**Bevezetés**: miért, hogyan kaptam, kinek milyen munkáját folytatom, köszönet. Ezeket milyen sorrendben?

A szakdolgozati témaválasztó szemináriumon, amikor hallottam Kusper Gábor tanár úr magyarázatát a kutatásukról, annak céljáról és felhasználásáról, akkor nagyon megtetszett ez a téma. A SAT megoldó széleskörű felhasználásáról beszélgettünk. Korábbi előadásokon, gyakorlatokon is voltak tanáraim, akik ezt a témát felvezették, és már akkoriban meghozták a kedvemet hozzá. Amikor választanom kellett, nem volt nagy kérdés, hogy ez egy számomra érdekes téma, ami lehetőséget ad a fejlődésre.

Szaktársammal, Rajna Franciskával csak mi ketten érdeklődtünk ebben a témában, úgyhogy mindenki örömmel beszélte meg a részleteket és közös megegyezéssel találtuk ki melyik ágát dolgozza ki a témának. Pozitív és energikus első benyomás után örömmel kezdtünk a munkának. Bíró Csaba és Balla Tamás tanár urakkal dolgoztunk a témával kapcsolatos házi TDK-hoz hasonló előadásokon és kutatásokon (ICAI2020, AM2020 – Agria Média) vettünk részt. Az egyik alkalommal egy plakátot is készítettünk. Ezekkel megalapozva egy lendületes kezdést.

Az eddig felsorolt tanáraim korábbi munkájához kaptam hozzáférést és felhatalmazást, hogy folytassam munkájukat. Hálás köszönettel tartozok a fáradalmaikért, segítségükért, emberségükért és a rengeteg alaptudáshoz, amit tálcán nyújtottak át nekem korábbi, részletes munkájuk formájában. Ennek köszönhetően többet tudtunk gondolkodni lehetséges megoldásokon, fejlesztéseken. Megbeszéltük hol szorul fejlesztésre a SAT megoldó, amin tudok programozással javítani. Valamint a korábbi általuk írt angol nyelvű szakirodalmakkal elsajátíthatom az elméleti hátteret, felzárkózhatok a jelenlegi helyzethez és ezeken dolgozva könnyedén belerázódjak a szakdolgozatom megfogalmazásába.

TODO: rework the whole thing

**v0.1**: szakirodalom fordítása; érthetőre fordítás (rework); alapok elsajátítása; mi az, hogy plot; kódértelmezés: importálások, függvények, változók

A munkámat azzal kezdtem, hogy szakirodalmakat olvastam, fordítottam és értelmeztem, amiket korábban témavezetőim írtak. Anyagot gyűjtöttem és dolgoztam fel a SAT megoldókról és a Python programozási környezetről. Amikor megkaptam a kódot azon voltam, hogy megértsem, mi van elém írva. Ahogy volt lemásoltam és elkészítettem egy pár próba verziót, amik csak részletek kiemelése volt, és a program beüzemelése az általam használt Python verzióval. Ezután készítettem pár egyszerű átiratot, amit már kényelmesebben tudok értelmezni. Melyik importálás mit csinál, mihez kell, miért van egy import a kód közepén? Mert rosszul írtam át. És hasonló kérdések. Networkx: segít gráfként kezelni az adatokat. Pylab: a matplotlib.pyplot-ot és a Numpy-t egyesíti egy névtér alatt. A többi már érthetőbb csomagnévvel van ellátva. És persze alapfogalmaknak is utána néztem, mit jelent a plot-olás: egy adat szerkezet grafikai megjelenítéséhez egy technika. Hogy a helper\_funcs(G, orientation) az egy gráfot készít. Az eredeti find\_cycle() pedig egy irányított gráfot kér csak. Emellett azt is láttam, hogy sok meg nem valósított ötlet is lebegett a program kódban. Használaton kívüli változó nevek, amiknek csak a neve árulja el mire akarták használni és függvények, amiknek csak nevük van.

Összefoglalva sok baj van vele. Viszont ezeket nem kell kijavítanom, csak egy másik gyorsabb megoldást találjak. Robert Tarjan algoritmusa tökéletes megoldás, hiszen gyorsan talál meg **erősen összetett komponenseket**. Donald B. Johnson pedig ezt alapul véve körkereső algoritmust készített. „Bizonyos számolási problémák már meg lettek oldva. Mint például egy teljes irányított gráfban, ahol n csúcs van, ott pontosan elemi kör van. Tehát az elemi körök száma irányított gráfban n-el többel nőhet, mint az exponenciális . Így világos, hogy az algoritmusunk, aminek az időkorlátja bármely gráfon, aminek csúcsa, éle és elemi köre van, megvalósítható a lényegesen nagyobb problémaosztályokra, mint a korábban ismert legjobb algoritmusok, ami időkorlátos.” Ezzel bizonyítva, hogy tényleg gyorsabb az algoritmusa a korábbiaknál.

begin

integer list array Ak(n), B(n);

logical array blocked (n);

integer s;

logical procedure CIRCUIT (integer value v);

begin logical f;

procedure UNBLOCK (integer value u);

begin

blocked (u):= false;

for w ∈ B(u) do

begin

delete w from B(u);

if blocked(w) then UNBLOCK(w);

end

end UNBLOCK

f := false;

stack v;

blocked(v):= true;

L1: for w ∈ Ak(v) do

if w == s then

begin

output circuit composed of stack followed by s;

f := true;

end

else if NOT blocked(w) then

if CIRCUIT(w) then f := true;

L2: if f then UNBLOCK(v)

else for w ∈ Ak(v) do

if v !∈ B(w) then put v on B(w);

unstack v;

CIRCUIT := f;

end CIRCUIT;

empty stack;

s:=l;

while s < n do

begin

A:= adjacency structure of strong component K with least

vertex in subgraph of G induced by {s, s+ 1, ..., n};

if Ak != empty set then

begin

s := least vertex in Vk;

for i ∈ Vk, do

begin

blocked(i) := false;

B(i) := empty set;

end;

L3: dummy := CIRCUIT(s);

s:=s+l;

end

else s := n;

end

end;

Ebből kiindulva pár implementálását megkerestem. Találtam mind Tarjánéra, mind Johnson-éra Python kódot. Egy kis időbe telt mire észrevettem a két kód különbözik.

https://github.com/qpwo/python-simple-cycles

https://github.com/qpwo/python-simple-cycles/blob/master/johnson.py

Kéne írjak pár tesztet, hogy beállíthassam a graph\_cnf\_GEN\_0.8-checkpoint.py programkód generálását úgy, hogy teljes gráfokat készítsen.

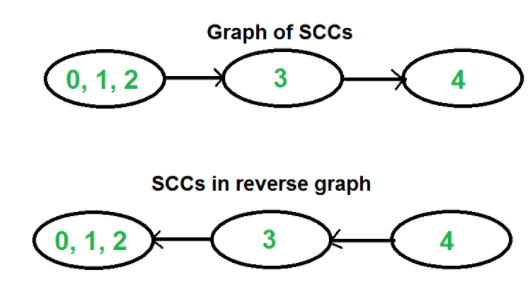
**v0.2**: Működésre bírás, pip install minden importált névtér, xrange = range, használaton kívüli változók kivéve, Az új algoritmus használatba helyezése, beüzemelése. g.add\_path[1,2,3] = networkx.add\_path(g, [1,2,3])

networkx eljutott addig, hogy megtalálja a köröket, viszont nem törli a dupla éleket. így egy teljes irányított gráf neki 4 csomópont esetén 12 éllel jár. Ami nagyon nem jó.

**v0.3**: Észrevettem, hogy nem jó implementálást akarok használni. Nekem tökéletesen elég a networkx-es megvalósítás, hiszen a saját függvényeivel együtt használom. Átírtam, hogy az elején a rajzoláshoz még az adatokkal felszerelt gráfot használja. Később már csak a műveletekhez szükséges adatok legyenek rajta úgy, mint az élek és a csúcsok. Viszont elakadtam annál a résznél, hogy mit takar a cic, isCici és milyen módon kéne átadjam az adatokat. Kicsit rendeztem a kódot, bár nem töröltem mindent, amit akartam. Már közel járok a végéhez úgy érzem.

**v0.4**: A teljes program működését ideiglenesen a háttérbe helyeztem, és nem debugolok olyan kódot, amit nem értek. Lásd: cic, isCici. A networkx működését próbáltam felhasználni, hogy lekódoljam az úgy nevezett weak modelt. Ehhez készítenem kell egy DAG-ot az eredeti gráfból. Amihez scc-ket kell csúcsokkal helyettesítsek. Így elkészül egy SCC-DAG azaz a weak model-ünk. (még akadályokba ütköztem a megvalósítással)

**v0.5**: Átírtam egy algoritmust, és kijavítottam, hogy működjön az én környezetemben. Ehhez kellett delegate/monkey patchinget csináljak, utána nézzek, megfejtsem, tanuljak. Hogy elő tudjam kiegészíteni az nx.DiGraph() környezetet. Elkezdtem összeegyeztetni az algoritmusokat, és működésre bírni a Weak Model-t.

**v0.6**: A weak model definíciója alapján szerintem kijelenthető, hogy a NetworkX-nek van egy metódusa, ami ugyan azt a modellt állítja elő. Lásd (https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.condensation.html?highlight=condensation) condensation, minek egy gráfot, és annak erősen összetett komponenseit átadva legenerálja egy gráfnak a gyenge modeljét. 

**Kis újra dolgozás**:

Azon dolgoztam, hogy a kód tisztább legyen, érthetőbb és átláthatóbb. Értelmeztem, amit csak tudtam, és azon dolgoztam, hogy lefusson a kód, és ne csak szintaktilkailag legyen helyes. Ne álljon meg hibával. Ebbe először bele buktam. Újra neki kellett álljak.

**Kód részletek**:

**find\_cycle**: Emiatt hagytam abba a program teljes átdolgozását is. Hiszen több hatásköre volt, mint amire először gondoltam. Nem tudtam megoldani, hogy egy függvény tudja a körök hosszát, vissza tudja adni azokat egyesével, vagy ha talált kört, akkor adja át a hosszát. És mivel nem egy scc-t vizsgál a függvény, hanem egy egész gráfot, így elég nehéz lett volna ezeket elvégezni. Egy függvénnyel három kérdést nem tudunk megválaszolni. Találtunk kört? Mennyi kört találtunk? Itt kellene az összes kör, ide adod?

if find\_cycle(g):

isCici=len(find\_cycle(g))

…

cic = find\_cycle(g)

1. ábra: sorban a kérdésekre a válasz

Az elsőre kérdésre igennel, vagy nemmel, a másodikra egy darabszámmal, a harmadikra pedig egy algráffal tudnánk helyesen válaszolni.

Ezt a tiszta kód semelyik szabálya, és az egy felelősségi kör elve sem támogatja. Úgyhogy megpróbáltam megoldani a jobb körkeresést. Ehhez mások munkáját vettem alapul. Bíró Csaba tanár úr kódjából indultam ki, valamint a kérésből, és javaslatból, hogy Donald B. Johnson munkásságát próbáljam meg alkalmazni.

**Johnson saját – simple\_cycles**: Találtam egy implementációt a körkeresésre qpwo által elkészítve (Donald B. Johnson algrotimusára) (https://github.com/qpwo/python-simple-cycles). Tőle megpróbáltam hasznosítani a saját implementálását, viszont ott nem jártam sikerrel, nem értettem a hibát, miért nem akar működni. Csak egy nappal később vettem észte, hogy a saját IronPython-hoz írt verzióját próbáltam működésre bírni. Ugyanakkor rájöttem, hogy a NetworkX implementációját követte, és én nekem az eredeti tökéletes (hiszen nem IronPhyton-ban dolgozok, amire qpwo átfordította). Mivel az eredeti algoritmus úgy szint symple\_cycles néven megtalálható volt, ezért kicseréltem az enyémmel kompatibilisre (https://github.com/networkx/networkx/blob/main/networkx/algorithms/cycles.py#L110). Ez fél megoldás volt, mert az előző bekezdésben felhozott ok miatt nem tudtam haladni. Elvetettem hát a hibás kód részt, hogy működjön a körkeresés, és tudjam a lényeget tesztelni. Persze csak ideiglenesen hagytam ott, hiszen az egész programra szükség van. De sok időt vett el belátni, hogy a lényeget kell csinálnom.

**Weak model**: Megpróbáltam az angol megfogalmazás után magamtól megcsinálni modell legenerálását. Kezdtem azzal, hogy lefordítottam, majd utána néztem, hátha valaki már megcsinálta. Ezek után találtam rá Neelam Yadav kódjára, aki Tarján Róbert algoritmusát használva kereste az összes scc-t. És mivel nem jó a név memóriám, eztért nem vettem észre hogy más algoritmusát követtem (https://www.geeksforgeeks.org/strongly-connected-components/). Viszont ezzel előre haladtam egy lépést és új dolgokat tanultam, mint a monkey pathcing, ami olyasmi lehet a pythonban, mint más objektum orientált programozásra (oop) alapuló nyelvekben a delegáltak és a kiterjesztő metódusok (nx.DiGraph.dfs\_util = dfs\_util). Egy már meglévő csomag, meglévő osztályát egészítettem ki a saját kézzel írt függvénnyel.

Viszont eközben észrevétlenül hátra felé is léptem, hiszen nagyon lelassultam azzal, hogy megpróbáltam az scc keresést megoldani, és helyére csúcsokat rakni. (#? \*\* rész utáni kód jöhet.) Hátra lépés alatt pedig azt értem, hogy random generált gráfokon teszteltem a kódomat, nem pedig egy ismert gráfon, amiről tudom, hogy milyen eredményt kell várnom (ezzel tudom, hogy egy alapelvet sértettem meg a teszteléshez).

Valamint, hogy körülbelül négy verzió készülhetett el, mire rájöttem, hogy létezik egy mindezt megvalósító NetworkX algoritmus a condensation(https://github.com/networkx/networkx/blob/77c49c16e10693dbe566d20601b28dd2b1e8df03/networkx/algorithms/components/strongly\_connected.py#L342) függvényformájában. Erről pedig kiderítettem, hogy minden tulajdonsága megvan, ami nekünk kell. Leírása szerint minden scc-t meg kell adni neki, különben alapértelmezetten „none” értéket kap, és összeszedi a gráfon magának. Majd megjegyzi az scc-kből ki vagy be vezető éleket, hogy összeköthesse őket. Összesűríti a gráfot, ezért kapta a condensation nevet. És ez egy irányított aciklikus gráfot (DAG) eredményez.

Még egyéb megfigyeléseim is voltak, mint például, hogy érdemes halmazban tárolni az scc-ket. Mivel permutációját is megtalálhatnánk egy scc-nek, de ugyan azt az elemet már nem tárolhatjuk egy halmazban, így nagyon sokkal meggyorsítja a folyamatot. Ha pedig összetett scc-ről beszélünk, akkor egyszerű megfogalmazni, hiszen minden egyszerű scct letárolunk, ha pedig két vagy több scc közt is találunk erős összeköttetést, akkor csak egybe olvasztjuk, és egy komponens lesz az egészből.

Néhány szükséges vizsgálat pedig már eleve benne volt a NetworkX csomagban. Ilyen függvények: is\_semiconnected, is\_weakly\_connected, is\_strongly\_connected, is\_directed\_acyclic\_graph, number\_strongly\_connected\_components, strongly\_connected\_components, topological\_sort, neighbors (linkeket plíz)

Majdnem működő kódokat készítettem Tarján algoritmusa alapján. (#? \*\* rész előtti kód jöhet.)

Majd egy hajszál híján sikerült magamnak is elkészítenem a condensation algoritmusát. De a legegyszerűbb lehetőség mégis csak az előbb említett függvény meghívása.