Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Hibrid erőforrás vezénylés hálózati szolgáltatásláncokhoz

Készítette:

Dóka János, Szalay Márk

2016. október 27.

Konzulensek:

Megyesi Péter

Dr. Sonkoly Balázs

Dr. Toka László

Tartalomjegyzék

Köszönetnyilvánítás

Köszönjük mindenekelőtt Megyesi Péternek, Dr. Sonkoly Balázsnak és Dr. Toka Lászlónak, hogy konzultációs tevékenységükkel folyamatosan irányt mutattak munkánk során és beszélgetéseinkkel hozzájárultak a témában való szemléletmódunk formálásához. Továbbá köszönettel és hálával tartozunk Németh Balázsnak, aki nagyban hozzájárult az algoritmusunk fejlesztéséhez és segítette előrehaladásunkat. Az ő szellemiségük nélkül nem készülhetett volna el ez a dolgozat.

Szintén köszönettel tartozunk a BME TMIT tanszékének, hogy rendelkezésünkre bocsájtotta a munkánkhoz szükséges eszközöket és infrastruktúrát.

Kivonat

A távközlési hálózatok teljesítményére és az azokon nyújtott szolgáltatások sokszínűségére folyamatoson nő a felhasználói igény és ezt a szolgáltatók a hálózataik fejlesztésével próbálják is kiszolgálni. A hálózati teljesítmény növelése és az új szolgáltatások megvalósítása azonban rendkívül költséges és időigényes folyamat mivel jelenleg a hálózati funkciókat (pl. kapcsoló, útvonalválasztó, tűzfal, vagy a komplexebb Broadband Network Gateway és mobil hálózati vezérlők) egy-egy speciálisan a célra tervezett hardver eszköz látja el. A meglehetősen statikus megvalósítás évtizedek óta működőképes volt, de elkerülhetetlen következményei a hosszú szolgáltatási termék ciklus, a csekély erőforrás kihasználtság, a lassú fejlesztés és a speciális, drága hardverek. E problémák megoldására egy teljesen új architektúra bevezetése a megoldás: a hálózati funkciók virtualizálásának (Network Function Virtualization, NFV) célja, hogy az eddigi céleszközöket kiváltsa, és az általuk megvalósított hálózati funkciók feladatait szoftverben valósítsa meg. Ennek előnye, hogy a szoftver általános hardveren is alkalmazható, illetve lehetőség van a futtatás helyének tetszőleges módosítására a hálózat aktuális forgalma alapján. Az NFV alkalmazásához arra van szükség, hogy a fizikai erőforrások egy dinamikusan programozható hálózatban (Software Defined Networking, SDN) helyezkedjenek el. Ezen technológiák alkalmazása számos előnyt kínál, többek között lehetőséget ad az új szolgáltatások rendkívül gyors megvalósítására, illetve a már meglévők módosítására.

Amíg a virtualizált hálózati funkciók önmagukban képesek egy mai hálózati funkció kiváltására, addig egy szolgáltatás végpont-végpont megvalósítására ezek egymás utáni láncolata használható, melyet Service Chain-nek (SC) nevezünk. Egy SC logikailag fogalmazza meg, hogy milyen hálózati funkciókat valósít meg és ehhez milyen erőforrásokra (CPU, memória, sávszélesség, stb.) van szükség. Ahhoz, hogy képesek legyünk a SC-eket gyorsan és hatékonyan leképezni a fizikai erőforrásokra, szükségünk van egy speciális algoritmusra, ami biztosítja a források optimális kihasználását és elég gyors ahhoz, hogy még a következő szolgáltatási igény beérkezése előtt elvégezze a szükséges számításokat és így időben képes legyen a beérkező igények megvalósítására a valós fizikai eszközökön. A jelenlegi megoldások nem tökéletesek ezen két feltétel egyidejű teljesítésében, mert létezik ugyan gyors „online” algoritmus a probléma megoldására, de a mostani megvalósítások nem adnak opti-

mális leképezést. Ezenkívül léteznek úgynevezett „offline” algoritmusok is, melyek közel tökéletes leképezést képesek biztosítani, de az online algoritmusokkal ellentétben, nagy számú igény és összetettebb hálózati infrastruktúra esetén a futási idejük jelentős mértékben megemelkedik.

Tudományos dolgozatunk az általunk megvalósított „hibrid” algoritmust mutatja be, amely a fent említett feltételeket teljesíti az online és az offline működés kombinálásával. A megvalósítás során célunk volt egy olyan algoritmus elkészítése, amely egyrészt hosszú távon a lehető legtöbb igényt képes kiszolgálni, másrészt az újonnan beérkező kérést gyorsan, akár néhány másodperc alatt hozzá tudja rendelni az erőforrásokhoz, ami lehetővé teszi a szolgáltatás létrehozási idő drasztikus csökkentését. További szempont volt a terhelés minél egyenletesebb szétosztása az erőforrások között. A dolgozatunkban kiterjedt szimulációs helyzetekben mutatjuk be az általunk javasolt algoritmus teljesítményét, összehasonlítva azt az online és offline eljárásokéval. A szimulációs beállításokat az analitikus modellünk alapján származtatjuk, és az algoritmusokat a valós életből vett szolgáltatási példákon keresztül értékeljük.

Abstract

Users demand ever-growing performance of telecommunication networks and larger diversity of the offered services thus providers are continuously developing their networks in order to keep up with their customers' need. Increasing network performance and implementing new services are both very expensive and time-consuming processes since currently the network functions (e.g. switch, router, firewall, or the more complex Broadband Network Gateway and mobile network controllers) are realized by dedicated unipurpose hardware. The fairly static network setup was operable for decades, but with the inevitable consequences of long product cycle, low resource utilization, slow development and expensive specialized hardware. To solve this problem a completely new network architecture is required: the purpose of Network Function Virtualization (NFV) is to substitute current target devices and to implement network functions in software. The advantage of this is that the software is applicable over general purpose hardware, and there is an opportunity to choose to run it at a suitable location depending on the actual network conditions. To apply NFV, the physical resources must be in a dynamically programmable network (Software Defined Networking, SDN). The application of these technologies offers many advantages, among other things it allows fast implementation of new services, and easy modification of existing ones.

While the virtualized network functions are capable of replacing the functions of a modern network on their own, the sequential chains of virtualized network functions - called Service Chain (SC) - are applied for the end-to-end implementation of a service. An SC logically defines the kind of network functions to be implemented and the necessary resources (CPU, memory, bandwidth, etc). In order to be able to map the SCs quickly and efficiently to physical resources, we need a special algorithm which ensures the optimal utilization of resources and provides fast computation time in order to finish before the next service demand gets in. The current solutions cannot fulfill these two conditions simultaneously. Although a fast „online" algorithm exists to solve this problem, the actual realizations provide suboptimal mapping. There are also so-called „offline" algorithms which ensure an almost perfect mapping but in contrast to the online algorithms their runtime significantly increase in case of numerous demands.

In this paper we present a „hybrid" algorithm which meets the above conditions by combining the online and offline operations. Our main goal was to provide a novel mapping (embedding) algorithm which i) tries to maximize the number of mapped service requests in long term, ii) maps the new requests within a few seconds enabling extremely short service creation time, and iii) uniformly distributes the load among the resources. A comprehensive performance analysis of the proposed algorithm based on simulations confirms the advantages of our approach comparing to the online and offline methods. We set the simulation settings according to the outcome of our analytical model, and we evaluated the performance of the algorithm on real life service examples.

1. Hálózati virtualizáció

Manapság a távközlésben és az informatikában meghatározó tendencia a virtualizáció: adatközpontokban számítógépeket és szervereket, hálózatokban hálózati funkciókat és szolgáltatásokat virtualizálnak. Ez utóbbi irány manapság alapjaiban változtatja meg a távközlő hálózatok működését: a teljesítmény növelésére, a hardver erőforrások jobb kihasználására és új szolgáltatások gyors létrehozására kítűnő módszer a virtualizáció. A virtualizáció célja ezen területen lényegében az, hogy egy-egy szolgáltatást megvalósító dedikált hardvert kiválthassunk szoftveres megoldással ami általános célú hardveren fut. Ezt a megoldást Network Function Virtualization-nek (NFV) [6] nevezzük, amelynek használata a fenti számos előnnyel jár a hálózatüzemeltetők számára. Gyorsabban és költséghatékonyabban lehet módosítani és lecserélni a már meglévő futó szolgáltatásokat, illetve az újakat elindítani. További hatalmas gazdasági előny az eddigi rendszerekhez képest, hogy egy funkció ellátása általános hardveren is megoldható, nincs szükség speciális drága eszközökre.

Az NFV [7] egy olyan hálózati architektúra, amely az IT virtualizációs technológiák segítségével lehetőséget ad arra, hogy hálózati funkciókat példányosíthassunk valós időben és tetszőleges helyen egy cloud platformon belül. Ezen megoldás előtérbe helyezi az erőforrások optimális kihasználtságát és a hatékonyságot, továbbá a rendkívüli gyorsaság és produktivitás új üzleti lehetőségeket eredményez, és nélkülözhetetlen az 5G [3] hálózatok megvalósításához, mivel a tervek szerint ezekben szükségünk van arra a képességre, hogy egy új szolgáltatást akár másodpercek alatt el tudjunk indítani. Ahhoz, hogy az architektúra magasan tudja tartani az erőforrások kihasználtságát, egy olyan hálózaton kell futnia, ami dinamikusan vezérelhető. Erre nyújt megoldást a Software-Defined Networking (SDN), ahol a hálózati eszközöket szoftveresen tudjuk irányítani.

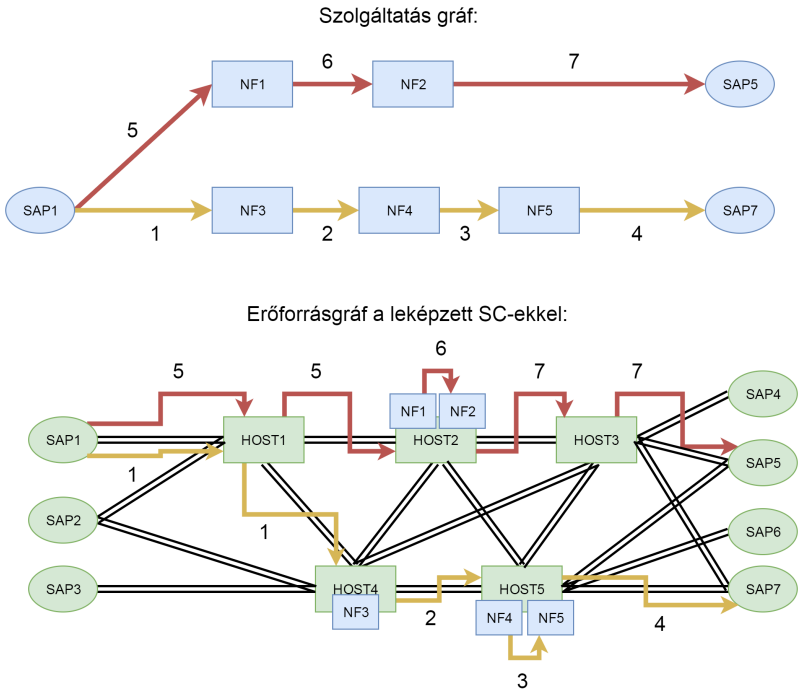
1.1. Szolgáltatásgráfok

Egy-egy hálózati funkció kiváltását egy Virtual Network Function (VNF)-el valósíthatjuk meg [8], amely egy virtualizált hálózati funkciót jelent. Egy ilyen VNF erőforrásigénye és felépítése természetesen nagyon heterogén lehet (egy VNF több virtuális számítógépet is tartalmazhat) attól függően, hogy mennyire komplex fel-

adatot kell ellátnia. Sokszor azonban nem elég csak egy hálózati funkciót megvalósítani, mivel ezek önmagukban nem alkotnak egy teljes szolgáltatást, ezért gyakran alkalmazzuk a VNF-ek láncolatát, az úgynevezett szolgáltatás láncot (Service Chain - SC). Egy ilyen lánc erőforrás igénye nem csak a benne található VNF-ek erőforrásigényeinek összege, hiszen ezen VNF-ek kapcsolatban is állnak egymással és ezen összeköttetéseknek, hálózati útvonalaknak is vannak bizonyos erőforrás igényei, jellemzően késleltetés és sávszélesség. Az ezekre vonatkozó követelményeket nem csak egy-egy VNF között fogalmazhatjuk meg, hanem egy teljes SC-re vonatkozóan is. Ez teljesen életszerű igény, hiszen a szolgáltatást használónak az a fontos, hogy egy általa igényelt szolgáltatás milyen gyorsan fut le vagy hajtódik végre és nem az, hogy a szolgáltatáson belül az egyes részegységek milyen gyorsan képesek kommunikálni egymással és mekkora a késleltetés közöttük. A fentiek alapján láthatjuk, hogy egy SC erőforrás igényeit leírni meglehetősen összetett feladat, főleg akkor ha megengedjük azt, hogy a SC-ek átlapolódjanak (VNF sharing) és hogy az egyes csomópontokból több lehetőség is legyen a továbbindulásra, vagyis forgalmak elágazhassanak valamilyen változó vagy állapot függvényében. Mivel ilyen bonyolultak lehetnek az egyes SC-ek, ezért minden bejövő szolgáltatáslétrehozási kérést egy gráffal (Service Graphs - SG) írunk le (1. ábra), ahol a csomópontok jelentik az egyes SC-ben szereplő VNF-eket és az élek a köztük futó összeköttetéseket. Egy SG több SC leírására is alkalmas és jellemzően a SC-ek összegére gondolunk SG alatt.

1.2. Erőforrásgráfok

A fizikai erőforrásaink összessége szintén nagyon változatos csomópontokból és összeköttetésekből állhatnak: tartalmazhatnak switcheket, hosztokat, különböző linkeket és ezek mindegyike más és más paraméterekkel rendelkezhet, így célszerű a rendelkezésünkre álló infrastruktúrát is egy gráffal leírni, ez az erőforrásgráf (Resource Graph - RG) (1. ábra). Ebben a gráfban a csomópontok jelölik az egyes hosztokat, amelyek lehetnek adatközpontok, szerverek vagy akár egy másik erőforrásgráf is. Ebből kifolyólag az egyes hosztokon belül további leképezések szükségeltethetnek, de az RG-ben szereplő hosztokat mi egy csomópontként kezeljük és az azon belül lévő leképezés kérdéseivel jelen dolgozatban nem foglalkozunk. Az erőforrásgráf élei az egyes csomópontokat összekötő linkek, amelyeknek szintén vannak tulajdonságai:



1. ábra. Szolgáltatásgráf, erőforrásgráf és Service Graph Embedding bemutatása

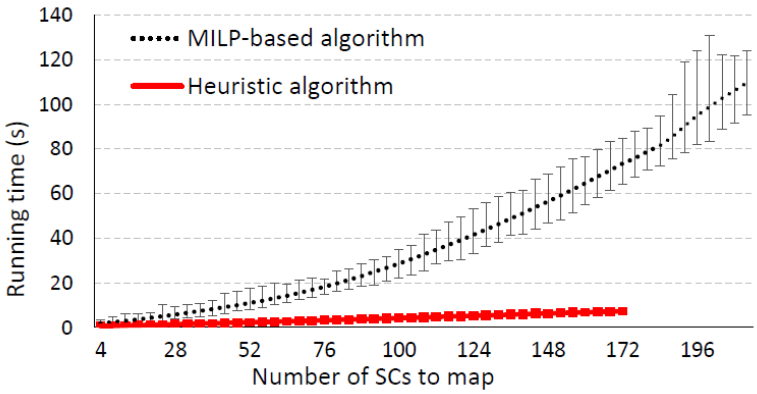
maximális adatátviteli sebesség, link késleltetése. Az infrastruktúránknak a határain helyezkednek el az úgynevezett Service Access Point (SAP)-ok: ezek olyan be- és kilépési pontok, ahol a hálózatunk kapcsolatot tart a külvilággal, jelenthetnek egy végfelhasználót illetve, egy másik hálózathoz való csatlakozási pontot is. Egy SC kiindulási és befejezési csomópontja mindig egy-egy SAP kell hogy legyen, mivel egy szolgáltatást a végfelhasználók valamilyen másik végfelhasználó irányában kérik (hálózati vezénylés szempontjából a szervereket és más hálózatokat is végfelhasználóknak tekintjük). Az egyes SC-ek ezen SAP-ok között fogalmaznak meg igényeket, megkötéseket, például végpont-végpont (end-to-end) közti maximális késleltetést.

1.3. Szolgáltási modell leképzése erőforrás modellre

Ha megvannak az igényeink és az erőforrás infrastruktúrát megfogalmazó gráfjaink, akkor egy algoritmusra van szükségünk, ami valamilyen szempontból megfelelően képzi le az Resource Graph-ra a Service Graph-okat. Ezt a szakirodalom Service Graphs Embedding (SGE) problémának nevezi. Az 1. ábrán láthatunk erre

egy példát, ahol két SC-t tartalmazó SG-ot képzünk le az erőforrás gráfra. SGE egy NP-nehéz probléma, vagyis nemdeterminisztikusan polinomiális, ami azt jelenti, hogy nem oldhatóak meg polinom időben (exponenciális algoritmus), vagyis nem P-beli problémák [13] [14].

Létezik a problémára optimális leképezést adó algoritmus, de mint exponenciális algoritmus, a futási ideje nagyon meredeken emelkedhet a rendszerben lévő kérések, az erőforrásgráf komplexitása és a bejövő szolgáltatás láncok hosszának függvényében. Az ilyen optimális leképezést adó, de lassú futási idejű algoritmusokat nevezzük ebben a dolgozatban *„offline" algoritmusoknak*. Továbbá a problémát gyorsan megoldó heurisztikus algoritmusok is léteznek, amelyek valós időben képesek végrehajtani a leképzést, de közel sem optimális módon. Jellemzően ezek a legelső olyan helyre, ahol minden feltételt kielégítően le tudnak képezni egy kérést le is képzik azt (persze sokszor valamilyen okosabb heurisztikát is alkalmaznak, hogy az eredmény konvergáljon az optimálishoz). Ezeket nevezzük ebben a dolgozatban *„online" algoritmusoknak*. A két algoritmus futási idejének különbségeit jól láthatjuk a 2. ábrán.



2. ábra. Az offline és az online algoritmus futási idejének alakulása [11]. Az X tengelyen a rendszerben lévő kérések száma, az Y tengelyen a futási idő látható másodpercben.

Egy olyan valós rendszerben, ahol túl gyakran érkeznek az új igények ahhoz, hogy egy offline algoritmus végezze a leképzést és ezáltal mindig optimális kihasználtságon tartsa a rendszerünket, kénytelenek vagyunk az online megoldáshoz folyamodni.

Ennek az a következménye, hogy nem használjuk ki maximálisan az erőforrásaink lehetőségeit és előfordulhat, hogy visszautasítjuk igények kiszolgálását erőforráshiányra hivatkozva, miközben optimális leképezés esetén még teljesíteni tudtuk volna a kérést vagy a kéréseket. Érthető tehát, hogy egyszerre kell biztosítanunk az újonnan érkező igények gyors kiszolgálását és az erőforrásaink folyamatos optimális kihasználtságon tartását (vagy legalább is azt, hogy a kihasználtság folyamatosan tartson az optimális felé). Az általunk megalkotott *hibrid algoritmus* az előbb említett mindkét feltételt teljesíti egy online és egy offline algoritmus egyidejű alkalmazásával. A további fejezetekben részletesen kifejtjük hogyan alkalmazzuk ezeket az algoritmusokat és alkotunk egy, a fent említett problémákat kiküszöbölő hibrid megoldást.

2. Létező erőforrásvezénylési megoldások

Az alábbi fejezetben részletesebben ismertetjük a hálózati szolgáltatásláncok erőforrásvezényléséhez szükséges ismereteket, különösen a dolgozatunk szempontjából relevánsakat.

2.1. Virtual Network Embedding

Virtual Network Embedding (VNE) alatt azt értjük, amikor egy fizikai erőforrásokat reprezentáló hálózatra több logikai topológiát képzünk le és törekszünk az erőforrások optimális kihasználására. A VNE egy NP-nehéz, részletesen tanulmányozott probléma és számos algoritmus ismert a megoldására [5] [13] [9] [4] [14]. Matematikai optimalizálás szemszögéből a Virtual Network Embedding nagyon szoros kapcsolatban áll a Service Graphs Embedding-el [11]. A SGE annyiban különbözik és ezáltal nehezebbé is válik, hogy további követelményeket kell figyelembe vennünk a leképzés és optimalizálás során, mint például a végpont-végpont késleltetés (end-to-end delay) egy SC-re vonatkozóan vagy a VNF sharing. Ez utóbbi azt teszi lehetővé, hogy ha egy olyan SC-t akarunk leképezni amelynek egyes hálózati funkcióiból már szerepel egy példány a rendszerünkben, akkor nem példányosítjuk újra az adott VNF-et, hanem átirányítjuk a láncot a meglévő példányhoz, majd vissza. Ez nagyban növeli a rendszer kihasználtságát, hiszen az adott VNF erőforrásait megspóroljuk. Hátránya viszont, hogy szintén növeli a leképzési időt is, mivel minden VNF leképzése előtt meg kell vizsgálnunk, hogy szerepel-e már példány belőle a rendszerben és tudjuk-e azt használni (elég lesz-e az oda vezető linkek erőforrása). Némely felhasználási területeken nem ajánlott használni, mert biztonsági kockázatot jelenthet.

2.2. Offline algoritmus

A hibrid algoritmusunk időközönkénti optimalizálásáért egy kevert egész értékű lineáris programozás (MILP) alapú offline algoritmus a felelős, amelynek központi eleme a Gurobi Optimizer [1]. A továbbiakban ezeket mutatjuk be a teljesség igénye nélkül.

2.2.1. Lineáris programozás

A lineáris programozás (LP) meghatározó és elterjedt módszer az optimalizálás területén, amelynek több oka is van: lineáris programozási feladatok megoldására hatékony algoritmusok léteznek, számos nem LP feladatot is jól lehet közelíteni LP modellel, nemlineáris feladatok is átalakíthatóak ekvivalens LP feladatokká, stb.

Az egész értékű lineáris programozás (Integer Linear Programming - ILP) általános modellezési eszköz, amit először olyan feladatokra használtak, amikor a termelési döntési[1] változók csak egész értéket vehettek fel, de manapság egyre szélesebb körben alkalmazzák többet között ütemezési feladatok, hátizsák probléma[2], utazó ügynök probléma[3], elosztási és hozzáférési problémák, hálózati optimalizálási problémák és még számos más feladat modellezésére és megoldására. Ha egy lineáris programozási feladat néhány változója csak egész értéket vehet fel, akkor kevert egész értékű lineáris programozásról (Mixed Integer Linear Programming - MILP) beszélünk. Egy MILP feladat megoldása nagyságrendekkel több időt vehet igénybe, ezért a megoldás során általában eltekintenek az egész értékűségi követelményektől és a feladatra vonatkozó úgynevezett LP relaxációt[4] oldják meg. Ha erre a megoldásban minden változó egész értéket vesz fel, akkor a megoldás az eredeti MILP probléma megoldására is jó, ha nem, akkor kerekítjük egészre az értékeket és megközelítőleg jó megoldást kapunk.

2.2.2. Gurobi Optimalizer

A Gurobi Optimalizer [1] egy state-of-the-art megoldás matematikai programozási feladatokhoz. Az optimalizáló motorját a modern architektúrák és több processzoros rendszerek maximális kihasználására tervezték, továbbá a legújabb algo-

[1]Termelési döntés az inputok és outputok mennyiségeinek megválasztása.

[2]Hátizsák probléma: Egy ismert tulajdonságokkal rendelkező tárolót úgy kell megtöltenünk szintén ismert tulajdonságú dolgokkal, hogy a tárolt dolgok összértéke maximális legyen.

[3]Utazó ügynök probléma: Adott n darab pont és az azokat összekötő költséggel ellátott utak. Keressük meg a legolcsóbb utat egy pontból kiindulva, amely az összes pontot egyszer érinti és visszatér a kiindulási pontba.

[4]Relaxáció: csökkentjük a probléma komplexitását általában azzal, hogy megkötéseket vagy paramétereket hagyunk el az eredeti problémából. Sokszor a relaxált probléma megoldása nagyon közel van az eredeti probléma megoldásához, de nagyságrendekkel könnyebb lehet megoldani.

ritmusokat is tartalmazza, többek között a kevert egész értékű lineáris programozást is.

2.3. Online algoritmus

A Service Graph Embedding probléma megoldására számos heurisztikát alkalmazó algoritmus látott napvilágot. Ezek legtöbbször relaxálják a problémát oly módon, hogy kihagynak valamilyen paramétert a számítás során és így bár nem a minden szempontból legjobb leképzést találják meg, de nagyságrendekkel csökkentik a leképzés idejét.

A dolgozatunkban kifejtett hibrid megoldáshoz a [11] cikkben szereplő algoritmust alkalmaztuk mint online algoritmus. Ez egy mohó visszalépéses módszert követő megoldás, ami a szolgáltatásgráfban szereplő csomópontokat és éleket egyszerre képzi le. Számos vezénylési paramétert és metrikát kínál, amelyek segítségével testre szabhatjuk a működését és lehetővé teszi különböző felhasználási területeken való alkalmazásra. A vezénylő algoritmus lényege, hogy mohón leképez a SG-ból egy csomópontot (node-ot) és a hozzá tartozó éleket (linkeket). Egy ilyen összerendelést leg-nek nevezünk és egy leg (vagyis a node és a kapcsolódó linkek egyszerre) mindig egy lépésben képződik le a vezénylés, más szóval az orkesztrálás során. Ha a mohó algoritmus egy lépésben nem talál megfelelő helyet a leképzésnek, akkor a bejárási fájában visszalép, hogy egy másik helyet keressen a SC-nek. Paraméterekkel megadható, hogy a visszalépés mértéke mekkora legyen. Ha ezt az értéket nagyra állítjuk, akkor nagyobb eséllyel találunk leképezést, de a futási idő lényegesen megnövekedik. Ha sikeres volt a leképzés, akkor a SC-ben lévő VNF-ek és linkek által igényelt erőforrás értékek levonásra kerülnek a RG azon csomópontjaiból és linkjeiből ahova ténylegesen le lettek képezve. Az így keletkezett RG-ot kapja meg a következő kérés beérkezésekor az algoritmus. Ebből következik, hogy az online algoritmus minden új kérés érkezésekor tudatában van a már leképzett előző kérésekről.

A módszer egyszerűen közelít meg egy NP-nehéz problémát és talál rá megfelelő, közel optimális megoldást. A tesztjeink során az algoritmus azonnal képes volt leképzést adni bármilyen SC-re, és ezért döntöttünk ezen algoritmus használata mellett, mint online megoldás. Ugyanakkor sok olyan eshetőségre nincs felkészítve, amelyekre mindenképpen szükségünk volt egy realisztikus szimuláció megalkotása

során, ezért számos kiegészítést és módosítást kellett eszközölnünk rajta mielőtt használhattuk volna.

2.4. Network Function Forwarding Graph

A munkánk során használt algoritmusok és a hibrid algoritmusunk kimenete is egy úgynevezett Network Function Forwarding Graph (NFFG) [10]. Ez egy a BME által készített hálózati szolgáltatásokat és erőforrásokat *egységesen* leíró gráf. Célja, hogy az erőforrásokra leképzett hálózati funkciókat leírja. Egy NFFG tehát egyszerre tartalmazza az erőforrásgráfot és az azokra eddig leképzett szolgáltatás láncok összességét, vagyis a szolgáltatásgráfot:

$$\text{SG} + \text{RG} = \text{NFFG}.$$

Az NFFG használatának előnye, hogy egy leíró objektumban képesek vagyunk minden a hálózatot érintő releváns információhoz hozzájutni, mint például a RG és a SG tartalma, az erőforrások és a linkek kapacitásértékei, a hosztok által támogatott VNF típusok és egyéb fontos metaadatok. A hibrid algoritmusunk megvalósításában ezt a formátumot használtuk bemeneti formátumnak annyi módosítással, hogy amikor SG-ot akartunk megadni, akkor a RG-ot leíró részeket töröltük ki az NFFG-ből, amikor pedig RG-ot, akkor a SG-ot leíró részeket töröltük ki.

3. Az erőforrásvezénylési megoldásunk: hibrid algoritmusok

Munkánk célja az volt, hogy egy olyan algoritmust hozzunk létre, amely egyszerre képes optimális állapotban tartani az erőforrásokat ezáltal növelve az adott erőforrásgráfra leképezhető maximális SC-ek számát, és biztosítani a valós idejű működést, vagyis azonnal le tudjunk képezni egy érkező kérést. A hibrid megoldásunkban az előbbi feladatot az offline algoritmus, míg az utóbbit az online látja el: egy offline és egy online algoritmust párhuzamosan futtatunk egymás mellett és mindkettő egy általunk generált *származtatott erőforrásgráfra* képzi le a neki adott kéréseket. A továbbiakban a következő nevekkel fogunk hivatkozni a különböző erőforrásgráfokra: onlineRG - az online algoritmus által látott erőforrásgráf, offlineRG - az offline algoritmus által látott erőforrásgráf, fullRG - a hálózat teljes erőforrásgráfja, FC(onlineRG) - onlineRG szabad kapacitása, melyeket a jövedőbeli kérések lefoglalhatnak, FC(offlineRG) - offlineRG szabad kapacitása, FC(fullRG) - teljes hálózat szabad kapacitása), RC(onlineRG) - onlineRG foglalt kapacitása, és így tovább az offline és a full RG-re. Attól függően, hogy hogyan osztjuk szét a teljes hálózat erőforrásait és kapacitásait, többféle különböző hibrid algoritmust valósítottunk meg. Az alábbiakban ezeknek különböző működéseit részletezzük, a következő fejezetben pedig a teljesítményüket hasonlítjuk össze realisztikus szcenáriók szimulálásával.

3.1. Teljes erőforrásgráfot a két algoritmusnak

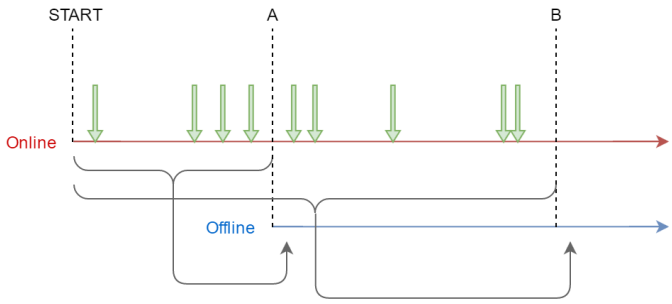
A kezdeti állapotban csak az online algoritmus indul el és várja a kéréseket. Ekkor mivel még nem volt SC leképzés az erőforrásgráfot üresen látja, tehát az onlineRG szabad kapacitása megegyezik a hálózat valódi kapacitás értékeivel (onlineRG = fullRG). Az offline algoritmus még nem fut, hiszen azt csak a már rendszerben lévő, leképzett SC-ek optimalizálására használjuk. A futás során folyamatosan érkezett kéréseket az online algoritmus valós időben képzi le. A működést szimuláló 3. ábrán ez a fázis a START és az A időpillanat között eltelt időt és a kiszolgált négy kérést jelenti.

A kezdeti állapot végével, amikor elérünk egy meghatározott SC számot, a futó online mellett elindítjuk párhuzamosan az offline algoritmust is. Az FC(offlineRG),

vagyis hogy az offline algoritmus mennyit lát a valódi hálózatból erőforrás tekintetében, az eredeti fullRG-val kapacitásával egyenlő (FC(offlineRG) = FC(fullRG)). Ennek könnyen látható előnyei és hátrányai vannak. Nagy előny például, hogy ekkor az offline algoritmus a hálózati erőforrások és kapacitások összességét látja, így az egész rendszerre nézve az optimális megoldást képes kiszámolni. Hátránya azonban, hogy az offline futási ideje alatt az online algoritmus az időközben érkező kéréseket folyamatosan leképzi, ennek következtében előfordulhat az az eset, hogy mire az offline módszer végez az optimalizálással, addigra a hálózatban lévő szolgáltatások száma - és így a foglalt erőforrások és link kihasználtságok mértéke - olyan szinten megváltozik, hogy az offline által kapott optimális megoldás és az addigra kialakult állapot nem fésülhető össze, vagyis együttes megvalósításuk nem lehetséges. Az erőforrásgráfok összefésülésének miként jéről részletesebben a 3.3-as fejezetben még lesz szó, de elöljáróban annyit megemlítünk, hogy egy összefésülés már akkor is kudarcba fullad, ha csak egy csomópontban ütközik a két algoritmus leképezése.

Az offline algoritmus számolás az 3. ábrán az A időpillanatban kezdődik el és B időpillanatban lesz vége, miután egyesítjük a két módszer által létrehozott NFFG-t. Ez azt jelenti hogy az offlineRG és az onlineRG leképzéseit összefésüljük és a kapott eredményt az onlineRG-ben eltároljuk. Szerencsés esetben ekkor úgy néz ki a rendszer, hogy a kezdeti négy szolgáltatás optimálisan helyezkedik el a hálózatban, amihez csatlakozik még az A és B között beérkezett és leképzett öt darab szolgáltatás. Ezesetben az első négy szolgáltatás eredeti leképzését, amit még az online algoritmus döntött el, megváltozatjuk az offline algoritmus kimenetének megfelelően, azaz az érintett VNF-ket a megfelelő csomópontokba migráljuk és újrakonfigurájuk a csomópontokat összekötő hálózati éleket.

Rosszabb esetben az A és B között érkezett kérések annyira megváltoztatták a FC(onlineRG)-t, vagyis az online erőforrásgráfban elérhető szabad kapacitásokat, hogy az egész hálózatra nézve nem valósítható meg az előbb ismertetett állapot, így ekkor mind a 9 kérés az online által leképzett állapotban marad, ami sajnos nem feltétlenül optimális. Vagyis ilyenkor eldobjuk az offline algoritmus által generált leképzési eredményt. Ezt követően a folyamat kezdődik elölről és az offline megkapja az eddigi összes beérkezett kérést azokkal együtt, amiket már az előző körben is megkapott.



3. ábra. Az offline algoritmus szolgáltatás lánc kezelése

A módszer lényege természetesen az, hogy az offline algoritmus úgy képzi le a neki adott kéréseket, hogy azok erőforrás felhasználása kisebb lesz mint az optimalizálás előtt, így amikor vége egy offline algoritmus futási ciklusának, akkor a nekiadott offlineRG-ban szabad erőforrásoknak kell keletkeznie. Ebben az esetben az összefésülést követően ezt a felszabadított forrást megkapja az onlineRG.

Erre a működésre a továbbiakban a **duplaszázas hibrid** algoritmusként hivatkozunk utalva arra, hogy mindkét algoritmus az eredeti hálózat erőforrásainak száz százalékát látja.

3.2. Dinamikusan változó erőforrásgráfot a két algoritmusnak

E módszer lényege, hogy a *duplaszázas* működéstől eltérően, itt az online és az offline algoritmusok nem látják teljes egészében az elérhető hálózatot és erőforrásokat.

Első megközelítésben a hibrid program indulása előtt egy meghatározott százalékban szétosztjuk a fullRG szabad kapacitásait az online és az offline algoritmusok között, vagyis ennél a működésnél az FC(onlineRG) és az FC(offlineRG) összege teszi ki a FC(fullRG)-t. Fontos megjegyeznünk, hogy ekkor az online és az offline erőforrásgráf is a teljes topológiát látja, csak a kapacitásértékek különbözőek a két RG-ben. Ennek előnye, hogy nem állhat elő olyan helyzet a működés során, hogy az online leképzés eredményét és az offline optimális állapotát ne lehessen összefűzni, hiszen az FC(onlineRG) + FC(offlineRG) = FC(fullRG). Hátránya viszont az, hogy amennyiben az offline algoritmus kezdésének és befejezésének ideje között annyi kérés érkezik a rendszerbe amennyinek nincs elegendő hely az onlineRG-ben, akkor a

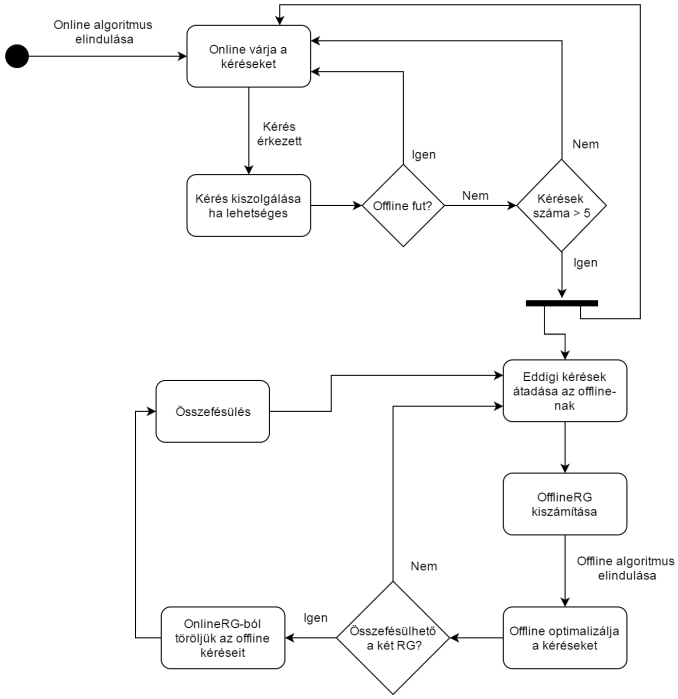
SC igények dobásra kerülnek, holott előfordulhat, hogy az egész rendszerben még lenne hely a számukra az offlineRG-ben, az optimális erőforráskihasználtság miatt fennmaradó tartalékban. Ezért második megközelítésben nem előre definiált módon szabjuk meg a két algoritmus számára kiadott erőforrásgráfokat, hanem ezek méretét dinamikusan módosítjuk az aktuális terheltség függvényében. Az ötlet a következő: az online algoritmus kapja meg a hálózat kapacitásainak 100%-át, viszont az offline minden egyes indulásakor változó kapacitásméretű erőforrásgráfot kapjon.

Ahogy az a 4. ábrán is látható, a kezdeti állapotban csak az online algoritmus indul el és várja a kéréseket. Ekkor mivel még nem volt SC leképzés az erőforrásgráfot üresen látja. Az inicializációs állapot végével elindul párhuzamosan az offline algoritmus is. A 4. ábra alsó részén láthatjuk az offline algoritmus működésének lépéseit. Először megkapja az online által teljesített kérések mindegyikét, majd ezután következik az offlineRG paramétereinek beállítása. Ezt úgy határozzuk meg, hogy az eddig online által lemappelt SC-khez tartozó erőforrás mennyiségek maximumát állítjuk be mint offlineRG.

A könnyebb értelmezéshez nézzük meg a működést egy példán keresztül. A 3. ábrán látható, hogy a START és az A időpillanat között 4 darab kérés érkezett a rendszerbe. Tegyük fel, hogy ezen kérések leképzésére az online algoritmus felhasznált X mennyiségű erőforrást, ekkor az offlineRG összes kapacitása közel X lesz. (Pontosabban több lesz, mint X, de az offlineRG kiszámításáról részletesebben később még lesz szó.) Ezt azért tehetjük meg mert feltételezhetjük, hogy az offline algoritmus legalább olyan jó leképezést talál ugyanazokra a SC-ekre mint az online, így „el kell férnie” ugyanannyi erőforráson mint amennyin az online elfért. Tehát az offline algoritmus az így kiszámolt erőforrásgráfot és az online által lerakott kéréseket kapja meg, de itt már nem egyesével, hanem egyszerre az összeset, így szabadon választhatja meg a leképzés sorrendjét ezáltal is elősegítve az optimalizált állapot elérését. Az online algoritmus pedig folytatja a futását az eddigi 100%-os kapacitás értékekkel.

Amikor befejezte a számolást az offline program, akkor egyesítjük a két algoritmus által létrehozott eredményt. Ezt úgy tehetjük meg, hogy elsőnek is az onlineRG-ből kitöröljük azokat a leképzéseket, amelyeket az offline algoritmus optimalizált. A törlés után a következő lépés az offlineRG és a már „megcsonkított” onlineRG

összefésülése. Ez az esetek többségében sikeresen végrehajtható, azonban telített állapotban előfordulhat az, hogy mégsem mivel az onlineRG és offlineRG foglalt kapacitásainak összege nagyobb mint az eredeti fullRG összkapacitása. Ez csak telített állapotnál lehetséges és a következő fejezetben kifejtjük, hogy hogyan kerülhetünk ilyen állapotba.



4. ábra. Hibrid algoritmus állapotdiagramja dinamikus erőforrásgráf kiosztással

Sikeres összefésülés esetén olyan állapotba kerülünk, amikor az offline indulása előtt érkezett kérések már optimalizáltan kerülnek leképzésre, de az indulása után érkezettek nem, hiszen azokat az online rakta le, nem feltétlenül optimálisan. Ekkor a folyamat kezdődik elölről: az offline megkapja az eddigi *összes* beérkezett kérést (azokkal együtt, amiket már az előző körben optimalizált) és egy olyan offlineRG-t amelyre az online és az offline algoritmus az összefésülés után képes volt leképezni a SC-eket, így önállóan az offline-tól is elvárhatjuk ezt. Az online pedig a továbbiakban az összefésült erőforrásgráfot kapja.

Erre a működésre a továbbiakban a **dinamikus hibrid** algoritmusként hivatkozunk.

1. táblázat. A különböző hibrid megvalósítások jellemzői

	Rendszert leíró NFFG	**OnlineRG**	**OfflineRG**
Duplaszázas	onlineRG	100 %	100%
Dinamikus	onlineRG	100%	fullRG - max(onlineRG)

Az általunk javasolt két különböző hibrid algoritmus jellemzői az 1. táblázatban láthatóak. Összefoglalásként nézzük át a köztük lévő különbségeket. A *duplaszázas* esetén a rendszer éppen aktuális állapotát az OnlineRG írja le. Az offline algoritmus lefutásakor az offlineRG SC-jeit próbáljuk összefésülni az onlineRG-ben lévő állapotokkal. Ha ez sikerül akkor az onlineRG tartalmazza az optimalizált és az online által annak indulása óta lerakott SC-ket. Mivel mindkét algoritmus a teljes topológiát látja, így az összefésülés nem mindig lehetséges, legrosszabb esetben előfordulhat, hogy egy hoszt, vagy egy link 200%-os terhelést kap.

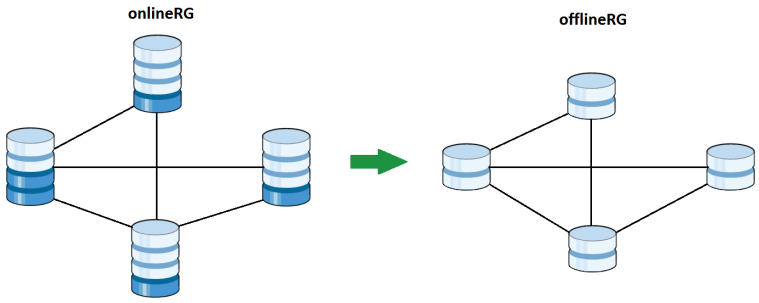
A dinamikus működésnél próbáltuk az előző rendszer előnyeit megtartani és a hátrányokat pedig minimalizálni. Itt az online algoritmus a topológia kapacitásainak 100%-át látja, hogy ne fordulhasson elő SC kérés eldobása olyan okok miatt, hogy az online program kevesebb erőforrást lát, mint amennyi valóban rendelkezésre áll. Az offline algoritmusnak azonban minden indulása előtt ki kell számolni, hogy mennyi szabad kapacitást lásson. Ezt úgy határozzuk meg, hogy vesszük az eredeti topológia minden paraméterének aktuális foglaltságának maximumát és a kapott kapacitás értékekkel létrehozzuk az offlineRG-t, ahogy a 5. ábrán láthatjuk, de nem csak az egyes hosztok terheltségét vesszük figyelembe, hanem a linkekét is. Az összefésülés nagy eséllyel sikeres lesz, mivel az onlineRG-ben lévő foglalt kapacitások és az offlineRG kapacitásainak összege kis mértékben haladja meg az eredeti rendszer 100%-át. Így csak a leginkább terhelt és legrosszabb esetekben fordulhat elő az összefésüléses hiba.

3.3. Erőforrásgráfok különbségképzése és összefűzése

Bár az online algoritmus működéséből kifolyólag elkeni az egyes hosztok terhelését CPU felhasználás szempontjából [11], de ez persze közel sem jelenti azt, hogy minden hoszton egyforma terhelés van. Ezért amikor kiszámoljuk, hogy mekkora offlineRG-ot adjunk át az offline algoritmusnak (pontosabban az abban szereplő

hosztok meghatározásakor), akkor a paraméterek mindegyikére meghatározzuk a terhelési maximumot a rendszerben. Azon link kapacitás, CPU, memória és tárhely adatokkal hozzuk létre az offline erőforrásgráfját, amelyik a legjobban van terhelve az online leképezések után[5]. Így bár látszólag több erőforrást adunk át az offline-nak mint amennyi elég lenne, de ezzel biztosan sikeres lesz a leképzés és van mozgástere az offline algoritmusnak, hogy javítson az optimalizálás előtti megoldáson.

A 5. ábrán látható, hogy az onlineRG legterheltebb hosztja határozza meg az offlineRG-ben lévő hosztok méretét. Az online algoritmus a kéréseket összesen 5 egységnyi erőforrásra tudta leképezni, az offline-nak pedig összesen 8 egységnyi helye van ugyanarra a feladatra. Ez látszólag pazarlás, de **a)** így biztosan tud javítani a leképezésen az offline algoritmus (link terhelés javítása érdekében lehet, hogy csak pár hosztra képzi le az összes kérést) **b)** a fel nem használt erőforrás úgy is visszakerül az online rendelkezése alá és **c)** az online algoritmus elkeni a CPU terhelést az erőforrásgráfon, így a legterheltebb hoszt és a legkevésbé terhelt között kicsi a különbség.



5. ábra. OfflineRG meghatározása az offline algoritmus indulásakor

[5]Munkánk során olyan erőforrásgráfokkal dolgoztunk, ahol a hosztok homogének.

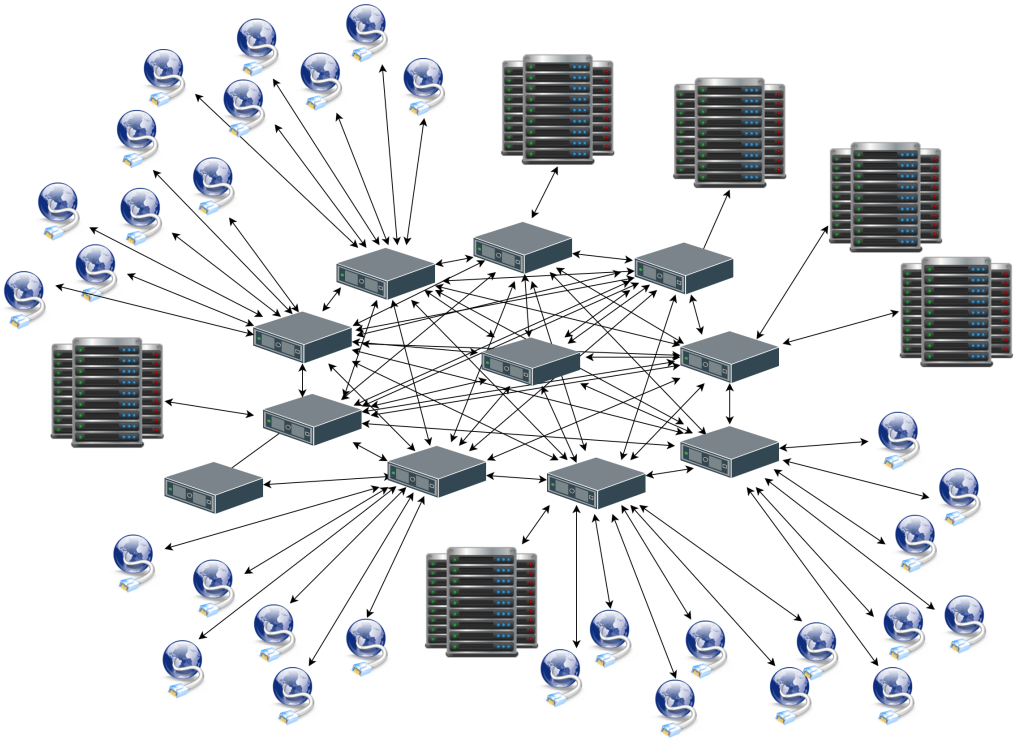
4. Szimulációs eredmények a hibrid megoldások értékelésére

Triviálisan látható, hogy a hibrid alkalmazások legrosszabb esetben is ugyanazzal a teljesítménnyel működnek, mintha a hálózati operátor pusztán az online algoritmust futtatná. Ebből következtetve, akkor fog a hibrid megvalósítás hatékonyabb leképzést elérni, ha valamilyen ok miatt az alkalmazott online algoritmus nem képes az optimálishoz közeli működésére. Célunk az volt, hogy a vizsgálati elrendezésekben megközelítsük hálózatunk maximális kihasználtságát és ebben a közel telített állapotban vizsgáljuk a sikeresen leképzett és kiszolgált kérések számát. A tesztjeinket egy python-ban implementált szimulátor segítségével futtatuk, melyben a már korábban bevezetett online algoritmust és a MILP alapú offline algoritmust használtuk.

4.1. A szimuláció során használt erőforrásgráf

A szimulációk során olyan erőforrásgráfot használtunk, amelynek topológiai felépítése (6. ábra) egy létező hálózaton alapul [12] [2], de a hálózatban szereplő switch-ek, hosztok és linkek paramétereit mi állítottuk be. A switch-eknek és a linkeknek csak sávszélesség (bandwith) és késleltetés (delay) paraméterei vannak. Egy hálózatban többfajta linket is fel lehet venni. Egy hoszt a következő paraméterekkel rendelkezik: processzor magok száma (CPU), operatív tár (memória), háttértár, késleltetés és adatátviteli sebesség. A szimulált hálózatunk felépítése egy gerinchálózatot szemléltet, ahol a SAP-ok közvetlenül a gerinchálózatra csatlakoznak: ezek jelentik az adott gerinchálózaton működő szolgáltatók be- és kilépési pontjait. A gráfunk összesen 17 csomópontból áll, amiből 6 darab hoszt van, a többi switchként üzemel. A SAP-ok összesen 5 darab switch-re csatlakoznak. Egy-egy switch-re összesen 6 SAP csatlakozik. Dolgozatunkban a szimulált hálózatban minden hosztra úgy tekintünk mint egy-egy teljes adatközpontra, amely a gerinchálózatra csatlakozik.

Továbbá a hosztoknak van egy úgynevezett `nf_type` paramétere is. Ezzel azt



6. ábra. A szimulációk erőforrás topológiája

tudjuk modellezni, hogy nem minden hosztra vagyunk képesek leképezni egy bizonyos típusú VNF-et. Ez a valóságban értelmezhető úgy például, hogy az egyik hoszt csak docker konténerek, míg a másik KVM-es virtuális gépek futtatására képes. További értelmezés lehet például, hogy az `nf_type` paraméterek biztonsági kérdéseket jelentenek, amiknek az adatközpont megfelel vagy sem. Következőképpen működik a használata: amikor egy VNF-et generálunk akkor sorsolunk neki egy értéket 1 és 10 között. Ez fogja jelölni az ő típusát. Az egyes hosztok az összes 10 típus közül csak 6 fajta VNF-et képesek leképezni, persze mindegyik más hat típust. Így a leképzés során az algoritmusnak azt is figyelembe kell vennie, hogy az egyes hosztok milyen VNF-eket tudnak fogadni.

4.2. A szimuláció során használt szolgáltatásgráf

Szimulációs környezetben a kérések generálását is meg kellett oldani. Törekedtünk itt is egy valós vagy jövőbeli rendszer igényeit modellezni. A dolgozatunkban

már több helyen említettük, hogy az egyes kérések milyen igényekkel rendelkezhetnek, de itt szeretnénk összegezni és kibővíteni ezeket az információkat. Minden általunk generált SC maximum nyolc VNF láncolatából áll (ezt véletlenszerűen sorsoljuk ki). Az egyes VNF-eknek processzor, operatív tár és háttértár igénye van, továbbá egy típusa, aminek a szerepét az előző fejezetben részleteztük.

Minden SC-hez tartozik továbbá egy sávszélesség és egy végpont-végpont késleltetés. A sávszélesség igényt úgy kell értelmezni, hogy az erőforrásgráf azon linkjein és hosztjain amin áthalad az adott SC (vagyis amelyeket a leképzés érint) levonásra kerül az elérhető sávszélesség értékből a SC sávszélesség igénye. A késleltetés igény nem erőforrásként használunk, mert feltételeztük, hogy az egyes hosztok és linkek késleltetése nem növekszik, akkor ha leképzünk rá egy SC-t. Így ezt csak mint feltételként adjuk meg, ha egy lehetséges leképzésben az erőforrások késleltetéseinek összege nagyobb mint a kért érték, akkor oda nem képezhetjük le a SC-t, keresnünk kell egy másik utat a két SAP között. Ha nem találunk ilyet akkor eldobjuk a kérést.

	Intervallumok
SC hossza:	1 - 8
Sávszélesség:	1 - 20 Mb/s
CPU magok:	1 - 4
Memória:	160 - 1600 MB
Háttértár:	300MB - 3000MB
Késleltetés:	60ms - 220ms

2. táblázat. SC lehetséges értékei

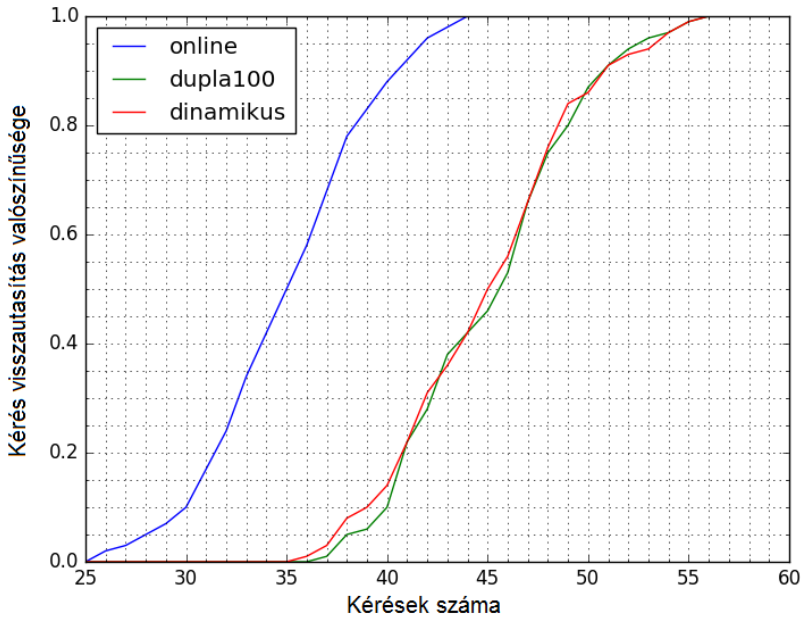
Mivel jelenleg még csak teszthálózatok vannak NFV-re, pontos és a való életből vett adatokkal nem rendelkeztünk arról, hogy az egyes SC-k milyen követelményeket támaszthatnak a hálózatokkal szemben. Ebből kifolyólag arra jutottunk, hogy előre meghatározott intervallumokon belül random választjuk ki a szolgáltatási utak követelményeit: véletlenszerű SC-ket tartalmazó kéréssorozatokat generáltunk. Ezeket különböző úgynevezett `request_seed` értékekkel láttuk el, amelyek segítségével akár többször is visszajátszható volt egy-egy konkrét szimuláció a kívánt kéréssorozattal.

Egy SC különböző paramétereit a 2. táblázatban látott intervallumokból sorsoltuk ki egyenletes valószínűséggel.

Mint látható, a generált SC-k igényei széles skálát lefednek. Az így kisorsolt kérések jelenthetnek például egy fullHD vidó stream-elését (8-12Mbps), vagy egy élő VoIP szolgáltatást (60ms latency), de akár egy egyszerű alacsony sávszélesség-, és magas késleltetésigényű szolgáltatást is.

4.3. Stressztesz

A szimulációk során elsőnek is stressztesztet hajtottunk végre, vagyis arra voltunk kíváncsiak, hogy mikor éri el az adott algoritmus a hálózat telítettségét. Összesen 100 kérést küldtünk másodpercenként egyesével az online és a hibrid algoritmusnak és azt vizsgáltuk, hogy a bejövő 100 kérésből hányat tud kiszolgálni az éppen futó algoritmus.



7. ábra. Online, Duplaszázas és Dinamikus Hibrid algoritmusok futásának eredményei többszöri (100 db) szimuláció után. X tengelyen a sikeresen leképzett kérések száma, Y tengelyen az adott számú kérés eldobásának tapasztalati valószínűsége.

A stressszteszt eredményei a 7. ábrán láthatóak. A 44 darab kérés sikeres leképzésére a sima online algoritmus futásakor már nincs lehetőség, 100%-os a ki nem szolgálásra az esély. Ezzel szemben a két hibrid megvalósításnál a 44 SC leképzési igény sikeres kiszolgálására majdnem 60%-os esélyünk van (az ábrán 42%-os dobási esély szerepel). Szintén látható a *duplaszázas* és a *dinamikus* hibrid algoritmusok működése közti különbség: mivel a duplaszáz esetében az offline algoritmus a hálózat teljes erőforráskészletét látja, képes optimális leképzést megvalósítani több kérés esetén is. Ez a 7.ábrán az X tengely 35 és 40 értékei között figyelhető meg. A duplaszázas görbe alacsonyabban van mint a dinamikus hibridé. Ez azt jelenti, hogy a duplaszázas módszer jobb megoldást ad a kérések leképzésére, vagyis 40 (és annál kevesebb) kérés sikeres megvalósítására nagyobb az esély mint a dinamikus hibrid esetén.

3. táblázat. Duplaszázas és Dinamikus Hibrid algoritmus javításai az Online algoritmushoz képest

	Maximális javulás	**Minimális javulás**	**Átlagos javulás**
Duplaszázas	20%	2%	10.2%
Dinamikus	19%	0%	10%

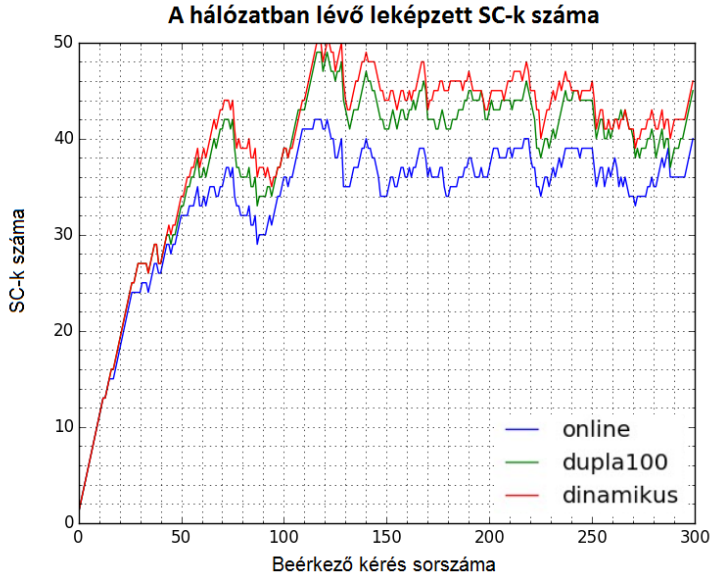
A 3. táblázatban látható, hogy különböző a hibrid megoldások hány százalékkal tudtak több kérést kiszolgálni mint az online algoritmus. Érdekesség, hogy a duplaszázas hibrid program általában jobb javítási eredményeket ért el a dinamikus megoldásnál, vagyis a stressszteszt során hatásosabbnak bizonyult.

4.4. Tesztek megszűnő szolgáltatásokkal

Az előző fejezetben bemutatott tesztelés során azt feltételeztük, hogy ha egy kérést leképeztünk, akkor az onnantól kezdve soha sem fog megszűnni, ezzel egy idő után mindenképp túlterheljük a rendszert és láthatjuk, hogy mikor éri el a hálózat a telített állapotát. Ez persze nem az általános felhasználás, de arra tökéletesen megfelel, hogy összehasonlítsuk egymással az algoritmusokat. Az általános működés szimulálására egy olyan funkciót is beépítettünk a szimulátorba, ahol a kérések érekezésének gyakoriságával és megszűnésével is számolunk. A szimulációkhoz az

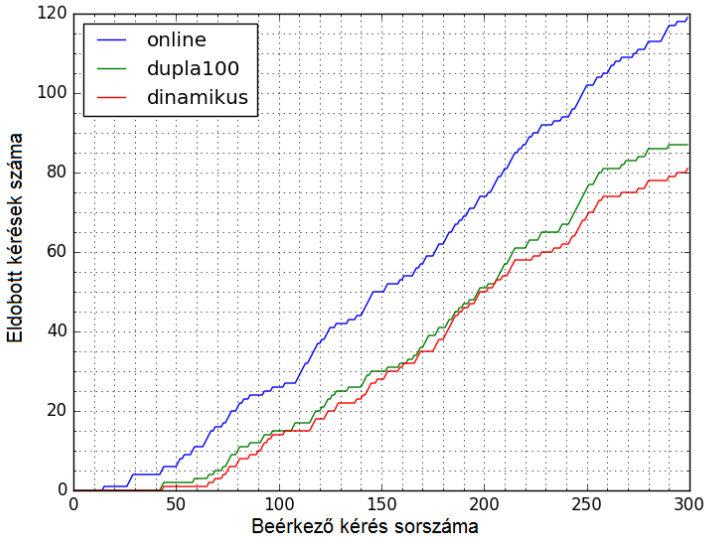
alábbi modellt alkalmaztuk: a szolgáltatás kérések egy Poission folyamattal érkeznek, λ paraméterrel, és egy μ paraméterű exponenciális eloszlású ideig tartanak. Annak érdekében, hogy a rendszer egyensúlyi állapotban legyen (tehát a kihasználtság a maximum közelében ingadozzon, de ne legyen túlterhelt), a szolgáltatások időtartalmának μ paraméterét úgy választottuk meg, mintha egy $M/M/\infty$ sort feltételeznénk, és a rendszerben lévő átlagos kérések száma 45 lenne. A λ értéket pedig azért választottuk 1/15-re, hogy a offline algoritmusnak legyenek olyan periódusai, amikor több kérés is érkezik be a futása alatt, illetve olyan is, hogy nem érkezik kérés. Így tehát a μ értékére a 1/15/45 lett állítva.

A fenti beállításoknak köszönhetően a rendszerbe átlagosan 15 másodpercenként érkeznek az SC leképzési kérések. Fontos kiemelni, hogy ez az érték átlagos, vagyis előfordultak olyan börsztös esetek is, mikor egyszerre több kérés érkezett, de az sem volt lehetetlen, hogy két bejövő kérés között több mint 15 másodperc teljen el. Továbbá a rendszerben átlagosan 45 darab leképzett SC helyezkedik el. Ennek a beállításnak az az oka, hogy a stressztesztek során tapasztaltuk, hogy az online algoritmus átlagosan 35 kérést volt képes leképezni a 100-ból. Mivel az átlagos javulás 10%-ot mutatott így kaptuk meg a rendszerben lévő kérések kívánt számát, a 45-öt.



8. ábra. A rendszerben lévő sikeresen leképzett szolgáltatás láncok száma

A kapott eredmény a 8. ábrán látható. Az X tengelyen az éppen beérkező kérés sorszámát láthatjuk, míg az Y tengely az RG-re sikeresen leképzett kérések számát jeleníti meg. Az 50. kérés érkezése után tapasztalható, hogy a hibrid rendszerek eltérnek a sima online működéstől és egyszerre több állapot van leképezve. Mivel az SC-k halálozásai mind a három lefutás esetén azonosak, így ez csak úgy lehetséges, hogy az érkező kérésekből a hibrid többet ki tud szolgálni. Látható, hogy a két hibrid algoritmus jobban közelíti a beállított paraméterek szerinti működést, vagyis hogy egyszerre átlagosan 45 leképzett SC legyen a hálózatban. Érdekesség, hogy a stressztesttel ellentétben itt a pirossal jelzett dinamikus algoritmus hatásosabb működést képes biztosítani a duplaszázasnál, ebből következtetve kijelenthető, hogy a stressztesztnél életszerűbb működés során valószínűleg többször fordul elő az összefésülési hiba a duplaszázasnál. Ekkor az aktuális javítási ciklus optimális értékei nem kerülnek érvényesítésre, így lehet az, hogy rosszabb teljesítményre képes a dinamikus megvalósításnál, ahol az összefésülési hibár kevesebb az esély.



9. ábra. A kérések során nem teljesített leképzések száma

A 9. ábrán a fentebb leírt teszt során eldobott SC-k számát láthatjuk a beérkező kérések függvényében. Ha egy görbe vízszintesen halad, az azt jelenti, hogy az X tengelyen kiolvasott szolgáltatási lánc sikeresen leképzésre került és az eldobott

kérések száma nem növekedett. Megfigyelhető, hogy a görbék egy tölcsért rajzolnak le egymáshoz képest. Ez azt jelenti, hogy a teszt futásának elején még nincs nagy különbség a különböző algoritmusok között az eldobott SC-k számát illetően, azonban ahogy haladunk tovább az időben egyre nagyobb hátránya lesz az online algoritmusnak a hibridekkel szemben. A tölcsér elején, mikor a hibrid algoritmusok is elkezdenek kéréseket eldobni (a 40. beérkező SC körül), az offline még csak 4-5 szolgáltatáslánccal tudott kevesebbet leképezni, mint duplaszázas és a dinamikus algoritmus. Azonban ahogy haladunk tovább a 300. leképzési igénynél már 40-nel több kéréssel dobott el többet az eredeti online algoritmus az általunk létrehozott dinamikus megvalósítással szemben, vagyis ez a hibrid módszer több mint **33%-os javulást** mutatott az eldobott kérések számát illetően.

5. Értékelés

A dolgozatunkban bemutatásra került egy általunk létrehozott új erőforrásvezénylési algoritmus, mely vegyíti az irodalomban megjelent eddigi megoldások legjobb tulajdonságait. Az algoritmus képes a bejövő kéréseket valós időben leképezni és közben bizonyos időnként optimalizálni az aktuális állapotot, melynek köszönhetően a rendszerben lévő SC-k leképzése jobban közelít az ideálishoz. A javasolt eljárást többféle szimulációs környezetben teszteltük, melyek során minden esetben jobb eredményt értünk el.

Az általunk javasolt hibrid erőforrásvezénylési eljárással elérhető átlagosan 10%-os kihasználtság javulás könnyen kiszámolható gazdasági hasznot nyújt. A a beérkező igények kiszolgálása nem késlekedik, ám az offline algoritmus által kiszámolt optimális elrendezés megkövetelheti, hogy az előzőleg az online algoritmussal leképzett szolgáltatásláncokat átmigráljuk más erőforrásokra. A migrálás költsége ugyanakkor eltörpül a folyamatosan 10%-kal több igény kiszolgálásával termelt pénzügyi haszonnövekedéshez képest. A szimuláció mindössze egyetlen csomópontjának számítási és hálózati kapacitásainak 10%-a az Amazon EC2-től vásárolva óránként 100$-ba kerülne[6].

A továbbiakban célunk egyéb heurisztikával rendelkező online algoritmusok használatával megvalósított hibrid alkalmazások elkészítése, tesztelés és összehasonlítása a mostani megoldásunkkal. Ezen kívül további tervünk, hogy a szimulációk helyett valós, kiépített hálózatokon teszteljük a hibrid működést, összehasonlítva a többi elérhető technológiával.

[6] Az erőforrások 10%-a megfelel 960 CPU-nak, 2,5TB memóriának, 50TB háttértárnak és 1Gbps be- és kimeneti forgalomnak. https://aws.amazon.com/ec2/pricing/on-demand/

Hivatkozások

[1] Gurobi Optimization. `Mentve:http://www.gurobi.com/index`, (2016.10.22).

[2] SND Library. `Mentve:http://sndlib.zib.de`, (2016.10.24).

[3] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. K. Soong, and J. C. Zhang. What will 5g be? *IEEE Journal on Selected Areas in Communications*, 32(6):1065–1082, June 2014.

[4] Z. Cai, F. Liu, N. Xiao, Q. Liu, and Z. Wang. Virtual network embedding for evolving networks. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, Dec 2010.

[5] NM Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In *INFOCOM 2009, IEEE*, pages 783–791. IEEE, 2009.

[6] Chunfeng Cui, Hui Deng, Uwe Michel, and Herbert Damker. Network functions virtualisation. ETSI, Deutsche Telekom, 2012.

[7] ETSI GS NFV-PER 001, Dec 2014. *Network Functions Virtualisation (NFV); Architectural Framework*. ETSI, 2014 Dec.

[8] ETSI GS NFV-PER 002. *Network Functions Virtualisation (NFV); NFV Performance & Portability Best Practises* . ETSI, 2013 Okt.

[9] Wendy Myrvold and William Kocay. Errors in graph embedding algorithms. *Journal of Computer and System Sciences*, 77(2):430–438, 2011.

[10] Balázs Németh, János Czentye, and Balázs Sonkoly. Network Function Forwarding Graph API documentation. Technical report, 2016.03.01.

[11] Balázs Németh, Balázs Sonkoly, Matthias Rost, and Stefan Schmid. Efficient Service Graph Embedding: A Practical Approach. In *Second IEEE International Workshop on Orchestration for Software Defined Infrastructures (O4SDI @ IEEE NFV-SDN 2016)*, Nov 2016.

[12] Sebastian Orlowski, Roland Wessäly, Michal Pióro, and Artur Tomaszewski. Sndlib 1.0—survivable network design library. *Networks*, 55(3):276–286, 2010.

[13] Ashraf A Shahin. Virtual network embedding algorithms based on best-fit subgraph detection. *arXiv preprint arXiv:1502.02768*, 2015.

[14] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.