Sistem Inteligent de comutare a semafoarelor

Ştefan Răzvan

UTCN

25.10.2022

Cuprins:

1. Introducere

1.1 Context

Semaforul este un dispozitiv de semnalizare optică, alcătuit dintr-un stâlp fix, cu 2 sau 3 lumini (roşu, galben, verde), folosit pentru reglementarea circulaţiei în oraşe.

Acest proiect îşi propune realizarea unui sistem inteligent de comutare a unei serii de semafoare, care să comute culorile între roșu și verde astfel încât traficul să fie optimizat, adică timpul de așteptare mediu să fie minim.

De asemenea, proiectul va conține o interfață de intrare/ieșire pentru a primi datele generate de senzori și pentru a genera comenzile pentru semafoare. Cu ajutorul acestei interfețe, utilizatorul va putea introduce numărul de mașini și direcția din care să vină acestea.

Proiectul poate fi util pentru simularea traficului real și realizarea unei imagini de ansamblu a timpului de așteptare în trafic, deci poate fi utilizat în proiectarea într-un mod eficient a infrastructurii rutiere a unui oraș.

1.2 Specificație

Proiectul va fi realizat în python, folosind anumite module și librării specifice dezvoltării unei interfețe interactive, cu o grafică 2D a drumurilor și a mașinilor, vizualizarea fiind de sus. IDE-ul ales este PyCharm, datorită ușurinței testării și a realizării procesului de debugging. Pentru realizarea interfeței grafice, va fi folosită libraria Pygame, care facilitează realizarea de animații și de reprezentări grafice.

1.3 Obiective

Implementarea unui algoritm care să controleze timpii de comutare între culorile roșu și verde ale fiecărui semafor, astfel încât traficul să fie optimizat, adică timpul mediu de așteptare al mașinilor să fie minimizat, indiferent de numărul de mașini dat ca intrare de către utilizator pe fiecare drum.

Algoritmul va calcula timpul la care ajunge un grup de mașini la fiecare intersecție și va comanda semafoarele să afișeze culoarea verde cât mai repede după ajungerea, sau chiar înaintea ajungerii grupului de vehicule la respectiva intersecție.

2. Studiu bibliografic

Sincronizarea semafoarelor reprezintă una dintre cele mai bune soluții pentru evitarea blocajelor în trafic și a întârzierilor cumulate dintr-un oras. Un sistem de semafoare prost sincronizat reprezintă una din cauzele principale ale problemei enunțate mai sus.

Algoritmii de sincronizare pot fi de mai multe tipuri, câteva dintre acestea fiind:

- Sisteme pretemporizate: acestea folosesc intervale de timp fixe, adică o fază a semaforului este activă pentru un anumit interval de timp, după care sistemul de control comută în starea următoare, pentru același interval de timp, iar acest proces se repetă încontinuu.
- Sisteme semiacționate: în cazul în care 2 drumuri de capacități diferite se intersectează, anumiți senzori de trafic sunt montați pe drumul cu capacitatea mai mică. Drumul mai mare primește culoarea verde până când senzorii indică prezența unor mașini pe drumul mai mic, moment în care drumul mic primește verde și cel mare roșu, până cand trec toate mașinile de pe drumul mic, sau până cand se atinge un interval de timp fixat.
- Sisteme acționate: acestea sunt asemanatoare cu sistemele semiacționate, insa folosesc senzori pe toate drumurile, indiferent de capacitate, astfel putând să raspunda mai bine la fiecare situație din trafic.

Pentru acest proiect voi folosi un algoritm deterministic, care se mulează pe al treilea tip de algoritmi dintre cei prezentați mai sus.

Algoritmul va avea ca date de intrare numărul de mașini de pe fiecare drum, viteza fiecărei mașini, precum și timpul mediu de pornire al fiecarei mașini, la vederea culorii verzi. De asemenea, știind numărul de mașini, distanța ultimei mașini față de intersecție poate fi determinată și valorificată în proiectarea algoritmului.

Sistemul inteligent de comutare a semafoarelor poate fi abstractizat în 4 etape:

1. Preluarea datelor de intrare în timp real.
2. Prelucrarea acestor date folosind anumite ecuații pentru obținerea unei soluții cât mai bune în materie de timp de așteptare pentru șoferi
3. Aplicarea rezultatelor ecuațiilor pentru a obține datele de ieșire, adică timpii de afișare a culorilor roșu și verde pentru fiecare drum.
4. Repetarea primilor 3 pasi.

Am ales limbajul Python datorită librariei Pygame, librarie prin care interfata grafica poate fi realizata ușor și în puține linii de cod, astfel rămânând mai mult timp pentru proiectarea algoritmului în sine. Librăria Pygame ușurează acest proces datorită multitudinii de structuri de date și metode pe care le conține. O altă variantă ar putea fi limbajul Java, unde realizarea proiectului ar fi de asemenea posibilă, însă cu anumite costuri mai ridicate în materie de complexitate și lungime a codului.

Proiectul va fi rulat în timp real, reprezentând o intersecție în formă de cruce, unde utilizatorul va putea seta în orice moment de timp un număr de mașini care să fie adaugat pe oricare dintre cele 2 drumuri, apasând click pe unul din cele 4 butoane sau apăsând tastele < > ^ v.

3. Design

După cum am menționat în capitolul anterior, algoritmul este compus din 4 mari pași:

1.Preluarea datelor de intrare în timp real.

2.Prelucrarea acestor date folosind anumite ecuații pentru obținerea unei soluții cât mai bune în materie de timp de așteptare pentru șoferi

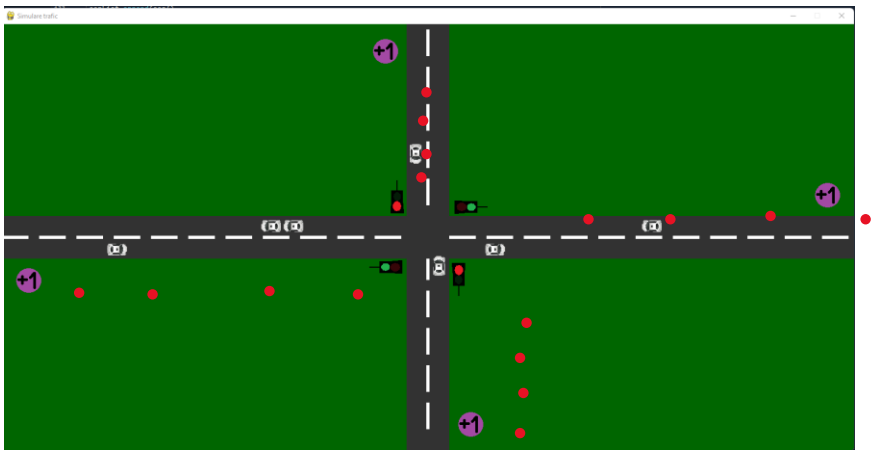
3.Aplicarea rezultatelor ecuațiilor pentru a obține datele de ieșire, adică timpii de afișare a culorilor roșu și verde pentru fiecare drum.

4.Repetarea primilor 3 pași.

Intersecția asupra căreia vom studia algoritmul este una simplă, în formă de cruce. Numărul de mașini de pe fiecare drum poate fi incrementat cu ajutorul a patru butoane plasate lângă locul unde vor apărea vehiculele după apăsarea acestora.

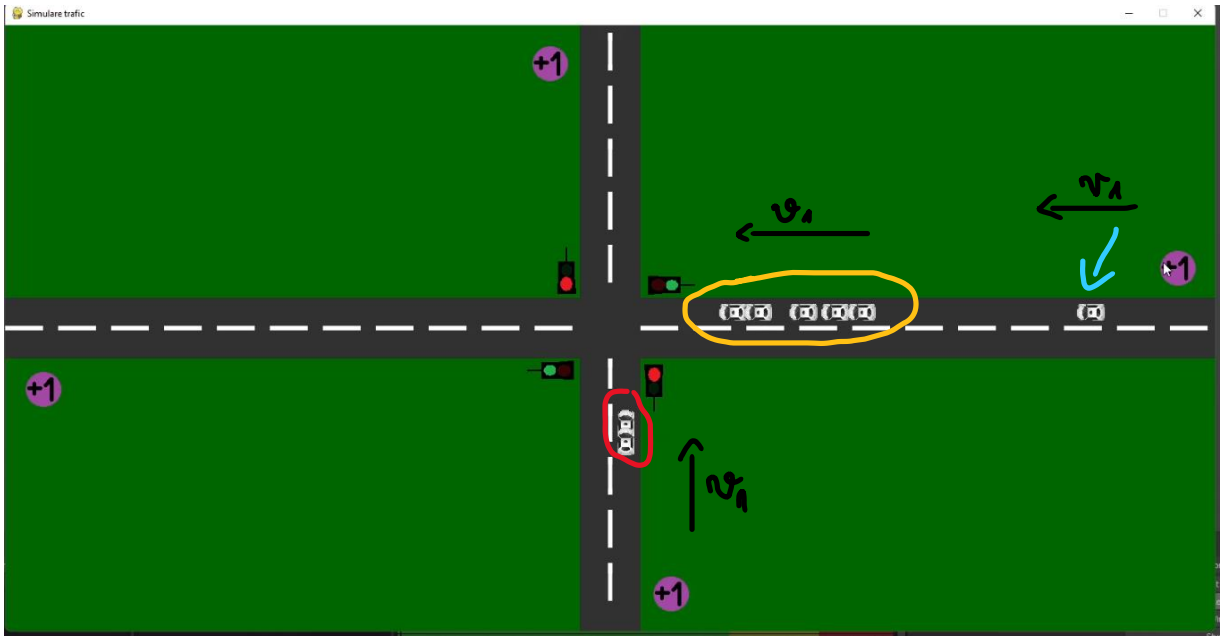
Algoritmul ales este unul determinist, neavând alegeri aleatoare în componența sa. Ne asumăm că pentru fiecare drum știm mai multe informații sigure despre mașini, pentru că există o serie de câțiva senzori cu ajutorul cărora putem determina numărul de mașini, pozitia acestora în raport cu intersecția, viteza precum și acceleratia mașinilor.

Senzorii sunt imaginari în aplicație, însă în implementarea reală aceștia ar putea folosi un sistem RADAR format din mai mulți senzori amplasați în locul punctelor roșii din imaginea de mai jos.



Informațiile pe care le extragem cu ajutorul senzorilor sunt lungimea cozii de mașini de pe un sens, viteza fiecarei mașini, distanța vehiculului față de intersecție, iar în plus ne asumăm un anume timp de pornire pentru fiecare mașină. O coada de mașini reprezintă mașinile dintr-un moment dat, de pe un sens al drumului, care nu au ajuns încă la intersecție. Dacă avem un număr de mașini în coadă, iar după o perioada mai apar câteva pe același sens de drum, acestea vor fi adăugate la acea coadă doar dacă timpul de așteptare mediu ar fi mai mic prin adăugarea acestora, decât timpul de așteptare rezultat prin schimbarea culorii semafoarelor, acordând astfel prioritate unei alte cozi, de pe alt sens.

Un exemplu în acest sens se află în imaginea de mai jos, unde coada incercuită cu galben are prioritate. Daca am adăuga la coadă și mașina indicată cu albastru, timpul total de așteptare ar fi mai mare decât dacă am lăsa mașinile încercuite cu roșu să treacă înaintea celei indicate cu albastru. Pentru acest exemplu ne asumăm o viteza constantă (v1) pentru toate mașinile, deoarece dacă viteza celei indicate cu albastru ar fi mult mai mare decat a celor încercuite cu rosu, ar trebui ca aceasta să fie adaugată la coadă.



Cât timp nu există mașini pe drumul vertical, vom afișa culoarea verde pe drumul orizontal în mod constant.

Vom utiliza viteza și distanta față de intersecție pentru a determina timpul necesar pentru traversarea intersecției de către o coadă de mașini.

Timpul de traversare cel mai mare (al ultimei mașini) este ales ca fiind durata fazei verzi a semaforului, doar daca acest timp este mai mic decat timpul maxim de comutare între faze ales (60 de secunde) . Acest timp de traversare va fi egal cu viteza ultimei mașini din coada imparțită la distanța până la intersecție.

In urma tuturor explicațiilor de mai sus, algoritmul va funcționa astfel:

1. Preluarea datelor de intrare și selectarea celei mai mari cozi de pe toate sensurile.
2. Calcularea timpului de traversare al acestei cozi și afișarea indicatorului verde pentru aceasta, în cazul în care acest timp nu depășește 60 de secunde.
3. Actualizarea cozii și timpului de traversare în cazul aparitiei unor mașini pe acelasi sens, prin a căror adăugare, timpul de așteptare rămâne minim. De asemenea, coada și prioritatea acesteia pot fi actualizate și în cazul în care incetinesc câteva mașini (scurtând astfel coada inițială), iar de pe alt sens vine o coadă cu mai multe mașini care ar ajunge simultan cu cea inițială (dar existenta mai multor mașini ii dă prioritate acesteia). În ultima situație, prioritatea se va schimba, ultima coada menționată primind culoarea verde.
4. Repetarea acestor pași.

4. Implementare

Pentru implementarea interfeței grafice am folosit diverse functii și structuri de date din librăria Pygame.

Mașinile reprezintă o imagine 2D cu fundalul transparent și au fost incarcate cu pygame.image.load:

Semafoarele au fost implementate în aceeași manieră, folosind 2 poze: una pentru roșu și una pentru verde.

Mașinile au coordonate pentru pozitiile x și y și se misca datorită unei funcții care modifică acele coordonate și a unei functii ce le redesenează. În imaginea de mai jos este surprinsă bucla while care rulează programul (apar doar părțile de cod relevante pentru mișcarea mașinilor):

```
227     k=0
228     while running:
229         k+=1
230         if k%7==0:
231             for c in carList:
232                 c.move()
233
234         for c in carList:
235             c.draw(screen)
236         pygame.display.update()
237     pygame.quit()
```

După cum se vede în imagine, o variabilă globală k se incrementează la fiecare iterație a buclei while. Pentru a reduce viteza mașinilor, funcția ce actualizează coordonatele se apelează doar din 7 în 7 iterații, deci dacă se dorește ca mașinile să meargă mai repede trebuie scăzut numărul cu care se face modulo. carList reprezintă o lista cu toate mașinile care nu au trecut de intersecție.

Implementarea algoritmului de control al semafoarelor:

Pentru fiecare directie pe care o pot avea mașinile (sus, jos, stânga, dreapta), am implementat câte o listă de mașini și o listă de “late arrivals”, adică mașini care nu au fost încă introduse în lista principală, dar vor fi introduse după ce toate mașinile din lista principală trec de intersecție. În cod aceste liste au primit denumirea de cozi deoarece fiecare listă poate fi imaginată ca o coadă de mașini care trebuie să treacă în intregime. Denumirile specifice sunt: QJos, QSus, QStg, QDr, iar listele de late arrivals au același nume cu completarea “LateArrivals”. Aceste liste sunt de tipul CarQueue:

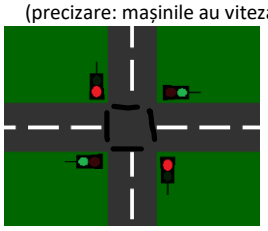
```
class CarQueue:
    def __init__(self, priority, direction, amount):
        self.carlist = []
        self.priority = priority
        self.direction = direction
        self.amount = amount
        self.lastCar = None
        self.firstCar = None
        self.dist_head_inceput = 0 # distanta dintre inceputul primei ma-
șini la inceputul intersecției
        self.dist_tail_sfarsit=0 # distanta dintre sf ultimei mașini la
sfarsitul intersecției
        self.latecarlist = [] #mașinile ce intra mai tarziu în queue
```

La fiecare iterație se realizează următoarele:

•dacă utilizatorul apasă unul dintre cele 4 butoane, sau una dintre tastele cu sageată, se creeaza mașina corespunzătoare.

•daca mașina nu este prea departe de coada corespunzatoare, aceasta este adaugată în acea coadă. Altfel, este adăugată în coada de LateArrivals, coada pentru care se verifică la fiecare iterație dacă coada principală a trecut de intersecție, moment în care mașina este scoasă din LateArrivals și introdusă în coada principală (fiind singurul element din ea, inițial). Motivul existentei cozilor de LateArrivals va fi explicat în secțiunea de testare, unde vor fi prezentate cazurile care nu ar functionă fără existența acestor cozi.

(precizare: mașinile au viteza constantă, iar începutul și sfârșitul intersecției este diferit pentru fiecare dintre cele 4 direcții:



•se calculează următoarele:

- numărul de mașini pe direcțiile sus-jos (**nrsusjos**) și pe direcțiile stânga-dreapta (**nrdrst**)

-**distsusjos_inceput** : (distanța dintre “botul” primei mașini din coada de jos și începutul intersecției) + (distanța dintre “botul” primei mașini din coada de sus și inceputul intersecției).

-**distsusjos_sfarsit**: (distanța dintre capătul ultimei mașini din coada de jos și sfârșitul intersecției) + (distanța dintre capătul ultimei mașini din coada de sus și sfârșitul intersecției).

-**distdrst_inceput**: (distanța dintre “botul” primei mașini din coada din dreapta și începutul intersecției) + (distanța dintre “botul” primei mașini din coada din stânga și începutul intersecției).

-**distdrst_sfarsit**: (distanța dintre capătul ultimei mașini din coada din dreapta și sfârșitul intersecției) + (distanța dintre capătul ultimei mașini din coada din stânga și sfârșitul intersecției).

• folosind variabilele de mai sus, se acordă prioritate drumul stânga-dreapta (“LR”) sau drumului sus-jos (“UD”) astfel:

```
102         if(nrsusjos==0):
103             self.selectPriority("LR")
104         elif(nrdrst==0):
105             self.selectPriority("UD")
106         elif(nrsusjos == nrdrst):
107             if(distdrst_inceput <= distsusjos_inceput):
108                 self.selectPriority("LR")
109             else:
110                 self.selectPriority("UD")
111         elif(nrsusjos > nrdrst):
112             if(distsusjos_inceput <= distdrst_sfarsit):
113                 self.selectPriority("UD")
114             else:
115                 self.selectPriority("LR")
116         elif(nrsusjos<=nrdrst):
117             if(distdrst_inceput <= distsusjos_sfarsit):
118                 self.selectPriority("LR")
119             else:
120                 self.selectPriority("UD")
121
```

Variabilele de distanță față de inceput, respectiv față de sfârșit sunt justificate datorită faptului că dacă “botul” unei mașini este mai departe de începutul intersecției decât spatele altei mașini este de sfârșitul intersecției, atunci

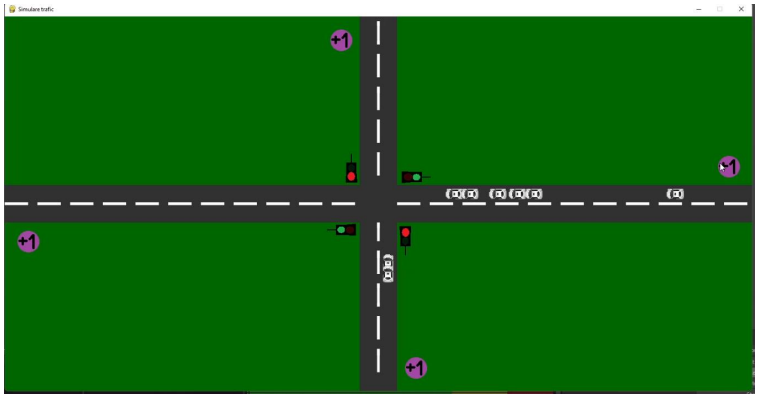
acestea nu se vor lovi trecând prin intersecție. Deci, de exemplu, în cazurile când numărul de mașini din stânga este mai mare decât nr. de mașini de jos, dar sfârșitul cozii de jos este mai aproape de sfârșitul intersecției, decât începutul cozii din stânga este de începutul intersecției, mașinile de jos pot să treacă înainte ca cele din stânga să ajungă. Astfel, trece coada cu mai puține mașini deoarece nu o afectează pe cea cu mai multe mașini (dacă ar afecta-o, nu ar trece, deoarece ar aștepta mai multe mașini, ceea ce creste timpul mediu de așteptare).

5. Testare

Testând programul, una dintre primele probleme întâlnite a fost faptul că uneori prioritățile se modificau în timp ce unele mașini erau în mijlocul intersecției ceea ce făcea ca unele să rămână blocate sau chiar să se lovească între ele (deoarece dimensiunea cozilor se modifica de fiecare dată când trecea o mașină). Am rezolvat această problemă prin modificarea dimensiunii unei cozi la 0 doar când toate mașinile trec de intersecție (astfel coada își păstrează prioritatea până trece în totalitate).

O altă problemă care a apărut a fost tot modificarea priorității în timp ce unele mașini erau în mijloc, dar din cauza adaugării de noi mașini, care măreau coada, deci aceasta primea prioritate deși coada care avea prioritate inițial încă era în trecere prin intersecție. Am rezolvat această problemă prin adăugarea unei conditii în plus în funcția ce modifică semafoarele, anume ca centrul intersecției să fie gol.

O problemă semnificativă a fost situația prezentată în capitolul de design:

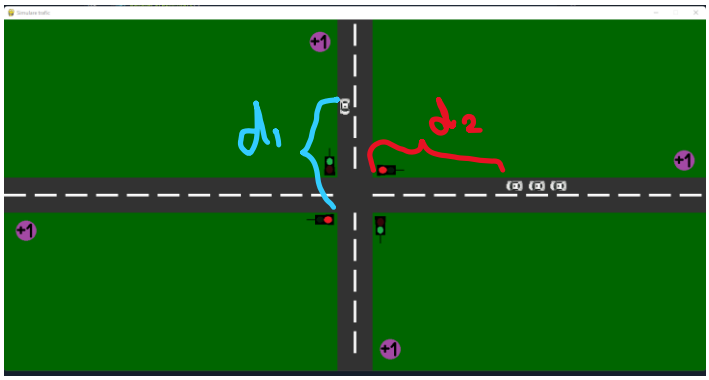


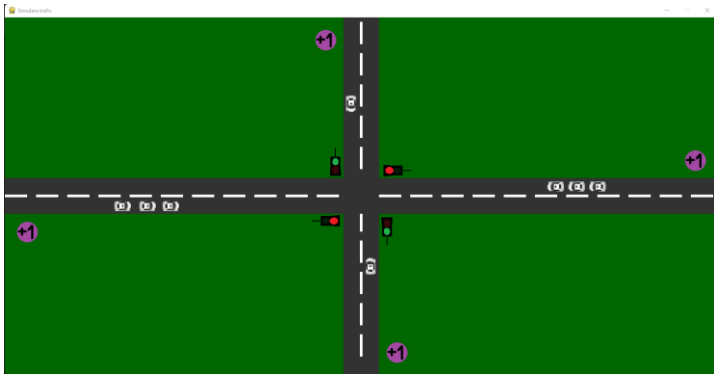
Mașinile de pe drumul din dreapta se află toate într-o coadă și au prioritate deoarece sunt mai multe. Astfel, mașinile de jos trec doar după ce trec toate mașinile din dreapta, ceea ce nu este eficient deoarece mașinile de jos ar putea trece înaintea ultimei mașini din dreapta, fara să o afecteze.

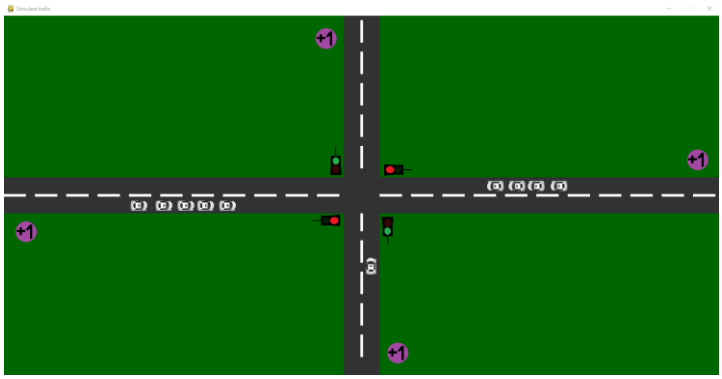
Pentru acest caz au fost implementate cozile de “LateArrivals”. Ultima mașină din dreapta este adăugată în coadă doar după ce trec celelalte mașini din fața ei. Acest lucru face ca, în momentul în care trec primele 5 mașini din dreapta, mașinile de jos să reprezinte o coadă mai mare decât cea din dreapta (care conține o singură mașină), fapt care le dă prioritate.

Mai jos sunt prezentate câteva teste pentru situațiile în care trece coada mai mică:

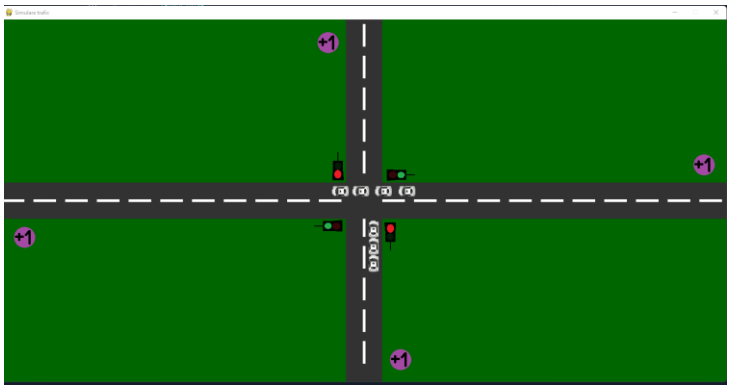
(d1 < d2)







În exemplul acesta, coada de jos așteaptă să treacă cea din dreapta (care are mai multe mașini):



6. Concluzie

Scopul acestui proiect a fost realizarea unui sistem inteligent de comutare a unei serii de semafoare, care să comute culorile între roșu și verde astfel încât traficul să fie optimizat, adică timpul de așteptare mediu să fie minim.

Am încercat să realizez algoritmul incremental, adică luând cazuri simple și rezolvându-le, iar apoi adăugând noi cazuri și actualizând soluția. Deși inițial găsirea unui algoritm eficient nu mi se părea o cerință dificilă, după fiecare

actualizare a soluției apăreau multe alte cazuri netratate, fapt care mi-a schimbat opinia despre dificultatea proiectării algoritmului.

Consider că acest proiect a contribuit semnificativ la dezvoltarea mea în domeniul ingineriei software, dar și a ingineriei în general.

7. Bibliografie

A comparison of algorithms used in traffic control systems by ERIK BJÖRCK, FREDRIK OMSTEDT:

http://www.diva-portal.org/smash/get/diva2:1214166/FULLTEXT01.pdf

https://ladot.lacity.org/sites/default/files/documents/ladot-atsac-signals-_-fact-sheet-2-14-2016.pd

Informații pentru interfața grafică:

https://www.youtube.com/@PythonSimplified