Dokumentáció vázlat

Ez a témalab beszámoló végére kerülne

1. Új klaszterezés: Ugyanúgy DBSCAN használata, viszont most azt is figyelembe veszi, hogy ugyan azon az úton vannak (irány is számít)

A DBSCAN:

done\_clusters = DBSCAN(eps=40, min\_samples=3, metric=similarity).fit(arr.astype(np.float64))

a „done\_clusters.labels\_” attributuma egy tömb ami tárolja az egyes járművek klaszter sorszámát. A metric paramétert azért kell változtatni hogy az utakat is figyelembe vegye mint dimenzió(ha más úton van két jármű akkor tekintse nagyon nagynak a távolságukat). Ezt majd beraktja a temp\_clusters tömbbe.

DBSCAN-ben használt „similarity” függvény:

def similarity(a, b):  
 if int(a[1]) == int(b[1]): # első autó edge-e megegyezik-e a másik autó edgével?  
 return norm(a[2:4] - b[2:4]) # ha igen, akkor adjuk meg a tavolsagot. dataframe oszlopait címzem itt  
 else:  
 return 1000000 # egy meglehetosen nagy ertek ami megakadalyozza hogy a nem azonos edge-n levok azonos klaszterben legyenek

Ha az első és a második jármű ugyanazon az úton van akkor simán visszaadja a távolságot, ha nem egyezik meg akkor meg egy nagy értéket, hogy ne kerülhessenek egy klaszterbe.

1. Klaszterek követése: A klaszterek ugyan azt a sorszámot kapják meg amit kaptak az előző lépésben. Így követhető grafikusan, valamint információkat lehet követni, például, hogy milyen hosszú ideig volt „életben” a klaszter

for i in temp\_clusters:  
 overwritten = False  
  
 for j in old\_clusters:  
 if overwritten == False:  
 if jaccard\_similarity(old\_clusters[j], temp\_clusters[i]) >= 0.25:  
 old\_temp\_clusters[j] = temp\_clusters[i]  
 overwritten = True  
  
 if overwritten == False:  
 # egy temp dictionaryba berakja  
 for k in range(0, 100):  
 if k not in new\_clusters.keys():  
 new\_clusters[k] = temp\_clusters[i]  
 break

Ez a ciklus végig megy a DBSCAN által besorolt klasztereken. Majd szétosztja, az alapján hogy az előző lépésben létezett e, ha nem létezett az elozőben akkor a legkisebb üressorszámhoz rendeli. Ezt úgy vizsgálom, hogy a klasztereket összehasonlítom az összes előző klaszterrel, és ha a „jaccard similarity” eléri a 0.25-öt akkor elég hasonlónak minősítjük, és ugyanazt a sorszámot kapja meg. Tesztelés alapján ennél a számál jól működött, és konzisztensen a kívánt eredményt kaptuk.

if len(old\_clusters) == 0:  
 for i in temp\_clusters:  
 for k in range(0, 100):  
 if k not in new\_clusters.keys():  
 new\_clusters[k] = temp\_clusters[i]  
  
 break  
  
old\_clusters = copy.deepcopy(old\_temp\_clusters)  
  
for k in new\_clusters:  
 for i in range(1, 100):  
 if i not in old\_clusters.keys():  
 old\_clusters[i] = new\_clusters[k]  
  
 break

Az első ciklus akkor fut le ha az előző lépésből nem maradt klaszter, ilyenkor a temp\_clusters-ből berakjuk a szabad new\_clusters indexekre az elemeit.

Az old\_clusters-be átmásoljuk az old\_temp\_clusters-t, majd a new\_clusters elemeit bemásoljuk az old\_clusters szabad indexeire.

1. A lámpák állítása: mindegyik közlekedési lámpa külön kiszámolja, hogy melyik irányba érdemes zöldet adni. Az alapján dönthet, hogy; melyik a legnagyobb klaszter, ami a kereszteződés felé halad, milyen távol van a keresztezéstől, a klaszterek közötti méret is számít, hogy mikor érdemes egy nagyobb klaszternek elsőbbséget adni.  
   Itt fontos az is hogy mindegyik zöld fázis után a jó zöld->sárga fázis jöjjön, és csak aztán váltson a kívánt fázisra. Ez generikusan van megírva, hogy minden lámpára működjön, minimális állítással.

Az is fontos itt, hogy nem feltétlen kell felülírni az alap fázis ciklust, vagyis ha nem talál nagyobb prioritású klasztert akkor végig megy az eredeti fázison teljes hosszáig (vagy akár ha nincsnek járművek az alap fázisokon megy végig).

if True:  
 tls\_x = 1784.5  
 tls\_y = 515.21  
  
 new\_prio = find\_priority\_edge(cluster\_nominees, old\_clusters, current\_prio, tls\_x=tls\_x, tls\_y=tls\_y)  
  
  
 # Csak akkor kell váltani ha másik klaszter kap priot  
 if current\_prio != new\_prio and new\_prio != 0:  
 # current\_prio = new\_prio  
  
 # check which edge prio cluster is on, and find with state gives it green, set to green  
  
 prio\_state = clusters\_edge[new\_prio]  
  
 if state\_of\_change == 0:  
 time\_passed\_since\_switch = traci.trafficlight.getPhaseDuration(tls) - (  
 traci.trafficlight.getNextSwitch(tls) - traci.simulation.getTime())  
  
 if time\_passed\_since\_switch > 23 and state\_of\_change == 0:  
 current\_phase = traci.trafficlight.getPhase(tlsID=tls)  
 desired\_phase = control\_Dict[prio\_state]  
  
 time\_passed\_since\_switch, state\_of\_change = ChangeToDesiredPhase(tls=tls, current\_phase=current\_phase, desired\_phase=desired\_phase, state\_of\_change=state\_of\_change, time\_passed\_since\_switch=time\_passed\_since\_switch)  
  
if state\_of\_change == 0:  
 time\_passed\_since\_switch = traci.trafficlight.getPhaseDuration(tls) - (  
 traci.trafficlight.getNextSwitch(tls) - traci.simulation.getTime())

Ezen az IF-en belül történik a beavatkozás, ha ez False-ra van állítva akkor beavatkozás nélkül megy végig a szimuláció.  
Megadjuk a kereszteződés koordinátáit (tls\_x, tls\_y-nal), ezt átadjuk a find\_priority\_edge függvénynek. A függvény visszaadja azt a klaszterindexet, aminek jelen esetben a legnagyobb a súllya, annak segítségével megkapjuk azt a lámpa fázist, aminek elsőbbséget akarunk adni.

A state\_of\_change ellenőrzéssel azt nézzük, hogy mikor volt a legutolsó lámpaváltás, ez azért fontos mivel, a program még magának is tudja változtatni a fázisait. Ha a time\_passed\_since\_switch meghaladja a minimális zöldidőt (ami jelen esetben 23 s) akkor meghívja a ChangeToDesiredPhase-t ami, abban az esetben hogy másik fázisba akarjuk állítani, biztonságosan átváltja sárgára, majd a kívánt fázisra. A minimális zöldidő az az idő amit legalább kap egy zöld fázis, ha elötte akarna váltani a program akkor nem engedi. Tesztek alapján 1.5-szörös forgalomnál 23 másodperc minimális zöldidő eredményezte a legnagyobb átbocsájtást.

def find\_priority\_edge(clus\_nom, all\_clusters, current\_prio, tls\_x, tls\_y):  
 # *TODO: Távolság alapján is nézni* # Ha még létezik a current\_prio akkor az legyen a benchmark  
 if current\_prio in all\_clusters:  
 priority\_cluster = current\_prio  
  
 current\_distance = find\_distance(current\_prio, all\_clusters, tls\_x, tls\_y)  
 biggest\_weight = (len(all\_clusters[current\_prio]) / max(current\_distance, 1)) \* 1.05 # Súlyozzás  
 # Ha nem akkor biztosan új lesz  
 else:  
 priority\_cluster = 0  
 biggest\_weight = 0  
  
 for i in clus\_nom:  
  
 distance\_from\_tls = find\_distance(i, all\_clusters, tls\_x, tls\_y)  
   
  
 candidate\_weight = len(all\_clusters[i]) / max(distance\_from\_tls, 1)  
 if candidate\_weight >= biggest\_weight:  
 biggest\_weight = candidate\_weight  
 priority\_cluster = int(i)  
  
 return priority\_cluster

A find\_priority\_edge megkeresi, hogy melyik clusternek kell prioritást adni, majd abból ki lehet deríteni, hogy melyik úton van.

A függvény csak azokat a klasztereket hasonlítja össze, amik szóba jöhetnek a kereszteződést illetően.  
Először megnézzük, hogy az előző prioritásos klaszter létezik még, ha igen akkor súlyozzuk egy tetszőleges számmal (amit az alapján döntünk el, hogy mennyire akarjuk hogy az kapjon prioritást), ha nem akkor szimplán megkeressük a legnagyobb súlyú klasztert. A súlyozás úgy működik, hogy a klaszter méretét elosztja a klaszter középpontjának távolságával (find\_distance függvény). Azért így döntöttem, hogy a klaszter középpontjához képest nézi, mert úgy gondoltam, hogy ez reálisabban mutatja a klaszter pozícióját, mint ha csak az első járművet néznénk. A legnagyobb klaszter sorszámát adja vissza, és az alapján megkapjuk az az utat, amin van az a klaszter.

def ChangeToDesiredPhase(tls, current\_phase, desired\_phase, state\_of\_change, time\_passed\_since\_switch):  
 if current\_phase == desired\_phase or 23 >= time\_passed\_since\_switch:  
 print("Nothing happened")  
 return time\_passed\_since\_switch, state\_of\_change  
  
 if state\_of\_change == 0:  
 traci.trafficlight.setPhase(tlsID=tls, index=current\_phase + 1)  
 state\_of\_change += 1  
 print("Setting state to yellow! State of Change:", state\_of\_change)  
 return time\_passed\_since\_switch, state\_of\_change  
  
 elif 1 <= state\_of\_change < 1 + 2: #Sárga fázis hossza, jelen esetben 3:  
 state\_of\_change += 1  
 print("Yellow state!")  
 return time\_passed\_since\_switch, state\_of\_change  
  
 elif state\_of\_change == 1 + 2:  
 traci.trafficlight.setPhase(tlsID=tls, index=desired\_phase)  
 print("Phase Change successful!!!:", desired\_phase)  
 state\_of\_change = 0  
 return time\_passed\_since\_switch, state\_of\_change  
  
 else:  
 return time\_passed\_since\_switch, state\_of\_change # current state is good

A ChangeToDesiredPhase a biztonságos fázis->sárgafázis->kívántfázis, műveletet valósítja meg. Mivel ez a folyamat több szimulációs lépésen keresztül történik meg, ezért meg kell várni, hogy a sárga fázis lemenjen, és csak aztán állíthatjuk a kívánt fázisba. Ezt a state\_of\_change követésével valósítottam meg. Ez biztosítja a biztonságos váltást akkor is, amikor nem egymás követő fázis a kívánt.

1. Eredmények:

Az eredményeket úgy tudtam számszerűsíteni, hogy a kereszteződés átbocsátóképességet megmértem. Ehhez a kimenő útszakaszra szenzorokat tettem, amik megszámolja az áthaladó járművek számát.

Az algoritmusom beavatkozásával megnőtt a kereszteződés átbocsájtó képessége. Ez a legnagyobb mértékben 1.5x forgalomnál látszik. Ahogy elérjük a gridlock közeli állapotot, már nem segít annyit. Kedvező esetben 5-10%-os javulást mutat.

A minimális zöldhossz állításán látszik, hogy eltérések lehetnek, hogy milyen forgalmi viszonyban melyik minimálzöldérték a legjobb.

Ezen a diagramon látszik, hogy forgalom sűrűségtől függően változik, hogy melyik a jobb minimális zöldhossz. Ebből le tudjuk szűrni, hogy ezt érdemes lehetne dinamikusan állítani forgalomtól függően.

1. Felmerült problémák:
   1. Egy klaszter csak azt mutatja, hogy van egy úton pár kocsi közel egymáshoz, azt nem, hogy melyik irányba akarnak menni, így hogy érdemes állítani a fázisokat? Maradhatnak e ugyanazok a fázisok, amik voltak vagy mindegyik lámpát átkéne állítani? Ez valószínűleg túl sok külön konfigurációval járna, más megoldás kell.
   2. Mikor érdemes zöldre váltani? Ez mindegyik kereszteződésnél más, és más lehet, mivel az számít, hogy mi a maximális sebesség (hogy mennyit kell fékezni a járműveknek, tehát hogy milyen hosszú legyen a sárga fázis) Mekkora méret különbségnél adunk elsőbbséget egy nagyobb klaszternek? Mikor éri meg?
2. Jövőbeli célok:
   1. Az algoritmus generalizálása, hogy minimális konfigurációval akármilyen új lámpás kereszteződéshez hozzá lehessen adni.
   2. Az algoritmust bővíteni gépi tanulási módszerekkel, hogy még nagyobb pozitív különbség legyen az alapbeállításhoz képest.