Branchelés az egészértékű programozásban gépi tanulással

Karl János

1. Bevezetés

Számos kombinatorikus optimalizálási probléma felírható egészértékű programozási feladatként. Az ilyen feladatok kezelésére az első számú megoldási módszer a branch and bound, amely a megoldások terét rekurzívan diszjunkt részekre osztja. Az algoritmus hatékonysága nagyban függ a kiválasztott változótól, amely alapján létrehozzuk a két részfeladatot. A gyakorlatban több heurisztikus algoritmus ismert, például a strong branching, amely az LP optimum legnagyobb változását eredményező tört változót választja. A projekt célja az volt, hogy megpróbáljuk imitálni a heurisztikus stratégiákat gépi tanulással, majd teljesen lecseréljük őket, ugyanis ezek nagyon magas számítási kapacitást igényelnek. A kiindulási pont egy 2019-es cikk volt [G19], amiben gráfkonvolúciós hálók segítségével imitálták a strong branchinget.

A cél az algoritmus futásidejének minimalizálása, ami nagyban összefügg a branch and bound fa méretével. Az algoritmus során két kiválasztási feladatunk van: választhatunk, hogy melyik csúcsból branchelünk, illetve adott csúcsban melyik változó mentén branchelünk. Az utóbbinak nagyobb szerepe van a kialakuló fa méretében, így a legtöbb szabály a változókiválasztásra fókuszál. Ilyen a strong branching rule is, ami általában a legkisebb fát eredményezi a többi stratégiával összevetve. Ez a szabály minden szétválasztás előtt megnézi, hogy adott változó mentén branchelve mekkora javulást érünk el. Viszont ehhez minden potenciális változóhoz megoldja a hozzá tartozó két LP-t, ami nagy számítási kapacitást igényel, ezért a gyakorlatban nem is használják önmagában ezt a szabályt. Mi a projekt során a Scip nevű state of the art solvert [S] használtuk a MILP-ek megoldására, és az ebbe beépített strong branching szabályt imitáltuk.

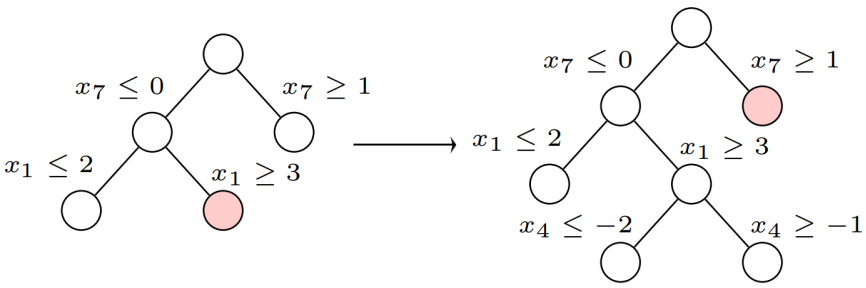
2. Alapfeladat

Az alapfeladat tehát a vegyes egészértékű programozási feladatok megoldása. Ezek a feladatok felírhatók a következő formában:

$$\arg\min_{\mathbf{x}} \left\{ \mathbf{c}^\top \mathbf{x} \,\middle|\, \mathbf{A}\mathbf{x} \leq \mathbf{b},\, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u},\, \mathbf{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \right\},$$

ahol $\mathbf{c} \in \mathbb{R}^n$ a célfüggvény együttható vektora, $\mathbf{A} \in \mathbb{R}^{m \times n}$ a feltételek együttható mátrixa, $\mathbf{b} \in \mathbb{R}^m$ a feltételek konstansainak vektora, $\mathbf{l}$, $\mathbf{u} \in \mathbb{R}^n$ rendre a változók alsó és felső korlátjának vektorai, és $p \leq n$ jelöli az egészértékű változók számát. A feladat relaxáltjának nevezzük azt a lineáris programozási feladatot, amelyben az eredeti feladatból elengedjük az egészértékűségi megkötést. A relaxált hatékonyan megoldható, és optimuma alsó korlátot ad az eredeti feladatra.

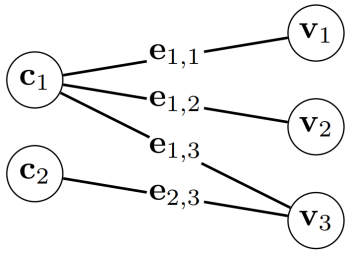
Amennyiben a relaxált optimális megoldása kielégíti az egészértékűségi megkötést, akkor az optimális megoldása az eredeti feladatnak is. Ellenkező esetben szétbonthatjuk az eredeti feladatot két részfeladatra, a megoldások terének kettévágásával egy olyan változó mentén, amely nem elégíti ki az egészértékűségi feltételt a jelenlegi megoldásban. Formálisan az eredeti feladathoz hozzávesszük rendre az $x_i \leq \lfloor x_i^* \rfloor$ és az $x_i \geq \lceil x_i^* \rceil$ feltételeket, ahol $x_i^* \notin \mathbb{Z}$ jelöli az x_i változó jelenlegi értékét. A branch and bound algoritmus rekurzívan alkalmazza ezt a lépést, és így létrehoz egy bináris fát. Ezen fa fontos tulajdonsága, hogy bármely lépésben a levelekben található legjobb LP optimum alsó becslést ad az eredeti feladat optimum értékére, míg a legjobb egészértékű megoldás - amennyiben létezik - felső becslést ad az eredeti feladat optimumára. Így könnyen látható, hogy az algoritmus vagy akkor áll le, amikor már nem tudjuk tovább bontani a feladatokat, vagy pedig a két becslés megegyezik egymással.



1. ábra. B&B fa egy lépése.

3. Adattudományi megközelítés

Fontos kérdés még, hogy hogyan hozzuk a feladatot olyan formájúra, amit a neurális háló kezelni tud. Ebben nyújt segítséget a gráfkonvolúció, amely bármilyen gráfot elfogad inputként. Ahogy a 2. ábrán is látszik egy LP feladat felírását gráfként a következőképpen tesszük [G19]. Létrehozunk egy G páros gráfot, ahol az egyik csúcsosztály a változóknak, a másik pedig a feltételeknek felel meg, és pontosan akkor megy él két csúcs között, ha az egyik végpontjában lévő változó szerepel a másik végpontjában lévő feltételben. Ezen kívül minden csúcshoz illesztünk feature-öket, azaz olyan információkat, amik együttesen jól megragadják a feladat struktúráját. Formálisan jelölje rendre $\mathbf{V} \in \mathbb{R}^{n \times d}$, $\mathbf{C} \in \mathbb{R}^{m \times c}$ és $\mathbf{E} \in \mathbb{R}^{m \times n \times e}$ a változók, a feltételek és az élek feature márixszait, ahol n, m rendre a változók és feltételek száma, d és c a hozzájuk tartozó feature-ök száma, e pedig az élek feature-einek száma. A feature-ök lehetnek például a változók korlátai, célfüggvénybeli együtthatóik, jelenlegi megoldásban az értékeik stb. A feltételek esetében például a duális megoldásait, a nemnulla együtthatók számát, a konstansokat, illetve több a Scip-ből még kinyerhető tulajdnoságot használtunk, az éleknél pedig a változót, a feltételt és az együtthatót. Így kódoltunk el egy MILP-et egy állapottá, ezt már tudjuk kezelni gráfkonvolúcióval.

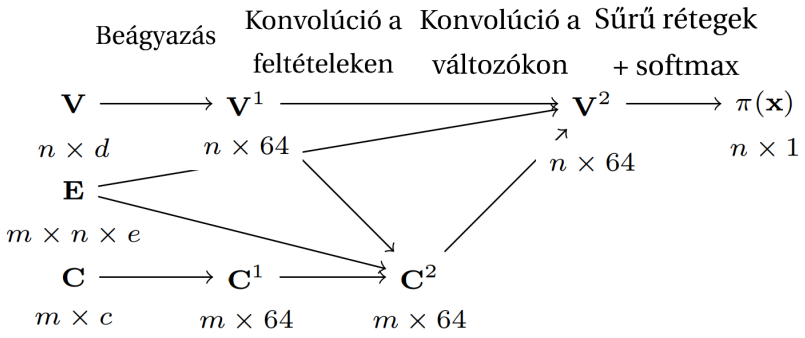


2. ábra. Egy páros gráf reprezentáció $n = 3$ változóval és $m = 2$ feltétellel.

A gráfkonvolúciós háló egyfajta kiterjesztése a konvolúciós neurális hálóknak. A sima konvolúció során egy filtert használva az egész inputon megtanuljuk a szomszédos cellák együttes feature-eit. A gyakorlatban ez azt jelenti, hogy egy kisebb súlymátrixot mozgatunk végig az input mátrixon és minden pozícióban kiszámoljuk a lefedett elemek szorzatát a megfelelő súlyokkal majd vesszük ezeknek az összegét. A gráfkonvolúció is hasonló abban az értelemben, hogy az input elemeinek együttes feature-eit számoljuk,

viszont a gráf struktúrának megfelelően itt a szomszédos elemeket vizsgáljuk egyszerre. Három tulajdonsága miatt jó választás a mi esetünkben: bármekkora gráfot fel tud dolgozni, számítási igénye a gráf sűrűségétől függ, és permutációra invariáns, azaz bármilyen sorrendben dolgozza fel a csúcsokat, ha ugyanaz a gráf, akkor ugyanaz lesz az output is.

Nálunk a teljes struktúra a változók és feltételek beágyazása után egy gráfkonvolúcióból és egy sűrű rétegből áll, ez látható a 3. ábrán. A modell első lépésként megkapja az eredeti gráfreprezentációt és elkódol minden csúcsot egy 64 dimenziós vektorrá a featuröket felhasználva. Mivel ez egy páros gráf, a konvolúciót két részre bonthatjuk, először elvégezzük a feltételek osztályára, majd a változók osztályára. Ezen lépések után egy az eredetivel izomorf gráfot kapunk, csak a featurök lehetnek különbözőek. Majd ebből a potenciális csúcsokra, amik mentén branchelhetünk kiszámolunk egy valószínűségi eloszlást egy 2 rétegű perceptronnal. Végül ez alapján kiválasztjuk melyik változó mentén branchelünk. Ez az architektúra látható nagy vonalakban a 3. ábrán.



3. ábra. A használt háló felépítése.

4. Előkészületek

A célunk az volt, hogy kimérjünk több különböző modellt több különböző feladaton. Jelenleg öt különböző modellel rendelkezünk, amik kisebb módosításai egymásnak. Az első modell a baseline, ami nagyban hasonlít a kiindulási cikkben bemutatotthoz, amit a 3. fejezetben ismertettünk. A másodikban mindenhonnan kivettük a dropoutokat csak a végső regressziós modulban hagytuk benne. A harmadik modellnél a nemlinearitá-

sok számát minimalizáltuk, csak a konvolúció során használunk nemlineáris függvényt, dropout egyáltalán nincs. Ezekkel ellentétben a negyedik modellt próbáltuk minél inkább feldúsítani nagyobb rétegekkel, több különböző konvolúcióval, dropouttal, nemlinearitással. Az ötödik modell pedig szimplán a baseline dropout rétegek nélkül. Mivel az elsődleges célunk a az alapok lefektetése volt, ezért csak a baseline modellel végeztünk méréseket.

Feladatból is öt darabot tűztünk ki: a csúcsfedést, ahol adott n elem, mint alaphalmaz, ennek részhalmazai, és ki kell választani a legkevesebb olyan részhalmazt, amik fedik az összes elemet. A független csúcshalmazt, ahol egy adott gráfban keresünk maximális független csúcshalmazt. A kombinatorikus aukciót, ahol egyszerre licitálnak termékcsomagokra és úgy kell eladni a termékeket, hogy a lehető legnagyobb legyen a bevétel. A létesítmény elhelyezést, ahol úgy kell elhelyezni adott számú létesítményt, hogy a távolságösszegük az összes ponttól minimális legyen. Végül a strip packinget, ahol előre adott téglalapokat kell elhelyeznünk tengelypárhuzamosan a sík egy adott szélességű részén úgy, hogy az általuk elfoglalt magasság a legkisebb legyen. Ezeken belül próbáltunk négy kategóriát létrehozni az alapján, hogy a Scip-nek mennyi ideig tart átlagosan megoldani őket. Teszteket eddig csak a normal mérettel végeztünk amire az egy percet határoztuk meg. Ehhez a legtöbb adat méretét könnyen be tudtuk állítani. Ahogy az a **??**. ábrán is látszik, az egy perc minden esetben úgy jött ki, hogy volt pár magasan kiugró érték, a legtöbb instancia megoldási ideje pedig egy perc alatt volt. A legnagyobb gondot a strip okozta, ugyanis bárhogy próbáltuk állítani a hiperparamétereket nem tudtuk kihozni az egy percet, és nagyon nagy szórást tapasztaltunk, volt, hogy ugynazon paraméterek mellett egy másodperc alatt és egy óra felett is oldott meg instanciákat.

A tanító adat generálás úgy történik, hogy létrehozunk tetszőleges számú véletlen instanciát egy adott problémából. Majd véletlenül mintavételezve a Scip segítségével elkezdjük felépíteni a branch and bound fát. Előre adott stratégia szerint a folyamat során néhány csúcsban megállunk, és az itt található LP-t elkódoljuk a fent ismertetett módon, eltároljuk a potenciális változókat, amik mentén branchelhetünk, kiszámoljuk az imitálni kívánt branching szabály által adott pontszámokat a változókhoz, végül kiválasztjuk a legjobbat. Illetve pár - a tanulás során közvetlenül nem felhasznált - információt is számon tartunk, például a használt seedet, az instancia sorszámát, amiből az adat készült stb. Ezeket mind lementjük és így kapunk egy darab tanító adatot. Az adat kinyerés

módja miatt előfordulhat, hogy egy instanciából egyetlen csúcsot sem használunk fel. Ez a jelenség rendszeresen megtörténik, ugyanis a 100000 generált instanciából kicsit több mint a felét használtuk csak fel az adat gyártása során, és érdekes, hogy az eredeti cikkben csak 20000 instanciát használtak a 100000 tanító adat kinyeréséhez, és abban az esetben is csak az instanciák fele volt hasznosítva.

Ezen kívül módosítottuk a kódot, úgy, hogy egy új fajta branching szabályt hozzunk létre. A folyamat ugyanaz volt mint korábban, viszont egy adott csúcsban, ahol mintavételeztünk nem a strong branching pontszámait számoltuk ki, hanem minden potenciális változó mentén manuálisan elbrancheltünk, és a két új feladatot külön-külön megoldottuk a Scip segítségével, majd az így kapott fák méreteiből számoltuk ki az új pontszámokat, amit a későbbiekben imitálni szeretnénk. Ezzel a szabállyal még nem végeztünk teszteket.

5. Eredmények

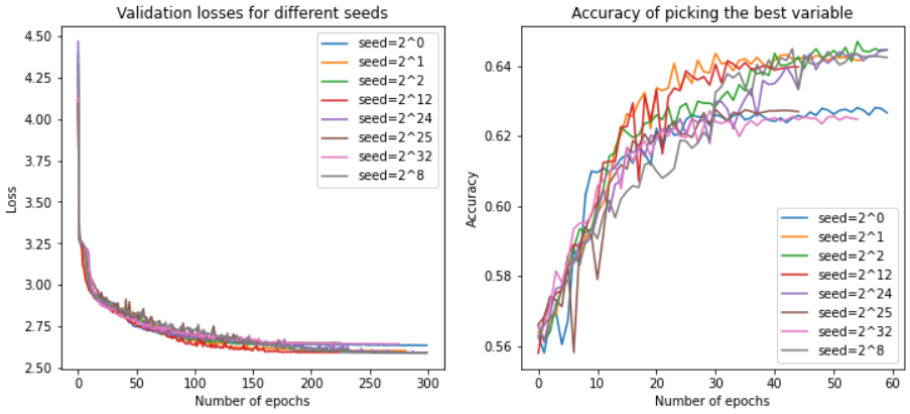
A félév legfontosabb feladata az volt, hogy az eredeti cikkhez képest készített változtatásokat kiértékeljük és lefektessük az alapokat, amikre később építkezhetünk és amihez az új eredményeket hasonlíthatjuk. Az eredeti kódhoz képest több változtatás is történt. Fontos technikai módosítás, hogy tensorflow helyett pytorchot használunk, illetve a teljes inputon végigiterálunk, így minden tanító adatot felhasználunk, míg eredetileg egy epoch során egy kisebb mintán tanítottak. Másik jelentős változtatás, hogy több különböző modell változatot hoztunk létre, például változtattuk a különböző rétegek számát, és méretét, használtunk dropoutot, illetve kihagytuk a prenorm layert. Ezek mindegyikét nem teszteltük behatóan, ugyanis kiindulási pontként csak a baseline modellt használtuk.

A kiindulási cikkben a betanított hálók ugyanazon a feladaton kiértékelve, amin tanultak a Scip-hez képest, kb. 40%-os gyorsulást értek el átlagosan az instanciák megoldási sebességében. Ezt a kutatócsoport is tudta reprodukálni. Azóta átálltunk nagyobb méretű tanító adatokra, ugyanolyan háló méret mellett, és az így készített mérések messze elmaradnak ettől az eredménytől, teljesítményük sokkal rosszabb, mint a Scip-é.

Ahhoz, hogy minél jobb eredményt kapjunk, a hiperparamétereket kezdtük vizsgálni. Ezen belül is négyre koncentráltunk: a batch méretre külön, és a learning rate-tel együt-

tesen, a learning rate decay-re és a modell véletlen seedjére, mindet a baseline modellen vizsgáltuk.

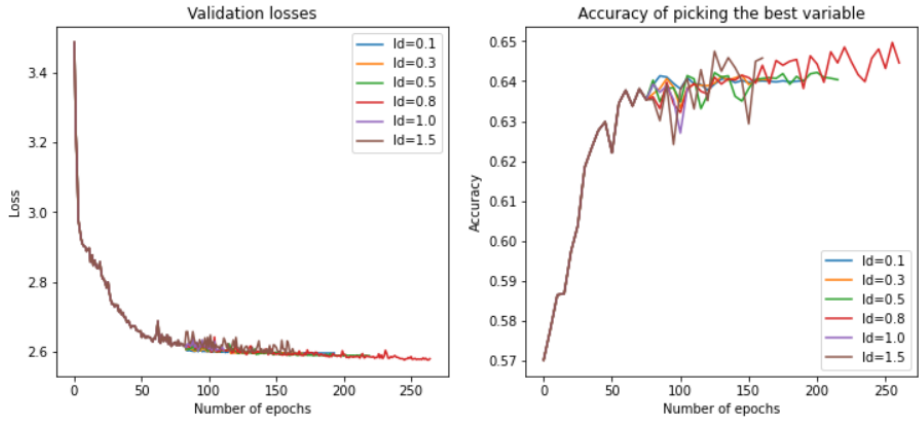
A seed tesztelését azért vettük előre, mert volt olyan tapasztalata a kutatócsoportnak, hogy különböző seedeket használva, jelentős eltérést eredményez. Az ezzel kapcsolatos futtatásaink során 2 hatványokat vettünk, véletlenül választva a seedek lehetséges intervallumából, ami a $[0, 2^{32})$. Ezek a tesztek azt mutatták, hogy nem számít, milyen seeddel inicializáljuk a kódban használt véletlen generátort, a modell nagyságrendileg ugyanolyan sebességgel tanult, és ugyanolyan jól teljesített. Az 4. ábra is azt mutatja, hogy a két fontos mértékegység, amit a tanulás során figyelembe veszünk ugyanazt a görbét eredményezi minden modellen. Ez a két mértékegység pedig a loss, esetünkben cross-entropy és az accuracy, hogy milyen gyakran találjuk el a legjobb strong branching score-ral rendelkező változót.



4. ábra. A különböző seedel inicializált baseline modellek tanulása a kis méretű setcover feladaton.

A learning decay tesztelésre azért volt szükség, mert úgy láttuk, hogy nagyon hamar kilapul a tanítási görbe, és a learning rate mesterséges csökkentése potenciális okozója lehet ennek a jelenségnek. Ezen kívül a modellben Adam optimizert használunk, amibe alapból be van építve, hogy automatikusan állítsa a learning rate-et a tanítás során, így ez a hatás csak fokozódott. Azt tapasztaltuk, hogy nagyságrendileg minden teszten ugyanazt az eredményt értük el, ugyanabban az ütemben. A 5. ábrán megfigyelhető, hogy, ameddig folyamatosan javul a modell teljesítménye, addig együtt haladnak a gör-

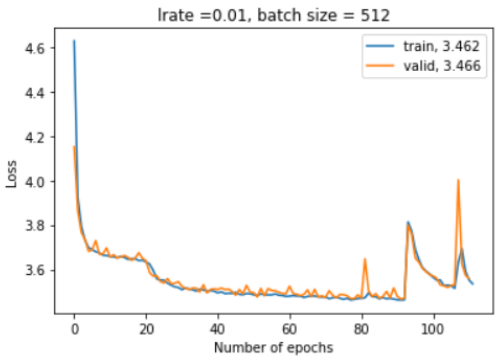
bék, és amint alkalmazzuk a learning rate csökkentését szétválnak. Ez igaz a loss-ra és az accuracy-re egyaránt.



5. ábra. Különböző learning rate decay-ek tanulási görbéi.

A batch size-zal azt szerettük volna megvizsgálni, hogy tudjuk-e növelni a tanítás sebességét. Ugyanis a batch mérete nagy szerepet játszik abban, hogy a tanítás mekkora számítási kapcitást, illetve memóriát igényel. A mi esetünkben a tanítás GPU-kon zajlott és 1024-es batch mérettel használtuk ki majdnem teljesen a memóriát, viszont, ahogy csökkentettük a méretet, rohamosan nőtt a CPU használat is, ami legfeljebb egyszerre egy tesztet tett lehetővé 64-es batch mérettel. Emellett nem tapasztaltunk jelentős eltérést, sem a tanulás sebességében, sem pedig annak eredményességében.

A batch size és a learning rate együttes vizsgálatával szerettük volna megtalálni a legjobb beállítást, amivel a modell a leggyorsabban tanul. A legtöbb esetben itt is nagyságrendileg ugyanolyan eredményeket tapasztaltunk. Ez alól kivétel volt, amikor nagyobb, 0.1 körüli learning rattel próbálkoztunk kis batch mérettel párosítva, ekkor egy darabig javult a modell, viszont egy idő után rohamos romlást mutatott és leállt. A 6. ábrán is egy ilyen eset figyelhető meg, pár epochon át a szokásos



6. ábra

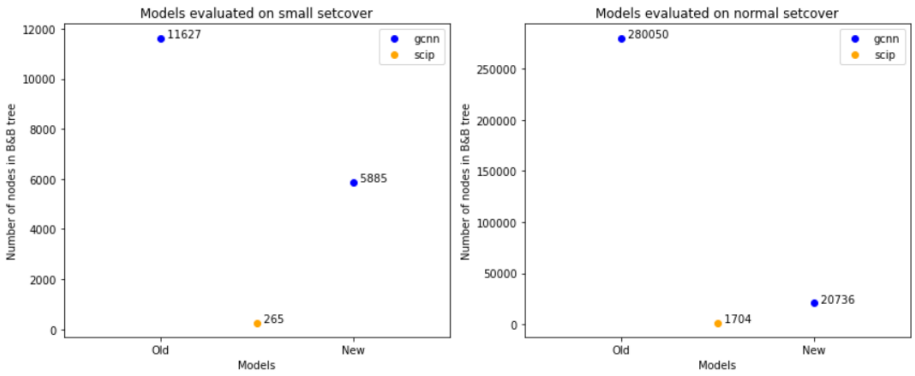
módon csökkent a loss a tanítás során, majd egy idő után elkezdett egyre rosszabb értékeket produkálni, és végül leállt.

5.1. További tapasztalatok

Egyetlen olyan tényezővel találkoztunk, ami nagyban befolyásolta a háló eredményességét, ez pedig az adat mérete volt, amin a tanítást és a kiértékelést végeztük. A projekt elején az örökölt adatokkal dolgoztunk és később átálltunk az általunk normálnak hívott méretre, ami körülbelül a setcover esetében másfélszer annyi sort és oszlopot jelentett, de a háló méretét nem változtattuk. Így várható volt valamiféle romlás, viszont meglepően rossz eredményeket kaptunk. Ugyanaz a háló architektúra a kisebb adaton majdnem 100%-os eredményességgel választott olyan változót, amiknek a strong branching scoreja az első tízben volt, és több mint az esetek 60%-ában választotta a legjobbat, a nagyobb adaton a legjobb tízből az esetek felében választott, a legjobbat pedig a csúcsok kb hatodában találta el.

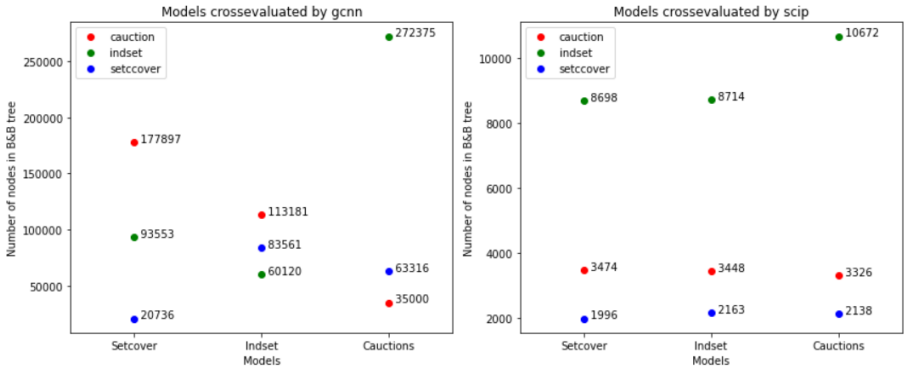
Ennek ellenére egy furcsa jelenség jött létre. A Scip által épített fa átlaga hat és félszeresére nőtt a nagyobb adathalmazon. Az a modell, amit a régi adaton tanítottunk, nagyon rosszul általánosodott a nagyobb adathalmazra. Míg az új adaton tanított modell viszonylag kis B&B fa méretet produkált és jobban teljesített a kisebb adathalmazon is, mint az a modell, amit azon tanítottunk, pedig abban az esetben sokkal hatékonyabban imitáltuk a strong branching szabályt. A 7. ábrán a bal oldali értékek a régi, kisebb setcover, míg a jobb oldaliak az új, nagyobb setcover instanciáin végzett kiértékelés eredményét mutatják. Mindkét ábrán a bal oldalon a kis feladaton tanított modell által épített fák átlagos csúcsszáma látható, középen a Scip által építetteké, jobb oldalon pedig a nagyobb adathalmazon tanítotté.

A tesztek nagyon különböző sebességgel futnak, így nem ugyanannyi kiértékelt adattal rendelkezünk a különböző feladattípusokból. Ezért a végzett tesztek sajnos nem teljesen pontosak, azt láttuk, hogy az ezres minta és az abból vett véletlen 150-es minták átlaga között körülbelül 1000-es eltérés van a B&B fa csúcsainak számában.
Ennek ellenére az eddigi értékekből kiolvasható, hogy az általunk tanított modellnek a legnehezebb feladat a kombinatorikus aukció és a független csúcshalmaz keresés volt, a Scip-nek pedig messze a független csúcshalmaz választás. Illetve azt is megfigyelhet-



7. ábra. A különböző méretű setcover feladatokon kiértékelt modellek és a Scip megoldása.

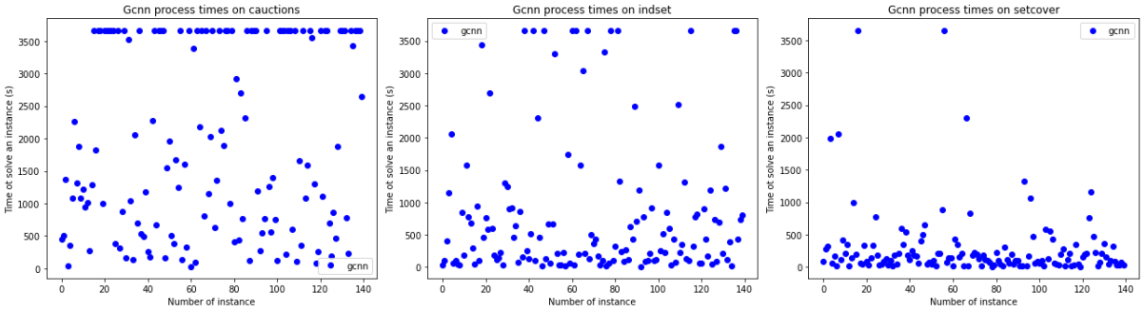
jük, hogy azon a feladaton kiértékelve egy modellt, amin tanítottuk, kb. tízszer annyi csúcsot produkál, mint a Scip, legjobban a független csúcshalmazon tanított háló általánosodott. Ez kiolvasható 8. ábrából, ahol a bal oldalon a három feladattípuson tanított hálókat értékeltük ki mindhárom feladaton és az ezek során épített B&B fák csúcsszámának átlagát ábrázoltuk, jobb oldalon pedig, az ugyanezeken az instanciákon futtatott Scip által épített fák csúcsszámának átlagai láthatók.



8. ábra. A modellek és a Scip által épített fák csúcsszámának átlaga a három feladat instanciáin.

A modell értékei pedig a valóságban még ennél is rosszabbak ugyanis a kiértékelés során a futási időt egy órában maximalizáltuk és sok esetben elérte ezt a maximumot. A 9. ábrán is ez a jelenség látható, a három grafikon tartalmazza instanciákra bontva a Scip

futásidejét, a pontok pedig a felső részen, mind a 3600 másodperces értéket veszik fel, ami azt jelenti, hogy a feladat megoldása az időkorlát miatt félbeszakadt.



9. ábra. Az egyes feladatok instanciáinak megoldásával eltöltött idő a modell által.

6. Összefoglaló

A félév során apróbb változtatásokat végeztem a kódban, hogy könnyen el tudjuk végezni a méréseket. Első lépésként hiperparaméter optimalizálást végeztem figyelembe véve a learning rate-et, a batch size-t, a learning rate decay-t és a véletlen seed-et. Betanítottam a baseline modellt három feladattípuson és kiértékeltem őket egymáson. Illetve megvizsgáltam hogy viszonyul egymáshoz a két különböző méretű adaton tanított modell. Ezen kívül írtam egy újabb - adott csúcs alatti részfa méretét figyelembe vevő - branching score-t számoló kódot.

A következő félévben első lépésként újra reprodukálni szeretnénk a Scip-nél jobban teljesítő modelleket, mind a kisebb, mind a nagyobb adathalmazon. Ezután három nagyobb területet tervezünk vizsgálni: az egyik a branching score-ok, a második az LP feladatok állapotokká való elkódolása, a harmadik pedig a használt háló architektúrák. Mindegyikben benne van a lehetőség, hogy jelentős javulást eredményezzen.

Hivatkozások

[G19] Gasse, Maxime, et al. "Exact combinatorial optimization with graph convolutional neural networks." Advances in Neural Information Processing Systems 32 (2019).

[S] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Le Bodic, P., Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Muhmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, & Jakob Witzig (2020). The SCIP Optimization Suite 7.0 [White paper]. Optimization Online.