Kombinatorikus optimalizálási sejtések számítógépes ellenőrzése

Kelemen Ádám Olivér

2022. május

Az előző félév során a projekt keretein belül kezdtem el foglalkozni ismét a C++ nyelvvel és a témakör kapcsán a LEMON library-vel is sikerült némileg megismerkedni. A félévre kitűzött cél a Woodall-sejtés egy speciális esetének a számítógépes ellenőrzése volt kis gráfokra. A sejtés azt mondja ki, hogy ha $D = (V, A)$ irányított gráf, melyben a legkisebb irányított vágás mérete k – azaz, ha a csúcsainak van olyan $X \subseteq V$ részhalmaza, melyből nem vezet ki él, akkor az ilyen X részhalmazba belépő élek száma legalább k, – akkor D-ben van k db diszjunkt dijoin. Egy *dijoin* alatt az élek egy olyan részhalmazát értjük, mely D minden irányított vágását lefogja, azaz minden irányított vágásból tartalmaz legalább egy élt. Ennek a sejtésnek egy, a síkgráfokra vonatkozó speciális esetével foglalkoztam, melyhez a síkgráfok duálisát felhasználva juthatunk el. Az alap sejtés azt állítja, hogy ha D egy síkbarajzolható digráf, melynek az összes irányított köre legalább k hosszú, akkor D tartalmaz k db, páronként diszjunkt feedback arc set-et. Ennek a $k = 3$ esetét felhasználva mondható ki az a sejtés, melynek kis gráfokon való számítógépes ellenőrzése az előző félévem feladatát képezte, azaz, ha D irányított, erősen összefüggő gráf, akkor tartalmaz három, páronként diszjunkt feedback arc set-et. Egy *feedback arc set* alatt a digráf éleinek egy olyan részhalmazát értjük, ami minden irányított körből legalább egy élt tartalmaz. Ebből következően, ha a gráfból elhagyom a feedback arc set-be tartozó éleket, akkor a gráf körmentes lesz, azaz *DAG*.

A probléma megoldásához első lépésként egy feedback arc set megtalálására volt szükség, melyhez egy frissen megjelent cikket[1] használtunk fel. A cikkben több megoldás is található minimum feedback arc set keresésére, többnyire IP feladatra építve, így ebbe az irányba indultunk el és egy háromszög-egyenlőtlenséget felhasználó IP feladat programozásába kezdtem bele. A programozási feladat tekintetében ez egyszerűbb megoldásnak számít, de szükség is volt erre a választásra, mert mind a C++, mind a LEMON esetén találkoztam több olyan problémával is, amiken csak hosszas böngészés és kódelemzés útján tudtam túllendülni. A kitűzött célt végül sikerült elérni és a legkisebb elemszámú (minimális összsúlyú is lehetett volna) feedback arc set megtalálására létrehozott IP feladatot követően, a három diszjunkt – ekkor már nem feltétlenül minimális méretű – feedback arc set-et megtaláló modell is elkészült. A program több, mint 17 ezer gráfot ellenőrzött, feldolgozva ezzel az összes, izomorfia erejéig különböző, legfeljebb 6 csúcsú irányított síkgráfot és ezeken nem talált ellenpéldát a sejtésre.

A feladat érdekes volt és sokat tanultam közben, de amivel a tavalyi beszámolóm

is zártam, szerettem volna egy gráfalgoritmust implementálni C++-ban. Elsőre túl nagy feladat lett volna, most már viszont elérhető célnak tűnt, így ebben a félévben kicsit eltávolodtunk az eredeti „kombinatorikai sejtések ellenőrzése számítógépes módszerekkel" témakörtől és az előző félévből már ismert, minimum feedback arc set-et megtaláló algoritmust tűztünk ki célul, immáron már IP feladat használata nélkül.

Forrásnak még mindig rendelkezésünkre állt a korábbi Gurobi cikk[1], de abban, néhány heurisztikát leszámítva, IP feladatot tartalmazó algoritmusok vannak, így a félév egy kis kutatómunkával indult. A témában viszonylag sok cikket lehet találni, de a legtöbb speciális gráfokra vagy esetekre vonatkozik. Több cikket is találtam, ami tournament-ekkel foglalkozik, illetve approximációs algoritmusokkal foglalkozó cikkeket lehet még találni, ami nem is meglepő, hiszen a minimum feedback arc set megtalálása NP-nehéz feladat. Végül találtam egy cikket[2], ami egy Monte-Carlo algoritmust használt a minimum feedback arc set megtalálására. Bár a Monte-Carlo sem ad optimális megoldást, de a cikkben szerepel egy *Branch and Bound* algoritmus, ami felkeltette az érdeklődésem. A cikk egy ISSDO, *Information System Subsystems Development Order* nevű problémával foglalkozik, ami megértésem szerint, egymással kommunikáló és / vagy összefüggésben lévő rendszerek fejlesztési sorrendjének optimális meghatározása. Ilyenek lehetnek pl. microservice struktúrában felépített rendszerek adatbázis eléréssel. A cikk nem a legjobb, néhol olyan, mintha egyértelmű állításokat bizonyítana feleslegesen, máshol pedig ködös és nem derül ki pontosan, hogy mi támasztja alá az adott állítást vagy adatot. Továbbá a cikk egy jelentős részén azzal foglalkozik, hogy bizonyítja, hogy az általa felvetett ISSDO probléma NP-nehéz, miután már korábban párhuzamba állította a minimum FAS problémával és bemutatta, hogy felfogható annak, egy speciális esetének. A használt matematikai jelölésrendszer nagyon nehezen értelmezhető és a cikk bár leírja, hogy az algoritmusok Python-ban lettek implementálva és bemutat futásidőket, de a forráskód nem elérhető, ellenőrzésre nem ad lehetőséget, nem úgy, mint pl. a Gurobi cikk[1]. Ezek után nem meglepő, hogy Péterben kétségek merültek fel a cikk használhatóságát illetően, de végül megbeszéltük, hogy adunk neki egy esélyt, hátha sikerül megfejteni a leírt algoritmust és esetleg alapot adhat a továbbfejlesztésre. Így el is kezdtem vele foglalkozni. Az algoritmus nem bonyolult, de a leírása nem egyértelmű, egy jelentős lépése felett sikerült átsiklanom, egy konzultáció alkalmával Péter figyelt fel rá.

Az algoritmus alapvetően azt a szisztémát használja, amit az előző félévben használt IP feladat is, felépíti a csúcsok egy permutációját és az ott visszafele mutató élek súlyait adja össze. Ezt úgy teszi meg, hogy egy fát kezd el építeni, minden csúcshoz az eredeti gráf egy csúcsa fog tartozni és ezek mentén növeszti a fát. Ha nem tudja tovább növeszteni, az azt jelenti, hogy elfogytak a gráf csúcsai, azaz a felépített fában a gyökértől indulva az utoljára beszúrt levélig, az eredeti gráf csúcsainak egy permutációja fog szerepelni. Ekkor ezen az ágon kiszámítva a visszafele mutató élek összsúlyát, kapunk egy potenciális megoldást a minimum FAS problémára, mert a visszafele mutató élek pont egy FAS elemei. Az algoritmus elején egy heurisztika segítségével ad egy felső becslést a minimum FAS súlyára, így a kapott potenciális megoldásokból mindig a legjobbra cseréljük az aktuális értéket. Itt észrevehető, hogy a leírás szerint a gyökér elem mindig azonos lesz minden növesztett ág esetén.

Ezt az algoritmus úgy oldja meg, hogy egy ciklusban az eredeti gráf minden csúcsát felhasználva elkezd egyszer fát építeni, így minden csúcs lesz gyökér elem.

Ezzel a fa építés folyamata meg is van, viszont így a teljes fa, az összes ágával együtt létrejönne, azaz az eredeti gráf csúcsainak összes permutációján kiszámoltuk a visszaélek összsúlyát, amivel így biztosan megkapjuk az optimális megoldást, ellenben lassú. Ahhoz, hogy ne kelljen a teljes fát feldolgozni, „stop" vagy „vágás" feltételeket definiálhatunk, melyek megállítják egy ág növesztését, ha arról már tudható, hogy biztosan nem kapunk a végén jobb megoldást, mint a jelenlegi. Az algoritmus ehhez a fában lévő minden csúcshoz hozzárendeli az ágon lévő, az adott csúcsig számított visszaélek összsúlyát és ezt örökölteti. Azaz a gyökér elem esetén ez az érték 0. A következő csúcsnak az értéke 0, plusz a belőle, a gyökérbe menő visszaél súlya, ha létezik ilyen. Az öröklést felhasználva egy új csúcs növesztése esetén a fára, csak azoknak az éleknek a súlyait kell összeadni, amik belőle az őt megelőző csúcsokba mutatnak. Ez az érték valójában az, hogy ez a csúcs a permutáció adott pozíciójában mennyivel járul hozzá az ágon számítható FAS összsúlyához. Könnyen belátható, hogy ha egy új csúcs bevétele esetén átlépjük a jelenleg tárolt optimális súlyt, akkor ehhez az ághoz újabb csúcsokat hozzávéve – mivel negatív éleket nem engedélyezünk – csak rosszabb lesz, így ekkor ez a részfa „levágható".

Egy további, személyes észrevétel, hogy ebben az esetben nem csak ez a gyerek csúcs vágható le, hanem a vele egy szinten, azonos szülőhöz tartozó összes ág, azaz maga a szülő csúcs és az összes leszármazottja törölhető. Vegyük észre, hogy mivel az új csúcs esetén a kiszámított súly, mely meghaladta a jelenleg tároltat, az a szülőig kiszámított részpermutációhoz tartozó FAS súlya, valamint az új gyerekből az őt megelőző csúcsokba mutató élek összsúlya. Azaz a szülőben tárolt súly értéket minden gyereke meg fogja kapni és mivel ez alatt a szülő alatt a még fel nem dolgozott gráf csúcsok minden permutációja szerepelni fog egy ágon, ezért az a gyerek, amely bevételével az összsúly átlépte a jelenlegi értéket, minden ágon szerepelni fog és minden ág legalább ekkora súlyt fog elérni, így azok is levághatóak lesznek.

Az algoritmus utolsó eleme a következő feldolgozandó csúcs kiválasztása. Egy csúcs feldolgozása abból áll, hogy létrehozzuk az összes gyerekét és mindnek kiszámítjuk a súlyát, majd bekerülnek a feldolgozandó csúcsok közé. A következő feldolgozandó csúcs a legkisebb súllyal rendelkező csúcs lesz, mert potenciálisan ezen az ágon haladva tudunk majd javítani a jelenlegi optimális értéken. Ez azt jelenti, hogy a fát nem áganként növesztjük, hanem ugrálva az ágak között, inkább egy szélességi bejáráshoz hasonló feldolgozás történik. Van azonban még egy feltétel az algoritmusban, hogy amennyiben több azonos legkisebb súllyal rendelkező csúcs is van a feldolgozandó listán, akkor azt kell választani, amelyik a fában a legmélyebben van, azaz a leghosszabb permutációt növesztjük tovább. Ennek az az előnye van, hogy lehetőséget adunk az optimum javításának, ami korábban levágott ágakat eredményezhet, ami az algoritmus futásidején is javít.

Ez tehát a cikkben szereplő algoritmus, aminek az implementálásába belekezdtem. A fejlesztés közben az algoritmus is több plusz funkciót, módosítást kapott. A cikkben is említik, hogy lenne még lehetőség optimalizálni rajta, de nekik csak referenciának kell, a Monte-Carlo algoritmus mellé.

Csináltam egy véletlen gráf generátort, ezzel hoztam létre gráfokat és azokon próbálgattam. Végül három gráf maradt, amikkel dolgoztam. Egy 5 csúcsú, mely a

Monte-Carlo cikkben[2] szerepel és még alkalmas arra, hogy logolás mellett futtassam az algoritmust, egy 10 csúcsú, mely 65 irányított éllel rendelkezik, amik esetén az oda-vissza él lehetséges, de hurokél nem, és minden élsúly 1 v 2, valamint egy 11 csúcsú 92 éllel és fix 1 élsúlyokkal, szintén hurokélek nélkül. A két utóbbi gráf feldolgozási ideje között jelentős különbség volt. A cikkben is említik és várható is volt, hogy a gráfok tulajdonságai jelentősen kihatnak az algoritmus futásidejére. Kezdetben a 10 csúcsú 19s alatt végzett, míg a 11 csúcsúhoz 450s körüli időre volt szükség.

Az első nagyobb módosítás, Péter javaslatára, hogy próbáljuk meg kivenni az algoritmus elejéről a ciklust, mely a gyökér elemért felel és kezdjük el egyben feldolgozni az egész gráfot. Ez a 11 csúcsú gráf esetén kb. 15s javulást eredményezett, ellenben a 30–180MB memória helyett $3,5$GB-ot fogyasztott el.

Egy másik, a futásidőket leginkább érintő változtatás, szintén Pétertől, hogy az élsúlyok tárolására összedobott class-t cseréljem le NodeMap-re. Számítottam javulásra és nem is szántam végleges megoldásnak a lecserélt class-t, de ekkorára nem: a futásidők a negyedükre estek, 100s és 5s körülre. Ekkor elkezdtem számolni, hogy az algoritmus hány fa csúcsot hoz létre, mert a kis futásidők között nehéz volt különbséget tenni, ez meg egy mérőszáma a futásnak.

Ezután belekezdtem az algoritmus működését leginkább átalakító módosításba, mely sok hibával, hibás futással és hibás, de szerencsés futással járt, mely során reprodukálhatatlanul jó futásidőket kaptam, melyek túlvágás eredményei voltak. A fa csúcsokban tárolt részleges FAS súly mellett összegyűjtöttem az új gyerek csúcs esetén a csúcsba beérkező, tőle hátrébb lévő csúcsokból érkező élek összsúlyát. Ezek ugyanúgy előre mutató, a teljes permutációt tekintve a FAS-be tartozó élek, mint az eredeti tárolt súly, azzal a jelentős különbséggel, hogy ezáltal a még fel nem dolgozott csúcsokról kapok információt. Ezek a bemenő súlyok ugyanúgy örökölthetők, mint a korábbiak és a teljes permutáción végigérve ugyanazt az értéket adják, mint a gyökértől lefelé gyűjtött részleges FAS súly, viszont ez nagyobb értékeket vesz fel, ezáltal jobb vágásokat eredményez, mert a fa gyökérhez közelebbi részén már meghaladja az aktuális optimumot, ha amúgy is meghaladná később. Ezzel, valamint a kezdeti optimumot szolgáltató heurisztikában talált hiba javítása után a 11 csúcsú gráf feldolgozási ideje a legelső 450s-ről 13–16s-re volt redukálható, a korábban az algoritmus futása során létrejött $28,5$ millió fa csúcs helyett pedig $3,5$ millió alatt jutott eredményre az algoritmus.

Persze ezektől az eredményektől függetlenül, az algoritmus még mindig valami $k * n!$ lépést tesz, így ezen még bőven van mit javítani. Összehasonlításnak, mind eredmény, mind futásidő, beemeltem az előző féléves IP alapú minimum FAS kereső programot és az eredmény nem összehasonlítható, az IP sokkal gyorsabb. Viszont bőven van még ötlet az algoritmushoz, most folyamatban van egy plusz vizsgálat, amivel a vágásokon tudnék javítani, az új gyerek csúcs utáni, feldolgozatlan csúcsok között a diszjunkt körök kiszámítása, mely olyan plusz éleket tud felfedni, amik csak akkor kerülnek elő, ha már eljutott odáig a növesztett ág. Emellett a memóriahasználaton is van mit javítani, egyelőre a létrehozott fa csúcsokat nem törlöm dinamikusan, csak foglalják a helyet, ami nagyobb csúcsszámú gráfok esetén már komoly problémát jelent. Egy másik ötletem a memória jobb kihasználására, hogy a jelenlegi ugrálós, inkább szélességi bejárást dinamikusan változtatnám a feldolgo-

zási állapot függvényében. Lehetőségek vannak még bőven, a távlati cél pedig a Gurobi cikkben[1] bemutatott algoritmussal összevetni, de az 100 csúcs fölötti, bár ritka gráfokon is jól szerepelt.

Hivatkozások

[1] Ali Baharev, Hermann Schichl, and Arnold Neumaier: An Exact Method for the Minimum Feedback Arc Set Problem, University of Vienna, Tobias Achterberg, Gurobi GmbH, *ACM Journal of Experimental Algorithmics* Vol. 26, No. 1, Article 1.4., March, 2021.

[2] Robert Kudelić: Monte-Carlo randomized algorithm for minimum feedback arc set, University of Zagreb, Faculty of Organization and Informatics, *Applied Soft Computing* Vol. 41, pp. 235-246, 2016.

[3] `https://bitbucket.org/sheepconquest/math_projects/src/develop/`