

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

LUCRARE DE DIPLOMĂ

Coordonator științific:
Prof.dr.inf. Craus Mitică

Absolvent:
Adrian-Ioan Chihalău

Iași, 2022

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

Simularea traseului unei drone de supraveghere a străzilor unei zone rezidențiale

LUCRARE DE DIPLOMĂ

Coordonator științific:
Prof.dr.inf. Craus Mitică

Absolvent:
Adrian-Ioan Chihalău

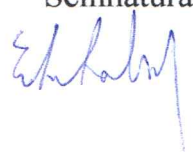
Iași, 2022

**DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII
PROIECTULUI DE DIPLOMĂ**

Subsemnatul CHIHALĂU ADRIAN-IOAN,
legitimat cu CI seria NZ nr. 163188, CNP 1990829270851
autorul lucrării SIMULAREA TRASEULUI UNEI DRONE DE
SUPRAVEGHERE A STRĂZILOR UNEI ZONE
REZIDENTIALE

elaborată în vederea susținerii examenului de finalizare a studiilor de licență, programul de studii CTI organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea SEPTEMBRIE a anului universitar 2022, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTI.POM.02 - Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data
02.09.2022

Semnătura


Cuprins

Introducere	1
I Motivația alegerii temei	1
II Relevanța și contextul temei alese	1
II.I Supravegherea traficului prin intermediul dronelor	1
II.II Aplicații matematice de rezolvare al problemelor de cost	1
III Structura și metodele lucrării	2
1 Fundamentarea teoretică și documentarea bibliografică	3
1 Elemente de teorie a grafurilor	3
1.1 Grafurile, grafurile simple și componentele acestora	3
1.2 Matricile de incidență și adiacență	5
1.3 Gradul unui nod	5
1.4 Drumuri, conexiuni și cicluri	6
2 Turul Eulerian	7
3 Problema poștașului chinez	8
3.1 Enunțul problemei și variantele acesteia	8
3.2 Soluții de rezolvare al problemei - cazul grafului eulerian	9
3.3 Soluții de rezolvare al problemei - cazul grafului non-eulerian	11
4 Paradigma Greedy	12
4.1 Caracteristici generale	12
4.2 Modelul matematic	12
4.3 Analiza paradigmei greedy	13
2 Proiectarea aplicației	15
1 Componente hardware	15
2 Componente software	16
2.1 OpenStreetMap și formatul de fișier .osm	16
2.2 Limbajul de programare Python	19
2.3 Module Python utilizate în cadrul aplicației	19
2.3.1 Modulul numpy	19
2.3.2 Modulul threading	19
2.3.3 Modulul concurrent.futures	20
2.3.4 Modulul xml	20
2.3.5 Modulul networkx	20
2.3.6 Modulul matplotlib	20
2.3.7 Modulul PyQt5	21
2.4 Mediul de dezvoltare PyCharm	21
2.5 Algoritmi folosiți în cadrul aplicației	22
2.5.1 Algoritmul Floyd-Warshall	22
2.5.2 Algoritmul lui Dijkstra	23

2.5.3	Studiu de caz: Floyd-Warshall versus Dijkstra	24
2.5.4	Paradigma Greedy	25
2.5.5	Algoritmul lui Hierholzer	27
2.5.6	Algoritmul BFS	28
2.6	Proiectarea interfeței aplicației	29
3	Implementarea aplicației	31
1	Descrierea generală a implementării	31
1.1	Componenta de prelucrare a hărții într-un graf	31
1.2	Componenta de găsim și eliminare al nodurilor izolate	34
1.3	Componenta de calcul al drumurilor între noduri de grad impar	36
1.4	Componenta de calcul al soluției problemei poștaşului chinez	39
1.5	Componenta de determinare al turului eulerian	40
1.6	Componenta principală: funcția de generare al turului eulerian pornind de la un fișier .osm	41
2	Implementarea interfeței cu utilizatorul (GUI)	45
4	Testarea aplicației și rezultate experimentale	47
1	Elemente de configurare și instalare	47
2	Limitări în procesarea datelor din fișier	47
3	Limitări din punct de vedere al operațiilor multitasking	48
4	Testarea aplicației	49
4.1	Măsurarea performanței	49
4.1.1	Testul de performanță al algoritmilor de drumuri minime	49
4.1.2	Testul de performanță al aplicației în funcție de dimensiunea grafurilor analizate	50
4.1.3	Testul de performanță al aplicației în funcție de sistemul de operare	50
	Concluzii	53
	Bibliografie	55
	Anexe	57
1	Funcția de extragere a grafului unui oraș aflat în harta stocată într-un fișier	57
2	Funcția de căutare al nodurilor izolate dintr-un graf	60
3	Funcții de recreere al drumurilor dintr-un graf	62
4	Funcția de determinare al turului eulerian folosind algoritmul Hierholzer	63
5	Funcția de generare al turului eulerian pentru graful unui oraș extras din harta corespunzătoare unui fișier XML OSM	66
6	Interfața cu utilizatorul	70

Simularea traseului unei drone de supraveghere a străzilor unei zone rezidențiale

Adrian-Ioan Chihalău

Rezumat

Această lucrare prezintă o aplicație pentru calcularea și determinarea traseului unei drone care parcurge toate străzile unui oraș, al cărei hărți este oferită ca și intrare. Având în vedere că toate străzile trebuie parcurse cel puțin o dată, programul rezolvă un caz particular al problemei circuitului eulerian, intitulată „Problema poștașului chinez”, situație ce va fi explicată în detaliu în cadrul documentației.

Astfel se vor descrie fundamentul teoretic al grafurilor, în special al grafului eulerian, drumul și circuitul eulerian, cât și problema poștașului chinez. De asemenea se vor discuta despre algoritmi necesari pentru determinarea lungimii străzilor, pentru determinarea costului minim al circuitului eulerian, cât și a obținerii soluției finale.

Mai departe, sunt prezentate caracteristicile necesare execuției aplicației, și bibliotecile necesare realizării programului, printre care `xml.etree` pentru lucru cu fișiere `.xml`, `networkx` pentru crearea de grafuri, `PyQT5` pentru construirea interfeței aplicației, cât și `pyplot` pentru afișarea grafului orașului.

În cadrul descrierii implementării aplicației, se va prezenta succint formatul `.osm`, asemănător cu `.xml` care conține date despre poziția străzilor, orașelor dintr-o regiune, preluată de pe website-ul `OpenStreetMap`, ce permite preluarea open-source al acestor date, față de API-ul `Google Maps`, care necesită costuri suplimentare.

Apoi, se explică modul de realizare al prelucrării informațiilor din aceste fișiere, calcularea distanțelor nodurilor grafului obținut și afișarea rezultatului sub formă de animație, în format `.gif`.

La final, sunt evidențiate performanțele aplicației din punct de vedere al timpului și spațiului, cât și date de comparație între diverse metode de determinare a circuitului eulerian, din punct de vedere al timpului de lucru, în urma cărora se vor trage concluzii. De aceea, se constată că rezolvarea problemei poștașului chinez este cea mai potrivită pentru astfel de situații, iar metodele folosite sunt optime pentru obținerea soluției dorite.

Introducere

I. Motivația alegerii temei

Algoritmica m-a atras încă din perioada liceului, datorită construirii logice a soluțiilor unor probleme, într-un mod inventiv și practic. De asemenea teoria grafurilor a fost un domeniu care m-a fascinat, și a căror descoperiri le-am aprofundat mai bine în perioada facultății. Prin urmare, am decis să combin cele 2 domenii și să realizez o aplicație practică.

Aplicațiile circuitului eulerian sunt nenumărate, însă m-am hotărât să aleg ca și tematică supravegherea străzilor printr-o dronă într-un mod eficient. În acest fel am avut ocazia să lucrez și cu API-uri pentru obținerea informațiilor despre locații, lucru care l-am învățat în practica de vară. În ceea ce privește drona, nu am avut resursele necesare de a obține una, deci m-am limitat la o aplicație de simulare a traseului acestuia. Până la urmă, este bine să se știe dinainte care este capacitatea dronei și cât de mult va parcurge un traseu, iar obiectul muncii va arăta acest lucru.

II. Relevanța și contextul temei alese

II.I. Supravegherea traficului prin intermediul dronelor

În orice comunitate urbană, apare inevitabil problema gestiunii traficului autovehiculelor, fie din cauza infrastructurii insuficiente pentru a gestiona volumul de mașini care vin și pleacă, fie dirijarea acesteia lasă de dorit. Oricare ar fi cauza, aceasta necesită întâi observarea fluxului de trafic, pentru a putea lua decizii cu scopul ameliorării, sau ideal, eliminării totale al aglomerației.

Conform indexului de trafic TomTom, în 2021, în București, durata medie a unei drum parcurs este cu 50% mai lungă decât ar fi în mod normal. Cele mai aglomerate perioade ale anului sunt zilele pascale, weekend-urile dinainte de mini-vacanțe (13-14 iulie, respectiv 12-13 august), cât și înainte de Crăciun și Anul Nou. În ceea ce privește cele mai congestionate ore ale zilei, indexul recomandă evitarea perioadei de 5-6 seara, atunci când majoritatea oamenilor ies de la muncă [1].

Totuși într-un oraș există sute de intersecții și străzi, iar măsurarea traficului devine dificilă dacă se folosesc mijloace obișnuite. Bineînțeles se pot folosi camere video pentru a putea vizualiza din oră în oră câte autovehicule trec printr-o intersecție și unde au loc ambuteiaje, dar aceste instrumente devin costisitoare dacă sunt amplasate la fiecare colț de stradă.

Dronele sunt alternative mai ieftine pentru supravegherea video din orașe, indiferent din ce motive. În plus, nu sunt invazive, pentru că nu deranjează nici traficul, nici persoanele care se deplasează pe trotuar, iar când sunt dotate cu camere video, acestea devin camere de înregistrare mobile. Atu-l principal reprezintă supravegherea mai precisă a străzilor - oferă o imagine de ansamblu a acestora și ambuteiajele ce au loc acolo. Totuși trebuie să ținem cont că sunt dispozitive electrice, ce necesită energie pentru a funcționa, iar atunci când se dorește utilizarea unei drone pentru a supraveghea fiecare stradă din oraș, consumul bateriei devine o altă problemă.

II.II. Aplicații matematice de rezolvare al problemelor de cost

Când vine vorba de o dronă care să parcurgă toate străzile dintr-un oraș, fără a-și consuma toată bateria în timpul traseului, este fără doar și poate necesar să fie găsită o rută de cost minim. În matematică, o astfel de problemă se referă la determinarea unui circuit eulerian, a cărei denumire provine de la matematicianul german *Leonhard Euler*. Acesta a pus bazele teoriei grafurilor,

pornind de la *Problema celor Șapte Poduri din Königsberg*, formulată în anul 1736: dacă există o cale care trece prin toate podurile o singură dată și să se întoarcă în același punct de plecare.

Matematicianul a demonstrat faptul că pentru un astfel de drum, este necesar ca în toate intersecțiile căi trebuie să treacă un număr par de străzi, iar reciproca a fost demonstrată mai târziu în 1873 de către Carl Hierholzer, formând astfel *Teorema lui Euler*: Un graf conex are un ciclu eulerian dacă și numai dacă orice nod are gradul par.

În cele mai multe cazuri, o localitate nu are un astfel de ciclu, deci este imposibil de determinat o astfel de soluție, fără a trece mai mult de o dată pe străzile deja vizitate. Astfel în 1960, matematicianul chinez Kwan Mei-Ko a studiat o astfel de problemă, botezată „*Problema poștaşului chinez*” în onoarea sa, care presupune găsirea celui mai scurt circuit care vizitează fiecare stradă cel puțin o dată. Dacă graful specific localității are un circuit eulerian, atunci acesta este soluția optimă; altfel să se găsească cel mai mic număr de muchii sau cel mai puțin costisitor set de muchii care vor fi duplicate, astfel încât graful rezultat să conțină un circuit eulerian.

III. Structura și metodele lucrării

Obiectul muncii va fi prezentat în 5 capitole principale:

- Fundamentarea teoretică completă asupra principalelor noțiuni ale grafurilor, despre graful eulerian, discuție asupra problemei „poștaşului chinez”, și algoritmi folosiți în rezolvarea acesteia, cât și pentru determinarea circuitului eulerian;
- Descriere în amănunte asupra proiectării aplicației, atât din punct de vedere hardware (sistemul de calcul folosit pentru rularea aplicației), cât și din punct de vedere software (limbajul de programare folosit, tehnologii folosite pentru implementarea componentelor aplicației);
- Prezentarea modalității de implementare a aplicației, conținând o descriere a funcționării programului, evidențierea ideilor noi, alături de dificultățile întâmpinate și soluțiile de rezolvare;
- Testarea aplicației și afișarea rezultatelor experimentale, în care se vor explica modul de lansare a aplicației, împreună cu elementele de configurat sau instalat, datele de test folosite pentru testare, iar la urmă rezultatele obținute;
- Rubrica de discuții, unde se va prezenta gradul de finalizare a temei propuse, comparații față de alte teme similare, și în final se vor preciza viitoarele direcții de dezvoltare.

La finalul lucrării va fi atașată anexa, care include codul sursă al componentelor realizate în original din aplicație, cât și date statistice suplimentare.

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică

Teoria grafurilor reprezintă studiul structurilor matematice folosite pentru a modela relații între obiecte, structuri denumite *grafuri*. Elementele principale ale teoriei grafurilor, ale căror noțiuni vor fi importante în explicația funcționării programului, vor fi descrise în următoarea secțiune. Explicațiile sunt bazate pe cartea [2].

1. Elemente de teorie a grafurilor

1.1. Grafurile, grafurile simple și componentele acestora

Un *graf* G reprezintă o pereche de două mulțimi $(N(G), M(G))$, unde $N(G)$ reprezintă mulțimea nevidă a nodurilor, în timp ce $M(G)$ reprezintă mulțimea de perechi de noduri, fiecare pereche reprezentând o muchie. În [2], $M(G)$ - notat în carte ca $E(G)$ - este reprezentat ca mulțimea muchiilor notate doar cu litere mici, iar graful devine un triplet, adăugând o funcție „ ψ_G ” care asociază fiecare muchie din mulțimea $E(G)$ cu o pereche neordonată, dar nu neapărat distinctă, de noduri din mulțimea $N(G)$. Având în vedere că perechile de noduri pot fi mai degrabă reprezentate direct în mulțimea $M(G)$ (fiind mai compactă și mai ușor de înțeles), se va folosi prima definiție menționată.

Astfel un graf va fi reprezentat prin formula:

$$G = (N(G), M(G)).$$

Un *nod* dintr-un graf reprezintă un punct desenat oriunde în spațiu, în timp ce o *muchie* este o linie care leagă două noduri, nu neapărat distincte, între ele. De aceea grafurile sunt numite astfel, pentru că sunt reprezentate grafic, iar lucrul acesta ne ajută să înțelegem proprietățile acestora. Nodurile reprezintă diverse entități corelate cu obiecte, evenimente din viața reală, în timp ce muchiile întruchipează relațiile dintre acestea.

În toate cazurile din viața reală, grafurile sunt reprezentate drept *finite*, adică mulțimile $N(G)$ și $M(G)$ au număr finit de elemente. De aceea, termenul de „graf finit” va coincide cu termenul „graf” și viceversa de-a lungul lucrării. Un exemplu este reprezentat în figura de mai jos:

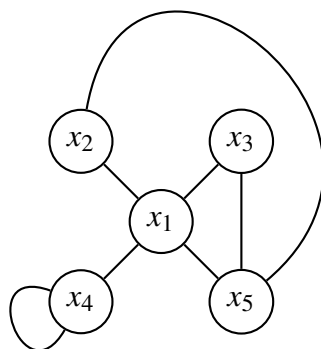


Figura 1.1. Exemplu de graf neorientat

Din figură reiese un graf $G = (N(G), M(G))$, unde $N(G) = \{x_1, x_2, x_3, x_4, x_5\}$, iar $M(G) =$

$\{(x_1, x_2), (x_1, x_3), (x_1, x_4), (x_4, x_4), (x_1, x_5), (x_2, x_5), (x_3, x_5)\}$. Un astfel de graf se mai numește *ne-orientat*, pentru că muchiile pot fi parcurse din ambele direcții, deci ordinea nodurilor din perechi este irelevantă. În schimb dacă considerăm totuși ordinea, graficul G va deveni *orientat*, iar muchiile vor avea un singur sens de parcurgere: de la primul nod din pereche până la celălalt nod; în acest caz, muchiile se vor numi arce.

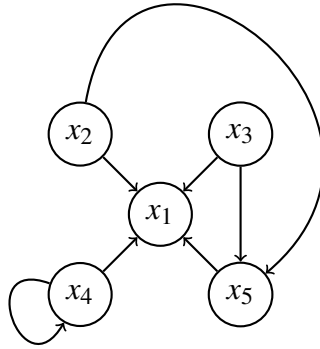


Figura 1.2. Graful din 1.1 în cazul în care mulțimea $M(G)$ se consideră ordinea de parcurgere al nodurilor, rezultând un graf orientat

Pentru următoarele definiții și ținând cont de contextul aplicației (orice stradă din oraș poate fi parcursă din ambele direcții de către dronă, indiferent dacă are sens unic sau nu), se vor folosi doar grafuri neorientate finite. De asemenea se va nota cu V mulțimea tuturor nodurilor dintr-un graf, cu numărul acestora notat cu v , iar mulțimea tuturor muchiilor notat cu E , cu numărul acestora notat cu ε .

Cele mai multe definiții și concepte din teoria grafurilor sunt sugerate prin reprezentarea grafică. Capetele unei muchii sunt considerate *incidente* față de aceasta, și viceversa. Două noduri care sunt incidente cu o muchie comună sunt numite *adiacente*, la fel și două muchii care sunt incidente cu un nod comun. O muchie care are capetele identice se numește *bucă*, iar muchia cu capete distincte o *legătură* (en. *link*). De exemplu, în Figura 1.1, muchia (x_4, x_4) este o buclă, în timp ce celelalte muchii sunt legături.

Un graf este *simplu* dacă nu are nicio buclă și nu există două legături care să se unească în aceeași pereche de noduri. Mare parte din teoria grafurilor se bazează pe studiul acestor grafuri [2, pag. 3].

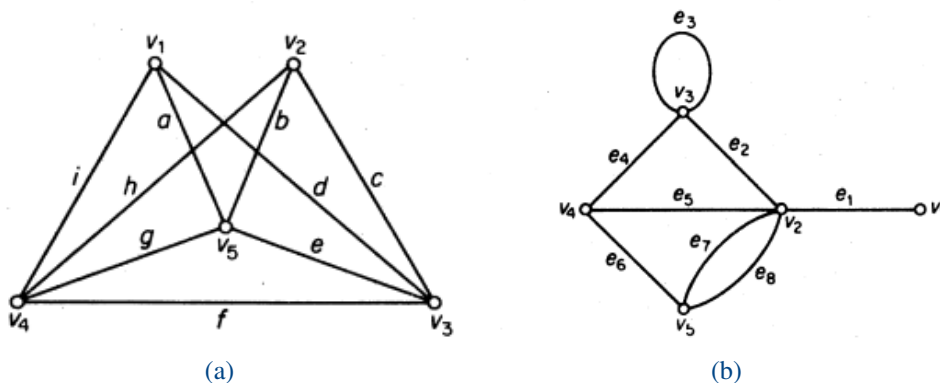


Figura 1.3. Diferența între un graf simplu (a) și un graf non-simplu (b), imagini preluate din [2]

1.2. Matricile de incidență și adiacență

Fiecare graf G corespunde unei matrici de forma $[v \times \varepsilon]$ numită *matrice de incidență*. Considerăm nodurile din G notate prin v_1, v_2, \dots, v_v , și muchiile din acesta prin $e_1, e_2, \dots, e_\varepsilon$. *Matricea de incidență* a lui G este reprezentată prin $M(G) = [m_{ij}]$, unde m_{ij} este numărul de ori nodul v_i și muchia e_j sunt incidente. Matricea de incidență este doar un alt mod de a descrie graful.

O altă matrice asociată cu G este *matricea de adiacență*; este o matrice de forma $v \times v$, $A(G) = [a_{ij}]$, unde a_{ij} este numărul de muchii care au capetele v_i și v_j . Un exemplu de matrice de incidență, cât și o matrice de adiacență, este afișat în figura de mai jos [2, Pag. 7]. În cadrul aplicației, se va folosi matricea de adiacență a unui graf pentru a reprezenta străzile dintr-un oraș.

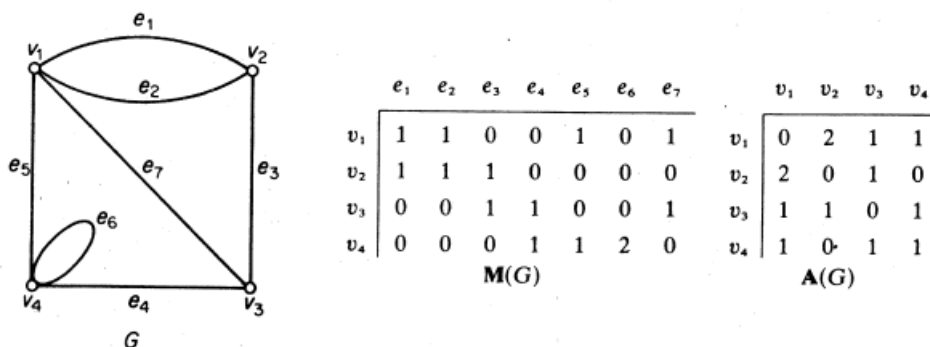


Figura 1.4. Reprezentarea pentru graful G a matricii de incidență $M(G)$, respectiv a matricii de adiacență $A(G)$; figură preluată din [2]

1.3. Gradul unui nod

Gradul $d_G(v)$ a unui nod v din G este numărul de muchii ai lui G incidente cu v , fiecare buclă considerându-se a fi 2 muchii.

Din [2] se prezintă următoarea teoremă și corolar:

Teorema 1.1.

$$\sum_{v \in V} d(v) = 2\varepsilon$$

Demonstrație. Fie matricea de incidență M . Suma tuturor elementelor din M pe linia corespunzătoare nodului v este exact $d(v)$, prin urmare $\sum_{v \in V} d(v)$ este suma tuturor elementelor din M . Dar aceasta este de asemenea egală cu 2ε , deoarece cele ε sume pe fiecare coloană a lui M , corespunzător fiecărei muchii din E , este 2 (o muchie este întotdeauna incidentă cu 2 noduri, distincte sau nu). \square

Corolarul 1.1. În orice graf, numărul de noduri de grad impar este par.

Demonstrație. Fie V_1 și V_2 seturi de noduri de grad impar, respectiv par din G . Atunci

$$\sum_{v \in V_1} d(v) + \sum_{v \in V_2} d(v) = \sum_{v \in V} d(v)$$

este pară, conform teoremei (1.1). Având în vedere că $\sum_{v \in V_2} d(v)$ este pară, atunci și $\sum_{v \in V_1} d(v)$ este pară. Deci $|V_1|$ este par. \square

Corolarul (1.1) va fi util în rezolvarea problemei poștașului chinez, a cărei discuție va avea loc mai târziu.

1.4. Drumuri, conexiuni și cicluri

Un *drum* în G este o secvență nenulă finită $W = v_0v_1v_2 \dots v_k$, ai căror termeni sunt noduri, astfel încât, pentru $1 \leq k$, unde k este lungimea lui W , 2 noduri alăturate v_{i-1} și v_i reprezintă capetele unei muchi, formând o pereche care aparține lui E . Spunem că W este un drum de la v_0 la v_k , sau un *drum* (v_0, v_k) . Nodurile v_0 și v_k sunt numite *origine*, respectiv *destinație* ai lui W , în timp ce $v_1v_2 \dots v_{k-1}$ sunt *nodurile interne* ai acestuia.

Dacă muchiile, formate din nodurile adiacente, din W sunt distincte, atunci W este numit *traseu* [en. *trail*]; în acest caz lungimea lui W este $\varepsilon(W)$. Dacă, în plus, nodurile $v_0, v_1, v_2, \dots, v_k$ sunt distincte, atunci W este numit *cale* [en. *path*]. Figura de mai jos ilustrează un drum, un traseu, respectiv o cale într-un graf. Se va folosi termenul de „cale” pentru a descrie un graf ai căror noduri și muchii sunt componentele unei căi.

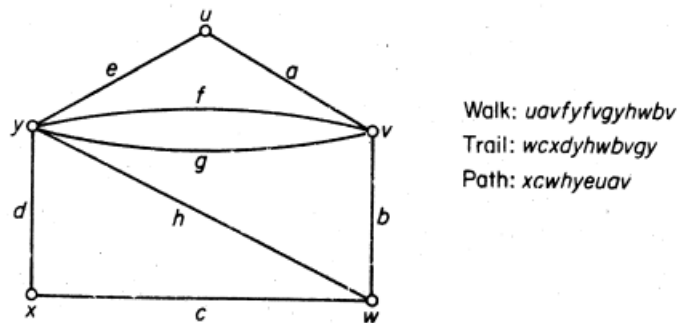


Figura 1.5. Exemplu de drum (path), traseu (trail) și cale (path) pornind de la grafurile alăturate; exemplu preluat din [2]

Două noduri u și v din G sunt *conexe* dacă există o *cale* (u, v) în G . Conexiunea este o relație de echivalență în setul de noduri V . Astfel fie o partiție a lui V în submulțimi nenule $V_1, V_2, \dots, V_\omega$ astfel încât două noduri u și v sunt conexe dacă și numai dacă ambele noduri aparțin aceluiași set V_i . Subgrafurile $G[V_1], G[V_2], \dots, G[V_\omega]$ sunt numite componente ale lui G . Dacă G are exact o componentă, atunci G este *conex*; altfel G este *neconex*. Se notează numărul de componente ale lui G prin $\omega(G)$. Mai jos sunt prezentate un graf conex, respectiv unul neconex.

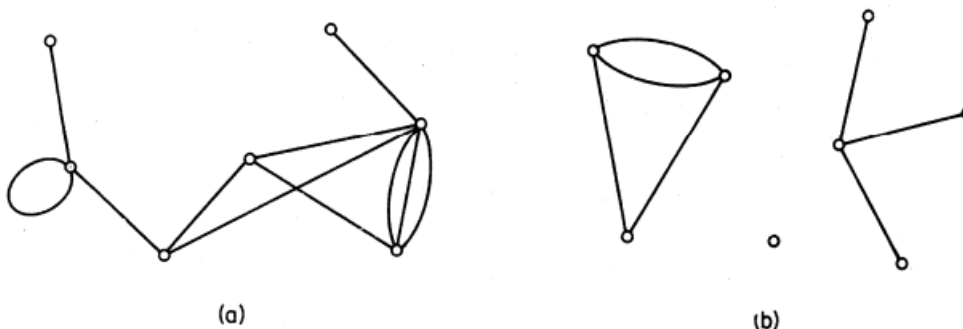


Figura 1.6. (a) Graf conex; (b) Graf neconex; figură preluată din [2]

Un drum este *închis* dacă are lungime pozitivă, iar originea și destinația acestuia este aceeași. Un traseu închis ai cărei origine și noduri interne sunt distincte se numește *ciclu*. Asemenea căilor, se va folosi termenul „ciclu” pentru un graf care reprezintă un ciclu. Figura 1.7 ilustrează diferența dintre cele 2 noțiuni.

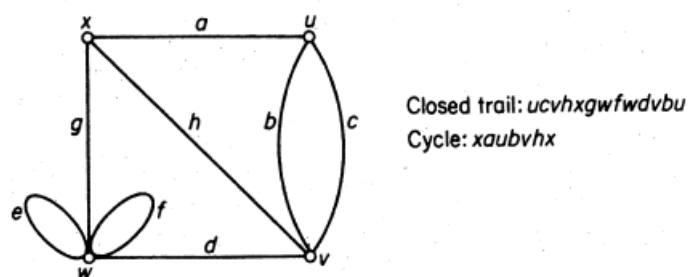


Figura 1.7. Diferența între un traseu închis și un ciclu; figură preluată din [2]

2. Turul Eulerian

Un traseu care traversează fiecare muchie dintr-un graf G se numește *traseu eulerian*. Denumirea lui provine de la matematicianul Leonhard Euler, care a fost primul care a investigat existența lor în grafuri. În cel mai vechi articol cunoscut al teoriei grafurilor (Euler, 1736), acesta a demonstrat că este imposibil să fie traversate cele șapte poduri din Königsberg o singură dată în timpul unei plimbări în oraș. Planul podurilor din oraș și râul Pregel peste care trec acestea este afișat în figura 1.8a. După cum se poate vedea, demonstrând faptul că un astfel de drum este imposibil este asemenea demonstrării faptului că graful din figura 1.8b nu conține nici un traseu Eulerian.

Un *tur* din G este un drum închis care traversează toate muchiile din G cel puțin o dată. Un *tur eulerian* este un tur care traversează toate muchiile exact o dată; altfel spus un tur eulerian este un traseu eulerian închis. Un graf este *eulerian* dacă conține un tur eulerian.

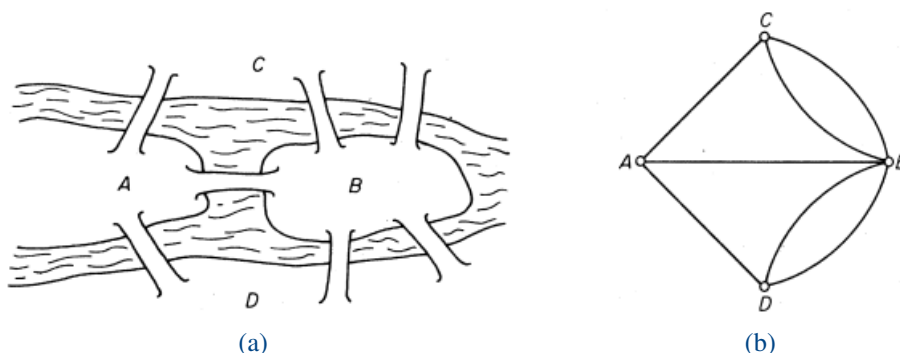


Figura 1.8. Reprezentare a problemei celor 7 poduri din Königsberg: (a) planul podurilor, (b) graful podurilor; figuri preluate din [2]

Teorema 1.2. *Un graf conex este eulerian dacă și numai dacă nu are noduri de grad impar.*

Demonstrație. Fie G un graf eulerian, iar C un tur eulerian al lui G cu originea și destinația într-un nod u . De fiecare dată când un nod v apare ca un nod intern al lui C , cele două muchii incidente cu acesta sunt luate în considerare. Având în vedere că un tur eulerian conține fiecare muchie din G , $d(v)$ este par pentru orice $v \neq u$. Similar, pentru că C începe și se termină cu u , $d(u)$ este de asemenea par. Deci G nu are niciun nod de grad impar. \square

Corolarul 1.2. *Un graf conex are un traseu eulerian dacă și numai dacă are cel mult două noduri de grad impar.*

Demonstrație. Dacă G are un traseu eulerian atunci, ca în demonstrația teoremei 1.2, fiecare nod în afară de originea și destinația traseului au gradul par. \square

3. Problema poștaşului chinez

3.1. Enunțul problemei și variantele acesteia

Este prezentată următoare situație: În timpul slujbei sale, un poștaş preia plicuri de la poștă, le trimite, și apoi se întoarce înapoi de unde a plecat. El trebuie, fără îndoială, să parcurgă fiecare stradă din zona sa cel puțin o dată. Fiind astfel condiționat, poștaşul dorește să aleagă ruta în așa fel în cât el să parcurgă o distanță cât mai mică posibilă.

Această problemă este cunoscută drept „*Problema poștaşului chinez*”, având în vedere că a fost prima dată analizată de matematicianul chinez Kwan Mei-Ko, în anul 1962.

Într-un graf de costuri (fiecare stradă - muchie - are atașată un cost de parcurgere), definim costul turului $v_0v_1 \dots v_nv_0$ ca fiind

$$\sum_{i=1}^n w(e_i),$$

unde e_i este muchia formată din nodurile v_{i-1}, v_i . Clar, problema poștaşului chinez dorește găsirea unui tur de cost minim într-un graf conex de cost, cu valorile specifice acestora pozitive; un astfel de tur va fi considerat *optim*.

De-a lungul timpului, au fost formulate și studiate câteva variante ale problemei poștaşului chinez:

- Problema poștaşului „bătut de vânt” [*en. Windy postman problem*] este asemănătoare originalei, cu diferența în ceea ce privește muchiile: acestea pot avea un cost diferit în funcție de direcția de parcurgere al acestora (ca și cum „ar bate vântul puternic” într-o direcție pe acele străzi);
- Problema poștaşului chinez „hibrid” [*en. Mixed Chinese postman problem*], în care unele muchii din graf au o singură direcție de parcurgere, ca un arc dintr-un graf orientat;
- Problema poștaşului rural [*en. „Rural Postman Problem”*] este o altă alternativă a originalei, unde se rezolvă problema fără a mai trece prin unele muchii;
- Problema poștaşului chinez cu cicluri de k muchii [*en. k -Chinese postman problem*], care urmărește găsirea unor cicluri formate din k muchii, toate pornind de la același nod de plecare, astfel încât fiecare muchie din graf să fie parcursă de cel puțin un ciclu. Soluția problemei determină minimizarea valorii totale a celui mai costisitor ciclu.

Dacă G este graf eulerian, atunci orice tur eulerian din G este un tur optim, pentru că acesta traversează toate muchiile o singură dată. În acest caz, problema poștaşului chinez este ușor de rezolvat [2] prin obținerea unei astfel de soluții folosind diverși algoritmi ce servesc pentru acest lucru.

În cazul în care G nu este graf eulerian, atunci orice tur din acesta, inclusiv turul optim, traversează unele muchii mai mult de o dată. De exemplu, dacă luăm graful din figura 1.9a, atunci $xuywvzwyxuvwzyx$ este un tur optim. Se poate observa că nodurile ux, xy, yw și wv sunt traversate de două ori în turul reprezentat.

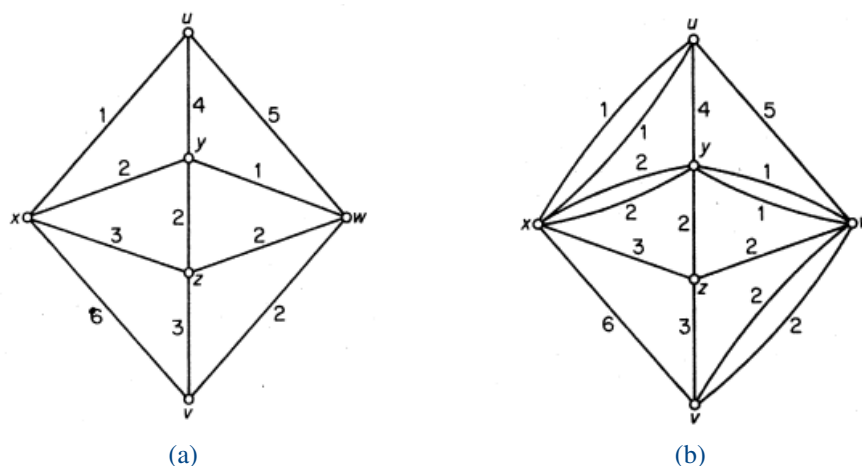


Figura 1.9. Exemplu de graf care nu este eulerian (a) și o soluție a acesteia (b); figură preluată din [2]

Astfel este de preferat, în această situație, să fie introdusă operația de multiplicare a unei muchii. O muchie e este considerată *multiplicată* dacă capetele acesteia formează o nouă muchie de cost $w(e)$. Prin dublicarea muchiilor ux, xy, yw și wv în graful din figura 1.9a, obținem graful din figura 1.9b.

Prin urmare, problema poștaşului chinez poate fi reformulată în felul următor: având un graf de costuri G cu valori pozitive,

- (i) să se caute, prin duplicarea muchiilor, un supergraf eulerian de cost G^* obținut din G astfel încât $\sum_{e \in E(G^*) \setminus E(G)} w(e)$ este cât mai mic posibil;
- (ii) să se găsească un tur eulerian în G^* .

Această cerință este o generalizare a problemei poștașului chinez, atât pentru cazul în care graful este eulerian, cât și în caz contrar. Ea pornește de la observația faptului că un tur a lui G în care muchia e este traversată de $m(e)$ ori corespunde unui tur eulerian din graful obținut din G , prin duplicarea lui e de $m(e) - 1$ ori, și viceversa.

Mai departe, vor fi prezentați doi algoritmi pentru determinarea turului eulerian: *Algoritmul lui Fleury*, sugerat în carte, și *Algoritmul lui Hierholzer*, care a fost folosit în cadrul aplicației. După prezentarea acestora, se va face o comparație între cele două și se va justifica de ce a fost ales al doilea algoritm în implementare. De asemenea va fi discutat și cazul în care graful nu este eulerian.

3.2. Solutii de rezolvare al problemei - cazul grafului eulerian

Aşa cum am menţionat la finalul subsecţiunii anterioare, în cazul în care graful are deja un tur eulerian, este de ajuns să determinăm soluţia problemei prin determinarea turului. Sunt doi algoritmi care ajută la determinarea soluţiei în această situaţie: Algoritmul lui Fleury şi Algoritmul lui Hierholzer.

Algoritmul lui Fleury, sugerat în [2, Pag. 62-63], construiește un tur eulerian prin găsirea unui traseu, având o singură condiție: la fiecare iterație, o muchie nevizitată va fi preluată doar dacă nu este altă alternativă. Pașii algoritmului sunt următorii:

1. Se alege un nod arbitrar v_0 , și se atribuie setului $W_0 = v_0$.
2. Presupunem că traseul $W_i = v_0 v_1 \dots v_i$ a fost ales. Atunci se alege o muchie e_{i+1} formată din nodurile v_i și un nod oarecare v_{i+1} astfel încât:

- (a) muchia să nu existe deja în traseu;

- (b) muchia să fie incidentă cu v_i ;
 - (c) dacă nu este altă alternativă, atunci să nu determine formarea unui subgraf cu muchii nevizitate și imposibil de vizitat prin alte muchii: $G_i = G - \{e_1, e_2, \dots, e_i\}$, unde e_i este muchie formată din nodurile v_{i-1} și v_i .
3. Se repetă pasul 2 până când au fost vizitate toate muchiile.

Teorema de mai jos demonstrează corectitudinea algoritmului:

Teorema 1.3. *Dacă G este graf eulerian, atunci orice traseu din G construit prin Algoritmul lui Fleury este un tur eulerian.*

Demonstrație. Fie G graf eulerian, și $W_n = v_0v_1 \dots v_n$ traseul din G construit prin Algoritmul lui Fleury. Destinația v_n trebuie să aibă gradul nul în G_n . Astfel $v_n = v_0$; cu alte cuvinte, W_n este un traseu închis.

Apoi se presupune, prin reducere la absurd, că W_n nu este un tur eulerian în G , și fie S un set de noduri de grad pozitiv în G_n . Atunci $S \neq \emptyset$ și $v_n \in \bar{S}$, unde $\bar{S} = V \setminus S$. Fie m cel mai mare număr întreg pentru care $v_m \in S$ și $v_{m+1} \in \bar{S}$. Deoarece W_n se termină în \bar{S} , e_{m+1} este singura muchie din $[S, \bar{S}]$ în G_m , și deci este o muchie care taie subgraful G_m (vezi figura 1.10).

Fie e o altă muchie din G_m incident cu v_m . Asta înseamnă că, după pasul 2 al algoritmului, că e trebuie să fie de asemenea o muchie care taie graful G_m și astfel al grafului $G_m[S]$. Dar pentru că $G_m[S] = G_n[S]$, fiecare nod din $G_m[S]$ are gradul par. Totuși, asta înseamnă că $G_m[S]$ nu are nici o muchie care taie graful, o contradicție. \square

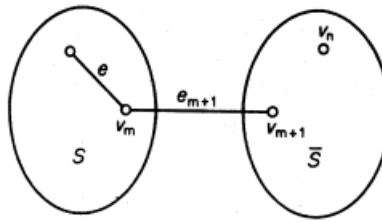


Figura 1.10. Muchia e_{m+1} taie graful $G_m = [S, \bar{S}]$; figură preluată din [2]

Algoritmul lui Hierholzer este o alternativă mai eficientă față de metoda precedentă, originară din lucrarea matematicianului german cu același nume, în anul 1873. El se bazează pe următoarea idee: este posibil să se obțină un circuit pentru un graf eulerian, având muchii distincte, iar originea, respectiv destinația să fie oricare nod din graf. Bineînțeles că acest circuit nu este întotdeauna un tur eulerian, întrucât este posibil ca unele muchii să nu fie vizitate. Totuși, putem forma diferite circuite în graful eulerian fără a ne bloca, deoarece toate nodurile au grad par (teorema 1.2). Astfel algoritmul constă în următorii pași:

1. Se alege un nod oarecare v_k ca originea turului eulerian din graful G ;
2. Se obține inițial un circuit din graful eulerian, cu punctul de plecare v_k , $C_k = v_kv_0v_1 \dots v_nv_k$, unde fiecare pereche de noduri adiacente reprezintă o muchie distinctă; se marchează acestea ca fiind vizitate.
3. Cât timp în setul de noduri parcurse în C_k , există un nod incident cu muchii nevizitate:
 - (a) Se consideră primul nod cu un astfel de criteriu v_l ca o nouă origine pentru un alt circuit C_l ;

- (b) Se obține circuitul $C_l = v_l v_{l_0} v_{l_1} \dots v_{l_m} v_l$ din subgraful lui G format din toate muchiile nevizitate până acum, apoi sunt marcate muchiile membre ca fiind vizitate;
- (c) Se adaugă C_l la C_k în locul nodului v_l : $C_k = v_k v_0 v_1 \dots v_l v_{l_0} v_{l_1} \dots v_{l_m} v_l \dots v_n v_k$.

4. Se returnează C_k , care acum reprezintă turul eulerian al lui G .

Figura 1.11 exemplifică procedeul pentru un graf dat.

Alegerea unuia dintre cei doi algoritmi pentru a fi folosit în aplicație a fost influențată de un singur factor: timpul. Luând în considerare faptul că o localitate are o mulțime de străzi, de la zeci în sate/comune până la mii sau chiar zeci de mii în metropole, o metodă de construcție a turului eulerian necesită parcurgerea tuturor străzilor o singură dată cât mai rapid posibil.

Pentru algoritmul lui Fleury, deși parcurgerea grafului determină complexitatea timp $O(\varepsilon)$, la fiecare iterație se găsesc muchiile ce pot tăia graful. Asta determină o parcurgere adițională a tuturor muchiilor nevizitate, ajungând în cel mai rău caz la o complexitate timp $O(\varepsilon^2)$.

În schimb, algoritmul lui Hierholzer determină iterarea prin toate muchiile din graful eulerian o singură dată, datorită formării ciclurilor și a marcării muchiilor membre ca fiind vizitate. Atât în cel mai bun caz (primul circuit conține toate muchiile din graf), cât și în cel mai rău caz (primul circuit conține o buclă într-un nod), complexitatea timp este liniară: $O(\varepsilon)$. De aceea, a fost aleasă pentru determinarea turului eulerian în cadrul proiectului.

În subcapitolul 2 va fi prezentat pseudocodul, respectiv contextul utilizării algoritmului.

3.3. Soluții de rezolvare al problemei - cazul grafului non-eulerian

În cazul în care în care graful nu este eulerian, problema poștașului chinez se extinde prin găsirea unui supergraf G^* , obținut prin duplicarea muchiilor grafului non-eulerian G , astfel încât $\sum_{e \in E(G^*) \setminus E(G)} w(e)$ este minim, urmat de găsirea turului eulerian în G^* , folosind unul din cei doi algoritmi descriși la subsecțiunea anterioară.

În [2, Pag. 64-65] este sugerat algoritmul lui Edmonds și Johnson [3]. Fiind foarte elaborat, Bondy și Murty au decis să prezinte un caz special care permite o soluție ușoară, pentru cazul în care G are exact două noduri de grad impar.

Se presupune că G are exact două noduri u și v care au grad impar, iar G^* un supergraf eulerian obținut din G prin duplicarea unor muchii, și E^* va reprezenta $E(G^*)$. Clar subgraful $G^*[E^* \setminus E]$ din G^* (indus de muchiile din G^* care nu sunt în G) are de asemenea doar cele două noduri u și v de grad impar. Conform corolarului (1.1), rezultă că u și v fac parte din aceeași componentă din $G^*[E^* \setminus E]$ și astfel sunt conectate de un drum (u,v) , notat P^* . Fără doar și poate:

$$\sum_{e \in E^* \setminus E} w(e) \geq w(P^*) \geq w(P),$$

unde P este drumul (u,v) de cost minim din G . Deci $\sum_{e \in E^* \setminus E} w(e)$ este minim când G^* este obținut din G prin duplicarea fiecărei muchii din drumul (u,v) de cost minim. [2]

Pe baza ideii de mai sus, a fost realizat în aplicație un algoritm de obținere a unui supergraf G^* eulerian din G , care va fi explicat în cadrul discuției implementării aplicației. În cadrul algoritmului este folosită și paradigma Greedy, despre care se va discuta în următoarea secțiune.

4. Paradigma Greedy

Paradigma greedy constă în o serie de algoritmi care se bazează pe rezolvarea problemelor de optim realizând alegeri locale optime (de aici și denumirea *greedy* = „lacom”). Explicațiile următoare sunt preluate din cursul de Proiectare al algoritmilor din cadrul Facultății de Automatică și Calculatoare de la Universitatea Tehnică „Gheorghe Asachi”, predat de domnul Prof.dr.inf. Craus Mitică.

4.1. Caracteristici generale

Considerăm S o mulțime de date și \mathcal{C} un tip de date cu proprietățile:

- obiectele din \mathcal{C} reprezintă submulțimi ale lui S ;
- operațiile includ inserarea $(X \cup \{x\})$ și eliminarea $(X \setminus \{x\})$.

Clasa de probleme la care se aplică paradigma greedy sunt problemele de optim.

Intrare: S ;

Ieșire: O submulțime maximală B din \mathcal{C} care optimizează o funcție f cu valori reale.

Paradigma greedy prezintă următoarele proprietăți:

1. *Alegerea locală*: soluția problemei se obține făcând alegeri locale optime, care pot depinde de alegerile de până atunci, dar nu și cele viitoare. Acestea nu asigură automat că soluția finală este într-adevăr optimul global, mai exact o soluție a problemei. Acest fapt trebuie demonstrat, dar de cele mai multe ori, dovezile nu sunt foarte simple, devenind astfel un inconvenient major al paradigmei.
2. *Substructura optimă*: soluția optimă a problemei conține soluțiile optime ale subproblemelelor.

4.2. Modelul matematic

Fie S o mulțime finită de intrări și \mathcal{C} o colecție de submulțimi ale lui S . Spunem că \mathcal{C} este *accesibilă* dacă satisface *axioma de accesibilitate*:

$$(\forall X \in \mathcal{C}) X \neq \emptyset \Rightarrow (\exists x \in X) X \setminus \{x\} \in \mathcal{C} \quad (1.1)$$

Dacă \mathcal{C} este accesibilă, atunci (S, \mathcal{C}) se numește *sistem extensibil*. O submulțime $X \in \mathcal{C}$ se numește *extensibilă*. Cu alte cuvinte, dacă X este extensibilă, atunci există $y \in S \setminus X$ astfel încât $X \cup \{y\} \in \mathcal{C}$.

Clasa de probleme care pot defini algoritmi greedy este definită de următoarea schemă:

Se consideră date un sistem accesibil (S, \mathcal{C}) și o funcție obiectiv $f : \mathcal{C} \rightarrow \mathbb{R}$.

Problema constă în determinarea unei baze $B \in \mathcal{C}$ care satisface:

$$f(B) = \text{optim}\{f(X) | X \text{ bază în } \mathcal{C}\}$$

În general, prin optim se înțelege maxim sau, în cazul programului, minim. Strategia *greedy* constă în găsirea unui criteriu de selecție a elementelor din S care candidează la formarea bazei optime (care dă optimul pentru funcția obiectiv), numit *alegere greedy* sau *alegere a optimului local*. Formal, optimul local are următoarea definiție [4]:

$$f(X \cup \{x\}) = \text{optim}\{f(X \cup \{y\}) | y \in S \setminus X, X \cup \{y\} \in \mathcal{C}\} \quad (1.2)$$

4.3. Analiza paradigmei greedy

Pseudocodul algoritmului este prezentat în 1.1.

Algoritmul 1.1 Algoritm general greedy

```

1: procedure GREEDY( $S, B$ )
2:    $S_1 \leftarrow S$ 
3:    $B \leftarrow \emptyset$ 
4:   while  $B$  este extensibilă do
5:     Alege un optim local  $x$  din  $S_1$  conform cu (1.2)
6:      $S_1 \leftarrow S_1 \setminus \{x\}$ 
7:      $B \leftarrow B \cup \{x\}$ 
8: end
    
```

Din păcate, numai condiția de accesibilitate nu asigură întotdeauna existența unui criteriu de alegere locală care să conducă la determinarea unei baze optime.

Pentru anumite probleme, se pot proiecta algoritmi greedy care nu furnizează soluția optimă, ci o bază pentru care funcția obiectiv poate avea valori apropiate de cea optimă.

Presupunem că pasul de alegere greedy selectează elemente x în timpul $O(k^p)$ unde $k = \#S_1$ și că testarea condiției „ B este extensibilă” se face în timpul $O(l^q)$ cu $l = \#B$; $k + l \leq n$. De asemenea se consideră costul operațiilor $S_1 \setminus \{x\}$ și $B \cup \{x\}$ egal cu $O(1)$.

Deoarece pasul de alegere este executat de n ori rezultă că metoda are complexitatea timp

$$T(n) = O(n^p + 1^q) + \dots + O(1^p + n^q) = O(1^p + \dots + n^p + 1^q + \dots + n^q) = O(n^{p+1} + n^{q+1}) = O(n^{\max(p+1, q+1)})$$

Preprocesarea intrărilor poate conduce la o reducere considerabilă a complexității metodei. Un caz particular al algoritmului 1.1 va fi prezentat în capitolul 2, când se va discuta despre proiectarea aplicației.

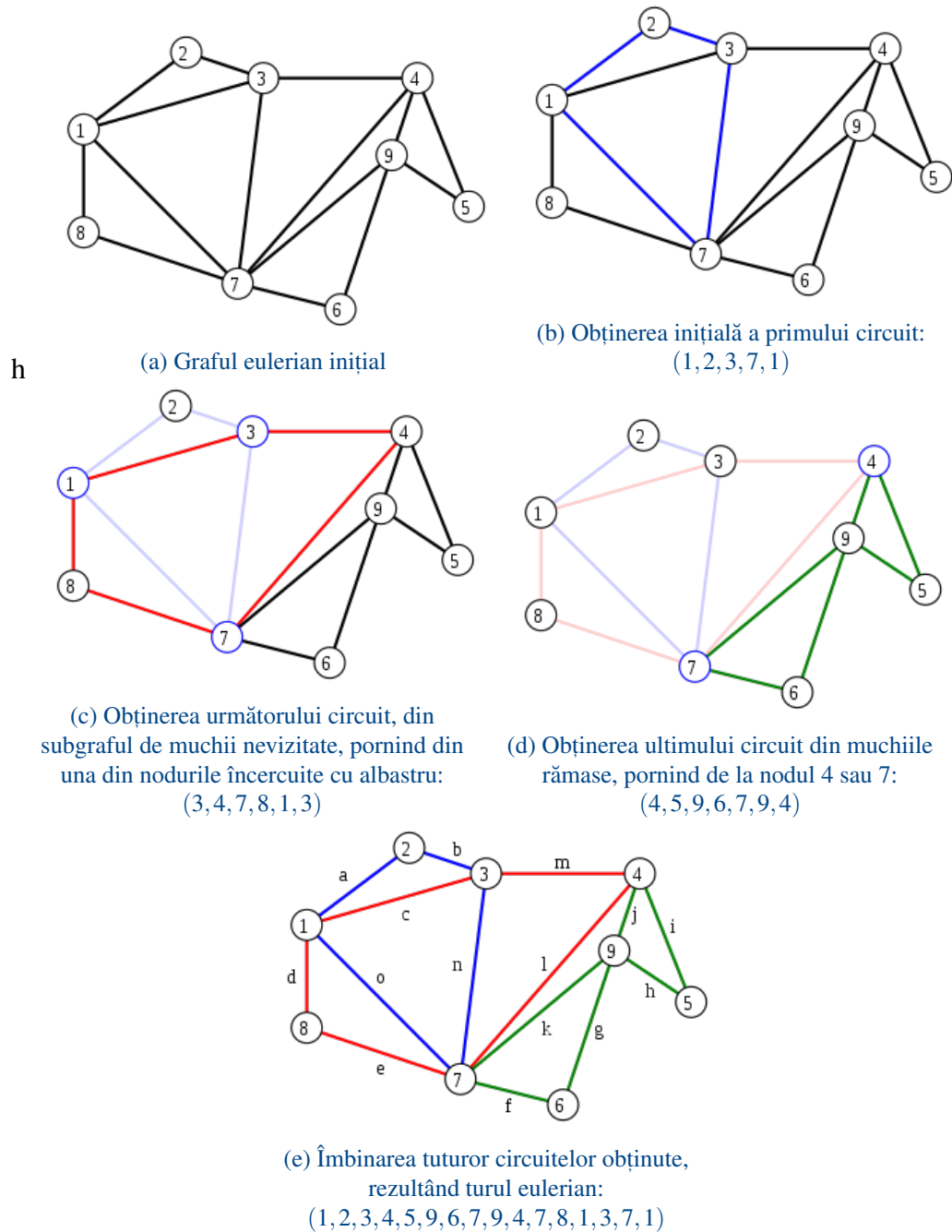


Figura 1.11. Procedeeul de determinare al turului eulerian folosind algoritmul lui Hierholzer; sursa: https://second.wiki/wiki/algorithmus_von_hierholzer

Capitolul 2. Proiectarea aplicației

1. Componente hardware

Aplicația a fost testată pe un singur dispozitiv, dar aceasta poate fi rulată pe orice sistem de operare, atâta timp cât sunt instalate toate framework-urile necesare, care vor fi discutate în secțiunea 2.3. Specificațiile dispozitivului folosit pentru testare sunt prezentate mai jos.

Laptop ASUS VivoBook K513EA

- Sistem de operare Windows 10 pe 64 biți
- Procesor Intel Core i7-1165G7 @ 2.80GHz, 4 core-uri
- Memorie 8GB RAM
- Placă video Intel Iris Xe Graphics
- Samsung SSD 860 EVO 500GB și SSD Micron_2210_MTFDHBA512QFD 500GB

Singurele componente relevante care vor influența performanțele aplicației sunt fără îndoială procesorul și memoria internă. Acestea au fost capabile să ruleze programul fără probleme, plus că acesta este programat secvențial. Mai jos sunt precizate și performanțele procesorului (CPU-ului) pentru un astfel de caz.

CPU Information	
Name	Intel Core i7-1165G7
Topology	1 Processor, 4 Cores, 8 Threads
Base Frequency	2.80 GHz
Maximum Frequency	4694 MHz
Package	Socket 1449 FCBGA
Codename	Tiger Lake-U
L1 Instruction Cache	32.0 KB x 4
L1 Data Cache	48.0 KB x 4
L2 Cache	1.25 MB x 4
L3 Cache	12.0 MB x 1
Memory Information	
Memory	7.70 GB DDR4 SDRAM 1595 MHz
Channels	1
Single-Core Performance	
Single-Core Score	1243

Figura 2.1. Rezultatul unui benchmark realizat pentru CPU-ul laptop-ului, pe modul single core

2. Componente software

2.1. OpenStreetMap și formatul de fișier .osm

OpenStreetMap este un website (<https://www.openstreetmap.org>) care oferă o hartă a tuturor locurilor de pe planetă, folosind informații descentralizate de la diverși utilizatori globali (imagini aeriene, dispozitive GPS, hărți de turism, etc.). După cum sugerează și numele, datele oferite în hărți sunt gratuite și pot fi folosite în orice scop, atâta timp cât sunt acreditate contribuitorilor website-ului. Serviciile acestora sunt de asemenea legale, operate formal de OpenStreetMap Foundation (OSMF) (https://wiki.osmfoundation.org/wiki/Main_Page).

Sunt câteva variante de preluare al acestor date:

- Prin marcarea zonei de pe hartă dorite pentru preluarea datelor acestor într-un fișier XML, în interfața grafică a website-ului (figura 2.2). Aceasta are o limită totuși, permițând maxim 50.000 elemente în zona marcată. De aceea este utilă pentru a prelua doar datele unei localități, și prin urmare este alegera făcută pentru preluarea datelor în proiect.
- *Overpass API* este un API read-only care are scopul asemănător variantei anterioare, dar pentru un număr mult mai mare de elemente (până la 10 milioane de elemente în câteva minute). Această opțiune necesită în schimb conexiune la internet și se bazează pe rata de transfer al datelor din server-ul website-ului.
- *Planet OSM* - reprezintă copia bazei de date al OpenStreetMap, ce poate fi achiziționată în mod gratuit. Avantajul acestei opțiuni este clar setul complet de date ce poate fi folosit în aplicații asemănătoare Google Maps. Dezavantajul este spațiul necesar stocării bazei de date: la momentul scrierii acestei documentații, fișierul XML conține 118GB date, iar descărcarea se face în general prin torrent.

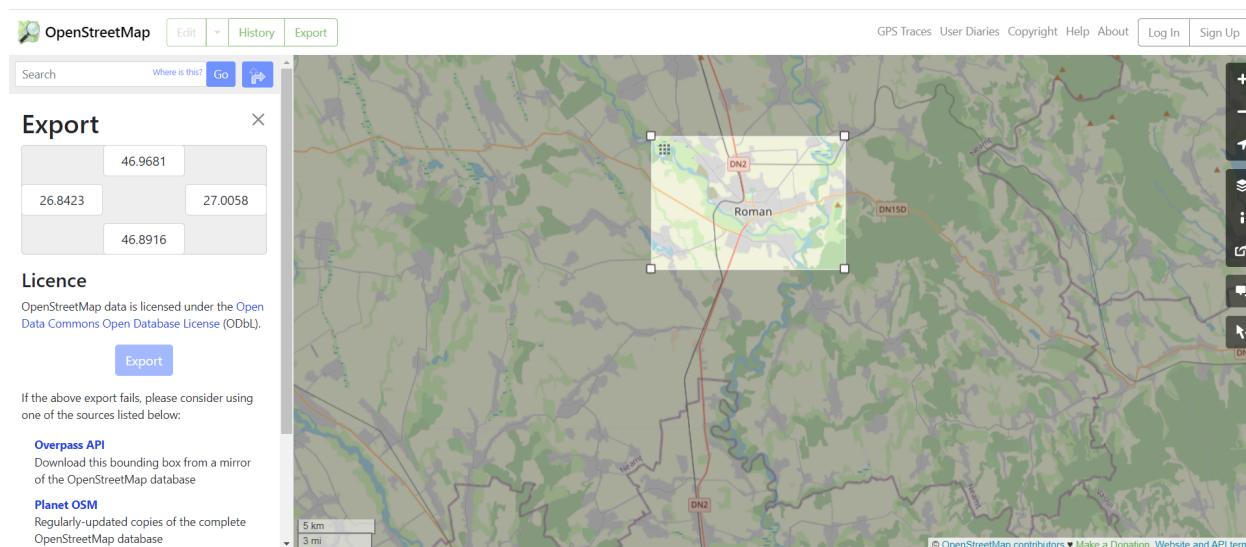


Figura 2.2. Exemplu de captură al unei zone din hartă care va fi transformată în fișier XML

Indiferent de care metodă este folosită, la final se va obține un fișier XML, în *format .osm*. XML a fost ales pentru că furnizează date ușor de citit de utilizatori, având o structură clară și independentă de mașină. Singurul impediment care îl poate avea un astfel de fișier este parsarea acestuia, care poate lua ceva timp și resurse de memorie, dacă se folosesc toate capacitățile formatului. Din fericire, este folosit versiunea de bază al XML, fără namespace-uri sau DTD [en.

Document Type Definition], ceea ce permite o prelucrare mai rapidă și eficientă al datelor. Un exemplu de conținut al unui fișier OSM XML este dat mai jos. Punctele de suspensie din cod reprezintă conținut ce poate fi adăugat, dar este prea lung ca să fie inclusă în exemplu.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <osm version="0.6" generator="CGImap 0.0.2">
3    <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900"
      ↪ maxlon="12.2524800"/>
4    <node id="298884269" lat="54.0901746" lon="12.2482632"
      ↪ user="SvenHRO" uid="46882" visible="true" version="1"
      ↪ changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
5    <node id="261728686" lat="54.0906309" lon="12.2441924"
      ↪ user="PikoWinter" uid="36744" visible="true" version="1"
      ↪ changeset="323878" timestamp="2008-05-03T13:39:23Z"/>
6    <node id="1831881213" version="1" changeset="12370172"
      ↪ lat="54.0900666" lon="12.2539381" user="lafkor" uid="75625"
      ↪ visible="true" timestamp="2012-07-20T09:43:19Z">
7      <tag k="name" v="Neu Broderstorf"/>
8      <tag k="traffic_sign" v="city_limit"/>
9    </node>
10   ...
11   <node id="298884272" lat="54.0901447" lon="12.2516513"
      ↪ user="SvenHRO" uid="46882" visible="true" version="1"
      ↪ changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
12   <way id="26659127" user="Masch" uid="55988" visible="true"
      ↪ version="5" changeset="4142606"
      ↪ timestamp="2010-03-16T11:47:08Z">
13     <nd ref="292403538"/>
14     <nd ref="298884289"/>
15     ...
16     <nd ref="261728686"/>
17     <tag k="highway" v="unclassified"/>
18     <tag k="name" v="Pastower Straße"/>
19   </way>
20   <relation id="56688" user="kmvar" uid="56190" visible="true"
      ↪ version="28" changeset="6947637"
      ↪ timestamp="2011-01-12T14:23:49Z">
21     <member type="node" ref="294942404" role=""/>
22     ...
23     <member type="node" ref="364933006" role=""/>
24     <member type="way" ref="4579143" role=""/>
25     ...
26     <member type="node" ref="249673494" role=""/>
27     <tag k="name" v="Küstenbus Linie 123"/>
28     <tag k="network" v="VWV"/>
29     <tag k="operator" v="Regionalverkehr Küste"/>
30     <tag k="ref" v="123"/>
31     <tag k="route" v="bus"/>
32     <tag k="type" v="route"/>
33   </relation>
34   ...
35 </osm>

```

Listing 2.1. Exemplu de conținut într-un fișier OSM XML

Din cod reies trei elemente care se regăsesc cel mai des în acest tip de fișier și sunt considerate „primitivele” datelor: nodul [*en. node*], calea [*en. way*] și relația [*en. relation*]. Explicațiile aferente sunt preluate din documentația formatului OSM (https://wiki.openstreetmap.org/wiki/OSM_XML#JOSM_file_format).

Nodul este elementul de bază în modelul de date din OpenStreetMap. El constă într-un singur punct în spațiu definit prin latitudine, longitudine și id-ul acestuia. Pot fi folosite pentru a defini puncte centrale ale unor locații (benzinării, restaurante, intrări în clădiri, etc.), dar de cele mai multe ori construiesc forma unei *căi*.

Exemplu de nod reiese din liniile 185-188 din codul 2.1.

Calea reprezintă un alt element fundamental dintr-o hartă, reprezentată printr-o singură linie care leagă două noduri. Ea întruchipează trăsăturile liniare de la suprafață (drumuri, zid, sau râu). În fișier, calea este construită printr-o listă ordonată de noduri care au cel puțin o etichetă și sunt incluse într-o *relație*. Lista poate conține minim două noduri și maxim 2.000; totuși pot exista căi inserate greșit, cu mai puțin de 2 noduri. Există trei tipuri de căi ce pot fi reprezentate:

- *Calea deschisă* este asemenea unui drum într-un graf: primul și ultimul nod din listă nu sunt identice. Exemple de astfel de reprezentări liniare sunt majoritatea drumurilor rutiere, a râurilor și a căilor ferate. În baza de date, ele au întotdeauna o direcție de deplasare, chiar și dacă aceasta poate fi parcursă din ambele sensuri (cum sunt drumurile rutiere fără sens unic).
- *Calea închisă* reprezintă un circuit într-un graf: nodul de origine și cel de destinație sunt la fel. Ea poate fi interpretată fie ca un perimetru, prin intermediul tag-ului `highway=*` (pentru sens giratoriu sau promenade circulare) sau tag-ul `barrier=*` (pentru garduri vii, ziduri), fie ca o suprafață pentru a reprezenta prezența unei clădiri, prin tag-ul `area=yes`, sau chiar amândouă, în funcție de etichetele sale și cele dintr-o relație.
- *Aria* este cazul particular al unei căi închise, atunci când are eticheta `area=yes`. Ea reprezintă suprafața unui teritoriu (tag-ul `leisure=park` pentru a sugera perimetrul unui parc), sau pentru a defini prezența unei clădiri (tag-ul `amenity=school` pentru a sugera o școală). Se pot combina mai multe etichete pentru sugera, de exemplu, promenada dintr-un oraș: `highway=pedestrian + area=yes`.

Exemplu de cale este redată pe liniile 191-198 din codul 2.1.

Relația este ultimul element fundamental dintr-un fișier OSM XML, și constă într-un element care are cel puțin eticheta `type=*` și un grup de membrii constituit printr-o listă ordonată de unul sau mai multe noduri, căi și/sau alte relații. Scopul ei este de a defini relații logice sau geografice între diferite obiecte (de exemplu, un lac și insula sa, câteva drumuri ce se află pe o rută de autobuz). Un membru al relației poate avea opțional un rol care descrie cum contribuie acesta în ea.

Este obligatoriu într-o relație să existe măcar un nod sau muchie, iar în caz contrar să fie o relație ca membru cu aceleași reguli sau nu, altfel să fie copilul altei relații cu aceeași regulă. Dacă nu se respectă, o relație va apărea invizibilă pe hartă, căci nu este atașat de nimic concret de pe aceasta.

Liniile 199-212 din codul 2.1 exemplifică o relație.

2.2. Limbajul de programare Python

Pentru implementarea aplicației, a fost folosit limbajul de programare *Python*. Acesta este un limbaj de nivel înalt, interpretat, de uz general, scris în mod dinamic și „garbage-collected” (formă de gestiune a memoriei care eliberează locațiile acesteia alocate în program, dar care nu mai sunt referite, astfel de zone denumite „garbage”). El este des utilizat în domeniul IT datorită suportului mai multor paradigme de programare, atât programarea orientată pe obiect, cât și cea funcțională (unde programul este construit prin folosirea subprogramelor/funcțiilor). Versiunea utilizată de-a lungul proiectului este Python 3.9.11.

Limbajul de programare a fost ales și datorită familiarității cu acesta, fiind limbajul principal utilizat în realizarea proiectelor de-a lungul facultății [5][6]. În plus, tema proiectului este inspirată din una din aplicațiile realizate în timpul practicii de vară: problema comisului voiajor folosind algoritmi genetici. În cadrul acestuia, a fost folosit Google Maps API pentru a putea prelua informațiile în ceea ce privește locațiile prin care va trece drumul (folosind Geocoding API), cât și distanțele între fiecare locație în parte. Toate acestea au fost de asemenea realizate cu ajutorul limbajului de programare Python.

Mai departe sunt prezentate principalele module Python utilizate în cadrul aplicației.

2.3. Module Python utilizate în cadrul aplicației

2.3.1. Modulul *numpy*

Modulul *numpy* este destinat realizării calculelor științifice în cadrul limbajului Python. Aceasta aduce puterea de calcul a limbajelor de programare cum ar fi C și Fortran, în tandem cu simplitatea și învățarea ușoară a limbajului Python, contribuind astfel în diverse domenii științifice: de la procesarea de imagini și semnale, până chiar la calculul cuantic, arhitectură și chimie. Este unul din librăriile folosite în obținerea primei imagini a unei găuri negre, în 2019 (<https://numpy.org/case-studies/blackhole-image/>).

În cadrul aplicației, modulul este folosit în special pentru păstrarea și interogarea datelor despre costurile străzilor dintr-un oraș, în cazul implementării algoritmilor, unde viteza de accesare este cheia către eficiența acestora.

2.3.2. Modulul *threading*

Modulul *threading* este des folosit pentru procesarea paralelă a datelor, oferind rapiditate programelor realizate în Python. Principala componentă a modulului este *firul de execuție* [*en. thread*], fiind cea mai mică unitate de secvență de instrucțiuni dintr-un program, care poate fi programată spre execuție de sistemul de operare. De aceea este mai rapidă și ocupă memorie mai puțină decât alte moduri de prelucrare paralelă (procesele).

Majoritatea limbajelor de programare de la ora actuală suportă paradigma *multithreading*, care constă în folosirea paralelă a mai multor fire de execuție pentru a realiza o un set de instrucțiuni costisitor din punct de vedere al timpului (dacă era rulat secvențial). În schimb, interpretorul Python folosește mecanismul *GIL* [*en. Global Interpreter Lock*], care asigură execuția unui thread câte una o dată. Acest lucru protejează obiectele (mai ales tipurile iterabile: dict, set, list, set) de „race condition”, adică situația în care nu este controlată ordinea de rulare a firelor de execuție, având consecință suprascrierea nedorită a datelor sau rularea în ordine incorectă a instrucțiunilor. De asemenea, ajută interpretorul să aibă la dispoziție mai multe fire de execuție, în detrimentul paralelismului necesar în cadrul sistemelor de calcul multi-procesor.

De-a lungul timpului, s-a încercat crearea unui interpretor care blochează datele împărtășite la un nivel de programare mult mai jos, dar cauza eșecului realizării unui astfel de soluție constă în performanța suferită dacă se utilizează un singur procesor, cât și faptul că implementarea va deveni mai complicată și prin urmare mai dificil de întreținut.

În ciuda lipsei paralelismului în firele de execuție, modulul *threading* este des utilizat în

aplicațiile cu interfață utilizator (*GUI* [en. *Graphical User Interface*]), pentru a prelucra în fundal diverse date fără a bloca firul de execuție pe care rulează GUI-ul.

2.3.3. Modulul `concurrent.futures`

Pentru a realiza totuși operații paralele, *modulul* `concurrent.futures` oferă o interfață de nivel înalt pentru a executa asincron programe și/sau instrucțiuni dintr-un program. Execuția paralelă poate fi implementată fie prin thread-uri (dar nu este fiabilă, vezi subsecțiunea 2.3.2), fie prin *procese* separate.

Un *proces* reprezintă elementul principal al unui program dintr-un sistem de calcul, care este executată prin una sau mai multe fire de execuție. Astfel un proces este alcătuit din una sau mai multe thread-uri, ceea ce determină un timp de execuție mai mare și ocupă mai multă memorie decât un simplu fir de execuție.

Avantajul semnificativ față thread-uri este că sistemul de operare este cel care se ocupă cu apelul unui proces, contribuind la gestiunea sănătoasă al acesteia. În plus, blocarea unui proces din diferite motive nu va determina același comportament și pentru celelalte procese în execuție, față de firele de execuție, unde blocarea unui fir la nivelul utilizatorului determină blocarea celorlalte de același tip. În schimb, procesele comunică mai greu între ele, și nu împărtășesc o memorie comună - până la urmă în fiecare se execută câte un singur program care are propriul spațiu de memorie asignat de sistemul de operare.

Paradigma multi-processing este posibilă în Python, având în vedere această metodă se folosește de *modulul* `multiprocessing`, care permite ocolirea mecanismului GIL din interpretor. Acest fapt este de asemenea aplicat în modulul `concurrent.futures` prin *clasa* `ProcessPoolExecutor`, care folosește un set (sau termenul en. *pool*) de procese care sunt apelate paralel, în funcție de câte sunt asignate în acesta (argumentul `max_workers` din constructorul clasei). De precizat este că pe sistemele de operare Windows, numărul maxim de procese asignate este de 61, iar în cazul asignării unui număr mai mare decât cel stabilit va arunca excepția `ValueError`. Oricum lăsarea implicită a parametrului `max_workers` va determina numărul maxim posibil în funcție de sistemul de operare pe care rulează.

2.3.4. Modulul `xml`

Pentru accesarea și manipularea fișierelor în format `.xml`, *modulul* `xml` din Python oferă toate instrumentele necesare implementării acestor operații.

În cadrul aplicației, este folosit pentru a vizualiza și extrage informațiile străzilor și nodurilor din orașe, stocate în fișierele în format `.osm` (vezi secțiunea 2.1 pentru mai multe detalii despre acest format). Se folosește în special biblioteca `xml.etree.ElementTree` pentru parsarea și accesul datelor din fișierele aferente.

2.3.5. Modulul `networkx`

Modulul `networkx` ajută la reprezentarea, studierea, și manipularea facilă a rețelelor complexe, inclusiv grafuri.

Biblioteca are ca scop în proiectul de față reprezentarea nodurilor și muchiilor specifice grafului unei zone rezidențiale, iar mai apoi schema să fie reprezentată grafic, prin module specializate în acest scop.

2.3.6. Modulul `matplotlib`

Fără îndoială, *modulul* `matplotlib` este una dintre cele mai utilizate biblioteci de reprezentare grafică în cadrul limbajului Python. Aceasta are incluse submodule pentru compatibilitate backend în diverse tipuri de interfețe GUI (TKinter, Qt), cu scopul desenării pe „pânză” [en. *canvas*] a graficelor dorite, în interfața cu utilizatorul.

Prin acest scop, modulul a contribuit la evidențierea circuitului eulerian al dronei prin străzile zonei rezidențiale detectate, obținut prin crearea grafului corespunzător folosind modulul `networkx`, împreună cu soluția circuitului pentru problema poștașului chinez.

2.3.7. Modulul PyQt5

Qt este un framework cross-platform pentru realizarea interfețelor cu utilizatorul performante și atractive. Are de asemenea instrumente pentru crearea GUI-urilor într-un mod mult mai ușor: de la simplul Qt Designer, până la programele mai recente Qt Creator și Design Studio. Biblioteca este inclusă și în Python în 2 versiuni: PyQt5 (versiunea 5.2) și PyQt6 (cea mai recentă fiind versiunea 6.3).

În cadrul aplicației este folosit *modulul PyQt5* pentru construirea GUI-ului specific încărcării și vizualizării grafurilor orașului dorit, împreună cu afișarea informațiilor esențiale în redarea animației circuitului eulerian, alături de framework-urile *threading* și *matplotlib*.

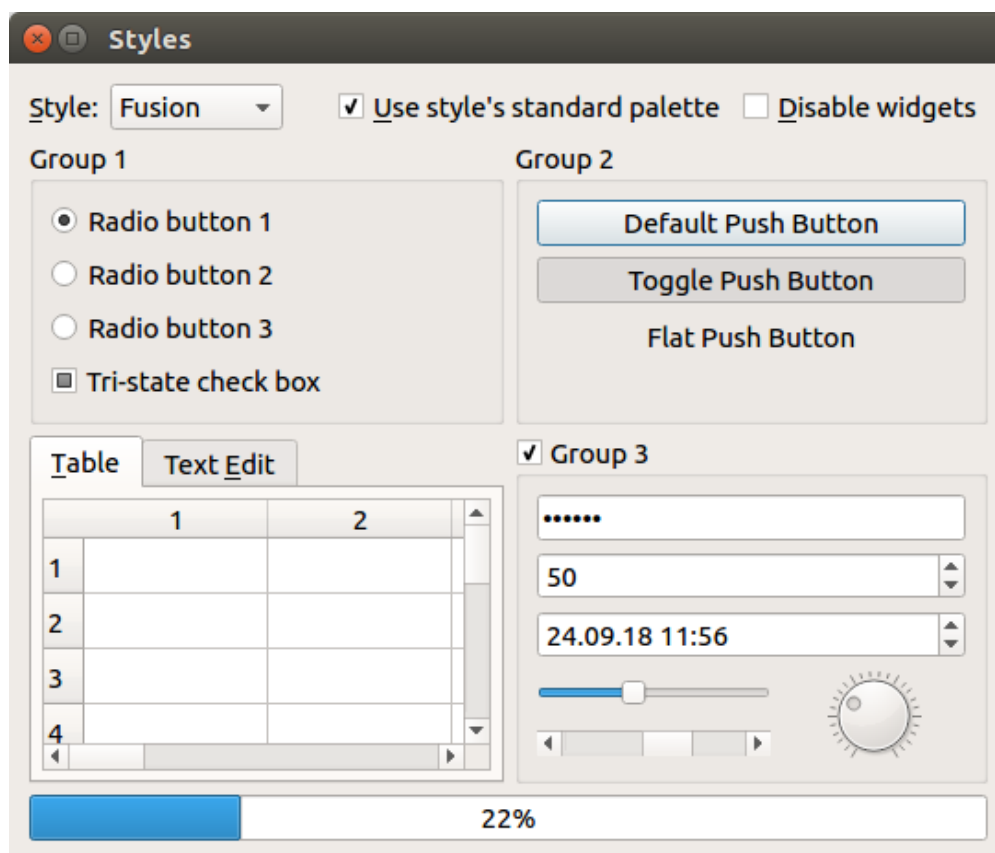


Figura 2.3. Exemplu de GUI realizată în PyQt5; imagine preluată din <https://build-system.fman.io/pyqt5-tutorial>

2.4. Mediul de dezvoltare PyCharm

PyCharm este un IDE [en. *Integrated Development Environment*] realizat de compania JetBrains, cu scopul implementării asistate a programelor, cu suport principal în Python, JavaScript, TypeScript, CSS și multe altele. Platforma oferă completare de cod în mod inteligent, detectarea automată a erorilor din cod, plus rezolvări imediate ale acestora, precum și navigare rapidă prin program cu ajutorul căutărilor inteligente din orice clasă, fișier, simboluri, chiar și acțiuni din cadrul mediului de dezvoltare.

Există două versiuni ale acestuia: *PyCharm Community*, care oferă tot ce este necesar pentru dezvoltarea „pur” Python, plus că este open-source, și *PyCharm Pro*, unde include, pe lângă funcționalitățile ediției Community, și framework-uri web (Node.js, Angular, React, Flask),

dezvoltare remote (Jupyter Notebook), cât și biblioteci pentru dezvoltarea bazelor de date în mediul Python (SQL), dar această versiune este comercială și oferă doar 30 zile de încercare gratuită.

Pentru nevoile curente, a fost ales IDE-ul în varianta Community pentru implementarea proiectului de licență, mai ales că a existat experiență dobândită cu acesta de-a lungul anilor de facultate.

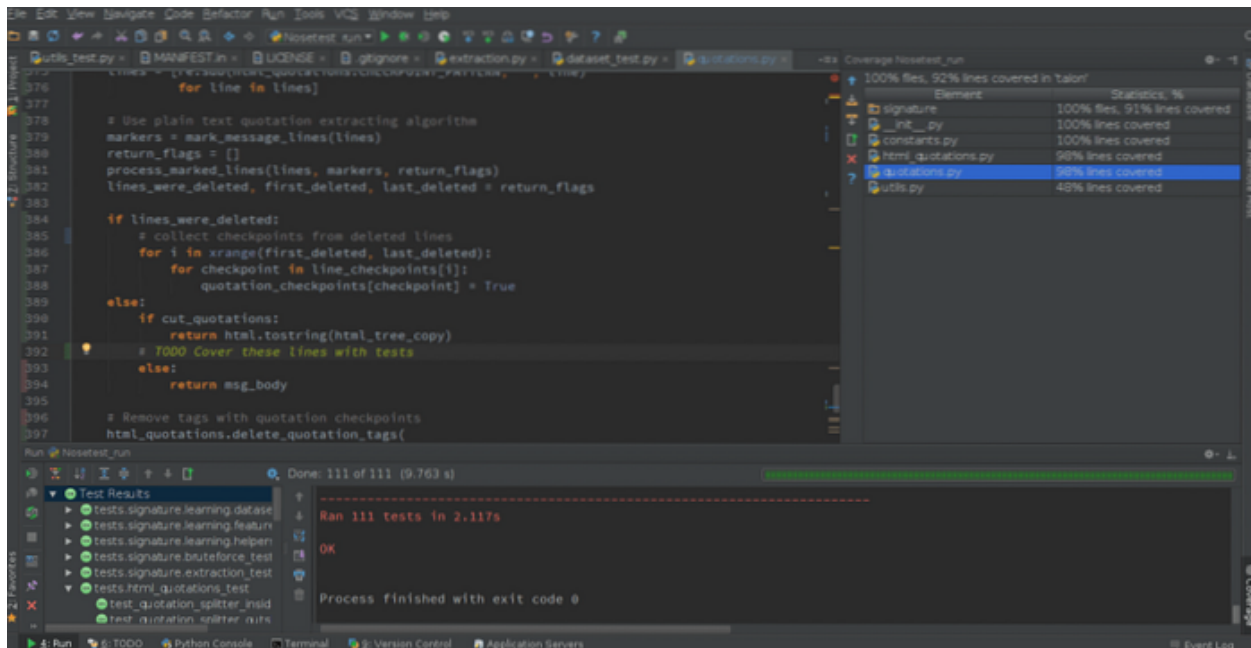


Figura 2.4. Interfața IDE-ului PyCharm; imagine preluată din https://ro.linuxteaching.com/article/pycharm_community

2.5. Algoritmi folosiți în cadrul aplicației

2.5.1. Algoritmul Floyd-Warshall

Algoritmul Floyd-Warshall constă în găsirea celor mai scurte drumuri între fiecare pereche de noduri dintr-un graf ponderat (fiecare arc/muchie are un cost de parcurgere). Deși metoda nu returnează implicit lista muchiilor parcurse pentru fiecare drum minim, ea poate să le reconstruiască printr-o matrice de succesori.

Fie G un graf neorientat ponderat determinat de perechea (V, E) , unde V este mulțimea nodurilor din G , în timp ce E este mulțimea muchiilor din G determinat de o pereche de noduri (v_i, v_{i+1}) , notat cu e_i . Se consideră $M = [m_{ij}]$ matricea de distanțelor minime între fiecare 2 noduri v_i și v_j din graf, de dimensiune $v \times v$, și $S = [s_{ij}]$ matricea de succesori în drumul dintre fiecare 2 noduri din graf, de aceeași dimensiune.

Inițial,

$$m_{ij} = \begin{cases} d((v_i, v_j)), & \text{dacă } (v_i, v_j) \in E \\ 0, & \text{dacă } v_i = v_j \\ \infty, & \text{altfel} \end{cases},$$

unde $d(e_i)$ este costul muchiei, și

$$s_{ij} = \begin{cases} v_j, & \text{dacă } (v_i, v_j) \in E \text{ sau } v_i = v_j \\ \text{null}, & \text{altfel} \end{cases}.$$

Odată inițializate matricele, algoritmul în sine poate începe:

1. Pentru fiecare nod v_k din graf

- (a) Se consideră v_k nod de tranzit în drumul dintre diferite 2 noduri;
- (b) Pentru fiecare pereche de noduri posibilă (v_i, v_j) :
 - i. Dacă m_{ij} este mai mare decât $m_{ik} + m_{kj}$, atunci se schimbă m_{ij} cu valoarea cea mai mică dintre ele și s_{ij} cu noul succesor s_{ik}

Algoritmul în varianta pseudocod este afișat mai jos.

Algoritmul 2.1 Algoritmul Floyd-Warshall

```

1: procedure ALG_FLOYD_WARSHALL( $V, E$ )
2:   Inițializare matrici  $M$  și  $S$ 
3:   for fiecare nod  $v_k$  din  $V$  do
4:     for fiecare  $v_i$  nod origine din  $V$  do
5:       for fiecare  $v_j$  nod destinație din  $V$  do
6:         if  $m_{ij} > m_{ik} + m_{kj}$  then
7:            $m_{ij} = m_{ik} + m_{kj}$ 
8:            $s_{ij} = s_{ik}$ 
9: end
    
```

Rezultatul va consta în matricile M și S , care vor conține costul minim între fiecare două noduri din graf, respectiv nodurile succesoare pentru construirea drumurilor minime.

Construirea drumului de cost minim între două noduri este evidențiat în algoritmul următor. Se notează $v_i, v_d \in V$ nodurile origine, respectiv destinație ale drumului minim.

Algoritmul 2.2 Algoritmul de construcție al unui drum minim

```

1: procedure BUILD_ROAD( $v_o, v_d, S$ )
2:   Inițializare drum  $path$  cu listă goală
3:   if  $s_{od} \neq null$  then
4:     Adăugare  $v_o$  în  $path$ 
5:     Inițializare nod succesor inițial  $next_v$  cu  $v_o$ 
6:     while  $next_v \neq v_d$  do
7:        $next_v = s_{next_v v_o}$ 
8:       Adăugare nod  $next_v$  în coada drumului  $path$ 
9:   return  $path$ 
10: end
    
```

2.5.2. Algoritmul lui Dijkstra

Acest algoritm este un caz particular al metodei Floyd-Warshall, descrisă în subsecțiunea anterioară (2.5.1), care determină, pornind de la un *singur* nod v_s dintr-un graf $G = (V, E)$, drumurile de cost minim între celelalte noduri din V . Rezultatul este sugerat prin doi vectori, care conțin costul, respectiv precedenții nodurilor, ale drumurilor minime între v_s și celelalte intersecții.

Fie $R = [r_i]$ vectorul de distanțe minime între nodul v_s și celelalte noduri $v_i, v_i \in V$, și $P = [p_i]$ vectorul de predecesori în drumurile minime menționate. Față de algoritmul Floyd-Warshall, este inclus un vector $visit = [visit_i]$ ce marchează care dintre noduri sunt vizitate în timpul iterării prin acestea. Se consideră $d(e_{kl})$ costul muchiei dintre nodurile v_k și $v_l, e_{kl} \in E$.

La început:

$$r_i = \begin{cases} 0, & \text{dacă } v_i = v_s \\ \infty, & \text{altfel} \end{cases}, p_i = null \text{ și } visit_i = fals, \forall v_i \in V.$$

Atât pseudocodul algoritmului, cât și cel de reconstruire al drumului minim folosind matricea de precedenți, sunt evidențiate mai jos.

Algoritmul 2.3 Algoritmul lui Dijkstra

```

1: procedure ALG_DIJKSTRA( $V, E, v_s$ )
2:   Inițializare vectori  $R, P$  și  $visit$ 
3:   while  $\exists v_k \in V$  pentru care  $visit_k = fals$  do
4:     Selectare nod  $v_k$  astfel încât  $visit_k = fals$  și  $r_k = \min(R)$ 
5:      $visit_k = adevărat$ 
6:     for fiecare  $v_l$  din  $V$  adiacent cu  $v_k$  do
7:       if  $visit_l = fals$  și  $r_l > r_k + d(e_{kl})$  then
8:          $r_l = r_k + d(e_{kl})$ 
9:          $p_l = v_k$ 
10:    Returnare  $R$  și  $P$ 
11: end

```

Se notează cu v_d nodul destinație din drumul cu originea în v_s , reprezentat de vectorul P_s .

Algoritmul 2.4 Determinare drum minim folosind vectorul P_s din algoritmul lui Dijkstra

```

1: procedure BUILD_ROAD_DIJKSTRA( $v_d, v_s, P_s$ )
2:   if  $v_d = v_s$  then
3:     Returnează lista  $[v_s]$ 
4:   else
5:      $prev_d = P_s(v_d)$ 
6:     Returnează  $build\_road\_dijkstra(prev_d, v_s, P_s) \cup \{v_d\}$ 
7: end

```

2.5.3. Studiu de caz: Floyd-Warshall versus Dijkstra

Atât algoritmul Floyd-Warshall, cât și metoda lui Dijkstra se află în același domeniu de rezolvare al problemelor, celor de determinare a drumurilor minime între nodurile dintr-un graf. Bineînțeles că al doilea algoritm rezolvă această situație pentru un singur nod origine, în timp ce primul determină pentru toate perechile posibile de noduri. Totul se rezumă, desigur, la aplicabilitatea lor în viața reală, și în consecință în cadrul proiectului realizat.

Decizia de alegere a unuia dintre cele două variante a depins de dilema următoare: dacă numărul de intersecții(noduri) dintr-o zonă rezidențială este mai mare sau nu decât numărul de străzi(muchii). De cele mai multe ori, numărul de muchii poate depăși cu mult numărul de noduri doar dacă graful este complet sau aproape complet (când există câte o muchie către fiecare pereche de noduri posibilă). În cazul acesta, algoritmul Floyd-Warshall este cel mai fiabil, iar în cazul contrar, cel mai bine este să se folosească algoritmul Dijkstra, datorită complexității timp mult mai mici: în cel mai rău caz, complexitatea este de $O(V^2)$, față de Floyd-Warshall, de $O(V^3)$.

În plus, problema de soluționat în cadrul proiectului este cea a poștaşului chinez, care se rezolvă prin căutarea perechilor de noduri de grad impar pentru care distanța dintre fiecare este minimă (vezi subsecțiunea 2.5.4). Astfel este mult mai rentabil să se aplice algoritmul lui Dijkstra pentru fiecare nod de grad impar pentru a facilita determinarea soluției problemei. De asemenea, procedeul de aplicare al metodei doar pe anumite noduri poate fi accelerată prin paradigma multi-processing, unde în fiecare proces se va asigura câte un apel al algoritmului incluzând ca parametru un nod de grad impar.

Este bine de menționat faptul că în majoritatea cazurilor, zonele rezidențiale dintr-un oraș rar formează grafuri aproape complete, și prin urmare algoritmul Floyd-Warshall mai mult încetinește

rezolvarea situației poștaşului chinez.

De aceea, considerând argumentele pro și contra ale celor doi algoritmi, s-a luat decizia, în contextul aplicației, că Dijkstra este cel mai potrivit, îmbunătățindu-l prin procesarea paralelă.

2.5.4. Paradigma Greedy

Partea teoretică al paradigmei a fost menționată în capitolul 1, secțiunea 4. Tot atunci a fost precizată folosirea ei într-un caz particular (4.3), în cadrul proiectului. Având în vedere că metoda greedy este folosită în problemele de optim, ea poate fi aplicată pentru rezolvarea parțială a problemei poștaşului chinez.

După cum a fost descris în 3.1, dilema necesită găsirea unui tur eulerian dacă graful este deja eulerian, sau, în caz contrar, să se caute setul de muchii care pot fi multiplicare astfel încât graful să devină eulerian și apoi să se obțină turul corespunzător.

În subsecțiunea 3.3 a fost menționată cazul particular de determinare al grafului eulerian dacă sunt numai două noduri de grad impar, prin dublarea tuturor muchiilor de pe drumul minim între nodurile menționate. În aplicațiile reale, se vor găsi rar astfel de grafuri. În contextul unei localități, aceasta are un număr semnificativ de intersecții prin care trec un număr impar de străzi, asemenea unui nod de grad impar. Totuși, conform corolarului 1.1, un graf care reprezintă o astfel de situație are un număr par de noduri de grad impar. Asta înseamnă că este posibil să formăm perechi de câte două intersecții de acest tip, fiecare cuplu neavând nici un nod în comun.

Folosind paradigma greedy, cazul particular specificat, cât și observația anterioară, se determină combinația de perechi de noduri de grad impar, astfel încât costul suplimentar al grafului pentru a deveni eulerian este aproape de minim, plus să nu existe nod comun între fiecare pereche. Prin urmare, se propune următorul pseudocod pentru a obține un graf eulerian:

Algoritmul 2.5 Algoritmul greedy pentru obținerea grafului eulerian

```

1: procedure GREEDY_DISTANCES( $V_{imp}, E, M, S$ )
2:   Inițializare  $dist_{imp} = \emptyset, D_{double} = \emptyset$ 
3:   for  $\forall v_i \in V_{imp}$  nod origine drum do
4:     for  $\forall v_j \in V_{imp}$  nod destinație drum do
5:       if  $v_i \neq v_j$  și  $m_{ij} \in M$  then
6:          $dist_{imp} = dist_{imp} \cup \{(v_i, v_j), m_{ij}\}$ 
7:   Se sortează crescător  $dist_{imp}$  după  $m_{ij}$ 
8:   Inițializare  $visit_{imp}$  vector de vizitat noduri de grad impar,  $visit_{impi} = fals, \forall v_i \in V_{imp}$ 
9:   for  $\forall dist_{impk} \in dist_{imp}$  do
10:     $v_o, v_d, m_{od} = visit_{impk}$ 
11:    if  $visit_{impo} = fals$  și  $visit_{impd} = fals$  then
12:       $visit_{impo} = adevărat$ 
13:       $visit_{impd} = adevărat$ 
14:       $D_{double} = D_{double} \cup \{(v_o, v_d)\}$ 
15:   Inițializare  $E_{double}, e_{doublej} = (e_j, v(e_j)), v(e_j) = 1$ 
16:   for  $\forall (v_o, v_d) \in D_{double}$  do
17:      $road_{od} = BUILD\_ROAD\_DIJKSTRA(v_d, v_o, S_o)$  (algoritmul 2.4,  $S_o \subset S$  vectorul de
    precedenți în drumurile care pleacă din  $v_o$ )
18:     for  $\forall v_i, v_{i+1} \in road_{od}$  noduri adiacente do
19:        $e_{doublei}(v(e_j)) ++$ 
20:        $grad(v_i) ++ ; grad(v_{i+1}) ++$ 
21:   Returnează  $E_{double}$ 
22: end

```

Notățiile din cadrul algoritmului sunt următoarele:

- $G = (V, E)$ un graf cu mulțimea de noduri V și mulțimea de muchii E ;
- $grad(v_k), v_k \in V$ gradul unui nod din V ;
- $V_{imp} \subset V$ mulțimea de noduri de grad impar;
- Folosind algoritmul lui Dijkstra pe toate nodurile din V_{imp} se obține matricile $M = [m_{ij}]$ pentru costurile, respectiv $S = [s_{ij}]$ pentru predecesorii în drumurile minime către toate nodurile din graf;
- $dist_{imp}$ vectorul tuturor drumurilor între fiecare 2 noduri distincte din V_{imp} ;
- $D_{double} \subset dist_{imp}$ lista celor mai mici distanțe din acesta pentru care perechile de noduri ce formează drumurile nu au nici un nod în comun;
- E_{double} mulțimea derivată din E , care determină formarea grafului eulerian, unde $e_{doublej} = (e_j, v(e_j)), v(e_j)$ reprezentând de câte ori poate fi vizitată muchia e_j .

Soluția algoritmului 2.5 este aproape de optim, căci paradigma greedy, aplicată de la linia 7 până la linia 14 în pseudocod, nu garantează că setul de drumuri are și totalul de cost cel mai mic (conform subsecțiunii 4.3). Totuși, această soluție este aproape de costul minim, ceea ce poate fi considerată o soluție acceptabilă în lipsa unei optimizări a soluției, având în vedere că, până la urmă, graful oferit prin mulțimea de noduri și muchii va deveni eulerian.

2.5.5. Algoritmul lui Hierholzer

Algoritmul 2.6 Algoritmul lui Hierholzer

```

1: procedure ALG_HIERHOLZER( $V_c, E_c$ )
2:   Inițializare tur eulerian  $T = \emptyset$ , sub forma unei liste
3:   Selectare aleatorie nod  $v_i$  din  $V_c$  ca nod de plecare în  $T$ 
4:   while  $\exists v_{ci} \in V_c$  astfel încât  $\text{grad}(v_{ci}) > 0$  do
5:     if  $|T| = 0$  (este primul tur creat) then
6:       Punctul de plecare  $v_k$  este considerat  $v_c$ 
7:     else
8:       Se găsește din setul de noduri distincte din  $T$  primul nod  $v_k$  care are  $\text{grad}(v_{ci}) > 0$ 
       (adică încă mai sunt străzi incidente cu acesta nevizitate)
9:       Inițializare tur nou  $T_c$ , cu nodul de plecare  $v_k$ 
10:      Căutare muchie  $e_{ck}$  incidentă cu nodul  $v_k$  care are cel mai mic  $v(e_{ck}) \neq 0$ 
11:      Adăugare capăt muchie  $e_{ck}$  diferit de  $v_k$  în coada lui  $T_c$ 
12:      Decrementare grade ale capetelor lui  $e_{ck}$  și  $v(e_{ck})$  (se marchează muchia ca fiind vizitată)
13:      while Ultimul element al lui  $T_c$  nu coincide cu  $v_k$  do
14:        Se consideră  $v_l$  nodul coadă din  $T$ 
15:        Căutare muchie  $e_{cl}$  incidentă cu nodul  $v_l$  care are cel mai mic  $v(e_{cl}) \neq 0$ 
16:        Adăugare capăt muchiei  $e_{cl}$  diferit de  $v_l$  în coada lui  $T_c$ 
17:        Decrementează gradele capetelor lui  $e_{cl}$  și  $v(e_{cl})$ 
18:      if  $|T| = 0$  (este primul tur creat) then
19:         $T = T_c$ 
20:      else
21:        Adăugare tur  $T_c$  în  $T$  în locul nodului  $v_c$ 
22:      Returnează turul eulerian  $T$ 
23: end

```

Fundamentarea teoretică al acestuia a fost prezentată în 3.2. Algoritmul va fi folosit pentru a putea obține turul eulerian al dronei prin graful orașului, unde nodurile sunt intersecțiile, iar muchiile sunt străzile; termenii de noduri și muchii se vor interschimba de-a lungul lucrării cu termenii echivalenți din lumea reală (intersecții, respectiv străzi).

Fie G_c graful orașului și perechea de mulțimi (V_c, E_c) reprezentând nodurile, respectiv muchiile acestuia. Se va nota cu v_c numărul de intersecții și cu v_{ci} una dintre acestea ($0 \leq i < v_c$), iar ε_c numărul de străzi și cu e_{cj} una dintre acestea ($0 \leq j < \varepsilon_c$). Fiecărui v_{ci} i se va asocia gradul $\text{grad}(v_{ci})$ pentru a determina cu câte străzi este incidentă. Dacă G_c nu este eulerian, este necesar să fie aplicată paradigma Greedy (algoritmul 2.5.4) pentru a multiplica în cost, aproape de minim, străzile din oraș, astfel încât acesta să devină eulerian. Inițial: $\text{grad}(v_{ci}) \bmod 2 = 0$, conform teoremei 1.2. De asemenea, fiecărei muchii e_{cj} are asociată o variabilă $v(e_{cj}) \geq 0$ ce reprezintă de câte ori poate fi vizitată muchia; dacă aceasta are valoarea mai mare decât 1, atunci muchia a fost multiplicată în algoritmul greedy.

Odată stabilite datele de intrare, algoritmul lui Hierholzer poate fi aplicat prin pseudocodul 2.6. Liniile 10, respectiv 15 sunt operații suplimentare care nu fac parte din algoritmul original, pentru a trece mai întâi prin muchiile care pot fi vizitate doar o dată. Mai multe amănunte despre acest aspect vor fi explicate în capitolul 3, unde se vorbește despre implementarea aplicației.

2.5.6. Algoritmul BFS

Algoritmul 2.7 Algoritm de parcurgere în adâncime pentru găsire noduri izolate

```

procedure FIND_ISOLATED_NODES_BFS(V, E)
  Inițializare  $I, ind, best_{ind}, |best_{ind}|$ 
  while  $\exists v_k \in V$  astfel încât  $i_k = 0$  do
    Selectare nod  $v_o$  pentru care  $i_o = 0$ 
     $ind++$ ;  $count_{ind} = 1$ ;  $i_o = ind$ 
    Inițializare coadă  $Q$  și adăugare  $v_o$  în  $Q$ 
    while  $Q$  nu este goală do
      Extrage capul lui  $Q$  în  $v_l$ 
      for  $\forall e_k \in E$  do
        if  $v_l$  incident cu  $e_k$  then
          Fie  $v_k$  celălalt capăt al muchiei  $e_k$ 
           $i_k = ind$ 
          Adăugare  $v_k$  în  $Q$ 
           $count_{ind}++$ 
      if  $count_{ind} > |best_{ind}|$  then
         $|best_{ind}| = count_{ind}$ 
         $best_{ind} = ind$ 
    Returnează lista tuturor nodurilor  $v_k \in V$  pentru care  $i_k \neq best_{ind}$ 
end

```

Atunci când se extrage graful specific orașului din fișierul-hartă dat, pot apărea în acesta „insule”, adică subgrafuri izolate, ceea ce determină ca graful întru totul să nu fie conex. Această situație apare când sunt preluate secțiuni de străzi (de regulă de la marginea zonei), care nu au o legătură cu celelalte muchii printr-un drum. În consecință, algoritmul lui Hierholzer (descriș în subsecțiunea 2.6) riscă să ruleze în buclă infinită, deoarece nu va reuși să viziteze și străzile din insulele care apar în graf.

De aceea, este necesar ca înainte de a-l prelucra, spre obținerea turului eulerian, să se elimine mai întâi subgrafurile izolate astfel încât noul graf să devină conex. Există două variante facile de a determina soluția problemei curente: căutarea în adâncime (DFS - en. Depth-First Search), care constă în parcurgerea tuturor nodurilor, deplasându-se mai întâi pe prima ramură a grafului, până nu mai găsește vecini nevizitați, și a doua metodă: căutarea în lățime (BFS - en. Breadth-First Search), unde la fiecare nod curent se vor vizita, pe rând, vecinii acestuia. A doua metodă a fost aleasă în aplicație, deoarece este ideală pentru căutarea de subgrafuri izolate în graf.

Algoritmul BFS sugerat în această aplicație va vizita până la urmă toate nodurile și muchiile din graf, însă va evidenția care noduri fac parte din anumite insule. Pornind de la un nod oarecare, ce va deveni reprezentantul primului subgraf conex, va parcurge în lățime toate nodurile din acel subgraf. Odată ce prima insulă a fost complet vizitată, se găsește următorul nod din graf care nu a fost încă umblat, devenind reprezentantul următoarei insule, iar procedeul se repetă. Găsirea insulelor se termină atunci când toate intersecțiile din oraș au fost vizitate. La final se returnează nodurile care nu fac parte din subgraful conex cu cele mai multe noduri, și vor fi eliminate mai târziu din graful original, alături de muchiile incidente lor, în afara algoritmului propriu-zis.

Fie $G = (V, E)$ graful determinat de mulțimea de vârfuri (noduri) V și mulțimea de muchii E . Muchia dintre nodurile v_i și v_j se marchează cu e_{ij} . Se consideră I vectorul de apartenență a nodurilor la o insulă, unde inițial $i_k = 0, \forall v_k \in V$, adică niciun nod nu este vizitat. De asemenea, se notează cu ind indexul insulei curente de parcurs, unde inițial $ind = 0$, iar $best_{ind}$ și $|best_{ind}|$ indexul insulei cu cele mai multe noduri, respectiv numărul acestora, unde inițial $best_{ind} = null, |best_{ind}| =$

0. În timpul vizitei unui subgraf coada pentru parcurgerea în lățime al acestuia se va nota cu Q , iar numărul de noduri vizitate în acesta cu $count_{ind}$.

Odată stabilite notațiile, este prezentat în pseudocodul 2.7 algoritmul BFS pentru găsirea subgrafurilor conexe izolate. După cum se poate vedea în acesta, se consideră izolate doar nodurile care fac parte din subgrafuri conexe de dimensiune mult mai mici. Astfel, odată cu eliminarea acestora, se asigură formarea unui graf conex din cel original, fără a-l denatura prea mult. În cazul în care nu se găsesc vârfuri izolate, atunci graful va rămâne neschimbat; indiferent de deznodământ, procesarea turului eulerian poate avea loc fără probleme.

2.6. Proiectarea interfeței aplicației

Interfața a fost proiectată pe „hârtie”, folosind instrumentul Paint pentru a schița forma acesteia. În imaginile de mai jos sunt prezentate etapele proiectării interfeței.

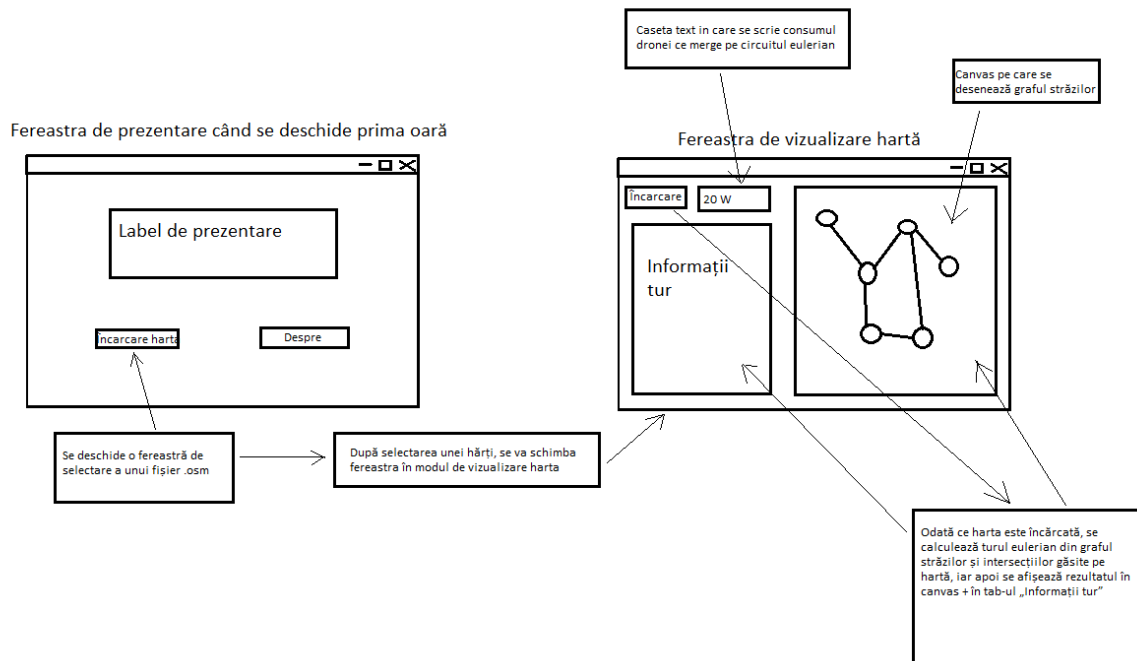


Figura 2.5. Prima etapă de proiectare: inițial, s-a gândit la o pagină de „Welcome” din care se poate încărca harta + buton despre aplicație; odată selectată harta, interfața se schimbă cu cea din dreapta, unde se redă harta animată, unde se prezintă circuitul dronei prin ea; În informații tur se regăsesc și consumul dronei pe parcursul traseului

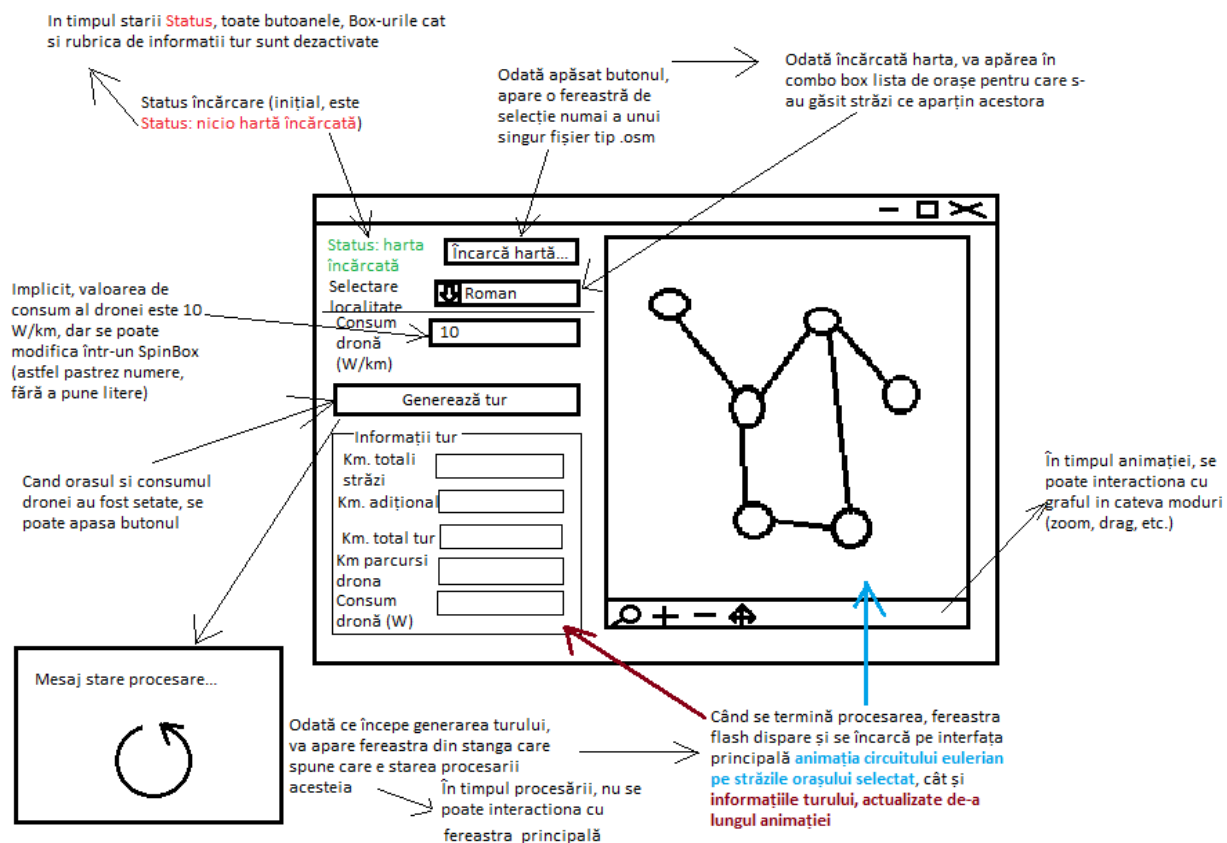


Figura 2.6. Etapa a doua, și cea definitorie pentru forma actuală a interfeței: s-a renunțat la pagina de „Welcome”, accentuându-se pe o formă compactă a GUI-ului, unde se poate vizualiza direct traseul dronei

Capitolul 3. Implementarea aplicației

1. Descrierea generală a implementării

Mare parte din implementarea aplicației s-a realizat prin programarea funcțională: folosirea de sub-programe(funcții) pentru ca mai apoi să fie asamblate în programul principal.

Fișierul *XMLParser.py* din componența programului are toate funcțiile necesare obținerii turului eulerian, dintr-un fișier XML OSM care redă harta unei zone rezidențiale.

În cuprinsul acestei secțiuni, se evidențiază construcția tuturor funcțiilor din acest fișier, împărțite pe câte o subsecțiune. De precizat este că funcția va fi sinonim cu o componentă, aceasta joacă un rol esențial în determinarea soluției finale. În contextul unei subsecțiuni se va prezenta:

- O scurtă descriere a funcționării componentei, alături de capturi de ecran care relevă aceasta;
- Explicații pe codul care o implementează: idei noi/original, probleme întâmpinate de-a lungul implementării și soluții de rezolvare, analiza și justificarea părților din cod folosite în funcție.

1.1. Componenta de prelucrare a hărții într-un graf

Prelucrarea hărții constă în analiza fișierului XML OSM încărcat din GUI și extragerea grafului format din toate străzile și intersecțiile prin care trec, dintr-un oraș al cărui nume este preluat tot din interfața cu utilizatorul. Funcția reprezentativă din fișierul *XMLParser.py* este *get_city_from_map*, care are doi parametri: *filename* - numele fișierului .osm din care este extrasă harta, și *city_name* - numele orașului al cărui graf va fi extras. Codul complet al acestei funcții este afișată în Anexa 1.

Pentru a începe extragerea, este necesar mai întâi să fie parsat fișierul .osm, care este în format XML. Modulul *xml.etree* oferă clasa *ElementTree* pentru a putea parcurge mai ușor conținutul unui fișier XML. Din aceasta s-a folosit funcția statică *parse*, căruia s-a furnizat parametrul *filename* pentru a obține arborele conținutului hărții, iar în cele din urmă să obțină rădăcina acestuia, folosind funcția arborelui *getroot()*. Din aceasta se poate accesa toate informațiile despre zona reprezentată: străzi, orașe, păduri, etc.

```
1 tree = ET.parse(filename)
2 root = tree.getroot()
```

Listing 3.1. Prelucrare conținutul hărții folosind *xml.etree.ElementTree(ET)*

Odată ce a fost obținută harta, se poate trece la obținerea intersecțiilor și străzilor din orașul dorit. Formatul .osm are corespondențe pentru cele 2 elemente: nodul și calea (subsecțiunea 2.1). Dar trebuie multă grijă în acest pas, căci nodurile pot reprezenta și centrul unei localități, treceri de cale ferată, punctele perimetrului unei clădiri/gard, etc., iar căile pot reprezenta chiar suprafața unei păduri. De aceea, fiecare cale va avea atașată etichete(tag-uri) care să sugereze ce reprezintă ea, cât și, în sub-arborele său, toate nodurile căruia îi aparține. În cazul aplicației, ne interesează doar căile și nodurile care formează o stradă, o șosea, ce aparține orașului nostru.

Pentru început, se extrag toate nodurile care se află în fișier, unde vor fi păstrate într-un dicționar, unde cheia este id-ul nodului, iar valoarea este dicționarul având latitudinea și longitudinea punctului:

```
1 node_dict = {}
2 for node in root.iter('node'):
3     node_dict[node.attrib['id']] = {'lat': float(node.attrib['lat']),
4                                     'lon': float(node.attrib['lon'])}
```

Listing 3.2. Codul de extragere al tuturor nodurilor de pe hartă

Funcția membră `root.iter(tag)` returnează un iterator al tuturor elementelor arborelui hărții, care sunt etichetate cu numele `tag`. De asemenea, metoda `node.attrib[attrib_name]` accesează lista de attribute din elementul XML `node` și preia valoarea atributului `attrib_name`.

Următoarea etapă este găsirea străzilor din orașul dorit. De data aceasta, se urmărește lista etichetelor fiecărei căi de pe hartă, unde o etichetă prezintă 2 attribute: `k` care înseamnă numele etichetei (cheia), și `v` - adică valoarea etichetei respective. Sunt urmărite două etichete cu următoarele chei: `'highway'` - calea reprezintă o șosea, a cărei valori marchează importanța acesteia (principală, secundară, etc.) - și `'is_in:city'` - calea se află în localitatea a cărei nume se găsește în valoarea etichetei. Dacă una dintre acestea nu este găsită, sau nu corespund cu orașul dorit, atunci calea nu este luată în considerare. Odată găsită o stradă, se preiau toate id-urile nodurilor pe care o alcătuiesc și se introduc într-o listă, iar în plus nodurile găsite se introduc în mulțimea de intersecții ale orașului (dicționarul `street_nodes_dict`). În cele din urmă, toate străzile găsite se adaugă într-un dicționar, a căror valoare conține lista nodurilor aparținătoare.

```

1  street_nodes_dict = {}; streets_dict = {}
2  for child in root.findall('way'):
3      highway, name, in_city, nodes = None, (), None, []
4
5      # De aceea, se cauta căile care sunt străzi principale (drumuri
6      ↪ nationale, europene, autostrazi),
7      # secundare, terțiare și rezidențiale (strazi din oras), care
8      ↪ sunt catalogate aparținând orașului dorit
9      for tag in child.iter('tag'):
10         if tag.attrib['k'] == 'highway' and tag.attrib['v'] in
11         ↪ ['primary', 'secondary', 'tertiary', 'residential']:
12             highway = tag.attrib['v']
13         elif tag.attrib['k'] == 'is_in:city':
14             in_city = tag.attrib['v']
15         elif tag.attrib['k'] == 'name':
16             name = tag.attrib['v']
17
18         # daca a fost gasita o astfel de strada
19         if in_city == city_name and highway is not None:
20             # toate nodurile din acesta sunt catalogate drept noduri
21             ↪ (intersecții) ce apartin grafului orasului
22             for node in child.iter('nd'):
23                 node_id = node.attrib['ref']
24                 if node_id not in street_nodes_dict.keys():
25                     street_nodes_dict[node_id] =
26                     ↪ node_dict[node_id].copy()
27                     street_nodes_dict[node_id]['grade'] = 0
28                 nodes.append(node.attrib['ref'])
29             if child.attrib['id'] not in streets_dict.keys():
30                 streets_dict[child.attrib['id']] = {'name': name,
31                 ↪ 'highway': highway, 'nodes': nodes}

```

Listing 3.3. Codul de extragere al străzilor și intersecțiilor orașului

Metoda `root.findall(tag)` este asemănătoare cu funcția `root.iter(tag)`, cu diferența în ceea ce returnează: primul returnează o listă cu toate elementele cu tag-ul `tag`, în loc de un iterator.

Deși s-au obținut nodurile și străzile specifice grafului orașului, încă nu există muchii cu un cost de parcurgere. În fișier .osm nicio cale nu are atribuită distanța în kilometri, deci este necesară o metodă de a calcula distanțele între fiecare pereche de noduri adiacente din oraș. Acestea vor deveni muchiile din graf.

Inițial, s-a propus folosirea API-ului Distance Matrix, care calculează distanța între două noduri având coordonatele latitudine și longitudine, fiind un instrument de nădejde în timpul activităților din practica de vară. Totuși are un mare dezavantaj: pentru a putea fi folosit, este necesar un card pentru plata serviciilor oferite de API, iar costurile pot crește semnificativ: 4 USD pentru fiecare 1000 cereri către API. Această parte ar fi depins și de conectivitatea la Internet, iar în urma dezavantajelor menționate API-ul nu a fost ales pentru a calcula costurile muchiilor.

În schimb, o metodă mult mai simplă pentru calculul distanțelor este *formula haversină*: determinarea lungimii liniei dintre două puncte aflate pe un glob. Formula este des utilizată în calculul distanțelor de pe Pământ, planeta însăși având forma apropiată cu cea a unei sfere. Vor exista desigur erori minore față de lungimea reală, pentru că, în realitate, Pământul este un geoid - ușor bombat la Ecuator și ușor turtit la poli. Distanțele calculate în cadrul aplicației sunt foarte mici față de circumferința planetei (40,075 km), deci eroarea de calcul poate fi neglijată, fără consecințe majore asupra rezultatului final.

Pentru fiecare stradă din graful orașului, se preia fiecare pereche de noduri adiacente din aceasta, și se calculează costul în kilometri al muchiei dintre ele utilizând formula haversină. Apoi se introduce muchia rezultată într-un dicționar, unde cheia va fi perechea de vârfuri, iar valoarea este lungimea muchiei dintre ele. În plus se măresc gradele nodurilor aparținente, în dicționarul de intersecții.

Codul operației de calcul al muchiilor este oferit mai jos, alături de explicațiile notațiilor formulei haversine:

```

1  # se foloseste formula haversina, ce calculeaza distanta intre 2
    ↳ puncte aflate pe glob
2  # formula preluată din
    ↳ http://www.movable-type.co.uk/scripts/latlong.html
3  graph_edges_dict = {}
4  R = 6371 # raza Pământului, în kilometrii
5  for street in streets_dict.keys():
6      nodes = streets_dict[street]['nodes']
7      for node1, node2 in zip(nodes[:-1], nodes[1:]):
8          lat1, lon1 = street_nodes_dict[node1]['lat'],
            ↳ street_nodes_dict[node1]['lon']
9          lat2, lon2 = street_nodes_dict[node2]['lat'],
            ↳ street_nodes_dict[node2]['lon']
10
11     # se convertesc coordonatele in radiani
12     phi1, phi2 = lat1 * math.pi / 180, lat2 * math.pi / 180
13     lambda1, lambda2 = lon1 * math.pi / 180, lon2 * math.pi / 180
14
15     diff_lambda = lambda2 - lambda1
16     diff_phi = phi2 - phi1
17
18     # se foloseste formula haversina, ce calculeaza distanta
        ↳ intre 2 puncte aflate pe glob
19     # a = patrutul jumatatii din lungimea arcului cercului dintre
        ↳ 2 puncte
20     # c = distanta unghiulara in radiani
21     # d = distanta obtinuta

```

```

22     a = math.sin(diff_phi / 2) ** 2 + math.cos(phi1) *
    ↪ math.cos(phi2) * (math.sin(diff_lambda / 2) ** 2)
23     c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
24     d = R * c
25
26     # pe langa adaugarea muchiei in graf, se maresc si gradele
    ↪ celor 2 noduri
27     graph_edges_dict[(node1, node2)] = d
28     street_nodes_dict[node1]['grade'] += 1
29     street_nodes_dict[node2]['grade'] += 1

```

Listing 3.4. Codul de determinare al muchiilor din graf, cost calculat cu formula haversină

La final, se obține atât dicționarul de noduri, cât și cel de muchii al grafului orașului. Înainte de a le returna, se simplifică notațiile acestora, asignând fiecărui vârf un număr de ordine:

```

1     node_index_dict = {graph_node: index for index, graph_node in
    ↪ enumerate(list(street_nodes_dict.keys()))}
2     indexed_nodes_dict = {node_index_dict[key]: value for key, value in
    ↪ street_nodes_dict.items()}
3     indexed_edges_dict = {(node_index_dict[edge[0]],
    ↪ node_index_dict[edge[1]]): value
4                             for edge, value in graph_edges_dict.items()}
5
6     return indexed_nodes_dict, indexed_edges_dict

```

Listing 3.5. Codul pentru simplificarea notațiilor dicționare

1.2. Componenta de găsire și eliminare al nodurilor izolate

Reamintim din subsecțiunea 2.5.4 că este necesară eliminarea nodurilor izolate din graful orașului pentru a preveni blocarea algoritmului de determinare al circuitului eulerian. Metoda de găsire al acestora a fost descrisă în algoritmul 2.7 și se bazează pe căutarea în lățime a subgrafurilor neconexe, cu un număr minim de noduri. Ele sunt incidente cu muchii care nu vor fi niciodată parcurse în algoritmul lui Hierholzer, pentru că se bazează pe ideea unui graf conex (subsecțiunea 2.5.5).

Algoritmul 2.7 este implementat în funcția *find_isolated_nodes(nodes_dict, edges_dict)*, ai cărei parametri sunt dicționarele nodurilor, respectiv muchiilor grafului orașului. Codul ei este afișat în Anexa 2. Mai departe sunt explicate câteva părți importante din implementarea metodei.

Pe linia 3 al codului s-a folosit funcția *numpy.full(format, value)* pentru a inițializa lista de vizitare/apartenență al vârfurilor la insule, unde parametrul *format* determină forma listei - dacă este un tuplu de două sau mai multe numere întregi, atunci devine o matrice, altfel este o simplă listă de lungimea numărului dat - iar *value* reprezintă valoarea care va umple fiecare poziție din listă.

Condiția de pe linia 15 *np.any(np_array)* evaluează toate elementele din lista *np_array* ca fiind adevărate. Dacă se introduce și o relație de egalitate (*np_array == val*), se verifică dacă toate valorile din listă sunt egale cu *val*. Funcția preluată din modulul *numpy* va returna *True* dacă condiția din funcție este adevărată, *False* în caz contrar.

În timpul vizitării insulelor grafului, coada care păstrează ordinea de parcurgere în lățime - variabila *island_queue* este instanțiată cu un obiect tip *Queue*, preluat din *modulul queue*, care oferă funcționalitatea unei cozi oarecare.

La final, ultima linie din cod care returnează lista nodurilor izolate, reprezentate de vârfurile care nu fac parte din insula cu cele mai multe dintre noduri, a fost creată folosind procedeul „List

Comprehension”, o variantă în Python de a crea mai simplu liste, în adaos cu filtrarea elementelor din acesta. Este mult mai ușor de citit și de înțeles decât operatorii repetitivi, deoarece se concentrează pe ceea ce trebuie să conțină lista, având încredere în interpretorul Python cu asignările de rigoare, decât pe declarațiile și operațiile specifice unei bucle.

Odată ce se obține lista menționată anterior, are loc în afara algoritmului etapa de eliminare al vârfurilor separate. Aceasta se execută doar dacă s-au găsit într-adevăr astfel de noduri (lungimea listei este nenulă). Practic, se elimină din dicționarul de muchii cele incidente cu cel puțin una dintre vârfurile detașate, urmată de ștergerea acestora din dicționarul de noduri. Se asigură că gradele nodurilor scad odată cu eliminarea muchiilor incidente. Codul procedurii este afișat mai jos:

```

1  # se cauta noduri care sunt deconectate de la graf, folosind BFS
2  isolated_nodes = find_isolated_nodes(nodes_dict, edges_dict)
3
4  # daca au fost gasite noduri izolate
5  if len(isolated_nodes) > 0:
6      print(f'Isolated nodes: {isolated_nodes}')
7      marked_edges = []
8      for edge in edges_dict:
9          node1, node2 = edge
10         if node1 in isolated_nodes or node2 in isolated_nodes:
11             marked_edges.append(edge)
12
13     for edge in marked_edges:
14         node1, node2 = edge
15         edges_dict.pop(edge)
16         nodes_dict[node1]['grade'] -= 1
17         nodes_dict[node2]['grade'] -= 1
18
19     for isolated_node in isolated_nodes:
20         nodes_dict.pop(isolated_node)
21
22 else:
23     print('No isolated nodes found')

```

Listing 3.6. Codul eliminării nodurilor izolate

Muchiile care au noduri izolate incidente sunt „marcate” în vectorul *marked_edges*. Din acesta se iterează fiecare din ele și se elimină corespunzător.

Deoarece în algoritmiile următoare indexul nodurilor este între 0 și numărul total de noduri curent, este necesară o calibrare a dicționarelor de muchii și noduri, datorită scăderii numărului de vârfuri în urma procedurii antecedent; operația este similară codului 3.5.

1.3. Componenta de calcul al drumurilor între noduri de grad impar

În subsecțiunea 2.5.3 s-au comparat doi algoritmi de determinare al drumurilor minime dintre nodurile unui graf: Floyd Warshall (algoritmul 2.1) și Dijkstra (algoritmul 2.3) pentru a vedea care dintre ei este mai potrivit în aplicația curentă. S-a decis folosirea metodei Dijkstra, rulată paralel pentru fiecare nod de grad impar pentru a accelera calculele. În proiect, au fost implementate ambele variante pentru a compara experimental timpul de procesare al celor două soluții, folosite la un moment dat în timpul implementării aplicației; rezultatele vor fi prezentate în capitolul 4.

Prima care a fost utilizată a fost algoritmul Floyd-Warshall, în rulare secvențială. Pseudo-codul acestuia (2.1) este implementată de funcția `floyd_warshall_alg(nodes_dict, graph_edges_dict)`, unde cei doi parametri sunt încă o dată dicționarele muchiilor și nodurilor grafului. Codul funcției este prezentat mai jos:

```

1  def floyd_warshall_alg(nodes_dict: dict, graph_edges_dict: dict):
2      nodes_count = len(nodes_dict.keys())
3      nodes_list = [graph_node for graph_node in nodes_dict]
4      dists = np.full((nodes_count, nodes_count), np.inf) # matricea
        ↳ de distante minime;
5      # initial, toate au valoarea infinit
6
7      next_node_m = np.full((nodes_count, nodes_count), None) #
        ↳ matricea de adiacenta a nodurilor;
8      # initial, toate au valoarea None (adica nu sunt adiacente cu vreun
        ↳ nod in drum)
9
10     # initializarea matricilor de distante minime, respectiv de
        ↳ adiacenta
11     for u in range(nodes_count):
12         for v in range(nodes_count):
13             edge_g = (nodes_list[u], nodes_list[v])
14             if u != v and edge_g in graph_edges_dict.keys():
15                 dists[u, v] = graph_edges_dict[edge_g]
16                 dists[v, u] = graph_edges_dict[edge_g]
17                 next_node_m[u, v] = v
18                 next_node_m[v, u] = u
19             elif u == v:
20                 dists[u, v] = 0
21                 next_node_m[u, v] = v
22
23     # determinarea propriu-zisa a distantelor minime, respectiv a
        ↳ adiacentei in drum
24     for k in range(nodes_count):
25         for i in range(nodes_count):
26             for j in range(nodes_count):
27                 if dists[i, j] > dists[i, k] + dists[k, j]:
28                     dists[i, j] = dists[i, k] + dists[k, j]
29                     next_node_m[i, j] = next_node_m[i, k]
30
31     return nodes_list, dists, next_node_m

```

Listing 3.7. Algoritmul Floyd-Warshall

Odată ce a fost gândită mai bine perspectiva, codul 3.7 a fost înlocuit cu funcția *dijkstra_alg(nodes_dict, edges_dict, start_node)*, ai cărei primi doi parametri sunt aceiași ca în algoritmul precedent, iar al treilea este nodul de origine al drumurilor către celelalte noduri din graf.

```

1  def dijkstra_alg(nodes_dict: dict, edges_dict: dict, start_node):
2      nodes_count = len(nodes_dict.keys())
3
4      lowest_dists = np.full(nodes_count, np.Inf) # lista celor mai
           ↪ mici distanțe între nodul de plecare(start_node)
5      # și celelalte noduri din graf; inițial, acestea sunt distanțe
           ↪ infinite (maxime)
6
7      prev_node = np.full(nodes_count, None) # lista noduri precedente
           ↪ drumului celui mai scurt între nodul start_node și
8      # celelalte noduri din graf; inițial nu se cunosc drumurile (deci
           ↪ nu au precedent)
9
10     visited_nodes = np.full(nodes_count, False) # lista de booleane,
           ↪ visited_node[i] reprezentând dacă nodul de indexul
11     # i din nodes_list a fost vizitat sau nu în cadrul algoritmului
           ↪ lui Dijkstra
12
13     lowest_dists[start_node] = 0
14
15     while not np.all(visited_nodes):
16         min_node_index = min([(index, lowest_dists[index]) for index
           ↪ in range(nodes_count)
17                               if visited_nodes[index] == False],
           ↪ key=lambda x: x[1])[0]
18         visited_nodes[min_node_index] = True
19
20         for node_index in range(nodes_count):
21             if not visited_nodes[node_index]:
22                 edge = None
23                 if (node_index, min_node_index) in edges_dict.keys():
24                     edge = (node_index, min_node_index)
25                 elif (min_node_index, node_index) in
           ↪ edges_dict.keys():
26                     edge = (min_node_index, node_index)
27                 if edge is not None:
28                     dist = lowest_dists[min_node_index] +
           ↪ edges_dict[edge]
29                     if dist < lowest_dists[node_index]:
30                         lowest_dists[node_index] = dist
31                         prev_node[node_index] = min_node_index
32
33     return lowest_dists, prev_node

```

Listing 3.8. Algoritmul Dijkstra

Codurile de recreere al drumurilor pentru fiecare metodă în parte (pseudocodurile 2.2 și 2.4) sunt evidențiate în Anexa 3.

Având în vedere că algoritmul Dijkstra determină drumurile doar de la un singur nod origine, el trebuie apelat pentru toate nodurile de grad impar. Calcularea secvențială era mai rapidă

decât algoritmul Floyd-Warshall, datorită faptului că numai o parte din drumuri sunt calculate, dar nu suficient de rapidă pentru a procesa în scurt timp acestea. Prin urmare, se implementează această procedură prin paradigma multi-processing, în ajutorul căruia se folosesc modulele `concurrent.futures` și `itertools` (modul util pentru crearea de iteratoare) din Python. Mai jos este prezentată codul procesării paralele al drumurilor minime între fiecare nod de grad impar, folosind algoritmul lui Dijkstra.

```

1  # se prelucrează paralel, folosind un executor de procese, vectorii
   ↳ de distante, cât și de precedenți,
2  # pentru fiecare nod de grad impar
3  with concurrent.futures.ProcessPoolExecutor() as executor:
4      nodes_list = list(odd_nodes_dict.keys())
5      for node_i, result in zip(nodes_list, executor.map(dijkstra_alg,
   ↳ repeat(nodes_dict), repeat(edges_dict), nodes_list)):
6          dist_nodes, prev_nodes = result
7          odd_nodes_dict[node_i]['dist_nodes'] = dist_nodes
8          odd_nodes_dict[node_i]['prev_nodes'] = prev_nodes

```

Listing 3.9. Codul procesării paralele al metodei lui Dijkstra aplicat pe toate nodurile de grad impar

Obiectul *executor* este instanța clasei `concurrent.futures.ProcessExecutor`, care permite executarea mai multor procese odată, fiind păstrate într-un *pool* care le gestionează. Cum a fost menționat în subsecțiunea 2.3.3, Pool-ul are un număr „nelimitat”(atât cât permite sistemul de operare) de procese disponibile dacă nu se precizează parametrul *max_workers*.

Declarația *with ... as* este specifică Python și contribuie la alocarea și dealocarea sănătoasă al obiectului *executor*, precum și tratarea excepțiilor în ceea ce privește cele doi acțiuni anterioare.

Funcția `executor.map(func, *args, **kwargs)` va apela funcția *func* de câte elemente sunt într-un parametru în **args* și ***kwargs*, ceilalți parametri având aceeași dimensiune ca și cel specificat. La final, funcția returnează un iterator din care se așteaptă fiecare rezultat din procesele executante. În contextul codului 3.9, pentru că nu pot fi puse dicționarele de noduri și muchii o singură dată, s-a utilizat funcția `repeat(obj, [,times])` din modulul `itertools` care returnează un iterator cu obiectul *obj* în număr de *times* ori, însă e de ajuns să nu fie specificat ultimul parametru pentru a returna la infinit obiectul. Astfel se pot alimenta toate procesele fără a arunca excepții din lipsă de argumente.

Funcția built-in `zip(iterable1, iterable1)` returnează un iterator, unde fiecare membru reprezintă un tuplu format din elementele din iteratorii *iterable1* și *iterable2* aflate la același index.

Pentru fiecare operație terminată pentru un nod de grad impar, se introduc dicționarele de drumuri minime și de predecesori în valoarea vârfului aferent în dicționarul de noduri de grad impar.

Folosind dicționarele de drumuri minime rezultate anterior, se formează un nou dicționar de drumuri minime între fiecare pereche de noduri de grad impar, unde cheia este tuplul reprezentativ al perechii, iar valoarea este distanța minimă aferentă. Codul operațiunii este evidențiat mai jos:

```

1  # se determina dicționarul de distante între fiecare pereche posibila
   ↳ de noduri de grad impar
2  odd_pairs_distances = {}
3
4  for odd_node1 in odd_nodes_dict.keys():
5      for odd_node2 in odd_nodes_dict.keys():
6          if odd_node1 != odd_node2 and (odd_node2, odd_node1) not in
   ↳ odd_pairs_distances.keys() \

```



```

7         and
          ↪ odd_nodes_dict[odd_node1]['dist_nodes'][odd_node2]
          ↪ != np.Inf:
8     odd_pairs_distances[(odd_node1, odd_node2)] =
          ↪ odd_nodes_dict[odd_node1]['dist_nodes'][odd_node2]

```

Listing 3.10. Codul de obținere al dicționarului de distanțe minime între fiecare pereche de noduri de grad impar

Componenta descrisă până acum este definitorie pentru următoarea parte, cea de calcul efectivă al minimizării costului circuitului eulerian, folosind paradigma greedy.

1.4. Componenta de calcul al soluției problemei poștaşului chinez

Odată ce s-au determinat toate drumurile minime între fiecare pereche de noduri de grad impar în componenta 1.3, se poate trece la aplicarea paradigmei greedy pentru a soluționa problema poștaşului chinez în contextul grafului orașului, cum a fost descrisă în subsecțiunea 2.5.4. Pseudocodul 2.5 aferent metodei prezentate este introdus în funcția *greedy_eulerian_graph(nodes_dict, edges_dict, odd_pairs_distances, odd_nodes_dict, fd_of_djk)*, unde primii doi parametri sunt dicționarele de noduri și muchii specifici grafului, următorii doi sunt dicționarele de drumuri minime (obținut prin codul 3.10), respectiv de noduri de grad impar care le formează, iar ultimul determină modul de construire al drumului (Floyd-Warshall sau Dijkstra). Codul funcției este redat mai jos:

```

1  def greedy_eulerian_graph(nodes_dict, edges_dict,
   ↪ odd_pairs_distances, odd_nodes_dict, fd_or_djk):
2      # metoda Greedy: se extrag cele mai mici distante intre fiecare
   ↪ pereche de noduri de grad impar, tinand cont ca
3      # marginile sa nu se repete (numar de muchii = numar de noduri
   ↪ grad impar / 2)
4      distances = [(dist, val) for dist, val in
   ↪ odd_pairs_distances.items()]
5      distances.sort(key=lambda x: x[1])
6
7      selected_nodes = []
8      selected_distances = []
9      for dist, val in distances:
10         node1, node2 = dist
11         if node1 not in selected_nodes and node2 not in
   ↪ selected_nodes:
12             selected_distances.append((dist, val))
13             selected_nodes.append(node1)
14             selected_nodes.append(node2)
15
16     # crearea noului graf de strazi, adaugand muchiile determinate in
   ↪ Procedura Greedy
17
18     # initial, nodurile au acelasi grad
19     # iar muchiile vor avea in plus o valoare de frecventa
20     # (de cate ori se poate parcurge nodul in circuitul eulerian)
21     for distance in edges_dict.keys():
22         edges_dict[distance] = {'d': edges_dict[distance], 'freq': 1}
23
24     for dist, val in selected_distances:

```

```

25     node1, node2 = dist
26     # se obtine calea drumului minim dintre nodurile alese
27     node_path = dijkstra_path_odd(node1, node2, odd_nodes_dict)
        ↪ if fd_or_djk is False else floyd_warshall_path(node1,
        ↪ node2, nodes_dict)
28     for l_node, r_node in zip(node_path[:-1], node_path[1:]):
29         # daca muchia dintre cele 2 noduri adiacente exista deja
30         dist_changed = None
31         if (l_node, r_node) in edges_dict.keys():
32             dist_changed = (l_node, r_node)
33         elif (r_node, l_node) in edges_dict.keys():
34             dist_changed = (r_node, l_node)
35
36         if dist_changed is not None:
37             # se mareste frecventa acesteia, si cresc gradele
            ↪ capetelor
38             edges_dict[dist_changed]['freq'] += 1
39             nodes_dict[l_node]['grade'] += 1
40             nodes_dict[r_node]['grade'] += 1

```

Listing 3.11. Codul funcției *greedy_eulerian_graph*

Codul prezentat doar modifică graful orașului astfel încât să permită obținerea unui tur eulerian. Mai întâi se alege cele mai puțin costisitoare drumuri din dicționarul *odd_pairs_distances*, astfel încât acestea să nu aibă niciun capăt în comun (liniile 4-14). În cele din urmă, se parcurge fiecare drum selectat (linia 24), din care se obține nodurile prin care trece acesta prin una din codurile 5 - dacă s-au obținut distanțele prin algoritmul Dijkstra - sau 4 - dacă s-a folosit algoritmul Floyd-Warshall (liniile 25-27). Fiecare muchie dintr-un drum va avea frecvența crescută (de câte ori poate fi parcursă în turul eulerian) în dicționarul *edges_dict*, și nodurile incidente cu gradul crescut în dicționarul *nodes_dict* (liniile 28-40).

Așadar dicționarele modificate vor determina un graf eulerian pe care se poate aplica orice algoritm de obținere al turului eulerian. O astfel de metodă este implementată în componenta următoare.

1.5. Componenta de determinare al turului eulerian

În subsecțiunea 3.2 s-au comparat diverse metode de determinare al soluției poștașului chinez când graful este deja eulerian. Algoritmul lui Hierholzer a fost ales drept o soluție mult mai bună pentru determinarea turului grafului, datorită complexității liniare față de numărul de muchii. Drept urmare, în subsecțiunea 2.5.5 a fost prezentată și contextul în care este folosită, plus un algoritm pentru obținerea soluției, în pseudocodul 2.6. Acesta a fost deja implementat în fișierul XMLParser, în funcția *hierholzer_alg(nodes_dict, edges_dict)*, unde parametrii constituie un graf eulerian. Codul funcției este relevant în Anexa 4.

S-a menționat în pseudocodul 2.6 de liniile 10 și 15, care reprezintă îmbunătățiri ale algoritmului. Scopul acestora este de a trece întâi prin muchiile prin care se pot traversa la momentul iterației o singură dată. Deși acest lucru nu impactează cu mult viteza de procesare al algoritmului, ba chiar riscă să îl încetinească (la fiecare nod din tur se verifică toate muchiile din graf), această operație asigură căile din jurul unui vârf să fie toate vizitate din timp.

Codul acestui procedeu, preluat din Anexa 4, este afișat mai jos, și se aplică în ambele linii precizate, cu mici variații:

```

1     for g_edge in edges_dict_copy.keys():
2         node_n, node_m = g_edge

```

```

3      # cand este gasit o muchie cu frecventa pozitiva (inca mai poate
      ↪ fi vizitata)
4      # si mai mica decat minimul curent
5      if (node_n == new_tour[-1] or node_m == new_tour[-1]) \
6          and 0 < edges_dict_copy[g_edge]['freq'] < lowest_freq:
7          # se pastreaza muchia si se inlocuieste minimul cu noua
          ↪ frecventa
8          lowest_freq = edges_dict_copy[g_edge]['freq']
9          chosen_edge = g_edge
10
11     # La finalul buclei for se va gasi o muchie cu frecventa cea mai
        ↪ mica
12     # Astfel ne asiguram sa trecem prin muchiile cu frecventa 1 mai
        ↪ intai, pentru a nu fi vizitate din nou in
13     # alte tururi
14     # Se adauga muchia cu cea mai mica frecventa strict pozitiva la
        ↪ turul curent
15     node_n, node_m = chosen_edge
16     new_tour.append(node_n if node_n != new_tour[-1] else node_m)

```

Listing 3.12. Codul de determinare al celei mai puțin frecventate muchie incidentă cu ultimul vârf din tur

1.6. Componenta principală: funcția de generare al turului eulerian pornind de la un fișier .osm

Această componentă este esența fișierului XMLParser.py, întrucât combine toate funcțiile și etapele descrise până acum, cu scopul obținerii circuitului eulerian pornind de la graful orașului extras din harta obținută într-un fișier OSM XML.

Funcția `generate_course(source_filename, settlement_name, roadExtractionEvent: threading.Event, routeProcessingEvent: threading.Event, animationProcessingEvent: threading.Event, stopEvent: threading.Event, solution_content)` este cea în care se întâmplă acest fapt, unde primii 2 parametri sunt numele fișierului .osm, respectiv numele orașului al cărui graf este extras. Următoarele patru argumente sunt obiecte tip `threading.Event` pentru a semnaliza interfeței GUI când s-a terminat o procedură a funcției. Detalii mai amănunțite despre aceste semnale vor fi în discuția interfeței cu utilizatorul (secțiunea 2). Ultimul argument reprezintă ieșirea funcției, soluția turului eulerian, împreună cu reprezentarea grafică a străzilor din oraș și alte informații statistice. Codul funcției este prezentată complet în Anexa 5.

În următoarele explicații, este descris succint implementarea funcției, alături de elementele adiționale care contribuie la completitudinea componentei.

Mai întâi, se marchează începutul etapei de extragere al grafului (`roadExtractionEvent.set()`), urmat de apelarea funcției de extragere al grafului orașului cu numele `settlement_name` din fișierul `source_filename` (componenta descrisă în subsecțiunea 1.1). Apoi se găsesc nodurile izolate prin funcția `find_isolated_nodes`, aprofundată în subsecțiunea 1.2 pentru ca pe urmă să fie șterse prin codul 3.6, urmat de o calibrare al grafului, dacă s-au găsit noduri izolate (codul 3.5).

Astfel se termină etapa de extragere prin marcarea următoarei: cea de procesare al turului eulerian, marcat prin setarea event-ului `routeProcessingEvent`. Aici se verifică dacă graful obținut este într-adevăr eulerian, dacă nu are nici un nod de grad impar, conform teoremei 1.2:

```

1  for node in nodes_dict.keys():
2      if nodes_dict[node]['grade'] % 2 == 1:
3          is_eulerian = False
4          break

```

```

5
6  print(f'Is Eulerian? Response: {is_eulerian}')
```

Listing 3.13. Codul de verificare dacă un graf este eulerian

În cazul în care graful nu este eulerian, începe procesarea grafului pentru a deveni așa ceva. Inițial, se identifică nodurile de grad impar din graf, iar cele detectate sunt adăugate într-un dicționar:

```

1  # se determina nodurile de grad impar, adaugându-le într-un dicționar
   → (se vor pastra aici si vectorii de
2  # costuri minime, respectiv de precedență a drumurilor între
   → celelalte noduri)
3  odd_nodes_dict = {}
4  for node_index in nodes_dict.keys():
5      if nodes_dict[node_index]['grade'] % 2 == 1:
6          odd_nodes_dict[node_index] = {'info': nodes_dict[node_index]}
```

Listing 3.14. Detectarea nodurilor de grad impar

Acesta contribuie la obținerea mulțimii de drumuri minime între fiecare pereche de noduri de grad impar, prin intermediul codului 3.10. În cadrul obținerii mulțimii, există varianta de a folosi algoritmul Floyd-Warshall în loc de algoritmul Dijkstra:

```

1  fd_or_djk = False # daca se aplica floyd-warshall sau dijkstra
2
3      if fd_or_djk:
4          new_nodes_list, dist_nodes, next_node_matrix =
           → floyd_warshall_alg(nodes_dict, edges_dict)
5
6          node_index_dict = {graph_node: index for index,
           → graph_node in enumerate(new_nodes_list)}
7          nodes_dict = {node_index_dict[key]: value for key, value
           → in nodes_dict.items()}
8          edges_dict = {(node_index_dict[edge[0]],
           → node_index_dict[edge[1]]): value
           for edge, value in edges_dict.items()}
9
10
11         for node_i in nodes_dict.keys():
12             nodes_dict[node_i]['dist_nodes'] = dist_nodes[node_i]
13             nodes_dict[node_i]['next_nodes'] =
           → next_node_matrix[node_i]
14
15             if node_i in odd_nodes_dict.keys():
16                 odd_nodes_dict[node_i]['dist_nodes'] =
           → dist_nodes[node_i]
17         else:
18             # algoritmul lui Dijkstra paralel, conform codului 3.9
```

Dicționarul rezultat, cât și cel de noduri de grad impar sunt puse în aplicare în funcția *greedy_eulerian_graph*, explicată în subsecțiunea 1.4, care va modifica graful curent pentru a deveni eulerian. La final se mai verifică încă o dată autenticitatea soluției, folosind codul 3.13.

Indiferent dacă graful este eulerian sau nu, în urma prelucrării eventuale al grafului, se obține soluția sistemului, din care se determină turul eulerian în funcția *hierholzer_alg* (codul 6 din Anexa 4). Prin urmare se încheie etapa de prelucrare a soluției.

Ultima etapă din componenta principală, marcată prin setarea event-ului *animationProcessingEvent*, este cea de construire al animației circuitului eulerian, care va fi furnizată interfeței cu utilizatorul, prin argumentul *solution_content*.

La început, se crează o reprezentare mai facilă pentru GUI al grafului eulerian, folosind modulul *networkx(nx)*, prin obiectul *nx.Graph()*, la care se adaugă toate nodurile și muchiile cu costurile lor din graf:

```

1  # metodă preluată din
   ↪ http://brooksandrew.github.io/simpleblog/articles
2  # /intro-to-graph-optimization-solving-cpp/
3  # creare graf gol
4  g = nx.Graph()
5
6  # adaugarea muchiilor(strazilor) grafului specific orasului
7  for edge, val in solution_edges.items():
8      node1, node2 = edge
9      g.add_edge(node1, node2, d=val['d'])
10
11  for node, val in solution_nodes.items():
12      nx.set_node_attributes(g, {node: {'lat': val['lat'], 'lon':
   ↪ val['lon']}}})

```

Listing 3.15. codul de creare al reprezentării în GUI al grafului

Odată construit modelul, se poate trece la asamblarea soluției: graful, soluția turului eulerian, date statistice - lungime totală străzi, costul suplimentar (dacă graful nu a fost înainte eulerian), distanța totală tur și lista de distanțe crescânde pe parcursul deplasării dronei în tur. Codul de mai jos relevă calculul datelor suplimentare și a construcției dicționarului *solution_content*.

```

1  solution_content['graph'] = g
2  solution_content['solution'] = solution
3  solution_content['lungime_totala_strazi'] = 0
4  solution_content['distanța_parcursa_suplimentar'] = 0
5  solution_content['distanța_totală_tur'] = 0
6  solution_content['distanța parcursa drona curentă'] = [0]
7
8  for strada in solution_edges.keys():
9      solution_content['lungime_totala_strazi'] +=
   ↪ solution_edges[strada]['d']
10
11  solution_content['lungime_totala_strazi'] =
   ↪ round(solution_content['lungime_totala_strazi'], 3)
12
13  total_sol_length = 0
14  for node1, node2 in zip(solution[:-1], solution[1:]):
15      if (node1, node2) in solution_edges.keys():
16          edge = (node1, node2)
17      else:
18          edge = (node2, node1)
19

```

```
20     solution_content['distanța parcursă drona curentă'].append(  
21         round(total_sol_length + solution_edges[edge]['d'], 3))  
22     total_sol_length += solution_edges[edge]['d']  
23  
24     solution_content['distanța totală tur'] = round(total_sol_length, 3)  
25     solution_content['distanța parcursă suplimentară'] =  
    ↪ round(solution_content['distanța totală tur'] -  
    ↪ solution_content['lungime totală străzi'], 3)  
26  
27     stopEvent.set()
```

Listing 3.16. Construirea dicționarului soluției plus alte date pentru trimiterea lui în GUI

Conform codului 3.16, lungimea totală a străzilor se obține din suma tuturor costurilor muchiilor din graf, dacă ar fi vizitate o singură dată (liniile 8-11). Lista de distanțe progresive, cât și cea totală a turului se calculează în tandem, parcurgând soluția turului eulerian (liniile 13-24). În cele din urmă, costul suplimentar al turului, în urma multiplicării muchiilor grafului în componenta 1.4, se calculează din diferența între costul total al circuitului eulerian și costul total al străzilor orașului (linia 25). Etapa se încheie cu marcarea event-ului *stopEvent* (linia 27).

Rezultatul componentei principale va fi furnizat GUI-ului, care va procesa informațiile primite spre a prezenta animat soluția. Acest procedeu, pe lângă implementarea acestuia, vor fi prezentate imediat.

2. Implementarea interfeței cu utilizatorul (GUI)

Modelul interfeței a fost evidențiată în capitolul 2, secțiunea 2.6. Din ultima schemă prezentată s-a implementat GUI-ul, al cărui cod este afișat în Anexa 6, listing-ul 8, iar ca vizualizare se vede în imaginea de mai jos. Atât interfața, cât și ferestrele aferente aplicației au fost realizate în aplicația Qt Designer, iar apoi transformat în cod python prin comanda `pyuic5 <nume fisier ui> -o <nume fisier Python>`. Funcționalitatea interfeței a fost realizată în clasa *Aplicație Interfață*, a cărei implementare este prezentată în Anexa 6, listing-ul 10.

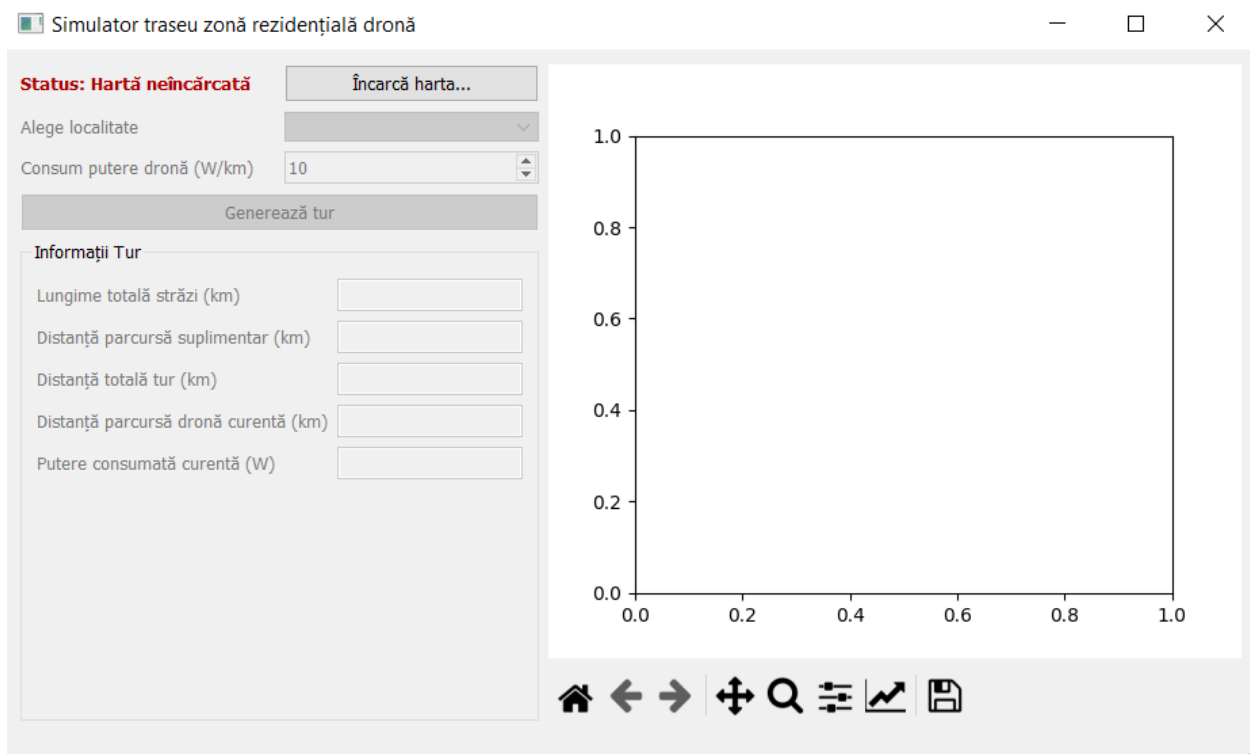


Figura 3.1. Conținutul interfeței cu utilizatorul, când pornește programul prima dată

Statusul va determina utilizatorul să încarce harta dorită apăsând butonul „Încarcă harta...”, din care se va deschide o fereastră de selecție a unui fișier tip .osm. Odată selectat unul, celelalte elemente de interfață, în afară de rubrica „Informații tur”, vor fi deblocate, și se poate selecta un oraș dorit din ComboBox, plus precizarea consumului dronei. Statusul se va schimba corespunzător pentru a marca încărcarea unei hărți.

După ce s-a stabilit și din ce oraș din hartă va fi vizitat de dronă, se apasă butonul „Generează tur”, care va apela într-un thread separat componenta de prelucrare și extragere al circuitului eulerian, descris în subsecțiunea 1.6. În timpul operației, se va afișa o fereastră tip „splash” care va anunța utilizatorul în ce stadiu se află prelucrarea turului. În timpul procesului, fereastra principală nu poate fi accesată. Codul clasei ferestrei de încărcare este *LoadingScreen*, al cărei formă este redată în Anexa 6, listing-ul 9.

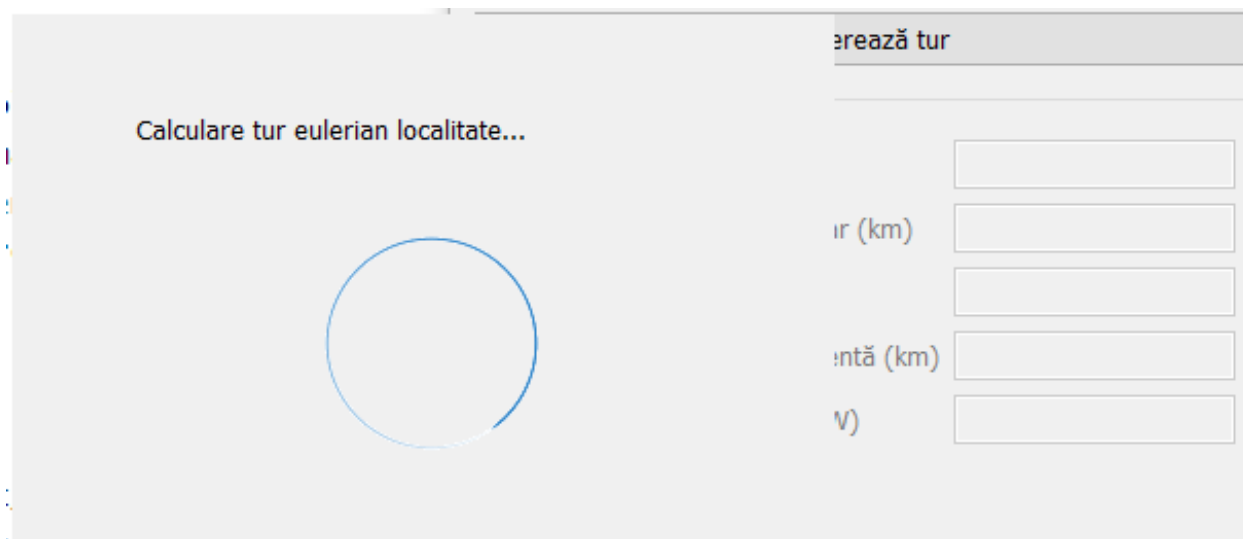


Figura 3.2. Fereastra de încărcare din timpul procesului de prelucrare al turului eulerian

La finalul prelucrării, se afișează animația circuitului eulerian pe străzile zonei rezidențiale detectate, împreună cu datele statistice, unele din ele (distanța curentă parcursă și consumul actual al dronei) fiind actualizate în timp real.

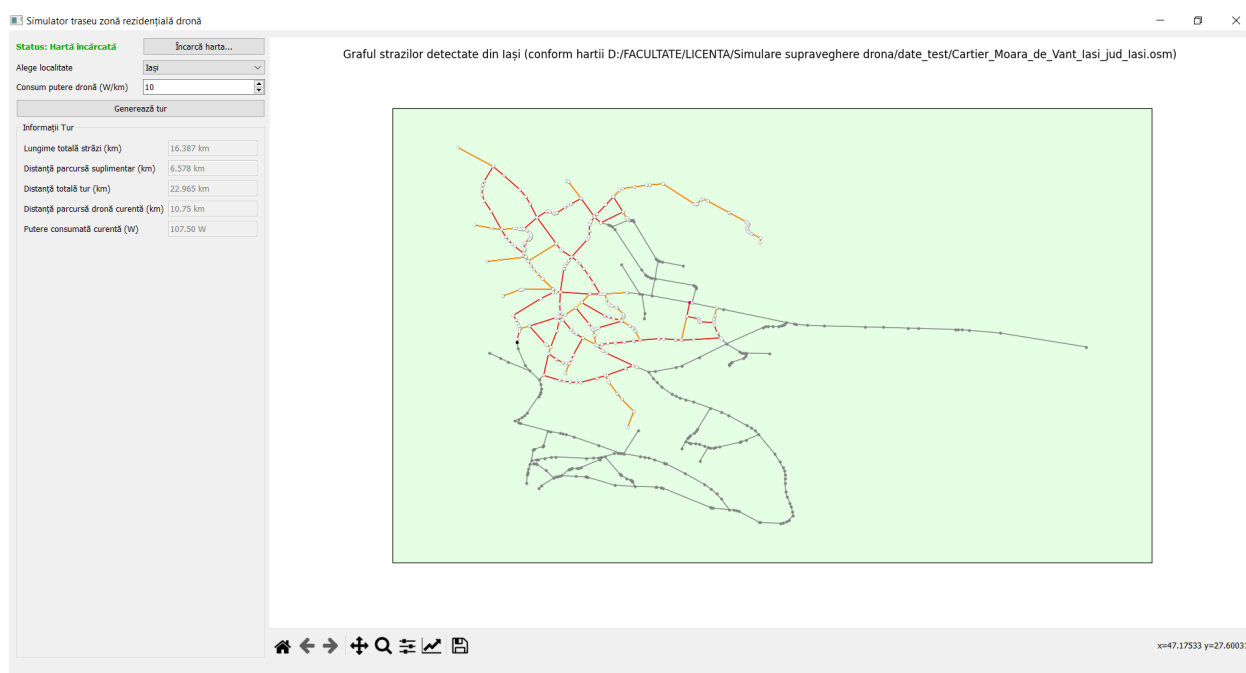


Figura 3.3. Afișarea animației turului eulerian pentru orașul detectat în hartă; fereastră maximizată pentru evidențierea clară a rezultatului

Muchiile vizitate de mai multe ori vor avea inițial culori calde, ajungând la culori din ce în ce mai reci cu cât sunt parcurse mai des. Nodurile albe sunt intersecțiile prin care drona a trecut deja, în timp ce nodul roșu este punctul de plecare în turul eulerian al orașului, iar punctul negru este poziția curentă a dronei. Liniile și vârfurile gri reprezintă zone nevizitate încă în turul eulerian, și se vor colora corespunzător de-a lungul animației. La finalul acesteia, punctele roșu, respectiv negru se vor suprapune, marcând finalul circuitului și afișarea consumului total al dronei.

Capitolul 4. Testarea aplicației și rezultate experimentale

1. Elemente de configurare și instalare

Având în vedere că aplicația este realizată prin intermediul limbajului de programare *Python*, versiunea 3.9, este necesară instalarea pachetelor necesare interpretorului *Python 3*. De asemenea, este necesară instalarea tuturor pachetelor utilizate de-a lungul programului, folosind comanda *pip* din interpretorul Python sau din linia de comandă al sistemului de operare. Enumerăm mai departe pachetele necesare pentru funcționarea și testarea corectă a aplicației:

- Modulul *PyQt5* pentru funcționalitățile GUI-ului și elementelor acestuia;
- Modulele *sys*, *threading*, *time* care sunt implicit instalate cu modulul Python;
- Modulul *xml* pentru prelucrarea fișierelor XML
- Modulul *networkx* cu scopul reprezentării grafului orașului dorit;
- Modulul *matplotlib* pentru reprezentarea grafică a străzilor și intersecțiilor orașului;
- Modulul *concurrent.futures* cu ajutorul căruia se implementează paradigma multi-processing din prelucrarea drumurilor;
- Modulele *math*, *queue*, *random* prin care se realizează operațiile matematice, respectiv permite implementarea de cozi, ultimul ajută la selectarea aleatorie a elementelor din liste;
- Modulul *itertools* cu ajutorul căruia s-a permis asignarea listelor de mai multe ori în procese;
- Modulul *numpy* pentru calcule matematice avansate, dar mai ales pentru gestiunea și manipularea eficientă a listelor.

Lansarea aplicației constă în rularea scriptului *AplicațiaPrincipala.py* folosind comanda *python AplicațiaPrincipala.py* sau *python3 AplicațiaPrincipala.py* (dacă în sistem este instalat și interpretorul Python2).

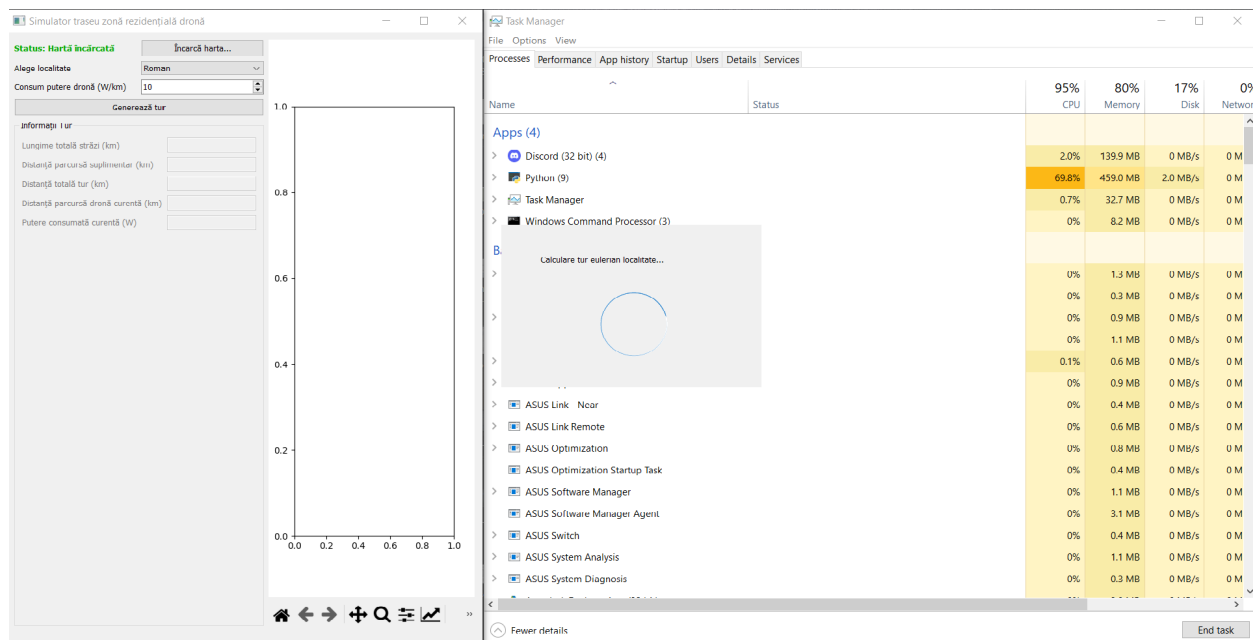
2. Limitări în procesarea datelor din fișier

De-a lungul testelor realizate pe aplicație, s-a detectat o problemă în ceea ce privește procesarea datelor dintr-un fișier *.osm*, unde sunt identificate într-o zonă foarte multe noduri. Dacă sunt pornite în timpul aplicației înregistrări video live, în special în platformele de socializare sau discuții în echipă (Microsoft Teams, Google Meets, Facebook, Discord, etc.), mai multe procese sunt folosite pentru a susține video-urile, iar în consecință mai puține procese sunt utilizate în prelucrarea drumurilor minime, cu ajutorul algoritmului lui Dijkstra. Viteza de procesare scade semnificativ, ceea ce ar duce la timpi de așteptare îndelungate pentru a finaliza operațiile.

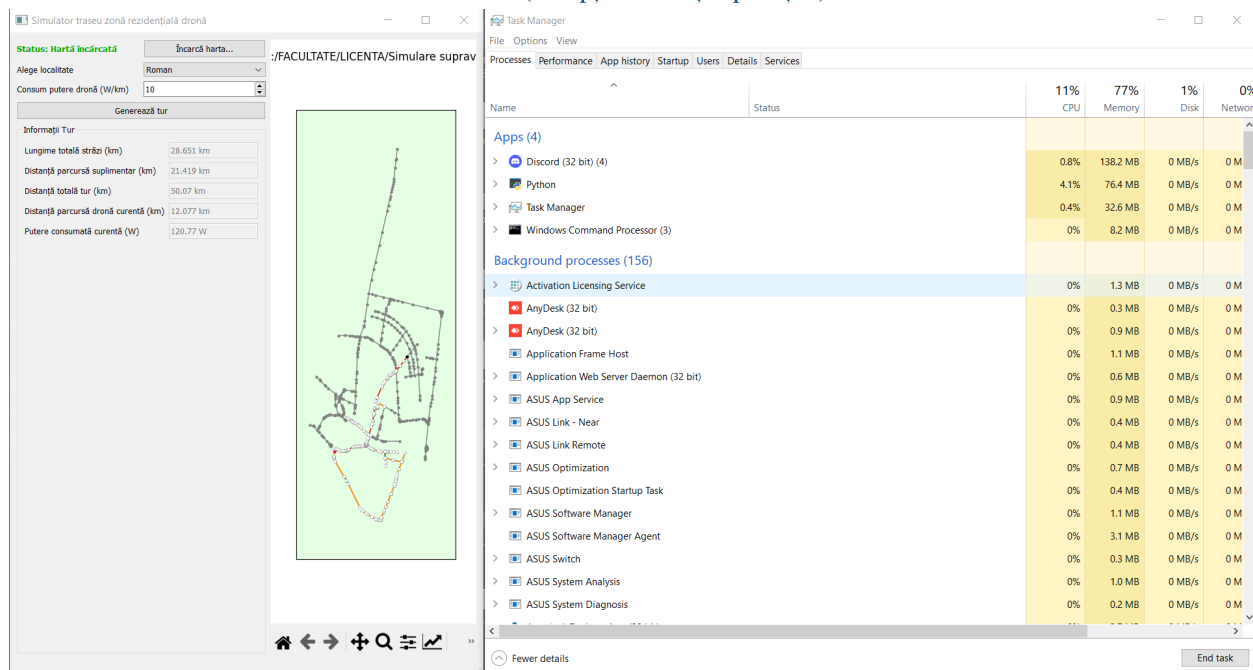
Aceasta este limitarea limbajului Python, care nu permite exploatarea paradigmei multi-threading, unde ar fi fost mult mai utilă în prevenirea unei astfel de situații. Thread-urile consumă mult mai puțină memorie și lucrează mult mai rapid decât procesele, iar în plus procesul este format, până la urmă, din mai multe thread-uri.

3. Limitări din punct de vedere al operațiilor multitasking

În timpul procesării unui tur eulerian pentru graful unui oraș dat, toate procesele din sistemul de operare sunt folosite pentru a calcula mult mai eficient soluția. Dar consecința acestui fapt este lipsa de control al celorlalte operații din sistemul de operare, inclusiv interacțiunea cu interfața acestuia. Este încurajat în timpul replicării testelor prezentate mai departe să nu se interacționeze cu niciun element GUI sau terminal din OS, în timpul ferestrei de încărcare.



(a) După cum se vede în task manager, aplicația Python ajunge să ocupe deja 69% din capacitatea procesorului. După realizarea capturii, a ajuns la un procent de 97%, fiind imposibil de interacționat cu elementele de interfață ale OS-ului (excepție interfața aplicației).



(b) După terminarea procesării, aplicația a ajuns la o capacitate de doar 4% (pentru thread-urile active ale aplicației)

Figura 4.1. Capturi de ecran care relevă problema performanței în multitasking

4. Testarea aplicației

4.1. Măsurarea performanței

Din punct de vedere al performanței, aplicația va fi măsurată din 3 puncte de vedere: din punct de vedere al algoritmului folosit în prelucrarea drumurilor minime (Warshall vs Dijkstra paralel), după performanța totală a aplicației pentru diferite dimensiuni ale grafului diferitor orașe, cât și compararea vitezei programului între sistemele de operare Windows și Linux.

Ca benchmark, se va folosi pentru toate testele Laptop-ul ASUS VivoBook K513EA, cu sistem de operare Windows 10 pe 64 biți, ale cărui specificații sunt prezentate în Capitolul 2, secțiunea 1. Pentru ultimul test de performanță se va utiliza o mașină virtuală în care se va rula sistemul de operare Linux, împărțind memorie cu sistemul de operare din laptop (4GB fiecare).

Pentru a testa performanțele, a fost folosit modulul *time*, funcția *time()* pentru a obține diferența de timp între momentul de început și cel de sfârșit al rulării unei secvențe de cod. Un exemplu al procedurii de calcul al timpului unui program este oferit mai jos:

```
1 import time
2
3 start_time = time.time()
4 for i in range(100000):
5     print(i)
6 end_time = time.time()
7 print(f'Execution: {(end_time - start_time)} seconds')
```

Listing 4.1. Exemplu de calcul al timpului de rulare al unui program

4.1.1. Testul de performanță al algoritmilor de drumuri minime

În acest test, algoritmii Floyd-Warshall și Dijkstra sunt puși umăr la umăr pentru a demonstra performanțele lor pe diferite fișiere tip .osm, cu scopul măsurării rapidității de a calcula distanțele minime între fiecare pereche de noduri dintr-un graf. Dijkstra va fi rulat paralel, unde va calcula doar nodurile de grad impar. Deși acest lucru determină ca metoda specificată să fie mai rapidă decât varianta Floyd-Warshall, această verificare ajută la înțelegerea alegerii algoritmului Dijkstra în acest context și de câte ori este mai rapid decât cealaltă metodă.

S-au analizat performanțele codurilor 3.9 pentru Dijkstra paralel, respectiv 3.7 secvențial. Mai jos este prezentat tabelul cu rezultatele algoritmilor pentru mai multe fișiere .osm de diverse dimensiuni.

Index	Nume fișier	Oraș analizat
1	Campus Tudor Vladimirescu, Iasi, jud Iași.osm	Iași
2	Zona Republicii-Anton_Pann-Mihai_Viteazul-Primaverii, Roman, jud Neamt.osm	Roman
3	Cartier_Moara_de_Vant_Iasi_jud_Iași.osm	Iași
4	Zona Roman, jud Neamt.osm	Roman
5	Zona Decebal-Bistritei-Petru_Rares-Orhei, Piatra-Neamt, jud Neamt.osm	Piatra-Neamț

Index	Număr noduri detectate	Număr muchii detectate	Durată Floyd-Warshall (s)	Durată Dijkstra paralel (s)
1	32	31	0,012	1,488
2	121	124	0,716	2,377
3	489	520	49,844	8,996
4	561	583	75,737	11,377
5	915	944	359,144	53,288

Tabelul 4.1. Rezultate performanțe algoritmi Floyd-Warshall și Dijkstra paralel

Rezultatele arată că algoritmul Floyd-Warshall este util pentru grafurile de mici dimensiuni (până la 200 noduri), fiind mai rapid decât algoritmul Dijkstra paralel. În schimb eficacitatea primei metode menționate scade exponențial, ajungând în primul exemplu de la o diferență de timp de 124 ori, la al doilea exemplu cu o diferență de aproximativ 3 ori. Odată ce numărul de noduri depășește 400, metoda Dijkstra paralelă își vede simțită prezența, fiind mai rapidă în al treilea exemplu de 5 ori, iar în ultimele 2 eșantioane de aproximativ 7 ori. În general, Dijkstra este algoritmul potrivit pentru analiza unor grafuri de dimensiuni medii și mari, în timp ce Floyd-Warshall ajută la prelucrarea cu o viteză majoră a grafurilor de dimensiuni mici.

4.1.2. Testul de performanță al aplicației în funcție de dimensiunea grafurilor analizate

O parte din rezultate se pot vedea în tabelul 4.1 pentru algoritmul Dijkstra paralel, dar în acest test se va măsura capabilitatea aplicației pentru dimensiuni mult mai mari ale grafurilor. Până acum, au fost folosite doar fișiere care conțin în graful orașelor sub 1000 noduri, iar acest test va demonstra capabilitatea aplicației în prelucrarea grafurilor cu peste 1000 noduri. Deoarece în această situație, în website-ul OpenStreetMap nu pot fi preluate mai mult de 50000 noduri (indiferent dacă sunt noduri din străzi sau clădiri), s-a folosit API-ul Overpass, accesând-ul direct de pe site. Mai jos se este evidențiat tabelul cu rezultatele aferente.

Index	Nume fișier	Oraș analizat
1	Zona Roman, jud Neamt.osm	Roman
2	Zona Decebal-Bistritei-Petru_Rares-Orhei, Piatra-Neamt, jud Neamt	Piatra-Neamt.osm
3	Centrul Constantei.osm	Constanța
4	Braila.osm	Brăila
5	Piatra-Neamt.osm	Piatra-Neamț
6	Zona Canta-Pacurari-Centru-Alexandru_cel_Bun-Dacia-Copou.osm	Iași

Index	Număr noduri detectate	Număr muchii detectate	Durată Dijkstra paralel (s)
1	561	583	11,377
2	915	944	53,288
3	1218	1362	219.166
4	1801	1989	657.785
5	2348	2466	1105.414
6	2758	2902	1914.305

Tabelul 4.2. Rezultate performanțe algoritm Dijkstra paralel pentru fișiere de dimensiuni mai mari

Din durata prelucrărilor hărților relevă faptul că aplicația este cel mai bine utilizată doar pentru cartiere sau zone rezidențiale de nivel mediu din orașe (sub 1000 noduri). Când numărul de muchii și noduri depășește numărul 1000, durata procedurii crește semnificativ: Harta cu index 3, reprezentând centrul Constanței, durează în jur de 4 minute pentru a fi procesată complet de program, în timp ce orașele cu un număr foarte mare de străzi mici (reprezentate de un număr peste 2000 de noduri și muchii), cum sunt hărțile cu index 4,5,6 ajung să dureze până la 30 minute! Este un timp de așteptare extrem de lung pentru hărți de asemenea dimensiuni, de aceea se recomandă folosirea hărților de mici dimensiuni.

4.1.3. Testul de performanță al aplicației în funcție de sistemul de operare

Reamintim că în Linux se pot folosi un număr nelimitat de procese, în timp ce în sistemul de operare Windows numărul acestora este limitat la 61 care pot fi utilizate de aplicații Python. Considerând faptul că rapiditatea aplicației ține mai mult de cât de repede sunt procesate drumurile minime, iar în acest consens s-au folosit procese, s-a decis testarea acestora pe fiecare OS menționat. Algoritmul de procesare folosit în ambele sisteme este Dijkstra paralel.

Pentru Linux, s-a folosit programul Oracle VM VirtualBox pentru a rula în mașină virtuală sistemul de operare Ubuntu Desktop 22.04.1 din familia Linux, căruia s-au alocat 40GB memorie

internă, cât și 4GB de memorie RAM pentru a egala cea rămasă de pe Windows.

Fișierele XML OSM testate sunt aceleași ca în testul de performanță între algoritmi Floyd-Warshall și Dijkstra paralel, ale cărei rezultate sunt evidențiate în tabelul 4.1.

Odată stabilite datele de test, s-a trecut la măsurarea performanței, iar în tabelul următor sunt afișate rezultatele aferente, atât pentru Windows, cât și pentru Linux:

Index	Nume fișier	Oraș analizat
1	Campus Tudor Vladimirescu, Iasi, jud Iasi.osm	Iași
2	Zona Republicii-Anton_Pann-Mihai_Viteazul-Primaverii, Roman, jud Neamt.osm	Roman
3	Cartier_Moara_de_Vant_Iasi_jud_Iasi.osm	Iași
4	Zona Roman, jud Neamt.osm	Roman
5	Zona Decebal-Bistritei-Petru_Rares-Orhei, Piatra-Neamt, jud Neamt.osm	Piatra-Neamț

Index	Număr noduri detectate	Număr muchii detectate	Durată Linux (s)	Durată Windows (s)
1	32	31	0,044	1,488
2	121	124	0,545	2,377
3	489	520	0,571	8,996
4	561	583	44,686	11,377
5	915	944	227,286	53,288

Tabelul 4.3. Rezultate performanțe algoritm Dijkstra paralel pentru OS-urile Linux, respectiv Windows

Surprinzător, rezultatele obținute pentru Linux au fost asemenea algoritmului Floyd-Warshall, fiind puțin mai rapid în exemplele cu noduri mai multe. În plus a fost găsit o problemă majoră în OS-ul specificat: când în program este inserat un alt fișier, și odată începută generarea unui alt tur, apare eroarea *segmentation fault*, cauzată de regulă de accesarea memoriei care nu aparține programului. Sursa acestei erori încă nu a fost găsită, deci este recomandată rularea aplicației pe sistemele de operare Linux o singură dată, până se rezolvă ulterior această problemă.

Concluzii

Concluzii principale

În urma implementărilor prezentate în capitolul 3, și experimentelor executate în capitolul 4, aplicația este funcțională deocamdată doar în sistemul de operare Windows, spre analizarea orașelor medii sau mici, sau a zonelor rezidențiale din metropole. Algoritmul Dijkstra rulat paralel pentru calculul drumurilor minime, împreună cu metoda Hierholzer pentru determinarea drumului eulerian în graful detectat din zona rezidențială asigură timpi de execuție rapizi - până la maxim 1 minut pentru un graf de aproximativ 1000 noduri (conform experimentului din capitolul 4, subsecțiunea 4.1.1).

În cazurile grafurilor mai mici, algoritmul Floyd-Warshall este cel mai pretabil în prelucrarea drumurilor minime în acestea, dar suferă performanțe slabe pentru grafuri cu un număr mai mare de 500 noduri. Prin urmare metoda Dijkstra este cea mai pretabilă în prelucrarea particulară a drumurilor minime, pentru un număr restrâns de puncte de plecare.

OpenStreetMap s-a dovedit o resursă foarte bună pentru hărți elaborate ale localităților din lume, comparabilă cu informațiile furnizate de API-uri specializate, cum ar Google Maps API. Pe lângă faptul că acestea sunt actualizate în fiecare săptămână de o comunitate semnificativă de cartografi, utilizatori obișnuiți sau prin alte resurse informaționale utile, și faptul că oricine le poate accesa oricând și oriunde în mod gratuit, oferind rezultatul în formatul XML OSM ușor de citit și prelucrat, website-ul, alături de API-ul Overpass oferit de acesta, a contribuit mult la preluarea datelor de intrare pentru aplicația curentă.

Obiective modificate și/sau nerealizate

Acest proiect a avut ca scop principal prelucrarea cât mai rapidă a turului eulerian prin străzile unui oraș, dar în pofida acestui obiectiv, a intervenit problema folosirii prea multor procese, care cauzează „înghețarea” tuturor elementelor de interfață ale sistemului de operare, în afară de cea pe care rulează aplicația. Din folosirea limbajului Python pentru paradigma multi-processing, s-a învățat faptul că trebuie avut mare grijă de câte procese rulează în fundal, pentru a nu întâmpina din nou situația precizată mai sus.

Fiind în stadiu avansat aplicația când s-a descoperit această problemă, era consumatoare de timp conversia codului sursă din Python într-un alt limbaj care gestionează mult mai bine programarea paralelă (cum ar fi C/C++). De aceea s-a păstrat programul în acest stadiu, însă rămâne ca obiectiv viitor conversia în alt limbaj de programare.

Grafurile nu conțin toate străzile unei zone rezidențiale, datorită modului de preluare al acestora: inițial, orașele erau detectate în fișierul XML OSM prin găsirea nodurilor care întru-chipează centrul localității respective. Însă s-a descoperit pe parcursul experimentelor și testelor realizate în aplicație cazul în care se preia un cartier din oraș, dar în hartă nu este inclus și centrul. În acest caz, nu se „găsea” nici o localitate, iar pe interfață combo-box-ul localităților era gol. De aceea s-a decis căutarea orașelor în funcție de străzile a căror etichetă marchează din ce localitate fac parte, permițând în consecință detecția orașelor. Chiar și așa, pot exista străzi care nu au mai fost etichetate de comunitatea OSM drept membre ale unei localități, deci sunt ignorate de funcția de generare al grafului.

Posibile direcții de dezvoltare

Cum a fost precizat mai sus, principalele direcții de dezvoltare sunt conversia aplicației în cod C/C++ pentru a putea lucra mult mai bine cu programarea paralelă, mai ales că aici se pot folosi și thread-uri, care sunt mult mai rapide și mai puțin consumatoare de memorie decât procesele. Această conversie va extinde aplicabilitatea proiectului și în orașe mai întinse.

De asemenea, este necesară o regândire a modului de preluare a străzilor în graful zonei rezidențiale, astfel încât și străzile neetichetate ca apartenente pe hartă să fie și ele incluse. În plus, pentru a nu aglomera graful cu multe noduri care să încetinească prelucrarea turului eulerian, se propune pe viitor un algoritm de detecție al nodurilor ce intersectează străzile unui oraș, fără a mai lua de seamă nodurile intermediare, care de regulă formează curbe.

Fiind un proiect pe care s-a construit o soluție a poștaşului chinez, aceasta poate avea extindere pe aplicații de rutare: de la găsirea rutelor eficiente pentru autobuzele de transport public (asemenea problemei poștaşului chinez din cicluri de lungime k , pomenit în capitolul 1, secțiunea 3), până la programarea transportului de mărfuri în mai multe orașe, unde necesită să treacă prin șoselele dintre localitățile majore pentru a aduce marfă magazinelor din satele ce trec prin rută.

Bibliografie

- [1] T. I. BV, “Bucharest traffic,” https://www.tomtom.com/en_gb/traffic-index/bucharest-traffic, 2022, Ultima accesare: 14.08.2022.
- [2] J. Bondy and U. Murty, *Graph Theory with Applications*. Elsevier Science Publishing Co., Inc., The Macmillan Press Ltd., 1976. [Online]. Available: http://www.maths.lse.ac.uk/Personal/jozef/LTCC/Graph_Theory_Bondy_Murty.pdf
- [3] J. Edmonds and E. L. Johnson, “Matching, euler tours and the chinese postman,” *Mathematical Programming*, vol. 5, pp. 88–124, 1973. [Online]. Available: <https://web.eecs.umich.edu/~pettie/matching/Edmonds-Johnson-chinese-postman.pdf>
- [4] B. Moret and H. Shapiro, *Algorithms from P to NP: Design & efficiency*. Addison-Wesley, 1991, no. vol. 1. [Online]. Available: <https://www.amazon.com/Algorithms-NP-Vol-Design-Efficiency/dp/0805380086>
- [5] C. Adrian-Ioan, “Proiect disciplina rețele de calculatoare (anul iii, semestrul 1),” <https://github.com/Chihalau-Adrian-Ioan/Proiect-RC>, 2020, Ultima accesare: 21.08.2022.
- [6] —, “Proiect disciplina sisteme cu multiprocesoare (anul iii, semestrul 2),” <https://www.hackster.io/chihalau-adrian-ioan/atv-robot-buggy-426ae8>, 2021, Ultima accesare: 21.08.2022.

Anexe

Anexa 1. Funcția de extragere a grafului unui oraș aflat în harta stocată într-un fișier

```

1  import xml.etree.ElementTree as ET
2  import math
3
4  def get_city_graph_from_map(filename, city_name):
5      # prelucrarea informatiilor hartii pentru a prelua graful
6      ↪ acestuia
7      tree = ET.parse(filename)
8      root = tree.getroot()
9      node_dict = {}
10
11     # extragerea nodurilor din graf
12     # Nota: nu toate nodurile reprezinta intersectii, unele pot fi
13     ↪ chiar noduri ce determina
14     # colturile unui parc sau a unei cladiri
15     for node in root.iter('node'):
16         node_dict[node.attrib['id']] = {'lat':
17             ↪ float(node.attrib['lat']), 'lon':
18             ↪ float(node.attrib['lon'])}
19
20     street_nodes_dict = {}
21     streets_dict = {}
22     graph_edges_dict = {}
23
24     # extragerea strazilor din graf
25     # in formatul osm, strazilor fac parte din tag-urile way (căi),
26     ↪ care pot reprezenta si perimetrele unor cladiri
27     for child in root.findall('way'):
28         highway, name, in_city, nodes = None, (), None, []
29
30         # De aceea, se cauta căile care sunt străzi principale
31         ↪ (drumuri nationale, europene, autostrazi),
32         # secundare, tertiare si rezidentiale (strazi din oras), care
33         ↪ sunt catalogate aparținând orașului dorit
34         for tag in child.iter('tag'):
35             if tag.attrib['k'] == 'highway' and tag.attrib['v'] in
36                 ↪ ['primary', 'secondary', 'tertiary', 'residential']:
37                 highway = tag.attrib['v']
38             elif tag.attrib['k'] == 'is_in:city':
39                 in_city = tag.attrib['v']
40             elif tag.attrib['k'] == 'name':
41                 name = tag.attrib['v']
42
43         # daca a fost gasita o astfel de strada
44         if in_city == city_name and highway is not None:
45             # toate nodurile din acesta sunt catalogate drept noduri
46             ↪ (intersectii) ce apartin grafului orasului
47             for node in child.iter('nd'):
48                 node_id = node.attrib['ref']

```

```

40         if node_id not in street_nodes_dict.keys():
41             street_nodes_dict[node_id] =
42                 ↳ node_dict[node_id].copy()
43             street_nodes_dict[node_id]['grade'] = 0
44             nodes.append(node.attrib['ref'])
45         if child.attrib['id'] not in streets_dict.keys():
46             streets_dict[child.attrib['id']] = {'name': name,
47                 ↳ 'highway': highway, 'nodes': nodes}
48
49 # totusi, trebuie determinate distantele intre fiecare
50 ↳ intersectie din oras
51 # se foloseste formula haversina, ce calculeaza distanta intre 2
52 ↳ puncte aflate pe glob
53 # formula preluată din
54 ↳ http://www.movable-type.co.uk/scripts/latlong.html
55 R = 6371 # raza Pământului, în kilometrii
56 for street in streets_dict.keys():
57     nodes = streets_dict[street]['nodes']
58     for node1, node2 in zip(nodes[:-1], nodes[1:]):
59         lat1, lon1 = street_nodes_dict[node1]['lat'],
60             ↳ street_nodes_dict[node1]['lon']
61         lat2, lon2 = street_nodes_dict[node2]['lat'],
62             ↳ street_nodes_dict[node2]['lon']
63
64 # se convertesc coordonatele in radiani
65 phi1, phi2 = lat1 * math.pi / 180, lat2 * math.pi / 180
66 lambda1, lambda2 = lon1 * math.pi / 180, lon2 * math.pi /
67     ↳ 180
68
69 diff_lambda = lambda2 - lambda1
70 diff_phi = phi2 - phi1
71
72 # se foloseste formula haversina, ce calculeaza distanta
73 ↳ intre 2 puncte aflate pe glob
74 # a = patrutul jumatatii din lungimea arcului cercului
75 ↳ dintre 2 puncte
76 # c = distanta unghiulara in radiani
77 # d = distanta obtinuta
78 a = math.sin(diff_phi / 2) ** 2 + math.cos(phi1) *
79     ↳ math.cos(phi2) * (math.sin(diff_lambda / 2) ** 2)
80 c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
81 d = R * c
82
83 # pe langa adaugarea muchiei in graf, se maresc si
84 ↳ gradele celor 2 noduri
85 graph_edges_dict[(node1, node2)] = d
86 street_nodes_dict[node1]['grade'] += 1
87 street_nodes_dict[node2]['grade'] += 1
88
89 # se simplifică notațiile nodurilor și muchiilor incidente prin
90 ↳ asignarea fiecăruia cu câte un număr de ordine
91 node_index_dict = {graph_node: index for index, graph_node in
92     ↳ enumerate(list(street_nodes_dict.keys()))}

```

```
79 indexed_nodes_dict = {node_index_dict[key]: value for key, value
    ↪ in street_nodes_dict.items()}
80 indexed_edges_dict = {(node_index_dict[edge[0]],
    ↪ node_index_dict[edge[1]]): value
81                       for edge, value in
    ↪ graph_edges_dict.items()}
82
83 return indexed_nodes_dict, indexed_edges_dict
```

Listing 2. Codul funcției *get_city_graph_from_map*

Anexa 2. Funcția de căutare al nodurilor izolate dintr-un graf

```

1  import queue
2  import numpy as np
3
4  # Folosind algoritmul de căutare în adâncime, caută insulele din graf
   ↳ (subgrafuri care nu sunt conectate între ele)
5  # si returnează nodurile care nu fac parte din insula cu cele mai
   ↳ multe noduri, adică nodurile izolate
6  # metodă bazată pe rezolvarea din https://en.wikipedia.org/wiki/Breadth-first\_search#:~:text=procedure%20BFS(, enqueue(w)
7  # wiki/Breadth-first_search#:~:text=procedure%20BFS(, enqueue(w)
8  def find_isolated_nodes(nodes_dict: dict, edges_dict: dict):
9      nodes_count = len(list(nodes_dict.keys()))
10     marked_island_nodes = np.full(nodes_count, 0) # vectorul de
   ↳ apartenență a nodurilor la o insulă,
11     # prin marcarea cu numărul de ordine specific insulei
12
13     island_index = 0 # indexul curent al insulei
14     busiest_island_index = None # indexul insulei cu cele mai multe
   ↳ noduri
15     busiest_island_count = 0 # numarul de noduri din subgraful cu
   ↳ cele mai multe din ele
16
17     # cat timp inca exista noduri care nu au fost vizitate de o
   ↳ insula
18     while np.any(marked_island_nodes == 0):
19         # se găsește primul nod care nu a fost vizitat inca
20         island_start_node = None
21         for node in range(len(marked_island_nodes)):
22             if marked_island_nodes[node] == 0:
23                 island_start_node = node
24                 break
25
26         # si este considerat ca punctul de plecare dintr-o noua
   ↳ insula
27         if island_start_node is not None:
28             island_index += 1
29             current_island_count = 1
30             marked_island_nodes[island_start_node] = island_index
31             island_queue = queue.Queue()
32             island_queue.put(island_start_node)
33
34             # se parcurge in adancime insula, pornind de la nodul de
   ↳ plecare
35             while not island_queue.empty():
36                 current_node = island_queue.get()
37                 for edge_d in edges_dict.keys():
38                     node_a, node_b = edge_d
39                     # daca se gaseste un nod adiacent cu nodul
   ↳ curent,
40                     # atunci acesta face parte din insula cu indexul
   ↳ curent
41                     if node_a == current_node and
   ↳ marked_island_nodes[node_b] == 0:

```

```

42         marked_island_nodes[node_b] = island_index
43         island_queue.put(node_b)
44         current_island_count += 1
45     elif node_b == current_node and
46         ↪ marked_island_nodes[node_a] == 0:
47         marked_island_nodes[node_a] = island_index
48         island_queue.put(node_a)
49         current_island_count += 1
50
51     # daca dupa vizitarea nodurilor din insula curenta, s-a
52     ↪ gasit un numar mai mare
53     # decat insula curenta cu cele mai multe noduri,
54     # atunci devine noua insula aglomerata
55     if current_island_count > busiest_island_count:
56         busiest_island_count = current_island_count
57         busiest_island_index = island_index
58
59     # se returneaza nodurile izolate, adica cele care nu fac parte
60     ↪ din insula cu cele mai multe noduri
61     return [node for node in range(nodes_count) if
62             ↪ marked_island_nodes[node] != busiest_island_index]

```

Listing 3. Codul funcției *find_isolated_nodes*

Anexa 3. Funcții de recreere al drumurilor dintr-un graf

```

1  # functie de determinare a nodurilor din drumul dintre 2 noduri,
   ↪ folosind matricea obtinuta in algoritmul Floyd-Warshall
2  # algoritm preluat din
3  # https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#
4  # :~:text=procedure%20Path(u%2C%20v)%0A%20%20%20%20if%20next%5Bu
5  # %5D%5Bv%5D%20%3D%20null%20then%0A%20%20%20%20%20%20%20return
6  # %20%5B%5D%0A%20%20%20%20path%20%E2%86%90%20%5B%5D%0A%20%20%20
7  # %20while%20u%20%E2%89%A0%20v%0A%20%20%20%20%20%20%20u%20%E2
8  # %86%90%20next%5B%5D%5Bv%5D%0A%20%20%20%20%20%20%20%20
9  # path.append(u)%0A%20%20%20%20return%20path
10 def floyd_warshall_path(u: int, v: int, odd_nodes_dict):
11     if odd_nodes_dict[u]['next_nodes'][v] is None:
12         return []
13     the_path = [u]
14     alt_u = u
15     while alt_u != v:
16         alt_u = odd_nodes_dict[alt_u]['next_nodes'][v]
17         the_path.append(alt_u)
18     return the_path

```

Listing 4. Codul funcției de obținere al drumului dintre 2 noduri, pornind de la matricea de succesori din algoritmul Floyd-Warshall

```

1  # algoritmul de construire al unui drum, pornind de la nodul de
   ↪ origine și ajungând la alt nod
2  # în dicționarul de noduri este stocat pentru fiecare nod vectorul de
   ↪ predeceși în drumurile ce pornesc din el
3  def dijkstra_path_odd(u, v, nodes_dict) -> list:
4      if u == v:
5          return [u]
6      else:
7          rec_lst = dijkstra_path_odd(u,
           ↪ nodes_dict[u]['prev_nodes'][v], nodes_dict)
8          rec_lst.append(v)
9          return rec_lst

```

Listing 5. Codul funcției de obținere al drumului dintre 2 noduri, pornind de la vectorul de predecesori ai nodului-origine u , obținut din algoritmul lui Dijkstra

Anexa 4. Funcția de determinare al turului eulerian folosind algoritmul Hierholzer

```

1  import numpy as np
2  import random
3
4  # algoritmul Hierholzer de determinare a ciclului eulerian
   ↪  dintr-un graf eulerian
5  def hierholzer_alg(nodes_dict: dict, edges_dict: dict):
6  # copiem dictionarele de noduri, respectiv muchii (pentru a nu
   ↪  denatura originalele)
7  nodes_dict_copy = {key: val for key, val in nodes_dict.items()}
8  edges_dict_copy = {key: val for key, val in edges_dict.items()}
9
10 start_node = random.choice(list(nodes_dict.keys())) # punctul de
   ↪  start al turului, ales aleatoriu
11 total_edges = sum(edges_dict[edge_d]['freq'] for edge_d in
   ↪  edges_dict.keys()) # numarul total de muchii din graf
12
13 tour = [] # turul propriu-zis
14 visited_edges = 0 # numar de muchii vizitate
15
16 while visited_edges < total_edges:
17     # se determina nodul curent de start
18     current_start_node = None
19     # la inceput, turul principal va avea ca prim element nodul
   ↪  de start ales
20     if len(tour) == 0:
21         current_start_node = start_node
22     # daca un tur a fost deja realizat, dar nu au fost vizitate
   ↪  toate muchiile
23     else:
24         # se gaseste primul nod din tur care are gradul nenul
   ↪  (inca mai are muchii adiacente nevizitate)
25         nodes_g = set(tour)
26         for node_n in nodes_g:
27             if nodes_dict_copy[node_n]['grade'] > 0:
28                 current_start_node = node_n
29                 break
30
31     # odata ce a fost gasit un nod de start pentru un tur
32     if current_start_node is not None:
33         # este creat un nou tur
34         new_tour = [current_start_node]
35
36         chosen_edge = None
37         lowest_freq = np.Inf
38         # si se gaseste prima muchie adiacenta cu nodul de start,
   ↪  cu frecventa cea mai mica
39         for g_edge in edges_dict_copy.keys():
40             node_n, node_m = g_edge
41             # cand este gasit o muchie cu frecventa pozitiva
   ↪  (inca mai poate fi vizitata)
42             # si mai mica decat minimul curent

```

```

43         if (node_n == new_tour[-1] or node_m == new_tour[-1])
44             ↪ \
45                 and 0 < edges_dict_copy[g_edge]['freq'] <
46                 ↪ lowest_freq:
47                 # se pastreaza muchia si se inlocuieste minimul
48                 ↪ cu noua frecventa
49                 lowest_freq = edges_dict_copy[g_edge]['freq']
50                 chosen_edge = g_edge
51
52     # La finalul buclei for se va gasi o muchie cu frecventa
53     ↪ cea mai mica
54     # Astfel ne asiguram sa trecem prin muchiile cu frecventa
55     ↪ 1 mai intai, pentru a nu fi vizitate din nou in
56     # alte tururi
57     # Se adauga muchia cu cea mai mica frecventa strict
58     ↪ pozitiva la turul curent
59     node_n, node_m = chosen_edge
60     new_tour.append(node_n if node_n != new_tour[-1] else
61     ↪ node_m)
62     visited_edges += 1
63     # se scad de asemenea gradele capetelor muchiei, cat si
64     ↪ frecventa acesteia
65     nodes_dict_copy[node_n]['grade'] -= 1
66     nodes_dict_copy[node_m]['grade'] -= 1
67     edges_dict_copy[chosen_edge]['freq'] -= 1
68
69     # cat timp ultimul nod din tur nu coincide cu nodul de
70     ↪ start
71     while new_tour[-1] != current_start_node:
72         chosen_edge = None
73         lowest_freq = np.Inf
74         # se repeta procedura de gasire a unei muchii, ca cea
75         ↪ de mai sus,
76         # dar de data asta cu o muchie adiacenta cu ultimul
77         ↪ nod din tur
78         for g_edge in edges_dict_copy.keys():
79             node_n, node_m = g_edge
80             if (node_n == new_tour[-1] or node_m ==
81             ↪ new_tour[-1]) \
82                 and 0 < edges_dict_copy[g_edge]['freq'] <
83                 ↪ lowest_freq:
84                 lowest_freq = edges_dict_copy[g_edge]['freq']
85                 chosen_edge = g_edge
86
87         node_n, node_m = chosen_edge
88         nodes_dict_copy[node_n]['grade'] -= 1
89         nodes_dict_copy[node_m]['grade'] -= 1
90         edges_dict_copy[chosen_edge]['freq'] -= 1
91         new_tour.append(node_n if node_n != new_tour[-1] else
92         ↪ node_m)
93         visited_edges += 1
94
95     # daca este prima oara cand se realizeaza un tur

```

```

82         if len(tour) == 0:
83             tour = new_tour.copy()
84         else:
85             # altfel se gaseste pozitia nodului curent de start
86             ↪ in tur
87             start_pos_index = tour.index(current_start_node)
88             # iar apoi se divide turul in 2, excluzand pozitia
89             ↪ nodului curent de start
90             if start_pos_index == 0:
91                 tour_half1, tour_half2 = [], tour[start_pos_index
92                 ↪ + 1:].copy()
93             elif start_pos_index == len(tour) - 1:
94                 tour_half1, tour_half2 =
95                 ↪ tour[:start_pos_index].copy(), []
96             else:
97                 tour_half1, tour_half2 =
98                 ↪ tour[:start_pos_index].copy(),
99                 ↪ tour[start_pos_index + 1:].copy()
100             # in cele din urma, se reconstruieste turul initial,
101             ↪ adaugand noul tur in locul pozitiei gasite
102             tour = tour_half1.copy()
103             tour.extend(new_tour)
104             tour.extend(tour_half2)
105
106     # odata ce s-au vizitat toate muchiile, se returneaza rezultatul
107     return tour

```

Listing 6. Codul funcției *hierholzer_alg*

Anexa 5. Funcția de generare al turului eulerian pentru graful unui oraș extras din harta corespunzătoare unui fișier XML OSM

```

1  def generate_course(source_filename, settlement_name,
    ↪ roadExtractionEvent: threading.Event,
2      routeProcessingEvent: threading.Event,
3      animationProcessingEvent: threading.Event, stopEvent:
    ↪ threading.Event, solution_content: dict):
4      roadExtractionEvent.set()
5      # se extrage din harta graful orasului
6      nodes_dict, edges_dict = get_city_graph_from_map(source_filename,
    ↪ settlement_name)
7
8      # se cauta noduri care sunt deconectate de la graf, folosind BFS
9      isolated_nodes = find_isolated_nodes(nodes_dict, edges_dict)
10
11     # daca au fost gasite noduri izolate
12     if len(isolated_nodes) > 0:
13         print(f'Isolated nodes: {isolated_nodes}')
14         marked_edges = []
15         for edge in edges_dict:
16             node1, node2 = edge
17             if node1 in isolated_nodes or node2 in isolated_nodes:
18                 marked_edges.append(edge)
19
20         for edge in marked_edges:
21             node1, node2 = edge
22             edges_dict.pop(edge)
23             nodes_dict[node1]['grade'] -= 1
24             nodes_dict[node2]['grade'] -= 1
25
26         for isolated_node in isolated_nodes:
27             nodes_dict.pop(isolated_node)
28
29         # calibrare dicționare de noduri și muchii după eliminarea
    ↪ nodurilor izolate și muchiilor incidente
30         node_index_dict = {graph_node: index for index, graph_node in
    ↪ enumerate(list(nodes_dict.keys()))}
31         nodes_dict = {node_index_dict[key]: value for key, value in
    ↪ nodes_dict.items()}
32         edges_dict = {(node_index_dict[edge[0]],
    ↪ node_index_dict[edge[1]]): value
33                        for edge, value in edges_dict.items()}
34     else:
35         print('No isolated nodes found')
36
37     routeProcessingEvent.set()
38     # se verifica daca graful este eulerian
39     # daca toate nodurile din graf au gradul par
40     is_eulerian = True
41
42     for node in nodes_dict.keys():
43         if nodes_dict[node]['grade'] % 2 == 1:
44             is_eulerian = False

```

```

45         break
46
47     print(f'Is Eulerian? Response: {is_eulerian}')
48
49     # in cazul in care graful nu este eulerian, avem de a face cu
50     # → problema postasului chinez
51     # (determinarea unui cost suplimentar minim al muchiilor astfel
52     # → incat graful sa devina eulerian)
53     if not is_eulerian:
54         # se determina distanta minima a drumurilor intre fiecare
55         # → pereche posibilă de noduri de grad impar din graful
56         # orasului, cat si predecesorii fiecărui nod, pentru
57         # → reconstrui drumurile
58         start_time = time.time()
59         fd_or_djk = False # daca se aplica floyd-warshall sau
60         # → dijkstra
61
62         if fd_or_djk:
63             # se extrag matricile de distante minime, respectiv de
64             # → succesori din algoritmul floyd-warshall
65             new_nodes_list, dist_nodes, next_node_matrix =
66             # → floyd_warshall_alg(nodes_dict, edges_dict)
67
68             # recalibrare dictionare in functie de notatia noua din
69             # → new_nodes_dict
70             node_index_dict = {graph_node: index for index,
71             # → graph_node in enumerate(new_nodes_list)}
72             nodes_dict = {node_index_dict[key]: value for key, value
73             # → in nodes_dict.items()}
74             edges_dict = {(node_index_dict[edge[0]],
75             # → node_index_dict[edge[1]]): value
76             # for edge, value in edges_dict.items()}
77
78             # introducerea vectorilor de drumuri minime, respectiv
79             # → succesori pentru fiecare nod, în acestia
80             for node_i in nodes_dict.keys():
81                 nodes_dict[node_i]['dist_nodes'] = dist_nodes[node_i]
82                 nodes_dict[node_i]['next_nodes'] =
83                 # → next_node_matrix[node_i]
84
85                 if node_i in odd_nodes_dict.keys():
86                     odd_nodes_dict[node_i]['dist_nodes'] =
87                     # → dist_nodes[node_i]
88
89             else:
90                 # se prelucrează în paralel, folosind un executor de
91                 # → procese, vectorii de distante, cât și de precedenți,
92                 # pentru fiecare nod de grad impar
93                 with concurrent.futures.ProcessPoolExecutor() as
94                 # → executor:
95                     nodes_list = list(odd_nodes_dict.keys())
96                     for node_i, result in zip(nodes_list,
97                     # → executor.map(dijkstra_alg, repeat(nodes_dict),
98                     # → repeat(edges_dict), nodes_list)):

```

```
80         dist_nodes, prev_nodes = result
81         odd_nodes_dict[node_i]['dist_nodes'] = dist_nodes
82         odd_nodes_dict[node_i]['prev_nodes'] = prev_nodes
83     end_time = time.time()
84
85     print(f'Execution: {(end_time - start_time) / 60} minutes')
86
87     # se determina dictionarul de distante intre fiecare pereche
88     ↪ posibila de noduri de grad impar
89     odd_pairs_distances = {}
90
91     for odd_node1 in odd_nodes_dict.keys():
92         for odd_node2 in odd_nodes_dict.keys():
93             if odd_node1 != odd_node2 and (odd_node2, odd_node1)
94                 ↪ not in odd_pairs_distances.keys() \
95                 and
96                 ↪ odd_nodes_dict[odd_node1]['dist_nodes'][odd_node2]
97                 ↪ != np.Inf:
98                 odd_pairs_distances[(odd_node1, odd_node2)] =
99                 odd_nodes_dict[odd_node1]['dist_nodes'][odd_node2]
100
101     print('\n')
102
103     # Folosind procedura greedy, se modifică graful oraşului
104     ↪ astfel încât acesta să devină eulerian
105     greedy_eulerian_graph(nodes_dict, edges_dict,
106         ↪ odd_pairs_distances, odd_nodes_dict)
107
108     # la final se verifica inca o data daca este eulerian, pentru
109     ↪ a demonstra corectitudinea solutiei
110     print()
111     isEulerian = True
112     odd_nodes_count = 0
113     for node in nodes_dict.keys():
114         if nodes_dict[node]['grade'] % 2 != 0:
115             isEulerian = False
116             break
117     print(f'Is Eulerian? {isEulerian}')
118     print(f'Odd nodes count: {odd_nodes_count}')
119
120     solution_nodes = nodes_dict.copy()
121     solution_edges = edges_dict.copy()
122
123     # determinarea circuitului eulerian
124     solution = hierholzer_alg(solution_nodes, solution_edges)
125
126     # pregătirea grafului si a solutiei pentru reprezentarea grafică
127     ↪ în GUI
128     animationProcessingEvent.set()
129     # metodă preluată din
130     ↪ http://brooksandrew.github.io/simpleblog/articles
131     # /intro-to-graph-optimization-solving-cpp/
132     # creare graf gol
```

```

123 g = nx.Graph()
124
125 # adaugarea muchiilor(strazilor) grafului specific orasului
126 for edge, val in solution_edges.items():
127     node1, node2 = edge
128     g.add_edge(node1, node2, d=val['d'])
129
130 for node, val in solution_nodes.items():
131     nx.set_node_attributes(g, {node: {'lat': val['lat'], 'lon':
132         ↪ val['lon']}}})
133
134 print(f'Eulerian circuit: {solution}')
135
136 solution_content['graph'] = g
137 solution_content['solution'] = solution
138 solution_content['lungime_totala_strazi'] = 0
139 solution_content['distanta_parcursa_suplimentar'] = 0
140 solution_content['distanta_totala_tur'] = 0
141 solution_content['distanta parcursa drona curenta'] = [0]
142
143 for strada in solution_edges.keys():
144     solution_content['lungime_totala_strazi'] +=
145     ↪ solution_edges[strada]['d']
146
147 total_sol_length = 0
148 for node1, node2 in zip(solution[:-1], solution[1:]):
149     if (node1, node2) in solution_edges.keys():
150         edge = (node1, node2)
151     else:
152         edge = (node2, node1)
153
154     solution_content['distanta parcursa drona curenta'].append(
155         ↪ round(total_sol_length + solution_edges[edge]['d'], 3))
156     total_sol_length += solution_edges[edge]['d']
157
158 solution_content['distanta_totala_tur'] = round(total_sol_length,
159     ↪ 3)
160 solution_content['distanta_parcursa_suplimentar'] =
161     ↪ round(solution_content['distanta_totala_tur'] -
162     ↪ solution_content['lungime_totala_strazi'], 3)
163
164 stopEvent.set()

```

Listing 7. Codul funcției *generate_source*

Anexa 6. Interfața cu utilizatorul

```
1  # -*- coding: utf-8 -*-
2
3  # Form implementation generated from reading ui file
4  #   'InterfataAplicatie.ui'
5  #
6  # Created by: PyQt5 UI code generator 5.15.4
7  #
8  # WARNING: Any manual changes made to this file will be lost when
9  #   pyuic5 is
10 # run again. Do not edit this file unless you know what you are
11 #   doing.
12
13
14 from PyQt5 import QtCore, QtWidgets
15
16 class Ui_MainWindow(object):
17     def setupUi(self, MainWindow):
18         MainWindow.setObjectName("MainWindow")
19         MainWindow.resize(924, 532)
20         self.centralwidget = QtWidgets.QWidget(MainWindow)
21         self.centralwidget.setObjectName("centralwidget")
22         self.horizontalLayout =
23             QtWidgets.QHBoxLayout(self.centralwidget)
24         self.horizontalLayout.setObjectName("horizontalLayout")
25         self.menuLayout = QtWidgets.QVBoxLayout()
26         self.menuLayout.setObjectName("menuLayout")
27         self.commandsLayout = QtWidgets.QGridLayout()
28         self.commandsLayout.setObjectName("commandsLayout")
29         self.droneWattPerKmLabel =
30             QtWidgets.QLabel(self.centralwidget)
31         self.droneWattPerKmLabel.setEnabled(False)
32         self.droneWattPerKmLabel.setObjectName("droneWattPerKmLabel")
33         self.commandsLayout.addWidget(self.droneWattPerKmLabel, 2, 0,
34             1, 1)
35         self.locChooseLabel = QtWidgets.QLabel(self.centralwidget)
36         self.locChooseLabel.setEnabled(False)
37         self.locChooseLabel.setObjectName("locChooseLabel")
38         self.commandsLayout.addWidget(self.locChooseLabel, 1, 0, 1,
39             1)
40         self.locGenCourseButton =
41             QtWidgets.QPushButton(self.centralwidget)
42         self.locGenCourseButton.setEnabled(False)
43         self.locGenCourseButton.setObjectName("locGenCourseButton")
44         self.commandsLayout.addWidget(self.locGenCourseButton, 3, 0,
45             1, 2)
46         self.loadMapButton =
47             QtWidgets.QPushButton(self.centralwidget)
48         self.loadMapButton.setObjectName("loadMapButton")
49         self.commandsLayout.addWidget(self.loadMapButton, 0, 1, 1, 1)
50         self.droneWattPerKmSpinBox =
51             QtWidgets.QSpinBox(self.centralwidget)
```



```

42         self.droneWattPerKmSpinBox.setEnabled(False)
43         self.droneWattPerKmSpinBox.setProperty("value", 10)
44
45         → self.droneWattPerKmSpinBox.setObjectName("droneWattPerKmSpinBox")
46         self.commandsLayout.addWidget(self.droneWattPerKmSpinBox, 2,
47         → 1, 1, 1)
48         self.mapStatusLabel = QtWidgets.QLabel(self.centralwidget)
49         self.mapStatusLabel.setObjectName("mapStatusLabel")
50         self.commandsLayout.addWidget(self.mapStatusLabel, 0, 0, 1,
51         → 1)
52         self.locChooseComboBox =
53         → QtWidgets.QComboBox(self.centralwidget)
54         self.locChooseComboBox.setEnabled(False)
55         self.locChooseComboBox.setObjectName("locChooseComboBox")
56         self.commandsLayout.addWidget(self.locChooseComboBox, 1, 1,
57         → 1, 1)
58         self.menuLayout.addLayout(self.commandsLayout)
59         self.tourInfoGroupBox =
60         → QtWidgets.QGroupBox(self.centralwidget)
61         self.tourInfoGroupBox.setEnabled(False)
62         self.tourInfoGroupBox.setObjectName("tourInfoGroupBox")
63         self.formLayout_3 =
64         → QtWidgets.QFormLayout(self.tourInfoGroupBox)
65         self.formLayout_3.setObjectName("formLayout_3")
66         self.consumedPowerWLabel =
67         → QtWidgets.QLabel(self.tourInfoGroupBox)
68         self.consumedPowerWLabel.setObjectName("consumedPowerWLabel")
69         self.formLayout_3.addWidget(4,
70         → QtWidgets.QFormLayout.LabelRole,
71         → self.consumedPowerWLabel)
72         self.consumedPowerWLineEdit =
73         → QtWidgets.QLineEdit(self.tourInfoGroupBox)
74         self.consumedPowerWLineEdit.setEnabled(False)
75         self.consumedPowerWLineEdit.setReadOnly(True)
76
77         → self.consumedPowerWLineEdit.setObjectName("consumedPowerWLineEdit")
78         self.formLayout_3.addWidget(4,
79         → QtWidgets.QFormLayout.FieldRole,
80         → self.consumedPowerWLineEdit)
81         self.tourTotalDistKmLabel =
82         → QtWidgets.QLabel(self.tourInfoGroupBox)
83
84         → self.tourTotalDistKmLabel.setObjectName("tourTotalDistKmLabel")
85         self.formLayout_3.addWidget(2,
86         → QtWidgets.QFormLayout.LabelRole,
87         → self.tourTotalDistKmLabel)
88         self.tourTotalDistKmLineEdit =
89         → QtWidgets.QLineEdit(self.tourInfoGroupBox)
90         self.tourTotalDistKmLineEdit.setEnabled(False)
91         self.tourTotalDistKmLineEdit.setReadOnly(True)
92
93         → self.tourTotalDistKmLineEdit.setObjectName("tourTotalDistKmLineEdit")

```

```
74         self.formLayout_3.addWidget(2,
        ↪     QtWidgets.QFormLayout.FieldRole,
        ↪     self.tourTotalDistKmLineEdit)
75     self.currentDistanceCrossedKmLabel =
        ↪     QtWidgets.QLabel(self.tourInfoGroupBox)
76
        ↪     self.currentDistanceCrossedKmLabel.setObjectName("currentDistanceCrossedKmLabel")
77     self.formLayout_3.addWidget(3,
        ↪     QtWidgets.QFormLayout.LabelRole,
        ↪     self.currentDistanceCrossedKmLabel)
78     self.currentDistanceCrossedKmLineEdit =
        ↪     QtWidgets.QLineEdit(self.tourInfoGroupBox)
79     self.currentDistanceCrossedKmLineEdit.setEnabled(False)
80     self.currentDistanceCrossedKmLineEdit.setReadOnly(True)
81
        ↪     self.currentDistanceCrossedKmLineEdit.setObjectName("currentDistanceCrossedKmLineEdit")
82     self.formLayout_3.addWidget(3,
        ↪     QtWidgets.QFormLayout.FieldRole,
        ↪     self.currentDistanceCrossedKmLineEdit)
83     self.totalStreetsLengthKmLabel =
        ↪     QtWidgets.QLabel(self.tourInfoGroupBox)
84
        ↪     self.totalStreetsLengthKmLabel.setObjectName("totalStreetsLengthKmLabel")
85     self.formLayout_3.addWidget(0,
        ↪     QtWidgets.QFormLayout.LabelRole,
        ↪     self.totalStreetsLengthKmLabel)
86     self.totalStreetsLengthKmLineEdit =
        ↪     QtWidgets.QLineEdit(self.tourInfoGroupBox)
87     self.totalStreetsLengthKmLineEdit.setEnabled(False)
88     self.totalStreetsLengthKmLineEdit.setReadOnly(True)
89
        ↪     self.totalStreetsLengthKmLineEdit.setObjectName("totalStreetsLengthKmLineEdit")
90     self.formLayout_3.addWidget(0,
        ↪     QtWidgets.QFormLayout.FieldRole,
        ↪     self.totalStreetsLengthKmLineEdit)
91     self.additionalDistanceKmLabel =
        ↪     QtWidgets.QLabel(self.tourInfoGroupBox)
92
        ↪     self.additionalDistanceKmLabel.setObjectName("additionalDistanceKmLabel")
93     self.formLayout_3.addWidget(1,
        ↪     QtWidgets.QFormLayout.LabelRole,
        ↪     self.additionalDistanceKmLabel)
94     self.additionalDistanceKmLineEdit =
        ↪     QtWidgets.QLineEdit(self.tourInfoGroupBox)
95     self.additionalDistanceKmLineEdit.setEnabled(False)
96     self.additionalDistanceKmLineEdit.setReadOnly(True)
97
        ↪     self.additionalDistanceKmLineEdit.setObjectName("additionalDistanceKmLineEdit")
98     self.formLayout_3.addWidget(1,
        ↪     QtWidgets.QFormLayout.FieldRole,
        ↪     self.additionalDistanceKmLineEdit)
99     self.menuLayout.addWidget(self.tourInfoGroupBox)
100    self.horizontalLayout.addLayout(self.menuLayout)
```

```

101     self.mapLayout = QtWidgets.QVBoxLayout()
102     self.mapLayout.setObjectName("mapLayout")
103     self.horizontalLayout.addLayout(self.mapLayout)
104     self.horizontalLayout.setStretch(1, 1)
105     MainWindow.setCentralWidget(self.centralwidget)
106     self.menubar = QtWidgets.QMenuBar(MainWindow)
107     self.menubar.setGeometry(QtCore.QRect(0, 0, 924, 26))
108     self.menubar.setObjectName("menubar")
109     MainWindow.setMenuBar(self.menubar)
110     self.statusbar = QtWidgets.QStatusBar(MainWindow)
111     self.statusbar.setObjectName("statusbar")
112     MainWindow.setStatusBar(self.statusbar)
113
114     self.retranslateUi(MainWindow)
115     QtCore.QMetaObject.connectSlotsByName(MainWindow)
116
117     def retranslateUi(self, MainWindow):
118         _translate = QtCore.QCoreApplication.translate
119         MainWindow.setWindowTitle(_translate("MainWindow", "Simulator
120             ↪ traseu zonă rezidențială dronă"))
121         self.droneWattPerKmLabel.setText(_translate("MainWindow",
122             ↪ "Consum putere dronă (W/km)"))
123         self.locChooseLabel.setText(_translate("MainWindow", "Alege
124             ↪ localitate"))
125         self.locGenCourseButton.setText(_translate("MainWindow",
126             ↪ "Generează tur"))
127         self.loadMapButton.setText(_translate("MainWindow", "Încarcă
128             ↪ harta..."))
129         self.mapStatusLabel.setText(_translate("MainWindow",
130             ↪ "<html><head/><body><p><span style=\" font-weight:600;
131             ↪ color:#aa0000;\">Status: Hartă
132             ↪ neîncărcată</span></p></body></html>"))
133         self.tourInfoGroupBox.setTitle(_translate("MainWindow",
134             ↪ "Informații Tur"))
135         self.consumedPowerWLabel\
136             .setText(_translate("MainWindow", "Putere consumată
137                 ↪ curentă (W)"))
138         self.tourTotalDistKmLabel\
139             .setText(_translate("MainWindow", "Distanță totală tur
140                 ↪ (km)"))
141         self.currentDistanceCrossedKmLabel\
142             .setText(_translate("MainWindow", "Distanță parcursă
143                 ↪ dronă curentă (km)"))
144         self.totalStreetsLengthKmLabel\
145             .setText(_translate("MainWindow", "Lungime totală străzi
146                 ↪ (km)"))
147         self.additionalDistanceKmLabel\
148             .setText(_translate("MainWindow", "Distanță parcursă
149                 ↪ suplimentar (km)"))

```

Listing 8. Codul clasei specifice setării UI-ului interfeței principale

```
1  # -*- coding: utf-8 -*-
2
3  # Form implementation generated from reading ui file
4  #   'loadingScreen.ui'
5  #
6  # Created by: PyQt5 UI code generator 5.15.4
7  #
8  # WARNING: Any manual changes made to this file will be lost when
9  #   pyuic5 is
10 # run again. Do not edit this file unless you know what you are
11 #   doing.
12
13 from PyQt5 import QtCore, QtGui, QtWidgets
14 from PyQt5.QtCore import QTimer, Qt
15 from PyQt5.QtGui import QMovie
16
17 class LoadingScreen(QtWidgets.QDialog):
18     def __init__(self):
19         super().__init__()
20         self.setModal(True)
21         self.setWindowFlags(Qt.SplashScreen | Qt.FramelessWindowHint)
22         self.loadingLabel = QtWidgets.QLabel(self)
23         self.notificationLabel = QtWidgets.QLabel(self)
24         self.movie = QMovie("Spinner-1s-108px.gif")
25
26     def setupUi(self):
27         self.setObjectName("Form")
28         self.setFixedSize(400, 261)
29         sizePolicy =
30             QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Fixed,
31                                   QtWidgets.QSizePolicy.Fixed)
32         sizePolicy.setHorizontalStretch(0)
33         sizePolicy.setVerticalStretch(0)
34
35         sizePolicy.setHeightForWidth(self.sizePolicy().hasHeightForWidth())
36         self.setSizePolicy(sizePolicy)
37         self.setCursor(QtGui.QCursor(QtCore.Qt.WaitCursor))
38         self.notificationLabel.setGeometry(QtCore.QRect(60, 10, 291,
39                                                         91))
40         self.notificationLabel.setObjectName("notificationLabel")
41
42         self.loadingLabel.setGeometry(QtCore.QRect(150, 100, 111,
43                                                     121))
44
45         self.loadingLabel.setCursor(QtGui.QCursor(QtCore.Qt.WaitCursor))
46         self.loadingLabel.setText("")
47         self.loadingLabel.setObjectName("loadingLabel")
48         self.loadingLabel.setMovie(self.movie)
49
50         self.retranslateUi()
```

```

44         QtCore.QMetaObject.connectSlotsByName(self)
45
46     def startAnimation(self):
47         self.movie.start()
48         self.show()
49
50     def stopAnimation(self):
51         self.movie.stop()
52         self.close()
53
54     def retranslateUi(self):
55         _translate = QtCore.QCoreApplication.translate
56         self.setWindowTitle(_translate("Form", "Form"))
57         self.notificationLabel.setText(_translate("Form",
            → "<html><head/><body><p
            → align=\"center\"><br/></p></body></html>"))

```

Listing 9. Clasa *LoadingScreen*, pentru implementarea fereastrei „splash”

```

1  import sys
2  import threading
3  import time
4  import xml.etree.ElementTree as ET
5  from threading import Thread
6
7  import networkx as nx
8  from PyQt5 import QtCore, QtGui
9  from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog,
    → QMessageBox
10 from matplotlib.backends.backend_qt5agg import (FigureCanvasQTAgg as
    → FigureCanvas, NavigationToolbar2QT as NavigationToolbar)
11 from matplotlib.figure import Figure
12
13 from InterfataAplicatie import Ui_MainWindow
14 from LoadingScreen import LoadingScreen
15 from XMLParser import generate_course
16
17
18 class InterfataAplicatie(QMainWindow):
19     def __init__(self):
20         super().__init__()
21         self.eventHandlerThread = threading.Thread() # thread de
            → tratare al event-urilor din XMLParser
22         self._stopThreads = False # boolean pentru oprirea tuturor
            → thread-urilor active când se închide aplicatia
23         self._selectedFilename = None # fisierul selectat din
            → fereastra de selecție al fișierelor tip .osm
24         self.solution_content = {} # rezultatul funcției
            → generateCourse din XMLParser
25
26         self.ui = Ui_MainWindow() # UI-ul ferestrei
27         self.ui.setupUi(self)
28

```

```

29         # asignarea funcțiilor atunci când butoanele sunt apăsate
30         self.ui.loadMapButton.clicked.connect(self.loadMap)
31         self.ui.locGenCourseButton\
32             .clicked.connect(self.generateCourse)
33
34         # fereastra de încărcare din timpul procesării turului
35         ↪ eulerian
36         self.loadingScreen = LoadingScreen()
37         self.loadingScreen.setupUi()
38
39         self.figure = Figure(figsize=(10, 5)) # figura pe care se va
40         ↪ desena grafuri
41
42         self.mapCanvas = FigureCanvas(self.figure) # planșa pe care
43         ↪ se va desena graful
44         self.ax = self.mapCanvas.figure.subplots() # axa care
45         ↪ contribuie la desenarea grafurilor în figura din canvas
46         self.ui.mapLayout.addWidget(self.mapCanvas)
47         self.ui.mapLayout.addWidget(NavigationToolbar(self.mapCanvas,
48         ↪ self)) # adaugarea barei de navigatie pentru graf
49
50         self.show()
51
52         # se suprascrie metoda de închidere al aplicației, pentru a opri
53         ↪ mai întâi toate thread-urile active
54         def closeEvent(self, a0: QtGui.QCloseEvent) -> None:
55             self._stopThreads = True
56             if self.eventHandlerThread.is_alive():
57                 self.eventHandlerThread.join()
58             a0.accept()
59
60         # metoda de pornire al animației turului eulerian pe graful
61         ↪ orasului selectat
62         def startCourseAnimation(self):
63             print("Started animation! ;)")
64
65             # se extrag mai intai datele din solutia obtinută
66             graph, solution = self.solution_content['graph'],
67             ↪ self.solution_content['solution']
68             total_streets_length, additional_distance, total_tour_length,
69             ↪ current_drone_dist_list = \
70                 self.solution_content['lungime_totala_strazi'],
71                 ↪ self.solution_content['distanța_parcursa_suplimentar'],\
72                 self.solution_content['distanța_totală_tur'],
73                 ↪ self.solution_content['distanța parcursa drona
74                 ↪ curenta']
75             drone_consumption = self.ui.droneWattPerKmSpinBox.value()
76
77             # sunt setate culorile nodurilor și muchiilor ce vor fi
78             ↪ vizitate de-a lungul animației
79             node_colors = {node: 'gray' for node in graph.nodes}
80             edge_colors = {(edge[0], edge[1]): 'gray' for edge in
81             ↪ graph.edges}

```

```

68     node_pos = {node[0]: (node[1]['lat'], node[1]['lon']) for
        ↪ node in graph.nodes(data=True)}
69     visited_edges_colors = ['gray', 'red', 'orange', 'yellow',
        ↪ 'green', 'blue', 'purple', 'pink']
70
71     # funcția de inițializare al animației (pregătirea planșei și
        ↪ al grafului, inițial nevizitat)
72     def init_animation():
73         self.ui.totalStreetsLengthKmLineEdit\
74             .setText(f'{total_streets_length} km')
75         self.ui.additionalDistanceKmLineEdit\
76             .setText(f'{additional_distance} km')
77         self.ui.tourTotalDistKmLineEdit\
78             .setText(f'{total_tour_length} km')
79         self.ui.currentDistanceCrossedKmLineEdit\
80             .setText(f'{current_drone_dist_list[0]} km')
81         self.ui.consumedPowerWLineEdit\
82             .setText(f'{current_drone_dist_list[0] *
        ↪ drone_consumption} W')
83         self.ax.cla()
84         self.mapCanvas.figure.suptitle(f'Graful strazilor
        ↪ detectate din
        ↪ {self.ui.locChooseComboBox.currentText()} (conform
        ↪ hartii {self._selectedFilename})')
85         self.ax.set_facecolor("#E5FFE5")
86
87         nx.draw_networkx(graph, node_color='gray',
        ↪ edge_color='gray', pos=node_pos, node_size=5,
        ↪ with_labels=False, ax=self.ax)
88
89         for node_index in range(len(node_colors)):
90             if node_index == solution[0]:
91                 node_colors[node_index] = 'yellow'
92                 break
93
94         nx.draw_networkx_nodes(graph, pos=node_pos,
        ↪ nodelist=[solution[0]],
        ↪ node_color=node_colors[solution[0]], node_size=5,
        ↪ ax=self.ax)
95         self.mapCanvas.draw()
96
97         # desenarea unui frame al animației
98         # culoarea muchiei se va schimba în funcție de câte ori va fi
        ↪ vizitată
99         def animation_frame(i):
100             self.ui.currentDistanceCrossedKmLineEdit\
101                 .setText(f'{current_drone_dist_list[i+1]} km')
102             self.ui.consumedPowerWLineEdit\
103                 .setText(f'"{:.2f}".format(current_drone_dist_list[i
        ↪ + 1] * drone_consumption)} W')
104             current_node = solution[i]
105             next_node = solution[i + 1]
106             if current_node == solution[0]:

```

```

107         node_colors[current_node] = '#FF007F'
108     else:
109         node_colors[current_node] = 'white'
110     node_colors[next_node] = 'black'
111
112     if (current_node, next_node) in edge_colors:
113         current_edge = (current_node, next_node)
114     else:
115         current_edge = (next_node, current_node)
116     for color_index in range(len(visited_edges_colors) - 1):
117         if visited_edges_colors[color_index] ==
118             ↪ edge_colors[current_edge]:
119             edge_colors[current_edge] =
120             ↪ visited_edges_colors[color_index + 1]
121             break
122     nx.draw_networkx_nodes(graph, pos=node_pos,
123         ↪ node_color=[node_colors[current_node],
124         ↪ node_colors[next_node]], nodelist=[current_node,
125         ↪ next_node], node_size=5, ax=self.ax)
126     nx.draw_networkx_edges(graph, pos=node_pos,
127         ↪ edgelist=[current_edge],
128         ↪ edge_color=edge_colors[current_edge], ax=self.ax)
129     self.mapCanvas.draw()
130
131     # procedeul de redare al animației
132     init_animation()
133     for index in range(len(solution) - 1):
134         # dacă s-a oferit semnalul de stop de la închiderea
135         ↪ aplicației, se oprește animația
136         if self._stopThreads:
137             break
138         # altfel se oferă un timp de 0.3 secunde între fiecare
139         ↪ frame al animației și apoi se redă frame-ul curent
140         time.sleep(0.3)
141         animation_frame(index)
142
143     # metoda de generare al soluției, apelat la apăsarea butonului de
144     ↪ generare tur
145     def generateCourse(self):
146         selectedSettlement = self.ui.locChooseComboBox.currentText()
147         if selectedSettlement == '':
148             msg = QMessageBox()
149             msg.setWindowTitle("Warning!")
150             msg.setText("No settlement selected or none found on
151             ↪ map!")
152             msg.setIcon(QMessageBox.Warning)
153             msg.exec_()
154         else:
155             # se oprește thread-ul de event-uri dacă este deja activ
156             ↪ de la o prelucrare anterioară
157             if self.eventHandlerThread.is_alive():
158                 self._stopThreads = True
159                 self.eventHandlerThread.join()

```



```

148         self._stopThreads = False
149
150         # se setează event-urile pentru etapele generării
151         ↪ soluției
152         finishSolutionEvent = threading.Event()
153         roadExtractionEvent = threading.Event()
154         routeProcessingEvent = threading.Event()
155         animationProcessingEvent = threading.Event()
156         print(f'Selected Settlement: {selectedSettlement}')
157
158         # după se pornește firul de execuție al generării
159         ↪ soluției
160         processingThread = Thread(target=lambda:
161             ↪ generate_course(self._selectedFilename,
162             ↪ selectedSettlement, roadExtractionEvent,
163             ↪ routeProcessingEvent, animationProcessingEvent,
164             ↪ finishSolutionEvent, self.solution_content))
165         processingThread.start()
166
167         # in timpul acesta, se porneste fereastra de incarcare,
168         ↪ care blocheaza interfata principala
169         self.loadingScreen = LoadingScreen()
170         self.loadingScreen.setupUi()
171         self.loadingScreen.startAnimation()
172
173         # se porneste manager-ul de event-uri, care schimba
174         ↪ textul din loading screen
175         self.eventHandlerThread =
176             ↪ Thread(target=self.handleEvents,
177             ↪ args=[roadExtractionEvent, routeProcessingEvent,
178             ↪ animationProcessingEvent, finishSolutionEvent])
179         self.eventHandlerThread.start()
180
181         # metoda de gestionare al event-urilor din functia de generare
182         ↪ al solutiei
183         # in functie de în ce etapa se afla, se schimba textul loading
184         ↪ screen-ului
185         # odata finalizat, se incepe animatia grafului
186         def handleEvents(self, roadExtraction: threading.Event,
187             ↪ routeProcessing: threading.Event,
188             ↪ animationProcessing: threading.Event, stop:
189             ↪ threading.Event):
190             if roadExtraction.wait():
191                 self.loadingScreen.notificationLabel.setText('Extragere
192                     ↪ străzi și intersecții localitate...')
193             if routeProcessing.wait():
194                 self.loadingScreen.notificationLabel.setText('Calculare
195                     ↪ tur eulerian localitate...')
196             if animationProcessing.wait():
197                 self.loadingScreen.notificationLabel.setText('Creare
198                     ↪ animație tur...')
199             if stop.wait():
200                 self.loadingScreen.notificationLabel.setText('Gata!')

```

```

183         self.loadingScreen.stopAnimation()
184         self.ui.tourInfoGroupBox.setEnabled(True)
185         self.startCourseAnimation()
186
187         # metoda de incarcare al oraselor si a harti spre prelucrarea
188         ↪ acestora in functia de generare al solutiei
189     def loadMap(self):
190         self._selectedFilename, _ = QFileDialog.getOpenFileName(self,
191         ↪ "Open Image", filter="OpenStreetMap Files (*.osm)")
192         print(self._selectedFilename)
193         if self._selectedFilename != '':
194             tree = ET.parse(self._selectedFilename)
195             root = tree.getroot()
196             settlements_list = []
197
198             # gasirea oraselor se face pe baza etichetelor strazilor
199             ↪ care le apartin
200         for way in root.iter('way'):
201             name = None
202             highway_type = None
203             for tag in way.iter('tag'):
204                 if tag.attrib['k'] == 'highway' and
205                 ↪ tag.attrib['v'] in ['primary', 'secondary',
206                 ↪ 'tertiary', 'residential']:
207                     highway_type = tag.attrib['v']
208                 if tag.attrib['k'] == 'is_in:city':
209                     name = tag.attrib['v']
210             if highway_type is not None and name is not None and
211             ↪ name not in settlements_list:
212                 settlements_list.append(name)
213
214         print(settlements_list)
215
216         _translate = QtCore.QCoreApplication.translate
217         self.ui.mapStatusLabel.setText(_translate("MainWindow",
218         ↪ "<html><head/><body><p><span style=\"
219         ↪ font-weight:600; color:#00aa00;\">Status: Hartă
220         ↪ încărcată</span></p></body></html>"))
221
222         self.ui.locChooseComboBox.setEnabled(True)
223         self.ui.locChooseLabel.setEnabled(True)
224         self.ui.locChooseComboBox.clear()
225         self.ui.locChooseComboBox.addItem(settlements_list)
226
227         self.ui.droneWattPerKmLabel.setEnabled(True)
228         self.ui.droneWattPerKmSpinBox.setEnabled(True)
229
230         self.ui.locGenCourseButton.setEnabled(True)

```

Listing 10. Clasa *InterfataAplicatie*, care redă aplicația propriu-zisă, împreună cu GUI-ul