia în calcul etc.

Etapa 1 – Crearea activității principale

În aceasta etapa vom începe prin crearea proiectului în Android Studio, se va alege o activitate goala, limbajul de programare Java, iar API-ul minim pentru android este 28, specific versiunii Android 9.0 (*Pie*). În acest moment, aplicația nu are nicio funcționalitate, urmează să introducem biblioteca *osmdroid*.

MapView-ul oferit de *osmdroid*, este un înlocuitor al *MapView*-ului oferit de *Google*. [46]

Vom adăuga bibliotecile *osmdroid* și *mapsforge* prin *Gradle*. *Gradle* este un instrument *open-source* de automatizare, proiectat destul de flexibil pentru a construi aproape orice tip de software. [47] În acest sens vom adăuga următoarele dependențe în fișierul „buid.gradle” al aplicației:

```
dependencies {
       implementation 'org.osmdroid:osmdroid-android:6.1.5'
       implementation 'org.osmdroid:osmdroid-wms:6.1.5'
       implementation 'org.osmdroid:osmdroid-mapsforge:6.1.5'
       implementation 'org.osmdroid:osmdroid-geopackage:6.1.5'
       implementation 'org.osmdroid:osmdroid-third-party:6.0.1'
}
```

Pentru ca aplicația să poată primi date de la API trebuie să îi permitem acces la internet, dar și la locația dispozitivului. În acest sens vom adăuga următoarele permisiuni în fișierul „AndroidManifest.xml”:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Momentan nu este sugerat să realizăm operații asupra hărții, deoarece trebuie să verificăm dacă utilizatorul a oferit permisiile necesare aplicației. Pentru acest verifica permisiile vom folosi metoda *checkSelfPermission(),* iar pentru a cere permisiunea vom folosi metoda *requestPermissions().*

În acest moment avem tratate cazurile legate de permisii, deci putem continua cu pașii necesari pentru afișarea hărții în interfața grafică a aplicației. În acest scop se va crea un loc unde să afișăm harta, în fișierul „activity_main.xml*"* (fișier creat automat de Android Studio la crearea proiectului), acest element grafic este de tip *mapView*.

După ce am creat locul unde o să fie afișată harta trebuie să îl legăm de un obiect MapView pe care îl vom crea în „MainActivity.java" (acest obiect aparține bibliotecii *osmdroid*), căruia îi vom apela metoda *setTitleSource()* pentru a putea randa harta. Daca dorim să testăm aplicația, aceasta nu o să aibă foarte multă funcționalitate în acest moment, pentru început vom adăuga posibilitatea de a mări harta și posibilitatea de a folosiri atingerilor multiple, pentru a putea mări harta mai ușor.

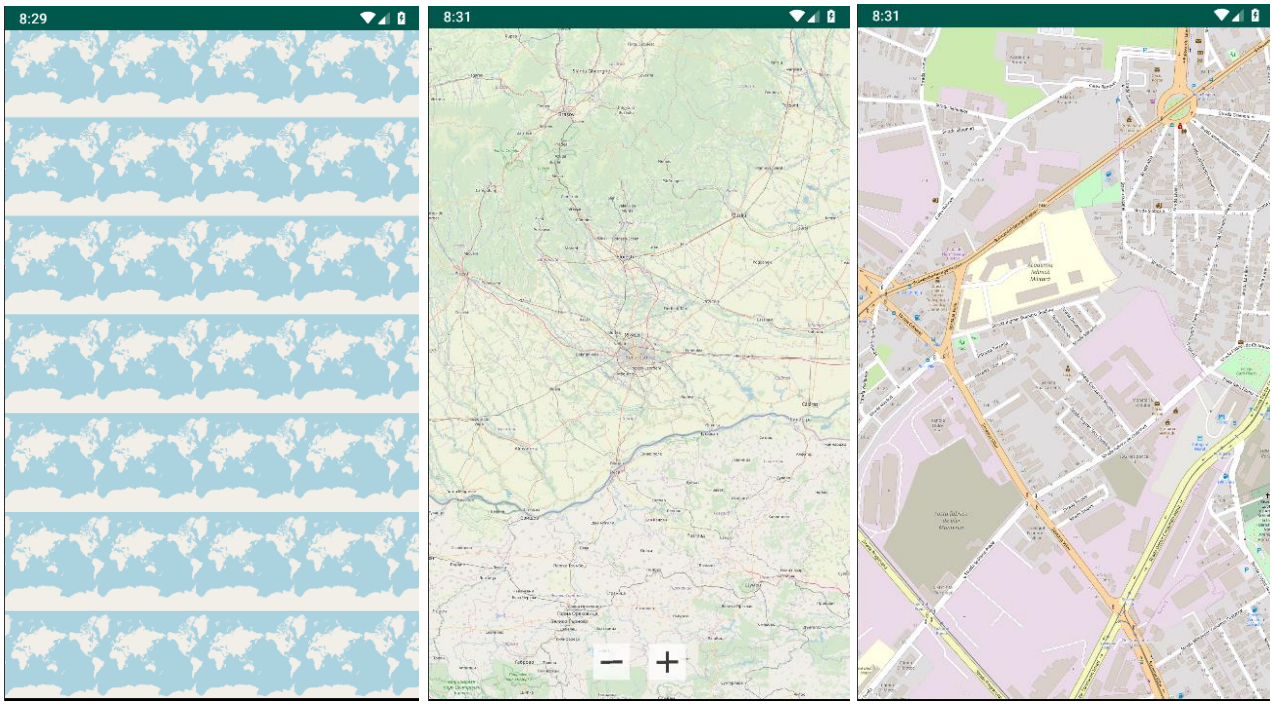


Figura 4.1. Aplicația după introducerea API-ului *OSMDroid*.

Se poate observa că aplicația este funcțională, dar nu foarte practică, de exemplu harta se poate mări la nesfârșit, dar și micșora la nesfârșit. În acest sens am adăugat următoarele caracteristici: am setat un nivel maxim de micșorare și un nivel maxim de mărire al hărții, am adăugat gesturi pentru rotirea hărții, am setat o singură hartă care se repetă doar pe orizontală. Anumite elemente vor fi adăugate ca straturi asupra hărții (de exemplu rotirea hărții sau marcatorii).

Un strat („*Overlay*") este responsabil pentru a randa conținut pe ecran. [48] După fiecare strat adăugat trebuie sa reîmprospătăm harta prin apelarea metodei *map.invalidate(),* altfel stratul nu o să fie vizibil.

Următoarea caracteristică importantă pe care orice aplicație cu hărți o are este găsirea locației utilizatorului, acest lucru se poate face în mai multe moduri: prin primirea locației prin GPS (aceasta a fost implementarea inițială, dar a funcționat forte greu) și prin *LocationServices*, ce reprezintă un serviciu de localizare pus la dispoziție de Google ce folosește serviciile Google Play. [49] În acest sens am creat un obiect de tipul *FusedLocationProviderClient*, pentru care vom adăuga o funcționalitate în cazul în care acesta o să găsească ultima locație cunoscuta. Acum aplicația o să știe ce are de făcut în cazul în care se găsește o locație, dar trebuie să trimitem cereri pentru locație constant (altfel nu vom putea găsi locația), pentru a face acest lucru vom avea nevoie să creăm următoarele obiecte: *SettingClient, LocationRequest, LocationSettingsRequest* și *LocationCallback.*

Fiecare dintre obiectele create trebuie să fie inițializate, vom face acest lucru în metodele: *buildLocationSettingsRequest(), createLocationCallback()* și *createLocationRequest().* Acum vom avea cereri pentru locație trimise constant, iar atunci când vom reuși să găsim ultima locație a utilizatorului, cu succes, clientul *FusedLocationProviderClient* o să realizeze operațiile indicate. În cazul nostru se vor reține datele locației curente, iar harta se va anima către locația curentă a utilizatorului.

O dată ce am obținut ultima locație a utilizatorului vom putea să o afișăm folosind un strat al hărții.

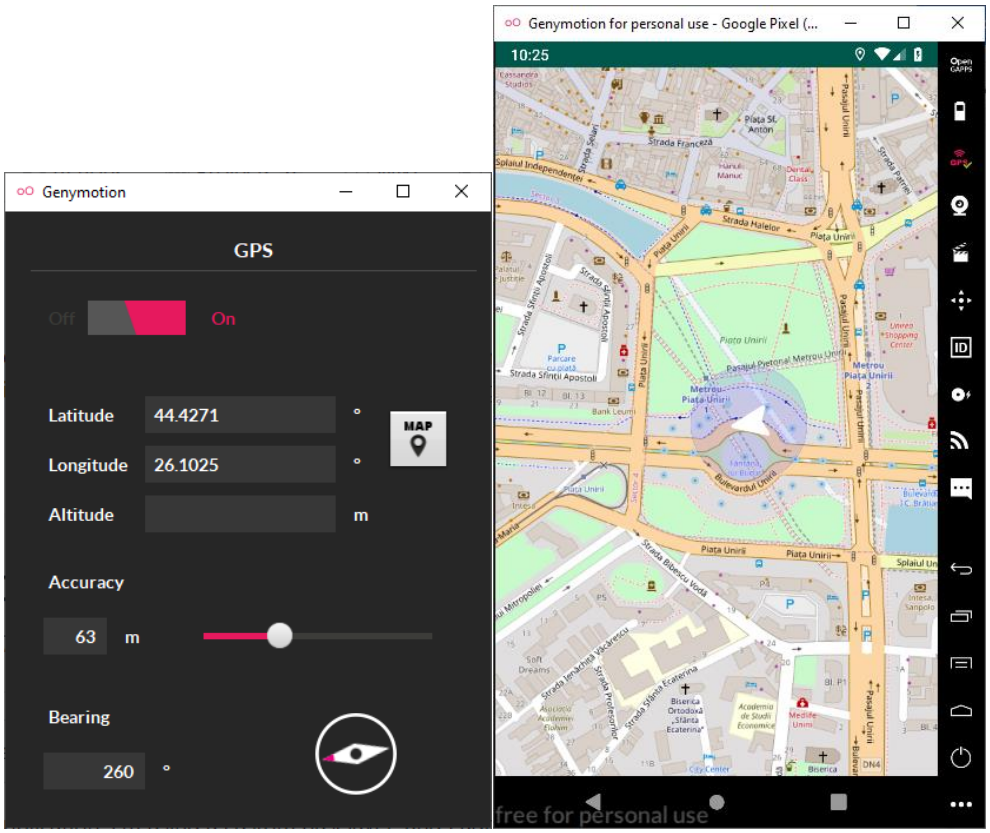


Figura 4.2. Aplicația arată locația GPS-ului (setata manual ca fiind centrul orașului București).

Etapa 2 – Introducerea datelor brute din API-uri

Există mai multe moduri prin care un programator poate prelua informațiile dintr-un API. În această etapa vom trimite cereri *HTTPS* către API-uri de tip *RESTful*, pentru a obține anumite date pe care le vom folosi ulterior în aplicație. Cererile sunt diferite pentru fiecare API, iar pentru a ști cum putem obține datele necesare vom fi nevoiți să respectăm documentația acestora.

În funcție de cererea pe care o vom face putem obține un răspuns sub forma unui obiect de tipul *JSONObject* sau sub formă de *XML*. Vom folosi pentru majoritatea API-urilor răspunsuri de tip *JSON*, deoarece sunt mai ușor de manipulat, însă anumite API-uri nu suportă un astfel de răspuns (cum ar fi *OpenStreetMap* API).

În mod normal, daca dorim să lucrăm cu datele brute obținute din API-uri, când vom face cererea de tip *HTTPS*, vom avea nevoie si de o cheie generată de către proprietarii acelui API, această cheie are rolul de a pune anumite limitări (un anumit număr de cereri/lună) pentru programatori.

Air Quality Programmatic API – oferă posibilitatea de a realiza cereri în funcție de o anumită locație dată. [50] Cerea este sub forma: *https://api.waqi.info/feed/geo:lat;lng/?token=cheie*, unde „lat" și „lng" reprezintă latitudinea, respectiv longitudinea punctului geografic pentru care dorim să realizăm cererea, iar „cheie" reprezintă cheia programatorului, primită de la producătorii API-ului, pe lângă acești parametrii există și alții, dar pe care nu îi vom folosi. [50] Pentru această cerere vom primi un răspuns de tip JSON (acest API nu poate trimite răspunsuri de tip XML), din acest răspuns avem nevoie doar de atributul „aqi" ce reprezintă nivelul de poluare din locația curentă.

Conform informațiilor din [33] putem observa că aceste date nu vor influența foarte mult găsirea drumului optim, nivelul de poluare fiind dat de anumite instituții, pe zone, deci nu vom avea un răspuns diferit pentru fiecare nod din graf (cum ar fi fost ideal).

TomTom Traffic API – asemeni API-ului prezentat anterior, acesta suportă și el cereri în funcție de o anumită locație dată, pe lângă acest lucru, putem adăuga și alți parametrii, în funcție de cerințele programatorului, în cazul aplicației prezentate o cerere este de forma: https://api.tomtom.com/traffic/services/4/flowSegmentData/absolute/10/format?key=cheie&point=lat,lng&unit=kmph, unde „format" reprezintă tipul de răspuns pe care îl dorim, „cheie" reprezintă cheia primită de programator, „lat" și „lng" reprezintă latitudinea, respectiv longitudinea punctului geografic pentru care se realizează cerea, iar „kmph" reprezintă unitatea de măsură pentru viteza traficului. [51] Pentru această aplicație a fost folosit un răspuns de tip JSON, iar datele de care avem nevoie sunt „currentTravelSpeed" și "freeFlowSpeed", avem nevoie de ambele, deoarece drumul nostru optim o să calculeze pentru fiecare nod care se afla mai aproape de viteza pe care o avem când nu este trafic pe drumul respectiv.

OpenWeatherMap API – deoarece vom fi nevoiți să reținem mai multe elemente din cadrul acestui API, am creat o nouă clasă numită *Weather,* urmând să creăm un obiect de acest tip, în activitatea principală a aplicației, în care vom reține toate elementele ce ne interesează din răspunsul API-ului. Astfel, clasa are următoarele atribute: *WeatherType* (tipul de vreme, acesta poate fi însorit, ploaie, zăpadă), *Temperature* (temperatura la momentul cererii, aceasta este in Kelvin), *Pressure, Humidity, RainVolume, SnowVolume, WindSpeed, WeatherIcon* (în funcție de tipul de vreme vom avea o iconiță diferită, aceasta o să fie afișată pe butonul de vreme din activitatea principala prin biblioteca Picasso), *AppDecision* (în funcție de parametrii prezentați, aplicația o sa afișeze o decizi, de exemplu: Nu uita umbrela, în cazul zilelor cu ploaie).

Acest API poate oferi răspunsuri doar de tipul JSON, iar cererile sunt sub forma: https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={cheie}, unde „{lat}" și „{lon}" reprezintă coordonatele pentru care se cer informațiile de vreme (se poate folosi și numele orașului), iar „{cheie}" reprezintă cheia primită de programator. Din răspunsul primit vom lua datele necesare pentru a popula obiectul de tip *Weather* creat în activitatea principală.

Pe lângă aceste date preluate din API-urile mai sus menționate, aplicația are la baza hărțile, iar cel mai important lucru este sa înțelegem cum trebuie văzută o harta pentru a putea aplica algoritmi de găsirea drumului optim pe acestea.

Cel mai simplu mod de a defini o hartă, conform dicționarului explicativ al limbii române, este: Reprezentare grafica in plan orizontal a suprafeței Pământului (totala sau parțială), generalizată și micșorată conform unei anumite scări de proporție și întocmită pe baza unei proiecții cartografice. Aceasta reprezentare grafica poate fi interpretată ca o matrice în care liniile reprezintă latitudinea, iar coloanele reprezintă longitudinea, iar fiecare element din aceasta matrice reprezintă un punct de pe Pământ, care are anumite caracteristici (apă, drum, oraș, sat etc.).

În acest moment încă nu putem aplica algoritmi de găsirea drumului optim pe aceasta matrice, ci avem nevoie de un graf orientat, acesta este obținut prin găsirea tuturor punctelor din matrice care au anumite caracteristici identice.

În final graful nostru trebuie să conțină două mulțimi, una de noduri și una de laturi.

Nodurile în cazul nostru vor fi reprezentate la intersecția de drumuri, având și alte caracteristici specifice, ce ne vor ajuta în aplicarea algoritmilor de găsirea drumului optim.

Laturile reprezintă legăturile dintre aceste noduri, acestea ne dau informații triviale precum distanța dintre noduri, care este nodul vecin nodului curent etc..

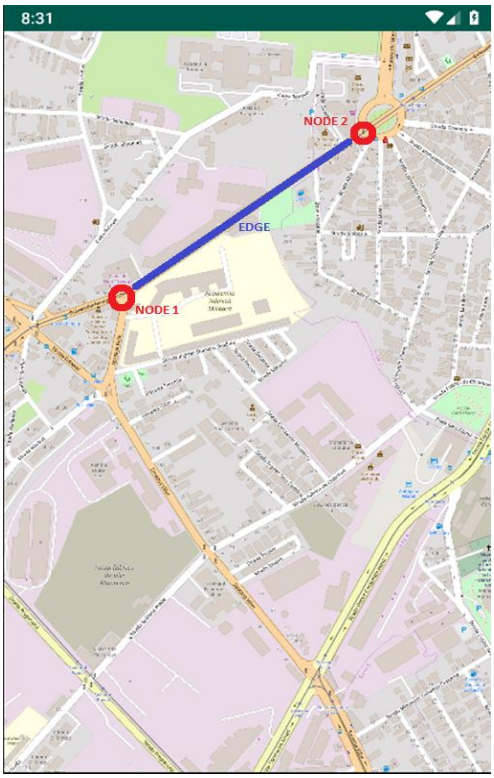


Figura 4.3. Reprezentare grafica a nodurilor si laturilor.

OpenStreetMap este un API ce oferă aceste caracteristici asupra hărții, fiind ceea ce am ales pentru a fi utilizat în proiect. Răspunsul pe care îl primim de la acest API este de tip *XML*, iar complexitatea hărții trebuie studiată mai amănunțit, în **Figura 4.3.** este un exemplu foarte simplu.

În implementarea inițială am avut local o hartă cu extensia „osm”, pe care am aplicat un script numit *osm4routing*, acesta realiza conversia din hartă în fișiere cu extensia „csv” ce conțineau direct datele pentru un graf orientat (noduri si laturi).

În fișierul *Nodes.csv* fiecare nod avea următoarele caracteristici: *index, longitudine, latitudine.* [52]

În fișierul *Edges.csv* fiecare latură are următoarele caracteristici: *index, indexul nodului sursa, indexul nodului ținta, lungimea* (în metri), *accesibilitatea mașinilor, a bicicletelor* și *a pietonilor.* [52]

Deoarece aplicația nu o să preia informațiile pentru graf în timpul rulării, am introdus un ecran secundar pe care vom afișa logoul și numele aplicației, dar și o bară de încărcare, până când aplicația va termina de analizat toate informațiile din fișierele cu extensia „csv". În acest scop am implementat o clasă publică *SearchAlgHelper*.

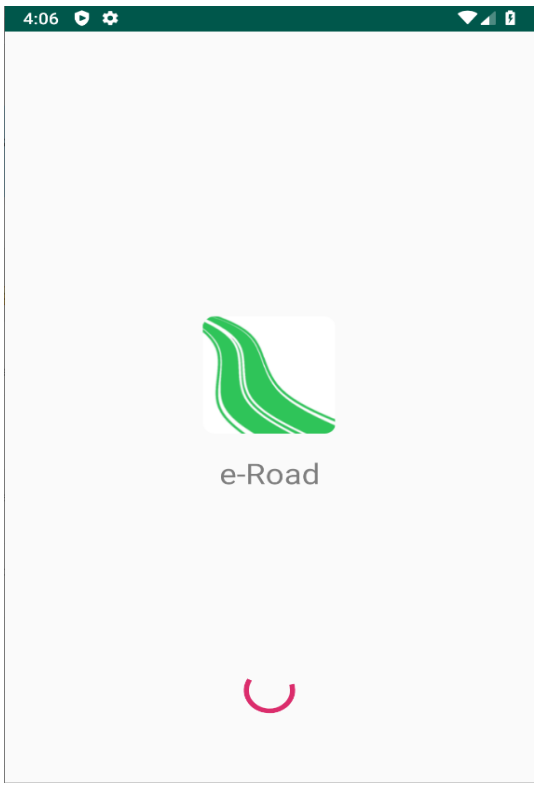


Figura 4.4. Ecranul de așteptare pentru încărcarea datelor.

Pentru a realiza graful orientat vom construi 3 clase noi: *Graph, Node* și *Way* (denumirea este preluată din API), unde obiectele de tip *Graph* vor avea două atribute statice de tip *HashMap<Long,Node>*, respectiv *HashMap<Long,Way>*, folosim această implementare și nu liste simple, deoarece indexurile pe care nodurile, respectiv laturile le au nu pot fi reținute ca *Integer*, iar metoda *get()* a listelor nu poate fi apelata cu un parametru de tip *Long*. În cele două atribute ale grafului vor fi reținute toate nodurile, respectiv laturile folosite în hartă. Atributele acestei clase sunt statice deoarece nu dorim să avem mai multe grafuri în aplicație, astfel atributele grafului vor arăta mereu către aceeași locație în memorie.

Pentru aplicație am analizat harta României și am obținut aproximativ: 1 012 034 noduri și 650 846 laturi, lucru ce îngreunează destul de mult algoritmul de găsire al drumului optim.

În **Figura 4.5.** se poate observa cum drumul optim găsit este foarte deformat, și nu consideră foarte multe căi care sunt mult mai optime decât cea găsită. Acest lucru se întâmplă deoarece graful nu este realizat conform așteptărilor. De asemenea, deoarece harta este stocată local, ci nu online (într-o bază de date), aplicația nu o să poată aplica algoritmul pe toată harta lumii.

De asemenea am obținut și alte rezultate, care din punct de vedere teoretic nu au sens, de exemplu pentru algoritmii bidirecționali căutarea drumului optim nu ia în considerare anumite laturii ce ar fi trebuit să găsească drumul optim foarte rapid, lucru ce face ca și algoritmii unidirecționali să fie mai eficienți. Pe de altă parte, pentru distanțe mai lungi de 2 km, algoritmii unidirecționali nu au reușit să găsească drumul optim, iar cei bidirecționali l-au găsit în aproximativ 2 minute, fără a mai exclude anumite laturi.

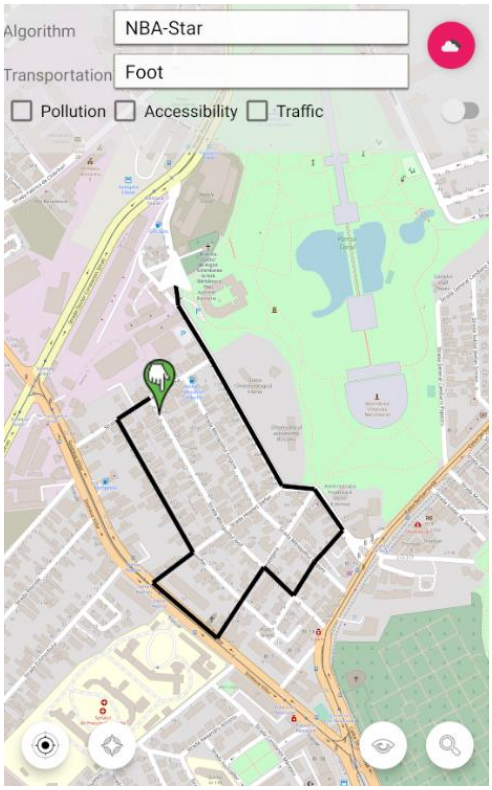


Figura 4.5. Drumul optim găsit folosind graful creat de scriptul osm4routing.

Datorită acestor motive am renunțat la implementarea respectivă și am ales să analizăm direct răspunsul *XML* pe care îl obținem de la API. Cererea pe care o trimitem va fi de tipul „casetă de încadrare" (bounding box) sau *bbox* acest tip de cerere o să trimită ca răspuns un fișier *XML* cu toate datele hărții ținând cont de anumite limite. Cererea *HTTPS* este de forma: https://api.openstreetmap.org/api/0.6/map?bbox=SouthBorder,WestBorder,NorthBorder,EastBorder/, unde „SouthBorder" este limita sudică a hărții primită ca răspuns, „WestBorder" este limita vestică, „NorthBorder" este limita nordică, iar „EastBorder" reprezintă limita estică.

Cerea prezentată mai sus este trimisă către *OpenStreetMap* API, dar aceasta acceptă un număr maxim de 5000 noduri, de aceea vom face toate cererile către *Overpass* API (acesta va da un răspuns cu aceleași date ca și *OpenStreetMap* API, dar cu mai puține restricții); cererea HTTPS este asemănătoare, doar ca are următorul prefix: http://www.overpass-api.de/api/xapi?.

Vom crea limitele casetei de încadrare astfel încât harta obținută să conțină atât nodul de start, cât și nodul destinație. În harta obținută ca răspuns vom avea toate nodurile și laturile din hartă, chiar și dacă nu avem nevoie de toate.

Pentru a prelua doar datele folositoare din răspuns vom proceda astfel:

- Deoarece la începutul fișierului vor exista doar noduri, urmate de laturi și relații (relațiile nu sunt interesante în cazul aplicației), vom prelua datele de la toate nodurile și le vom reține

într-un *HashMap* numit *AllNodes*. Nu vom folosi toate nodurile în graful final deoarece anumite noduri pot să reprezinte monumente/muzee deci nu vor fi legate de niciun alt nod printr-o latură.

- Urmează să preluam laturile, nici de data aceasta nu avem nevoie de toate datele primite de la API, deoarece anumite laturi pot reprezenta granițe, sau locuri prin care nu poți merge. Pentru a prelua doar laturile care contează trebuie să ne uităm la etichetele laturii. Pentru a crea un graf al drumurilor, vom prelua doar laturile ce au ca etichetate cheia *k=„highway”*. Pentru a face graful mai realist (să ținem cont de drumuri pentru biciclete, pentru mașini), trebuie să ținem cont și de valoarea *v* pe care o găsim asociată cu cheia mai sus specificată.
- Toate laturile utile vor fi adăugate într-un *HashMap* numit *UsedWays.* Pentru laturile ce au fost găsite ca fiind utile vom reține indexurile nodurilor ce fac parte din acea latură (pentru aceste noduri se vor lua mai multe detalii din *HashMap*-ul *AllNodes* și vor fi puse într-un nou *HashMap* numit *UsedNodes*). Se poate observa că o latură conține mai multe noduri (nu doar un nod de start și unul destinație), iar lungimea laturii nu este dată de răspunsul dat de API.
- Vom seta graful prin apelarea metoda *setGraph()* folosind ca parametrii listele *UsedNodes* și *UsedWays,* acestea vor reprezenta graful orientat pe care vom aplica algoritmii folosiți la găsirea drumului optim.

Figura 4.6. Reprezentare grafică realistă a nodurilor și laturilor.

În **Figura 4.6.** este o reprezentare mai realistă a laturilor existente în răspunsul dat de *OpenStreetMap* API sau *Overpass* API, unde fiecare punct roșu reprezintă un nod. A se observa faptul că există și o graniță ce este văzută ca și latură, dar reprezintă de fapt o clădire și nu trebuie luată în calcul la realizarea drumului optim.

Este foarte important să extragem toate datele de care avem nevoie pentru graf în timpul analizei răspunsului primit de la API, de exemplu datele ce ne permit să avem un graf special pentru mersul pe jos, pentru transportul cu mașina sau chiar bicicleta.

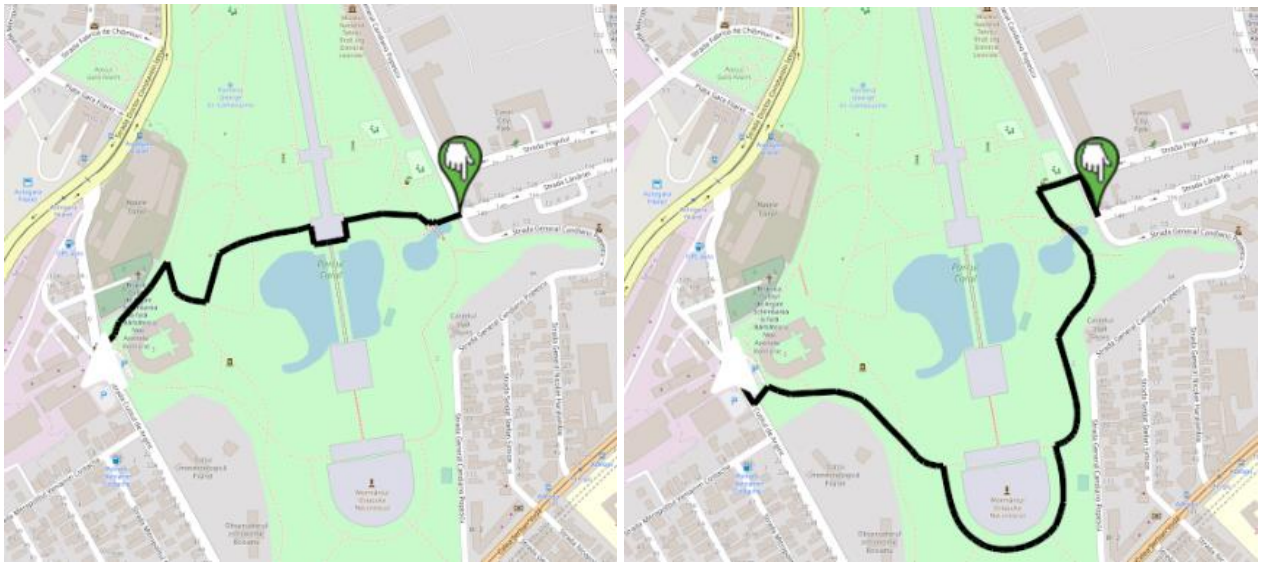


Figura 4.7. Drumul optim găsit prin mersul pe jos (stânga), respectiv prin mersul cu bicicleta (dreapta).

De asemenea, în funcție de tipul drumului putem adăuga și un alt atribut laturilor, denumit *Accessibility*, ce reprezintă cât de accesibil este un drum în funcție de metoda de transport aleasă. De exemplu mașinile au un grad de accesibilitate mai mare pe drumurile principale decât secundare.

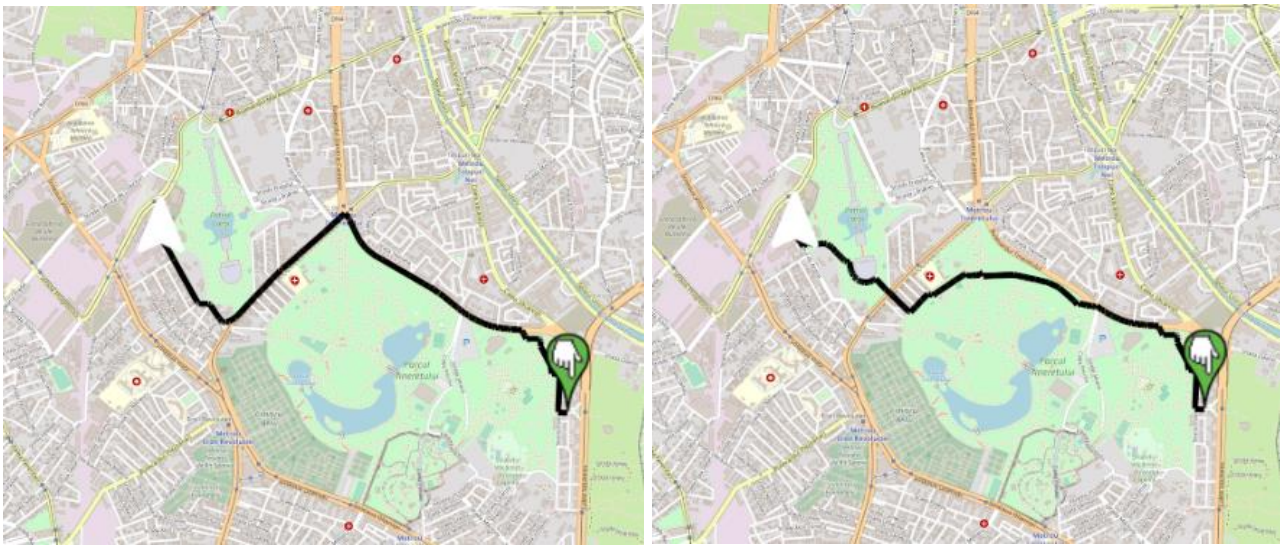


Figura 4.8. Drumul optim pentru metoda de transport „Bicycle”, când se ține cont de accesibilitate (stânga), dar și când nu se ține cont de accesibilitatea (dreapta)**.**

În **Figura 4.8.** se poate observa cum influențează accesibilitatea, crearea drumului optim, algoritmul va prioritiza drumul pe unde exista loc special amenajat pentru drumul cu biciclete, urmat de drumurile în care nu există și pietoni, având pe ultimele locuri drumurile ce sunt împărțite și de pietoni (de exemplu parcurile) și drumurile forestiere (nepavate).

Etapa 3 – Scrierea algoritmilor de găsirea drumului optim

Pentru găsirea drumurilor optime putem proceda în mai multe moduri: prin folosirea obiectelor de tip *RoadManager* puse la dispoziție prin biblioteci, prin folosirea unor API-uri special create în acest scop, cum ar fi *GraphHopper* (acesta poate realiza căutări și offline) sau prin implementarea specifică a unor algoritmi pentru găsirea drumului optim (necesită și crearea unui graf orientat pe care se vor aplica algoritmii).

În această aplicație am ales să realizăm a treia variantă, și anume implementarea algoritmilor de găsirea drumului optim: *Dijkstra, A-Star, Dijkstra Bidirecțional, A-Star Bidirecțional* și *NBA-Star*. Acești algoritmi au fost discutați din punct de vedere teoretic în **Capitolul 3**, în această parte a lucrării ne vom concentra asupra implementării în limbaj de programare a algoritmilor folosind structuri de date și caracteristici specifice programării orientate pe obiect.

Găsirea drumului optim folosind alte biblioteci poate fi observată în **Figura 4.9.** algoritmii implementați ar trebui, din punct de vedere teoretic, să găsească aceleași drumuri optime, sunt posibile mici diferențe, în funcție de modul în care a fost conceput graful orientat, modul de calcul al funcției euristice (în cazul algoritmului A-Star) etc..

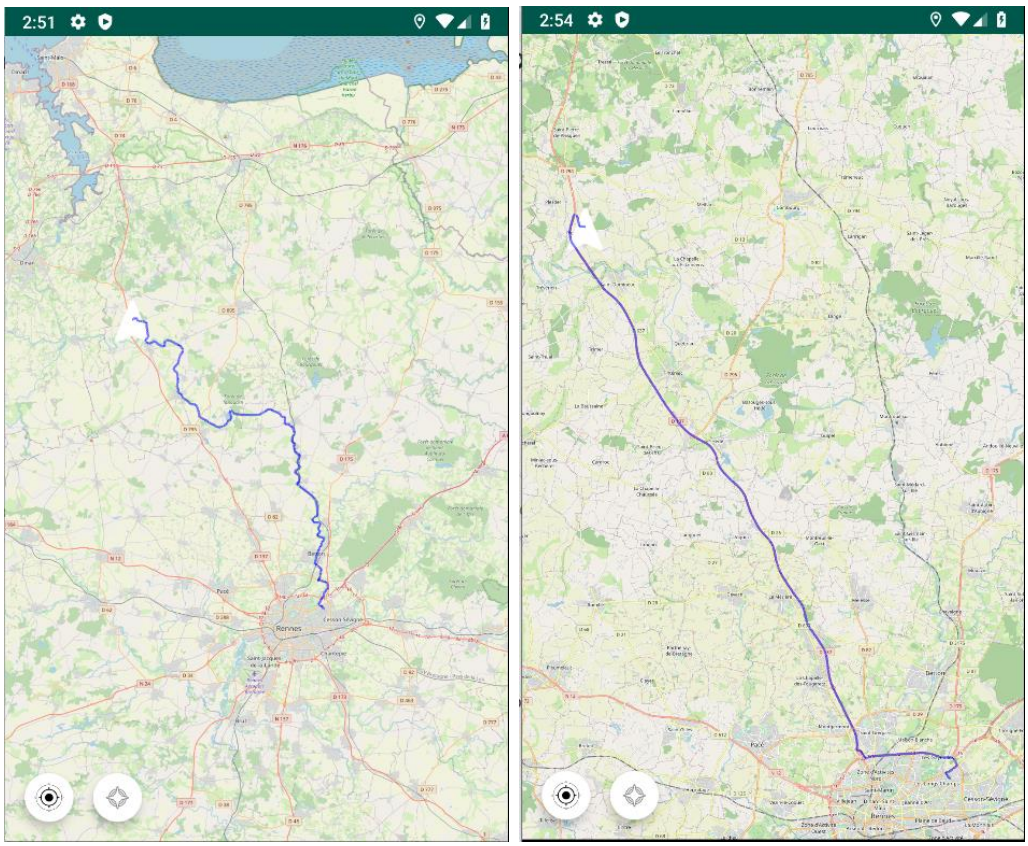


Figura 4.9. Ruta găsită prin obiectele de tip RoadManager create prin biblioteca OsmBonusPack (stânga) și prin biblioteca MapQuest (dreapta).

Algoritmii implementați vor fi realizați ca și metode ale unei clase publice denumită *Search*, cel mai important atribut al acestei clase este lista privată și statică de noduri denumită *Road*, în care vom reține toate nodurile prin care vom construi drumul optim, pentru a accesa această metodă din activitatea principală vom apela metoda *getRoad()*. Aceasta clasă are și alte atribute private, acestea

sunt folosite de metodele algoritmilor în funcție de nevoi (cum ar fi *ConnectingNode* care este folosit de algoritmii bidirecționali)

Graful pe care vom aplica algoritmii pentru găsirea drumului optim este cel analizat în **Etapa 2**, doar ca va trebui să adăugăm anumite atribute clasei specifice nodurilor, aceste atribute se numesc „G” și „F”, unde „G” reprezintă distanța drumului până la nodul curent, iar „F” reprezintă o aproximare a drumului total pe care îl putem găsi (acest atribut este specific algoritmului *A-Star*) și se calculează ca fiind: $F = G + H$, unde H reprezintă rezultatul funcției euristice (în cazul acestei aplicații, funcția euristică returnează distanța euclidiană dintre nodul curent și nodul destinație).

În cazul algoritmului *Dijkstra* și *Dijkstra Bidirecțional* atributul „F” nu ne interesează, deoarece dorim să căutăm toate nodurile existente în graf, până la găsirea drumului optim.

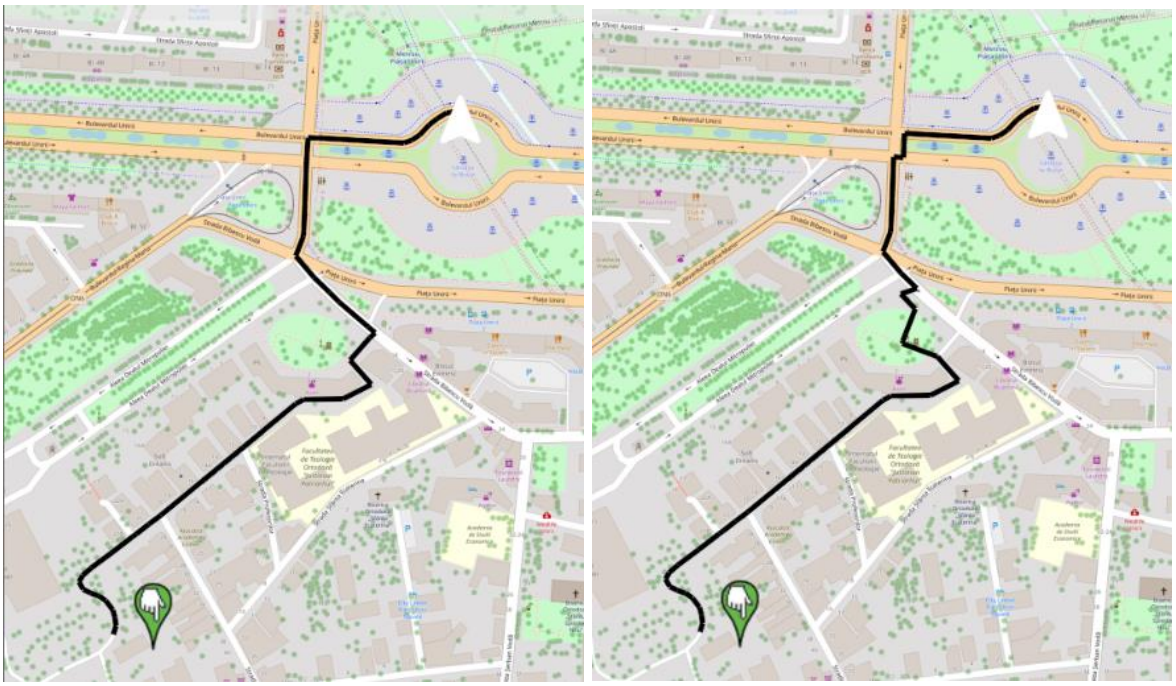


Figura 4.10. Găsirea drumului optim prin algoritmul Dijkstra (stânga) și prin algoritmul A-Star (dreapta).

Se poate observa în **Figura 4.10.** cum algoritmul *A-Star* tinde să mearga caute punctele mai apropiate de nodul țintă, chiar și dacă drumul rezultat este puțin mai lung, acesta are performanțe per total mai bune (numarul nodurilor căutate este mai mic).

Vom avea două metode de construcție al drumului optim (la finalul algoritmilor), una pentru căutarea unidirecțională și una pentru căutarea bidirecțională, acestea se numesc: *BuildUnidirectionalPath()* și *BuildBidirectionalPath()*.

Înainte de a începe procesarea propriu-zisă a grafului (găsirea drumului optim), trebuie să căutăm cel mai apropiat nod de locația de *Start*, respectiv *Finish*, nodurile pe care vom rula algoritmul nu vor aparține grafului, deci niciun drum nu o să fie găsit, în acest scop am implementat metoda *FindClosestGraphNode()* care returnează cel mai apropiat nod față de un nod specificat, calculând distanțele dintre toate nodurile din graf și alegându-l pe cel mai apropiat. Această metoda este folosită de absolut toți algoritmii implementați, o altă metodă care, de asemenea, este folosită de toți algoritmii se numește *EstimateDistance()*, pe care o folosim să calculăm distanța/„greutatea” dintre două noduri succesive.

Deoarece API-ul din care extragem datele pentru a crea graful nu ne oferă distanța reală dintre două noduri suntem nevoiți să implementăm singuri o metodă care să facă acest lucru. Pentru a nu folosii foarte multă putere de calcul, vom evita folosirea unei funcții precum funcția Haversine, ci este suficientă distanța euclidiană, dar trebuie să fim atenți deoarece rezultatul distanței dintre două noduri este foarte mic (de ordinul 10^{-5}), acest lucru este important deoarece in metoda pentru estimarea „greutății" laturii dorim să adăugăm și datele preluate din *Air Quality Programmatic API*, dar și din *TomTom Traffic API*. Nu putem lua în considerare doar poluarea aerului, ci trebuie să găsim o modalitate prin care să luăm în calcul și distanța parcursă (ar fi nedrept ca un drum de 40 de metri să fie ales în defavoarea unui drum de 20 de metrii, doar pentru ca are nivelul de poluare mai mic).

Cu scopul de a calcula cât mai corect „greutatea" dintre două noduri, am ales să înmulțesc toate datele colectate din API-uri cu 10^{-6} (deoarece acestea au șanse de a extrage date cu două cifre), iar accesibilitatea cu 10^{-5}, în acest fel, cu cât rezultatele obținute din API-uri sunt mai apropiate de zero cu atât sunt mai ideale. Inițial această hotărâre trebuia să fie realizată de algoritmul Minimax, dar acesta nu a mai fost implementat, pentru a nu folosi foarte multă putere de calcul.

Diferența de implementare între algoritmii clasici și variantele lor bidirecționale este modul în care aceștia se opresc. Pentru algoritmii clasici algoritmii se opresc când se ajunge la nodul destinație, în schimb, algoritmii bidirecționali se opresc atunci când ambele direcții de căutare au un punct comun (denumit în aplicație *ConnectingNode*).

Algoritmul *NBA-Star* are aceeași logică cu cea a algoritmului *A-Star bidirecțional*, dar introduce niște elemente noi, cum ar fi *BestPathLength*, aceasta are rolul de a micșora numărul de noduri căutate (atât timp cât atributul „F" este mai mare decât *BestPathLength* vom sări la următorul nod).

În timpul căutării se vor urma pașii explicați în **Capitolul 3 – Algoritmi de găsirea drumului optim**.

Etapa 4 – Finalizarea aplicației

În acest moment aplicația poate extrage date din API-uri și poate realiza găsirea drumului optim folosind algoritmii: *Dijkstra, A-Star, Dijkstra Bidirecțional, A-Star Bidirecțional* și *NBA-Star*, dar un utilizator obișnuit nu poate realiza aceste operații, deoarece nu există destule elemente grafice în aplicație.

Pentru a nu încărca interfața grafică, am ales sa nu implementez nicio căsuță text pentru a specifica destinația, în schimb utilizatorul va plasa un marcator la destinație, drumul optim fiind căutat între locația curentă și locația marcatorului.

Pentru a putea lucra cu marcatori, trebuie să introducem și biblioteca *OsmBonusPack*, în acest sens vom adăuga următoarea dependență in fișierul de Gradel al proiectului:

```
dependencies {
     implementation 'com.github.MKergall:osmbonuspack:6.6.0'
}
```

Acum putem lucra cu marcatori pe hartă, însă pentru a îi putea adăuga la atingere, mai întâi trebuie să creăm un nou strat („*overlay*") pentru hartă, pentru acest strat, atunci când se detectează o atingere se va realiza o acțiune (adăugarea marcatorului, în cazul actual).

Am ales să adaug moduri prin care se pot citi direct datele brute colectate din API-uri (*Toggle*-ul și butonul pentru vizualizarea rezultatelor) pentru a putea colecta datele mai ușor astfel încât să se poată trage o concluzie.

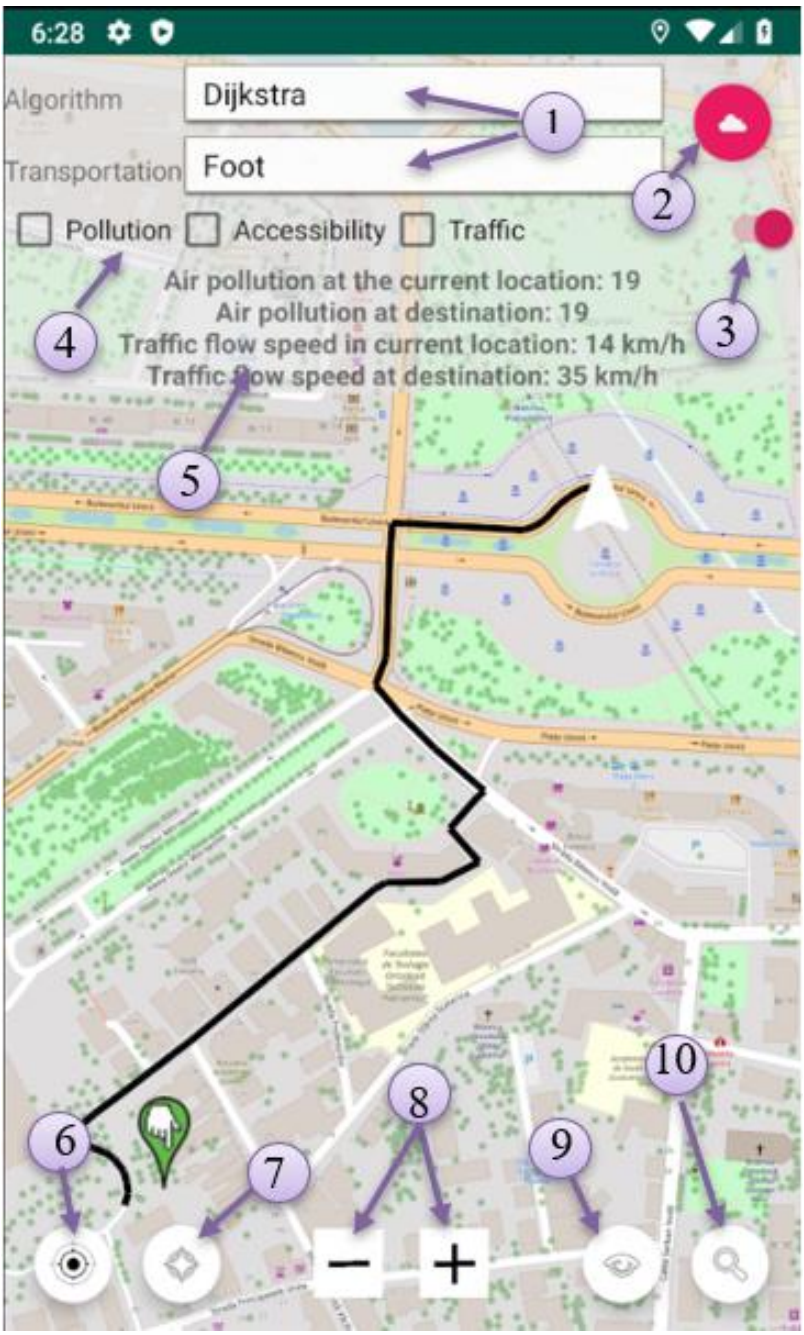


Figura 4.11. Interfața grafică a aplicației finalizate.

În continuare vom prezenta interfața grafică a aplicației, făcând referiri la **Figura 4.11.**:

1. Obiect de tip *Spinner* pentru alegerea algoritmului prin care utilizatorul dorește să găsească drumul optim, respectiv pentru alegerea metodei de transport (cu picioarele, mașina sau bicicleta);
2. Buton pentru vreme, acesta o să afișeze detaliile legate de vremea din locația curentă (în funcție de prognoză iconița butonului este schimbată). Detaliile sunt afișate într-un pop-up (vezi **Figura 4.12.**);

3. *Toggle*, pentru afișarea elementelor din *TextView*-ul prezentat la punctul 5, am ales această implementare pentru a nu avea o interfață foarte aglomerată;
4. *Checkbox*-uri ce permit utilizatorului să aleagă parametrii prin care se va construi drumul optim;
5. *TextView* pentru prezentarea detaliilor legate de trafic, dar și poluare de la punctul curent, cât și la destinație;
6. Buton pentru centrarea hărții în locația utilizatorului;
7. Buton pentru revenirea la 0^o cu rotația hărții;
8. Butoane pentru mărirea, respectiv micșorarea hărții;
9. Buton pentru vizualizarea rezultatelor în urma rulării algoritmilor de găsirea drumului optim (acesta nu afișează nimic, dacă nu este un drum afișat în aplicație;
10. Buton pentru începerea căutării (o căutare a drumului optim se realizează între locația curentă a utilizatorului și destinația aleasă prin marcator de către utilizator).

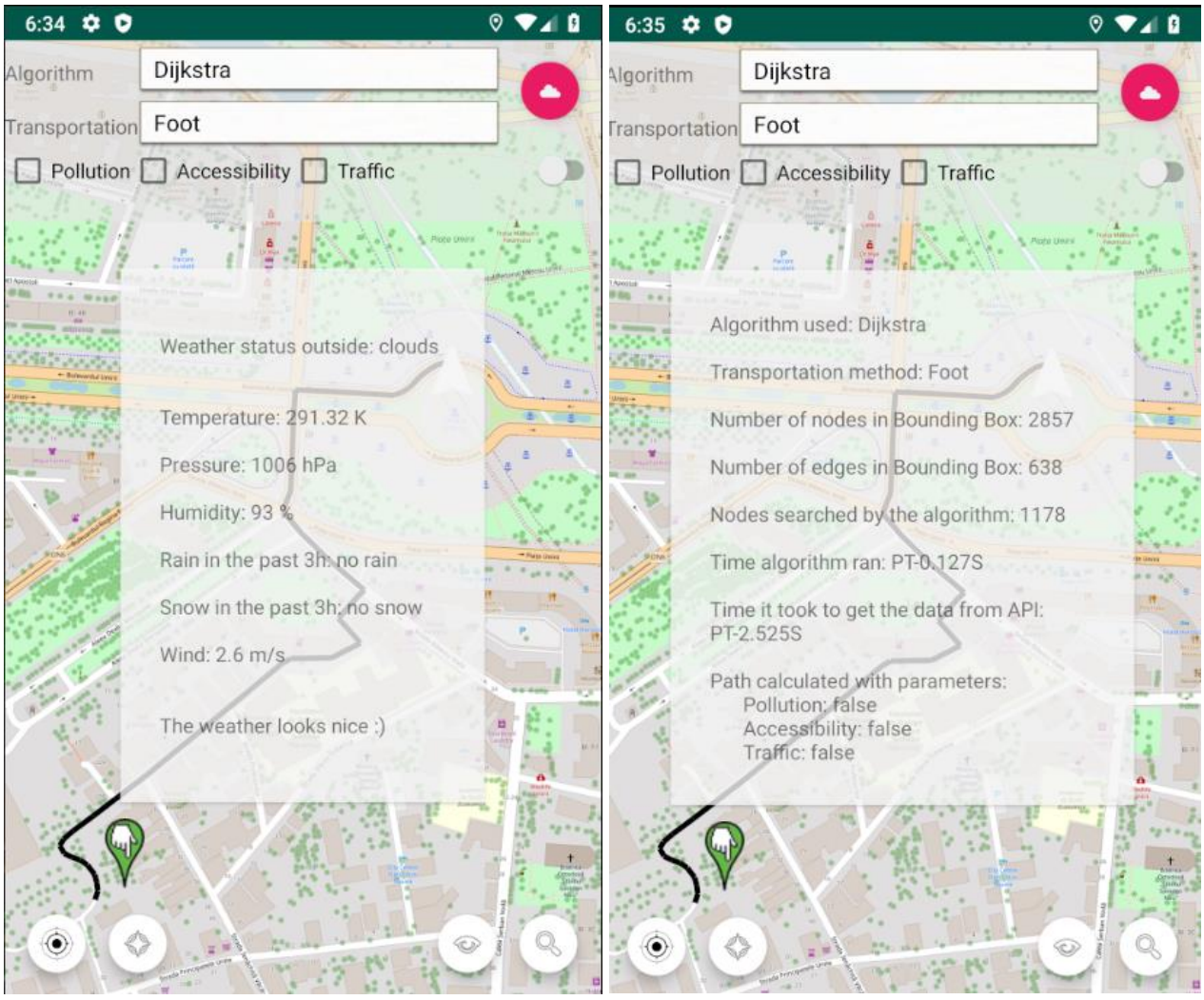


Figura 4.12. Ferestrele de pop-up ce apar la apăsarea butonului de vreme (stânga), respectiv la apăsarea butonului de vizualizare a rezultatelor (dreapta).

Adăugarea elementelor grafice prezentate mai sus, se fac în fișiere separate, față de cele în care introducem logică aplicației. Aceste fișiere au extensia „XML”, iar obiectele din aceste fișiere

(*Buttons*, *CheckBoxes*, *Switches* etc.) au un anumit *Id* setat de către programator. Pentru a putea adăuga logică în spatele acestor elemente trebuie să le îmbinăm cu elemente Java.

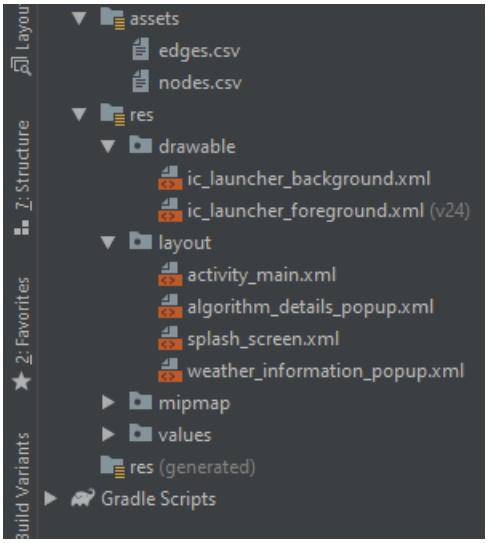


Figura 4.13. Fișierele XML din structura finală a proiectului.

Pentru a colecta datele necesare pentru vizualizarea rezultatelor am realizat o nouă clasa *StaticHelper* care are atribute statice, pe care le putem folosi pentru a citi rezultatele algoritmilor și să le aducem în activitatea principală, de asemenea am folosit această clasă pentru a transmite algoritmilor anumite date din activitatea principală (cum ar fi numele algoritmului pe care dorim să îl folosim), dar și o variabilă ce poate fi setată doar de către programator, numită *ChoseImplementationType*, care are rolul de a selecta modul în care vom crea graful (implementarea prin fișiere „csv" locale sau implementare prin care trimitem cereri *HTTPS* de la care vom primi un răspuns *XML*).

Deși implementarea curentă nu necesită încărcarea datelor stocate local, am păstrat ecranul de încărcare, deoarece dacă se dorește folosirea acelei implementări acest lucru este posibil.

De asemenea, am realizat o reducere paralelă asupra anumitor taskuri, rulând-u-le pe un *Thread* secundar, față de activitatea principală, printre aceste task-uri se numără rularea algoritmilor, adăugarea marcărilor, încărcarea datelor când deschidem aplicația. Această paralelizare este implementată prin metodele *Run**«NumeTask»**OnSecondaryThread()*, ce vor face o instanță asupra unui obiect de tip *Thread* și îl vor porni.

```
protected void AddMarkersOnTapOnSecondaryThread(final Context mContext){
    Thread mThread = new Thread(new Runnable() {
        public void run() {
            AddMarkersOnTap(mContext);
        }
    });
    mThread.start();
}

protected void RunSearchAlgorithmOnSecondaryThread() {
    Thread mThread = new Thread(new Runnable() {
        public void run() {
            findViewById(R.id.SearchShortestPath).setClickable(false);
            ColorStateList Gray=ColorStateList.valueOf(Color.parseColor("#d3d3d3"));
            findViewById(R.id.SearchShortestPath).setBackgroundTintList(Gray);
            RunSearchAlgorithm();
            findViewById(R.id.SearchShortestPath).setClickable(true);
```

```
                ColorStateList White=ColorStateList.valueOf(Color.parseColor("#ffffff"));
                findViewById(R.id.SearchShortestPath).setBackgroundTintList(White);
            }
        });
        mThread.start();
    }
```

Metodele ce vor rula pe al doilea *Thread*, în cazul acesta vor fi *AddMarkersOnTap()* și *RunSearchAlgoritm()*, daca nu facem acest lucru aplicația se va bloca și nu ar putea fi folosită de către utilizator.

Codul sursă al proiectului poate fi vizualizat în **Anexa 1 – Codul sursă**, aceasta conține toate implementările prezentate în acest capitol.

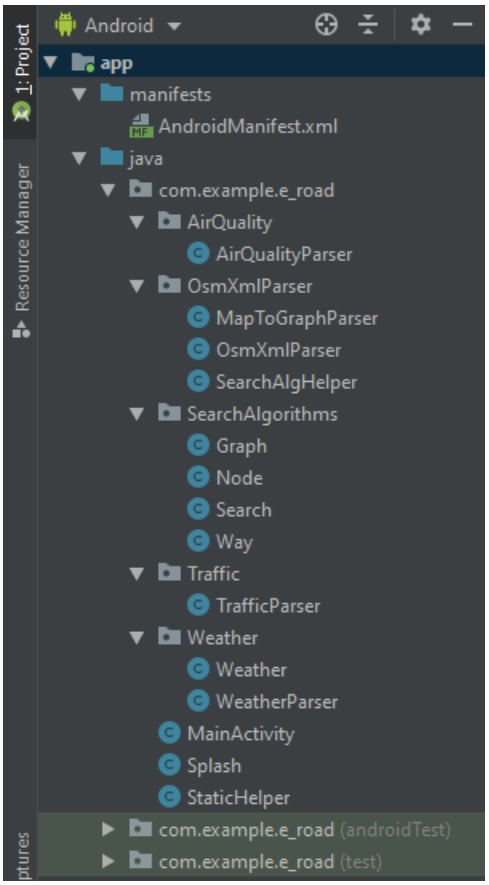


Figura 4.14. Fișierele Java din structura finală a proiectului.

Capitolul 5 – Rezultate practice

În acest capitol se vor discuta rezultatele obținute în timpul implementării aplicației și motivele ce au dus la modul final în care aceasta a fost implementată.

În primul rând vom discuta rezultatele obținute atunci când graful orientat a fost realizat prin intermediul scriptului *osm4routing*. Toate rezultatele pentru această implementare nu țin cont de tipul de transport folosit.

Algoritm	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Dijkstra	14	20.363	1 012 034	650 846	35.937
A-Star	14	20.4	1 012 034	650 846	35.937
Dijkstra bidirecțional	112	174.492	1 012 034	650 846	35.937
A-Star bidirecțional	94	150.012	1 012 034	650 846	35.937
NBA-Star	48	88.661	1 012 034	650 846	35.937

Tabelul 5.1. Rezultatele algoritmilor pentru un drum foarte scurt (distanța este mai mică de 500 m).

Se poate observa cum algoritmii bidirecționali au un timp mai mare de rulare. În **Figura 5.1.** se poate observa modul în care acest drum ar trebui să arate. Acesta a fost găsit prin algoritmii unidirecționali.

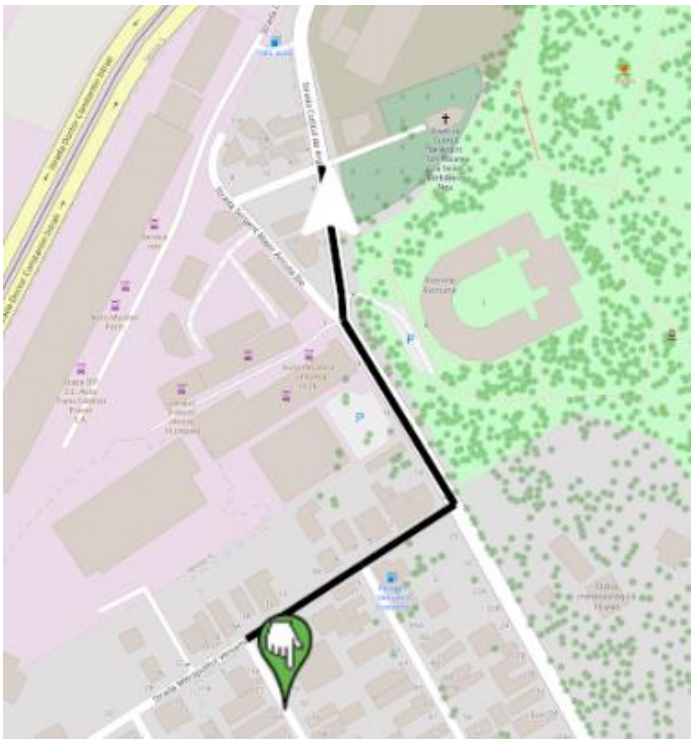


Figura 5.1. Drumul foarte scurt găsit prin algoritmi unidirecționali.

În figura următoare se poate observa de ce timpul de rulare crește o dată cu folosirea algoritmilor bidirecționali. Motivul pentru care algoritmii crează un drum mai lung este datorită grafului pe care îl folosim; în timpul căutării latura care ar fi legat cele două căutări nu este luată în calcul.

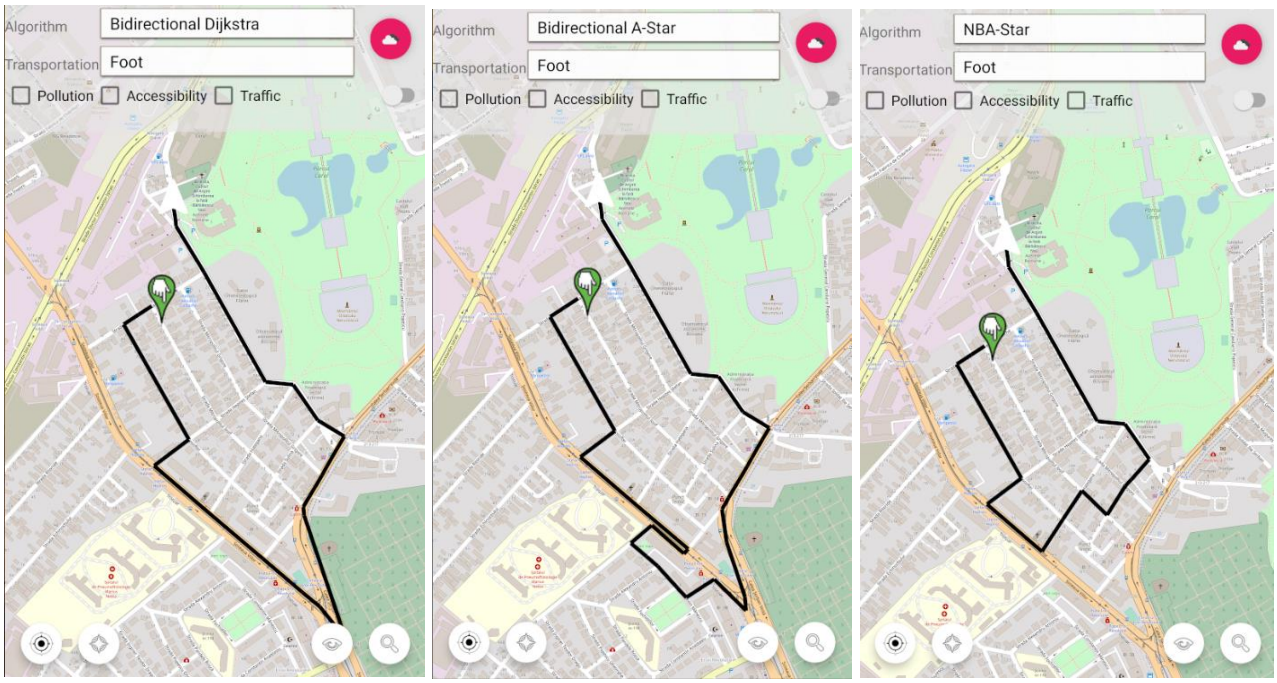


Figura 5.2. Drumul foarte scurt găsit prin algoritmi bidirecționali.

Din **Tabelul 5.1.**, putem trage concluzia încă de pe acum, ca această metodă de implementare a algoritmilor este ineficientă, deoarece dorim ca algoritmii să funcționeze și pe drumuri foarte mari. Un avantaj al acestui tip de implementare ar putea fi timpul de preluare al datelor din API, care nu variază (aceste date sunt preluate la intrarea în aplicație).

De asemenea, se poate observa că graful obținut nu este așa cum ne-am aștepta pentru toți algoritmii, deoarece algoritmul nu ia în calcul anumite drumuri drepte, care în funcție de caz, sunt mai eficiente; acest lucru indică lipsa anumitor laturi în graf.

Algoritm	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Dijkstra	-	-	1 012 034	650 846	39.621
A-Star	-	-	1 012 034	650 846	39.621
Dijkstra bidirecțional	332	505.182	1 012 034	650 846	39.621
A-Star bidirecțional	54	99.846	1 012 034	650 846	39.621
NBA-Star	54	99.254	1 012 034	650 846	39.621

Tabelul 5.2. Rezultatele algoritmilor pentru un drum scurt (distanța este de aproximativ 2 km).

În **Tabelul 5.2**. se pot observa rezultatele algoritmilor de căutare pe o distanță mai mare, față de primul caz.

Din datele experimentale extrase se poate vedea îmbunătățirea drastică a algoritmilor bidirecționali, aceștia fiind capabili să găsească drumul optim în mai puțin de 2 minute (pentru *A-Star Bidirecțional* și *NBA-Star*), iar algoritmii unidirecționali nu au reușit să obțină niciun rezultat, timpul de rulare după care au fost opriți fiind de aproximativ o oră. De asemenea se poate observa și că algoritmul *Dijkstra* bidirecțional a ajuns să aibă un timp de rulare ridicat (aproximativ 9 minute).

Timpul de preluare a datelor din API a crescut, dar acest lucru nu se datorează lungimii drumului, ci puterii de calcul a procesorului în acel moment.

Datele prezentate mai sus sunt reprezentative pentru drumul din **Figura 5.3.**. Se poate observa cum acest mod de a prelucra datele brute din *OSM* este foarte ineficient, găsirea drumurilor foarte lungi (între orașe) este aproape imposibil de realizat, într-un timp favorabil pentru o aplicație de acest gen, având rezultate uneori foarte greșite (vezi **Figura 5.2**). Totodată putem observa faptul că pentru distanțele mai lungi algoritmii par să găsească drumul optim mai bine (nu mai ocolesc anumite părți ale drumului).

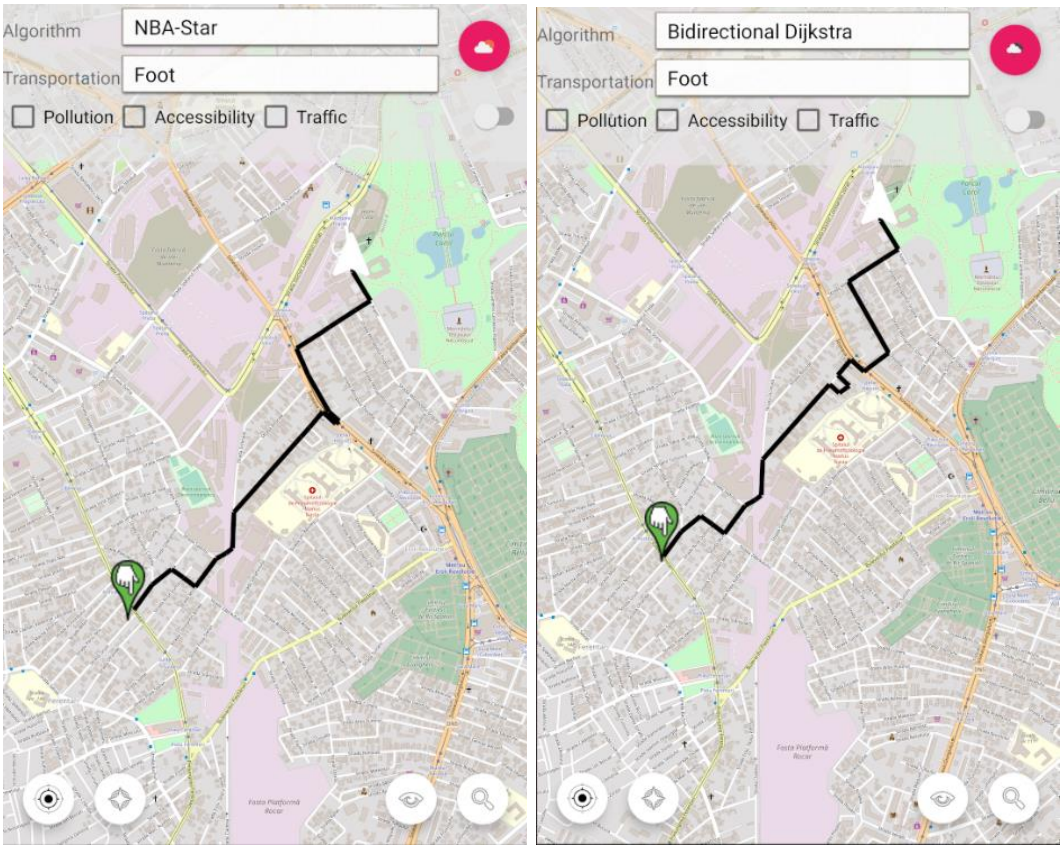


Figura 5.3. Drumul cu lungimea de aproximativ 2 km găsit prin algoritmul NBA-Star (stânga) și Dijkstra bidirecțional (dreapta).

Din aceste motive am ales să renunț la această implementare și să analizez singur datele brute primite ca răspuns al *OpenStreetMap* API, respectiv *Overpass* API.

Limitarea de 5000 de noduri a API-ului *OpenStreetMap* nu ne permite să căutăm drumul optim pentru distanțe mari. În acest sens, am ales ca datele de la *OpenStreetMap* să fie preluate prin *Overpass* API. Rezultatele următoare nu au ținut cont de poluarea aerului, accesibilitatea drumului sau de trafic, singurul criteriu fiind distanța dintre puncte și modalitatea de transport aleasă.

Algoritm	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Dijkstra	602	0.02	926	175	6.509
A-Star	195	0.007	926	175	5.596
Dijkstra bidirecțional	354	0.012	926	175	2.156
A-Star bidirecțional	138	0.005	926	175	4.465
NBA-Star	132	0.006	926	175	2.382

Tabelul 5.3. Rezultatele algoritmilor pentru metoda de transport „Foot” (pe jos) pentru un drum scurt.

Se poate observa că nu există diferențe semnificative de performanță între algoritmi, dar deja ne putem face o părere cu privire la care dintre algoritmii implementați este cel mai performant. Pe primele locuri fiind *A-Star bidirecțional* și *NBA-Star*, iar algoritmii cei mai ineficienți vor fi *Dijkstra* (ce a căutat peste jumătate din numărul total de noduri) și *Dijkstra Bidirecțional*. De asemenea, se poate observa că timpul de preluare al datelor din API variază deși dimensiunea grafului nu este schimbată, uneori este posibil să nu primim un răspuns de la API (dacă timpul de a primi un răspuns de la API este mai mare de 30 de secunde). Prelucrarea răspunsului se va mării doar atunci când dimensiunea hărții cerute o să fie mai mare.

În figura următoare pot fi observate drumurile pentru care am obținut rezultatele din **Tabelul 5.3.**. Deoarece distanța calculată de noi nu este cea reală, pot apărea diferențe între drumurile găsite de algoritmi.

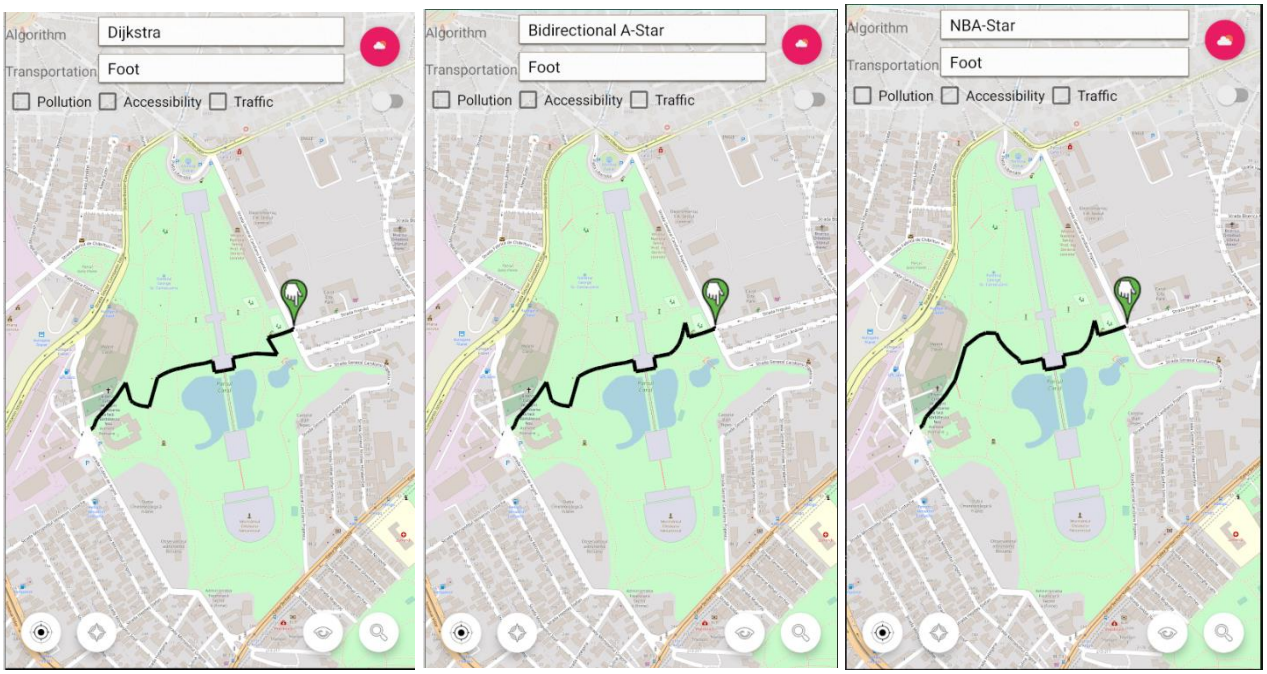


Figura 5.4. Drumul scurt găsit prin algoritmii din aplicație.

Pentru următorul experiment am dori să găsim drumul optim pentru o distanță mai mare, de aproximativ 8-9 km, vom face această căutare într-un oraș deoarece dorim ca graful pe care aplicăm algoritmii să fie cât mai complex.

Algoritm	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Dijkstra	11 470	5.827	12 104	3 064	8.981
A-Star	5 260	2.294	12 104	3 064	25.968
Dijkstra bidirecțional	8 410	3.571	12 104	3 064	11.983
A-Star bidirecțional	2 496	1.033	12 104	3 064	8.843
NBA-Star	2 496	0.986	12 104	3 064	9.546

Tabelul 5.4. Rezultatele algoritmilor pentru metoda de transport „Foot” (pe jos) pentru un drum cu o lungime de aproximativ 8-9 km.

Referitor la timpul de preluare al datelor din API, putem observa o ușoară creștere, față de drumul scurt, acest lucru se întâmplă datorită dimensiunii grafului de care avem nevoie, dar, în continuare, acest timp este variabil. Cât despre timpul de rulare al algoritmilor de găsire al drumului optim, putem observa o diferențiere mai mare; algoritmii *Dijkstra, Dijkstra bidirecțional* și *A-Star* sunt mai înceți, unde algoritmii *A-Star bidirecțional* și *NBA-Star* rămân la o performanță aproximativ egală (numărul de noduri căutate este identic pentru ambele), iar diferența de timp este neglijabila.

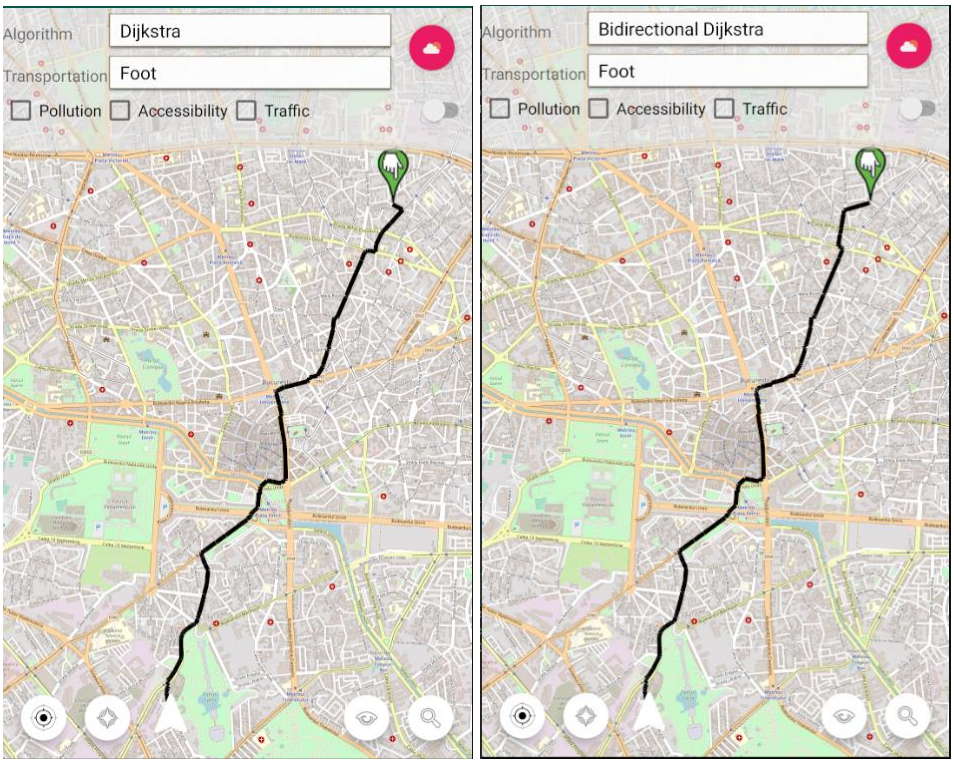


Figura 5.5. Drumul cu lungime de aproximativ 8-9 km găsit cu algoritmii din aplicație.

În continuare se vor testa algoritmii pe distanțe mai lungi, între orașe. Am ales ca punct de start o locație din București, iar ca destinație o locație din Alexandria, deci o distanță de aproximativ 88 km. Pentru astfel de distanțe, nu vom mai putea folosi mersul pe jos ca modalitate de transport, deoarece nu există suport pentru pietoni pentru drumurile din graf.

Algoritm	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Dijkstra	65 397	293.33	91 924	13 467	69.848
A-Star	28 822	80.556	91 920	13 467	49.838
Dijkstra bidirecțional	55 416	180.704	91 920	13 467	45.781
A-Star bidirecțional	17 022	40.814	91 920	13 467	35.231
NBA-Star	16 968	37.387	91 920	13 467	78.933

Tabelul 5.5. Rezultatele algoritmilor pentru metoda de transport „Car” (cu mașina) pentru un drum lung (între orașe).

Se poate observa și în acest caz o creștere a timpului de prelucrare a datelor din API, deci cu cât avem o hartă mai mare cu atât pre procesarea datelor o să dureze mai mult. În toate cazurile testate până acum durata de procesare a datelor din API este mai mare decât timpul de rulare al algoritmilor de găsirea drumului optim.

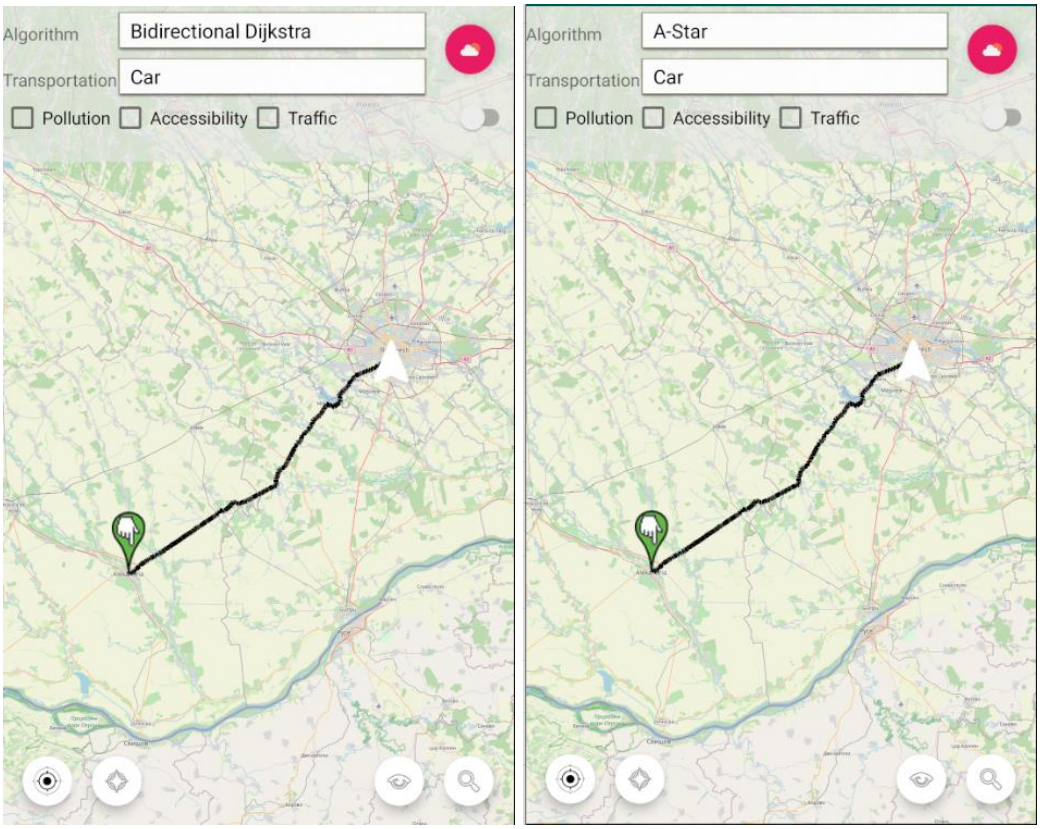


Figura 5.6. Drumul lung găsit cu algoritmii din aplicație.

Putem observa că și la distanțe mari, algoritmii cei mai eficienți sunt *A-Star bidirecțional* și *NBA-Star*. În **Figura 5.6.**, datorită dimensiunii grafului nu se pot observa diferențe în calea optimă găsită; însă dacă vom mări suficient de mult harta, mici diferențe vor putea fi vizualizate.

În continuare vom discuta aspectele de „environment" ale aplicației, cum ar fi nivelul de poluare al aerului, trafic și accesibilitatea drumurilor. Pentru aceste teste vom folosi algoritmul *A-Star*, deoarece lungimea drumului nu este foarte mare, deci orice algoritm este destul de eficient în cazul acesta. Lungimea drumului o să fie redusă, deoarece vom face foarte multe cereri la API-uri în timpul construcției drumurilor optime, lucru ce va crește timpul de rulare al algoritmilor.

Aspect al mediului	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Nimic	329	0.017	1 860	417	1.995
Poluarea aerului	374	253.796	1 860	417	1.819
Accesibilitate	458	0.023	1 860	417	1.685

Tabelul 5.6. Rezultatele algoritmului A-Star pentru drumul ales în cazul testării aspectelor de „*environment*".

Pentru a putea lua în calcul și accesibilitatea, am ales ca drumul să fie parcurs cu bicicleta (ca și metodă de transport), deoarece are cele mai multe nivele de accesibilitate. Drumul a fost ales astfel încât să existe și spațiu verde (păduri/ pracuri), în care, din punct de vedere teoretic ar trebui să avem un nivel de poluare mai scăzut.

Figura 5.7. Drumul găsit fără parametrii de poluare, accesibilitate sau trafic.

Se poate observa o creștere substanțială a timpului de rulare al algoritmului atunci când trebuie să luăm în calcul poluarea. Acest lucru se întâmplă deoarece algoritmul caută în aproximativ 329 noduri, deci va face tot atâtea cereri către API-urile de poluare al aerului. Daca am dori să ținem cont și de poluare și de trafic în același timp, atunci timpul de rulare o să crească considerabil.

Pentru accesibilitate timpul de rulare nu crește foarte mult, deoarece aceste date sunt preluate din API-ul *OpenStreetMap*, deci o dată ce am creat graful orientat, nu vom fi nevoiți să facem alte cereri pentru a le prelua. Timpul crește puțin, deoarece acum trebuie să verificăm mai multe noduri în graf (458, față de 329)

Deoarece traficul este înregistrat doar pe străzi, iar traseul nostru a fost ales astfel încât să existe treceri prin spații verzi, drumul optim nu o să se schimbe atunci când această caracteristică este dorită. Pentru a testa această caracteristică vom alege un alt drum, pentru care vom folosi ca și metodă de transport cu mașina.

Figura 5.8. Drumul găsit ținând cont de poluarea aerului.

În **Figura 5.8.** se poate observa drumul găsit luând în considerare poluarea mediului înconjurător. Traseul rezultat este asemănător cu cel în care nu se ține cont de acest parametru, nu pentru că traseul este traversat printr-un spațiu verde, ci pentru că nivelul de poluare nu contează în acest caz, toate datele de poluare fiind primite de la aceeași sursă (vezi [33]).

În cazul acesta datele sunt preluate de la stația din „Calea Victoriei, Cercul Militar” de către ANPM – Agenției Naționale pentru Protecția Mediului.

Viteza ce poate fi observată în **Figura 5.9**, se referă, de fapt, la viteza traficului în cel mai apropiat punct în care facem cererea și nu la viteza din punctul curent (de exemplu nu vom avea viteza traficului prin parc de 32 km/h, deoarece prin parc nu pot circula mașini).

Figura 5.9. Nivelul de poluare în câteva puncte cheie din traseul ales.

În continuare, pentru același drum vom testa caracteristica de accesibilitate, în care drumul este ales ținând cont de condițiile pe care le are utilizatorul pentru drumul respectiv, de exemplu, pentru mersul cu bicicleta, drumul cel mai accesibil este acela cu pistă specială pentru biciclete, iar cel mai puțin accesibil este drumul de munte, nepavat. În **Figura 5.10.** se poate observa cum, drumul ales nu o să fie prin parc, deoarece vor fi anumite obstacole (pentru o persoană care merge pe bicicletă), cum ar fi pietonii, astfel drumul ce are accesibilitatea cea mai mare este drumul principal deoarece oferă o viteză de deplasare mai mare.

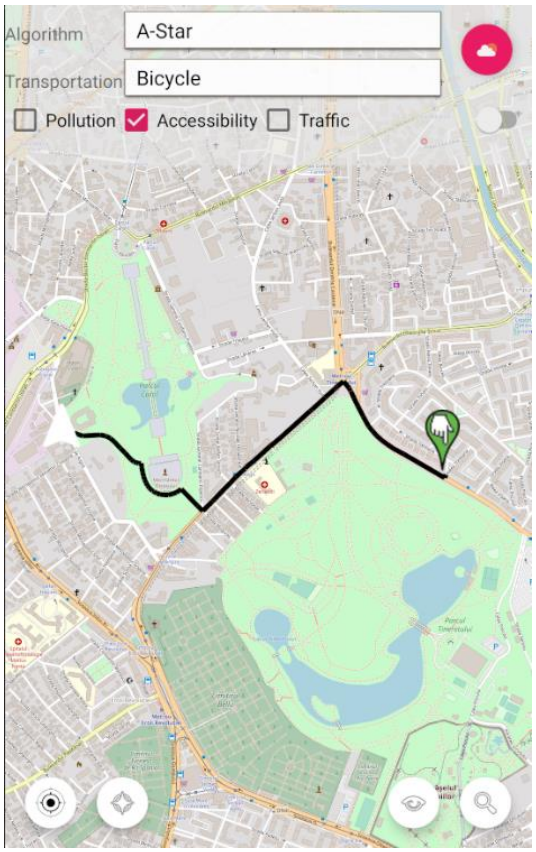


Figura 5.10. Drumul optim găsit ținând cont de accesibilitatea drumului.

Pentru a verifica algoritmul și datele primite de la API-ul de trafic, vom alege un drum foarte aglomerat din București. De asemenea vom folosi algoritmul *NBA-Star*, deoarece acesta este cel mai eficient, căutând cele mai puține noduri, acest lucru este foarte important, deoarece API-ul ales are o

limitare de 2 500 cereri (în cazul planurilor gratuite) zilnice, pentru a putea realiza mai multe cereri către API, ar trebui plătită o licență către producător.

Aspect al mediului	Nr. de noduri căutate	Timp rulare (secunde)	Nr. de noduri din graf	Nr. de laturi din graf	Timp preluare date din API (secunde)
Nimic	48	0.002	1 273	265	1.249
Trafic	64	18.119	1273	265	9.019

Tabelul 5.7. Rezultatele algoritmului *NBA-Star* pentru drumul ales în cazul testării datelor din *TomTom Traffic API*.

În **Figura 5.11.** se poate observa cum, inițial, algoritmul va sugera drumul principal, ca fiind optim, acesta fiind mai scurt, pe de altă parte, iar atunci când introducem datele trimise de trafic acesta va ocoli drumul respectiv.

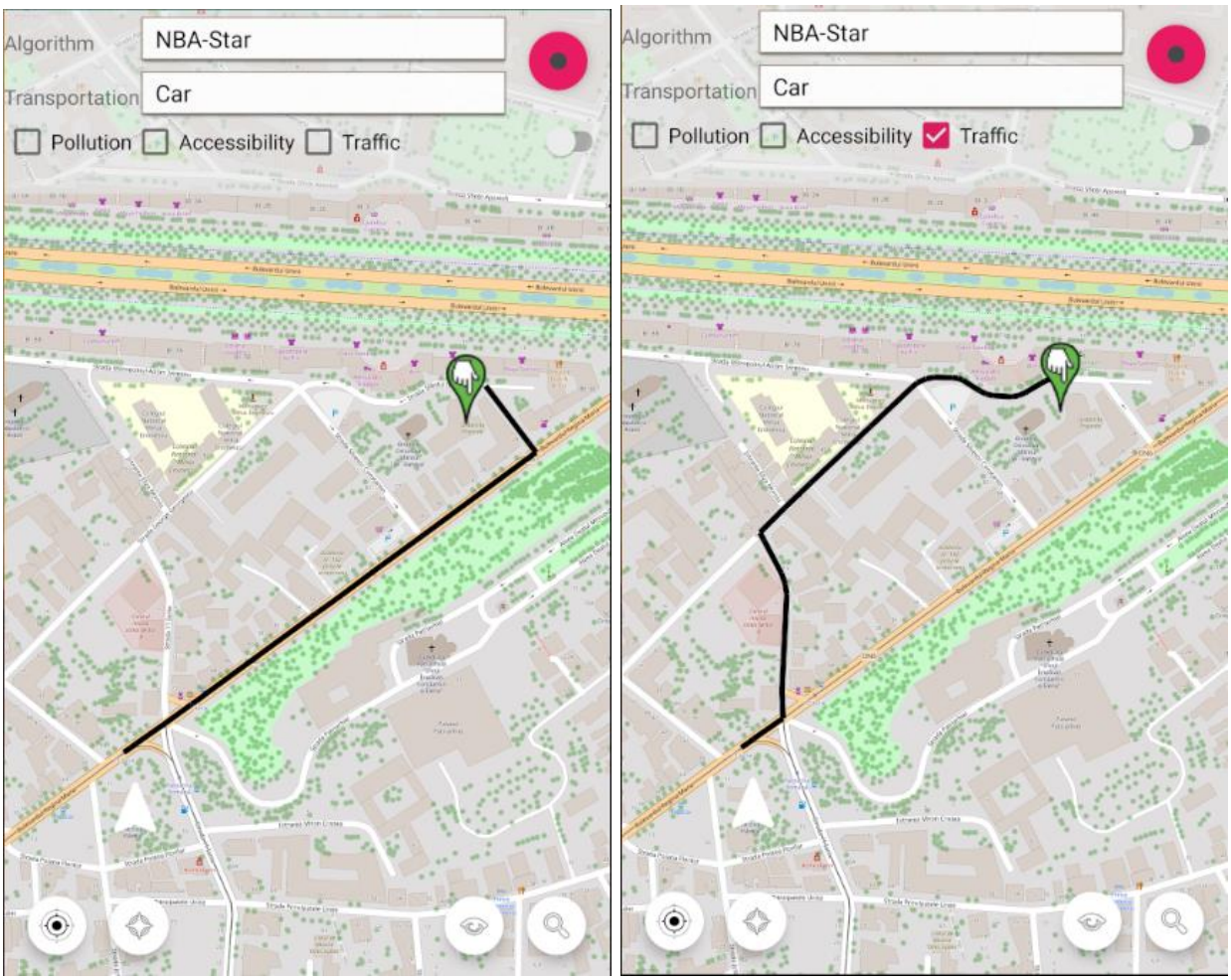


Figura 5.11. Drumul găsit fără datele din trafic (stânga), respectiv drumul găsit ținând cont de datele din trafic (dreapta).

În figurile următoare, pot fi vizualizate vitezele pentru drumurile alese de către algoritm, și se poate observa motivul pentru care a fost aleasă ruta mai lungă ca fiind mai optimă, față de cea mai scurtă..

Figura 5.12. Viteza traficului în câteva puncte cheie ale traseului optim ales (ținând cont de trafic).

Se poate vedea, că viteza din **Figura 5.12.** este cu aproximativ 6-10 km mai mare decât cea din **Figura 5.13.** această diferență de viteze face ca nodurile din prima figură să aibă prioritate în lista nodurilor pe care urmează să le căutăm, față de celelalte noduri.

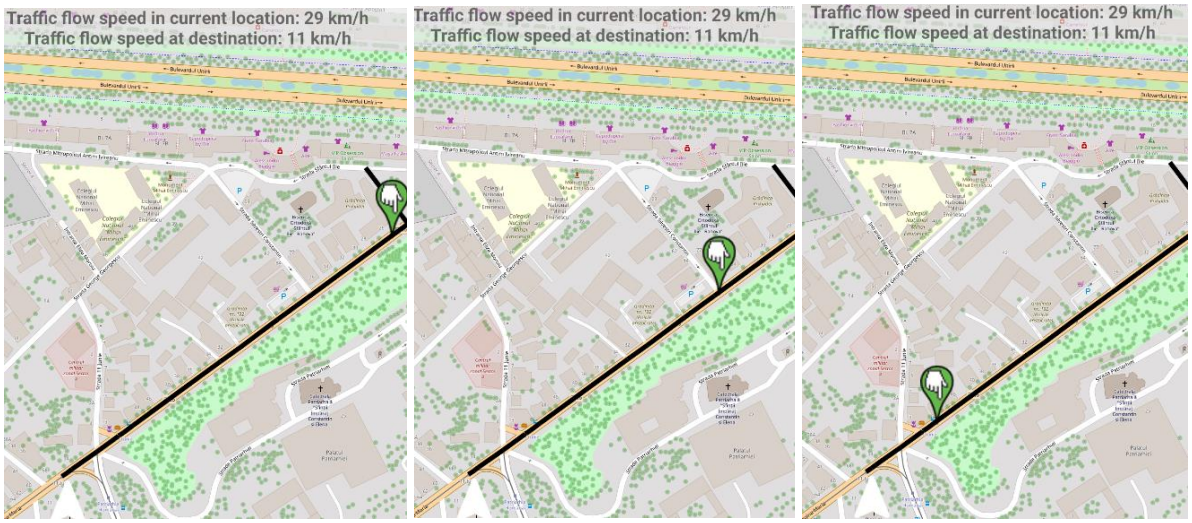


Figura 5.13. Viteza traficului în câteva puncte cheie ale traseului optim ales (când nu se ține cont de trafic).

Concluzii

Concluziile lucrării

Folosind rezultatele expuse în **Capitolul 5 – Rezultate practice** putem trage mai multe concluzii referitoare la tipul aplicației propuse, dar și asupra algoritmilor folosiți în aplicație.

În primul rând, aplicația propusă are ca și caracteristică principală găsirea drumului optim pe hărți reale, astfel graful necesar poate deveni foarte complex, iar anumite noduri putem să nu le folosim întrucât vor îngreuna foarte mult algoritmul, rezultând niște timpi de a rula foarte mari, chiar și pentru un algoritm destul de rapid. Acest lucru poate fi observat făcând o comparație între **Tabelul 5.2.** și **Tabelul 5.4.**, unde pentru un drum de o distanță relativ asemănătoare se obțin rezultate foarte diferite. Un alt aspect ce ar fi putut duce la această diferență de performanță este modul în care este construit graful orientat, deoarece drumul ar avea anumite porțiuni de drum ocolite (vezi **Figura 5.2.**) drumul rezultat nefiind cel mai optim drum.

În al doilea rând, se poate observa că răspunsurile pe care le primim de la API-uri devin din ce în ce mai mari, făcând aplicația mai puțin eficientă. În **Tabelul 5.5.** se poate observa cum pentru algoritmul *NBA-Star*, extragerea datelor din API durează dublu față de timpul de rulare al algoritmului. De asemenea, se poate observa că și cel mai optim algoritm, în cazul de față, *NBA-Star*, la distanțe foarte lungi nu este nici acesta destul de eficient pentru a fi folosit de utilizatori, la o distanță de aproximativ 90 km având deja un timp de rulare de 37 de secunde, iar folosirea și a altor API-uri (cum ar fi *Air Quality Programmatic* sau *TomTom Traffic* API) îl va face și mai ineficient.

O altă caracteristică importantă ce stă la baza aplicației sunt factorii de mediu, pentru a extrage poluarea aerului am folosit API-ul *Air Quality Programmatic*, dar datele acestui API nu vor influența algoritmii de găsirea drumului optim, deoarece nu există o bază de date cu suficient de multe date pentru a putea lua în calcul poluarea aerului, aceasta fiind făcută pe zone, ci nu la fiecare nod din graful existent (cum ar fi ideal), astfel anumite spații verzi, care din punct de vedere teoretic ar trebui să aibă un nivel de poluare mai mic ajung să aibă același nivel de poluare cu intersecțiile străzilor principale din orașe. Pentru a vedea stațiile de la care acest API își procură datele putem consulta sursa [33]. Astfel singurii factori pe care îi putem lua în considerare atunci când calculăm drumul optim sunt accesibilitatea în funcție de drumul pe care ne aflăm (de exemplu, autostradă, drum principal, drum care nu este pavat etc.) și aglomerația din trafic (dată extrasă cu ajutorul unui API).

Contribuții personale

Contribuțiile personale în realizarea aplicației reprezintă: extragerea datelor brute din API-uri precum *OpenStreetMap, Overpass, OpenWeatherMap, Air Quality Programmatic, TomTom Traffic API* și prelucrarea acestora pentru crearea grafului orientat asupra căruia se aplică algoritmii pentru găsirea drumului optim, datele necesare pentru evaluarea greutății laturilor, datele de care avem nevoie pentru a lua o decizie asupra modului de transport utilizat (în funcție de vremea ce se află la poziția curentă a utilizatorului). Pe lângă prelucrarea datelor primite de la API-uri, am realizat și: implementarea specifică a algoritmilor *Dijkstra, A-Star, Dijkstra Bidirecțional, A-Star Bidirecțional* și *NBA-Star*, metoda ce ne ajută în evaluarea a greutății unei laturii dintre două noduri, interfața grafică a aplicației; de asemenea, pentru a nu bloca *Threadul* principal și pentru a face experiența utilizatorului să fie cât mai fluidă, am realizat o reducere paralelă asupra anumitor sarcini, rulând-u-

le pe un *Thread* secundar (printre aceste sarcini se numără: încărcarea datelor în pagina de start a aplicației, rularea algoritmilor, adăugarea marcatorilor etc.).

Pe lângă realizarea grafului prin prelucrarea directă a datelor brute, primite ca răspuns de la *OSM*, am mai implementat încă o modalitate de crea graful, mai exact cu ajutorul unor fișiere „csv" stocate local pe dispozitiv (fișierele au fost create cu ajutorul scriptului *osm4routing*), aceste date sunt încărcate la intrarea utilizatorului în aplicație. Datorită ineficienței, în timpul testelor nu am folosit această metodă pentru a crea graful.

Direcții viitoare de cercetare

Printre direcțiile viitoare de cercetare se numără: paralelizarea algoritmilor (acest lucru presupune paralelizarea buclelor din algoritmi, acest tip de paralelizare necesită mai multă atenție decât cea folosită momentan, deoarece trebuie să știm atunci când trebuie să folosim bariere de sincronizare, mutexuri etc), folosirea datelor stocate pe un server personal (pentru a crește rapiditatea extragerii datelor), folosirea unui API de trafic ce ne oferă mai multe date pentru o anumită zonă a hărții (pentru a micșora numărul de cereri *HTTPS* către API) sau folosirea numărului de persoane active din aplicație pentru a detecta aglomerația din trafic, adăugarea posibilității de a stoca local anumite date pentru a fi folosite și în offline. Toate aceste elemente vor face aplicația mai practică, ușurând timpul de rulare pentru anumite sarcini.

Bibliografie

[1] *State-of-the-Art Trends in Mobile App Development*, https://www.codementor.io/@marshasely/state-of-the-art-trends-in-mobile-app-development-6k2mcmhfo, accesat la data: 31.05.2020

[2] Ian F. Darwin, *Android Cookbook*, Editura O'Reilly Media, Inc., Mai, 2017

[3] *Google's Android OS: Past, Present, and Future*, https://www.phonearena.com/news/Googles-Android-OS-Past-Present-and-Future_id21273, accesat la data: 31.05.2020

[4] *Android founder: We aimed to make a camera OS*, https://www.pcworld.com/article/2034723/android-founder-we-aimed-to-make-a-camera-os.html, acccesat la data: 31.05.2020

[5] *A Murky Road Ahead for Android, Despite Market Dominance*, https://www.nytimes.com/2015/05/28/technology/personaltech/a-murky-road-ahead-for-android-despite-market-dominance.html, accesat la data: 31.05.2020

[6] *Take a trip down memory lane with the quirky T-Mobile G1*, https://www.pcworld.com/article/3308157/first-android-phone-t-mobile-g1-10th-anniversary.html, accesat la data: 31.05.2020

[7] *Android versions and their names: Here's how Google named all the Android versions*, https://www.timesnownews.com/technology-science/article/android-versions-and-their-names-here-s-how-google-has-named-all-the-versions-of-its-android-versions/289386, accesat la data: 31.05.2020

[8] *From Nexus One to Nexus 10: a brief history of Google's flagship devices*, https://arstechnica.com/gadgets/2013/05/from-the-nexus-one-to-the-nexus-10-a-brief-history-of-nexus-devices/, accesat la data: 09.05.2020

[9] *The Google Phone*, https://www.theverge.com/a/google-pixel-phone-new-hardware-interview-2016, accesat la data: 31.05.2020

[10] *Google Pixel 2 – Noul Design in Imagini Oficiale*, https://www.idevice.ro/2017/10/03/google-pixel-2-design-imagini-oficiale/, accesat la data: 31.05.2020

[11] *A growing list of everyday things replaced by your smartphone*, https://www.gotechtor.com/things-replaced-by-smartphones/, accesat la data: 31.05.2020

[12] Ștefan Tansă, Cristian Olaru, Ștefan Andrei, *Java de la 0 la expert*, Editura Polirom, 2003

[13] Top apps built with Java, https://www.androidguys.com/community/top-apps-built-java/, accesat la data: 31.05.2020

[14] The Java Programming Language Platforms, https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html#gcrkk, accesat la data: 10.05.2020

[15] Differences between JDK, JRE and JVM, https://www.geeksforgeeks.org/differences-jdk-jre-jvm/, accesat la data: 31.05.2020

[16] What Will Be The Best Java IDE's in 2020?, https://www.geeksforgeeks.org/what-will-be-the-best-java-ides-in-2020/, accesat la data: 31.05.2020

[17] Android Studio intro, https://developer.android.com/studio/intro, accesat la data: 01.06.2020

[18] *Kotlin Standard Library*, https://kotlinlang.org/api/latest/jvm/stdlib/, accesat la data: 01.06.2020

[19] *Android Studio User guide*, https://developer.android.com/studio/debug/layout-inspector, accesat la data: 01.06.2020

[20] *Emulator*, https://www.computerhope.com/jargon/e/emulator.htm, accesat la data: 01.06.2020

[21] *Android Emulator – AMD Processor & Hyper-V Support*, https://android-developers.googleblog.com/2018/07/android-emulator-amd-processor-hyper-v.html, accesat la data: 01.06.2020

[22] *Genymotion*, https://www.genymotion.com/desktop/, accesat la data: 01.06.2020

[23] *Application Programming Interface (API): Definition & Example*, https://study.com/academy/lesson/application-programming-interface-api-definition-example.html, accesat la data: 01.06.2020

[24] *OpenStreetMap Main Page*, https://wiki.openstreetmap.org/wiki/Main_Page, accesat la data: 01.06.2020

[25] *OpenStreetMap Android*, https://wiki.openstreetmap.org/wiki/Android, accesat la data: 01.06.2020

[26] *Welcome to Vodafone Wayfinder OSS*, http://wayfinder.org/2010/07/13/welcome_to_vodafone_wayfinder_oss/, accesat la data: 01.06.2020

[27] *Wayfinder FAQ*, http://wayfinder.org/2010/07/20/faq/, accesat la data: 01.06.2020

[28] *Wayfinder Open Source Profect*, http://wayfinder.org/, accesat la data: 01.06.2020

[29] *OpenWeatherMap*, https://openweathermap.org/, accesat la data: 01.06.2020

[30] *OpenWeather*, https://openweather.co.uk/, accesat la data: 01.06.2020

[31] *Web of Things*, https://iot.mozilla.org/about/, accesat la data: 01.06.2020

[32] *Air Quality Programmatic*, https://aqicn.org/api/, accesat la data: 01.06.2020

[33] *Air Pollution in World: Real-Time Air Quality Index Visual Map*, https://aqicn.org/map/world/, accesat la data: 01.06.2020

[34] *TomTom Traffic API*, https://developer.tomtom.com/traffic-api, accesat la data: 01.06.2020

[35] *Osmdroid*, https://github.com/osmdroid/osmdroid, accesat la data: 01.06.2020

[36] *OsmBonusPack*, https://github.com/MKergall/osmbonuspack, accesat la data: 01.06.2020

[37] *MapsForge*, https://github.com/mapsforge/mapsforge, accesat la data: 01.06.2020

[38] *JSONObject*, https://developer.android.com/reference/org/json/JSONObject, accesat la data: 01.06.2020

[39] Mayank Patel, *Data structure and algorithm with C*, Editura Educreation, 2018

[40] Ramesh Bhandari, *SURVIVABLE NETWORKS Algorithms for Diverse Routing*, Editura Kluwer Academic, 1999

[41] Edsger Waybe Dijkstra, *A Note on Two Problems in Connexion with Graphs*, în Numerische Mathematik 1/1959, pp. 269-271

[42] Kurt Mehlhora, Peter Sanders, *Algorithms and Data Structures: The Basic Toolbox*, Editura Springer, 2007

[43] Harika Reddy, *PATH FINDING – Dijkstra's and A* Algorithm's, 2013*

[44] Daniel Sanchez-Crespo Dalmau, *Core Techniques and Algorithms in Game Programming*, Editura New Riders, 2004

[45] Wim Pijls, Hank Post, *A new bidirectional algorithm for shortest paths*, Electronic Institute Report EI 2008-25, 2008

[46] *How to use the osmdroid library*, https://github.com/osmdroid/osmdroid/wiki/How-to-use-the-osmdroid-library, accesat la data: 12.06.2020

[47] *What is Gradle?*, https://docs.gradle.org/current/userguide/what_is_gradle.html, accesat la data: 12.06.2020

[48] *Making Custom Overlays*, https://github.com/osmdroid/osmdroid/wiki/Making-Custom-Overlays, accesat la data: 12.06.2020

[49] *Get the last known location*, https://developer.android.com/training/location/retrieve-current, accesat la data: 12.06.2020

[50] *Air Quality Programmatic Documentation*, https://aqicn.org/json-api/doc/, accesat la data: 12.06.2020

[51] TomTom Traffic API Documentation, https://developer.tomtom.com/traffic-api/traffic-api-documentation-traffic-flow/flow-segment-data, accesat la data: 12.06.2020

[52] Osm4routing, https://github.com/Tristramg/osm4routing, accesat la data: 12.06.2020

Anexa 1 – codul sursă

StaticHelper.java

```
package com.example.e_road;

import java.time.Instant;

public class StaticHelper {
    //This method contains elements that we will use all around the project.
    //For example, with the SelectedAlgorithm got from the MainActivity we will be able
    //to chose the method to call in the Search class.

    static int ChoseImplementationType = 1; //0 - default (using OpenStreetMap API),
                                            //1 - using csv files (time consuming, not good)

    public static long NumOfNodesSearched;

    public static String SelectedAlgorithm;
    static String SelectedTransportation;

    public static boolean isCarRoad;
    public static boolean isBicycleRoad;
    public static boolean isFootRoad;

    public static boolean IsPollutionChecked;
    public static boolean IsAccessibilityChecked;
    public static boolean IsTrafficChecked;

    public static Instant AlgInitialTime;
    public static Instant AlgFinalTime;
    public static Instant OsmApiInitialTime;
    public static Instant OsmApiFinalTime;

    static void SetupTransportationMethods() {
        isCarRoad = StaticHelper.SelectedTransportation.equals("Car");
        isBicycleRoad = StaticHelper.SelectedTransportation.equals("Bicycle");
        isFootRoad = StaticHelper.SelectedTransportation.equals("Foot");
    }
}
```

Splash.java

```
package com.example.e_road;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.widget.TextView;

import com.example.e_road.OsmXmlParser.SearchAlgHelper;

public class Splash extends Activity {

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.splash_screen);
        TextView mTextView = findViewById(R.id.SplashText);
        getApplicationInfo().name = "e-Road";
        mTextView.setText(getApplicationInfo().name);

            /* New Handler to start the Menu-Activity
             * and close this Splash-Screen after some seconds.*/
            new Handler().postDelayed(new Runnable() {
                @Override
                public void run() {
                    if (StaticHelper.ChoseImplementationType == 1) {
                        //Use local CSV files to load data (bad implementation)
                        SearchAlgHelper.SetupNodesAndEdges(getApplicationContext());
                    }
                    /* Create an Intent that will start the Menu-Activity. */
                    Intent mainIntent = new Intent(Splash.this, MainActivity.class);
                    Splash.this.startActivity(mainIntent);
                    Splash.this.finish();
```

```
                }
            }, 500);
    }

}
```

Weather.java

```
package com.example.e_road.Weather;

import android.support.annotation.NonNull;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

import java.io.IOException;

import static com.example.e_road.Weather.WeatherParser.ParseWeatherByLocation;

public class Weather {
    private String WeatherType = "no"; //This can be Clear, Rain, Snow, Extreme; weather.main

    private String Temperature = "no";
    private String Pressure = "no";
    private String Humidity = "no";
    private String RainVolume = "no rain";
    private String SnowVolume = "no snow";
    private String WindSpeed = "no wind";

    private String WeatherIcon = "";

    private String AppDecision = "";

    public void SetWeatherParams(Double lat, Double lng) throws IOException, JSONException {

        JSONObject WeatherObject = ParseWeatherByLocation(lat,lng);

        this.WeatherType = ((JSONObject)((JSONArray)
WeatherObject.get("weather")).get(0)).get("main").toString();

        JSONObject WeatherMainFeatures = ((JSONObject) WeatherObject.get("main"));

        this.Temperature = WeatherMainFeatures.get("temp").toString() + " K";
        this.Pressure    = WeatherMainFeatures.get("pressure").toString() + " hPa";
        this.Humidity    = WeatherMainFeatures.get("humidity").toString() + " %";

        if (WeatherObject.has("rain") &&
                ((JSONObject) WeatherObject.get("rain")).has("3h")) {
            this.RainVolume = ((JSONObject) WeatherObject.get("rain")).get("3h").toString() + " mm";
        }

        if (WeatherObject.has("snow") &&
                ((JSONObject) WeatherObject.get("snow")).has("3h")) {
            this.SnowVolume = ((JSONObject) WeatherObject.get("snow")).get("3h").toString() + " mm";
        }

        if(WeatherObject.has("wind") &&
                ((JSONObject) WeatherObject.get("wind")).has("speed")) {
            this.WindSpeed = ((JSONObject) WeatherObject.get("wind")).get("speed").toString() + "
m/s";
        }

        this.WeatherIcon = ((JSONObject)((JSONArray) WeatherObject.get("weather"))
                                                    .get(0)).get("icon").toString();
        this.WeatherIcon = "http://openweathermap.org/img/wn/" + this.WeatherIcon + "@2x.png";

        this.AppDecision = this.WeatherType.equals("Rain") || this.WeatherType.equals("Snow") ?
                               "Please use an umbrella or a car" :
                               this.WeatherType.equals("Extreme") ?
                                    "Please don't walk or use bike (Extreme conditions outside)" :
                                    "The weather looks nice :)";
    }

    public String GetWeatherIcon(){
        System.out.println("AppDecision = " + this.WeatherIcon);
        return this.WeatherIcon;
    }
```

```
    @NonNull
    public String toString() {
        return "\nWeather status outside: " + WeatherType.toLowerCase() +
                "\n\n\nTemperature: " + this.Temperature +
                "\n\nPressure: " + this.Pressure +
                "\n\nHumidity: " + this.Humidity +
                "\n\nRain in the past 3h: " + RainVolume +
                "\n\nSnow in the past 3h: " + this.SnowVolume +
                "\n\nWind: " + this.WindSpeed + "\n\n\n" +
                this.AppDecision + "\n";
    }
}
```

WeatherParser.java

```
package com.example.e_road.Weather;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.IOException;
import java.util.Objects;

import okhttp3.OkHttpClient;
import okhttp3.Request;

class WeatherParser {

    static JSONObject ParseWeatherByLocation(Double lat, Double lng)
                                                throws IOException, JSONException {

        OkHttpClient client = new OkHttpClient();
        String url;

        JSONObject WeatherObject = null;
        url = "http://api.openweathermap.org/data/2.5/weather?lat=" +
                                                lat + "&lon=" + lng +
                                                "&appid=54e9df1b1a69ba901412d5f895f5e06f";
        Request request = new Request.Builder()
                .url(url)
                .build();

        String myResponse=Objects.requireNonNull(client.newCall(request).execute().body()).string();

        JSONObject jsonObject = new JSONObject(myResponse);

        if (!myResponse.isEmpty()){
//            System.out.println("DEBUGGING 201 - WeatherParser.java : "+jsonObject.get("weather"));
            WeatherObject = jsonObject;

        }
        return WeatherObject;
    }
}
```

TrafficParser.java

```
package com.example.e_road.Traffic;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.IOException;
import java.util.Objects;

import okhttp3.OkHttpClient;
import okhttp3.Request;

public class TrafficParser {

    public static String GetTrafficSpeed(Double lat, Double lng)
                                            throws IOException, JSONException {

        JSONObject TrafficObject = GetJSONResponse(lat, lng);

        return TrafficObject.get("currentSpeed").toString();
    }
    public static Double GetDataForWeightOfWay(Double lat, Double lng)
                                                throws IOException, JSONException {
```

```
        JSONObject TrafficObject = GetJSONResponse(lat, lng);

        return Double.parseDouble(TrafficObject.get("freeFlowSpeed").toString()) -
                Double.parseDouble(TrafficObject.get("currentSpeed").toString());
    }

    private static JSONObject GetJSONResponse(Double lat, Double lng)
                                                throws JSONException, IOException {

        OkHttpClient client = new OkHttpClient();
        String url;

        JSONObject TrafficObject = null;
        url = "https://api.tomtom.com/traffic/services/4/flowSegmentData/absolute/10/" + "json" +
                                                "?key=uOZsxG6Gy8Mbtz36ex5tmDFmIpNYP5GL&point=" +
                                                lat + "," + lng + "&unit=kmph";
        Request request = new Request.Builder()
                .url(url)
                .build();

        String myResponse=Objects.requireNonNull(client.newCall(request).execute().body()).string();

        JSONObject jsonObject = new JSONObject(myResponse);

        if (!myResponse.isEmpty()){
            TrafficObject = (JSONObject) jsonObject.get("flowSegmentData");

        }
        return TrafficObject;
    }
}
```

AirQualityParser.java

```
package com.example.e_road.AirQuality;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.IOException;
import java.util.Objects;

import okhttp3.OkHttpClient;
import okhttp3.Request;

public class AirQualityParser {

    public static String GetAQILevel(Double lat, Double lng) throws IOException, JSONException {

        OkHttpClient client = new OkHttpClient();
        String url;

        JSONObject AirQualityObject = null;
        url = "http://api.waqi.info/feed/geo:" + lat + ";" + lng +
                    "/?token=7d55351d1569de2f5a311c3975519b070d9782e1";
        Request request = new Request.Builder()
                .url(url)
                .build();

        String myResponse=Objects.requireNonNull(client.newCall(request).execute().body()).string();

        JSONObject jsonObject = new JSONObject(myResponse);

        if (!myResponse.isEmpty()){
//            System.out.println("DEBUGGING 101 - AirQualityParser.java : " +
jsonObject.get("data"));
            AirQualityObject = (JSONObject) jsonObject.get("data");

        }
        assert AirQualityObject != null;
        return AirQualityObject.get("aqi").toString();
    }
}
```

SearchAlgHelper.java

```
package com.example.e_road.OsmXmlParser;
```

```
import android.annotation.SuppressLint;
import android.content.Context;

import com.example.e_road.StaticHelper;
import com.example.e_road.SearchAlgorithms.Graph;
import com.example.e_road.SearchAlgorithms.Node;
import com.example.e_road.SearchAlgorithms.Way;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.time.Instant;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Objects;

public class SearchAlgHelper {
    // first implementation - slow. DO NOT USE
    @Deprecated
    @SuppressLint("UseSparseArrays")
    public static void SetupNodesAndEdges(Context mContext) {
        StaticHelper.OsmApiInitialTime = Instant.now();

        HashMap<Long,Node> UsedNodes= new HashMap<>();
        HashMap<Long, Way> UsedWays = new HashMap<>();

        BufferedReader NodesCsvFile = null;
        BufferedReader EdgesCsvFile = null;
        String line;
        String cvsSplitBy = ",";
        try {
            NodesCsvFile = new BufferedReader(new InputStreamReader(
                                mContext.getAssets().open("nodes.csv"),
                                StandardCharsets.UTF_8));
            EdgesCsvFile =  new BufferedReader(new InputStreamReader(
                                mContext.getAssets().open("edges.csv"),
                                StandardCharsets.UTF_8));

            while ((line = NodesCsvFile.readLine()) != null) {
                String[] node = line.split(cvsSplitBy);
                UsedNodes.put(Long.parseLong(node[0]), new Node(Long.parseLong(node[0]),
                                                        Double.parseDouble(node[2]),
                                                        Double.parseDouble(node[1])));
            }

            while ((line = EdgesCsvFile.readLine()) != null) {
                // use comma as separator
                String[] way = line.split(cvsSplitBy);

                List<Long> NodesInWay = new ArrayList<>();

                double CarAccessibility     =
                        Double.parseDouble(way[5]) > Double.parseDouble(way[6]) ?
                                Double.parseDouble(way[5]) : Double.parseDouble(way[6]);
                double BicycleAccessibility =
                        Double.parseDouble(way[8]) > Double.parseDouble(way[7]) ?
                                Double.parseDouble(way[7]) : Double.parseDouble(way[8]);
                double FootAccessibility    = Double.parseDouble(way[4]);
                if (UsedWays.containsKey(Long.parseLong(way[0]))) {
                    Objects.requireNonNull(UsedWays.get(Long.parseLong(way[0])))
                                                        .getNodesInWay()
                                                        .add(Long.parseLong(way[1]));
                    Objects.requireNonNull(UsedWays.get(Long.parseLong(way[0])))
                                                        .getNodesInWay()
                                                        .add(Long.parseLong(way[2]));
                } else {
                    NodesInWay.add(Long.parseLong(way[1]));
                    NodesInWay.add(Long.parseLong(way[2]));
                    UsedWays.put(Long.parseLong(way[0]), new Way(Long.parseLong(way[0]),
                                                            NodesInWay, CarAccessibility,
                                                            BicycleAccessibility,
                                                            FootAccessibility));
                }
            }

            Graph.setGraph(UsedNodes, UsedWays);
            System.out.println("\nNum of nodes = " + Graph.getNodes().size());
            System.out.println("\nNum of edges = " + Graph.getWays().size());
```

```
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (NodesCsvFile != null && EdgesCsvFile != null) {
                try {
                    NodesCsvFile.close();
                    EdgesCsvFile.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        StaticHelper.OsmApiFinalTime = Instant.now();
    }
}
```

MapToGraphParser.java

```
package com.example.e_road.OsmXmlParser;

import com.example.e_road.StaticHelper;
import com.example.e_road.SearchAlgorithms.*;

import org.xmlpull.v1.XmlPullParserException;

import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.time.Instant;

public class MapToGraphParser {
    // Using static attributes because we want to use them all over the application.
    // By using static they will always point to the same location.

    public static void setGraphFromAPI(Node Start, Node Finish)
                                    throws IOException, XmlPullParserException {
        StaticHelper.OsmApiInitialTime = Instant.now();
        String urlString;
        urlString = getUrlString(Start, Finish);

        URL url = new URL(urlString);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(30000 /* milliseconds */);
        conn.setConnectTimeout(30000 /* milliseconds */);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        // Starts the query
        conn.connect();

        InputStream Stream = conn.getInputStream();
        OsmXmlParser OsmXmlParser = new OsmXmlParser();
        OsmXmlParser.parse(Stream);
        System.out.println("Number of ways in use: " + Graph.getWays().size() +
                           "\nNumber of Nodes in use: " + Graph.getNodes().size());
        StaticHelper.OsmApiFinalTime = Instant.now();
    }

    private static String getUrlString(Node Start, Node Finish){
        String SouthBorder, WestBorder, NorthBorder, EastBorder;
        if (Start.getLongitude() > Finish.getLongitude()) {   // calculate Bounding Box parameters
            SouthBorder = (Double.toString(Math.floor((Finish.getLongitude()) * 100)/100));
            NorthBorder = (Double.toString(Math.ceil((Start.getLongitude()) * 100)/100));
        } else {
            SouthBorder = (Double.toString(Math.floor((Start.getLongitude()) * 100)/100));
            NorthBorder = (Double.toString(Math.ceil((Finish.getLongitude()) * 100)/100));
        }
        if (Start.getLatitude() > Finish.getLatitude()){
            WestBorder = (Double.toString(Math.floor((Finish.getLatitude()) * 100)/100));
            EastBorder = (Double.toString(Math.ceil((Start.getLatitude()) * 100)/100));
        } else {
            WestBorder = (Double.toString(Math.floor((Start.getLatitude()) * 100)/100));
            EastBorder = (Double.toString(Math.ceil((Finish.getLatitude()) * 100)/100));
        }
        System.out.println("BORDERS : " +
                "\nS: "+SouthBorder+"\nW: "+WestBorder +"\nN: "+NorthBorder+"\nE: "+EastBorder);
        //OpenStreetMap Api Request
//        urlString = "https://api.openstreetmap.org/api/0.6/map?bbox= +
//                        SouthBorder + "," + WestBorder + "," + NorthBorder + "," + EastBorder";
        //Overpass Api Request
```

```
//        urlString = "http://www.overpass-api.de/api/xapi?map?bbox=" +
//                        SouthBorder + "," + WestBorder + "," + NorthBorder + "," + EastBorder;

        return "http://www.overpass-api.de/api/xapi?map?bbox=" +
                SouthBorder + "," + WestBorder + "," + NorthBorder + "," + EastBorder;
    }
}
```

OsmXmlParser.java

```
package com.example.e_road.OsmXmlParser;

import android.annotation.SuppressLint;
import android.util.Xml;

import com.example.e_road.StaticHelper;
import com.example.e_road.SearchAlgorithms.Graph;
import com.example.e_road.SearchAlgorithms.Node;
import com.example.e_road.SearchAlgorithms.Way;

import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;

import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

class OsmXmlParser {

    private static final String ns = null;

    private static boolean isGoodWay;

    void parse(InputStream in) throws XmlPullParserException, IOException {

        try {
            XmlPullParser parser = Xml.newPullParser();
            parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
            parser.setInput(in, null);
            parser.nextTag();
            readOsm(parser);
        } finally {
            in.close();
        }
    }

    @SuppressLint("UseSparseArrays")
    private void readOsm(XmlPullParser parser) throws XmlPullParserException, IOException {
        HashMap<Long,Node> AllNodes         = new HashMap<>();
        HashMap<Long,Node> UsedFootNodes    = new HashMap<>();
        HashMap<Long,Node> UsedCarNodes     = new HashMap<>();
        HashMap<Long,Node> UsedBicycleNodes = new HashMap<>();
        HashMap<Long,Way>  UsedWays         = new HashMap<>();

        parser.require(XmlPullParser.START_TAG, ns, "osm");
        while (parser.next() != XmlPullParser.END_TAG) {
            if (parser.getEventType() != XmlPullParser.START_TAG) {
                continue;
            }
            String name = parser.getName();
            // Starts by looking for the entry tag
            if (name.equals("node")) {
                Node mNode = new Node(Long.parseLong(parser.getAttributeValue(null, "id")),
                                      Double.parseDouble(parser.getAttributeValue(null, "lat")),
                                      Double.parseDouble(parser.getAttributeValue(null, "lon")));
                SkipNode(parser);
                AllNodes.put(mNode.getId(), mNode);             //Get all nodes (both useful and
useless).
            } else if (name.equals("way")) {
                Way mWay = readWay(parser, AllNodes, UsedFootNodes, UsedCarNodes, UsedBicycleNodes);
                if (isGoodWay)
                    UsedWays.put(mWay.getId(), mWay);       //Create the graph, we will need all the
nodes to construct the ways
            } else {
                skip(parser);
            }
        }
        System.out.println("Number of nodes in the hole xml: " + AllNodes.size());
```

```
        HashMap<Long,Node> UsedNodes = StaticHelper.isFootRoad ? UsedFootNodes :
(StaticHelper.isCarRoad ? UsedCarNodes : UsedBicycleNodes);
        Graph.setGraph(UsedNodes, UsedWays);
    }

    private void SkipNode(XmlPullParser parser) throws XmlPullParserException, IOException {
        parser.require(XmlPullParser.START_TAG, ns, "node");
        while (parser.next() != XmlPullParser.END_TAG) {
            if (parser.getEventType() != XmlPullParser.START_TAG) {
                continue;
            }
            skip(parser);
        }
    }

    private Way readWay(XmlPullParser parser, HashMap<Long,Node> AllNodes, HashMap<Long,Node>
UsedFootNodes, HashMap<Long,Node> UsedCarNodes, HashMap<Long,Node> UsedBicycleNodes) throws
XmlPullParserException, IOException {

        long Id = Long.parseLong(parser.getAttributeValue(null, "id"));

        parser.require(XmlPullParser.START_TAG, ns, "way");

        List<Long> Nodes         = new ArrayList<>();
        isGoodWay                = false;
        double CarAccessibility     = 0;
        double BicycleAccessibility = 0;
        double FootAccessibility    = 0;

        while (parser.next() != XmlPullParser.END_TAG) {
            if (parser.getEventType() != XmlPullParser.START_TAG) {
                continue;
            }
            String name = parser.getName();
            if(name.equals("tag") && (parser.getAttributeValue(null, "k").equals("highway"))){
                if (StaticHelper.isFootRoad){
                    switch (parser.getAttributeValue(null, "v")) {
                        case "pedestrian":
                        case "corridor":
                            FootAccessibility = 1;
                            break;
                        case "footway":
                        case "steps":
                        case "service":
                        case "living_street":
                        case "primary_link":
                        case "secondary_link":
                        case "tertiary_link":
                        case "residential_link":
                        case "primary":
                        case "secondary":
                        case "tertiary":
                        case "residential":
                            FootAccessibility = 2;
                            break;
                        case "track":
                            FootAccessibility = 3;
                            break;
                    }
                }

                if (StaticHelper.isCarRoad){
                    switch (parser.getAttributeValue(null, "v")) {
                        case "motorway":
                        case "motorway_link":
                            CarAccessibility = 1;
                            break;
                        case "trunk":
                        case "trunk_link":
                            CarAccessibility = 2;
                            break;
                        case "primary":
                        case "primary_link":
                            CarAccessibility = 3;
                            break;
                        case "secondary":
                        case "secondary_link":
                            CarAccessibility = 4;
                            break;
                        case "tertiary":
```

```
                    case "tertiary_link":
                        CarAccessibility = 5;
                        break;
                    case "residential":
                    case "residential_link":
                    case "living_street":
                    case "unclassified":
                        CarAccessibility = 6;
                        break;
                }
            }

            if (StaticHelper.isBicycleRoad){
                if (parser.getAttributeValue(null, "v").equals("cycleway")){
                    BicycleAccessibility = 1;
                } else if (parser.getAttributeValue(null, "v").equals("trunk") ||
                        parser.getAttributeValue(null, "v").equals("trunk_link")){
                    BicycleAccessibility = 2;
                } else if (parser.getAttributeValue(null, "v").equals("primary") ||
                        parser.getAttributeValue(null, "v").equals("primary_link")){
                    BicycleAccessibility = 3;
                } else if (parser.getAttributeValue(null, "v").equals("secondary") ||
                        parser.getAttributeValue(null, "v").equals("secondary_link")){
                    BicycleAccessibility = 4;
                } else if (parser.getAttributeValue(null, "v").equals("tertiary") ||
                        parser.getAttributeValue(null, "v").equals("tertiary_link")){
                    BicycleAccessibility = 5;
                } else if (parser.getAttributeValue(null, "v").equals("residential") ||
                        parser.getAttributeValue(null, "v").equals("residential_link") ||
                        parser.getAttributeValue(null, "v").equals("living_street") ||
                        parser.getAttributeValue(null, "v").equals("unclassified")){
                    BicycleAccessibility = 6;
                } else if (parser.getAttributeValue(null, "v").equals("footway") ||
                        parser.getAttributeValue(null, "k").equals("bicycle")){
                    BicycleAccessibility = 7;
                } else if (parser.getAttributeValue(null, "v").equals("track")){
                    BicycleAccessibility = 8;
                }
            }
            isGoodWay = true;
        }
        if(name.equals("nd")) {
            Nodes.add(Long.parseLong(parser.getAttributeValue(null, "ref")));
            SkipNd(parser);
        } else {
            skip(parser);
        }
    }
    if (isGoodWay && StaticHelper.isFootRoad){
        for (Long NodeId : Nodes){
            UsedFootNodes.put(NodeId, AllNodes.get(NodeId));
        }
    }
    if (isGoodWay && StaticHelper.isCarRoad){
        for (Long NodeId : Nodes){
            UsedCarNodes.put(NodeId, AllNodes.get(NodeId));
        }
    }
    if (isGoodWay && StaticHelper.isBicycleRoad){
        for (Long NodeId : Nodes){
            UsedBicycleNodes.put(NodeId, AllNodes.get(NodeId));
        }
    }

    return new Way(Id, Nodes, CarAccessibility, BicycleAccessibility, FootAccessibility);
}

private void SkipNd(XmlPullParser parser) throws IOException, XmlPullParserException {
    parser.require(XmlPullParser.START_TAG, ns, "nd");
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.getEventType() != XmlPullParser.START_TAG) {
            continue;
        }
            skip(parser);
    }
}

private static void skip(XmlPullParser parser) throws XmlPullParserException, IOException {
    if (parser.getEventType() != XmlPullParser.START_TAG) {
        throw new IllegalStateException();
```

```
        }
        int depth = 1;
        while (depth != 0) {
            switch (parser.next()) {
                case XmlPullParser.END_TAG:
                    depth--;
                    break;
                case XmlPullParser.START_TAG:
                    depth++;
                    break;
            }
        }
    }
}
```

Graph.java

```
package com.example.e_road.SearchAlgorithms;

import android.annotation.SuppressLint;

import java.util.HashMap;

@SuppressLint("UseSparseArrays")
public class Graph {
    private static HashMap<Long,Node> Nodes = new HashMap<>();
    private static HashMap<Long,Way> Ways   = new HashMap<>();

    public static void setGraph(HashMap<Long, Node> Nodes,
                                HashMap<Long, Way> Ways){
        Graph.Nodes = Nodes;
        Graph.Ways = Ways;
    }
    public static HashMap<Long, Node> getNodes() {
        return Nodes;
    }
    public static HashMap<Long, Way> getWays() {
        return Ways;
    }
}
```

Node.java

```
package com.example.e_road.SearchAlgorithms;

import com.example.e_road.StaticHelper;

public class Node implements Comparable<Node>{
    private long   Id;
    private double Latitude;
    private double Longitude;
    private Double G;
    private Double F;

    public Node(long   Id,
                double Latitude,
                double Longitude) {
        this.Id        = Id;
        this.Latitude  = Latitude;
        this.Longitude = Longitude;
        this.G         = Double.MAX_VALUE;
    }

    public long getId() {
        return Id;
    }

    public double getLatitude() {
        return Latitude;
    }

    public double getLongitude() {
        return Longitude;
    }

    Double getG() {
        return G;
    }
```

```
    void setG(Double g) {
        G = g;
    }

    Double getF() {
        return F;
    }

    void setF(Double f) {
        F = f;
    }

    @Override
    public int hashCode() {
        return (int)Id;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Node) {
            return this.Id == ((Node) o).Id;
        }
        return false;
    }

    @Override
    public int compareTo(Node Node) {
        if ( StaticHelper.SelectedAlgorithm.equals("Dijkstra") ||
                StaticHelper.SelectedAlgorithm.equals("Bidirectional Dijkstra") )
            return this.G.compareTo(Node.G);
        // in case of A-Star or Bidirectional A-Star
        return this.F.compareTo(Node.F);
    }
}
```

Way.java

```
package com.example.e_road.SearchAlgorithms;

import java.util.List;

public class Way {
    private long       Id;
    private List<Long> NodesInWay;
    private double     CarAccessibility;
    private double     BicycleAccessibility;
    private double     FootAccessibility;

    public Way(long Id,
               List<Long> NodesIds,
               double CarAccessibility,
               double BicycleAccessibility,
               double FootAccessibility) {
        this.Id                   = Id;
        this.NodesInWay           = NodesIds;
        this.CarAccessibility     = CarAccessibility;
        this.BicycleAccessibility = BicycleAccessibility;
        this.FootAccessibility    = FootAccessibility;
    }

    public long getId() {
        return Id;
    }

    public List<Long> getNodesInWay() {
        return NodesInWay;
    }

    double getCarAccessibility() {
        return CarAccessibility;
    }

    double getBicycleAccessibility() {
        return BicycleAccessibility;
    }

    double getFootAccessibility() {
        return FootAccessibility;
    }
```

```
    @Override
    public int hashCode() {
        return (int)Id;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Node) {
            return this.Id == ((Node) o).getId();
        }
        return false;
    }
}
```

Search.java

```
package com.example.e_road.SearchAlgorithms;

import com.example.e_road.StaticHelper;

import org.json.JSONException;

import java.io.IOException;
import java.time.Instant;
import java.util.*;

import static com.example.e_road.AirQuality.AirQualityParser.GetAQILevel;
import static com.example.e_road.Traffic.TrafficParser.GetDataForWeightOfWay;

/**
 * @author Mihai
 */
public class Search {


    private static List<Node> Road = new ArrayList<>();

    private static Node ConnectingNode;

    private static Double BestPathLength = Double.MAX_VALUE;

    public static void SearchPath(Node Start, Node Finish) throws IOException, JSONException {
        //Heuristic by default will be Euclidian distance;
        //Dijkstra algorithm do not have any Heuristic value;
        //We can make A-Star Algorithm work like Dijkstra if it's Heuristic function will return 0;

        StaticHelper.AlgInitialTime = Instant.now();

        ConnectingNode = null;
        BestPathLength = Double.MAX_VALUE;

        Road.clear();

        Start = FindClosestGraphNode(Start);
        Finish = FindClosestGraphNode(Finish);

        System.out.println("Start: " + Start.getId() +
                " Lat,Long: " + Start.getLatitude() + "," + Start.getLongitude());
        System.out.println("Finish: " + Finish.getId() +
                " Lat,Long: " + Finish.getLatitude() + "," + Finish.getLongitude());

        switch ( StaticHelper.SelectedAlgorithm) {
            case "Dijkstra":
                Dijkstra(Start, Finish);
                break;
            case "A-Star":
                AStar(Start, Finish);
                break;
            case "Bidirectional Dijkstra":
                BidirectionalDijkstra(Start, Finish);
                break;
            case "Bidirectional A-Star":
                BidirectionalAStar(Start, Finish);
                break;
            case "NBA-Star":
                NBAStar(Start, Finish);
                break;
        }
        StaticHelper.AlgFinalTime = Instant.now();
    }
```

```
//-------------------------------------------------------------------------------------
//----------------------- METHODS FOR DIJKSTRA ALGORITHM ------------------------------
//-------------------------------------------------------------------------------------

private static void Dijkstra(Node Start, Node Finish) throws IOException, JSONException {
    ArrayList<Node> OpenNodesList = new ArrayList<>();
    List<Node> ClosedNodesList = new ArrayList<>();
    HashMap<Node, Node> Parents = new HashMap<>();
    Node CurrentNode;

    Start.setG(0.0);

    OpenNodesList.add(Start);
    while (!OpenNodesList.isEmpty()) {
        Collections.sort(OpenNodesList);
        CurrentNode = OpenNodesList.get(0);
        OpenNodesList.remove(0);
        ClosedNodesList.add(CurrentNode);

        //Build the path
        BuildUnidirectionalPath(Parents, Finish);
        if (Road.contains(Finish)) {
            StaticHelper.NumOfNodesSearched = (ClosedNodesList.size());
            return;
        }

        for (Map.Entry<Long, Way> entry : Graph.getWays().entrySet()) {
            Way WayValue = entry.getValue();

            //Check Accessibility
            if (!(StaticHelper.isFootRoad ? (WayValue.getFootAccessibility() > 0) :
                    StaticHelper.isCarRoad ? (WayValue.getCarAccessibility() > 0) :
                            WayValue.getBicycleAccessibility() > 0)) {
                continue;
            }
            if (WayValue.getNodesInWay().contains(CurrentNode.getId())) {
                int Index = WayValue.getNodesInWay().indexOf(CurrentNode.getId());
                //Need to treat the case in which our node it is in the end of the list or in
                //front of the list.
                for (int Id = (Index == 0 ? Index + 1 : Index - 1);
                     Id <= (Index == WayValue.getNodesInWay().size() - 1 ?
                             Index - 1 : Index + 1);
                     Id = Id + ((Index == 0 ||
                             Index == WayValue.getNodesInWay().size() - 1) ?
                             1 : 2)) {

                    Long NodeId = WayValue.getNodesInWay().get(Id);
                    Node Neighbour = Graph.getNodes().get(NodeId);
                    if (ClosedNodesList.contains(Neighbour) || Neighbour == null) continue;

                    Double nextG = CurrentNode.getG()
                            + EstimateDistance(CurrentNode, Neighbour, WayValue);

                    if (nextG.compareTo(Neighbour.getG()) > 0) continue;

                    if (!OpenNodesList.contains(Neighbour)) {
                        Neighbour.setG(nextG);
                        //save the index of the parent in a list,
                        //we will use it to make the final road
                        Parents.put(Neighbour, CurrentNode);
                        OpenNodesList.add(Neighbour);
                    }
                }
            }
        }
    }
}

//-------------------------------------------------------------------------------------
//------------------------- METHODS FOR A-STAR ALGORITHM ------------------------------
//-------------------------------------------------------------------------------------

private static void AStar(Node Start, Node Finish) throws IOException, JSONException {
    ArrayList<Node> OpenNodesList = new ArrayList<>();
    List<Node> ClosedNodesList = new ArrayList<>();
    HashMap<Node, Node> Parents = new HashMap<>();
    Node CurrentNode;

    Start.setG(0.0);
```

```
        Start.setF(CalculateHeuristic(Start, Finish));

        OpenNodesList.add(Start);
        while (!OpenNodesList.isEmpty()) {
            Collections.sort(OpenNodesList);
            CurrentNode = OpenNodesList.get(0);
            OpenNodesList.remove(0);
            ClosedNodesList.add(CurrentNode);

            //Build the path
            BuildUnidirectionalPath(Parents, Finish);
            if (Road.contains(Finish)) {
                StaticHelper.NumOfNodesSearched = (ClosedNodesList.size());
                return;
            }

            for (Map.Entry<Long, Way> entry : Graph.getWays().entrySet()) {
                Way WayValue = entry.getValue();
                if (!(StaticHelper.isFootRoad ? (WayValue.getFootAccessibility() > 0) :
                        StaticHelper.isCarRoad ? (WayValue.getCarAccessibility() > 0) :
                                WayValue.getBicycleAccessibility() > 0)) {
                    continue;
                }
                if (WayValue.getNodesInWay().contains(CurrentNode.getId())) {
                    int Index = WayValue.getNodesInWay().indexOf(CurrentNode.getId());
                    //Need to treat the case in which our node it is in the end of the list or in
                    //front of the list.
                    for (int Id = (Index == 0 ? Index + 1 : Index - 1);
                         Id <= (Index == WayValue.getNodesInWay().size() - 1 ?
                                 Index - 1 : Index + 1);
                         Id = Id + ((Index == 0 ||
                                 Index == WayValue.getNodesInWay().size() - 1) ?
                                 1 : 2)) {

                        Long NodeId = WayValue.getNodesInWay().get(Id);
                        Node Neighbour = Graph.getNodes().get(NodeId);
                        if (ClosedNodesList.contains(Neighbour) || Neighbour == null) continue;

                        Double nextG = CurrentNode.getG()
                                + EstimateDistance(CurrentNode, Neighbour, WayValue);

                        if (nextG.compareTo(Neighbour.getG()) > 0) continue;

                        if (!OpenNodesList.contains(Neighbour)) {
                            Neighbour.setG(nextG);
                            Neighbour.setF(Neighbour.getG()
                                    + CalculateHeuristic(Neighbour, Finish));
                            //save the index of the parent in a list,
                            //we will use it to make the final road
                            Parents.put(Neighbour, CurrentNode);
                            OpenNodesList.add(Neighbour);
                        }
                    }
                }
            }
        }
    }

    private static double CalculateHeuristic(Node FirstNode, Node SecondNode) {
        return Math.sqrt(Math.pow((FirstNode.getLatitude() - SecondNode.getLatitude()), 2)
                + Math.pow((FirstNode.getLongitude() - SecondNode.getLongitude()), 2));
    }

    //-------------------------------------------------------------------------------------
    //---------------------- METHODS FOR BIDIRECTIONAL DIJKSTRA ---------------------------
    //-------------------------------------------------------------------------------------

    private static void BidirectionalDijkstra(Node Start, Node Finish)
                                            throws IOException, JSONException {
        ArrayList<Node> ForwardOpenNodesList = new ArrayList<>();
        ArrayList<Node> BackwardOpenNodesList = new ArrayList<>();
        HashMap<Node, Node> ForwardParents = new HashMap<>();
        HashMap<Node, Node> BackwardParents = new HashMap<>();
        LinkedList<Node> ForwardClosedNodesList = new LinkedList<>();
        LinkedList<Node> BackwardClosedNodesList = new LinkedList<>();

        ConnectingNode = null;

        Start.setG(0.0);
        Finish.setG(0.0);
```

```
        ForwardOpenNodesList.add(Start);
        BackwardOpenNodesList.add(Finish);

        while (!ForwardOpenNodesList.isEmpty() && !BackwardOpenNodesList.isEmpty()) {
            DijkstraExpandInDirection(ForwardOpenNodesList, ForwardClosedNodesList,
                    ForwardParents, BackwardParents);
            DijkstraExpandInDirection(BackwardOpenNodesList, BackwardClosedNodesList,
                    BackwardParents, ForwardParents);

            BuildBidirectionalPath(ForwardParents, BackwardParents);
            if (Road.contains(Finish)) {
                StaticHelper.NumOfNodesSearched = (ForwardClosedNodesList.size()
                        + BackwardClosedNodesList.size());
                return;
            }
        }
    }

    private static void DijkstraExpandInDirection(ArrayList<Node> OpenNodesList,
                                                  LinkedList<Node> ClosedNodesList,
                                                  HashMap<Node, Node> Parents,
                                                  HashMap<Node, Node> OppositeParents)
                                                        throws IOException, JSONException {

        Collections.sort(OpenNodesList);
        Node CurrentNode = OpenNodesList.get(0);
        OpenNodesList.remove(0);
        if (ClosedNodesList.contains(CurrentNode)) return;
        ClosedNodesList.add(CurrentNode);

        for (Map.Entry<Long, Way> entry : Graph.getWays().entrySet()) {
            Way WayValue = entry.getValue();

            //Check Accessibility
            if (!(StaticHelper.isFootRoad ? (WayValue.getFootAccessibility() > 0) :
                    StaticHelper.isCarRoad ? (WayValue.getCarAccessibility() > 0) :
                            WayValue.getBicycleAccessibility() > 0)) {
                continue;
            }

            if (WayValue.getNodesInWay().contains(CurrentNode.getId())) {
                int Index = WayValue.getNodesInWay().indexOf(CurrentNode.getId());
                //Need to treat the case in which our node it is in the end of the list or in
                //front of the list.
                for (int Id = (Index == 0 ? Index + 1 : Index - 1);
                     Id <= (Index == WayValue.getNodesInWay().size() - 1 ?
                             Index - 1 : Index + 1);
                     Id = Id + ((Index == 0 ||
                             Index == WayValue.getNodesInWay().size() - 1) ?
                             1 : 2)) {

                    Long NodeId = WayValue.getNodesInWay().get(Id);
                    Node Neighbour = Graph.getNodes().get(NodeId);
                    if (ClosedNodesList.contains(Neighbour) || Neighbour == null) continue;

                    Double nextG = CurrentNode.getG()
                                        + EstimateDistance(CurrentNode, Neighbour, WayValue);

                    if (nextG.compareTo(Neighbour.getG()) > 0) continue;

                    if (!OpenNodesList.contains(Neighbour)) {
                        Neighbour.setG(nextG);
                        //save the index of the parent in a list,
                        //we will use it to make the final road
                        Parents.put(Neighbour, CurrentNode);
                        OpenNodesList.add(Neighbour);
                        if (OppositeParents.containsKey(Neighbour) && ConnectingNode == null)
                            ConnectingNode = Neighbour;
                    }
                }
            }
        }
    }

    //-------------------------------------------------------------------------------------
    //----------------------- METHODS FOR BIDIRECTIONAL A-STAR ----------------------------
    //-------------------------------------------------------------------------------------

    private static void BidirectionalAStar(Node Start, Node Finish) throws IOException,
```

```
JSONException {

        ArrayList<Node> ForwardOpenNodesList = new ArrayList<>();
        ArrayList<Node> BackwardOpenNodesList = new ArrayList<>();
        HashMap<Node, Node> ForwardParents = new HashMap<>();
        HashMap<Node, Node> BackwardParents = new HashMap<>();
        LinkedList<Node> ForwardClosedNodesList = new LinkedList<>();
        LinkedList<Node> BackwardClosedNodesList = new LinkedList<>();

        Start.setG(0.0);
        Finish.setG(0.0);
        Start.setF(CalculateHeuristic(Start, Finish));
        Finish.setF(CalculateHeuristic(Finish, Start));

        ForwardOpenNodesList.add(Start);
        BackwardOpenNodesList.add(Finish);

        while (!ForwardOpenNodesList.isEmpty() && !BackwardOpenNodesList.isEmpty()) {
            AStarExpandInDirection(ForwardOpenNodesList, Finish, ForwardClosedNodesList,
                    ForwardParents, BackwardParents);
            AStarExpandInDirection(BackwardOpenNodesList, Start, BackwardClosedNodesList,
                    BackwardParents, ForwardParents);

            BuildBidirectionalPath(ForwardParents, BackwardParents);
            if (Road.contains(Finish)) {
                StaticHelper.NumOfNodesSearched = (ForwardClosedNodesList.size()
                        + BackwardClosedNodesList.size());
                return;
            }
        }
    }

    private static void AStarExpandInDirection(ArrayList<Node> OpenNodesList, Node Finish,
                                               LinkedList<Node> ClosedNodesList,
                                               HashMap<Node, Node> Parents,
                                               HashMap<Node, Node> OppositeParents)
                                                    throws IOException, JSONException {

        Collections.sort(OpenNodesList);
        Node CurrentNode = OpenNodesList.get(0);
        OpenNodesList.remove(0);
        if (ClosedNodesList.contains(CurrentNode)) return;
        ClosedNodesList.add(CurrentNode);

        for (Map.Entry<Long, Way> entry : Graph.getWays().entrySet()) {
            Way WayValue = entry.getValue();

            //Check Accessibility
            if (!(StaticHelper.isFootRoad ? (WayValue.getFootAccessibility() > 0) :
                    StaticHelper.isCarRoad ? (WayValue.getCarAccessibility() > 0) :
                            WayValue.getBicycleAccessibility() > 0)) {
                continue;
            }

            if (WayValue.getNodesInWay().contains(CurrentNode.getId())) {
                int Index = WayValue.getNodesInWay().indexOf(CurrentNode.getId());
                //Need to treat the case in which our node it is in the end of the list or in
                //front of the list.
                for (int Id = (Index == 0 ? Index + 1 : Index - 1);
                     Id <= (Index == WayValue.getNodesInWay().size() - 1 ?
                             Index - 1 : Index + 1);
                     Id = Id + ((Index == 0 ||
                             Index == WayValue.getNodesInWay().size() - 1) ?
                             1 : 2)) {

                    Long NodeId = WayValue.getNodesInWay().get(Id);
                    Node Neighbour = Graph.getNodes().get(NodeId);
                    if (ClosedNodesList.contains(Neighbour) || Neighbour == null) continue;

                    Double nextG = CurrentNode.getG()
                            + EstimateDistance(CurrentNode, Neighbour, WayValue);

                    if (nextG.compareTo(Neighbour.getG()) > 0) continue;

                    if (!OpenNodesList.contains(Neighbour)) {
                        Neighbour.setG(nextG);
                        Neighbour.setF(Neighbour.getG() + CalculateHeuristic(Neighbour, Finish));
                        //save the index of the parent in a list,
                        //we will use it to make the final road
                        Parents.put(Neighbour, CurrentNode);
```

```
                    OpenNodesList.add(Neighbour);
                    if (OppositeParents.containsValue(Neighbour) && ConnectingNode == null)
                        ConnectingNode = Neighbour;
                }
            }
        }
    }
}

//This method it was proposed in order to simplify the processing power of the
//Bidirectional AStar
private static void NBAStar(Node Start, Node Finish) throws IOException, JSONException {
    ArrayList<Node> ForwardOpenNodesList = new ArrayList<>();
    ArrayList<Node> BackwardOpenNodesList = new ArrayList<>();
    HashMap<Node, Node> ForwardParents = new HashMap<>();
    HashMap<Node, Node> BackwardParents = new HashMap<>();
    LinkedList<Node> ForwardClosedNodesList = new LinkedList<>();
    LinkedList<Node> BackwardClosedNodesList = new LinkedList<>();

    Start.setG(0.0);
    Finish.setG(0.0);
    Start.setF(CalculateHeuristic(Start, Finish));
    Finish.setF(CalculateHeuristic(Finish, Start));

    ForwardOpenNodesList.add(Start);
    BackwardOpenNodesList.add(Finish);

    while (!ForwardOpenNodesList.isEmpty() && !BackwardOpenNodesList.isEmpty()) {
        NBAStarExpandInDirection(ForwardOpenNodesList, BackwardOpenNodesList,
                Start, Finish, ForwardClosedNodesList, ForwardParents, BackwardParents);
        NBAStarExpandInDirection(BackwardOpenNodesList, ForwardOpenNodesList,
                Finish, Start, BackwardClosedNodesList, BackwardParents, ForwardParents);

        BuildBidirectionalPath(ForwardParents, BackwardParents);
        if (Road.contains(Finish)) {
            StaticHelper.NumOfNodesSearched = (ForwardClosedNodesList.size()
                    + BackwardClosedNodesList.size());
            return;
        }
    }

}

private static void NBAStarExpandInDirection(ArrayList<Node> OpenNodesList,
                                             ArrayList<Node> OppositeOpenNodesList,
                                             Node Start, Node Finish,
                                             LinkedList<Node> ClosedNodesList,
                                             HashMap<Node, Node> Parents,
                                             HashMap<Node, Node> OppositeParents)
                                                    throws IOException, JSONException {

    Collections.sort(OpenNodesList);
    Node CurrentNode = OpenNodesList.get(0);
    OpenNodesList.remove(0);
    if (ClosedNodesList.contains(CurrentNode)) return;
    ClosedNodesList.add(CurrentNode);
    Collections.sort(OppositeOpenNodesList);
    Node OppositeNode = OppositeOpenNodesList.get(0);

    if (CurrentNode.getG() + CalculateHeuristic(CurrentNode, Finish) < BestPathLength ||
            CurrentNode.getG() + OppositeNode.getF()
                    - CalculateHeuristic(CurrentNode, Start) < BestPathLength) {

        for (Map.Entry<Long, Way> entry : Graph.getWays().entrySet()) {
            Way WayValue = entry.getValue();

            //Check Accessibility
            if (!(StaticHelper.isFootRoad ? (WayValue.getFootAccessibility() > 0) :
                    StaticHelper.isCarRoad ? (WayValue.getCarAccessibility() > 0) :
                            WayValue.getBicycleAccessibility() > 0)) {
                continue;
            }

            if (WayValue.getNodesInWay().contains(CurrentNode.getId())) {
                int Index = WayValue.getNodesInWay().indexOf(CurrentNode.getId());
                //Need to treat the case in which our node it is in the end of the list or in
                //front of the list.
                for (int Id = (Index == 0 ? Index + 1 : Index - 1);
                     Id <= (Index == WayValue.getNodesInWay().size() - 1 ?
                                Index - 1 : Index + 1);
```

```
                    Id = Id + ((Index == 0 ||
                                Index == WayValue.getNodesInWay().size() - 1) ?
                                1 : 2)) {

                    Long NodeId = WayValue.getNodesInWay().get(Id);
                    Node Neighbour = Graph.getNodes().get(NodeId);
                    if (ClosedNodesList.contains(Neighbour) || Neighbour == null) continue;

                    Double nextG = CurrentNode.getG()
                                    + EstimateDistance(CurrentNode, Neighbour, WayValue);

                    if (nextG.compareTo(Neighbour.getG()) > 0) continue;

                    if (!OpenNodesList.contains(Neighbour)) {
                        Neighbour.setG(nextG);
                        Neighbour.setF(Neighbour.getG()
                                        + CalculateHeuristic(Neighbour, Finish));
                        //save the index of the parent in a list,
                        //we will use it to make the final road
                        Parents.put(Neighbour, CurrentNode);
                        OpenNodesList.add(Neighbour);
                        if (Neighbour.getG()
                                + CalculateHeuristic(Neighbour, Finish) < BestPathLength &&
                                OppositeParents.containsKey(Neighbour) &&
                                ConnectingNode == null) {
                            BestPathLength = Neighbour.getG()
                                            + CalculateHeuristic(Neighbour, Finish);
                            ConnectingNode = Neighbour;
                        }
                    }
                }
            }
        }
    }
}

//-------------------------------------------------------------------------------------
//-------------------------COMMON METHODS FOR ALL ALGORITHMS---------------------------
//-------------------------------------------------------------------------------------

private static double EstimateDistance(Node FirstNode, Node SecondNode, Way WayValue)
                                        throws IOException, JSONException {
    double ValFromApi = 0;
    if (StaticHelper.IsPollutionChecked)
        ValFromApi += Double.parseDouble(GetAQILevel(SecondNode.getLatitude(),
                                    SecondNode.getLongitude())) * 0.000001;
    if (StaticHelper.IsAccessibilityChecked) {
        double Accessibility = (StaticHelper.isFootRoad ? (WayValue.getFootAccessibility()) :
                                StaticHelper.isCarRoad ? (WayValue.getCarAccessibility()) :
                                    WayValue.getBicycleAccessibility());
        ValFromApi += Accessibility * 0.00001;
    }
    if (StaticHelper.IsTrafficChecked)
        ValFromApi += GetDataForWeightOfWay(SecondNode.getLatitude(),
                                        SecondNode.getLongitude()) * 0.000001;

    return Math.sqrt(Math.pow((FirstNode.getLatitude() - SecondNode.getLatitude()), 2)
            + Math.pow((FirstNode.getLongitude() - SecondNode.getLongitude()), 2)) + ValFromApi;
}

private static Node FindClosestGraphNode(Node CurrentNode) {
    double aux;
    double min = Double.MAX_VALUE;
    Node ClosestNode = null;
    for (Map.Entry<Long, Node> entry : Graph.getNodes().entrySet()) {
        aux=Math.sqrt(Math.pow((entry.getValue().getLatitude() - CurrentNode.getLatitude()), 2)
                + Math.pow((entry.getValue().getLongitude() - CurrentNode.getLongitude()), 2));
        if (min > aux) {
            min = aux;
            ClosestNode = entry.getValue();
        }
    }
    return ClosestNode;
}

//-------------------------------------------------------------------------------------
//----------------------METHODS FOR CONSTRUCTING THE ROAD------------------------------
//-------------------------------------------------------------------------------------

private static void BuildUnidirectionalPath(HashMap<Node, Node> Parents, Node Finish) {
```

```
        if (Parents.containsKey(Finish)) {
            Road.add(Finish);
            Node AuxNode = Parents.get(Finish);
            while (AuxNode != null) {
                Road.add(AuxNode);
                AuxNode = Parents.get(AuxNode);
            }
        }
    }

    private static void BuildBidirectionalPath(HashMap<Node, Node> ForwardParents,
                                               HashMap<Node, Node> BackwardParents) {
        if (ConnectingNode != null) {
            Node AuxNode = ConnectingNode;
            while (AuxNode != null) {
                Road.add(AuxNode);
                AuxNode = ForwardParents.get(AuxNode);
            }
            Collections.reverse(Road);
            AuxNode = BackwardParents.get(ConnectingNode);
            while (AuxNode != null) {
                Road.add(AuxNode);
                AuxNode = BackwardParents.get(AuxNode);
            }
        }
    }

    public static List<Node> getRoad() {
        return Road;
    }
}
```