Gráfbeágyazások és alkalmazásaik

Diplomamunka

Írta: József Csaba

Matematika BSc. elemző szakirány

Témavezető:

Dr. Lukács András
Számítógéptudományi Tanszék
Eötvös Loránd Tudományegyetem, Természettudományi Kar

Eötvös Loránd Tudományegyetem
Természettudományi Kar
2025

Tartalomjegyzék

Köszönetnyilvánítás

Köszönettel tartozom témavezetőmnek, Lukács Andrásnak a dolgozat megírás során adott tanácsaiért és türelméért. Köszönöm azt is, hogy felhívta a figyelmem erre a témakörre, enélkül ez a dolgozat nem valósult volna meg. A szüleim, testvérem és párom által nyújtott támogatás felbecsülhetetlen volt, amiért mindig hálás leszek nekik.

Bevezetés

A tanulmányaim során a két legérdekesebb témakör számomra a gépi tanulás és a gráfok voltak, így mikor megtudtam, hogy a gráfokon végzett gépi tanulás témaköre létezik, természetes módon felkeltette az érdeklődésem. Ez a témakör a gráf struktúrájú adathalmazok kezelésével és a rajtuk tanulni tudó modellek fejlesztésével foglalkozik, ezáltal a matematika és az informatika érdekes határterületét képviseli.

A dolgozatom célja, hogy a jelenleg használt gráfbeágyazási módszerek alapjait bemutassa. A téma általános bevezetése után az elméleti alapokat két konkrét csúcsbeágyazó algoritmus részletesen bemutatásán keresztül folytatom, kiemelten fókuszálva a háttérben húzódó alapelvek részletes elsajátítására. Végül a gyakorlati részben egy kísérleten keresztül szemléltetem az elméletben megismert technikák kivitelezését és használatát, ezzel követve a "Watch one, try one, teach one." módszert, ami egy téma teljes körű és mélyreható megértését nagyban elősegíti.

Mivel nem csak specializált, hanem gyorsan fejlődő területről is van szó, erről a témáról magyar nyelven nehéz részletes és jól strukturált bemutatást találni. Dolgozatomban ezért igyekeztem úgy felépíteni bemutatott kéréskörök tárgyalását, ahogy én szerettem volna látni azokat a témával való ismerkedésem kezdetén. Ennek megfelelően remélem, hogy a dolgozat alapos és értő bevezetést nyújthat a témába a hozzám hasonló érdeklődő hallgatók számára.

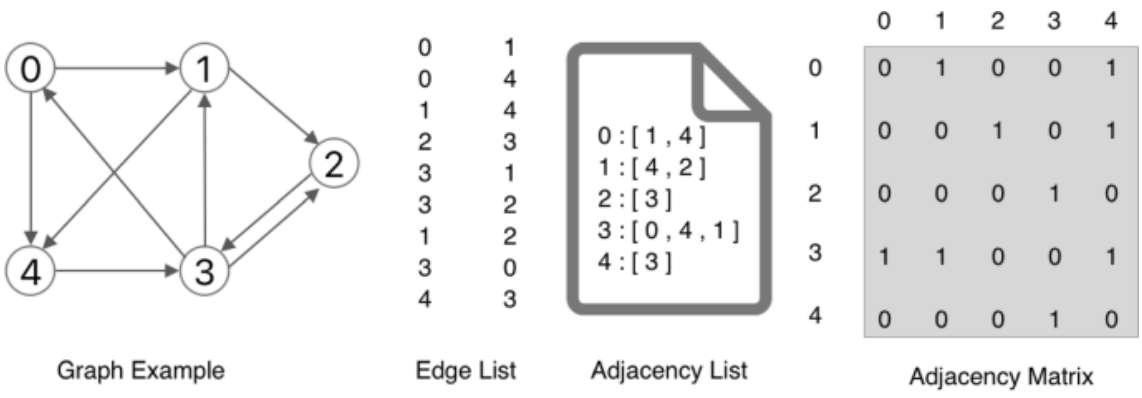
1. fejezet

Beágyazó algoritmusok

1.1. A gráfoktól a beágyazásokig

A gráf típusú adatok magas szintű feldolgozására egyre szélesebb körű igény jelentkezik, mivel a természetes módon gráf formában található adatok gyakoriak sok tudományterületen, illetve a mindennapjaink során is többször találkozunk velük, mint gondolnánk. A teljesség igénye nélkül példának megemlíthetők a különböző biológiai rendszerekre épülő adathalmazok, ilyen a protein-protein interakciók adathalmaza [1] vagy az agyi idegsejtek közötti kapcsolatok konnektomja. Szintén gyakran felmerülnek a jelenlegi szakirodalomban a különböző szenzorok által alkotott hálózatok, ahol egy adott szenzoron keletkező idősorok képeznek egy csúcsot, ezzel az idő dimenzióját is behozva a problémakörbe. Az elmúlt időszakban nagy népszerűségnek örvendő hálózattudományban vizsgált szociális hálók, illetve a nagy nyelvi modellek (LLM) jelenlegi forradalmában fontos szerepet játszó tudásgráfok szintén gráf formában létező adathalmazok. Ezek mind erős motivációt biztosítanak a tudományág fejlődéséhez, folyamatosan érkeznek az újabb és újabb interdiszciplináris felhasználások, ezekkel pedig kéz a kézben járnak a fejlődő számítógépes feldolgozási módszerek, algoritmusok, keretrendszerek.

Ezek megértéséhez és felhasználásához első lépéseként meg kell vizsgálnunk a gráfok adatként való kezelését. Egy gráfot többféleképpen tudunk számítógépek által kezelhető formában tárolni, ezek közül a három leggyakoribb az éllista, a szomszédsági lista, illetve a szomszédsági mátrix (1.1. ábra).



1.1. ábra. Egy gráf különböző tárolási módszerei [2]

Formálisan a következőképpen írhatók le. Adott egy $G(V, E)$ gráf, ahol V a csúcsok, E pedig az élek halmazát jelöli. A gráf n db csúcsot tartalmaz, ennek a jelölése: $|V| = n$. Az első esetben, egy egyszerű éllistában az éleket csúcspáronként adjuk meg, azaz $\forall e \in E$ élhez tartozik egy $[v_i, v_j]$ csúcspár. Egy adott csúcspárhoz tartozó él jelölése e_{ij}. A második eset a szomszédsági lista, ahol a gráf minden csúcsához egy lista tartozik, amiben a csúcshoz tartozó élek kerülnek felsorolásra. A harmadik eset a szomszédsági mátrix. Egy n csúcsú gráf esetén ez a mátrix $n \times n$-es méretű, a következő elemekkel

$$A(i,j) = \begin{cases} 1, & \text{ha } [v_i, v_j] \in E \\ 0, & \text{ha } [v_i, v_j] \notin E. \end{cases} \tag{1.1}$$

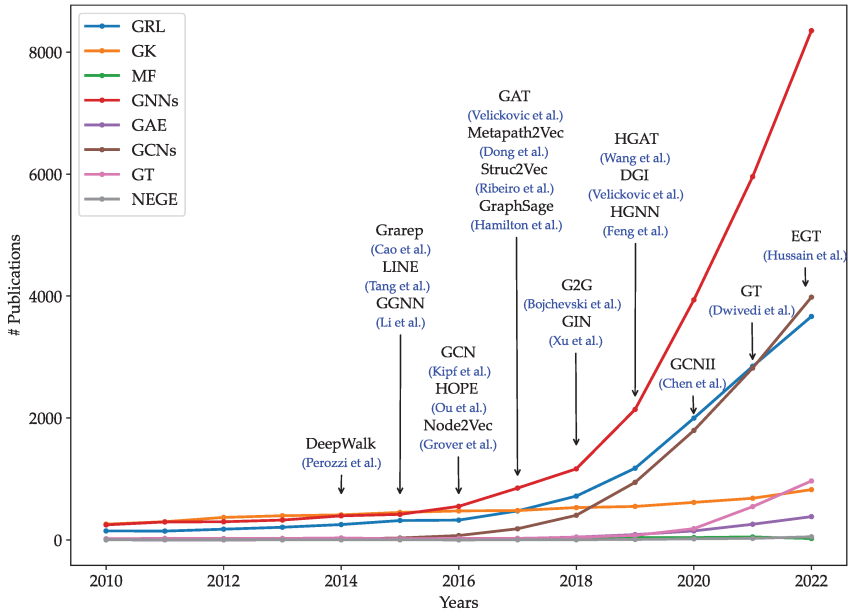
Amennyiben súlyozott gráfról van szó, a bináris jelölés helyett az él súlyát is lehet a mátrix elemeiben tárolni.

Egy tipikus, való életből vett gráf formájú adathalmaznál gyakran előfordul, hogy hatalmas mennyiségű csúcshoz jelentősen kevesebb mennyiségű él társul. Ilyenkor a szomszédsági mátrix az úgynevezett ritka mátrix kategóriába esik. Ezekben az elemek többsége nulla, így ilyenkor a mátrixnál hatékonyabb tárolási módhoz kell folyamodni, különben nagyon sok felesleges számjegy kerül tárolásra, annak ellenére, hogy ezek nem hordoznak igazi információt. Ennek a megoldása lehet a fenti módszerek egyike, vagy a mátrix egy hatékonyabban tárolt verziója. Ilyen például a COO (coordinate list), ahol *sor index, oszlop index, tárolt érték* formában listákban tároljuk a mátrixot. Ez a tárolás az általam használt PyTorch Geometric[3] csomag alapja is.

A gráfokat az előbb felsorolt módokon kifejezve fontos lépést teszünk a számítógéppel való feldolgozásuk felé, azonban ezekkel még csak az adatként való tárolásukat oldottuk meg. A mélyebben rejlő információk és összefüggések kinyerése, megértése és felhasználása sokkal nehezebben megoldható probléma. Itt jönnek be a képbe a gráfbeágyazások és a gépi tanulás.

A gépi tanulás már sok olyan területen bizonyított, ahol az adatok euklideszi térben találhatók. Ilyen a legtöbb táblázatos formában tárolt adat, amivel találkozunk – idősorok, szövegek és képek – hiszen ezek esetében az adat mögött húzódó struktúra mind euklideszi teret képez és ezáltal egyértelműen lehet két adott pont viszonyát számszerűsíteni. Gráfok esetében azonban több probléma is felmerül ebben a megközelítésben, ilyen például a csúcsok közötti többféle éltípus lehetősége, a tetszőleges fokszámok, és a csúcsok definiálatlan sorrendezhetősége. Ezek miatt nem adott semmilyen közös koordináta rendszer amiben rendezni tudjuk az adatokat, vagyis nincs struktúra, amire intuitívan tudunk tanítani egy modellt, így a globális paraméterezhetőség problémákba ütközik. A gráfok tehát nem euklideszi adathalmazok, emiatt a szokásos gépi tanulási modellek alapjai, mint például a konvolúció, amik euklideszi térben könnyen értelmezhetők, itt újradefiniálást igényelnek. Az ehhez hasonló problémák megoldására jött létre a gráf reprezentáció tanulás területe.

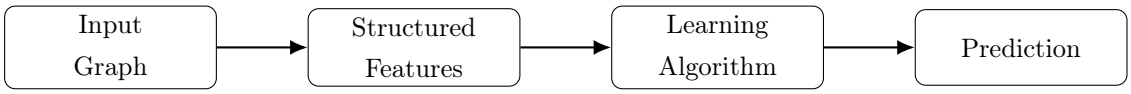
A terület növekvő népszerűsége tagadhatatlan, jól látszik az 1.2 ábrán is, ami a gráfokon történő gépi tanulás szerepének növekedését mutatja éves bontásban a hozzájuk köthető szakmai publikációk számában, illetve a különböző felhasznált módszerek szerint lebontva.



1.2. ábra. A témával kapcsolatos publikációk száma [4]

A fejlődés egyik fő mozgatórugója az vizsgálandó adatok egyre növekvő komplexitása és mérete, amelyek tradicionális gráfelemző módszerekkel már nem kezelhetőek. Az 1.2 ábrán láthatjuk az általam ebben a szakdolgozatban megvizsgált algoritmusokat is, ezek a node2vec, GCN és a GraphSage, amelyek a terület alapjait fektették le.

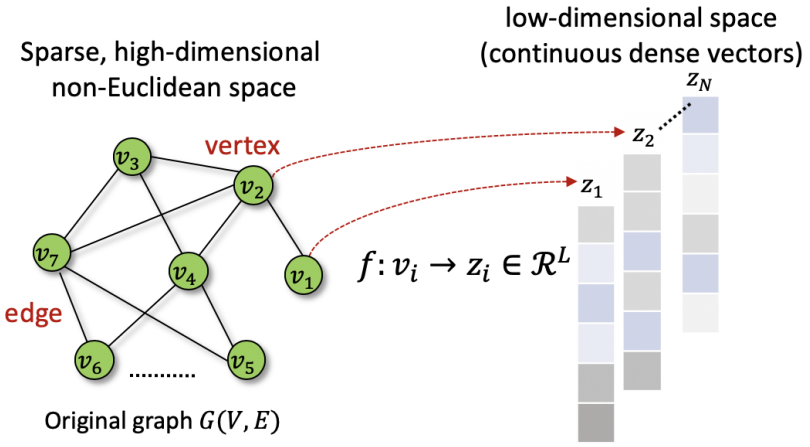
A gráf reprezentáció tanulás egy gyűjtőfogalom, amibe minden olyan módszer beletartozik, melynek célja, hogy a gráfokat egy alacsony dimenziós térbe képezze le úgy, hogy megőrizzék a struktúrájukban, élekben és csúcsokban lévő információkat. Ezeknek a módszereknek az eredményei a gráfbeágyazások, amik egy már a tradicionális modellek számára is tanulható vektorként, leképezett formában, az információvesztést minimalizálva fejezik ki a gráfot. Ezen vektorok legyártása az, ami a konkrét megoldandó feladatot jelenti a gráf reprezentáció tanulás során a modellek számára. Felmerül a kérdés, hogy egy ilyen vektor pontosan miben tud többet, mint a fentebb megismert, gráfok tárolására alkalmas módszerek. Ehhez meg kell értenünk a hagyományos gépi tanulás menetét gráfokon, ami az alábbi 1.3 ábrán látható módon működik.



1.3. ábra. A klasszikus gépi tanulás lépései

Láthatjuk, hogy ennek során a gráfban a kiválasztott tulajdonságokon (feature) hajtjuk végre a tanítást, tehát a modell tanítására kiválasztott input adatok előállításával ki is merül a gráfból való lényegi adatkinyerés. A gráf reprezentáció tanulással ezt a *feature engineering* lépést tudjuk automatizálni azzal, hogy a modellek a tanulás során a gráfra jellemző nem euklideszi információkon (gráf struktúra) is tanulnak, ezzel jelentősen fejlesztve ennek a lépésnek a hatékonyságán. Ha a tradicionális módszerekben meghagyjuk inputnak a teljes gráfot, akkor a csúcsok beviteli sorrendje módosítja a modellünk eredményét, hiába végezzük a tanítást akár ugyanazon a gráfon. Az új beágyazó módszerek ezt is kiküszöbölik, ezzel megint csak megoldva egy korábbi problémát.

Matematikai értelemben a gráfbeágyazás egy olyan f műveletet jelent, amely a gráf információját vektortérbe képezi le. A leképezés vezérelve, hogy a vektortérben megőrizze a gráfstruktúrában található információkat, azaz egy csúcsbeágyazás esetén a gráfban hasonló csúcsok a vektortérben is egymáshoz közel szerepeljenek. Ehhez definiálnunk kell a vektortérbeli és gráfbeli hasonlóságot is az egyes algoritmusok esetében, majd közelíteni azokat a modell tanítása során. A cél, hogy egy $v_i \in \mathcal{V}$ csúcshoz létrejöjjön egy z_i beágyazás egy tetszőleges dimenziójú vektorként, azaz $f : v_i \rightarrow z_i \in \mathbb{R}^L$. Ezt illusztrálja az 1.4 ábra.



1.4. ábra. Csúcsok beágyazása egy vektorérbe[5]

A gráfbeágyazásokat több elv szerint tudjuk kategóriákra bontani [6]. Az első egy gyakorlatorientált osztályozás, ami a feladat típusa és ezzel együtt a felhasznált beágyazás szintje szerint történik. Ez alapján beszélhetünk csúcs-, él- és gráfbeágyazásokról. Az irodalom a gráfbeágyazás kifejezést felcserélhetően használja a csúcsokon, éleken és teljes gráfokon képzett beágyazások leírására, én azonban az egyértelműség kedvéért különbséget fogok tenni ezek között. A dolgozatban a csúcsbeágyazások vannak fókuszban, mivel az erre használt módszerek kibontásával jól lefedhetők a jelenleg használt technikák, logikusan bevezethető a többi beágyazás típus, illetve sok alapvető modell is a csúcsokra fókuszál. A leggyakoribb feladattípusok a csúcs-, él- és gráfklasszifikáció, ahol különböző kategóriákba próbálunk besorolni adott gráfbeli elemeket. A csúcs- és élpredikció szintén fontos probléma, ebben az esetben egy konkrét csúcs vagy él létezését próbáljuk különböző modellekkel megjósolni. Egy adott hálózatban történő közösségek detektálása is kapcsolódó és releváns kérdéskör, ezekre mind kiterjedt irodalom található, már a gépi tanulás korszaka előttről is.

Egy másik lehetséges csoportosítás a homogén és heterogén gráfokon tanuló algoritmusok szétválasztása. A homogén gráfok egyféle csúcs- és éltípust tartalmaznak, míg a heterogén gráfok tetszőleges számút. A heterogén eset kezelése értelemszerűen különböző, kiterjesztett modellstruktúrát igényel a homogén esethez képest, ezért a dolgozatban a homogén esetekre vizsgálunk modelleket. Heterogén esetre jó példák a tudásgráfok, amik sok ma használt AI megoldás alapját képezik, ezért külön említést érdemelnek. A tudásgráfok különböző entitások közötti kapcsolatokat foglalnak össze gráf formájába, amelyben explicit és implicit információk alapján logikai összefüggések találhatók, így az ezekben való élpredikcióhoz extra információkat is figyelembe kell vennünk a szokásos gráfstruktúrán felül. Egy gyakori példájuk az eseménygráf, ahol az élek események, melyek

különböző státuszokat kötnek össze.

Ha a módszertanuk szerint közelítjük meg a beágyazásokat akkor három nagyobb irányt találhatunk, ezek a random séta alapú, a neurális háló alapú és a mátrix faktorizáció alapú beágyazó algoritmusok. Ezek közül az első kettőt vizsgáljuk meg részletesebben, mivel a mátrix faktorizációra épülő módszerek a jelenlegi konklúzió szerint[6] [5] a számítási komplexitásuk miatt nem jól skálázhatóak. Egy szintén fontos kategorizálás a felületes (shallow) illetve mély (deep) beágyazások, amiket később részletesebben kifejtek, itt csak megemlítem őket. A dolgozatban több, a területre nagy hatású és azt a mai napig meghatározó csúcsbeágyazó algoritmuson keresztül ismerjük meg a csúcsbeágyazások konkrét legyártásának lépéseit, illetve a terület ezekhez kapcsolódó alapjait.

1.2. A node2vec algoritmus bemutatása

A node2vec[7] egy gráfokon végzett random bolyongáson alapuló algoritmus, amelyet 2016-ban alkottak meg. Az algoritmus az addig már ismert megoldások és több kulcsfontosságú újítás kombinációjából áll össze, emiatt a mai napig hasznos és informatív példa a témakörrel ismerkedők számára. Az algoritmus alapötletét a Skip-gram [8] algoritmusból emelték át, amely egy NLP (Natural Language Processing) algoritmus. Ez az algoritmus egy szövegben megadott szóhoz prediktálja a környezetében előforduló szavakat, kiindulva a feltételezésből, hogy hasonló szavak hasonló környezetben találhatók. Az eredeti algoritmus szószekvenciákból dolgozott, a gráfok kezeléséhez pedig ennek mintájára bevezették a random sétákat, amelyekkel szekvenciálisan tudjuk vizsgálni a gráfokban létező szomszédsági információkat.

Ahogy fentebb említettük, a vektortérbeli és gráfbeli hasonlóságot definiálnunk kell minden algoritmus esetében. A node2vec alapötlete és célja, hogy a csúcsok random sétákban való együttes előfordulás által meghatározott valószínűségét lehető legjobban megközelítsük a vektortérbeli beágyazásukkal, feltételezve, hogy azon csúcsok, amelyek gyakran fordulnak elő közös random sétákban hasonlónak számítanak, ezért közel kell legyenek egymáshoz beágyazásként is. Formálisan leírva, adott egy u és v csúcspár, z_u és z_v beágyazásokkal. A két csúcs hasonlóságát a vektortérben a skalárszorzatukkal jellemezzük, ezzel kell közelítenünk a P értéket, amit az u csúcsból indított sétán v csúcs megjelenési valószínűségeként definiálunk, tehát:

$$\mathbf{z}_v^\top \mathbf{z}_u \approx P\left(v \mid \mathbf{z}_u\right). \tag{1.2}$$

Egy jó beágyazásban pontosan fejeződnek ki a gráfban található strukturális információk, mind a közeli mint a távolabbi szomszédokkal való viszonylatban, amire a random séták egy rugalmasan paraméterezhető módszert adnak. Az algoritmus másik előnye a hatékonysága, ami a párhuzamosítható részeiből és abból ered, hogy nem kell az összes csúcs páronkénti viszonyát figyelembe venni számítás közben, mivel csak a random sétákban egyszerre előforduló csúcsokat vizsgáljuk.

1.2.1. Az algoritmus részletes működése

Az algoritmus egészének működése kevés helyen található meg egyben kifejtve, mivel az alapok ismerete feltételezett a legtöbb szakirodalomban. Tekintve, hogy ez egy BSc szakdolgozat, így ez megfelelő hely a működés részletezésére a teljes körű megértés és az összefoglalás igényét is figyelembe véve. A pontos algoritmus a következőképpen néz ki.

1. Algorithm A `node2vec` algoritmus.

LearnFeatures *(Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)*

$\pi \leftarrow$ PreprocessModifiedWeights(G, p, q) $G' \leftarrow (V, E, \pi)$

Initialize *walks* to Empty

for *iter* $= 1$ **to** r **do**

  for *all nodes* $u \in V$ **do**

    $walk \leftarrow$ node2vecWalk(G', u, l) Append *walk* to *walks*

$f \leftarrow$ StochasticGradientDescent$(k, d, walks)$

return f

node2vecWalk *(Graph $G' = (V, E, \pi)$, Start node u, Length l)*

Initialize *walk* to $[u]$

for *walk_iter* $= 1$ **to** l **do**

  $curr \leftarrow walk[-1]$ $V_{curr} \leftarrow$ GetNeighbors$(curr, G')$ $s \leftarrow$ AliasSample(V_{curr}, π) Append s to *walk*

return *walk*

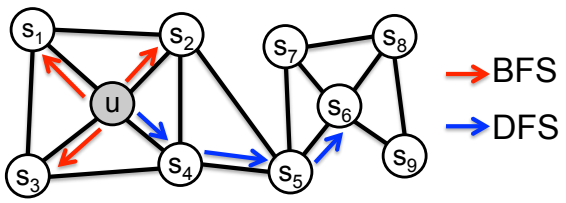
Az algoritmus működését átnézve láthatjuk, hogy három fő részből tevődik össze. Ezek a függvények formájában láthatók a kódban PreprocessModifiedWeights, node2vecWalk és a StochasticGradientDescent néven. Magyarul ezek a következő lépéseket takarják:

1. Előfeldolgozás az átmeneti valószínűségek meghatározásához
2. Random séták generálása
3. Optimalizáció Stochastic Gradient Descent módszerrel (továbbiakban SGD)

Az algoritmus bemenete egy G gráf, amelynek a csúcsaihoz készítjük a beágyazást. A d paraméter a beágyazó vektorok dimenzióját adja meg, az r az egy csúcsból indított random séták száma, az l a séták hossza, a k pedig az egyszerre vizsgált szomszédság méretét jelenti a sétán belül (context). A p és q értékek jelentését az alábbiakban részletesen kifejtem.

A véletlen séták generálása

Bár az algoritmusban a második lépés a séták generálása, érdemes előrébb venni a bemutatásukat, mivel ezzel tisztább kontextusba kerül az átmeneti valószínűségek paraméterezése is. Az algoritmusban egy adott csúcs random séták által létrehozott környezetét $N_S(u)$-val jelöljük, ahol u a kezdőcsúcs, S pedig a használt stratégia, esetünkben a random bolyongás. Az algoritmus kétfajta stratégia között egyensúlyoz, ez a kettő a BFS (Breadth-first Sampling) illetve a DFS (Depth-first Sampling)(1.5 ábra).



1.5. ábra. A BFS és DFS stratégiák u csúcsból indulva ($l = 6$,$k = 3$) [7]

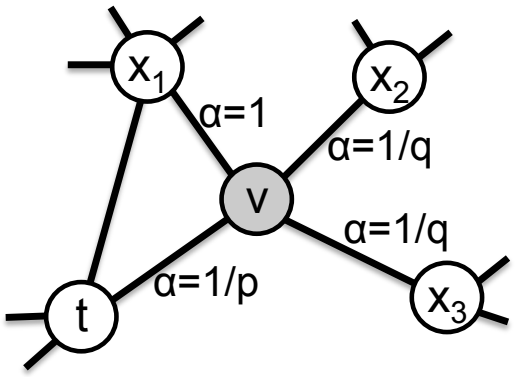
Ennek a kétfajta megközelítésnek az az eredménye, hogy a beágyazások generálásakor a gráfban a random séták által hangsúlyozott tulajdonságokra optimalizálhatunk. Ha a BFS stratégia dominál, akkor a generált séták és az így keletkező beágyazások is a szorosabb közösségek, csúcscsoportok információit tartalmazzák nagyobb pontossággal, feltéve, hogy a hasonló csúcsok egymáshoz közel találhatók egy gráfban (homofília). Ha a DFS stratégia dominál, akkor a séták és a beágyazások is tartalmazzák a gráfban egymástól távolabb lévő csúcsok információt, jobban tudják tükrözni, ha két csúcs strukturálisan hasonló szerepet tölt be a gráfban, még akkor is, ha nincsenek közvetlenül egymás mellett. Ahhoz, hogy a pontos sétákat legyártsuk, ki kell számolnunk az átmeneti valószínűségeket, amelyek a két stratégia közötti egyensúlyt meghatározzák.

Az átmeneti valószínűségek kiszámítása

Első lépésben a következő valószínűségeket határozzuk meg egy c_i elemekből álló séta során, ahol éppen az u csúcsból lépünk v-be

$$P(c_{i+1} = v \mid c_i = u) = \begin{cases} \frac{\pi_{vu}}{Z} & \text{ha } (v,u) \in E \\ 0 & \text{egyébként.} \end{cases} \tag{1.3}$$

Súlyozott élek esetében az él súlyát adjuk meg valószínűségnek, azaz $\pi_{vx} = w_{vx}$, súlyozatlan esetben pedig $w_{vx} = 1$ alkalmazandó. A Z egy általános normalizáló faktor, amivel a valószínűségek összegét 1-re állítja a modell. Ezután kell figyelembe vennünk a két alkalmazandó stratégiát, amihez definiálunk egy másodrendű véletlen bolyongást két paraméterrel. Ezek a már említett p és q, amelyek a bolyongás irányát szabályozzák a következő módon: Tegyük fel, hogy a (t,v) élen áthaladva, jelenleg a v csomóponton tartózkodik az algoritmus (lásd az 1.6 ábrát).



1.6. ábra. A node2vec átmeneti lépése egy csúcsban

A következő lépés kiválasztásához az algoritmus kiértékeli a v-ből kimenő (v,x) élekhez tartozó átmeneti valószínűségeket az alábbi képlettel:

$$\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$$

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{1.4}$$

Az $\alpha_{pq}(t,x)$ súlyfüggvényben található d_{tx} a t és x közötti legrövidebb út hosszát jelenti, a p paraméter a visszatérési (return) paraméter, q pedig a továbblépő (in-out) paraméter. A két

paraméter működésének intuitív magyarázata az, hogy a séta kezdő csúcsának közelében való maradást irányítják, azaz a fentebb megismert BFS és DFS séta között tudjuk velük súlyozni az algoritmust. Nagy p érték esetén a modell a továbblépést hangsúlyozza (DFS), nagy q érték esetén pedig a csúcs közelében maradást (BFS) és fordítva. Miután kiszámoltuk a valószínűségi átmeneteket, az eddigiek alapján legeneráljuk a sétákat, ezzel legyártva az $N_S(u)$ szomszédságokat minden $u \in V$ csúcsra.

Stochastic Gradient Descent (SGD)

Az SGD egy iteratív algoritmus, amelynek lényege, hogy egy konkrét modell paramétereit inputként használva egy célfüggvényben a modell paramétereit tudjuk optimalizálni a tanító adathalmazon vett gradiens kiszámolásával. A bementi térben (input space) vett gradiens a legnagyobb növekedés irányát mutatja a célfüggvényben, erre ellenkező irányú lépést téve értelemszerűen a legnagyobb csökkenést kapjuk. A lépés méretét a tanulási sebesség (learning rate) paraméter állítja. A módszer gyakran alkalmazott neurális hálók tanítására, ilyenkor a szokásos működés esetében a háló paraméterei alkotják a célfüggvény bemenetét, míg a paramétereit a tanító adathalmaz elemei képzik. Mivel a node2vec a beágyazásokat közvetlenül tanulja, így itt a célfüggvény bemenetei a csúcsbeágyazások, amik kezdésnek random értékekkel vannak inicializálva majd közvetlenül ezeken történik az optimalizálás.

A gradienst számolhatjuk az összes rendelkezésre álló tanító adaton (epoch), vagy pedig az adatok egy részén (batch), kihasználva, hogy ez már elégséges ahhoz, hogy egy megfelelő irányba konvergáljon a modell, ezzel pedig jelentősen növelve a számítás sebességét. Erre utal a modell nevében a stochastic szó. Matematikai formában az SGD egy lépése egy z_u csúcsbeágyazás, $\mathcal{L}$ célfüggvény és η tanulási sebesség esetén:

$$z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}. \tag{1.5}$$

A célfüggvény, amire alkalmazzuk az SGD módszert a node2vec esetén:

$$\arg\max_z \sum_{u \in V} \log Pr(N_S(u) \mid z_u), \tag{1.6}$$

azaz egy olyan beágyazást akarunk adott u csúcshoz megtanulni, ami a random séták által meghatározott szomszédságában előforduló csúcsokra a legnagyobb összesített valószínűséget adja. Az eredeti cikkben[7] z_u helyett $f(u)$ szerepel és $\max f(u)$ amire a szerzők optimalizáltak. Mivel az f függvény csak egy egyszerű hozzárendelést jelent, ami egy adott u csúcshoz a $z_u \in R^d$ beágyazó vektort rendeli, enélkül könnyebb követni a működést és jobban szem előtt marad az a tény, hogy az algoritmus közvetlenül a beágyazásokat optimalizálja. Következő lépésben, feltételezve, hogy egy csúcs előfordulása a szomszédságban független egy másik csúcs előfordulásától, a szomszédság valószínűségét a következő formában adjuk meg:

$$Pr(N_S(u) \mid z_u) = \prod_{v \in N_S(u)} Pr(v \mid z_u). \tag{1.7}$$

Ezután kihasználva a $log(ab) = \log(a) + \log(b)$ azonosságot, illetve átalakítva a standard minimalizálási formára, a következő formát kapjuk:

$$\arg\min_z \mathcal{L} = \sum_{u \in V} \sum_{v \in N_S(u)} -\log\left(P(v \mid \mathbf{z}_u)\right). \tag{1.8}$$

Láthatjuk, hogy a pontos számítás elvégzéséhez tovább kell gondolnunk az egy csúcspárhoz tartozó valószínűséget, amit $\mathbf{z}_v^\top \mathbf{z}_u \approx P\left(v \mid \mathbf{z}_u\right)$ formában definiáltunk korábban (1.2 egyenlet). Ebben a lépésben kötjük össze matematikailag a beágyazásokat a random sétákban előforduló valószínűségükkel. A modell erre a softmax függvényt használja, amellyel az u kezdőcsúcs összes másik gráfbeli csúccsal vett valószínűségi eloszlását képezi a beágyazó térben a hasonlóságuk alapján:

$$P(v \mid \mathbf{z}_u) = \frac{\exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^\top \mathbf{z}_n)}. \tag{1.9}$$

Az eredeti egyenletbe behelyettesítve elérkezünk a modell legkifejezőbb leírásához, ahol egyszerre láthatjuk, hogy az összes u csúcsból kiindulva, az összes v szomszédra nézve hogyan számolja ki az a valószínűségeket az egyes szomszédságokhoz, illetve azt, hogy az egy random sétában előforduló csúcsok beágyazásainak az együttes megjelenése hogyan csökkenti a negatív veszteséget.

$$\arg\min_z \mathcal{L} = \sum_{u \in V} \sum_{v \in N_S(u)} -\log\left(\frac{\exp\left(\mathbf{z}_u^\top \mathbf{z}_v\right)}{\sum\limits_{n \in V} \exp\left(\mathbf{z}_u^\top \mathbf{z}_n\right)}\right) \tag{1.10}$$

A fenti egyenletben két egymásba ágyazott szummában is az összes V beli csúcson iterálunk végig, ezzel $\mathcal{O}(|V|^2)$ futási időt elérve. Ennek a megoldására találták ki a negatív mintavételezést amelyben a következő módon közelítjük közelítjük a fenti eloszlást:

$$\frac{\exp\left(z_u^\top z_v\right)}{\sum\limits_{n \in V} \exp\left(z_u^\top z_n\right)} \approx \log\left(\sigma\left(z_u^\top z_v\right)\right) + \sum_{i=1}^{k} \log\left(\sigma\left(-z_u^\top z_{n_i}\right)\right), \quad n_i \sim P_V. \tag{1.11}$$

Ezzel a trükkel a teljes gráfon való normalizálást cseréljük le k darab negatív, a gráfból fokszámarányos P_V eloszlásból vételezett z_{n_i} csúccsal való normalizálásra, így jelentősen csökkentve a futási időt, de még mindig eredményesen reprezentálva a gráfban létező, nem hasonló csúcsokat is. Gyakorlatban így egy milliós csúcsszámú gráfnál is elég $k = 5 - 20$ közötti értékkel dolgozni. Az algoritmus futási idejét szintén csökkenti, hogy az átmeneti valószínűségek előre számíthatók, a random séták pedig párhuzamosan generálhatók, mivel nem függnek egymástól semmilyen módon.

Tekintve, hogy az élek predikciója egy nagyon gyakori gráf reprezentációs feladat, emiatt megjegyzendő, hogy mivel a node2vec algoritmus alapvetően egy gráf struktúrát tanuló algoritmus, így ha egy élt csúcspárként fogunk fel akkor természetes módon tudjuk az algoritmust az élekre kiterjeszteni. Ehhez definiálunk egy $\circ$ bináris operátort adott u és v csúcs vektorai között, ezzel generálva egy $g(u, v)$ reprezentációt, ahol $g : \mathcal{V} \times \mathcal{V} \to \mathbb{R}^{d'}$, d' pedig az (u, v) párhoz tartozó reprezentáció dimenziója. A $\circ$ operátor lehet egy számtani közép, elemenkénti szorzat vagy tetszőleges egyéb bináris művelet. Ezeket minden csúcspárra legyártva megkapjuk a feladathoz szükséges kiterjesztést az élekre. Bár a node2vec sokat javított az addigi beágyazási módszereken, azonban alaphiányossága, hogy csak felületes (shallow) csúcsbeágyazásokat tud képezni.

1.3. A shallow beágyazások hátrányai

Az eddig megismert esetben a modell egy adott csúcshoz egy beágyazást rendelt, amit utána közvetlenül optimalizált. Ennek a megközelítésnek sok felhasználása van és több másik gyakran használt modell is alkalmazza, az egyszerűségük miatt viszont a legtöbb ilyen framework transzduktív. A transzduktív algoritmusok egy fix gráfra generálnak konkrét beágyazásokat, azaz nincs olyan paraméterük amivel tudnának általános információt tanulni. Ebből adódóan nem általánosíthatók több gráfra, illetve a tanulás során nem látott csúcsokra nem tudnak később beágyazást

generálni. Egy dinamikus hálózatba újonnan bekerülő csúcs esetén tehát újra kell tanítani az algoritmust az egész gráfon, ami nagyon költséges művelet nagy adathalmazok esetében. Az is logikusan következik, hogy a gráf méretével az optimalizálandó beágyazások összesített mérete is növekszik $O(|V|)$ tempóban, ami a több milliós nagyságrendű gráfoknál sokszor teljesen tarthatatlan lenne, főleg ha figyelembe vesszük az előbb említett újratanítás szükségességét.

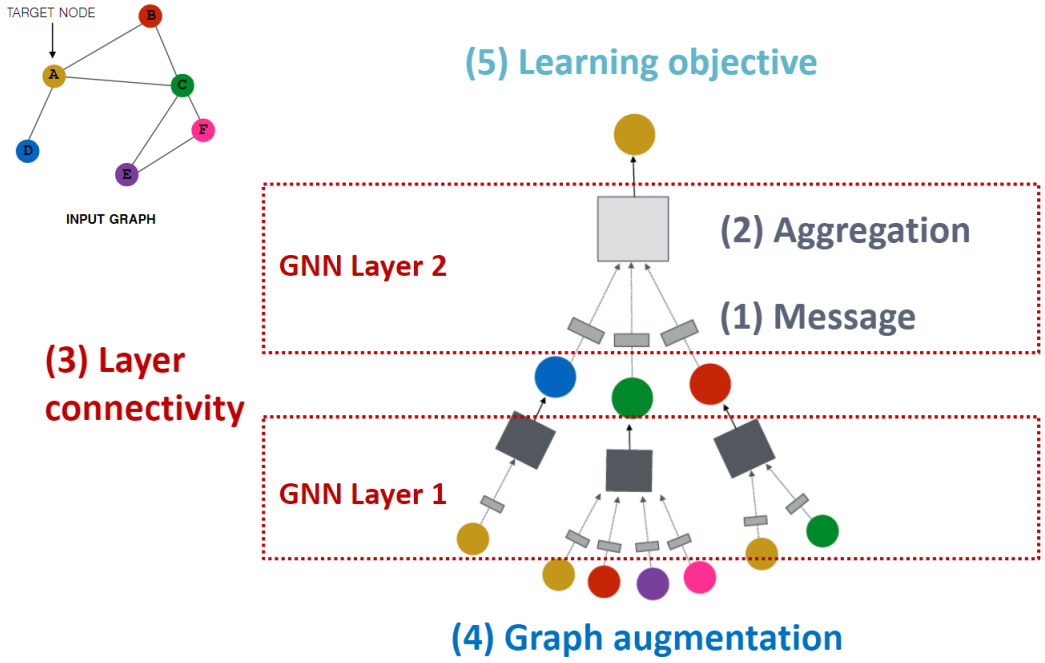
Egy másik probléma ami feltűnik az eddig megismertekkel kapcsolatban, hogy nagy gráfokban az egymástól távol lévő, de strukturálisan hasonló csúcsokat kis eséllyel találják meg a random séták a természetükből adódóan. Ezzel jelentősen megnő annak az esélye, hogy egymástól szignifikánsan eltérő beágyazás generálódik hozzájuk, így ignorálva a tényt, hogy a gráfon belül hasonló szerepet betöltő csúcsokról beszélünk. Szintén hiányzó tulajdonsága ezeknek a modelleknek, hogy az egyes csúcsokhoz illetve élekhez tartozó attribútumokat figyelembe vegyék tanuláskor, ezzel nagyon sok lényeges információt veszítve a gráfból.

Ezen hibák ellenére is gyakran alkalmazott módszerek ezek, fix méretű gráfokon beágyazások generálására továbbra is alkalmasak. Az így kapott eredményeket később akár inputként is lehet használni egy másik, komplexebb beágyazó modell tanítása során, hiszen a struktúrát alapvetően hatékonyan reprezentáló beágyazásokról beszélünk.

Az előbb felsorolt problémákra kínálnak megoldás a mély beágyazó módszerek, amiket a következő fejezetekben a gráf neurális hálókon és a GraphSAGE[9] algoritmuson keresztül fogunk részletezni. Ezek az algoritmusok orvosolják a felsorolt problémákat, azaz felhasználják a csúcsok saját információt a beágyazások legyártásához és a tanult paraméterek miatt képesek a tanítás során nem látott csúcsok beágyazására is. Ezekkel a megoldásokkal és azzal, ahogyan a csúcsinformációkat aggregálják, jelentősen kiterjesztik a beágyazások információtartalmát és a növelik a modellezés hatékonyságát.

1.4. A gráf neurális hálók bemutatása

A GraphSAGE algoritmus megértése és bemutatása előtt érdemes a gráf neurális hálók (GNN) alapjait áttekinteni, mivel ez a modell tekinthető egy gráf neurális hálónak is. A GNN alapokra épülő algoritmusok ezen felül is sokszor visszatérő elemei a gráf reprezentáció tanulás témakörének, ezért nem lehet őket kikerülni. Az intuíció kialakításához itt talán a szokásosnál is jobban hozzájárulnak a szemléltető ábrák, ezért a részletes leírás előtt nézzük meg a gráf neurális hálókban tipikusan előforduló rétegeket.



1.7. ábra. A GNN rétegek felépítése [10]

Jól látható, hogy egy tipikus GNN réteg két fő lépésre bontható le. A rétegekben a csúcsok a szomszédjaikból érkező információt fogadják – illetve később ugyanígy a saját információjukat továbbküldik – ez az üzenettovábbítás (message passing). Az egy csúcsba befutó üzeneteket egy összesítő művelet (aggregation) kombinálja, aminek segítségével az adott csúcs frissíti a saját állapotát az összesített üzenetek alapján. Később a GraphSAGE algoritmusnál használt 1.9 ábrán szintén jól látható a csúcsonként történő információfrissítés menete. Egy-egy konkrét gráf neurális háló struktúra sokszor csak a különböző módon implementált üzenettovábbító és aggregáló függvényekben tér el egymástól. Ezekben a modellekben a cél egy $h_v^{(l)}$ reprezentáció elkészítése az l.-ik réteg eredményeként, minden $v \in V$ csúcshoz. A réteg fent ábrázolt működését a következő módon írhatjuk le egy u szomszédos csúcsokból v csúcsba érkező m üzenet esetén, ahol v csúcs az l.-ik rétegben található:

$$m_u^{(l)} = \mathbf{MSG}^{(l)}\left(h_u^{(l-1)}\right), \quad u \in \{\mathcal{N}(v) \cup v\}. \tag{1.12}$$

Minden u csúcs generál egy üzenetet továbbküldésre az előző rétegből érkező információ alapján, ezek az üzenetek és a v csúcs saját eddigi állapota (h_v^{l-1}) képezik az új reprezentáció előállításának alapját. Az üzeneteken egy egyszerű lineáris réteg esetében egy W és B mátrixszal módosítunk, tehát:

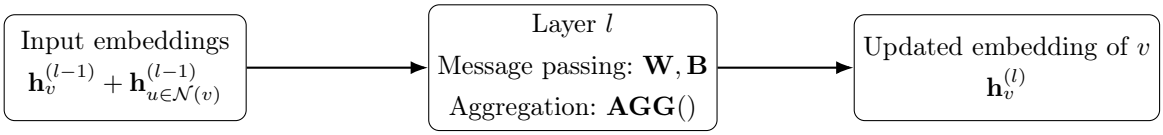
$$m_u^{(l)} = \mathbf{W}^{(l)} h_u^{(l-1)} \qquad \text{és} \qquad m_v^{(l)} = \mathbf{B}^{(l)} h_v^{(l-1)}, \tag{1.13}$$

ahol W és B a modell paraméterei. Ezek egy-egy konkrét réteghez tartozó, lineáris transzformációkat végrehajtó mátrixok, amelyek a szomszédos csúcsok illetve a saját csúcs beágyazásának minőségét szabják meg egy-egy rétegben. Ezzel megvannak az átadandó információk, ezekre alkalmazzuk az aggregáló függvényt:

$$h_v^{(l)} = \mathbf{AGG}^{(l)}\left(\{m_u^{(l)}, u \in \mathcal{N}(v)\}, m_v^{(l)}\right). \tag{1.14}$$

Ezzel a lépéssel az $l-1$. rétegben, az összes v-vel szomszédos u csúcsból összegyűjtve a csúcsbeágyazásokat, megkapjuk a keresett $h_v^{(l)}$ reprezentációt, amibe már belekerült v saját beágyazása

is. Mivel a transzformált m üzenetre alkalmazzuk az aggregáló függvényt, így elérjük, hogy minden csúcshoz a körülötte lévő gráfstruktúra alapján érkezzen egy egyedileg állítható üzenet, amibe a tanulás során beépül az információ.



1.8. ábra. Egy GNN rétegben történő műveletek

Észrevehetjük, hogy a tanult paraméterek egy-egy rétegre vonatkoznak, így a számuk független lesz a hálózat méretétől, emiatt jól skálázható a modell. Ennek köszönhetően lesz induktív is, azaz a már tanított hálóba új csúcsot beküldve, ugyanazokat a paramétereket alkalmazva kaphatunk egy beágyazást rá – mindezt anélkül, hogy a teljes hálózatra vonatkozó beágyazásokat újra kellene számolni.

Az aggregáló függvénynek permutáció invariánsnak kell lennie – mivel alapesetben nem teszünk különbséget a szomszédok között, ez nem szabad, hogy befolyásolja az eredményt. Az aggregáló függvények különböző verzióit a GraphSage algoritmusnál részletesebben is vizsgálom. Általánosan elmondható, hogy sokszor használjuk a számtani közepet, az összeadást és a maximumot aggregáláskor. Az aggregálás után rétegenként egy nem lineáris aktivációs függvényt alkalmazva eljutunk a végső neurális háló struktúrához. Ez a függvény lehet például egyike a bevett ReLU vagy Sigmoid függvényeknek, esetünkben az utóbbi kerül a modellbe, azaz:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{1.15}$$

$$h_v^{(l)} = \sigma\left(\mathbf{AGG}^{(l)}\left(\{m_u^{(l)},\ u \in \mathcal{N}(v)\},\ m_v^{(l)}\right)\right). \tag{1.16}$$

Az alapok után most nézzünk meg pár fontosabb modellt amik az eddig átvettekre épülnek.

1.4.1. A gráf konvolúciós háló

Az egyik nagyon népszerű GNN algoritmus a gráf konvolúciós háló (GCN). Esetünkben ez egy jó példa lesz arra, hogy megnézzük hogyan épül fel egy modell az előbb felvázolt logika mentén, illetve, hogy megnézzük a számítások mátrix formában történő kezelését is. A GCN egy l.-ik réteghez tartozó függvénye:

$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}_{l-1}\sum_{u\in\mathcal{N}(v)}\frac{\mathbf{h}_u^{(l-1)}}{|\mathcal{N}(v)|} + \mathbf{B}_{l-1}\mathbf{h}_v^{(l-1)}\right), \quad \forall l \in \{1,\dots,L\}. \tag{1.17}$$

A hálóban összesen L réteg van, a végső beágyazást az utolsó rétegben kapjuk meg, tehát $z_v = h_v^L$. Az üzenettovábbítást és aggregálást itt is megfigyelhetjük. Az üzenet kiszámításánál a fokszámmal való normalizálást vezeti be a modell:

$$\mathbf{m}_u^{(l)} = \frac{1}{|\mathcal{N}(v)|}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \tag{1.18}$$

az aggregálás pedig egy egyszerű szummázást jelent. Ezzel a kettővel gyakorlatilag számtani közepet veszünk a csúcsok beágyazásain. Az aggregálás megvalósítása mátrixműveletekkel történik, ami

hatékony kivitelezést biztosít a modell működéséhez. Ehhez egy H^l mátrixban soronként tároljuk az egyes csúcsok l-ik szinthez tartozó beágyazásait:

$$H^{(l)} = \begin{bmatrix} h_1^{(l)} & \dots & h_{|V|}^{(l)} \end{bmatrix}^\top. \tag{1.19}$$

Ebből következően $H \in \mathbb{R}^{|V| \times d}$, ahol d a beágyazások dimenziója. Ezután egy konkrét csúcshoz ebből kell kiválasztanunk a megfelelő, szomszédokhoz tartozó beágyazásokat. Ezt könnyen megtehetjük a gráf A szomszédossági mátrixával, a következő módon:

$$\sum_{u \in N_v} h_u^{(l)} = A_{v,:} \cdot H^{(l)}. \tag{1.20}$$

Definiálunk egy $D \in \mathbb{R}^{|V| \times |V|}$ diagonális mátrixot is, aminek az átlójában az egyes csúcsok fokszámai találhatók, azaz $d_{ii} = d(v_i) = |N(v_i)|$. Ennek az inverzét és az előző egyenletet felhasználva megkapjuk a következő réteghez tartozó H^{l+1} mátrixot.

$$D_{v,v}^{-1} = \frac{1}{|\mathcal{N}(v)|} \tag{1.21}$$

$$H^{(l+1)} = D^{-1} A H^{(l)} \tag{1.22}$$

Ezután bevezetünk egy $\tilde{A}$ rövidítést a következőre:

$$\tilde{A} = D^{-1} A. \tag{1.23}$$

A fenti függvény újraírása mátrix formába tehát:

$$H^{(l)} = \sigma\left(\tilde{A} H^{(l-1)} \mathbf{W}_{l-1}^\top + H^{(l-1)} \mathbf{B}_{l-1}^\top\right) \tag{1.24}$$

A mátrix formában történő számítás nem minden típusú aggregáló függvény esetében elérhető, viszont az egyszerűbb esetekben effektív számítást tesz lehetővé.

1.4.2. A szomszédok súlyozása

Az eddigi modellek nem tettek közvetlenül különbséget egy adott szomszédságon belül a csúcsok fontosságai között, annak ellenére sem, hogy logikus feltételezni ezek létezését. Ha az egyes csúcsok jelentőségét súlyozni akarjuk a modell tanítása során, akkor a GAT (Graph Attention Network) implementációjához tudunk fordulni:

$$h_v^{(l)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)}\right). \tag{1.25}$$

Ebben a modellben egy α paraméteren keresztül kerül bevezetésre az koncepció, miszerint ha egy jól definiált fontosságot tudunk az egyes csúcsokhoz rendelni, akkor a modell a gráfban lényeges információkat tartalmazó részekre tudja fordítani a számítási kapacitását. Az eddig megvizsgált modellekben az egyes csúcsok fontossága implicit módon volt definiálva a fokszámokon keresztül, tehát a modellek számára ez az információ nem volt direkt tanulható. Ebben a modellben az érték explicit módon számolódik a következő képlettel, ahol e_{uv} az u csúcsból érkező üzenet fontosságát jelenti v számára, az a pedig az a mechanizmus amivel kiszámoljuk az üzenetekből az értékét.

$$e_{vu} = a\left(W^{(l)} h_u^{(l-1)},\ W^{(l)} h_v^{(l-1)}\right) \tag{1.26}$$

Erre a számításra alkalmas lehet egy egyrétegű háló, ezt behelyettesítve megkapjuk a számítás menetét.

$$e_{uv} = \textbf{Linear}\left(\textbf{Concat}\left(W^{(l)}h_u^{(l-1)},\ W^{(l)}h_v^{(l-1)}\right)\right) \tag{1.27}$$

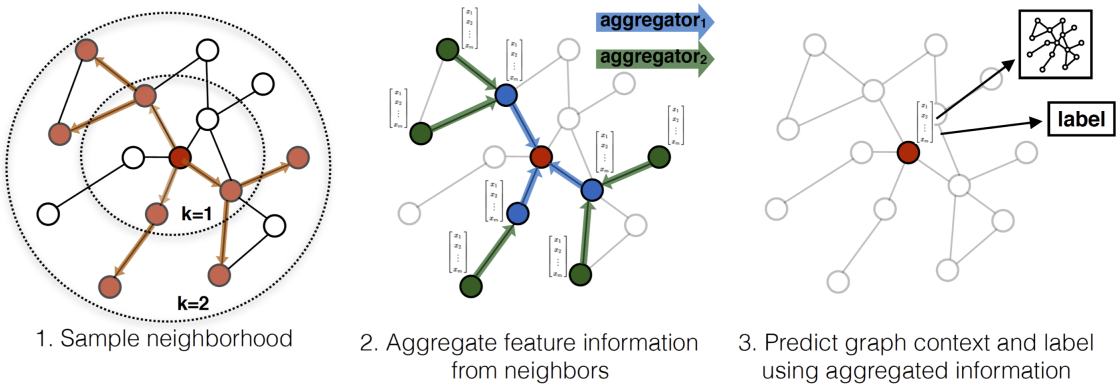
Az α értéket ebből közvetlenül kapjuk meg, miután a softmax függvény segítségével normalizáltuk a szomszédságra, azaz:

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k\in\mathcal{N}(v)} \exp(e_{vk})}. \tag{1.28}$$

A súlyozás értékei sok esetben nehezen konvergálnak, emiatt érdemes többszörösen inicializálni őket és utána ezeket párhuzamosan tanítani a modell tréningelése során, mivel az aggregált értékük hatékonyabban közelíti a keresett optimumot. Az eredeti és a súlyok számításához bevezetett paramétereken végezhető együttes tanítás, illetve a fix méretű paraméternövekedés miatt az algoritmus továbbra is hatékonyan tud működni a fontossági súlyok kiszámításával együtt is. Ezzel a modellel lezárul a gráf neurális hálókról szóló összefoglaló. A következő fejezetben mélyebben megvizsgáljuk egy konkrét modell működését. Ez a modell a GraphSAGE, ami a használt módszereit tekintve az eddigiekre épít. Szintén ezen keresztül mutatom be egy modell tanításának felügyelt és felügyelet nélküli esetét.

1.5. A GraphSAGE algoritmus

A GraphSAGE (Graph-SAmple-and-aggreGatE) algoritmus[9] neve egy rövidítés, amiben az algoritmus működési elemei szerepelnek. Ezek a "sample" és "aggregate" azaz mintavétel és aggregálás, ezek a lépések az alábbi 1.9 ábrán láthatók.



1.9. ábra. A GraphSAGE algoritmus működése[9]

Ez a modell képes megtanulni a strukturális információkat és a csúcsszinten lévő tulajdonságokat is egy adott csomópont előre definiált méretű szomszédságában, emiatt alkalmazható csúcsinformációt opcionálisan tartalmazó gráfra is. Ha a node2vec-hez akarjuk hasonlítani akkor láthatjuk, hogy itt a konkrét beágyazások helyett közös aggregáló függvényeket tanítunk, emiatt új csúcsokra is alkalmazható lesz az eredmény. Az algoritmus pontos leírása az alábbi módon néz ki.

2. Algorithm A `GraphSAGE` algoritmus

GraphSAGE *($\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Features $\{\mathbf{x}_v\}$, Depth K, Weight matrices $\mathbf{W}^k$, Non-linearity σ, Aggregator functions $AGGREGATE_k$, Neighborhood function $\mathcal{N}$)*

foreach $v \in \mathcal{V}$ **do**
  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v$
for $l = 1$ **to** K **do**
  foreach $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^l \leftarrow \text{AGGREGATE}_l\left(\left\{\mathbf{h}_u^{l-1}, \forall u \in \mathcal{N}(v)\right\}\right)$
    $\mathbf{h}_v^l \leftarrow \sigma\left(\mathbf{W}^l \cdot \text{CONCAT}\left(\mathbf{h}_v^{l-1},\ \mathbf{h}_{\mathcal{N}(v)}^l\right)\right)$
  foreach $v \in \mathcal{V}$ **do**
    $\mathbf{h}_v^l \leftarrow \mathbf{h}_v^l / \|\mathbf{h}_v^l\|_2$
foreach $v \in \mathcal{V}$ **do**
  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K$
return $\{\mathbf{z}_v\}_{v \in \mathcal{V}}$

Az eddigi gondolatmenetünket folytatva láthatjuk, hogy a GCN működéséhez képest kettő újítást vezet be ez az algoritmus. Az egyik az, hogy az aggregáló függvényre több opciót kínál a már megismerteken felül, a másik pedig, hogy a csúcsok saját beágyazását konkatenálja az aggregált információ mellé, ezzel fejlesztve a beágyazások kifejezőkészségét. A számítás két lépésből tevődik össze az alább látható egyenletek szerint.

$$\mathbf{h}_{\mathcal{N}(v)}^{(l)} \leftarrow \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)},\ \forall u \in \mathcal{N}(v)\right\}\right) \tag{1.29}$$

$$\mathbf{h}_v^{(l)} \leftarrow \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}\left(\mathbf{h}_v^{(l-1)},\ \mathbf{h}_{\mathcal{N}(v)}^{(l)}\right)\right) \tag{1.30}$$

$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}\left(\mathbf{h}_v^{(l-1)},\ \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)},\ \forall u \in \mathcal{N}(v)\right\}\right)\right)\right) \tag{1.31}$$

Az első lépésben (1.29) aggregálunk a szomszédos csúcsokon, a másodikban (1.30) pedig hozzáfűzzük a csúcs saját beágyazását. Ebbe behelyettesítve, az egész vektorra alkalmazva a szokásos lineáris és nem lineáris transzformációkat megkapjuk a pontos egyenletet(1.31). Minden ilyen lépésnél eggyel távolabbi környezetből érkező információk aggregálódnak egy adott csúcsban, a különböző aggregáló függvényeket lentebb részletezem. Szintén egy újítás az algoritmusban a beágyazó vektorok ℓ_2 hosszukkal történő normálása. Ezt minden rétegben elvégzi a modell, ezzel elérve, hogy mindegyik vektor egység hosszú legyen. Ez bizonyos esetekben tud segíteni a modell teljesítményén, mert így a nagyságrendi különbségeket kisimítjuk a vektorok között:

$$\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \quad \forall v \in V. \tag{1.32}$$

Az algoritmus futási idejének korlátozására egy fix méretű, szomszédságból vett mintával van lehetőség. Ilyenkor az $\mathcal{N}(v)$ egy előre definiált méretű, egyenletes eloszlású minta az $\{u \in V : (u, v) \in E\}$ halmazból, amit minden k iterációban újra generál az algoritmus, ezzel elkerülve a legrosszabb esetben $\mathcal{O}(|V|)$ futási időt egy adott batch-re nézve. Az új időkomplexitás ezzel a korláttal $\mathcal{O}\left(\prod_{i=1}^{K} S_i\right)$, ahol S_i, $i \in \{1, \ldots, K\}$. K egy felhasználó által állítható érték, amivel a szomszédságban tett lépések mennyiségét állítjuk, azaz, hogy milyen mélységig vizsgálja a modell egy adott csúcs környezetét, ez gyakorlatban a $K = 2$ értéket vesz fel alapesetben. S_i az adott lépésben a mintába bekerülő csúcsok halmaza, amire a gyakorlatban használt korlát az $|S_1| \cdot |S_2| \leq 500$. A GraphSage a csúcsreprezentációk közötti kapcsolat definiálásán keresztül ugyanúgy kiterjeszthető az élekre történő tanulásra mint a node2vec algoritmus.

1.5.1. A modell veszteségfüggvénye

A GraphSAGE algoritmust felügyelt (supervised) és felügyelet nélküli (unsupervised) tanítással is lehet tanítani, ezért ez egy jó lehetőség arra, hogy megvizsgáljuk ezt a két irányt a csúcsklasszifikáció feladatán keresztül. A tanítás lényege, ugyanúgy ahogyan eddig, az adott modell által generált beágyazás átadása egy veszteségfüggvénynek, ezzel kiértékelve a teljesítményét. Ezután a már megismert SGD algoritmussal módosítunk a paramétereken vagy addig, amíg a modell predikciós pontossága stabilan növekszik, vagy pedig egy előre meghatározott számú epoch elérésig. Csúcsklasszifikációnál, felügyelt esetben a csúcsokhoz tartozó címkékhez képest tudjuk számítani a veszteséget. Ennek az általános formája egy f beágyazó függvény esetén, θ paraméterekkel:

$$\min_{\Theta} \mathcal{L}(y, f_\theta(\mathbf{z}_v)). \tag{1.33}$$

Ebben a formában egyszerűen átlátható a feladat, miszerint a beágyazó modell paramétereit kell úgy változtatnunk, hogy a csúcsok valódi, y-al jelölt értékei minél jobban közelíthetők legyenek a generált beágyazások alapján. A node2vec algoritmus esetén már láttunk erre a működésre példát, ott θ a beágyazó vektorok mátrixát jelentette. Mivel a node2vec nem veszi figyelembe a csúcsok kategóriáját, csak a gráfstruktúra megtanulására épít, így azt egy felügyelet nélküli tanulásnak tekinthetjük, annyi kitétellel, hogy a séták tanult paraméterezésére létezik megoldás, amivel együtt már semi-supervised algoritmust kapunk.

A veszteségfüggvény egy részletesebb, de még mindig általános képlete több kategóriás klasszifikáció esetén:

$$\mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v)), \tag{1.34}$$

ahol az $y_{u,v}$ 0 és 1 közötti érték, amelyben a csúcsok hasonlósága fejeződik ki, a DEC pedig a skaláris szorzatot jelenti, azaz a vektortérben vett közelséget reprezentálja. A CE a keresztentrópia (cross entropy) rövidítése, ami az alábbi módon számítható egy z_u csúcsbeágyazás esetén:

$$\mathrm{CE}(y, f(z_u)) = -\sum_{i=1}^{C} \left(y_i \log f_\theta(z_u)_i\right), \tag{1.35}$$

ahol y_i és $f_\theta(z_u)_i$ az i-edik kategóriához tartozó valós és prediktált értékek C darab kategóriával számolva. A keresztentrópia gyakori veszteségfüggvény több kategóriára történő felügyelt tanításkor. Bináris klasszifikáció esetén ezt két kategória között kell egyensúlyoznunk a következő módon:

$$\mathcal{L} = -\sum_{u \in V} y_u \log\left(f(z_u)\right) + (1 - y_u) \log\left(1 - f(z_u)\right), \tag{1.36}$$

valós számok predikciója esetén pedig az L2 veszteség számítása a megszokott veszteségfüggvény:

$$\mathcal{L}(y, f(z_u)) = \left\| y - f(z_u) \right\|_2. \tag{1.37}$$

Felügyelt esetben látjuk, hogy a "tökéletes", valódi címkékkel definiált gráfbeli hasonlóságot vetjük össze a leképezett vektortérbeli hasonlósággal. Ennek a leváltására a GraphSage felügyelet nélküli esetében a gráfbeli hasonlóságot a node2vec algoritmusban már megismert módszerekkel definiálja újra. Ugyanúgy felhasználja a gráfban random séták alapján felépített csúcshasonlóságokat és a k negatív mintavételezést, az egy csúcsra kiszámolt veszteség egyenlete így a veszteség:

$$J_G(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - k \cdot \mathbb{E}_{v_n \sim P_V)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right), \tag{1.38}$$

ahol v az u-ból indított random sétán gyakran előforduló csúcs, P_V a negatív mintavételezés eloszlása, k pedig a negatív minták száma. A különbséget a két modell között az input z_u beágyazásban találjuk, amit itt nem a séták alapján határozunk meg, hanem az aggregáló függvényeken keresztül a környező csúcsokból érkező információkból épül. A modell tanítása során az aggregáláshoz és a rétegekhez tartozó paraméterek tanítása egyszerre történik.

1.5.2. Három különböző aggregáló függvény

A GraphSAGE algoritmusban az üzenettovábbításra és az aggregálásra több használt módszer létezik, amik a gyakorlatban egymással kombinálhatók. Ezt a rugalmasságot kihasználva, az adott input gráftól függően különböző kombinációkat alkalmazva növelhetjük a modell hatékonyságát. A legjobb kombinációt sokszor csak kísérleti úton lehet megtalálni, az alábbiakban a leggyakrabban használt verziókat mutatom be. Ezeken keresztül jól megfigyelhető, hogy az aggregálás és üzenetmódosítás mennyire könnyen állítható és újradefiniálható része az ilyen típusú algoritmusoknak, ezzel sok kísérlethez és gyorsan tesztelhető fejlesztéseknek adva táptalajt.

Átlag Az első aggregáló függvény opció a GCN-en keresztül megismert átlag, ahol egy v csúcsra és a szomszédaira számoljuk ki a beágyazások elemenkénti átlagát minden lépésben. Ennek a függvénynek a már megismert sajátossága, hogy elveti a konkatenálást és egybeolvasztja a szomszédokból érkező információt a csúcsban. Ezzel a csúcs saját előző rétegbeli reprezentációját nem tartja meg elkülönítve, így a modell elveszít egy explicit módon megtartott információt. Ezt a fajta információmegtartást más modellekben skip (átugró) kapcsolatként figyelhetjük meg. A GraphSAGE esetében mindkét verzió használható, általánosságban a konkatenálást használó verzió teljesít jobban.

$$\text{AGG} = \mathbf{MEAN}\left(\left\{h_v^{l-1}\right\} \cup \left\{h_u^{l-1},\ \forall u \in \mathcal{N}(v)\right\}\right) \tag{1.39}$$

Maximum pooling Ebben az esetben a megszokott lineáris transzformáció helyett egy egyrétegű Multi Layer Perceptron-t (MLP) alkalmazunk minden vektorra, utána pedig egy elemenkénti maximum pooling (maximum kiválasztás) operátor végzi el az aggregálást. Gyakorlatban a maximum helyére kerülhetne az átlag itt is. Az egyrétegű háló itt a szomszédságon értelmezett tanulást végez, a maximum kiválasztás pedig a legdominánsabb tulajdonságot hivatott reprezentálni és hangsúlyozni a csúcsok szomszédságaiból.

$$\text{AGG} = \mathbf{MAX}\left(\left\{\mathbf{MLP}\left(h_u^{(l-1)}\right),\ \forall u \in \mathcal{N}(v)\right\}\right) \tag{1.40}$$

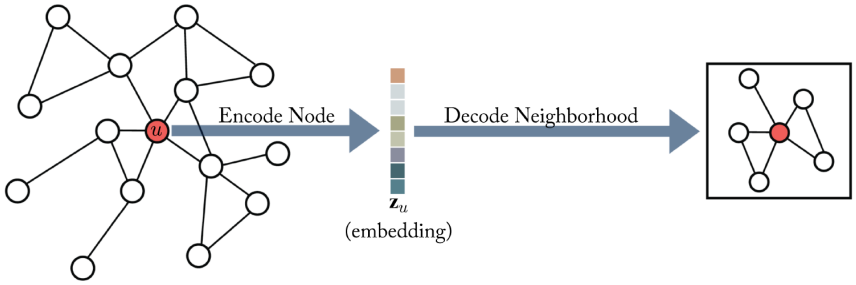
LSTM háló Az LSTM (Long Short-Term Memory)[11] a legkomplexebb aggregátor az eddig megismertek közül, ezért nagyobb eséllyel vesz figyelembe kevésbé domináns gráfbeli jellemzőket is. Miközben az eddigieknek megfelelően aggregálja az input csúcsok vektorait, képes a fontosnak ítélt információkat megtartani és folyamatosan frissíteni a tanulás során. Ezzel a szomszédok között folyamatosan tanult módon tud szabályozni és képes kiszűrni az irreleváns információkat. Az LSTM háló sorozatban érkező adat kezelésére optimalizált modell, így nem permutáció invariáns, ezért a szomszédok sorrendjét folyamatosan változtatni kell. Ennek a megoldására szolgál a π operátor, ami minden iterációban random sorrendet vesz a szomszédságon belül, ezzel elérve azt, hogy a csúcsok között ne legyen szignifikáns különbség a tanítás során.

$$\text{AGG} = \mathbf{LSTM}\left([h_u^{(l-1)},\ \forall u \in \pi(\mathcal{N}(v))]\right) \tag{1.41}$$

1.6. Az encoder-decoder keretrendszer

Az encoder-decoder keretrendszer[12] közös alapot ad a gráf reprezentációt képező modelleknek. Az összes eddig áttekintett modellt le lehet írni vele, ezzel nagyon hasznos eszközt biztosít az algoritmusok vizsgálatához. A megközelítés alapja, hogy a beágyazások létrehozását egy általános encoder (kódoló) és egy decoder (dekódoló) függvényre bontjuk. Az encoder függvény előállítja a beágyazásokat egy d dimenziós vektortérbe történő leképezéssel, ezután a decoder páronként felhasználva ezeket rekonstruálja az eredeti gráfban található információt, legtöbbször valamilyen tanulási célra felhasználva.

$$\text{ENC}\colon V \to \mathbb{R}^d \qquad \text{DEC}\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$$



1.10. ábra. Az encoder-decoder keretrendszer [12]

A páronként történő dekódolás csúcsklasszifikáció esetében a két csúcs közötti hasonlóságot jelenti, de ugyanígy lehet élek vagy akár teljes szomszédságok valószínűségének kiszámítására is használni. A cél az, hogy az S-el jelölt gráfbeli hasonlóságot közelítsük, ezzel a két csúcs hasonlóságát rekonstruálva, azaz:

$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(z_u, z_v) \approx S[u, v]. \tag{1.42}$$

A veszteségfüggvény formája szintén felírható ez alapján. Ezzel a definícióval módosítva, a rekonstrukció veszteségét számoljuk ki a következő formában:

$$\mathcal{L} = \sum_{u,v \in V} \ell\left(\text{DEC}(z_u, z_v), S[u, v]\right). \tag{1.43}$$

Az encoder függvény a node2vec és az összes shallow beágyazást generáló algoritmus esetében egy egyszerű, a beágyazást direktben lekérdező függvény, amely egy $Z \in \mathbb{R}^{d \times |\mathcal{V}|}$, beágyazásokat soronként tároló mátrixból keresi ki a megfelelő csúcshoz tartozót, tehát:

$$\text{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v, \tag{1.44}$$

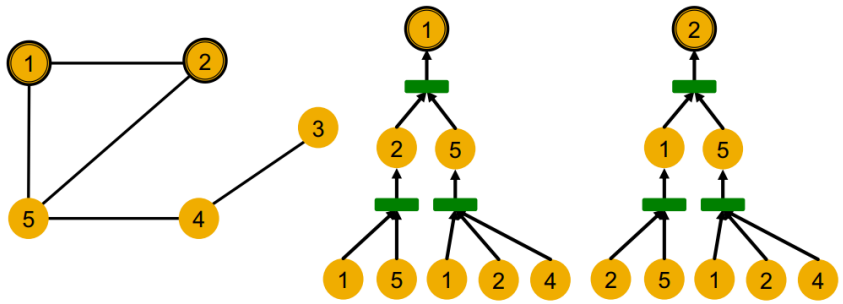
ahol $v \in \mathbb{I}^{|\mathcal{V}|}$ az adott csúcshoz tartozó indikátor vektor, a decoder pedig a szokásos skalárszorzat. A konvolúciós és egyéb, mély beágyazást a szomszédságokból generáló modellek esetében az aggregáló függvények felelnek meg az encoder funkciónak. Az 1.1 táblázatban összehasonlítjuk a két, dolgozatban használt algoritmus felépítését, lebontva a használt decoder, gráfbeli hasonlóság és veszteségfüggvény szerint. Feltűnő módon a két modell leírása megegyezik, mivel az alapjaikat képező koncepciók hasonlóak. Ez jól kihangsúlyozza keretrendszer általánosító erejét, különösen, ha figyelembe vesszük, hogy sok egyéb, a dolgozatban nem szereplő módszerre is ráhúzható megközelítésről beszélünk.[13][14]

Módszer	Decoder	Gráfbeli hasonlóság	Veszteségfüggvény
node2vec	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in V} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$P(v\|u)$	$-\mathbf{S}[u,v] \log(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v))$
GraphSAGE	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in V} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$P(v\|u)$	$-\mathbf{S}[u,v] \log(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v))$

1.1. táblázat. A dolgozatban részletezett algoritmusok az encoder-decoder keretrendszerben

1.7. Modellek kifejezőképessége

Most, hogy láttuk több modellt működésének alapjait, tudunk beszélni egy általános koncepcióról, ez pedig a gráf neurális háló információs kifejezőképessége egy adott gráf viszonylatában. Ezt pontosabban azon keresztül tudjuk definiálni, hogy mit képes és mit nem képes megtanulni a gráfban található információkból egy adott modell. Egy konkrét, modelleknek problémát okozható eset lehet például egy azonos csúcs és él szintű információkat tartalmazó csúcspár, melyeknek azonos fokszámúak a szomszédai is. Ezek teljesen azonosak lesznek ha csak a hozzájuk tartozó számítási gráfokat vesszük alapul (1.11 ábra).



1.11. ábra. Azonos számítási gráffal rendelkező csúcsok [15]

Mivel a csúcsazonosítók (esetünkben az egyes és kettes csúcsok címkéje) a modellünk számára nem elérhető információk, ezért ezek a csúcsok a beágyazási térben ugyanoda kerülnek, így a modell számára megkülönböztethetetlenné válnak. Ebből következik, hogy a ha minden csúcsbeágyazáshoz egy egyedi gyökeres fa társítható, azaz a hozzárendelő függvény injektív, akkor a modell számára is érthető módon, teljesen egyedien tudjuk lefedni a csúcsokat. Ezzel tehát elérjük, hogy a gráfban lévő összes információt meg tudja tanulni a modell. Ha szintenként nézünk egy számítási gráfot, akkor az aggregáló függvényre is át kell vinnünk ezt a tulajdonságot, tehát minden szinten egy injektív aggregáló függvényre van szükségünk.

Konkrétabb példán megvizsgálva ezt a gondolatmenetet, egy GCN esetében, amikor számtani középpel aggregálunk, akkor azonos arányban ugyanolyan információt továbbadó csúcsok esetén találunk is példát információvesztésre. Ebben az esetben, ha egy csúcsba 1-1, 2-2 ... azonos információ érkezik be, akkor a számtani közepük azonos lesz, tehát ezekben az esetekben az aggregált üzenet ugyanaz lesz a háló számára, hiába alkalmazunk később bármilyen transzformációt. Maximummal való aggregálás esetén is létezik az elveszett információ jelensége. Ebben az esetben egy adott csúcsba beérkező szomszédok halmazából eltűnik az az információ, hogy a nem maximális értékekből pontosan hány darab érkezett be, illetve azonos maximumok esetében szintén azonossá válnak a

csúcsok a modell számára. Ez alapján kijelenthetjük, hogy sem az alap GCN, sem a GraphSage algoritmus nem érték el a teoretikus maximális kifejezőképességet az aggregátor függvényeik miatt.

Injektív aggregálást az összes elemre egyesével alkalmazott nem lineáris f függvénnyel, majd az így kapott információk szummázásával és egy újabb nem lineáris Φ függvény alkalmazásával tudunk elérni. Ezzel kihasználjuk, hogy mivel az aggregálás előtt egyedi értéket tudunk gyártani minden elemhez, ezért az aggregált érték is egyedi lesz.

$$\Phi\left(\sum_{x\in S} f(x)\right) \tag{1.45}$$

Ezt továbbgondolva, ha felhasználjuk az univerzális approximációs tételt[16], amely kimondja, hogy bármilyen folytonos, korlátos tartományon értelmezett függvényt tetszőlegesen jól tudunk közelíteni egy megfelelően nagy méretű neurális hálóval, akkor egy-egy MLP-nal végzett közelítésre cserélhetjük mindkét használt függvényt.

$$\mathrm{MLP}_{\Phi}\left(\sum_{x\in S} \mathrm{MLP}_{f}(x)\right) \tag{1.46}$$

Ezzel eljutottunk a maximális kifejezőképességű GNN-hez, ez pedig nem más mint a GIN[17] (Graph Isomorphism Network). Jogosan felmerülő kérdés, hogy számítási kapacitásban ez mennyivel növeli meg a modellt. A gyakorlat azt mutatja, hogy 100-500 rejtett paraméter elégséges a függvények megfelelő működéséhez. A legtöbb modellnek a GIN maximális kifejezőképességének ellenére is megmarad a létjogosultsága. Mivel a hálózatok nagyon diverz módon épülnek fel, bizonyos esetekben bizonyos modellek alkalmasabbak egy-egy adott feladatra. Például ha kevés, címkézett adat áll rendelkezésre és csúcsklasszifikálás a célunk akkor a GCN jó választás, ha egy dinamikus gráfban akarunk a beérkező csúcsokra új beágyazást generálni akkor a GraphSage működhet jól, ha pedig gyorsan számítható strukturális beágyazást kell generálnunk akkor a node2cev adhat megoldást.

2. fejezet

Gráfok a gyakorlatban

2.1. A kísérletek felállítása

Az első kísérletben a gráfokon végzett tanulás hagyományos gépi tanulási módszerekre támaszkodó lehetőségeit nézzük meg a dolgozatban részletezett modellek összevetésével. Ehhez előtte végigvesszük azokat a gyakran használt gráftulajdonságokat, amikre szükség lehet egy gráf elemzésekor. A második kísérletben az algoritmusok időbeli skálázódását vizsgáljuk meg. Ehhez a feladathoz szükséges a saját gráfok generálása, ami egy messzire mutató és a témában releváns feladat, ezért ebből is megvizsgáljuk a bevett módszereket működését.

2.2. Coding setup

A kódolást a Python 3.10-es verziójával és a Pytorch Geometric[3][18] 2.6.1 csomagjával végzetem. A modellek tanítása során a CUDA (Compute Unified Device Architecture) platformot használtam, amely lehetővé teszi, hogy a modellek tréningelése során a számításokat a grafikus kártyán végezzük el a processzor helyett, ezzel szignifikánsan felgyorsítva például a mátrix műveleteket. A kódbázis publikusan elérhető az itt belinkelt GitHub felületen:
`https://github.com/csabajozsef/thesis_coding`

2.3. Gráfok jellemzése

Egy gráf adathalmaz első megvizsgálásakor sok olyan számszerűsíthető tulajdonság van, aminek kiszámításával és értelmezésével információt nyerhetünk a hálózat struktúrájáról. Ilyenek – a teljesség igénye nélkül – a csúcsok és élek száma, az átlagos fokszám, a legnagyobb összefüggő részgráf mérete, a klaszterezési együttható, a csúcscentralitás és a fokszámeloszlás. Ezek közül az első négy értelemszerű, a többi pontosabb kifejtést igényel.

A klaszterezési együttható értelmezhető gráf és csúcs szinten is. Egy adott v_i csúcs esetén a képlete:

$$C_i = \frac{2e_i}{k_i(k_i - 1)} \quad C_i \in [0, 1], \tag{2.1}$$

ahol k_i a csúcs fokszáma, e_i pedig a csúcsok közötti élek száma. A képletben a szomszédságban létező csúcsok számát osztjuk el a lehetséges összes él számával. Ez abba enged betekintést, hogy egy csúcs szomszédai mennyire vannak összekapcsolva egymással, tehát az érték ezzel a nevében

is említett klaszterezettséget számszerűsíti lokális szinten. Ha ennek az értéke 1, akkor a csúcshoz tartozó összes szomszéd ismeri egymást is. Gráfszinten az összes csúcsra átlagolva értelmezhető, azaz:

$$\bar{C} = \frac{1}{n} \sum_{i=1}^{n} C_i. \tag{2.2}$$

A klaszterezési együttható szorosan kapcsolódik a gráfokban a csúcs által alkotott, élek által lezárt háromszögek számához, hiszen egy csúcs szomszédai között akkor létezik az összes háromszög, ha az összes szomszéd ismeri egymást. A háromszögeket továbbgondolva, előre definiált, komplexebbek graphletek számolásával szintén hasznos információt nyerhetünk a gráf felépítéséről. A graphlet-ek egy gyökeres, összefüggő, nem izomorf részgráfot takarnak, amelyek a méretüktől függően véges számú kombinációkban léteznek. Ezekből a relevánsakat kiválasztva, minden csúcshoz képezhetünk egy GD (graphlet degree) vektort, amelyben az adott csúcshoz tartozó graphlet-ek számát tároljuk, ezzel a teljes lokális topológiáját számszerűsítve.

A következő jellemző a csúcscentralitás, ami egy összefoglaló fogalom, így több verziója is használatban van. Ezeken megyünk most végig, egy v csúcs esetében c_v-vel jelölve a centralitás értékeket típustól függetlenül. A legegyszerűbb, triviális centralitás érték a csúcs fokszáma, azaz $c_v = deg(v)$. Ennek a gráfra kiterjesztett verziója:

$$c_G = \frac{\sum_{i=1}^{|V|} [c(v^*) - c(v_i)]}{H}, \tag{2.3}$$

ahol H értéke a következő módon van definiálva:

$$H = \sum_{j=1}^{|Y|} [c(v^*) - c(v_j)]\,. \tag{2.4}$$

Y egy maximálisan centralizált csúcshalmaz, $|V| = |Y|$ mérettel, v^* pedig az a csúcs, aminek a fokszáma legnagyobb az adott gráfban. A fenti módon kiszámított nagy c_G érték esetén néhány csúcs dominálja a kapcsolatokat, kis érték esetén pedig a csúcsok fokszáma közel egyenletesen oszlik meg.

A következő fogalom a sajátvektoros centralitás, amiben eggyel komplexebb megközelítéssel az adott csúcs fontosságát a környező csúcsokból származtatjuk. A képlete:

$$c_v = \frac{1}{\lambda} \sum_{u \in \mathcal{N}(v)} c_u, \tag{2.5}$$

ahol λ egy pozitív konstans. Ezt átrendezhetjük a szomszédsági mátrix sajátvektoros formájára, a következő módon:

$$c_v = \frac{1}{\lambda} \sum_{u=1}^{n} A_{vu} \cdot c_u = \lambda \mathbf{c} = A\mathbf{c} \tag{2.6}$$

$$A_{uv} = \begin{cases} 1 & \text{ha } u \in N(v) \\ 0 & \text{egyébként.} \end{cases} \tag{2.7}$$

A $\mathbf{c}$ vektor a centralitás vektor, amiben az iterálva történő információfrissítés során konvergálnak az adott csúcsokhoz tartozó értékek a centralitásuk felé. Az iterációk során folyamatosan normalizáljuk a vektort, ezzel elkerülve a szélsőséges értékek kialakulását, végül a legnagyobb sajátértékhez tartozó vektor az, amit használunk a centralitások meghatározásánál.

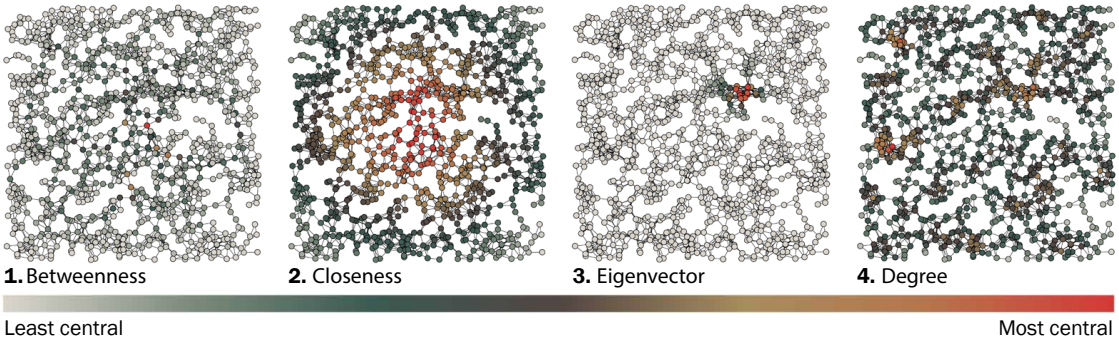
A következő centralitás érték az átmeneti (betweenness) centralitás, ami megmutatja, hogy egy csúcs milyen gyakran fordul elő más csúcsok között vett legrövidebb utakon, azaz mennyire fontos kapcsolatot jelent a hálózaton belül. Ennek a számításához egy v csúcs esetén minden (s, t) csúcspárra vesszük a legrövidebb utakat közöttük, jelölje ezeket σ_{st} . Ezután megkeressük közülük az összes olyat, amin v csúcs előfordul, ezt jelölje $\sigma_{st}(v)$, ezek aránya pedig megadja a keresett centralitást:

$$c_v = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{2.8}$$

A legrövidebb utak koncepciójából adódik a közelségi (closeness) centralitás is, ami egy adott v csúcsból összes többi csúcsba vett legrövidebb utak hosszát átlagolja, ezzel abba engedve betekintést, hogy mennyire központi helyen helyezkedik el egy csúcs.

$$c_v = \sum_{v \neq t \in V} \frac{1}{\sigma_{vt}} \tag{2.9}$$

Az alábbi 2.3 ábrán láthatjuk a különböző csúcsokhoz tartozó centralitási értékeket egy véletlen gráfon.



2.1. ábra. Centralitások vizualizációja egy random gráfon [19]

Az eddigieken felül, az egész gráfra vett fokszámeloszlás szintén egy olyan gráfjellemző, amit fontos vizsgálnunk egy gráf elemzésekor. Ez megadja, hogy az adott gráfban random módon kiválasztott csúcsnak milyen eséllyel van k fokszáma, azaz $P(k) = N_k/|V|$, ahol a N_k a k fokszámú csúcsok száma. A gráf méretével való normalizálás elhagyható, ez esetben közvetlenül, darabszámmal jellemezzük ezt a tulajdonságot. Egy való életből vett gráf fokszámeloszlása jellegzetes mintát mutat, amelyik gráf ettől eltér azt érdemes tovább vizsgálni és kideríteni a különbség hátterében húzódó okokat.

2.4. Kísérletek

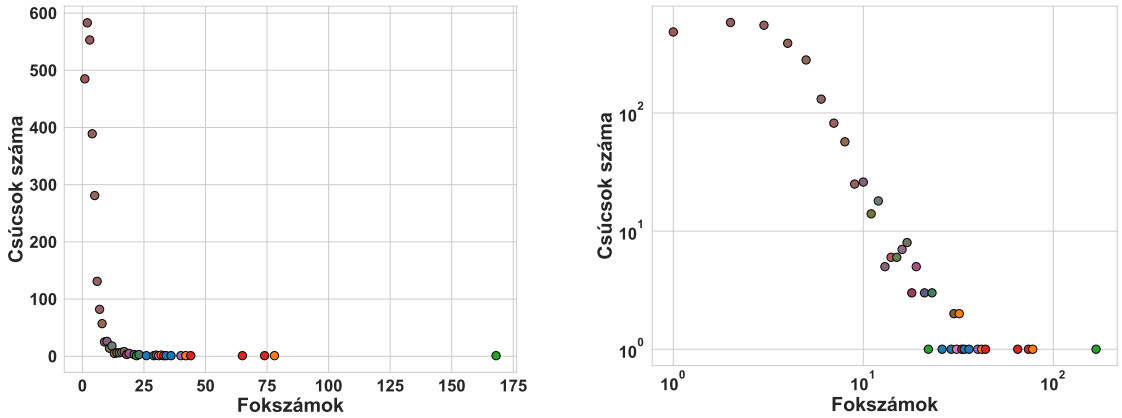
2.4.1. Egy konkrét gráf feldolgozása - esettanulmány

Ebben szekcióban a dolgozat struktúráját követve végignézzük azokat a lépéseket, amivel egy konkrét gráf gépi tanulással történő feldolgozását megtehetjük. Bemutatásra kerül a gyakorlatban használható gráftulajdonságok felhasználása és megvizsgáljuk azt is, hogy a tradicionális gépi tanulási módszerek és a gráf reprezentációs modellek hogyan teljesítenek egymáshoz képest egy konkrét csúcsklasszifikációs feladat esetén. Az ennek alapjául szolgáló gráf a Cora adathalmaz [20], ami egy

nagyon gyakran használt példa különböző gráfokon tanuló modellek tanítására és tesztelésére.

A gráf reprezentációkat készítő modellek definiálása az algoritmusokat leíró cikkekben talált alapértelmezett paraméterek alapján történt, azaz nem végeztem el az amúgy megszokott paraméter optimalizálást. Ennek oka, hogy mivel nem egy konkrét adathalmazon kellett elérni legjobb eredményt, hanem a modellek egymás közötti viszonyát vizsgáljuk, ezért egységes feltételeket kellett biztosítani. Így a klasszifikációs algoritmusok alkalmazása során sem használtam semmilyen optimalizációt, mindenhol az alapmodellek pontosságának konvergálásig történt a tanítás.

Visszatérve a Cora adathalmazhoz, az első vizsgálatkor láthatjuk, hogy ez egy 2708 publikációból álló hivatkozási hálózatot tartalmaz, amelyben összesen 5429 kapcsolatot találunk. Az átlagos fokszám 7.7, a fokszámeloszlást a következő 2.2 ábra mutatja.



(a) Fokszámeloszlás lineáris skálán

(b) Fokszámeloszlás logaritmikus skálán

2.2. ábra. A Cora adathalmaz fokszámeloszlása

A fokszámok hatványfüggvény szerinti eloszlást mutatnak, ami a skálafüggetlen, való életbeli hálózatok jellemzője. Az egyes pontok színezése az adott fokszámú csúcsok kategóriánként vett színeinek az átlagából tevődik össze, ezzel egy gyors vizuális jelzést adva, ha egy adott fokszámnál egy szín, azaz egy kategória dominál. Ebben a gráfban az elvártak szerint minél több csúcs van az adott fokszámmal, annál diverzebb csúcshalmazt képeznek kategóriák szempontjából, tehát egyre homogénebb átlagolt színt kapunk.

A Cora adathalmazban a csúcsok hét kategóriába tartozhatnak a cikkek témája alapján, ilyenek például a 'Genetic Algorithms' és a 'Neural Networks' kategória. Az adathalmazhoz tartozik egy táblázat, amiben 1433 egyedi szó található, melyek előfordulása bináris formában van megadva minden egyes publikációhoz. Ha ezen a gráfon akarunk csúcsklasszifikációt végezni, akkor kiindulásnak használhatjuk ezt a megadott szótárat. Bár ez nem minden gráf esetében adott, mivel itt szövegeket tartalmazó csúcspontokról beszélünk, ezért ez a táblázat egy könnyen előállítható mesterséges adathalmaz-bővítés. Az ilyen típusú adatoknál ez értelemszerűen nagyon sok információt hordoz az egyes cikkek kategóriájáról, ezért komoly predikciós erővel bír már kezdéskor. Ezzel egy klasszikus, tabuláris adaton végzett gépi tanulási feladatot állítottunk elő a gráfból, ez lesz az alapmodellünk. Ennek a teljesítményét vizsgálva láthatjuk, hogy be is igazolódott a hipotézisünk és az így definiált klasszifikációs feladatban máris jól teljesítő alapmodell kaptunk, lásd a 2.1. táblázatot.

Ha tovább vizsgáljuk a problémát akkor észrevehetjük, hogy a gráfban lévő strukturális adatok felhasználása eddig nem történt meg. A fentebb bemutatott gráfbeli metrikákat felhasználva a csúcsokhoz egyesével hozzá tudjuk rendelni a fokszámukat, a klaszterezettségi együtthatójukat, illetve a különböző centralitási értékeiket. Ezekkel együtt futtatva a klasszifikációt a modell teljesítménye romlott, bármilyen kombinációban véve. A 2.1 táblázatban a fokszámmal, klaszterezettségi együtthatóval és sajátvektoros centralitási értékkel augmentált adathalmazon végzett osztályozás eredménye látható a második oszlopban. Ez az eredmény vagy a modell túltanulásának következménye lehet, vagy pedig annak, hogy a hozzáadott új oszlopok zavaró információt tartalmaztak. Emiatt logikus következő lépés, ha egyszerűsítünk a modell felépítésén és megvizsgáljuk, hogy csak a mesterségesen legyártott strukturális metrikák alapján hogyan tanul a modell, mennyi információt kapunk ezeken keresztül egy csúcs kategóriájáról. Az ehhez felhasznált tulajdonságok a csúcsok fokszáma, klaszterezési együtthatók illetve a három fentebb kifejtett centralitási értékek voltak. A harmadik oszlopban láthatjuk, hogy beigazolódott a feltevés, miszerint ezek az adatok nem rendelkeznek elég információval a klasszifikáció elvégzéséhez.

A strukturális információkon továbblépve és tovább építve a dolgozatban felvázolt logikára, nézzük meg, hogy egy csúcsra milyen pontos predikciót kapunk a szomszédságában lévő aggregált információ alapján. Ezzel bekerül a csúcsattribútumok aggregált, átlagolt információja is, ami egy erősen klaszterezett gráf esetében komoly predikciós erővel bírhat. Ennek megfelelően a felhasznált szótár és szomszédos csúcsok adatkombinációval az eddigi legjobb eredmény kaptuk, szignifikánsan javítva az alapmodellünk teljesítményén, ez látható a negyedik oszlopban.

Ez eddigieken keresztül láthatjuk, hogy euklideszi irányból megközelítve a feladatot körülményes előállítanunk a szükséges adathalmazt, sok iterációra és feature engineering lépésre van szükség mire használható eredményt kapunk. Ez sok gráf esetében a komplexitás miatt nagyon költséges feladat. Itt jönnek be a fentebb megismert, gráf reprezentációs algoritmusok.

A node2vec algoritmus beágyazásai a gráf struktúráján tanulnak, emiatt elméletben hasonló eredményt várhatnánk tőle, mint a strukturális metrikákon tanult modell esetén. Láthatjuk, hogy bár a modell nem használja fel a csúcsok attribútumait tanuláskor, pusztán a szerkezeti adatokból sokkal jobb eredményt képes elérni, azaz sokkal jobban számszerűsíti a csúcsok ilyen típusú jellemzőit mint a korábban felhasznált triviális strukturális metrikák, sőt bizonyos klasszifikációs algoritmusok esetében felülmúlja az eddigi legjobb modellünket. Mivel a node2vec algoritmus a p és q paraméterekre épül, ezért érdemes minden gráf esetében egy gyors teszttel ellenőrizni, hogy melyik paraméter dominanciájára reagál jobban a tanítás során a modell. Ez gráfonként változó lehet, a modellben az alap érték az 1 mindkét paraméterre. A Cora adathalmaz esetében, mivel nem nagy gráfról van szó, ezért a tanulás során az alapértelmezett 10 db, 80 hosszú séta minden csúcsból indítva elégségesen lefedi a gráfot, ezért a két verzió között itt nincs mérhető különbség.

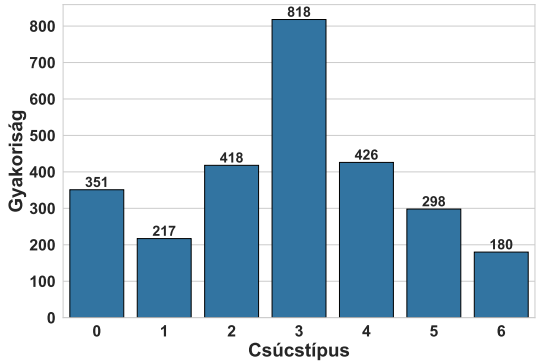
A GraphSAGE algoritmus esetében kapjuk a legjobb eredményeket. Ez nem meglepő, mivel ez a modell az eddigi két leghatásosabb rendelkezésre álló módszert használja fel. Az első a node2vec-ben használt random sétákra alapuló veszteségfüggvény, amin keresztül itt is ugyanazt a strukturális információt részesítjük előnyben a tanítás során, a második pedig a csúcsok szomszédságából aggregált formában érkező szótár adatok.

Az átlag és maximum aggregátor közül az átlag teljesített jobban, az LSTM verziót a memóriaigénye miatt nem lehet asztali gépen a jelenlegi implementációját felhasználva tesztelni. A felügyelt és felügyeletlen verziók között szintén nem volt észlelhető különbség a teljesítményben, az 2.1 táblázatban az átlagot használó, felügyelet nélküli modell eredményei szerepelnek.

	Alapmodell	**Szótár és strukturális jellemzők**	**Strukturális jellemzők**	**Szótár és szomszédok kategóriái**	**node2vec**	**GraphSAGE**
Logistic regression	0.7657	0.7638	0.3229	0.8100	0.7970	0.8782
SVM	0.7528	0.6845	0.3026	0.8339	0.8265	0.8708
Decision Tree	0.6458	0.6347	0.4686	0.7546	0.6365	0.7656
Random Forest	0.7657	0.7638	0.5185	0.8321	0.8394	0.8782
Gradient Boosting	0.7362	0.7399	0.5258	0.8247	0.8321	0.8579
MLP	0.7638	0.7675	0.3985	0.8118	0.8118	0.8487

2.1. táblázat. A Cora adathalmazon elért, kategóriákon átlagolt pontosságok

A táblázatos adathalmazokon és a beágyazásokon is ugyanazokkal a klasszifikációs modellekkel történt a kategorizálás az összehasonlíthatóság megtartása miatt. A használt klasszifikációs algoritmusok részletes leírása nem része a dolgozatnak. A tanítás során a gyakran használt 80-20-as arány adtam meg a tanító és tesztelő adathalmaz arányának. A tanítások során feltűnő módon kiemelkedő pontosságot kaptunk a hármas kategóriába tartozó csúcsokra. Ennek oka az ebbe a kategóriába tartozó csúcsok kimagasló száma, ami miatt a tanulás során erre a típusra több adat volt felhasználható (2.3).



2.3. ábra. Csúcstípusok gyakorisága a Cora adathalmazban

Következtetésnek levonhatjuk tehát, hogy a gráf reprezentációs algoritmusok által generált beágyazások felülmúlják a feladathoz elkészített hagyományos adathalmazt csúcsklasszifikáció esetén. Érdemes ismét megemlíteni a paraméteroptimalizálás hiányát, illetve, hogy például egy node2vec

segítségével gyártott beágyazást lehetséges hozzácsatolni az egyes csúcsokhoz mint tulajdonság, ezzel egy következő algoritmusnak inputként biztosítva a már megtanult információkat, tehát ezzel az elemzéssel még nem értünk a gráfon elérhető maximum pontosság közelébe.

2.4.2. Az Erdős-Rényi-modell

Ha egy modellt megfelelően akarunk tesztelni akkor a valódi életből nyert gráfokon nehéz lemérni a gráftól független teljesítményüket és hatékonyságukat. Ha például a futási idejük skálázódását akarjuk vizsgálni, akkor rögtön felmerül az igény az ehhez megfelelő, egymáshoz hasonló módon felépülő, de méretben növekvő gráfok előállítására. Ebben a szekcióban megvizsgálunk egy gráfelőállítási stratégiát, amelyben kontrollált módon, előre meghatározott tulajdonságokkal tudjuk elkészíteni a teszteléshez a gráfokat.

Ez a módszer nem más mint az Erdős–Rényi-modell, az egyik leggyakrabban használt gráfgenerátor ami kisvilág-tulajdonságú gráfot képez. Ezekre a gráfokra jellemző, hogy az átlagos úthossz kicsi bennük, míg a klaszterezettségi együttható alacsony. Az Erdős-Rényi gráfnak két verziója létezik. Az első jele $G_{n,p}$, ahol az n a csúcsok számát, p pedig az élek létrejöttének az esélyét jelölik egymástól függetlenül. A másodikat $G_{n,m}$ jelöléssel írjuk fel, ahol az összes lehetséges n csúcsú, m élő gráfból választunk egyet, egyenletes eloszlás szerint. Mindkét esetben véletlen gráfról beszélünk, tehát egy sztochasztikus folyamat segítségével kerülnek legyártásra, ami miatt ezek a paraméterek nem egy egyértelmű gráfot határoznak meg, viszont a megadott paraméterek alapján keletkeznek. Gyakorlatban a $G_{n,p}$ modell preferált a feltételezett függetlenség miatt. A fokszámeloszlás egy $G_{n,p}$ gráf esetén binomiális, ez a következő képlettel írható le:

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}, \tag{2.10}$$

mivel p eséllyel kell k db élet kiválasztanunk egy csúcs esetében az $n-1$ szomszédhoz. Emiatt az eloszlás miatt az átlagos fokszám $p(n-1)$ lesz, ezt jelölje $\overline{k}$. Egy i csúcshoz tartozó e_i élek várható értéke a következő képlettel írható le k_i darab szomszéd esetén:

$$\mathbb{E}[e_i] = p \cdot \frac{k_i(k_i-1)}{2}, \tag{2.11}$$

hiszen ennek a kiszámításához csak a gráfhoz tartozó valószínűséget kell megszoroznunk az összes lehetséges él számával. Ezt behelyettesítve a klaszterezési együttható képletébe megkapjuk a keresett értéket erre is:

$$E[C_i] = \frac{p \cdot k_i(k_i-1)}{k_i(k_i-1)} = p = \frac{\overline{k}}{n-1} \approx \frac{\overline{k}}{n}. \tag{2.12}$$

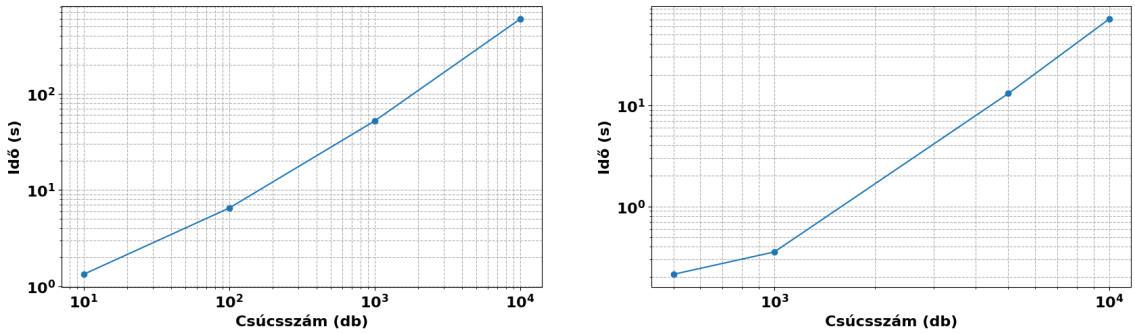
Láthatjuk, hogy az átlagos klaszterezési együttható a gráf növekedésével csökken, így látjuk a már említett alacsony klaszterezettség okát. A másik gráfjellemzőt a legrövidebb utak átlagos mérete biztosítja, melynek jele l, a képlete[21] pedig:

$$l = \frac{\ln(n) - \gamma}{\ln(\overline{k})} + \frac{1}{2}, \tag{2.13}$$

ahol γ az Euler-állandó. A képletben a logaritmus biztosítja az érték alacsonyan maradását a gráf méretének gyors növekedésekor is.

2.4.3. Az algoritmusok futási ideje

A tesztelésre alkalmas gráfok előállítása után megnézhetjük, hogy hogyan teljesítenek a dolgozatban megismert algoritmusok különböző méretű gráfokon, ezt az alábbi 2.4 ábra szemlélteti. A futási időket 5 futtatás átlagából vettük.



(a) node2vec

(b) GraphSAGE

2.4. ábra. Az algoritmusok futási ideje gráfméret függvényében (log-log)

A node2vec algoritmus eredeti publikációjában[7] elért eredményekkel konzisztensen, a futási idők $O(|V|)$ nagyságrendben, lineárisan változnak az elvégzett kísérletben. A GraphSAGE esetében az elvárt futási idő $O(|V| \cdot S^K)$, ahol S és K konstans, tehát $|V|$ függvényében itt is közel lineáris eredményt kell kapjunk, aminek meg is felelt a kapott eredmény. A 10^3 méretű gráfok esetében megfigyelhető teljesítményváltozás hátterében állhat az algoritmus kódbeli implementációja, vagy pedig a gráfok generáláshoz használt modell. Ennek a megoldását nagyobb gráfokhoz elégséges számítási kapacitással vagy további kísérletekkel lehetne megkapni. Ezzel sikeresen leteszteltük a használt algoritmusok futási idejét, ahol az eredmények az elvártaknak megfelelően alakultak.

3. fejezet

Összegzés

A dolgozatban megismertük a gráfbeágyazásokat készítő módszerek motivációját, típusait és implementációs nehézségeit. Felvázoltuk és gyakorlatban megmutattuk, hogy miért nem triviális feladat a tradicionális gépi tanulási módszerek gráfokra való átültetése. A node2vec algoritmusban a véletlen séta alapú beágyazásokat, a GraphSAGE modellben pedig a csúcsok információinak aggregálását részletesen kifejtve feltártuk az ezen modellek fontosságát adó működésbeli újításaikat. A gráf neurális hálókról szóló bemutatással teljesebb képet kaptunk a témában jelenleg rendelkezésre álló eszköztárról. Megnéztük azt is, hogy hogyan lehet rendszerszinten gondolkodva vizsgálni az algoritmusokat, felhasználva ehhez az encoder-decoder keretrendszert és a modellek kifejezőképességének definícióját, ezekkel a további gondolkodásnak teret adva. A gyakorlati részben egy konkrét gráfon keresztül láthattuk, hogyan néz ki egy csúcsklasszifikációs feladat megoldása, illetve bemutattuk, hogy a dolgozatban megismert módszerek miért játszanak ebben megkerülhetetlen szerepet. Ezen felül érintettük a gráfgenerálás és az algoritmustesztelés témakörét is.

A megvizsgált modellek és kérdések a gráfokon történő gépi tanulás témakörének csak kis töredékét fedik le, sok nagyon érdekes, tovább vizsgálható kérdés maradt nyitva, amelyekkel a jövőben szeretnék foglalkozni. Az egyik ilyen a dolgozatban nem használt paraméteroptimalizálás témája. A gráfokat leíró metrikák, illetve egy adott modell paraméterei és teljesítménye közötti összefüggéssel az egyes algoritmusok tanulási erősségeit és gyenge pontjait lehetne feltérképezni. Ezzel az információval egy adott modellhez elméletben el lehetne készíteni a számára legjobb és legrosszabb adathalmazokat. Az egyik jelenleg is releváns irány a témában a gráfok csúcspontjában idősorokat tartalmazó, szenzor alapú adathalmazok, amelyek szintén érdekelnek tekintve, hogy szenzorokból érkező adatokkal gyakran foglalkozom. Ezek vizsgálata egybefonódik az idősorokat vizsgáló matematikai eszközökkel, ezzel izgalmas új területet nyitva a kutatásban. Ha a dolgozatban részletezett modelleket nézzük, akkor felmerült kérdésnek a GraphSAGE esetén az LSTM aggregálás hatékony implementációjának kérdésre, valamint a node2vec által legyártott beágyazások felhasználása más algoritmusok tanítására, amelyekre szintén szeretnék kísérletet tenni. A modellek hatékonyabb programozási módszerekkel való implementálására már találtam példákat, ezeket tervezem összevetni az általam használt verziókkal. A teljes gráfokon végzett klasszifikáció és alkalmazásai, illetve ennek a csúcsbeágyazásokkal való kapcsolata természetes módon épít erre a dolgozatra. Az utolsó kézenfekvő jövőbeli haladási irány az újabb, elmúlt pár évben publikált modellek vizsgálata.

Irodalomjegyzék

[1] S. Brohée and J. van Helden, „Evaluation of clustering algorithms for protein-protein interaction networks," *BMC Bioinformatics*, vol. 7, p. 488, 2006.

[2] M. Loukil, L. Sfaxi, and R. Robbana, „How to create graphs in hardware-constrained environments? Choosing the best creation approach via machine learning-based predictive models," *International Journal of Data Science and Analytics*, vol. 19, pp. 283–302, 2025.

[3] M. Fey and J. E. Lenssen, „Fast graph representation learning with pytorch geometric," 2019. [Online]. Available: https://arxiv.org/abs/1903.02428

[4] V. T. Hoang, H.-J. Jeon, E.-S. You, Y. Yoon, S. Jung, and O.-J. Lee, „Graph representation learning and its applications: A survey," *Sensors*, vol. 23, no. 8, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/8/4168

[5] M. Xu, „Understanding graph embedding methods and their applications," *SIAM Review*, vol. 63, no. 4, pp. 825–853, 2021. [Online]. Available: https://doi.org/10.1137/20M1386062

[6] H.-C. Yi, Z.-H. You, D.-S. Huang, and C. K. Kwoh, „Graph representation learning in bioinformatics: trends, methods and applications," *Briefings in Bioinformatics*, vol. 23, no. 1, p. bbab340, 09 2021. [Online]. Available: https://doi.org/10.1093/bib/bbab340

[7] A. Grover and J. Leskovec, „node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 855–864. [Online]. Available: https://doi.org/10.1145/2939672.2939754

[8] T. Mikolov, K. Chen, G. Corrado, and J. Dean, „Efficient estimation of word representations in vector space," 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[9] W. Hamilton, Z. Ying, and J. Leskovec, „Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

[10] J. Leskovec, „Cs224w: Machine learning with graphs," https://web.stanford.edu/class/cs224w/slides/08-graph-transformer1.pdf, 2024, lecture 8 slides, Stanford University.

[11] S. Hochreiter and J. Schmidhuber, „Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[12] W. L. Hamilton, *Graph Representation Learning*. Springer Cham, 2022.

[13] W. L. Hamilton, R. Ying, and J. Leskovec, „Representation learning on graphs: Methods and applications," 2018. [Online]. Available: https://arxiv.org/abs/1709.05584

[14] Y. Zhu, F. Lyu, C. Hu, X. Chen, and X. Liu, „Encoder-decoder architecture for supervised dynamic graph learning: A survey," 2022. [Online]. Available: https://arxiv.org/abs/2203.10480

[15] J. Leskovec, „Cs224w: Machine learning with graphs," https://web.stanford.edu/class/cs224w/slides/06-theory.pdf, 2024, lecture 6 slides, Stanford University.

[16] K. Hornik, M. Stinchcombe, and H. White, „Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208

[17] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, „How powerful are graph neural networks?" in *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019, accessed: Jun. 2025. [Online]. Available: https://arxiv.org/abs/1810.00826

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library.* Red Hook, NY, USA: Curran Associates Inc., 2019.

[19] Phlome, „Six centrality measures applied to the same random geometric graph." https://commons.wikimedia.org/wiki/File:Wp-01.png, 2022, wikimedia Commons, licensed under CC BY-SA 4.0.

[20] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, „Automating the construction of internet portals with machine learning," *Inf. Retr.*, vol. 3, no. 2, p. 127–163, Jul. 2000. [Online]. Available: https://doi.org/10.1023/A:1009953814988

[21] A. Fronczak, P. Fronczak, and J. A. Hołyst, „Average path length in random networks," *Physical Review E*, vol. 70, no. 5, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.70.056110

Appendix

1. táblázat. A Cora adathalmazon használt modellparaméterek

Modell	**Paraméter**	**Érték**
GraphSAGE	Tanulási ráta	0.005
	Rejtett dimenziók száma	64
	Rétegek száma	2
	Aggregációs függvény	mean
	Batch méret	128
	Epochok száma	100
	Dropout	0.5
	Kimeneti dimenziók száma	32
Node2Vec	Beágyazási dimenzió	128
	Séták hossza	80
	Kontextus mérete	10
	Séták száma/csúcs	10
	Negatív minták száma	1
	p paraméter	1
	q paraméter	1

A seed-ek minden könyvtárban fixen a 42 értéket kapták. A modellek tanítása egy Nvidia GeForce RTX 3060 kártyán történt.