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. 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.

2. 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. Ezt úgy vizsgálom hogy a klasztereket összehasonlítom az összes előző klaszterrel, és ha eléggé hasonló a „jaccard similarity" (ez két halmaz hasonlóságát vizsgálja) alapján akkor ugyanazt a sorszámot kapja meg.

```
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
```

3. 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.

```
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.

```
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
```

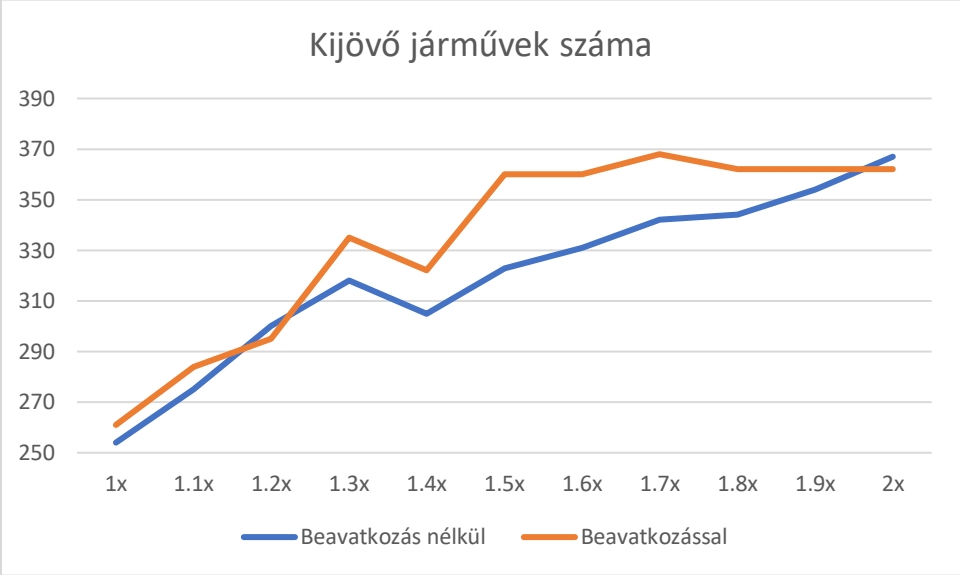
A find Priority_edge megkeresi, hogy melyik clusternek kell prioritást adni, majd abból ki lehet deríteni hogy melyik úton van.
A függvény csak azokat a klasztereket hasonlítja össze, amik szóba jöhetnek a kereszteződést illetően.

Először megnézzük, hogy az előző prioritás létezik még, ha igen akkor egy kicsit súlyozzuk, 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). 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
```

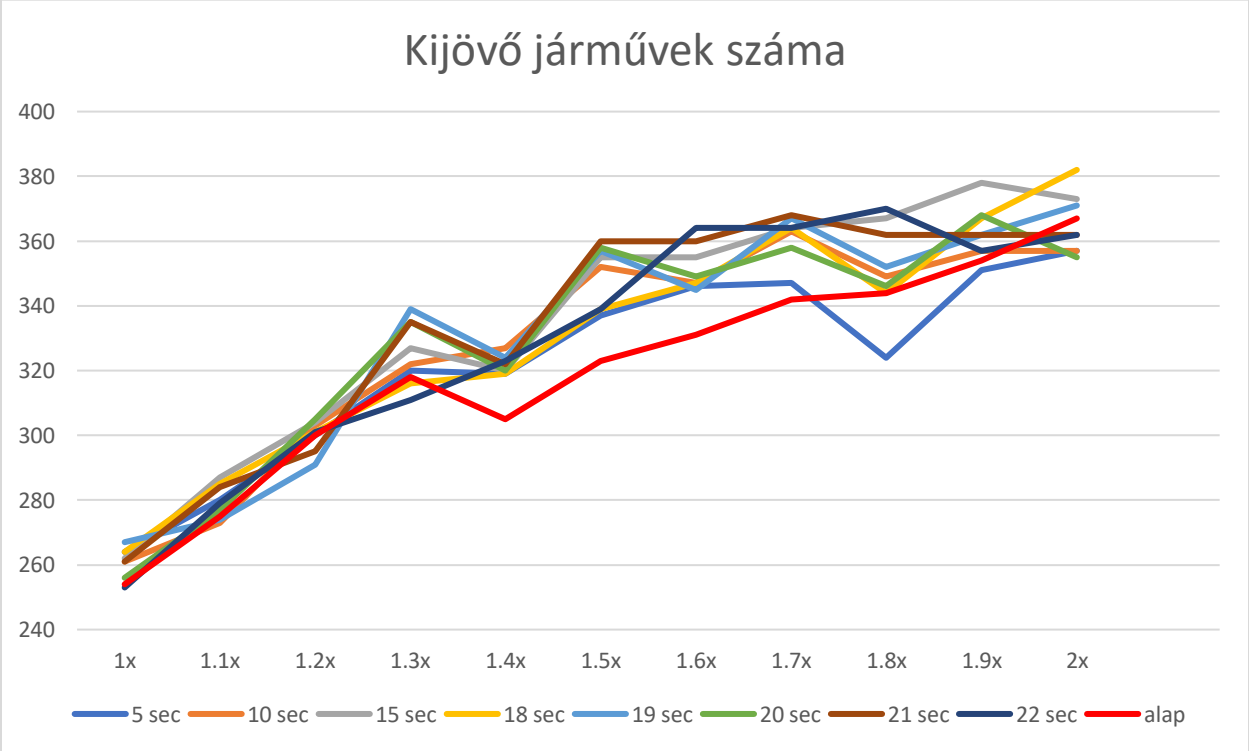
A ChangeToDesiredPhase a biztonságos fázis->sárgafázis->kívántfázis, műveletet valósítja meg. Mivel ez a folyamat több szimulációs lépésen keresztül történik meg, ezért meg kell várni hogy a sárga fázis lemenjen, és csak aztán állíthatjuk a kívánt fázisba. Ezt a state_of_change követésével valósítottam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

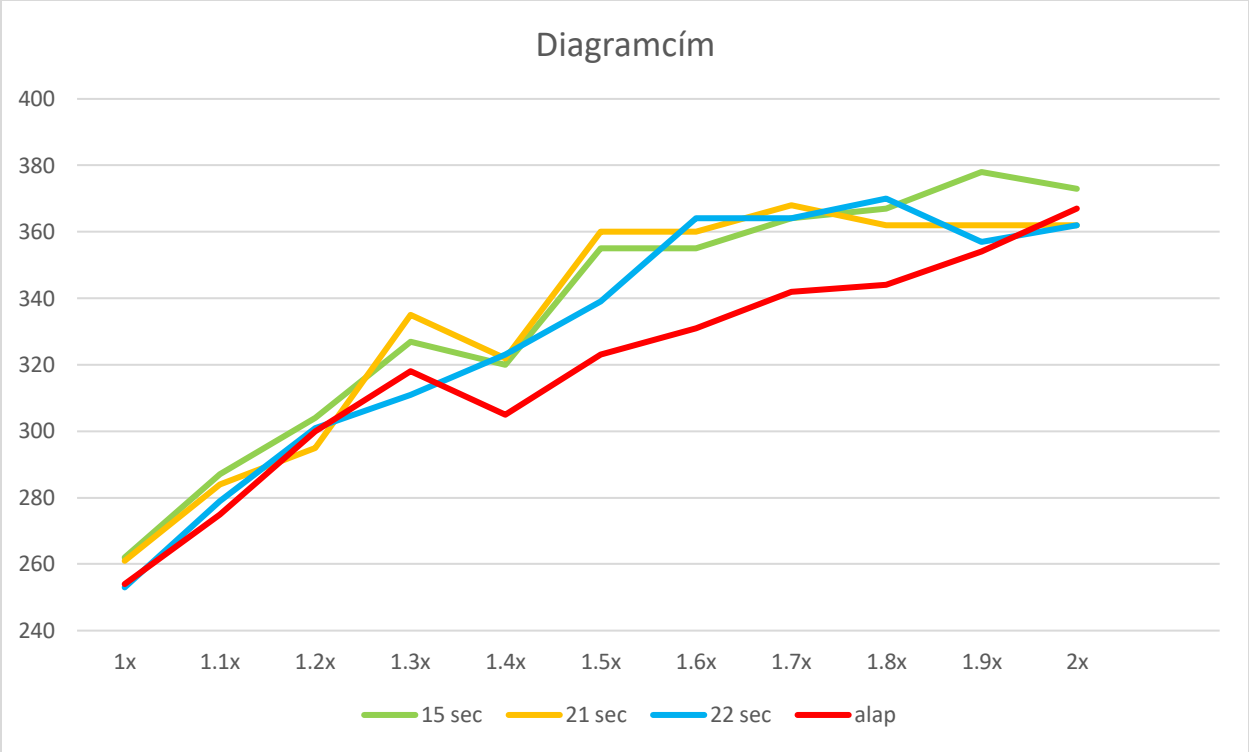
4. Végeredmények:



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.

5. Felmerült problémák:
    a. 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.

    b. 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?
6. Jövőbeli célok:
    a. Az algoritmus generalizálása, hogy minimállis konfigurációval akármilyen új lámpás kereszteződéshez hozzá lehessen adni.
    b. Az algoritmushost bővíteni gépi tanulási módszerekkel, hogy még nagyobb pozitív különbség legyen az alapbeállításhoz képest.