Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Számítástudományi és Információelméleti Tanszék

Kvadratikus optimalizálás kvantum alapú számítógéppel

TDK dolgozat

Készítette:

Zalavári Márton

Konzulens:

dr. Friedl Katalin

2021

Tartalomjegyzék

Kivonat

A kvadratikus programozás a lineáris programozásnál egy általánosabb technika, hiszen megengedi négyzetes tagok jelenlétét a célfüggvényben. Ezzel az alkalmazások körét jóval kibővíti, ugyanakkor az általános feladat megoldása sokkal nehezebbé válik.

Ez a fajta optimalizációs technika többek között azért is érdekes és hasznos, mert ha a változók binárisak és nincsenek további korlátjaink, akkor a probléma megoldásához felhasználható egy kvantumállapotokat használó számítógép, ezzel remélhetőleg jelentősen csökkentve az optimalizáláshoz szükséges időt. A szakirodalom egyszerűen csak QUBO (Quadratic Unconstrained Binary Optimization) néven hivatkozik erre a fajta felírásra.

Gráfokon vágások keresése a (számítógép)hálózatok megjelenése óta egy sokat kutatott tématerület. Maximális vágást találni közismerten NP-nehéz probléma, ugyanakkor gyakorlati szempontból fontos, hiszen például a tipikus klaszterezési problémák megfogalmazhatók így, ha az adatot gráfként tudjuk reprezentálni.

A maximális vágással kapcsolatos problémákra többféle QUBO felírást is adok, melyeket elméleti és gyakorlati szempontból is összehasonlítóan elemzek. Az elkészített formulákat több szempontból elemzem, például a legegyszerűbb ilyen összehasonlítási metrika a felhasznált változók száma.

A QUBO-k optimalizálásához a D-Wave Ocean nevű programcsomagját használtam fel, mely több lehetőséget kínál a formulák megoldására. A klasszikus megoldók mellett lehetőség van például valódi, a D-Wave Systems által forgalmazott kvantumszámítógépeket is használni, vagy a klasszikus és kvantum eljárásokat hibrid módon ötvözni.

Összehasonlítom a különböző lehetőségekből adódó módszereket azok eredményessége és hatékonysága alapján. Természetesen a kapott eredményeket összevetem más, klasszikus heurisztikus megoldók használatával is.

A munka jelentős részét tette ki számos tapasztalat gyűjtése a D-Wave-es programcsomaggal kapcsolatban, hiszen a terület újdonságából kifolyólag az elérhető dokumentációk meglehetősen limitáltnak bizonyultak.

Abstract

Quadratic programming is a more general process than the linear programming, since we are allowed to use a quadratic function as the objective function. This extends the possible applications, but solving the general problem becomes much more complex.

One of the reasons this kind of optimization technique can gain interest, is because if we restrict the variables to be binary and do not use any constraint functions, then for solving the problem we can utilize a computer using quantum states, thus hopefully drastically reduce the time needed for optimization. The literature simply calls this formula QUBO (Quadratic Unconstrained Binary Optimization).

Finding a maximum cut in a graph is a well-known NP-hard problem, but at the same time, it is a very significant question in practical sense, because typical clustering problems can be described this way, if we convert the data into graph representation.

I present multiple different formulas related to maximum cut, which I compare from a theoretical and practical point of view. I analyze the formulas by different metrics, such as the number of used variables, which is one of the simplest example.

For optimizing the QUBOs I use the D-Wave Ocean tools, which provide multiple methods for solving the formulas. Besides the classical solvers, there is possibility to utilize real quantum machines issued by D-Wave System or combining classical and quantum approaches in a hybrid way.

I compare the different methods based on their effectiveness and efficiency. Naturally I juxtapose the results with using other classical heuristic solvers as well.

A significant part of the work consisted of obtaining plenty of experience by using the D-Wave tools, since the available documentation proved to be quite limited, because the field has just newly emerged.

1. fejezet

Bevezetés

A dolgozat motivációjának hátterében a D-Wave Systems által forgalomba hozott kvantumszámítógépei állnak, melyekről azt állítják, hogy hatékonyan oldanak meg kvadratikus korlátmentes bináris optimalizálási (QUBO) feladatokat, melyek egy nagyon speciális alesete a kvadratikus programozási feladatoknak, bonyolultságelméleti szempontból még mindig NP-nehezek.

QUBO segítségével viszonylag természetesen felírhatók különböző problémák, mint például a maximális vágás, melyet találni közismerten NP-nehéz probléma. Ugyanakkor gyakorlati szempontból fontos, hiszen például a tipikus klaszterezési problémák megfogalmazhatók így, ha az adatot gráfként tudjuk reprezentálni, illetve rengeteg más alkalmazási terület mellett nagy jelentősége van például a VLSI huzalozásban [24][23].

Dolgozatomban arra keresem a választ, hogy vajon az új technológiák mennyivel tették könnyebbé különböző problémák formalizálását és megoldását mind elméleti, mind gyakorlati síkon, valóban segíti-e a D-Wave rendszere bizonyos NP-nehéz optimalizálási problémák megoldását. Bár a létrehozható qubitek száma manapság még rendkívül limitált, így is érdekes kérdés, hogy kis bemenetekre látszik-e valami biztató eredmény, illetve a nagy bemenetekkel miképpen birkózik meg akár egy kvantum alapokon működő, akár egy klasszikus optimalizáló.

Munkámban az 2. fejezetben röviden áttekintem az optimalizálás témakörét definiálva a lineáris illetve kvadratikus programozás fogalmát, majd konkrétan bevezetem a QUBO fogalmát (2.2. szakasz) is. Elemzem a korlátmentes és korlátos optimalizálási feladatok közötti különbséget, megállapítva, hogy korlátos feladatból gyakorlatban mindig lehet korlátmenteset csinálni, noha ez nem mindig célravezető (2.3. szakasz). Végül a fejezet végén kitérek a QUBO-specifikus megfontolásokra, annak motivációs hátterére, a D-Wave gépek működési elvére, és a formalizálás nehézségeire illetve korlátaira (2.4. szakasz). Ezen felül itt definálom még a kvadratizálás, és a változók gráfjának fogalmát, melyekre később is többször visszautalok.

A 3. fejezetben konkrét feladatok felírását nézem meg, több különböző alakban. Az egyik konkrét feladat az élsúlyozott gráfon maximális vágás keresése, melyet a 3.1. szakaszban fejtek ki. A feladatot röviden elemzem bonyolultsági szempontból, és vázolok egy gyors 2-approximációt elérő, polinom idejű algoritmust is (3.1.1. alszakasz). Ezután megadom a probléma felírását lineáris program segítségével, illetve QUBO használatával is (3.1.2. alszakasz, 3.1.3. alszakasz).

A másik nagy volumenű probléma, melyet górcső alá veszek, az a maximális K-vágás témaköre, mely a sima maximális vágás egy fajta általánosítása (3.2. szakasz). Miután

definiálom a problémát, és megadom egy lineáris programként való felírását (3.2.1. alszakasz), általánosítom az egyszerű maximális vágás QUBO alakját, és megfogalmazom ezt a feladatot is ilyen formátumban (3.2.2. alszakasz). Felismerve, hogy a qubiteket pazarlóan használjuk, olyan értelemben, hogy a csoportok elkódolását „one-hot" módon végezzük. Bizonyos feladatokban segíthet, ha inkább tömörebb bináris felírásban kódolva tároljuk a számokat, ezért elkészítek egy másik QUBO formulát is ugyanerre a feladatra (3.2.3. alszakasz), bár itt sajnos egyéb segédváltozók bevezetésére is lesz szükségem. A kettő felírást ezután összehasonlítom különböző metrikák alapján (pl. legegyszerűbb metrika változók száma), ezzel becslést adva, hogy milyen gyakorlati tapasztalatokat várunk a megoldásuk során (3.2.4. alszakasz).

A 3. fejezet végén egy kicsit más témakört boncolgatok. A maximális K-vágás felírása során részproblémaként szembesültünk azzal, hogy bizonyos változók közötti logikai összefüggéseket teljesítsünk. Ez a téma messzebb mutat, így végül egy teljes alfejezetté nőtte ki magát (3.3. szakasz), melyben arra keresem a választ, hogy szokásos logikai függvényeket, vagyis logikai kapukat miként tudunk átvinni QUBO alakra. Az, hogy az egészen egyszerű kapuk viszonylag könnyen megvalósíthatók ismert tény, – bár forrást nem találtam, ahol ezt összefoglalóan meg is mutatják, – hiszen viszonylag rövid számolás után kijönnek, de ezeket nekem is sikerült levezetnem a 3.3.1. alszakaszban. Ezzel szemben például a XOR kaput nem lehet segédváltozó nélkül megvalósítani, amelyre viszont egyáltalán nem találtam korábbi eredményt, így egy teljesen saját bizonyítást közlök rá a 3.3.1.6. alalszakaszban. Hasonlóan módszerrel bizonyítom azt is, hogy nem lehet kettőnél több bemenetű OR kaput segédváltozó használata nélkül QUBO-val implementálni, egészen pontosan a 3 bemenetes OR kapuhoz legalább 2 segédváltozóra van szükségünk, mellyel viszont a feladat meg is oldható. Ezzel kapcsolatos bizonyítások a 3.3.2.1. alalszakaszban kaptak helyet.

A dolgozat 4. fejezete a gyakorlati eredményekre és tapasztalatokra fókuszál, amelyben összehasonlítom a különböző lehetőségekből adódó módszereket azok eredményessége és hatékonysága alapján. A QUBO-k optimalizálásához főként a D-Wave Ocean nevű programcsomagját használtam fel, mely több lehetőséget kínál a formulák megoldására (4.1.1. alszakasz). A klasszikus megoldók mellett lehetőség van például valódi, a D-Wave Systems által forgalmazott kvantumszámítógépeket is használni, vagy a klasszikus és kvantum eljárásokat hibrid módon ötvözni. Egy másik, szintén kereskedelmi forgalomban lévő megoldószoftver, amely alkalmas lehet a QUBO-k megoldására a Gurobi, mely teljesen a klasszikus, heurisztikus irányt képviseli (4.1.2. alszakasz). Megvizsgáltam más megoldószoftvereket is, de egyelőre ezeket még nem sikerült a célnak megfelelően felkonfigurálni (4.1.3. alszakasz).

A 4.2. szakaszban lefektetem az alapjait, hogy milyen tesztbemeneteket generáltam a megoldó szoftverek számára, majd a további részekben grafikonokkal és táblázatokkal elemeztem a tapasztalati eredményeket, melyekben természetesen a legnagyobb hangsúlyt a futásidő, és a kapott eredmény optimalitása kapta.

A munka jelentős részét tette ki számos tapasztalat gyűjtése a D-Wave-es programcsomaggal kapcsolatban, hiszen a terület újdonságából kifolyólag az elérhető dokumentációk meglehetősen limitáltnak bizonyultak.

2. fejezet

Optimalizációs technikák

Ebben a fejezetben általános optimalizációs technikákat tekintünk át. Történelmi okokból gyakran hívjuk ezeket a technikákat programozásnak, és a felírt formulát programnak, holott természetesen ennek nincs köze ahhoz, amit manapság programozás illetve program alatt értünk. Ezért is a modern szakirodalom preferálja az optimalizálás szó használatát, de még mindig szinonimaként tekintve a programozást is.

2.1. Lineáris optimalizálás

Egy lineáris optimalizálási probléma (vagy lineáris programozás) azt jelenti, hogy több keresett mennyiség (ún. változók) lineáris függvényének maximumát vagy minimumát kell meghatározni, mindeközben a rendszerre bizonyos mellékfeltételeket (korlátok) szabunk meg, lineáris egyenlőtlenségek vagy egyenletek formájában. Belátható, hogy a fenti definíciónak megfelelő minden lineáris optimalizálási feladat leírható az alábbi tömör formájában, ahol n a változók száma.

Paraméterek:

c	$\in \mathbb{R}^n$	A célfüggvény együtthatói
A	$\in \mathbb{R}^{m \times n}$	$m \times n$-es valós mátrix, a korlátokban szereplő együtthatók
b	$\in \mathbb{R}^m$	A korlátokban szereplő konstans tagok

Változók:

x	$x \in \mathbb{R}^n$	változók

Célfüggvény:

$$\min_x c^T x \tag{2.1}$$

Korlátok:

$$Ax \leq b \tag{2.2}$$

A feladat relatív hatékonyan megoldható, ugyan létezik rá polinom idejű algoritmus is, de a gyakorlatban gyakran inkább a Simplex-módszernek elnevezett eljárás valamilyen

változatát használják. A helyzet azonban egészen más, ha a változók nem vehetnek fel bármilyen valós értéket, hanem csak egészek lehetnek. Ekkor a probléma már bizonyítottan NP-nehéz.

2.2. Kvadratikus optimalizálás

A kvadratikus optimalizálás, vagy kvadratikus programozás a lineáris programozásnál egy általánosabb technika, hiszen megengedi négyzetes tagok jelenlétét a célfüggvényben, amíg a korlátok továbbra is lineárisak.[1] Ezzel az alkalmazások körét jóval kibővíti, ugyanakkor az általános feladat megoldása sokkal nehezebbé válik.

Az általános n változós, m korláttal rendelkező feladatot a következő mátrixos alakban írhatjuk le tömören.

Paraméterek:

Q	$\in \mathbb{R}^{n\times n}$	$n\times n$-es valós, (szimmetrikus) mátrix, a négyzetes tagok együtthatói
c	$\in \mathbb{R}^n$	A lineáris tagok együtthatói
A	$\in \mathbb{R}^{m\times n}$	$m\times n$-es valós mátrix, a korlátokban szereplő együtthatók
b	$\in \mathbb{R}^m$	A korlátokban szereplő konstans tagok

Változók:

x	$x \in \mathbb{R}^n$	változók

Célfüggvény:

$$\min_x \frac{1}{2} x^T Q x + c^T x \tag{2.3}$$

Korlátok:

$$Ax \leq b \tag{2.4}$$

A felírásnál Q jellemzően egy szimmetrikus mátrix, ekkor a $q_{i,j}$ jelentése, a $x_i \cdot x_j$ változószorzat együtthatója, azonban mivel a pár kétszer is meg fog jelenni, így normálni kell $\frac{1}{2}$-vel. Másik szokásos felírás, hogy Q egy felső háromszög mátrix. Ekkor ha $i \leq j$, akkor $q_{i,j}$ a megfelelő együtthatóval egyezik meg, különben nullával.

Említésre méltó megfigyelés még, hogy ha a kvadratikus polinomok helyett tetszőlegesen nagy fokszámú polinomok szerepelhetnek, melyet a QUBO sémájára PUBO-nak (Polinomial Unconstrained Binary Optimization) hívhatunk, a probléma mindig átírható klasszikus kvadratikus alakra, amely folyamatot a továbbiakban hívjuk kvadratizálásnak. Belátható, hogy bármely polinom kvadratizálható. A részletekkel most nem foglalkozunk, de a bizonyítás alapötlete, hogy egyszerűen csak új változókat kell bevezetni, úgy, hogy a fokszámok csökkenjenek. Ezzel persze mind a változók száma, mind a kifejezések hossza rendkívüli módon megnőhet. Ha csak egyszerű mohó módszerrel próbálkozunk, akkor akár exponenciálisan is. Nem ismert, hogy van-e jó stratégia a polinomok fokszámának ilyen módon történő csökkentésének, sőt ez a probléma egy jelenleg is futó kutatás alapkérdése.

[1] Egyes források úgy definiálhatják az általános feladatot, hogy a korlátokban is megengedik négyzetes tagok jelenlétét. Ez szempontunkból most kevésbé lényeges, hiszen a dolgozat jórészt a korlátmentes esetre fókuszál.

A kvadratikus optimalizálással több rokon optimalizálás fajta is aktívan kutatott tématerület. Ezeknek egyik fő irányvonala a szemidefinit programozás [19]. Ugyanakkor a problémának több speciális esete is érdekes lehet. Például tudjuk, hogy ha a Q mátrix szimmetrikus pozitív definit, akkor a probléma ekvivalens a legkisebb négyzetek megkeresésének problémájával, és az optimumot egy lineáris egyenletrendszer megoldásából kaphatjuk [21].

Ebben a dolgozatban most egy másik speciális esetet fogok elemezni, méghozzá megkötöm, hogy a változók csak és kizárólag binárisak lehetnek, és több korlát nem adható meg. Mivel a szakirodalom egyszerűen csak QUBO (Quadratic Unconstrained Binary Optimization) néven hivatkozik erre a fajta felírásra[25][3], én is így teszek a továbbiakban.

Bár a probléma rendkívül speciális, gondolhatnánk, hogy akár könnyen megoldható, hiszen csak egy polinom maximum vagy minimumhelyét keressük. Ugyanakkor, mint a későbbi fejezetekben látni fogjuk, több közismerten NP-nehéz feladat visszavezető erre a problémára, így ő maga is NP-nehéz.

A minket érdeklő általános n változós, feladat így a következő alakban írható.

Paraméterek:

Q	$\in \mathbb{R}^{n\times n}$	$n\times n$-es valós, (szimmetrikus) mátrix, a négyzetes tagok együtthatói
c	$\in \mathbb{R}^n$	A lineáris tagok együtthatói

Változók:

x	$x \in \mathbb{B}^n$	változók

Célfüggvény:

$$\min_x \frac{1}{2} x^T Q x + c^T x \tag{2.5}$$

Korlátok:

$$\emptyset \tag{2.6}$$

A bináris változók alatt szokásosan 0 vagy 1 értékeket jelölünk, így a továbbiakban is $\mathbb{B} = \{0, 1\}$. Ugyanakkor itt érdemes kitérni, hogy bizonyos alkalmazásoknál inkább a $\{-1, 1\}$ alaphalmazt tekintik. Erre elterjedt módon Ising modellként hivatkozhatunk, a fizikai spin irányultságok miatt. Bizonyos esetekben ezt könnyebb lehet elméleti síkon is kezelni, azonban a kettő között (QUBO és Ising modell) egyszerű lineáris transzformáció ad átjárást, így lényegi különbséget végül nem ad. A D-Wawe Systems gyűjtőnéven BQM-ként (Binary Quadratic Model) hivatkozik a két problémára együttesen[3].

A továbbiakban feltesszük, hogy a bináris változóink 0 vagy 1 értéket vesznek fel. Ekkor a korábban felírt általános alakot rögtön egyszerűbb alakra hozhatjuk, hiszen bármely változó megegyezik saját négyzetével. Így elég a kvadratikus tagokat felírni, mert az esetleges lineáris tagokat belevehetjük a kvadratikus tagok közé. Továbbá az $\frac{1}{2}$-del való szorzás sem tesz hozzá így már érdemben a felíráshoz, hiszen csak az optimum értékét skálázza, de az optimum helyek nem változnak. Ennek ellenére az általános felírásban ezen a helyen még benne hagytam.

Paraméterek:

$Q \quad \in \mathbb{R}^{n \times n}$ $n \times n$-es valós, (szimmetrikus) mátrix, a négyzetes tagok együtthatói

Változók:

$x \quad x \in \mathbb{B}^n$ változók

Célfüggvény:

$$\min_x \frac{1}{2} x^T Q x \tag{2.7}$$

Korlátok:

$$\emptyset \tag{2.8}$$

2.3. Korlátolt illetve korlátmentes programozás

A 2.4. szakaszban röviden látni fogjuk, hogy a kvantumszámítógépek segítségével elméletileg hatékonyan megoldhatóak a bináris kvadratikus programozási feladatok, amennyiben nem szabunk további korlátokat.

Azonban felmerül a kérdés, hogy miként tudjuk mégis használni a gyakorlatban ezt a technikát, nem szorítja meg a kezünket túlságosan az, hogy nem adhatunk meg korlátot, és egy "egyszerű" függvény szélsőértékét keressük? A válasz szerencsére nemleges, mely látszik a következő, röviden bemutatott technikából [9].

Általánosan elmondható, hogy egy optimalizációs esetben kétféle követelményt állítunk a rendszerrel szemben. Ezek közül az egyik típus a erős (hard) követelmény. Ezek olyan követelmények, melyek teljesülését mindenképpen előírjuk a rendszer számára, mert bármelyiknek a sérülése érvénytelenné tenné az eredményt. Általában ezeket a típusú követelményeket korlátként adjuk hozzá a programozási feladathoz.

A követelmények másik fajtája a gyenge (soft) típus. Ezeket is szeretnénk minél jobban teljesíteni, de nem követeljük meg az összestől egyidejűleg. Ez olykor nem is lenne lehetséges, hiszen elég valószínű egy olyan való életbeli probléma, hogy minden gyenge követelmény hozzávétele inkonzisztenssé tenné a rendszert. (Hiszen nem lehet minden tökéletes.) Ezeket a követelményeket általában igyekszünk az optimalizálandó célfüggvénybe belefogalmazni.

A két követelménytípus megfogalmazása között azonban szerencsére van átjárás. Csak annyiról van szó, hogy az erős követelményeket is bele tehetjük a célfüggvénybe, ehhez egyszerűen csak szorozzuk meg őket valamilyen jó nagy együtthatóval (úgynevezett büntetőtaggal), hogy bármelyik erős követelmény sérülése esetén a minimumtól nagyon távol essen a függvény értéke. Ezt a technikát én is felhasználom a megoldásokban, a képletekben és kódmintákban inf szimbólummal jelölve ezt az alkalmasan megválasztott nagy konstans számot.

Ugyanakkor a büntető konstans(ok) megválasztása koránt sem triviális feladat. Bár elméletileg az eredmény ugyanaz, mégis ha túl nagyra választjuk a büntetőtagot, akkor a nem optimális megoldások nagyon messze esnek az optimálistól, ha túl kicsire, akkor pedig túl közel. Egy heurisztikus elven működő optimalizáló szoftvert mind a két szélsőséges eset hátrányosan befolyásolhat. Hiszen ha a helytelen megoldások túl messze vannak az

optimumtól, akkor lehet, hogy rossz helyen keresgetünk, és nem találjuk meg a ”szűk” kis optimumot, amíg ha túl közel esnek, akkor egy nagyon rossz megoldásra is rámondhatjuk, hogy már ”elég jó”.

Ráadásul az explicit korlátokat a megoldó programok sokkal jobban tudják kezelni, mintha a célfüggvénybe fogalmaznánk bele mindent, mert ez utóbbi meglehetősen ”ködösít” a megoldó szoftver számára, hiszen nem tud lényegi különbséget tenni a különböző funkciójú változók között. Ez persze nem törvényszerű, mert a konkrét megoldó algoritmuson múlik, azonban általánosságban mégis ez várható, és ezt alátámasztja a 4. fejezetben kapott tapasztalat.

2.4. QUBO-k formalizálása

Mielőtt belevágnánk konkrét feladatok elemzésébe, érdemes áttekinteni az elméleti és gyakorlati hátterét, hogy miként érdemes QUBO feladatokat megkonstruálni, milyen egyszerű mérőszámokkal írható le egy program, mellyel megbecsülhető, hogy milyen könnyen kezelhető az optimalizálás során, és hogy egyáltalán mi a ad gyakorlati jelentőséget ennek a megközelítésnek.

Az egyik legegyszerűbb metrika, mellyel a QUBO-t jellemezhetjük, az a változók száma. Ettől szinte közvetlenül függ maga a program mérete is, hiszen nagyságrendileg legfeljebb a változószám négyzetével arányos. Így a változók száma egy nagyon triviális, ugyanakkor egy nagyon fontos mérőszám lesz, bármilyen megoldó programot is használjunk végül az optimalizálási folyamat elvégzésére.

A változók számán felül fontos, hogy megpróbáljuk jellemezni a változók közötti összefonódások bonyolultságát. A legegyszerűbb, ha ehhez definiáljuk a változók gráfját, a következőképpen.

A változók gráfjában a csúcsoknak a változók felelnek meg, és két csúcs között pontosan akkor van él, amennyiben a nekik megfelelő változók kapcsolatban vannak, azaz szorzatuk megjelenik a QUBO felírásban nem nulla együtthatóval.

Ekkor a változók gráfját különböző, a gráfelméletben megszokott alapvető metrikákkal jellemezhetjük, amely közvetlenül utal a QUBO felírás összetettségére is. Ilyen metrika, a már említett csúcsok száma (vagyis a változók száma), élek száma (hány kvadratikus tag szerepel), maximális fokszám (a legtöbb másik változóval kapcsolatban álló változó hány kvadratikus tagban szerepel), vagy az átlagos fokszám (átlagosan hány kvadratikus tagban szerepel egy változó). Ez utolsó természetesen az élek és csúcsok számából már kiszámolható. A fenti metrikákon felül érdemes lehet még nézni a klikkszámot, minimális lefogó méretét, vagy bármely más, a gráf nagyságát vagy bonyolultságát leíró értéket.

Megfigyelhető, hogy amennyiben nem QUBO, hanem PUBO alakban formalizáljuk a feladatot, vagyis kvadratikus tagok helyett megengedjük tetszőlegesen nagy fokszámú tagok jelenlétét, akkor is definiálhatjuk a változók gráfját a fent megadott módon, ám ekkor egy hipergráf keletkezik. A 2.2. szakaszban leírt állítás szerint, mivel minden polinom kvadratizálható új változók bevezetésével, most egy gráfelméleti problémát ad, amely arra keresi a választ, hogy miként érdemes új csúcsokat bevezetni, hogy az eredeti hipergráfot egyszerű gráffá konvertáljuk, miközben ne veszítsünk el bizonyos strukturális információkat.

A dolgozatnak nem célja bemutatni részletesen, hogy például a D-Wave gépek miként oldják meg a QUBO feladatokat, hiszen sokkal inkább a QUBO-k megfelelő formalizálásán van a hangsúly, úgy gondolom, hogy mégis elengedhetetlen ennek rövid áttekintése, hiszen

csak így van értelme arról beszélni, hogy milyen, és miért ezen metrikákra szeretnénk optimalizálni egy QUBO feladat formalizálása közben.

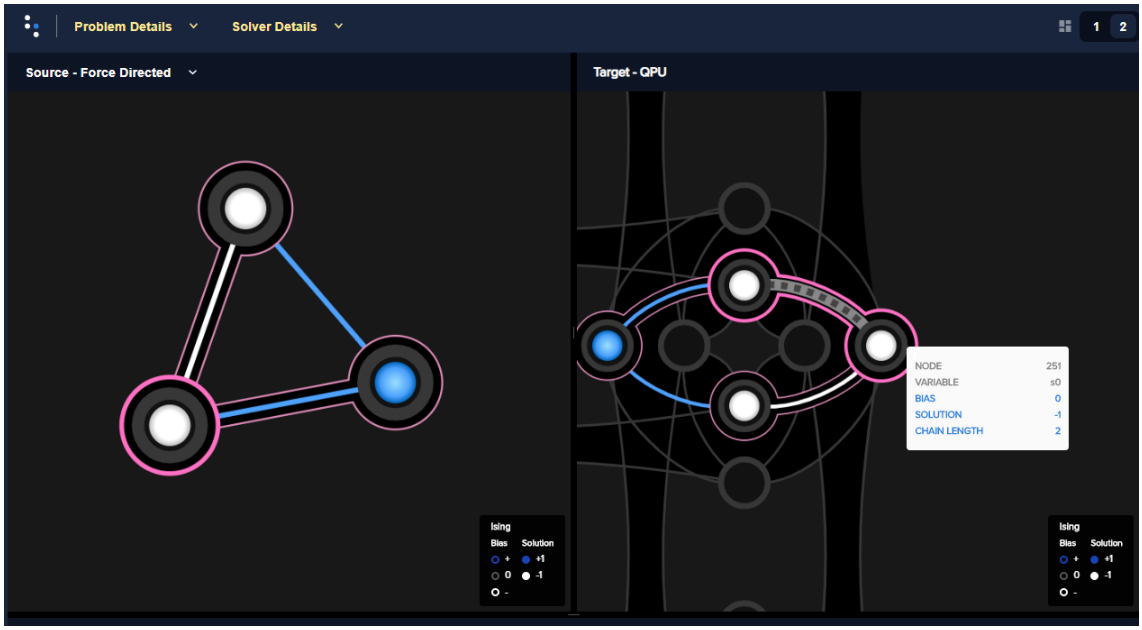
A D-Wave által nyújtott megoldás alapvető, elméleti hátterét tekintem csak át, hiszen a dolgozat motivációját is lényegében ez adja. A D-Wave Systems egy 1999-ben alapított kanadai cég, mely elsőként hozott kereskedelmi forgalomba kvantumjelenségeket használó számítógépeket. A kvantumgépeik úgynevezett lehűtéses elv alapján működnek, azaz konkrét számolás vagy heurisztika helyett fizikai folyamatok, pontosabban az energiaminimumra törekvés jelenségétől várjuk a megoldást. [22]

A számítási modell alapját egy gráfba (jellemzően az úgynevezett kiméra vagy az újabb gépeken az úgynevezett pegazus gráf) rendezett fizikai qubitek adják, melyek egymással a gráf struktúrája szerint vannak kapcsolatban. A gráf éleit és csúcsait egyaránt súlyokkal láthatjuk el, mely így egy QUBO-t definiál. Végül a megoldás várhatóan a legalacsonyabb energiaszintű állapot lesz, azaz, amikor a qubitek aszerint vesznek fel 0 vagy 1 értéket, hogy az általuk súlyokkal meghatározott kvadratikus függvény értéke vagyis a hozzájuk tartozó QUBO minimális.

Érdemes megkülönböztetni a fizikai qubiteket, melyeken a tényleges optimalizálás történik, illetve a logikai qubiteket, melyek a QUBO-ban felírt változóknak felelnek meg. A kettő között bijektív leképezés lenne a legjobb, azonban sajnos ez általában nem kivitelezhető, hiszen amíg a logikai qubitek között tetszőleges gráfot definiálhatok, (ezt hívtuk a 2.4. szakaszban a változók gráfjának), addig a fizikai qubitek struktúrája nagyon kötött. Így a logikai qubiteket ügyesen rá kell képezni a fizikai hardware-re, amely folyamatot hívjuk beágyazásnak. Például a kiméra gráfban a maximális fokszám 6, és lévén páros gráf, már egy háromszöget (K_3-at) sem lehet beágyazni új változó bevezetése nélkül. Ekkor a módszer jellemzően az, hogy a fizikai qubitek gráfjában láncokat alakítunk ki, amely fizikai qubiteket köt össze, és ugyanannak a logikai qubitnek, azaz változónak felelnek meg. Szintén az erősen korlátos fokszámból látható, hogy olyan csúcsok beágyazása, melynek nagy a fokszáma gondokat okoz. A módszer itt is láncok létrehozása, azonban ezek még több fizikai qubitet emésztenek fel. Egy teljes gráf beágyazása a fenti okok miatt pedig kimondottan körülményes, így megoldó hardver szempontjából mindenképpen az volna az előnyös, ha a gráf klikkszáma kicsi lenne.

Mint ahogy a 2.1. ábra is mutatja, a K_3 beágyazása csak úgy lehetséges, ha igénybe veszünk legalább még egy 4. fizikai qubitet, és valamelyik logikai qubit reprezentálásához kettő fizikai qubitet használunk fel, melyek értékét egyenlővé tesszük.

Bár a QUBO problémák beágyazását fizikai hardware-re ennél mélyebben most nem vizsgáljuk, fontos észrevétel, hogy a dolgozat szempontjából van jelentősége a konstruált QUBO formulák elemzésének, a korábbi szempontok alapján.



2.1. ábra. K_3 beágyazása

3. fejezet

Problémák formalizálása

Ebben a fejezetben különböző vágási problémákat tekintek, különös tekintettel a maximális, és a maximális-K-vágásra, illetve megvizsgálom, hogy miként fogalmazhatók meg valamilyen optimalizációs programként. Az egyik ötlet megvalósításához szükség van logikai kifejezések QUBO-ként való felírására is, ezért erre még egy teljes külön alfejezetet szánok a fejezet végén.

Ha máshogy nem jelzem, a továbbiak $G = (V, E)$ jelöljön egy gráfot, ahol szokásos módon $n = |V|$ és $m = |E|$. A gráf élei legyenek súlyozva nem negatív valós számokkal, ahol az $uv \in E$ súlya w_{uv}.

Jelölje továbbá $[k]$ az 1-től k-ig tartó egészek halmazát. P pedig egy elegendően nagy számkonstanst jelöljön, amelyet büntetőszorzóként fogok használni.

3.1. Maximális vágás

A maximálás vágás problémája arra keresi a választ, hogy (élsúlyozott) G gráfban miként lehet szétosztani a csúcsokat két halmazra, aszerint, hogy azon élek összsúlya, amelyek végpontjai különböző halmazokba kerültek, maximális legyen. A problémának több változata is ismert és kutatott, ennek a dolgozatnak a keretében a pozitív valós élsúlyokkal ellátott esetet vizsgáljuk, amely kellően általános. Ennek nagyon speciális esete, ha minden élsúly 1 értékű, hiszen ekkor elég az elvágott élek számára maximalizálni. Ezt a problémát ismerhetjük még maximális páros részgráf néven is, amely, bár az általános problémánál sokkal speciálisabb, még mindig NP-nehéz annak az eldöntése is, hogy adott gráfban létezik-e, e élű páros részgráf.

3.1.1. Kombinatorikus megoldás

Polinom idejű algoritmust sajnos nem várhatunk, mivel a feladat egy a leghíresebb NP-nehéz problémák közül. Természetesen exponenciális futásidejű algoritmus adható, hiszen a gráf csúcsait összesen 2^{n-1} féleképpen lehet szétválasztani két csoportra, egy vágás súlyának kiszámítása, pedig lineáris idejű a bemenet méretének függvényében. Ezzel így adódok egy $O(m2^n)$ lépésszámú algoritmus.

Ha gyors algoritmust szeretnénk, akkor a pontos optimum helyett egy közel optimális megoldást célszerű keresni. Szerencsére ilyen létezik is. 2-approximációt viszonylag gyors és egyszerű algoritmus segítségével lehet már adni, például úgy, hogy a csúcsokat egyesével kiválasztva döntünk, hogy melyik csoportba osszuk be őket, aszerint, hogy az

aktuálisan kiválasztott csúcsból kiinduló élek összsúlyának nagyobb része menjen a másik csoportba. Ezzel elértük, hogy minden bekerülő csúcsnál legalább annyi élsúly lesz a két csoport között, mint a csoportokon belül. Mivel tudjuk, hogy az optimum legfeljebb az élek összsúlya lehet, és ez a megoldás legalább az összsúly felét kiválasztja, ezért ez egy (legrosszabb esetben is) 2-approximáció[1].

Viszont az is korábbról ismert tény, hogy (amennyiben P $\neq$ NP) nem létezik rá polinomiális approximációs séma. A legjobb eddig ismert közelítés approximációs faktora körülbelül 1.139, amely egyébként a dolgozatom témájához is közel eső szemidefinit programozással lett először elérhető, sőt amennyiben a Unique Games sejtés igaz, bizonyított, hogy ennél jobb nem is létezik [10, 23].

A problémának ismert felírása egészértékű- illetve kvadratikus programozással is, melyeket a következő alfejezetekben vizsgálok.

3.1.2. Lineáris programozással

Egész értékű lineáris programozási feladat felírható rá, minden $\{i, j\} \in E(G)$ élre (pontosabban később látjuk majd, hogy minden csúcspárra) vezetünk be egy d_{ij} bináris változót, melynek értéke akkor 1, ha az él benne van a vágásban. Ekkor a célfüggvényben világos, hogy azon élekre adjuk össze a súlyokat, melyek benne vannak a vágásban, és ezt szeretnénk maximalizálni.

$$\max_{d} \left\{ \sum_{\{i,j\} \in E(G)} w_{ij}\, d_{ij} \right\} \tag{3.1}$$

További korlátokat is fel kell vegyünk, különben a maximum akkor áll elő, ha a változók azonosan 1 értéket vesznek fel. Amit ki szeretnénk fejezni további korlátokkal, hogy bármely háromszögből pontosan nulla vagy kettő él lehet benne a vágásban. Ez nyilvánvalóan egy szükséges feltétele egy helyes vágásnak, viszont az elégségesség is következik, ha a gráf teljes. Ehhez viszont nem létező éleket kell hozzávegyünk a gráfhoz (és így a változók száma is $\binom{n}{2}$ -re növekszik.) Az élsúlyokat ekkor persze úgy kell megválasszuk, hogy ne zavarják a maximális vágást, (például ha az élsúlyok nem negatívak, akkor az azonos 0 választás megfelelő) de általában még az sem probléma, ha nem rendelünk az új élekhez explicit súlyokat, hiszen elég ha a célfüggvényben csak az eredeti élekre összegezzük a súlyokat.

$$d_{ij} \leq d_{ik} + d_{kj} \tag{3.2}$$

$$d_{ij} + d_{ik} + d_{kj} \leq 2 \tag{3.3}$$

Érdemes pár szót ejteni a relaxált, azaz az egészértékűségi kényszerek elhagyása után kapott feladatról. Ekkor egy lineáris programot kapunk, amely hatékonyabban megoldható, viszont a változók értékeit a végén még kerekíteni kell, hogy egészértékű megoldást kapjunk. Ezzel a módszerrel ismét egy 2-közelítő algoritmust kapunk. Ez persze egyáltalán nem biztos, hogy gyakorlatban is hasznos, hiszen ahogyan korábban említettem, egyszerű mohó algoritmussal is elérhető a 2-approximáció [6, 17]. Ugyanakkor, ahogyan

[1] Itt nem bizonyítjuk, de valójában létezik rá éles példa, így az approximációs faktora nem jobb 2-nél.

azt is már kiemeltem, a legjobb ismert közelítést mégis pont egy ehhez nagyon hasonló módszer, a szemidefinit programozás adja [10].

3.1.3. QUBO

QUBO-val a feladat természetesen módon felírható. Az élekre (csúcspárokra) felírt változók helyett, azonban csak a csúcsokra definiáljuk a bináris változóinkat, és ezek kvadratikus polinomjaikból állnak majd össze az élekre kikövetkeztetett változók. Tehát az ötlet, hogy minden csúcsot jelöljünk egy x_i bináris változóval, amelynek értéke mutatja, hogy az i. csúcs melyik csoportba tartozik. A célfüggvényt úgy konstruáljuk, hogy egy él súlyát csak akkor számoljuk bele az összegbe, ha az a két csoport közt fut, és ezt az összeget szeretnénk maximalizálni.

$$\max_x \left\{ \sum_{\{i,j\}\in E(G)} w_{ij} \left(x_i - x_j\right)^2 \right\} \tag{3.4}$$

Ugyanez a négyzetes részt kibontva a következőképpen alakul. Ne felejtsük, hogy mivel változóink binárisak, ezért bármely változó azonos saját maga négyzetével. ($x \in \mathbb{B} \Rightarrow x = x^2$)

$$\max_x \left\{ \sum_{\{i,j\}\in E(G)} w_{ij} \left(x_i + x_j - 2x_i x_j\right) \right\} \tag{3.5}$$

3.2. Maximális K-vágás

A maximális K-vágás, vagy max-K-vágás problémája arra keresi a választ, hogy miként tudjuk a gráf csúcsait K részre szétosztani, aszerint hogy azon élek összsúlya, melyek a csoportok között mennek, maximális legyen. Ez a probléma nyilvánvalóan egy általánosítása a sima maximális vágásének, hiszen a $K = 2$ speciális esetben pontosan egyenértékű a két feladat. Ezáltal az is világos, hogy mivel már a $K = 2$ speciális eset is NP-nehéz, ezért az általános eset is legalább ennyire bonyolult. A probléma ismert még más neveken is, egy másik elterjedt megnevezés, a minimum K-partíció, mely azt fejezi ki, hogy a részeken belül futó élek összsúlya legyen minimális. Ezt természetesen pontosan akkor fog előfordulni, amikor a partíciók között lévő élek súlya maximális.

A feladat felírható egészértékű lineáris programmal, mely bemutatása után két különböző QUBO felírást mutatok be rá, melyek különböző megközelítés mentén készültek el.

3.2.1. Lineáris programozással

Az ötlet itt is hasonlít a sima maximális vágáséhoz. Az élekre ezúttal is bevezetünk egy bináris változót, mely azt jelzi, hogy az él két különböző csoport között fut. Vagyis $y_{uv} = 1$, akkor és cs ak akkor, ha u, v él csoportok között fut. Ezen felül viszont azonosítanunk kell minden egyes csúcsot, hogy ő melyik csoportba fog kerülni. Ezeket szintén binárisan

kódoljuk nK db változóban, ahol x_{vi} akkor és csak akkor 1, ha a v. csúcs az i. halmazba kerül.

A célfüggvény egyértelműen elkészíthető, egyszerűen azon élekre kell összeszámolnunk a súlyokat, melyek két különböző csoport között futnak.

A korlátoknál az elsőként felveendő legfontosabb dolog, hogy bármely v pontosan egy csoportba tartozhasson, ezt a 3.7 szummával adhatjuk meg.

A további korlátok (3.8 - 3.10) pedig azt biztosítják, hogy a változók konzisztensek, azaz y_{uv} valóban csak akkor legyen 1, amikor u és v különböző csoportba kerülnek.

$$\max_{y} \sum_{\{u,v\}\in E} w_{uv}\, y_{uv} \tag{3.6}$$

$$\sum_{i\in[K]} x_{vi} = 1, \qquad v \in V, \tag{3.7}$$

$$x_{ui} - x_{vi} \leq y_{uv}, \qquad \{u,v\} \in E,\ i \in [K], \tag{3.8}$$

$$x_{vi} - x_{ui} \leq y_{uv}, \qquad \{u,v\} \in E,\ i \in [K], \tag{3.9}$$

$$x_{ui} + x_{vi} + y_{uv} \leq 2, \qquad \{u,v\} \in E,\ i \in [K], \tag{3.10}$$

$$x_{vi} \in \{0,1\}, \qquad v \in V,\ i \in [K], \tag{3.11}$$

$$y_{uv} \in \{0,1\}, \qquad \{u,v\} \in E, \tag{3.12}$$

Egy ehhez nagyon hasonló (több korlátot igénylő) probléma felírása megtalálható az interneten, részletes indoklással [15], így a dolgozatban nem részletezem ennek a felírásában ennél mélyebben.

3.2.2. QUBO-val (One-hot encoding)

A maximális vágásnál a QUBO forma segítségével jelentősen le tudtuk csökkenteni a változók számát, hiszen nem kellett az élekre bevezetnünk változókat. Ezt a trükköt próbáljuk meg itt is. Vagyis a lineáris programhoz hasonlóan, a x_{vi} változókat megtartjuk, az a tény pedig, hogy egy uv él kettő különböző csoport között fut, egyértelműen következik, ha $x_{vi}\, x_{uj} = 1$ valamely $i \neq j$ és $u \neq v$-re.

Ez alapján meg is konstruálhatjuk és fel is írhatjuk a maximalizálandó célunkat. (Egyelőre megengedjük további korlátok létezését is.) Világos, hogy minden, csoportok között futó élet pontosan egyszer fogunk megszámolni, mivel minden csúcs pontosan egy csoportnak lehet az eleme.

$$\max_x \left\{ \sum_{\{u,v\}\in E} \sum_{\{i,j\}\in[K]} w_{uv}\,(x_{vi}\,x_{uj}) \right\} \tag{3.13}$$

$$\sum_{i\in[K]} x_{vi} = 1, v \in V \tag{3.14}$$

A korlátokat viszont szeretnénk kiiktatni a felírásból, így azokat valamilyen büntető tagként kell hozzáadni a célfüggvényhez. Amit büntetni szeretnénk, azok azok az esetek, amikor egy csúcs két különböző csoportba is bekerül. Amennyiben azt szeretnénk, hogy K darab változóból pontosan k darab legyen egyes, erre általános technika, hogy büntető tagként a $(\sum_{i\in[K]} x_i - k)^2$ kifejezést kell hozzávennünk a célfüggvényhez, hiszen ennek értéke akkor és csak akkor 0, ha az x_i változók közül k darab 1-es, és minden más 0. Ellenkező esetben a kifejezés értéke valamilyen pozitív szám lesz.

Ezt a módszert is alkalmazhatnánk $k = 1$ helyettesítéssel, azonban most egy másik, bár lényegét tekintve hasonló megoldást mutatok. Hiszen azt is mondhatnánk, hogy egy v csúcs pontosan akkor kerül bele két különböző csoportba, ha $x_{vi}\,x_{vj}$ szorzat értéke 1 valamilyen $i \neq j$-re. Így ezeket az eseteket büntetve a célfüggvényünk kiegészül még a 3.15 taggal.

$$\sum_{v\in V} \sum_{\substack{i,j\in[K] \\ i\neq j}} (-P)\,x_{vi}\,x_{vj} \tag{3.15}$$

Fontos megjegyezni, hogy a korábban vázolt általános módszerhez képest, elég a nagyobb esetet büntetni, vagyis amikor legalább 2 változó értéke is 1-es. Hiszen az optimumnál impliciten teljesül, hogy az egy csúcshoz tartozó bináris változók közül legalább az egyik 1 értéket vesz fel. Mivel ha lenne egy változóbehelyettesítés, melynél egy csúcs nem kerülne bele egyetlen csoportba sem, a belőle kimenő élek nem kerülnének bele a vágásba. Így bármelyik csoportba is kerüljön bele, a célfüggvény értéke biztosan nőni fog.

Így a végső QUBO forma:

$$\max_x \left\{ \sum_{\{u,v\}\in E} \sum_{\{i,j\}\in[K]} w_{uv}(x_{vi}x_{uj}) + \sum_{v\in V} \sum_{\substack{i,j\in[K] \\ i\neq j}} (-P)\,x_{vi}\,x_{vj} \right\} \tag{3.16}$$

3.2.3. QUBO-val (Binary encoding)

Más megoldást is megpróbáltunk keresni, amellyel a QUBO-t tovább egyszerűsíthetjük, a változók számát csökkenthetjük. Erre egy lehetséges irányzat, hogy azt, hogy a csúcs mely csoportba tartozik nem úgy kódoljuk, hogy minden csoport-csúcs párhoz egy bináris változót rendelünk, és a csúcs csoportját "one-hot" módon kódoljuk, hanem kifinomultabb módon, a csoport sorszámát binárisan kódoljuk le. Ennek előnye lehet, hogy így nem szükséges $n \cdot K$ darab bit, hanem elég volna $n \cdot \log K$, ráadásul a problémás korlátot,

amely azért szükséges, hogy biztosítsuk, hogy egy csúcs pontosan egy csoportba kerüljön, ezt impliciten teljesítené. Motivációt ad egy friss kutatás, ahol ezt a módszert sikeresen alkalmazták kvantumgép esetében, bár ezt natív módon tették meg [8]. A mi kérdésünk, hogy vajon ezt meg lehet-e csinálni eggyel magasabb absztrakciós szinten, csupán QUBO felírást használva.

A továbbiakban az egyszerű szemléltetés kedvéért feltesszük, hogy a készítendő csoportok száma valamilyen kettő hatvány, vagyis $K = 2^m$, így $\log K$ egész. Ezzel mindig feltehető, hogy minden bit szabadon lehet 0 és 1, mert nem lesznek "tiltott" csoportok. Később kitérek arra az esetre, ha ez nem így volna.

A megoldáshoz elsősorban minden csúcshoz definiálunk kell az ő csoportját tartalmazó változókat. Az u. csúcs csoportját jelölje x_u, illetve ennek bináris felírását tekintve, az x_u i. bitjét jelölje x_{ui}.

Amire szükségünk van, az egy bináris függvény, amely megmondja, hogy két csúcs különböző csoportba esik-e. Ha igen, akkor természetesen a köztük lévő él súlya hozzá kell adódjon a maximalizálandó célfüggvényhez. Ellenkező esetben viszont figyelmen kívül kell hagyni ezt a súlyt.

$$D_{uv} = \text{isDifferent}(x_u, x_v) \tag{3.17}$$

$$\max_x \left\{ \sum_{\{u,v\} \in E} D_{uv}\, w_{uv} \right\} \tag{3.18}$$

A D_{uv} meghatározása azonban problémákat vet fel. Első körben megpróbálhatjuk, hogy a korábban látott formulák szerint pontosan meghatározható egyes bitek különbsége, hiszen csak a különbségük négyzetét kell venni. A négyzetek, mint tagok összeadása természetesen adja magát, azonban ekkor egy olyan számot kapunk, amely pontosan meghatározza, hogy hány bitben különbözik a két szám, vagyis a Hamming-távolságot. Nekünk elegendő, ha bármely helyiértéken van különbség, akkor szeretnénk végeredményben 1-et kapni. Erre a módszer, hogy minden négyzetkülönbséget binárisan negálunk, azaz kivonjuk az értékét 1-ből. Az így kapott kifejezéseket összeszorozzuk, így ha bárhol volt különbség a szorzat valamely tényezője 0, így természetesen maga a szorzat is 0. Amennyiben sehol nincs különbség, a szorzat minden tagja 1, így a szorzat is. Mivel a D_{uv} definíciója szerint akkor 1, ha a két kettes számrendszerben felírt szám különbözik, ezért még a végén egy plusz negálást kell elvégezni.

$$D_{uv} = 1 - \left(1 - (x_{u1} - x_{v1})^2\right) \cdot \left(1 - (x_{u2} - x_{v2})^2\right) \cdot \ldots \cdot \left(1 - (x_{u \log K} - x_{v \log K})^2\right) \tag{3.19}$$

Az így kapott formula azonban nem kvadratikus, hanem jóval nagyobb, akár $2 \cdot \log K$ is lehet a változók összes kitevője egy tagban. Sőt, ha végiggondoljuk majdnem minden lehetséges változókombináció megjelenik, ha elvégezzük a szorzásokat. (Csak annyi a szabály, hogy ha x_{ui} megjelenik, akkor az vagy a négyzeten szerepel, vagy meg van szorozva x_{vi}-vel.)

Kellő számú változó bevezetésével, és azok behelyettesítésével természetesen a formula kvadratikus alakra hozható, ahogyan azt már láttuk a 2.4. szakaszban, de ez a konkrét

esetben nem könnyű feladat. Bár mechanikusan elvégezhető, de nem ismert jó módszer erre a feladatra, amely az újonnan bevezetett változókra és a keletkező kvadratikus tagokra minimalizálna. Így a továbbiakban, bár az eredményként lényegében az előzővel azonos, egy másik szemléletet fogok alkalmazni, amely természetesebben adja, hogy hol és mennyi változó bevezetése szükséges.

Annak jelölésére, hogy az u és v csúcsok különböző csoportokba kerülnek-e, használjuk továbbra is a D_{uv} bináris változót. Két egyedi bit összehasonlítására pedig a d_{uvi} változót definiáljuk, melynek jelentése, hogy az x_u és x_v számok i. bitje különböző. d_{uvi} így a XOR műveletének eredménye x_{ui} és x_{vi}-re alkalmazva.

Mivel két szám pontosan akkor különböző, ha legalább egy helyiértékükön eltérnek, ezért a D_{uv}-t érdemes úgy előállítani, hogy minden bit különbözőségét OR kapcsolatba állítjuk.

$$D_{uv} = \bigvee_{i \in [\log K]} d_{uvi} \tag{3.20}$$

$$d_{uvi} = x_{ui} \oplus x_{vi} \tag{3.21}$$

Ha feltesszük, hogy a logikai kapuk által definiált relációk megvalósíthatók további változók bevezetése nélkül, akkor kiszámolható egy alsó becslés arra, hogy összesen hány változóra van szükségünk. A csoportok elkódolásához kell $n \log K$ db. Minden csúcspár összehasonlításához kell $\log K + 1$ darab, hiszen minden bitpárhoz kell egy, és végül maga a D_{uv}. Így ez összesen $\binom{n}{2}(\log K + 1) + n \log K$ változó.

Valamennyit spórolhatunk, ha a gráf ritka, és ezáltal csak azokat a csúcspárokat hasonlítjuk, melyek között fut él. Ekkor ha az élek száma m, akkor $m(\log K + 1) + n \log K$ változó szükségeltetik.

3.2.4. A két felírás összehasonlítása

A bináris elkódolás módszerével kapott változószám első közelítésre mindenképpen rosszabb, mint a másik felírásban kapott nK, hiszen n legalább akkora, de tipikusan jóval nagyobb, mint K, így az $\binom{n}{2}$ tényező (vagy az m) várhatóan önmagában is nagyobb lesz, mint nK.

A valóságban ráadásul sajnos még ennél is több segédváltozó szükségeltetik. Mint ahogyan a 3.3.1.6. alalszakaszban látni fogjuk a XOR kapu megvalósításához egy további segédváltozót igénybe kell venni, és $\log K$ darab bitre alkalmazott OR művelet sem megy egy lépésben, amely kiderül a 3.3.2.1. alalszakaszban. Ez utóbbinál például egymás után láncolt, vagy bináris fában felépített OR kapukkal érhető el a kívánt eredmény, amely további $\log K - 1$ változót igényel, bár ebben már benne van maga a D_{uv} is. Ez így minden csúcspárnál $3 \log K - 1$ változót jelent, vagyis összesen $\binom{n}{2}(3 \log K - 1) + n \log K$ változóra van szükség.

Térjünk át a 2.4. szakaszban definiált változók gráfjára, és vizsgáljuk a továbbiakban azt. Itt a maximum fokszám szempontjából is érdemes lehet elemezni az eredményt, azaz egy változó legfeljebb hány másik változóval van kapcsolatban. A segédváltozók fokszáma többnyire kicsi, hiszen csak néhány másikkal állnak kapcsolatban, viszont a csoportot

elkódoló változókra ez már nem igaz, hiszen egy x_{ui} változó kapcsolatban lesz a d_{uvi} és x_{vi} változókkal, minden $v \neq u$-ra. Így ezen változók fokszáma az eredeti gráf csúcsszámának közel kétszeresével egyenlő. Sőt, valójában ennél még rosszabb a helyzet, mert ahogyan korábban említettem, a XOR kapu implementálásánál bejön még egy segédváltozó, amely miatt a csúcsszám majdnem háromszorosa lesz a maximum fokszám.

Az eredeti, 3.2.2. alszakaszban látott felírásnál, legrosszabb esetben, ha teljes a gráfunk, akkor minden változó minden másik változóval kapcsolatba kerül, így ott a változók gráfja, egy nK csúcsú teljes gráf, tehát a maximum fokszám $nK-1$. Ez bár rosszabb, mint a 3.2.3. alszakaszban bináris felírással adott megoldás, ugyanakkor előbbinél egy nagy teljes gráfról beszélhetünk, amíg utóbbinál, csupán n méretű teljes részgráfok jelennek meg, hiszen minden x_ui kapcsolatban van minden x_{vi}-vel. Ezekből az egymástól független klikkekből $\log K$ darab van, hiszen minden helyiértékre 1 darab ilyen klikk található. Ezek a klikkek viszont a segédváltozók segítségével valami bonyolultabb struktúrán keresztül közvetetten kapcsolódnak egymáshoz, hiszen a D_uv változók összekapcsolják az u illetve v csúcsokhoz tartozó változókat.

Ahogyan a 2.4. szakaszban röviden bemutattam, egy teljes gráf beágyazásához nagyon sok qubitre van szükségünk, így az a szerencsés, ha a gráf klikkszáma minél kisebb. Bár a klikkszám illetve a maximális fokszám az új felírásunkban valóban kisebb, a rengeteg újonnan bejövő változó, és bonyolult struktúra miatt nem sejtjük, hogy ezzel valóban egyszerűbb lett a probléma gyakorlati szempontból, különösképpen, hogy a klikkszámnál a különbség csak nagy K értékekre lenne megfigyelhető.

A két felírás különbségeit a változók gráfján a 3.1. táblázat foglalja össze.

	one-hot encoded	binary encoded
Csúcsok száma	nK	$\binom{n}{2}(\log K + 1) + n \log K$
Max-fokszám	$nK - 1$	$3(n-1)$
Max-klikk	nK	n

3.1. táblázat. Max-K-vágás QUBO alakjai

3.3. Logikai kapuk megvalósítása

A 3.2.3. alszakaszban részproblémaként került elő, hogy bizonyos logikai függvényeket szeretnénk megvalósítani QUBO segítségével. Ez viszont egy jóval általánosabb kérdéskörre vezet, mint ahogyan a konkrét példában alkalmaztuk, így a probléma motivációját nem csak ez a feladat adja. Nagyon egyszerű példa, hogy ha fel tudunk írni tetszőleges logikai kifejezést QUBO segítségével, akkor az választ adhat például a kielégíthetőség kérdésére, amely egy közismert NP-teljes probléma[20].

Ennek tudatában, a következő részben szeretném áttekinteni a leggyakrabban előforduló logikai kapukat, és hogy azok miként valósíthatók meg QUBO-val. Valójában logikai kapu alatt jelen esetben csupán azt értjük, hogy változók közötti valamilyen reláció mindig teljesüljön. Tehát pl. ha az AND kapu bemenete x és y, kimenete pedig z, akkor elvárható, hogy az $x \wedge y = z$ logikai kifejezés mindig teljesüljön. Így általános elv, hogy a logikai kapuk megvalósításánál igazából arra törekszünk, hogy a változók helyes konfigurációi mellett egy adott értékű, praktikusan 0-t adó kifejezést konstruáljunk, amíg minden más, nem megengedett változóbehelyettesítés mellett a kifejezés értéke ennél nagyobb legyen. Ekkor minimalizálás esetén biztosak lehetünk, hogy a minimumhely ott van, ahol minden logikai kapunak megfelelő változóbehelyettesítést kapunk. A kifejezést természetes

megszorozhatjuk egy kellően nagyra választott konstanssal, így biztosak lehetünk, hogy a büntetés értéke elég nagy lesz.

Ebből következik az is, hogy bármely logikai függvény QUBO alakban történő felírásának helyességérőlkönnyen meggyőződhetünk, ha elkészítjük annak igazságtábláját, minden lehetséges konfigurációra kiszámítjuk a kifejezés értékét, és ellenőrizzük, hogy csak az igaz állítások mellett szerepel 0, minden más esetben pedig egy pozitív szám lesz a kifejezés értéke a behelyettesítés után.

Nyilvánvalóan, ha önmagának ellentmondó kombinációs hálózatot képzünk le így, akkor nem lehet majd minden büntetésként megfogalmazott korlátot teljesíteni, és az optimum sem lesz ennek megfelelő. Tehát a formula ilyen módon alkalmas lehet akár a kielégíthetőség eldöntésére is.

Végül még egyszer hangsúlyozom, itt lényegében arról van szó, hogy változók bizonyos konfigurációt szeretnénk csak megengedni, ezért a logikai kapukra is jobb így tekinteni, és nem mint olyan (áramköri/logikai) elemek vagy függvények, melyeknek adott bemenete(i) és arra adott kimenete(i) van.

3.3.1. Elemi kapuk

3.3.1.1. „Azonos kapu” ($x = y$)

Első ilyen „kapu” a legtriviálisabb mind közül, az egyenlőség vagy azonosság. (Nem mint eldöntendő logikai kifejezés, hanem mint értékadás.) Igazából nem is kapu, hanem a fizikai interpretációja a vezeték vagy a buffer volna. Azt már láttuk korábban, hogy két bináris változóról hogyan lehet eldönteni, hogy ők különbözőek-e, ehhez csak a különbségük négyzetösszege kell. Természetesen itt is elég ennyi, ekkor a megfelelő büntetőtagot ezzel a kifejezéssel szorozhatjuk. Tehát amennyiben azt szeretnénk elérni, hogy x és y változóink azonos értékkel bírjanak, a következő kifejezést kell használnunk a célfüggvényben. Ne felejtsük, hogy mivel változóink binárisak, ezért bármely változó azonos saját maga négyzetével. ($x \in \mathbb{B} \Rightarrow x = x^2$)

$$(x - y)^2 = -2xy + x + y \tag{3.22}$$

x	y	büntetés
0	0	0
0	1	1
1	0	1
1	1	0

3.2. táblázat. „Azonos kapu”

3.3.1.2. NOT kapu ($x = \neg y$)

A következő hasonlóan egyszerű eset a nem-egyenlőség esete. Ha azt szeretnénk leírni, hogy az x bináris változó nem egyenlő az y értékével, ez könnyen megkapható az előző esetből, hiszen akkor 1 értéket kaptunk a rossz, és 0-t a jó esetekben, így csak ezt kell logikailag negálnunk. Vagyis a konstans 1-ből kivonni az így kapott kifejezésünket.

$$1 - (x - y)^2 = 1 + 2xy - x - y \tag{3.23}$$

x	y	büntetés
0	0	1
0	1	0
1	0	0
1	1	1

3.3. táblázat. NOT kapu

3.3.1.3. AND kapu ($x \wedge y = z$)

Mivel két változóra felírható összefüggéseket az előző két esettel lényegében kimerítettük, hiszen két bináris változó vagy azonos, vagy különböző értéket vesz fel, ezért nézzük meg néhányat a klasszikus értelemben vett logikai kapuk közül, melyeknél általában, (de minden esetben legalább) 3 változó szerepel.

Első ilyen kifejezésünk az AND kaput fogja leírni, vagyis a $z = x \wedge y$ kifejezést. Ellenőrizhető, hogy, amennyiben az x és y változók közül legfeljebb egy értéke 1, addig a kifejezés $3z$, illet z értékével lesz egyenlő. A kifejezés értéke így természetesen akkor lesz minimális, ha $z = 0$. Amennyiben viszont $x = y = 1$, akkor a kifejezés x és y behelyettesítése után $1 - z$ lesz, mely a minimumhelyét $z = 1$ esetben veszi fel.

Fontos megfigyelni azt a korábban kiemelt tényt, hogy itt is lényeges, hogy az érvényes konfigurációknál mind azonos legyen az optimum értéke. Gondolhatnánk, hogy az xy szorzat felesleges a kifejezésben, hiszen lefixált x illetve y értékeknél is igaz a fenti állítás. Azonban ekkor a minimumhely $x = y = z = 1$ esetében lenne a legkisebb, tehát a logikai kapu jobban preferálná, ha a bemenetein is 1-esek jelennének meg, amely visszahatna az x és y változók értékeire. Értelemszerűen ezt nem szeretnénk, hiszen ez olyan, mintha egy "bias" lenne beiktatva a logikai kapuba.

$$xy - 2(x + y)z + 3z \tag{3.24}$$

x	y	z	büntetés
0	0	0	0
0	0	1	3
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

3.4. táblázat. AND kapu

3.3.1.4. OR kapu ($x \vee y = z$)

Az AND kapu mellett a másik alapvető logikai kapu az OR kapu. Ez persze a De Morgan azonosság szerint felírható csupán az AND és a NOT kapu használatával, azonban ez plusz változókat hozna be. Az OR kaput direkt módon is fel tudjuk írni a következő kifejezéssel:

$$\begin{aligned}(x+y)+z-2(x+y)z+xy=\\ x+y+z-2xz-2yz+xy\end{aligned} \tag{3.25}$$

x	y	z	büntetés
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	3
1	1	1	0

3.5. táblázat. OR kapu

3.3.1.5. NOR kapu ($x \vee y = \neg z$)

A NOR kaput megkaphatjuk az OR kapu kimeneti változójának negálásával, azonban ez plusz egy változót kellene jelentsen. Ehelyett direkt módon próbáljuk meg felírni a kifejezést. Használhatnánk a NOT kapunál látott logikai negálást, azonban amíg ott az kifejezés értékkészlete is bináris volt, ezúttal a kifejezés akár a 3-at is felveheti értékként. Ennek ellenére a NOR kapu kifejezése majdnem valóban az OR logikai negáltja lesz, csupán az xy szorzat együtthatója változatlan marad.

$$\begin{aligned}1-(x+y)-z+2(x+y)z+xy=\\ 1-x-y-z+2xz+2yz+xy\end{aligned} \tag{3.26}$$

x	y	z	büntetés
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	3

3.6. táblázat. NOR kapu

3.3.1.6. XOR kapu ($x \oplus y = z$)

Ahogyan korábban ez előtérbe került a "két szám bináris felírása különböző-e" kérdéskörben a 3.2.3. alszakaszban, nagyon hasznos lenne egy KIZÁRÓ VAGY avagy XOR kapu is. Közismert, hogy csupán az AND és a NOT segítségével is felírható bármely Boole-függvény, így ezek tetszőleges kombinálásával természetesen a XOR is. Ha megpróbáljuk direkt módon, csupán 3 változó segítségével felírni, problémába ütközünk, mivel ez nem lehetséges. Ezt a tényt alább bizonyítom, mely egyben megad egy általános módszert is hasonló problémák felírására.

Indirekt módon tegyük fel, hogy lehet megfelelő kifejezést alkotni. Legyen ez a kifejezés $A_x x + A_y y + A_z z + A_{xy} xy + A_{xz} xz + A_{yz} yz + C$. Tudjuk, hogy az érvényes változókonfigurációk behelyettesítésére 0-t, érvénytelen esetben pedig egy 0-nál nagyobb pozitív számot kell kapjunk. (Tekinthetnénk 0 helyett egy másik fix számot, de a C taghoz hozzáadva az eltolást ezt korrigálhatjuk, illetve ha minimalizálni szeretnénk, akkor elég a kifejezést -1-gyel beszorozni.)

Minden változókonfiguráció behelyettesítésével így kapunk egy egyenletet vagy egyenlőtlenséget, melyet az együtthatóknak és a konstans tagnak teljesítenie kell. Az igazságtáblázat 8 sorából keletkező egyenleteket és egyenlőtlenségeket az alábbi táblázat foglalja össze.

x	y	z	büntetés	egyenlet/egyenlőtlenség
0	0	0	0	$C = 0$
0	0	1	> 0	$A_z + C > 0$
0	1	0	> 0	$A_y + C > 0$
0	1	1	0	$A_y + A_z + A_{yz} + C = 0$
1	0	0	> 0	$A_x + C > 0$
1	0	1	0	$A_x + A_z + A_{xz} + C = 0$
1	1	0	0	$A_x + A_y + A_{xy} + C = 0$
1	1	1	> 0	$A_x + A_y + A_z + A_{xy} + A_{xz} + A_{yz} + C > 0$

3.7. táblázat. XOR kapu 3 változóval

Az első egyenletből látjuk, hogy $C = 0$, így a továbbiakban ezt mindenhova behelyettesíthetjük. A 4. 6. és 7. sorokból tudhatjuk, hogy $A_{yz} = -A_y - A_z$, $A_{xz} = -A_x - A_z$ és $A_{xy} = -A_x - A_y$. Ezeket behelyettesítve a 8. sorba, az azonos tagok összevonása után azt kapjuk, hogy $-A_x - A_y - A_z > 0$. Azonban a 2. 3. és 5. sorokból azt tudjuk, hogy A_x, A_y és A_z pozitív számok. Tehát ellentettjeik összege csakis negatív lehet, nem pedig pozitív. Ez ellentmondás, tehát nem létezhet ilyen kifejezés sem, mellyel a bizonyítás teljes.

Segédváltozók használatával azonban megoldható a probléma, például az alábbi módon. Elég egyetlen segédváltozót bevezetni, ráadásul ennek a változónak mellékesen szemantikusan is adhatunk értelmet, így mondhatjuk a t segédváltozó jelentése legyen $x \wedge y$. Ekkor a kifejezésünket megkonstruálhatjuk, hogy már a korábban látott módon megkötjük, hogy $t = x \wedge y$, az $xy - 2(x+y)z + 3z$ kifejezéssel, ezt a kifejezést aztán egy megfelelően nagy számmal beszorozzuk, hogy amikor a további együtthatókat határozzuk meg, biztosan ne romoljon majd el ez az összefüggés. $-2xy - 2(x+y)t + (x+y) + t + 4zt$ kifejezés minden esetben 0-t ad a megfelelő konfigurációkra, és egy pozitív számot a helytelenekre, amennyiben csak azokat az eseteket nézzük, ahol $t = x \wedge y$ teljesül. Amennyiben a feltétel nem adott, akár -3-at is kaphatunk eredményül. Ezért tehát a $t = x \wedge y$-t kikényszerítő feltételt legalább 4-gyel meg kell szorozzuk, ezáltal a büntető tag értéke minden esetben nagyobb lesz 0-nál, ha a konfiguráció helytelen. Tehát a végleges kifejezésünk:

$$\begin{aligned} 4(xy - 2(x+y)z + 3z) + (-2xy - 2(x+y)t + (x+y) + t + 4zt) = \\ = 4xy - 8z(x+y) + 12z - 2xy - 2(x+y)t + (x+y) + t + 4zt = \\ = (x+y) + t + 12z + 2xy - 2(x+y)t + 4zt - 8z(x+y) \end{aligned} \tag{3.27}$$

x	y	z	t	büntetés
0	0	0	0	0
0	0	0	1	12
0	0	1	0	1
0	0	1	1	17
0	1	0	0	1
0	1	0	1	5
0	1	1	0	0
0	1	1	1	8
1	0	0	0	1
1	0	0	1	5
1	0	1	0	0
1	0	1	1	8
1	1	0	0	4
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

3.8. táblázat. XOR kapu

Ezáltal minden alapvető logikai kaput le tudunk írni QUBO alakkal, és a XOR kapu 3 változóval történő felírásának bizonyításánál egy általános módszert is kaptunk további, összetettebb kapuk implementálására, amellett, hogy természetesen az alapszintű kapukból is megkonstruálhatunk tetszőleges kombinációs hálózatot.

3.3.2. Összetett kapuk

Összetett logikai kapuk közül most csak a több-bementű OR kaput fogom vizsgálni. Valószínűleg a legtöbb hasonló kapura, mint például a sokbemenetű AND kapura is, az alábbiak analóg módon átvihetők.

3.3.2.1. Sokbemenetű OR kapu ($x \vee y \vee z = t$)

A 3.2.3. alszakaszban részproblémaként került elő sokbemenetű OR kapu előállítása. Ekkor arra hivatkoztam, hogy ez csak több OR kapu kaszkádosításával érhető el. Az alább közlök egy, az előbb látotthoz hasonló bizonyítást, mely szerint nem létezhet QUBO alakban felírt 3 bemenetű OR kapu, segédváltozó használata nélkül.

Ismét tegyük fel indirekten, hogy létezik ilyen kifejezés, vagyis léteznek olyan együtthatók, melyre a $A_x x + A_y y + A_y y + A_z z + A_{xy} xy + A_{xz} xz + A_{xt} xt + A_{yz} yz + A_{yt} yt + A_{zt} zt + C$ kifejezés a $t = x \vee y \vee z$ logikai kapcsolat fennállása esetén 0-t, ellenkező esetben valamilyen pozitív számot kapunk behelyettesítés után. Mivel $0 = 0 \vee 0 \vee 0$ igaz, ezért $C = 0$, így a továbbiakban ezt a konstans tagot elhagyom.

Érdemes ismét összegyűjteni az egyenleteinket és egyenlőtlenségeinket.

x	y	z	t	büntetés	egyenlet/egyenlőtlenség
0	0	0	0	0	$(C = 0)$
0	0	0	1	> 0	$A_t > 0$
0	0	1	0	> 0	$A_z > 0$
0	0	1	1	0	$A_z + A_t + A_{zt} = 0$
0	1	0	0	> 0	$A_y > 0$
0	1	0	1	0	$A_y + A_t + A_{yt} = 0$
0	1	1	0	> 0	$A_y + A_z + A_{yz} > 0$
0	1	1	1	0	$A_y + A_z + A_t + A_{yz} + A_{yt} + A_{zt} = 0$
1	0	0	0	> 0	$A_x > 0$
1	0	0	1	0	$A_x + A_t + A_{xt} = 0$
1	0	1	0	> 0	$A_x + A_z + A_{xz} > 0$
1	0	1	1	0	$A_x + A_z + A_t + A_{xz} + A_{xt} + A_{zt} = 0$
1	1	0	0	> 0	$A_x + A_y + A_{xy} > 0$
1	1	0	1	0	$A_x + A_y + A_t + A_{xy} + A_{xt} + A_{yt} = 0$
1	1	1	0	> 0	$A_x + A_y + A_z + A_{xy} + A_{xz} + A_{yz} > 0$
1	1	1	1	0	$A_x + A_y + A_z + A_t + A_{xy} + A_{xz} + A_{yz} + A_{xt} + A_{yt} + A_{zt} = 0$

3.9. táblázat. 3 bemenetű OR kapu segédváltozó nélkül

A táblázatban 4. és 6. egyenleteket kivonva a 8.-ból következik az $A_{yz} = A_t$ azonosság. Hasonlóan, szimmetria okok miatt is, a 4. 10. 12. egyenletekből következik $A_{xz} = A_t$ illetve 6. 10. 14. egyenletekből következik $A_{xy} = A_t$.

Ezeket behelyettesítve a 16. egyenletbe $A_x+A_y+A_z+4A_t+A_{xt}+A_{yt}+A_{zt} = 0$ adódik. Melyből kivonva a 4. 6. és 10. egyenleteket, $A_t = 0$ következik. Ez persze ellentmondásba kerül a 2. egyenlőtlenséggel, tehát az indirekt feltevésünk hibás, vagyis valóban nem lehet segédváltozó nélkül 3 bemenetű OR kaput definiálni QUBO segítségével.

Azt látjuk tehát, hogy 3 bit OR kapcsolatba állításához szükség van legalább 2 segédváltozóra, melyből egyik az eredmény. Ez el is érhető például úgy, hogy az első két bit-re kiszámítjuk az OR értékét, majd az így kapott segédváltozóra újra alkalmazzuk az OR operátort, így megkapjuk az eredményt. Ez a stratégia természetesen folytatható k darab változóra, így k bemenetű OR felírható $k - 1$ segédváltozóval, melybe beleértendő az eredmény bit is. Ebből persze nem következik triviálisan, hogy ez az optimális, azaz k darab bit OR kapcsolatba állításához mindig legalább további $k - 1$ segédváltozóra van szükség, de hirtelen úgy tűnik, hogy ennél jobbat nem lehet csinálni.

4. fejezet

Gyakorlati eredmények

Ebben a fejezetben alkalmazom a korábbiakban megismert és levezetett elméleti módszereket, megvizsgálom milyen futásidőt érnek el különböző megoldóprogramok használatával.

Az implementációkat a Google által üzemeltetett Colaboratory segítségével készítettem Python nyelven. A választás többek között azért esett erre a rendszerre, mert a D-Wave Systems által, Python nyelven publikált könyvtárat szerettem volna felhasználni, és ez így nem csak minimális konfigurációval megoldható, de az eredmények könnyen rekonstruálhatók a konténer technológia miatt, és nem befolyásolják őket a lokális, saját gépen eszközölt módosítások.

A teszteléshez generált gráfok előállításához és reprezentálásához a NetworkX könyvtárat használtam fel [16].

4.1. QUBO-k megoldása

A QUBO-k optimalizálásra több különböző módszert alkalmazok. A legtöbb megoldó a D-Wave által publikált könyvtárban található, de felhasználok tőlük független szoftvert is, ezzel erősítve a mezőnyt.

4.1.1. D-Wave

A D-Wave Ocean nevű programcsomagja több lehetőséget kínál QUBO formák megoldására. Van egzakt megoldója, ez azonban nagyon kicsi bemenetekre is használhatatlan a gyakorlatban, hiszen működési elvét tekintve, minden lehetőséget kipróbál. Ezen kívül van lehetőség kvantum számítás szimulálására, vagy egy valós kvantumgépnek is elküldhetjük a problémát a megfelelő API használatával [5].

Lényeges észrevétel, hogy a D-Wave megoldója – a 2.4. szakaszban leírtak szerint, az energiaminimumra törekvés elve alapján – csak minimalizálni tud, ezért a korábban bemutatott QUBO-kat is ilyen alakra kell hozni. Ez szerencsére könnyen megy, mert ha eredetileg maximalizálni szerettünk volna, egyszerűen minden tagot -1-gyel kell megszoroznunk, és az így kapott függvényt minimalizálni.

A helyes konfiguráció után van lehetőségünk a korábban bemutatott QUBO modellek tényleges próbájára.

A D-Wave rendszer esetén a QUBO megadására a Python collections könyvtárának defaultdict osztályát használom, ahol kulcsként adom meg a változópárt, amelyen a kap-

csolatban állnak, értékként pedig a szorzatukhoz tartozó együtthatót. Mivel elsődlegesen a D-Wave-et használom, mint megoldószoftvert, így a példakódokban is így konstruálom meg a QUBO-t, azaz, egy Q defaultdict osztályt használok.

A szimulált kvantumgépen való megoldás mellett két fő iránya van a D-Wave számítógépek használatának. Ebből az egyik a direkt módon beágyazása a problémának kvantumszámítógépre, a mások pedig egy hibrid megoldót használ, mely klasszikus optimalizációkat is végez.

A direkt QPU-t (Quantum processing unit) használatával sajnos probléma ütköztünk, ahogy nőtt a probléma mérete, nem túl nagy, vagyis pár száz csúcsot tartalmazó gráfokra is túl hosszú volt a beágyazás, és időnként túl sokat kellett kvantum erőforrásra is várni. Sajnos nem is teljesen sikerült kideríteni, hogy ennek miért van köze a probléma méretéhez, de nagyobb bemenetek esetén jellemzőbb volt ez a viselkedés. Azért is érdekes, mert a probléma megoldása a QPU-n hamar elkészül (amennyiben eljut odáig), amely kiderül a jelentésekből. Az erőforrásra várakozás azért tűnik valószínűnek, mert a kliens, ahonnan a hívást végzem többször egy `get_solvers` nevű függvény belsejében várakozik. Ugyanakkor viszont nem ez tűnik az egyetlen oknak, hiszen kisebb problémák megoldásánál nem jelentkezik ez a probléma, így azt is sejtjük, hogy a probléma beágyazása valamiért nem történik meg hatékonyan. További probléma a direkt beágyazással, hogy a fizikai qubitek száma rendkívül korlátos. Az egyszerű maximális vágás párszáz csúcs, így logikai qubit leképezése még nem reménytelen, de a maximális K-vágásnál, mivel a változók száma rendkívüli módon kezd növekedni, és sok klikk is kialakul közben, ez már nem bizonyult használható megoldásnak. Kimondottan a dokumentációban is megtalálható egy rész, amely a óvatosságra int, ha sok változót szeretnénk használni, ahol példaként 100 csúcs gráfot hoz, amely ezek szerint a kvantum világban már sok változónak számít. (Persze ennek ellenére szomorúan szembesültünk azzal, hogy a D-Wave API semmilyen hibaüzenet vagy figyelmeztetés formájában nem adja ezt tudtunkra, hanem probléma nélkül lefut, majd gyakorlatilag egy random vágást ad vissza eredményül. Hasonlóan, mint nagy bemenetek esetén a hibrid megoldók, ahogy az kiderül a 4.4. szakasz és 4.5. szakaszban is.)

Így a D-Wave-nél alapvetően maradt a hibrid megoldó használata. Ebből kétfélét tekintek, egyik egy beépített Hybrid megoldó, mely kevés módot ad paraméterezhetőségre, illetve lehetőség van HybridWorkflowt definiálni, ahol renget beállítás segítségével tudjuk finomhangolni az algoritmusunkat.

A QUBO-kat, és általánosságban a kvadratikus programozási feladatokat persze más, klasszikus megoldóprogramokkal is megoldathatjuk. A D-Wave maga is publikál egy qbsolv nevű csomagot, melyen kvantum erőforrások használata nélkül is lehetőség van a QUBO megoldására. A dokumentáció szerint ez képes megtalálni nagy QUBO problémák optimumát, a probléma részproblémákká bontásával, miközben egy klasszikus megoldót használ, amely a tabu algoritmust futtatja [4].

4.1.2. Gurobi

Egy kereskedelmi forgalomban lévő, egyik piacvezető lineáris programozás megoldó szoftver a Gurobi, melynek bár nem fő iránya a kvadratikus programok megoldása, ezeket is támogatja [13].

Sajnos az eredmények első körben nem voltak biztatóak, viszonylag kevés változószámnál is reménytelennek tűnt egy általános QUBO megoldása, illetve nem sokkal később az ingyenes licenc korlátaiba ütköztünk, amennyiben túl sok változót szerettünk

volna hozzáadni a modellhez. Az akadémia licenc igénylése kutatási célokra ugyan egyszerűen működik, ezek a licencek nem kompatibilisek konténerizált környezetekel, így a Colaboratory-ban sem használhatók. Egy viszonylag új licencelési módszer a Web License Service (WLS) használata[14], azonban bár a konfiguráció egyszerű, megfelelő dokumentáció hiányában, napok munkája volt ezt megfelelően elvégezni[11].

Ha mindent jól beállítottunk, akkor például a korábban elkészített Q szótárat is átkonvertálhatjuk Gurobi modellé, és megoldhatjuk a problémát, így kapva egy QUBO megoldót. A 4.1. kódrészlet bemutatja, hogy miként tudunk egy már felépített QUBO-t megoldani, végeredményként ugyanolyan típusú look-up-table (lut)-t kapva, amelyet a D-Wave válaszából is ki tudunk nyerni. Azaz a lut i. eleme megmondja, hogy az i. változó értéke 0 vagy 1.

```
def solveQUBOWithGurobi(Q):
  model = gp.Model('ModelFromQUBO', env=gurobiEnv)
  X = defaultdict(lambda: model.addVar(vtype=GRB.BINARY))
  expr = gp.QuadExpr()
  for ((u, v), w) in Q.items():
    expr.add(w*(X[u]*X[v]))

  model.setObjective(expr, GRB.MINIMIZE)

  model.optimize()

  lut = dict()
  for var in sorted(X):
    lut[var]=X[var].X

  return lut
```

4.1. kódrészlet. Max-cut QUBO

Persze, érdemes észben tartani, hogy a Gurobinak nem fő profilja a kvadratikus optimalizálás, és ha lehet, akkor érdemes korlátként hozzávenni a kényszereket, ahelyett, hogy azokat a büntetőtagokként a célfüggvénybe fogalmaznánk bele. Mivel a maximális vágásnál nem voltak kényszerek, így a direkt megfogalmazás teljesen egyenértékű volt a QUBO-ból átkonvertált modellel, tehát ott semmi különbség nincs.

Viszont például 4.4. szakaszban vizsgált maximális K-vágásnál valóban azt tapasztaltam, hogy a nyers QUBO megoldása reménytelen feladatnak tűnik a Gurobi számára, már 50-es csúcsszámnál is. Viszont amennyiben a követelményeket, hogy minden csúcs pontosan egy csoportba kerüljön, korlátként adom hozzá a modellhez, kivárható időn belül teljesít.

A Gurobi által produkált eredmények, bár pontosak, a futásidő még mindig messze elmarad a többi optimalizálótól. Azt a gyakorlati megfigyelést sikerült tennem, hogy a modell optimalizálása során viszonylag hamar megtaláljuk az optimumot, de annak belátása, hogy ez valóban az optimum, nagyon sokáig tart. Valójában arról van szó, hogy például egy maximalizálási folyamat során hamar megtaláljuk a primál feladat maximumát, de a duál feladat minimuma még jóval nagyobb, akár többszöröse az optimumnak, és ez csak nagyon lassan konvergál az optimum felé. Így érdemes lehet befolyásolni a megoldót, hogy ne pontos megoldást keressen, hanem megengedjük egy bizonyos hibaszázalékot. Azt a trükköt alkalmaztam, hogy egy paraméter beállításával jeleztem a megoldó felé, hogy amennyiben egy bizonyos számértéknél jobb megoldást talál, akkor terminálјon a folyamat[12]. Ennek az értékét a várható optimum 0.9-szeresére állítottam be. Ezt szerencsére meg tudtam tenni, mert a gráfot úgy generáltam (4.2. szakasz), hogy előre ismerjem a várható optimumot. Ez persze egy való életbeli problémánál nem lenne megtehető, azonban akkor a D-Wave által adott eredményekre sem lenne garanciánk, hogy optimális megoldást adnak,

illetve ezzel a módszerrel megkapható egy becslés, hogy mennyi ideig kell futtatni az adott algoritmust, hogy valószínűsítsük, a megoldásunk közel optimális.

4.1.3. Egyéb megoldók

Más megoldó szoftverek kipróbálását is tervben van, de sajnos, mint kiderült lehetőségeink rendkívül limitáltak. Szóba jött például a BiqCrunch, de mivel ennek nincs Python-os API-ja, így egyelőre félretettük [1]. Ígéretesnek tűnt a qpsolvers[2], amely egységes interfészt kínál több megoldóprogramhoz is, azonban nem támogatja az egészértékűségi kényszerek hozzáadását. További utánajárást igényel a CVXPY[7] és a quadprog[18], melyek akár alkalmasak lehetnek QUBO-k megoldására, eddig ezt mégsem sikerült elérni.

4.2. Gráf generálása

Az alkalmazások ellenőrzésére speciális gráfokat generáltam, melyen így előre meg tudtam határozni a várható minimális, illetve maximális vágásokat. A generált gráf rendelkezik négy különböző paraméterrel, ezek N, K, p és q. A gráfot ezek után jelöljük $G(N, K, p, q)$-val. Az ötlet csupán annyi, hogy előre meghatározunk K darab diszjunkt halmazt, hogy mindegyik csoportban pontosan N darab csúcs legyen. Ezután minden élről egyénileg döntünk, hogy őt hozzávesszük-e a gráfhoz. Amennyiben egy él egy csúcs N-esen belül fut, ez a valószínűség legyen q, amennyiben két különböző csúcs N-es között fut, akkor legyen ez a valószínűség p. Világos, hogy ha p kicsi, q pedig nagy, akkor K klasztert kapunk, ahol az élek sűrűek, a csoportok között viszont ritkák. Ellenkező esetben K darab egymástól majdnem független ponthalmazt kapunk, ahol a halmazok között viszont sok él hozzá van adva a gráfhoz. Az előbbivel egy minimális vágás várhatóan a K klasztert szétválasztó vágás lesz, utóbbi esetben pedig ugyanígy a maximális vágásra lesz egy jó becslésünk (4.2. kódrészlet).

Érdemes megvizsgálni az elfajuló eseteket. $G(N, K, 0, 1)$ esetén K darab különálló egyenként N csúcs teljes gráfot kapunk. $G(N, K, 1, 0)$ esetén pedig ennek a komplementerét kapjuk. A dolgozat keretein belül, minden közölt eredménynél egységesen a $p = 0.9$ és $q = 0.1$ paramétereket alkalmaztam.

Az élsúlyokat pedig egy véletlenszámmal generáltam minden élre. A gráf generáló függvényt úgy oldottam meg, hogy képes legyen bármilyen véletlenszám generátort paraméterként átvenni, de a példákban mindenhol 0 és 1 közötti egyenletes eloszlással (4.3. kódrészlet).

Érdemes észben tartani, hogy $n = |V| = N \cdot K$, tehát a kicsi n és nagy N nem keverendő.

```
def createRandomGraphClusters(n, K, p, q):
  G=nx.empty_graph(K*n)

  #Edges across
  for k1 in range(K):
    for k2 in range(K):
      if (k1!=k2):
        for u in range(n*k1,n*(k1+1)):
          for v in range(n*k2,n*(k2+1)):
            if (random.random()<p):
              G.add_edge(u,v)

  #Edges within
  for k1 in range(K):
    for u in range(n*k1,n*(k1+1)):
```

```
      for v in range(u+1,n*(k1+1)):
        if (random.random()<q):
          G.add_edge(u,v)

  #Assign color to each node
  for k in range(K):
    for i in range(0,n):
      G.nodes[n*k+i]['c']=k

  return G
```

4.2. kódrészlet. Gráf generálása

```
def addWeightsToGraphWithRandom(G, randAcross = random.random, randWithin = random.random):
  across_edges = [(u, v) for u, v in G.edges if G.nodes[u]['c']!=G.nodes[v]['c']]
  within_edges = [(u, v) for u, v in G.edges if G.nodes[u]['c']==G.nodes[v]['c']]

  for (u, v) in across_edges:
    G.edges[u, v]['weight']=randAcross()

  for (u, v) in within_edges:
    G.edges[u, v]['weight']=randWithin()

  return G
```

4.3. kódrészlet. Súlyok hozzáadása gráfhoz

4.3. Maximális vágás

A maximális vágás klasszikus példája a QUBO-nak, ezt könnyen elintézhetjük. Érdemes megfigyelni, hogy a változók száma pontosan $|V| = n$.

```
  for u, v in G.edges:
    w=G.edges[u,v]['weight']
    Q[(u,u)]+= -1*w
    Q[(v,v)]+= -1*w
    Q[(u,v)]+= 2*w
```

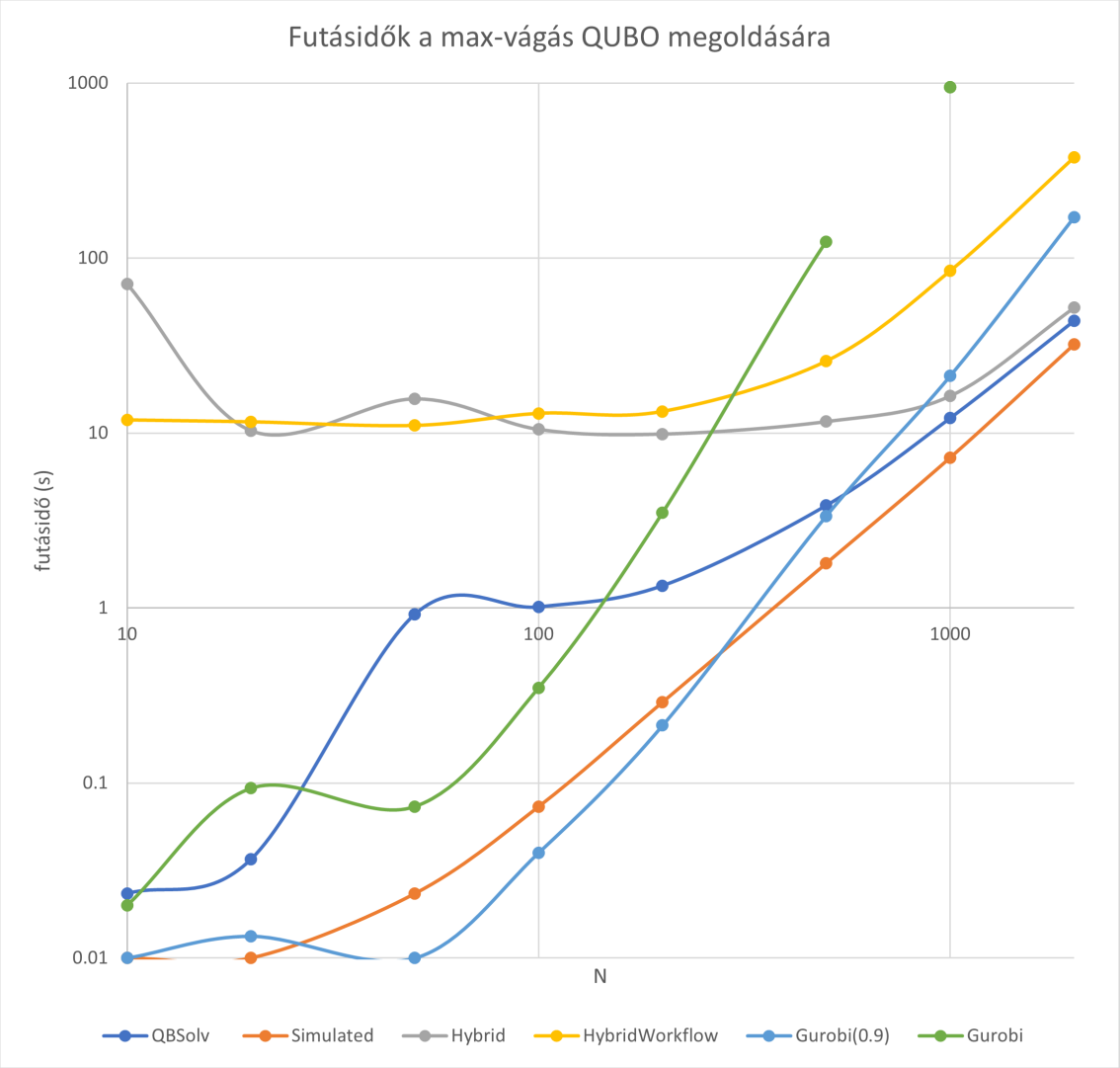
4.4. kódrészlet. Max-cut QUBO

Az egyszerű maximális vágásra felírt QUBO-t szinte minden megoldó szoftver jól kezelte. Nagy bemenetekre is általában pontos eredményt kaptunk vissza. A leglassabb megoldónak a Gurobi bizonyult, de a képet árnyalja, hogy az ő szoftverük pontosan megkeresi az optimum értékét. Ahogyan a 4.1.2. alszakaszban már leírtam, a folyamat jelentősen gyorsítható a BestObjStop beállításával, melyet a táblázatokban és a grafikonokon `Gurobi0.9`-del jelöltem.

A 4.1. ábra foglalja össze a különböző módszerekkel kapott futásidőket. (A tengelyeken logaritmikus skálát alkalmazok.) A futásidőknél sokszor jellemzőek voltak akár 1-2 másodperces eltérések is ugyanarra a bemenetre, a sok külső befolyásoló tényező miatt. (Kezdve csak azzal, hogy a Hibrid esetnél interneten keresztüli kommunikációval oldjuk meg a problémát, ezért csak a hálózati kommunikáció egy elég jelentős állandón változó tényező.) Kimondottan a Hybrid esetnél fordult elő többször az is, hogy időnként a futásidő kiugróan magas 1-2 perces időtartamú volt, ugyanakkora bemenetre, amelyre máskor 10-20 másodperc alatt lefutott.

Ami a 4.1. ábra ábrán nagyon jól látszik, hogy a sima Gurobis megoldás nagyon sok időt igénybe vett $N = 1000$-nél, ami 2000 csúcsú gráfnak felel meg, már közel 1000 másodpercig tartott a lefutás. A Gurobi módosított verziója viszont a kis bemenetek $N \leq 100$ esetében a leggyorsabb volt. Ráadásul, bár csak azt írjuk elő, hogy a megoldás az optimum

90%-ánál nagyobb legyen, így is általában megtaláljuk a pontos optimumot. (Egyedüli kivétel érdekes módon az $N = 20$ esetnél fordult elő, amikor az átlagos approximációs faktor 0.93 volt.) A bemenetek növekedésével viszont úgy tűnik, hogy a Gurobi elveszíti a versenyt, és a leggyorsabb algoritmusként a szimulált kvantumgép megoldás, és a klasszikus heurisztikus megoldó szerepel. Bár igaz, hogy ezek az algoritmusok a lokális környezeten futnak, így nincs szükség hálózati kommunikációra vagy erőforrásra várakozásra, ez akkor is meglepő eredmény. Ráadásul a grafikon alapján nem úgy tűnik, hogy ez a konstans overhead szép lassan eltűnne a bemenet méretének növekedése mellett.



4.1. ábra. Futásidők a max-vágás QUBO alakjához

4.4. Maximális K-vágás (one-hot encoded)

4.4.1. Megvalósítás

A maximális K-vágásnál több trükköt is be kellett vetnünk. Az elméleti megfontolásokon felül, itt már csupán annyit kell még kezelni, hogy a változókat jól osszuk ki. Vagyis amíg az x_{ui} változót kettő indexszel indexeltük, az elméleti felírásban, most egy indexszel

kell megadnunk egyértelműen, hogy a mátrixos formába leképezhető legyen. [1] Ezt pedig úgy oldjuk meg, hogy a változókat sorba rakva blokkonként következnek az egy csúcshoz tartozó változók, azaz x_{ui} az $(uK + i)$. változó lesz. Ennek megfelelően használjuk tehát az indexeket.

Ebből következik továbbá az a korábbi, de fontos megfigyelés is, hogy a változók száma itt $n \cdot K = N \cdot K^2$.

```
for u, v in G.edges:
  for i in range(K):
    for j in range(K):
      if (i!=j):
        w=G.edges[u,v]['weight']
        Q[(u*K+i,v*K+j)] += -1*w

for u in range(K*N):
  for i in range(K):
    for j in range(K):
      if (i!=j):
        Q[(u*K+i,u*K+j)] += P
```

4.5. kódrészlet. Max-K-cut QUBO (szimmetrikus mátrix)

```
for u, v in G.edges:
  for i in range(K):
    for j in range(i+1, K):
      w=G.edges[u,v]['weight']
      Q[(u*K+i,v*K+j)] += -1*w
      Q[(v*K+i,u*K+j)] += -1*w

for u in range(K*N):
  for i in range(K):
    for j in range(i+1, K):
      Q[(u*K+i,u*K+j)] += P
```

4.6. kódrészlet. Max-K-cut QUBO (háromszög mátrix)

A Gurobi használata során figyelembe vettem, hogy a korlátok célfüggvénybe való beleerőltetése nem előnyös (4.1.2. alszakasz) ezért a Gurobit nem csak a nyers QUBO megoldására használtam, hanem felhasználtam a korlátokat külön hozzáadva, és csak a ténylegesen optimalizálandó részeket célfüggvényként megadva (4.7 kódrészlet). Több mérés is azt igazolja, hogy ez a módszer sokkal jobban teljesít, mint az egyszerű QUBO megoldása (4.2. ábra).

```
  model = gp.Model('MaxKCut', env=gurobiEnv)

  X = model.addVars(len(G)*K, vtype=GRB.BINARY, name="Set")

  #Every node can be only in one group
  for u in range(len(G)):
    expr = gp.LinExpr()
    for i in range(K):
      expr.add(X[u*K+i])
      model.addConstr(expr == 1)

  expr = gp.QuadExpr()
  for u, v in G.edges:
    for i in range(K):
      for j in range(i+1, K):
        w=G.edges[u,v]['weight']
        expr.add(w*X[u*K+i]*X[v*K+j])
        expr.add(w*X[v*K+i]*X[u*K+j])

  model.setObjective(expr, GRB.MAXIMIZE)
```

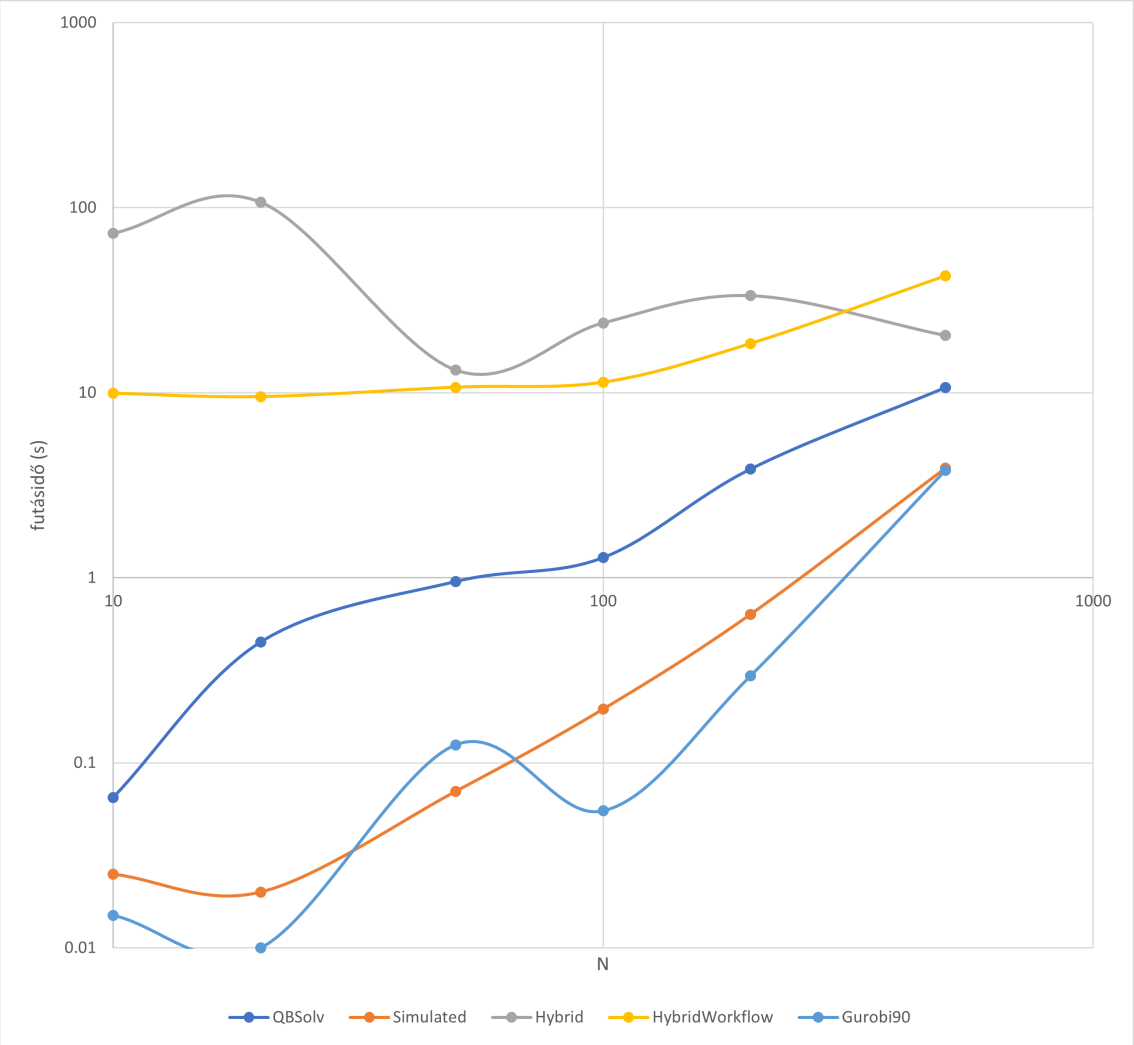
[1] Igazából ez nem szükségszerű, mert a mátrix felírásához, a Python dict típusa miatt, akár sztringeket is használhatunk kulcsként, vagyis a sor-oszlop koordináták azonosítására. Ezt a következő példában ki is használom, de ennél a feladatnál még könnyen kezelhetőek az indexek leképezése is.

4.7. kódrészlet. Max-K-cut QUBO (háromszög mátrix)

4.4.2. Eredmények

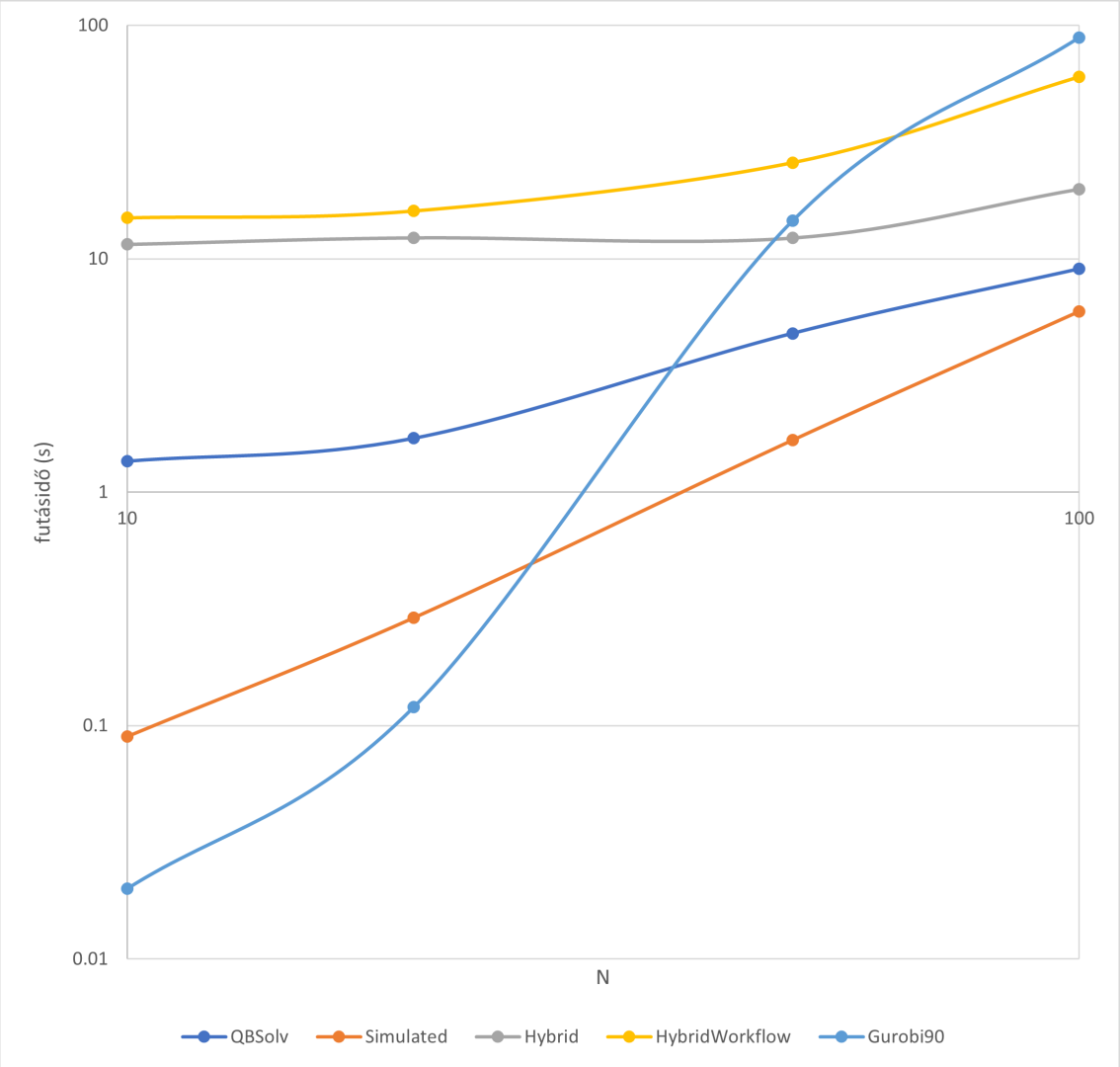
A maximális K-vágásra a futásidőket szemlélteti a 4.2 és a 4.3. ábra. Előbbhez $K = 2$, utóbbihoz $K = 4$ van beállítva paraméterként. Általános tapasztalat a grafikonoknál, hogy a Gurobi mellett – amelyet immár nem is klasszikus értelemben QUBO megoldásra használok, mert hozzáadok változókat is a modellhez –, a 4.3. szakaszhoz hasonlóan lokálisan futó algoritmusok futnak le gyorsabban.

A 4.3 grafikonon kimondottan érdekesnek tűnik, hogy a Gurobival történő megoldás hamar belassul, így összehasonlíthatatlanul rossz lesz a többi solver-hez képest. Az eredmények valóban összehasonlíthatatlanok, de sajnos más okból kifolyólag.



4.2. ábra. Futásidők a max-K-vágás QUBO megoldására (K=2)

Sajnos a futási idők elemzése önmagában nem elég, fontos megvizsgálni, hogy valóban jó (közelítő) eredményeket kaptunk-e. Amíg az egyszerű maximális vágásnál (4.3. szakasz) mindig közel jó eredményt kaptunk, és inkább a futásidő volt a szűkös erőforrás, addig a K-vágásnál gyakran kaptunk viszonylag rossz közelítéseket is, ezért érdemes mindig figyelni a



4.3. ábra. Futásidők a max-K-vágás QUBO megoldására (K=4)

talált megoldás és a várt optimum arányát, melyre továbbiakban approximációs faktorként is hivatkozok. Amennyiben ez a szám 1.0, az azt jelenti, hogy a keresett vágást pontosan megtaláltuk.

A vágások várható optimumát könnyen megmondhatjuk a generált gráfok alapján, de nagyságrendileg is érdemes kiszámolni. Ha $p = 1, q = 0$ közelítést használjuk, akkor $N \approx (N-1)$ miatt, expliciten kiszámolható a 4.1 képlettel, ahol $E(W)$ az élsúlyok várható értéke.

$$\binom{K}{2} N^2 \, E(W) \tag{4.1}$$

Érdemes utánagondolni, hogy mekkora az a közelítés, amely jónak mondható, vagy másképpen, mennyi legyen legalább az a megtalált megoldás, hogy az egyértelműen jobbnak titulálható legyen, mint egy teljesen véletlen vágás. Erre persze a válasz nem egyértelmű, de a nagyságrendek elhelyezését megkönnyítendő, érdemes kiszámolni egy teljesen véletlen vágás approximációs faktorát. Egy véletlenszerű vágás, amikor a K csoport

mindegyikébe, minden eredeti csoport K-ad részét tesszük. Ekkor ha ismét a $p = 1, q = 0$ közelítést használjuk akkor kiszámolhatjuk az élek számát, amelyet még szorozni kell az élsúlyozás várható értékével, így kapjuk a 4.2 szerinti formulát.

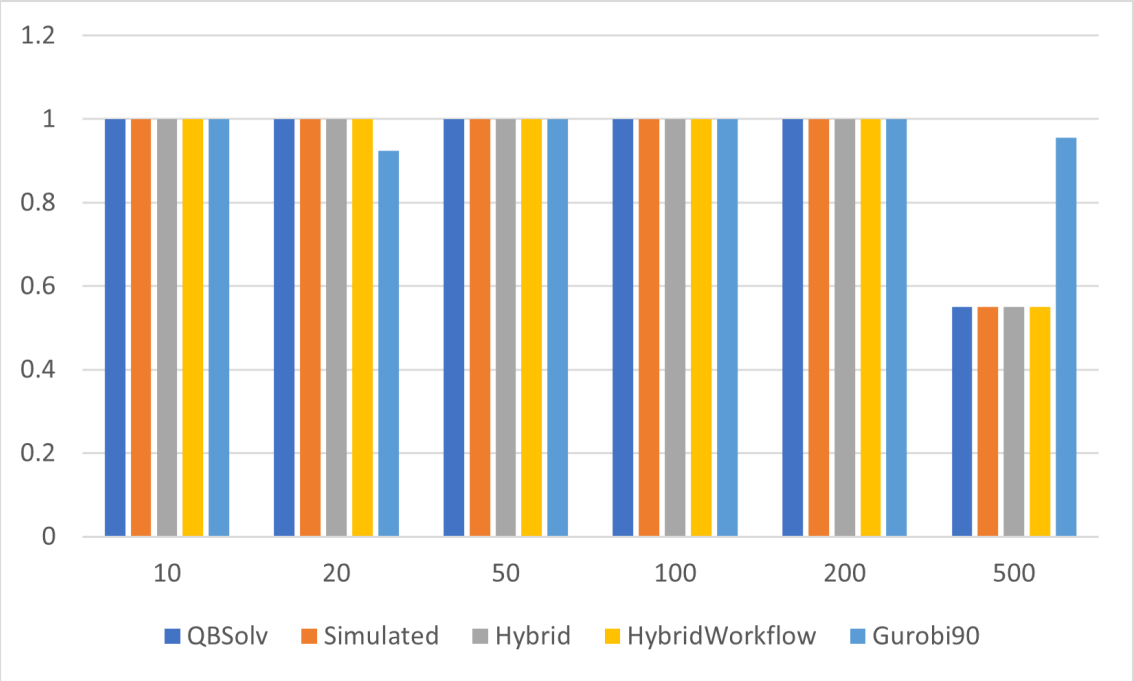
$$K \frac{N}{K} \left(N - \frac{N}{K} \right) \binom{K}{2} E(W) = N^2 \left(1 - \frac{1}{K} \right) \binom{K}{2} E(W) \tag{4.2}$$

A 4.2 formulát osztva a 4.1 kifejezéssel, megkapjuk a 4.3-ot, mint várható legrosszabb approximációs faktort.

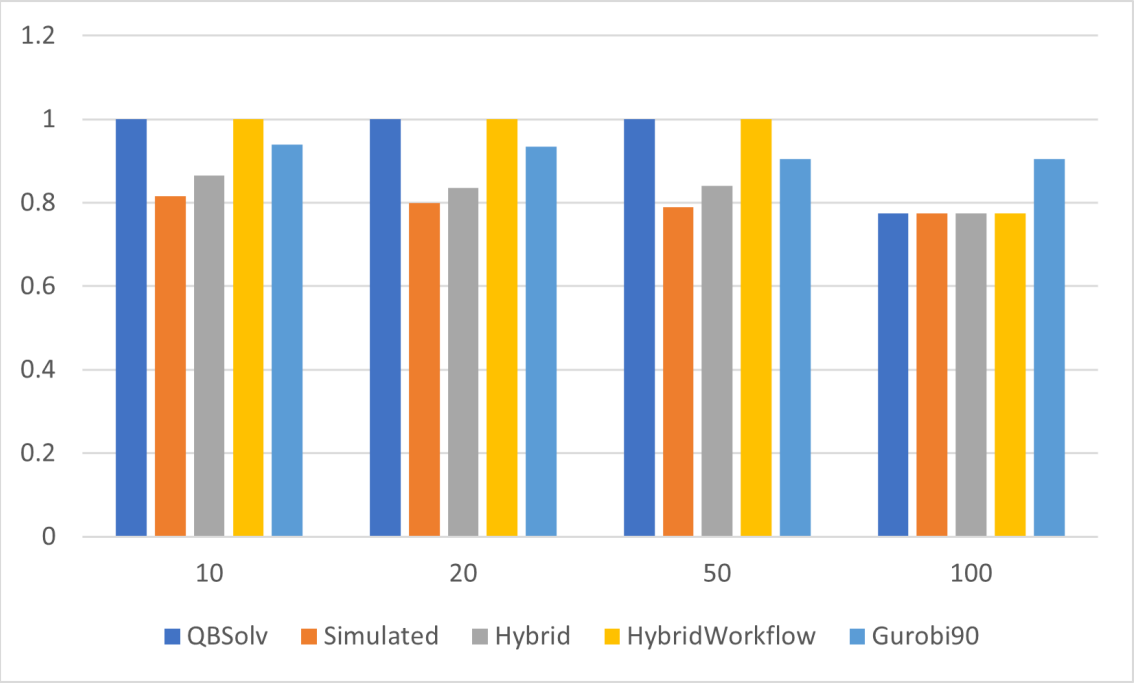
$$\left(1 - \frac{1}{K} \right) \tag{4.3}$$

Vagyis $K = 2$ esetben legalább 0.5 felett lesz az arányszám, amíg például $K = 4$ esetben legalább 0.75. Ezt fontos szem előtt tartani, hiszen ez azt jelenti, hogy $K = 4$ esetben egy 0.8 approximációt elérő algoritmus egyáltalán nem nevezhető jónak, hiszen alig jobb, mintha a gráfnak egy random felosztását adnánk vissza eredményül.

A 4.4. ábra és 4.5. ábra szemlélteti a megoldások pontosságát. Láthatjuk, hogy $K = 2, N \leq 200$ esetre minden esetben megtaláltuk az optimumot, bármely optimalizálót is használjuk. Sajnos viszont $N = 500$, illetve $K = 4$ esetben már gyakorlatilag használhatatlanul rossz vágásokat kapunk vissza. A problémával nagyon sokat foglalkoztam, így a hibakeresés során a kapott felosztáson „szemmel végignézve” is konstatálhattam, hogy valóban, mintha véletlen felosztásról lenne szó.



4.4. ábra. Approximációs faktorok a max-K-vágás QUBO megoldására (K=2)



4.5. ábra. Approximációs faktorok a max-K-vágás QUBO megoldására (K=4)

Hibrid számításnál általában valamivel jobb a eredményeket kapunk, és a HybridWorkflow volt a legmegbízhatóbb, amelyet a 4.5. ábra is mutat, de kellően nagy bemeneteket ennek a megoldónak sem sikerült kezelnie.

Azt sejtjük hogy valamilyen paraméterek konfigurálásával javítható lenne ez az eredmény, de egyelőre, a viszonylag kicsi felhasználóbázis és elérhető dokumentáció miatt nem sikerült ezt kideríteni. Az viszont sajnos kiderült, hogy ebben a formájában a megoldók nem alkalmazhatók, hiszen mint láttuk, viszonylag kicsi (néhány száz csúcsból álló) gráfokra sem adnak még jó közelítést sem.

4.5. Maximális K-vágás (binary encoded)

Mivel a korábban bemutatott elmélet alapján rengeteg segédváltozó bejön, így nehéz lett volna a természetes számok halmazára leképezni az összes változót, és így felírni a mátrixot. Ezért a QUBO definiálásánál kihasználtam, hogy a defaultdict kulcsainak nemcsak egész számokat, hanem akár stringeket (pontosabban string párokat) is megadhatunk, így string-ként kezelve a változóneveket kicsit követhetőbb a folyamat. Ezáltal a kódban a csoportokat elkódoló változókat `x_u_i`-vel jelöltem, melynek jelentése továbbra is, az u csúcs csoportját elkódoló szám `i`. bitje. `d_u_v_i`-vel jelöltem, hogy az u és v csoportját elkódoló számok `i`. bitje különbözik-e, illetve `dtemp_u_v_i`-vel a XOR kapu használatánál szükséges segédváltozót. Ennek mintájára `D_u_v` jelöli az u és v különböző csoportba kerülését. Ezen kívül további változók jönnek majd be az `or_gate_list` meghívásánál, ott a függvényen belül adom hozzá a Q-hoz az új változókat, úgy hogy azok nevükben egyediek legyenek, méghozzá a kimeneti változónév mögé mindig hozzákonkatenálom, hogy a lista mely indexű elemeit állítja OR kapcsolatba.

A kód egyébként két nagyobb részből áll, egyrészt a `D_u_v` változók előállítása, másrészt a `D_u_v` változókat össze kell szorozni a megfelelő súlyokkal(4.8. kódrészlet).

```
import math

# Initialize our Q matrix
Q = defaultdict(int)

# k is the number of bits needed to write a groupnumber
k = math.ceil(math.log2(K))

# Define the D_u_v variables with constraints
for u in range(K*N):
  for v in range(u+1, K*N):
    x_ui="x_"+str(u)+"_"+str(i)
    x_vi="x_"+str(v)+"_"+str(i)
    d_uvi="d_"+str(u)+"_"+str(v)+"_"+str(i)
    dtemp_uvi="dtemp_"+str(u)+"_"+str(v)+"_"+str(i)
    xor_gate(Q,x_ui,x_vi,d_uvi,dtemp_uvi)
    listOfds.append(d_uvi)
    or_gate_list(Q, listOfds, "D_"+str(u)+"_"+str(v))

# Update Q matrix for every edge in the graph
for u, v in G.edges:
  w=G.edges[u,v]['weight']
  Q[("D_"+str(u)+"_"+str(v),"D_"+str(u)+"_"+str(v))] += -1*w
```

4.8. kódrészlet. Max-K-cut QUBO (bináris kódolás)

4.5.1. Logikai kapuk megvalósítása

A fenti módszer működéséhez azonban implementálni kell a logikai kapukat is. Ezek az elemi kapuk esetében nem okoznak különösebb problémát, hiszen a korábbi fejezetben már megadtuk, hogy milyen együtthatókat kell használni, így az implementálásnál elég egyszerűen paraméterként átvenni egy Q objektumot, melybe a változók együtthatói elmentésre kerülnek, és magukat a változókat azonosító neveket, melyekre a logikai összefüggést alkalmazni akarjuk. A Q mátrix megfelelő mezőihez hozzáadjuk a korábban kiszámolt együtthatókat, és ezzel kész is vagyunk (4.9. kódrészlet).

```
def same(Q, in1, in2):
  Q[(in1,in2)] -= 2*p
  Q[(in1,in1)] += 1*p
  Q[(in2,in2)] += 1*p

def and_gate(Q, in1, in2, out):
  Q[(in1,in2)] += 1*p
  Q[(in1,out)] -= 2*p
  Q[(in2,out)] -= 2*p
  Q[(out,out)] += 3*p

def or_gate(Q, in1, in2, out):
  Q[(in1,in2)] += 1*p
  Q[(in1,out)] -= 2*p
  Q[(in2,out)] -= 2*p
  Q[(out,out)] += 1*p
  Q[(in1,in1)] += 1*p
  Q[(in2,in2)] += 1*p

def xor_gate(Q, in1, in2, out1, out2):
  Q[(in1,in2)] += 2*p
  Q[(in1,out1)] -= 2*p
  Q[(in2,out1)] -= 2*p
  Q[(out1,out1)] += 1*p
  Q[(in1,in1)] += 1*p
  Q[(in2,in2)] += 1*p
  Q[(out2,out2)] += 12*p
  Q[(out1,out2)] += 4*p
  Q[(in1,out2)] -= 8*p
```

```
  Q[(in2,out2)] -= 8*p
```

4.9. kódrészlet. Elemi kapuk

A több-bemenetű OR kapura is készítettem egy rövid függvényt, amely elvégzi az OR műveletet az összes bemenetére. Mindezt úgy teszi, hogy közben felépít egy bináris fát OR kapukból, időközben segédváltozókat bevezetve. Implementációját tekintve ”oszd meg és uralkodj” elven működik, vagyis rekurzívan meghívja saját magát a paraméterként átadott tömb első-, illetve második felére, majd a kapott eredményekre alkalmazza az OR műveletet (4.10. kódrészlet).

```
def or_gate_list(Q, in1, out):
  length = len(in1)
  if (length==1):
  same(Q, in1[0], out)
  return

  half = length // 2
  out1 = out +"_0-" +str(half)
  out2 = out +"_" +str(half) + "-" +str(length)
  or_gate_list(Q, in1[:half], out1)
  or_gate_list(Q, in1[half:], out2)
  or_gate(Q, out1, out2, out)
```

4.10. kódrészlet. Sokbemenetes OR kapu

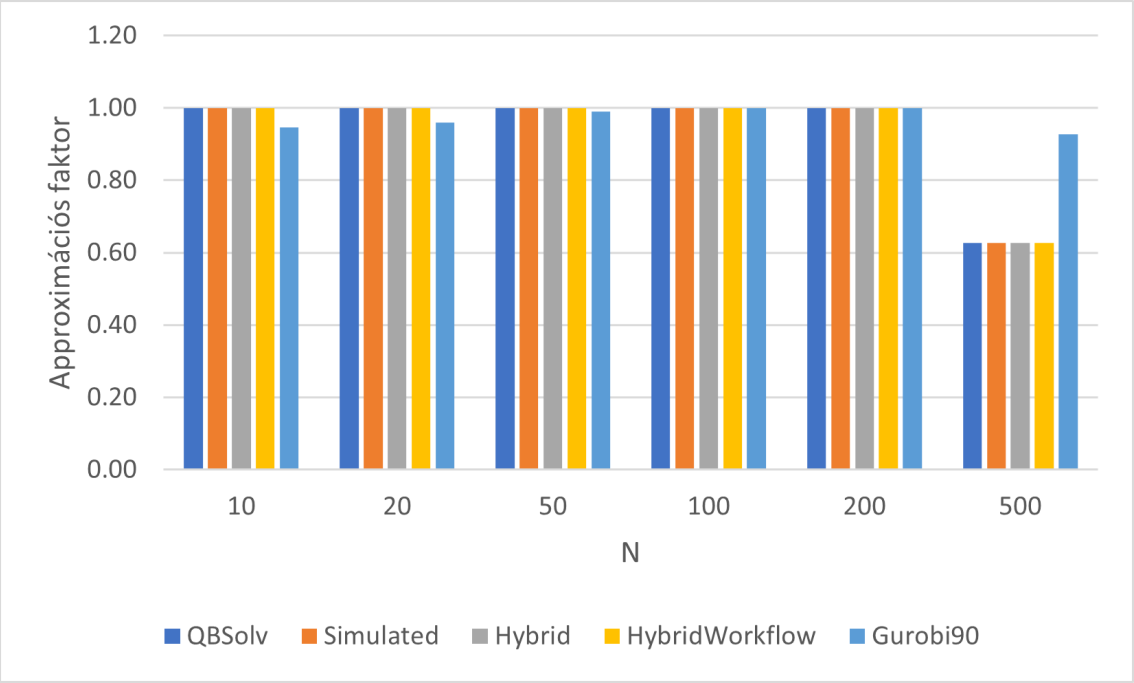
Illetve segédfüggvényként létrehoztam még egy bitenkénti XOR műveletet végző függvényt, mely csak annyit csinál, hogy a bemenetként kapott két listára elempáronként alkalmazza a korábban definiált `xor_gate` függvényt (4.11. kódrészlet).

```
def xor_list(Q, in1, in2, out1, out2):
  for i in range(len(in1)):
    xor(Q, in1[i], in2[i], out1[i], out2[i])
```

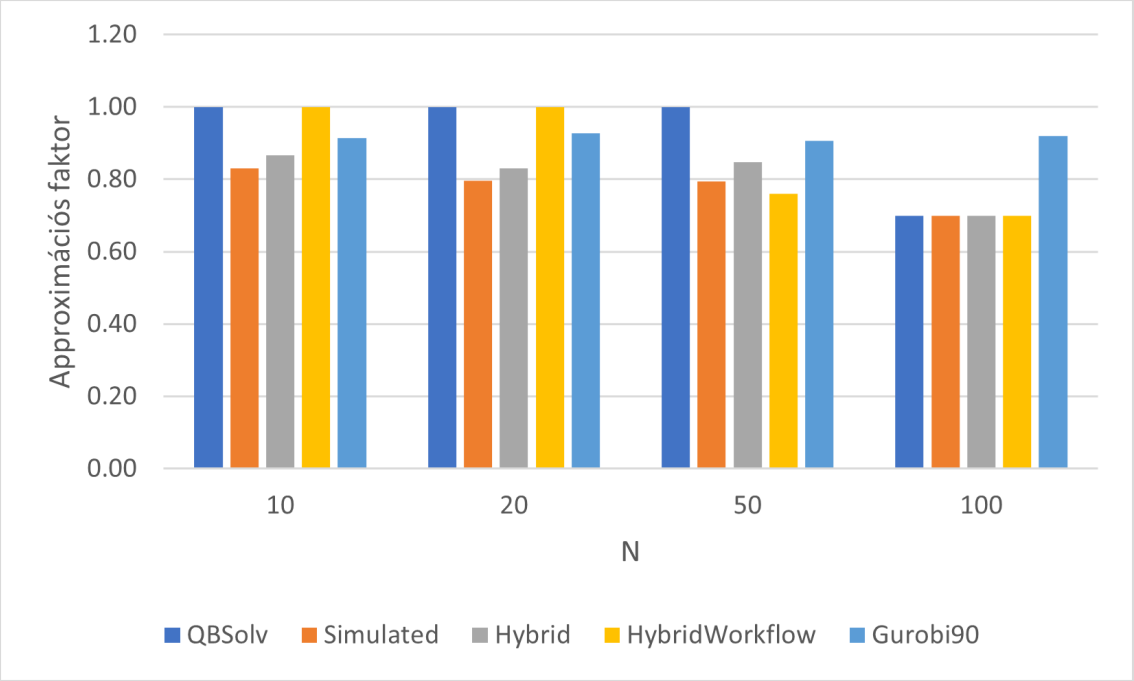
4.11. kódrészlet. Bitenkénti XOR művelet

4.5.2. Eredmények

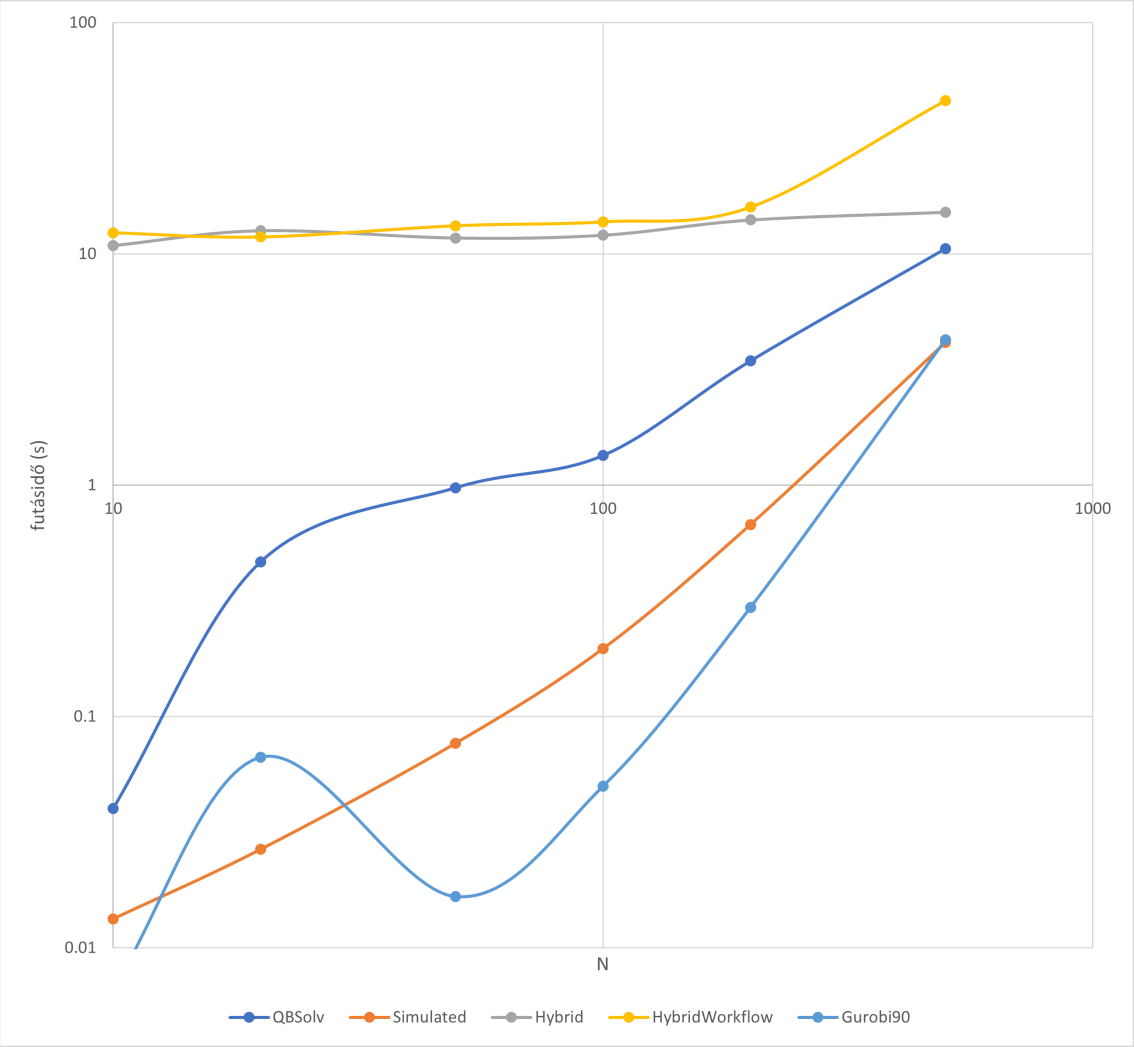
Az eredmények nagyon hasonlóan alakultak, mint a 4.5. szakaszban láthattuk. A séma is megegyező, kis bemeneteket bármelyik megoldó jól tud kezelni, de nagy bemenetekre elkezd elképesztően rosszul teljesíteni. A 4.4. szakaszban közölt ábrákkal analóg grafikonokat itt is előállítottam, melyekről jól leolvasható, hogy a megtalált vágás miként aránylik az optimumhoz (4.6, 4.7), illetve az egyes szoftverek futásidejét a 4.5. szakaszban megfogalmazott QUBO-n (4.8, 4.9). (Kivétel ismét persze a Gurobi, az megegyezik a korábban látott, korlátos programozási technikával, különben nem futna le kivárható időn belül.)



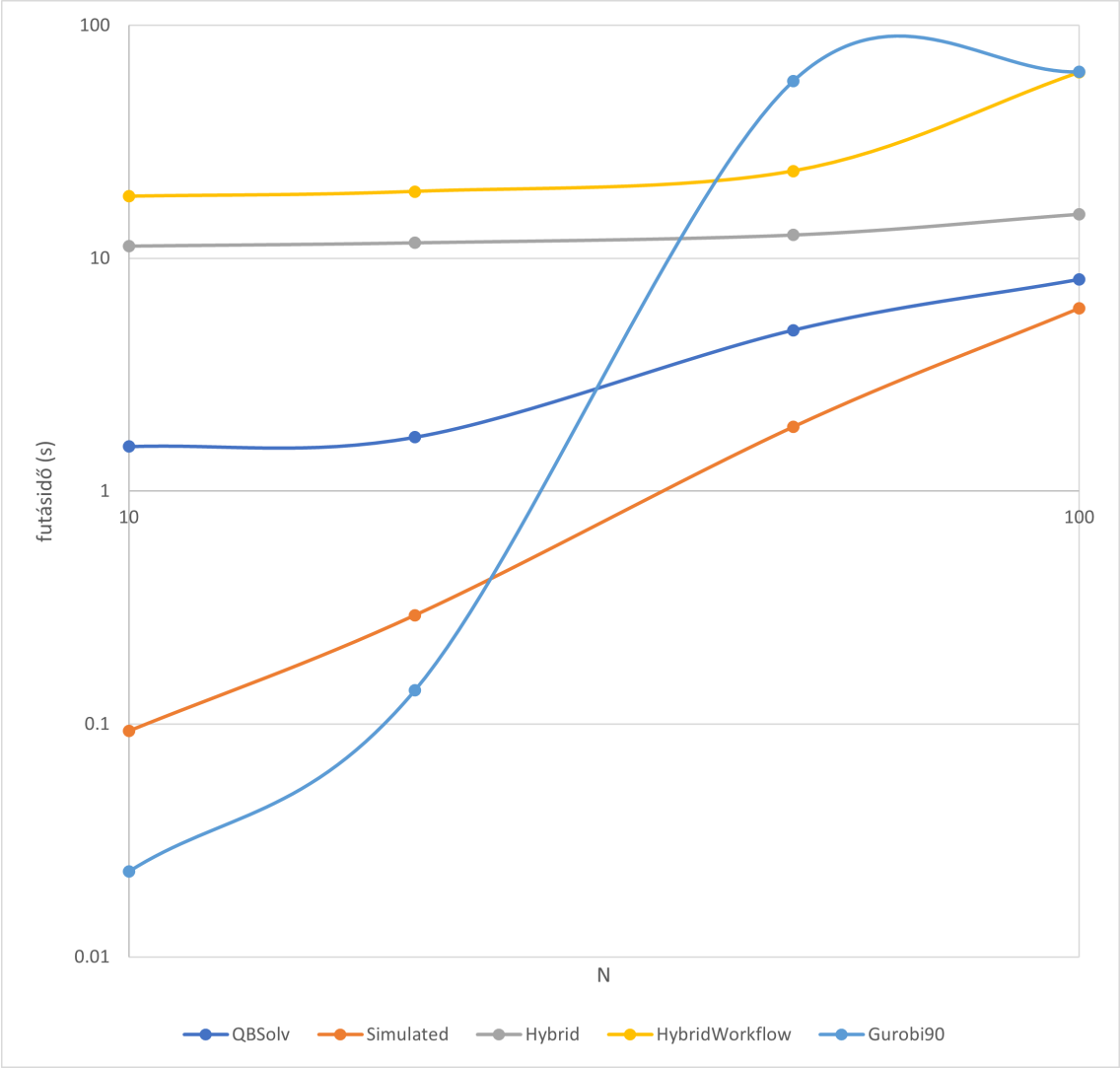
4.6. ábra. Approximációs faktorok a max-K-vágás QUBO megoldására (bináris felírás, K=2)



4.7. ábra. Approximációs faktorok a max-K-vágás QUBO megoldására (bináris felírás, K=4)



4.8. ábra. Futásidők a max-K-vágás QUBO megoldására (bináris felírás, K=2)



4.9. ábra. Futásidők a max-K-vágás QUBO megoldására (bináris felírás, K=4)

5. fejezet

Összegzés

A munka során jelentős irodalomkutatást tettem a kombinatorikus optimalizálás egyes részein, beleértve, hogy alaposan megvizsgáltam a kvadratikus bináris korlátmentes optimalizálás használatát. Ennek elméleti és gyakorlati jelentőségét is megvizsgáltam és megértettem.

Tetszőleges feladatok megfogalmazása bármilyen optimalizálási problémaként úgy gondolom, hogy nem magától értetődő feladat, de ezen a területen is sikerült sok tapasztalatot gyűjtenem, ismét különös tekintettel a QUBO-k formalizálására.

A munka nagy részét tette ki a D-Wave programcsomagjával való ismerkedés, példakódok és dokumentáció olvasása. A D-Wave gépek működési elvét is meg kellett értenem a további feladatokhoz, noha a dolgozat keretein belül igyekeztem a „matematikai letisztultságra” és az empirikus gyakorlatra fektetni a hangsúlyt.

Számomra a Python, mint programozási nyelv is újdonság volt, csakúgy, mint rengeteg más időközben felhasznált technológia. A munkát tovább nehezítette, hogy az elérhető dokumentációk mennyisége és minősége is erősen korlátozott volt.

Minden nehézség ellenére úgy gondolom, hogy az alkotott munka értelmes és egész, miközben betekintést nyújt a QUBO-k világába, és több általam konstruált példával (3.2.2, 3.2.3) gazdagítom a szakirodalmat is. Külön eredmény például a logikai kapuk megvalósításánál leírt és bizonyított állítások, melyek a XOR kapura (3.3.1.6) és a több bemenetű OR kapura vonatkoznak (3.3.2.1).

A kutatás a továbbiakban sokféle irányba folytatható. Például természetesen adja magát, hogy a maximális K-vágást nagyobb bemenetekre is tudjuk kezelni. Ehhez akár a meglévő, már felkonfigurált optimalizálókat finomítani, vagy újakat behozni (4.1.3). A repertoárt is tovább bővíthetjük további, akár nem feltétlen csak vágással kapcsolatos feladatokkal. Akár ide bekapcsolódhat a logikai kapuk implementálása (3.3), mely egy matematikailag letisztultabb, ugyanakkor gyakorlatilag is érdekesnek ígérkező témakör.

Természetesen ezeken kívül is rengeteg szál maradt elvarratlanul. Például a fizikai hardware-re való beágyazás éppen csak említésre került, vagy ahogy korábban volt szó a kvadratizálásról, azaz amikor egy tetszőleges fokszámú polinomból kvadratikus polinomot hozunk létre új változók bevezetésével (2.4). A feladat bár egyszerűen megfogalmazható, mégsem világos, hogy mi lenne az optimális módszer erre, sőt, még az is vita tárgya lehet, hogy egyáltalán milyen metrikára nézve legyen a kapott polinom „optimális”.

Irodalomjegyzék

[1] BiqCrunch developers: Biqcrunch – a semidefinite branch-and-bound method for solving binary quadratic problems, 2021.
URL `https://biqcrunch.lipn.univ-paris13.fr/BiqCrunch/`. [Online; accessed 27-Sep-2021].

[2] Stephane Caron: qpsolvers: Quadratic programming solvers in python with a unified api. URL `https://github.com/stephane-caron/qpsolvers`. [Online; accessed 7-Oct-2021].

[3] D-Wave Systems: D-Wave Concepts BQM. `https://docs.ocean.dwavesys.com/en/stable/concepts/bqm.html`.

[4] D-Wave Systems: D-Wave Ocean qbsolv documentation. `https://docs.ocean.dwavesys.com/projects/qbsolv`.

[5] D-Wave Systems: D-Wave Ocean Software Documentation. `https://docs.ocean.dwavesys.com/`.

[6] Wenceslas Fernandez de la Vega – Claire Kenyon-Mathieu: Linear programming relaxations of maxcut. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07 konferenciasorozat. USA, 2007, Society for Industrial and Applied Mathematics, 53–61. p. ISBN 9780898716245. 9 p.

[7] Steven Diamond – Stephen Boyd: CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 2016.
URL `http://stanford.edu/~boyd/papers/pdf/cvxpy_paper.pdf`. To appear.

[8] Franz Georg Fuchs – Herman Øie Kolden – Niels Henrik Aase – Giorgio Sartor: Efficient encoding of the weighted max k-cut on a quantum computer using QAOA. *SN Computer Science*, 2. évf. (2021) 2. sz., 1–14. p.

[9] Fred W. Glover – Gary A. Kochenberger: A tutorial on formulating QUBO models. *CoRR*, abs/1811.11538. évf. (2018). URL `http://arxiv.org/abs/1811.11538`.

[10] Michel X. Goemans – David P. Williamson: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42. évf. (1995. november) 6. sz., 1115–1145. p. ISSN 0004-5411.
URL `https://doi.org/10.1145/227683.227684`. 31 p.

[11] Gurobi Community: How to install academic license in google colab, 2021. URL `https://support.gurobi.com/hc/en-us/community/posts/4403075622929-How-To-Install-Academic-License-in-Google-Colab`.

[12] Gurobi Optimization, LLC: Bestobjstop, 2021.
URL https://www.gurobi.com/documentation/9.1/refman/bestobjstop.html#parameter:BestObjStop.

[13] Gurobi Optimization, LLC: Gurobi optimizer reference manual, 2021.
URL http://www.gurobi.com.

[14] Gurobi Optimization, LLC: Web license service - gurobi, 2021.
URL http://www.gurobi.com/web-license-service/.

[15] Christopher Hojny – Imke Joormann – Hendrik Lüthen – Martin Schmidt: Mixed-integer programming techniques for the connected max-k-cut problem. *Mathematical Programming Computation*, 13. évf. (2021) 1. sz., 75–132. p. ISSN 1867-2957. URL https://doi.org/10.1007/s12532-020-00186-3.

[16] NetworkX developers: NetworkX. https://networkx.org/.

[17] Svatopluk Poljak – Zsolt Tuza: The expected relative error of the polyhedral approximation of the max-cut problem. *Operations Research Letters*, 16. évf. (1994) 4. sz., 191–198. p. ISSN 0167-6377. URL https://www.sciencedirect.com/science/article/pii/016763779490068X.

[18] preeyan: quadprog: Quadratic programming solver (python).
URL https://github.com/quadprog/quadprog. [Online; accessed 7-Oct-2021].

[19] Motakuri V. Ramana – Panos M. Pardalos: *Semidefinite Programming*. Boston, MA, 1996, Springer US, 369–398. p. ISBN 978-1-4613-3449-1.
URL https://doi.org/10.1007/978-1-4613-3449-1_9.

[20] Lajos Rónyai – Gábor Ivanyos – Réka Szabó: *Algoritmusok*. 2008, Typotex.

[21] Pál Rózsa: *Lineáris algebra és alkalmazásai*. 1991, Tankönyvkiadó, Budapest.

[22] Dániel Szabó: Kvantuminformatika és gépi tanulás. Szakdolgozat (BME - VIK). 2018.

[23] Wikipedia: Maximum cut — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Maximum%20cut&oldid=1050440984, 2021. [Online; accessed 27-October-2021].

[24] Wikipedia: Very Large Scale Integration — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Very%20Large%20Scale%20Integration&oldid=1038382230, 2021. [Online; accessed 27-October-2021].

[25] Wikipedia contributors: Quadratic unconstrained binary optimization — Wikipedia, the free encyclopedia, 2021. URL https://en.wikipedia.org/w/index.php?title=Quadratic_unconstrained_binary_optimization&oldid=1020700695. [Online; accessed 28-Feb-2021].