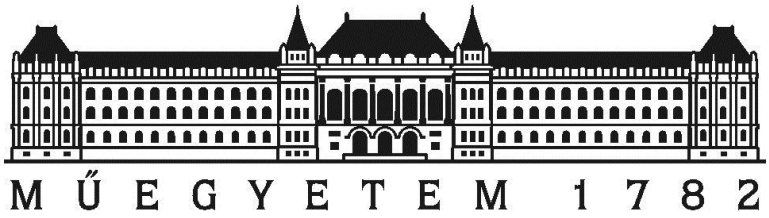


Gyors IP forgalomtovábbítás tömörített prefix fákkal

Készítette:

Mihálka Bence

IX. évfolyam, műszaki informatika, bence.mihalka@gmail.com

Konzulensek:

Dr. Rétvári Gábor, TMIT, retvari@tmit.bme.hu

Csernátony Zoltán, TMIT, csernatony@tmit.bme.hu

Budapest, 2012

Kivonat

Az elmúlt években az IP útvonalválasztók forgalomtovábbítási tábláinak (FIB) mérete, dinamikája és terhelése jelentősen megnőtt. Ennek okai a világhálózat növekedése, a mobilkommunikáció rohamos terjedése, illetve az IPv6 térhódítása. A mai IP eszközökben legelterjedtebben használt FIB adatstruktúra a prefix fa, mely másodpercenként több millió komplex lekérdezést illetve módosítási műveletet képes végrehajtani, azonban a memóriaigénye is jelentős. Ennek csökkentésére célszerű megoldás lehet a prefix fát valamilyen algoritmus alapján betömöríteni.

Munkám során megvizsgáltam több lehetséges tömörítési algoritmust, ezeken egy általam készített szimulációs környezetben méréseket végeztem, különös tekintettel a tömörítési hatékonyságukra és a futási idejükre. Ezen eredmények alapján a megvizsgált tömörítési módszerek közül az általam továbbfejlesztett prefix fa hajtogatás nevű algoritmus bizonyult a leghatékonyabbnak. Ennek működéséhez egy jelentős memóriaigényű segéd-adatszerkezetre, úgynevezett részfaindexre van szükség, aminek tárolására számos különböző (vektoron, piros-fekete fán, hash-táblán, illetve fix méretű hash táblán alapuló) módszert alkottam meg, majd ezek összehasonlítása céljából hatékonysági méréseket végeztem, legfőképpen a felhasznált memória mennyiségét, és a kiszolgálási idejét vizsgálva. Az eredmények alapján arra a következtetésre jutottam, hogy a prefix fa hajtogatás algoritmus fix méretű hash táblán alapuló részfaindex-szel megfelelő jelölt lehet arra, hogy az útvonalválasztók által kezelt forgalomtovábbítási táblákat a jelenleginél sokkal tömörebben tároljuk, és ezzel hatékonyabbá tegyük az IP forgalomtovábbítás feladatát.

Abstract

In recent years the forwarding information base (FIB) of IP routers has grown significantly in size, load, and dynamic properties. The cause of this is the rapid growth of the Internet, the widespread use of mobile communication, and the spreading of IPv6. In modern IP routers the most commonly used data structure for storing FIB is the prefix tree. It can handle millions of complex queries and modifications per second, but it also has a high memory requisite. In order to decrease the size of memory needed to store the FIB, it is a good idea to compress the prefix tree with an adequate algorithm.

In my paper I will examine several possible compression algorithms, and make measurements in a simulator I created, specifically focusing on the efficiency of the compression and the required time to carry out the compression. Based on these results I will stipulate that the so-called trie folding algorithm is the most efficient compression method, but it also needs an auxiliary data structure, called subtree index, which needs a significant amount of memory. I created several methods to store this data structure (based on vector, search tree, hash table, and fixed size hash table), and compared them in the simulator, regarding their memory requisites and their serving time. Analyzing the results I will reach the conclusion that the trie folding algorithm with fixed hash table based subtree index is a good candidate for storing FIBs in a more compressed way, making the task of IP forwarding more efficient.

Tartalom

1. Bevezetés ..... - 4 -
2. Alapfogalmak ..... - 7 -
2.1 FIB adatstruktúrák ..... - 7 -
2.2 Bináris prefix fa ..... - 8 -
2.3 Komplett bináris prefix fa ..... - 9 -
2.4 Optimális Forgalomtovábbítási Tábla Tömörítő algoritmus ..... - 10 -
2.5 Prefix fa hajtogatás ..... - 12 -
3. Prefix fa hajtogatás részfaindexek ..... - 16 -
3.1 Vektor alapú részfaindex ..... - 16 -
3.2 Piros-fekete fa alapú részfaindex ..... - 17 -
3.3 Hash tábla alapú részfaindex ..... - 18 -
3.4 Fix méretű hash tábla alapú részfaindex ..... - 19 -
3.5 Részfaindexek összefoglalása ..... - 21 -
4. A szimulációs környezet ismertetése ..... - 22 -
4.1 Szimulátor ..... - 22 -
4.2 Osztálydiagram, fontosabb osztályok bemutatása ..... - 23 -
4.3 FIB és leszármazott osztályai ..... - 25 -
4.4 Tömörítési algoritmusok ..... - 25 -
4.5 Részfaindexek ..... - 26 -
4.5.1 Vektor alapú részfaindex ..... - 26 -
4.5.2 Piros-fekete fa alapú részfaindex ..... - 27 -
4.5.3 Hash tábla alapú részfaindex ..... - 27 -
4.5.4 Fix méretű hash tábla alapú részfaindek ..... - 27 -
4.6 Egyéb megfontolások ..... - 27 -
5. Teljesítményvizsgálat ..... - 28 -
5.1 A kiértékelés alapjai ..... - 28 -
5.2 Tömörítési algoritmusok teljesítménye ..... - 29 -
5.2.1 A tömörítés hatékonysága ..... - 29 -
5.2.2 A tömörítési algoritmusok futási ideje ..... - 31 -
5.3 A részfaindex adatstruktúra hatékonysága ..... - 33 -
5.3.1 A részfaindexek tárhelyigénye ..... - 33 -
5.3.2 A részfaindexek kiszolgálási ideje ..... - 36 -
5.4 Az eredmények értékelése ..... - 39 -
6. Konklúziók, jövőbeli tervek ..... - 40 -
7. Irodalomjegyzék ..... - 42 -

1. Bevezetés

Az Internet rohamos terjedése napjaink egyik legmeghatározóbb jelensége, amely óriási hatással van társadalmunkra, szociális kapcsolatainkra, kommunikációs szokásainkra, vagyis röviden a mindennapi életünkre. De míg egy átlagos felhasználó egyre inkább magától értetődőnek tekinti a mindenhol elérhető gyors internetkapcsolatot, addig ugyanez a folyamat súlyos problémákat, pontosabban megoldandó feladatokat ró az ezzel foglalkozó mérnökök, tudósok vállára. Az Internet menedzselésének, bővítésének, fejlesztésének feladata rendkívül sokrétű, hiszen maga az Internet is egy csodálatosan komplex rendszert alkot. Jelen dolgozatomban az egyik leginkább égető kérdésre, [1] az útvonalválasztók (routerek) forgalomtovábbítási tábláinak (Forwarding Information Base, FIB) méretét kezelő algoritmusokra mutatok be lehetséges megoldásokat. Ezeket a táblázatokat az útvonalválasztók tartják karban, azzal a céllal, hogy a beérkező adatcsomagokat a táblázat alapján továbbíthassák a megfelelő irányba.

Az Internet útvonalválasztók forgalomtovábbítási tábláinak mérete riasztó mértékben növekedett az utóbbi években. [2][3][4] Ennek legfőbb okai a következők:

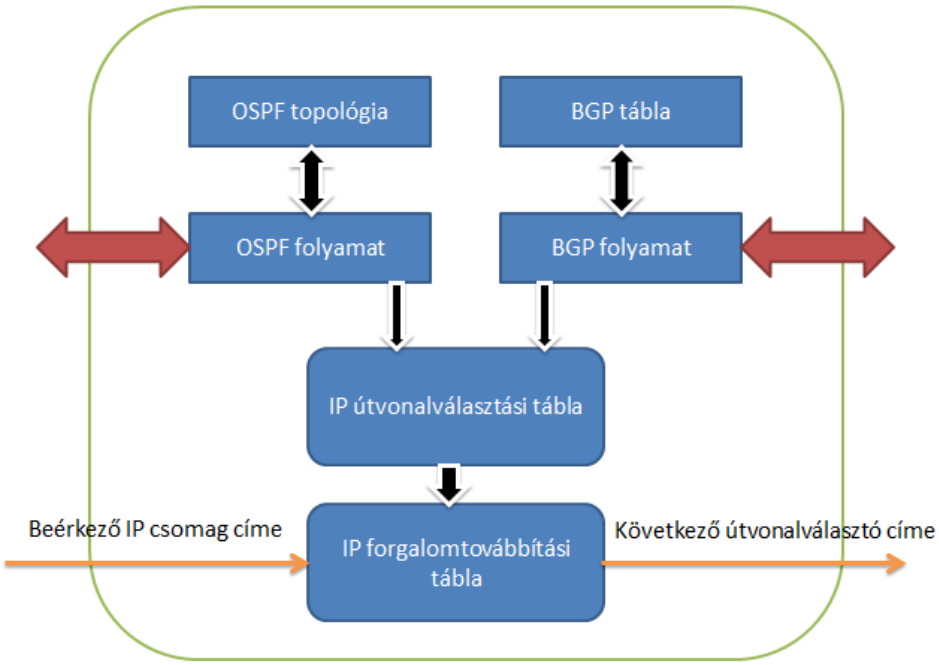
- a hálózat egyre növekvő méreteket ölt, ezért egyre több útvonalválasztóra van szükség.
- egyre több a szolgáltató-független címtartomány (provider-independent address). Ezek olyan címtartományok, amelyeket nem a szolgáltató nagyobb címteréből hasítanak le, hanem attól független, így nem aggregálható a szolgáltató címtartományába, hanem attól külön kell hirdetni, és így külön feltűnik a FIB-ben is. Ennek segítségével lehetővé válik az IP címek újraállítása nélküli szolgáltatóváltás.
- multi-homing: egy adott internetszolgáltató (Internet Service Provider, ISP) hálózata két szolgáltatón keresztül is elérhető, ilyenkor legalább az egyik szolgáltató címterébe nem aggregálható az adott ISP prefixe, ezért ez újabb FIB bejegyzéseket eredményez.
- az IPv4 címtartomány kimerült, a meglévő címtér másodlagos kereskedelme pedig felaprózza a címteret, ami nem teszi lehetővé a cím aggregációt.

Úgy tűnik, hogy ez a gyors növekvés „lekörözi” az útvonalválasztók memória kapacitásának növekedését, főként a hálózati csatolókártyákban használt memóriára nézve, amit a gyors IP cím kereséshez használnak. Ezen felül az internetszolgáltatókat arra készteti, hogy egyre gyorsabban cseréljék az útvonalválasztóikat, ami jelentős többletköltséggel jár, illetve kihatással van az energiafogyasztásra és környezetvédelmi megfontolásokra, ezért előtérbe helyezi a gyors IP cím keresések szükségességét.

A probléma kezelésére hosszú távon alapvető architekturális változtatásokra van szükség. Ugyanakkor az ilyen változtatások jelentős időt igényelhetnek, mint azt például az IPv6 hosszadalmas bevezetésekor is megtapasztalhattuk. Ezért nagy jelentősége van olyan rövid távú megoldásoknak, amik nem járnak architekturális változtatásokkal, de segítenek időt nyerni, mivel a probléma már ma is

fennáll, és nem lehet figyelmen kívül hagyni.

Ez a gyakorlatban annyit jelent, hogy olyan adatstruktúrákra van szükségünk, amikben az IP cím keresése nagyon gyorsan végrehajtható, vagyis azt várjuk az útvonalválasztótól, hogy a hozzá beérkező csomag fejlécében lévő cél IP cím alapján minél kisebb idő alatt képes legyen meghatározni a csomag útvonala mentén következő útvonalválasztó címét (a továbbiakban: next hop). Ezen felül elvárás az adatstruktúrával szemben, hogy dinamikus legyen, vagyis támogassa a különböző routing protokollok (pl. BGP, Border Gateway Protocol) használata során érkező útvonal-információk felvételét/módosítását/törlését egy már létező táblában. Egy útvonalválasztó egyszerűsített működési sémáját az 1. ábra szemlélteti. Ebben az eszközök egymás közt felépítik a kapocsolataikat az OSPF (Open Shortest Path First) és BGP (Boarder Gateway Protocol) alapján. Ezen információkat az IP útvonalválasztási tábla (Routing Information Base, RIB) tárolja, ami alapján az útvonalválasztó hálózati csatolókártyáiban eltárolt forgalomtovábbítási tábla (Forwarding Information Base, FIB) bejegyzései készülnek el. Ezen táblát használja az útvonalválasztó annak meghatározására, hogy egy bejövő IP csomag fejlécében szereplő cél IP cím alapján merre, melyik szomszédja felé továbbítsa a csomagot.



1. ábra: Egyszerűsített útvonalválasztó architektúra

Ezek mellett a kritériumok mellett pedig azt is el szeretnénk érni, hogy a forgalomtovábbítási tábla a lehető legkevesebb memóriát használja fel. Ezen feladat neve a FIB tömörítés,[5] amely egy olyan algoritmus, amely standard FIB adatstruktúrából (például az elterjedten használt bináris prefix fából (lásd 2.2 fejezet)) egy "memóriatakarékosabb", de az eredetivel teljesen megegyező logikai működést biztosító új adatstruktúrát gyárt.

Nézzük meg részletesen, hogy milyen előnyökkel jár a memóriagazdaságosság elvének következetes érvényesítése! Erre a kérdésre többféle válasz is adható. Hasznos a kevés biten reprezentált forgalomtovábbítási tábla például azért, mert így egy ugyanakkora gyorsítótárban (cache) a tábla nagyobb hányada tud egyszerre jelen lenni, így a feldolgozás is hatékonyabb lesz. Ezenkívül, mivel a hálózati csatolókártyákban használt speciális, nagy sebességű memóriamodulok rendkívül drágák, a memóriaspórolás költséghatékony is. Továbbá problémát okoz az is, hogy az internetszolgáltatók nagyon gyakran kell, hogy cseréljék az útvonalválasztóikat, mivel a régebbi modellek memóriájában már nem férnek el az aktuális forgalomtovábbítási táblák. Erre is megoldást szolgáltat, ha kevesebb biten reprezentáljuk a táblákat, hiszen ez lassabb eszközbeszerzési ciklust eredményez, illetve akár már leselejtezett eszközök is újra munkára foghatóak.

Összességében tehát kijelenthető, hogy a FIB tömörítés problémakörének megoldása kulcsfontosságú lehet ahhoz, hogy az Internet jelen formájában fenntartható legyen, illetve továbbfejlődhessen, hiszen segítségével egyaránt nyerünk előnyöket rövid távon (gyorsuló forgalomtovábbítás) és hosszú távon is (útvonalválasztók hosszabb élettartama).

Feladatomként azt tűztem ki, hogy megismerjem a különböző tömörítési algoritmusokat, (komplett bináris prefix fa, optimális forgalomtovábbítási tábla tömörítés, prefix fa hajtogatás) összehasonlító elemzésnek vessem alá, és megállapítsam, hogy mely módszer(ek) használatával érhető el a legjobb tömörítési hatásfok. Ennek érdekében létrehoztam egy szimulációs környezetet, amely képes valódi forgalomtovábbítási táblákat feldolgozni a különböző tömörítési módszerekkel, és segítségével mérések végezhetőek, különös tekintettel a tömörítés hatásfokára, és az ehhez szükséges memória- és időigényre. Ennek eredményeként megállapítottam, hogy ezek közül a módszerek közül egyértelműen a prefix fa hajtogatás algoritmus a leghatékonyabb. [3][6][7] A prefix fa hajtogatás módszerének lényege, hogy a prefix fában található azonos részfákat csak egyszer tároljuk el, és erre az összes előfordulási helyéről egy gyermek mutatót irányítunk. Ehhez a feladathoz elengedhetetlenül szükséges, hogy az azonos részfákat nyilvántartsuk. Az ehhez kapcsolódó adatstruktúrának, az úgynevezett részfaindexnek nagy szerepe van az algoritmus hatékonyságában és futási idejében, ezért ezt a segéd-adatszerkezetet alaposabban tanulmányoztam, és létrehoztam négy, különböző tárolási elven alapuló megvalósítását. Ezeken a részfaindex verziókon a szimulációs környezetben méréseket végeztem, melynek eredményeként megállapítottam, hogy a fix méretű hash táblán alapuló részfaindexet felhasználó prefix fa hajtogatás módszere hatékony tömörítés elérésére képes mind memória-, mind időigény szempontjából.

A dolgozat további részében először ismertetem az általam megvizsgált tömörítési módszereket és adatszerkezeteket (2. és 3. fejezet), majd a szimulátociós környezet implementációjáról szóló leírás következik (4. fejezet). Ezután a mérések eredményei, értékelése következik (5. fejezet). Végül dolgozatomat a konklúziók levonásával, és a vonatkozó szakirodalom jegyzékével zárom (6. és 7. fejezet).

2. Alapfogalmak

Ebben a fejezetben bemutatom az útvonalválasztók által tárolt forgalomtovábbítási táblázatokat (FIB), szerepüket az IP útvonalválasztók működésében, és ismertetem a tömörítésüknek néhány lehetséges megoldását, úgymint a bináris prefix fa, a komplett bináris prefix fa, az optimális forgalomtovábbítási tábla tömörítés, és a prefix fa hajtogatás algoritmus.

2.1 FIB adatstruktúrák

A FIB adatstruktúra feladata az útvonalválasztóban annak meghatározása, hogy egy bejövő csomagot (cél IP címe alapján) melyik szomszédos útvonalválasztónak kell továbbküldeni, vagyis mi lesz a next hop értéke. Ehhez a legspecifikusabb bejegyzés megkeresésére van szükség (Longest Prefix Match) [8], ami megadja a FIB azon bejegyzését, amelynek IP címe a lehető legnagyobb értékű hálózati maszkot, vagyis az IP cím minél több bitjét figyelembe véve illeszkedik a csomag cél IP címére.

A 2. ábrán néhány FIB bejegyzés látható, a későbbiekben mindegyik tömörítési módszert ezen példán mutatom be. A táblázat egy sorát úgy kell értelmezni, hogy az IP címből (amely a táblázatban bináris számként szerepel) hálózati maszknyi bitet figyelembe véve a FIB-nek az adott next hop-ot kell eredményül adnia a legspecifikusabb bejegyzés keresése során. Például tegyük fel, hogy a kapott IP csomagban lévő célállomás IP címe a ’*010*’ bitekkel kezdődik. Erre az IP címre három bejegyzés is illeszkedik, az alapértelmezett útvonal (default route) ’*0/0 → 1*’, továbbá a ’*01/2 → 3*’ és a ’*010/3 → 2*’. Mivel ezek közül az utolsó a legspecifikusabb (legtöbb bitig meghatározott), ezért a keresés eredményeként a csomag a ’*2*’-es azonosítójú szomszéd felé kerül továbbításra.

IP cím	Hálózati maszk	Next hop
'0'	*0*	*1*
'00'	*2*	*1*
'01'	*2*	*3*
'010'	*3*	*2*
'110'	*3*	*1*
'111'	*3*	*3*
'1110'	*4*	*2*

2.ábra: Egy IP útvonalválasztási tábla

Ez a táblázatos formátum természetesen a gyakorlatban nem alkalmazható, hiszen ebben a keresés ideje *O(N)* nagyságrendű, a tárolásához szükséges méret pedig *O(N*log K)*, ahol *N* a bejegyzések száma, *K* pedig a next hop-ok száma. Az Internet architektúrájából eredően nem minden útvonalválasztó van összekötve mindegyik másikkal, ezért a gyakorlatban *K* sokkal kisebb, mint *N*. (*K* tipikus értékei a 2 és 20 közötti intervallumba esnek). Fontos megjegyezni, hogy a gyakorlatban természetesen a next hop az adott szomszédos útvonalválasztó IP címét tárolják, ehelyett most az egyszerűség kedvéért jelöltem őket egész számokkal.

2.2 Bináris prefix fa

Mindegyik bemutatandó tömörítési algoritmus kiindulási alapja a bináris prefix fa. Ez egy hagyományos bináris fa, melyben bármely elem tartalmazhat egy címkét, vagyis next hop információt, és minden elemnek nulla, egy, vagy kettő gyermeke lehet. A fát a következő módon építjük fel: a forgalomtovábbítási táblában egy bejegyzés általános formátuma:

```
[IP, hálózati maszk, next hop]
```

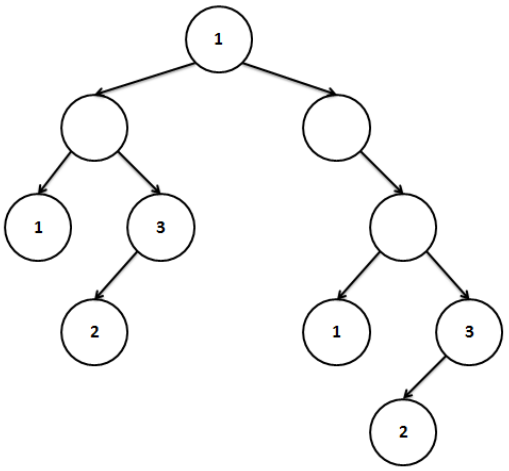
Egy ilyen bejegyzés érkezésekor az IP címet átalakítjuk 32 bites bináris számmá (természetesen IPv4 esetén), majd a bináris prefix fa gyökerétől indulva az IP cím ’*0*’ illetve ’*1*’ bitjei alapján lépünk a fában balra illetve jobbra. Tesszük mindezt olyan hosszan, amennyi a hálózati maszk értéke, majd az így eredményül kapott fa elem címkéjeként eltároljuk a next hop információt (lásd 3. ábra).

Ezzel a fában a gyökérhez közelebb levő elemek általánosabb, míg a levelekhez közelebb levő elemek speciálisabb, kevesebb IP címre illeszkedő bejegyzéseket fognak tartalmazni.

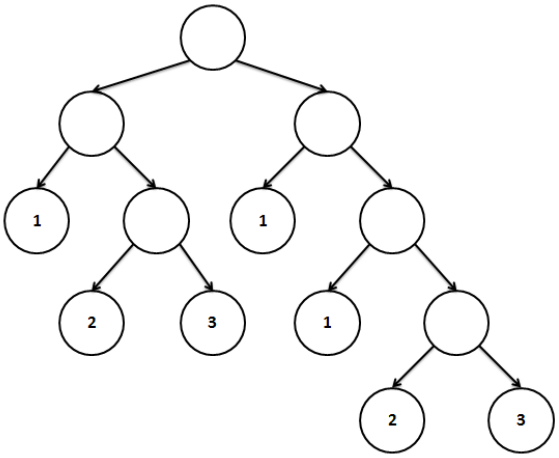
Ezután egy keresés úgy néz ki, hogy a beérkező csomag cél IP címe alapján lépkedünk a fában, és megkeressük a hozzá tartozó, a fában a lehető legmélyebben lévő, nemüres fa elemet (legspeciálisabb bejegyzés keresése), majd az abban található next hop-ra küldjük tovább a csomagot. Ehhez a bejárás során mindig meg kell jegyezni a legutoljára megtalált next hop értéket arra az esetre, ha már nem találunk ennél speciálisabb bejegyzést, és elérjük a fa legalsó szintjét.

Az előző pontban ismertetett példa (’*010*’ bitekkel kezdődő cél IP cím esetén) a bináris fában a legspeciálisabb bejegyzés keresése az alábbiak szerint történik: elindítjuk a keresést a bináris prefix fa gyökerénél. Itt találunk is egy érvényes bejegyzést, (’*1*’-es next hop), ezt megjegyezzük, majd továbblépünk a gyökér bal oldali elemére, hiszen az IP cím első bitje ’*0*’ volt. Itt nem találunk újabb bejegyzést, tehát még mindig a gyökérben talált érték a legspecifikusabb. Továbblépés után (az IP cím második ’*1*’ értékű bitje miatt) a ’*3*’-as értékű next hop-ot találjuk, ezzel felülírjuk az eddig megjegyzett értéket, majd ismét továbblépünk az IP cím harmadik bitje alapján. Itt megtaláljuk a ’*2*’-es számú next hop-ot, és mivel ezzel a bináris prefix fa legalsó szintjére értünk, ez lesz a legspeciálisabb bejegyzés keresésének eredménye, ami természetesen megegyezik az előző pontban talált eredménnyel.

A bináris prefix fa előnye az egyszerűsége, hiszen nagyon egyszerűen implementálható. Az adatstruktúra keresési ideje *O(log N) = O(W)* nagyságrendű, ahol *N* a bejegyzések száma, *W* a címben lévő bitek száma (IPv4 esetén 32), ugyanakkor tárolási igénye az üres fa elemek miatt valamivel nagyobb, mint a táblázatos megoldásé. De mivel a gyakorlatban a FIB által megoldandó feladatok túlnyomó többsége a legspeciálisabb bejegyzés keresése, ezért a bináris prefix fa jóval hatékonyabbnak értékelhető, mint a táblázatos megoldás.



3. ábra: Bináris prefix fa



4. ábra: Komplett bináris prefix fa

2.3 Komplett bináris prefix fa

Az első módszer, amely már várhatóan tömörítést is képes elérni, bár ennek mértéke nagyban függ a bemenet tulajdonságaitól, sőt bizonyos esetekben akár nőhet is a tárolási igény a bináris prefix fáéhoz képest. Ez az algoritmus két jelentős dologban tér el a bináris prefix fától: megszabjuk, hogy minden elemnek pontosan nulla, vagy kettő gyermeke lehet, és next hop információt csakis a nulla gyerekkel rendelkezők, vagyis csak a levelek tartalmazhatnak. (lásd 4. ábra) Amennyiben ezután az átalakítás után két levél-elem szülője, valamint a bennük tárolt next hop érték is azonos, akkor ezeket a leveleket megszüntethetjük, és a szülő elemben tároljuk el a közös next hop értéket. Ezzel a módszerrel nem érhető el különösebben jelentős tömörítés, de a fa mindenképpen strukturáltabb lesz az eljárás végén, illetve a használat közben az IP cím keresés során nem kell nyilvántartani a legutóbbi aktuális next hop-ot, hiszen ilyen érték mindenképp csak a levelekben érhető el.

Az 1. pszeudokód ezt az algoritmust valósítja meg. Ebben az első három sor az `inherited(N)` meghatározását írja le, amely megkeresi egy adott `N` elem felmenői közül az elsőt, amelyik rendelkezik next hop értékkel, és ezt adja értékül az `inherited(N)` függvénynek.Az ezután következő ciklus preorder módon hajtódik végre. A preorder bejárás egy olyan fabejárás, melynek során mélységi keresési sorrendben végiglátogatjuk a fa csúcsait, minden csúcson elvégzünk egy műveletet, majd rekurzívan folytatjuk a bejárást a csúcs gyermekein. A postorder bejárás esetében ezzel szemben egy csúcson csak akkor végezzük el az adott műveletet, ha már az összes gyermekét

bejártuk.Tehát az első bejárás során preorder módon megvizsgáljuk, hogy az adott elemnek mennyi gyermeke van, és ha csak egy, akkor létrehozunk mellé egy másodikat is. Emellett a next hop értékeket is terjesztjük lefelé: amennyiben az adott elemnek nincs next hop értéke, úgy a legközelebbi next hop-pal rendelkező felmenőjének next hopját kapja meg. A második bejárás során, amely postorder sorrendben hajtódik végre, minden elemre megvizsgáljuk, hogy az összes gyermekének ugyanaz-e a next hop értéke. Amennyiben igen, úgy nincs szükség az adott pont további kifejtésére, így gyermekei törölhetőek, az adott elem pedig megkapja a next hop értéket.

1. pszeudokód: Komplett bináris prefix fa

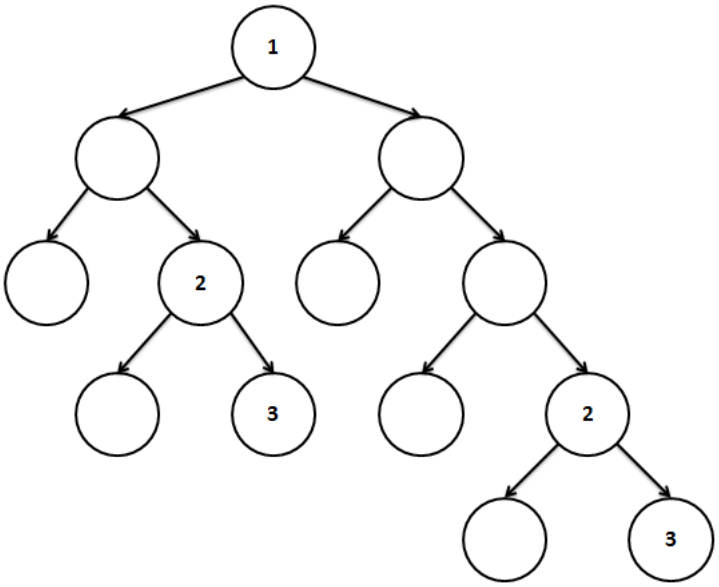
```
1: function inherited(N)
2:       if nexthop(parent(N)) != 0 then
3:            return nexthop(parent(N))
4:       else return inherited(parent(N))
5: for each node N (preorder traverse)
6:       if N has exactly one child node then
7:            create missing child node
8:       if N is leaf then
9:            if nexthop(N) = 0 then
10:                nexthop(N) = inherited(N)
11:           else
12:                nexthop(N) = NULL
13:for each node N (postorder traverse)
14:      children = getChildren(N)
15:      if nexthop of all children are equal (NH) then
16:           for each child C nexthop(C) = NULL
17:           nexthop(N) = NH
```

A beillesztés és a keresés műveletet hasonlóan történik, mint a bináris prefix fa esetén (hiszen a komplett bináris prefix fa ennek egy kiterjesztése), és a tárolási igényre és a beillesztés és a keresés futási idejére is hasonló megfontolások érvényesek. Bár az eredményül kapott fa már sokkal strukturáltabb, és bizonyos szintű tömörítést is el lehet érni vele, mégsem lehet túlságosan hatékonynak mondani, hiszen nem foglalkozik a benne eltárolt next hop értékek gyakoriságával, pusztán a bináris prefix fa „szabálytalanságait” küszöböli ki. Erre jelent megoldást a következő pontban bemutatott Optimális forgalomtovábbítási tábla tömörítő algoritmus (Optimal Routing Table Compression, ORTC). [9]

2.4 Optimális forgalomtovábbítási tábla tömörítő algoritmus

Ez egy olyan tömörítési módszer, amely egy bináris prefix fából előállít egy másik bináris prefix fát, amelyben a bejegyzések, vagyis a next hop-pal rendelkező fa-elemek száma a lehető legkisebb, méghozzá úgy, hogy az átalakított fa az eredetivel megegyező továbbítási logikát valósítson meg. Pontosabban megfogalmazva, ha kapunk egy forgalomtovábbítási táblát, amely információkat tárol az IP cím legspeciálisabb bejegyzése keresésének elvégzéséhez, akkor az algoritmus előállít egy új

forgalomtovábbítási táblát, amely a) ugyanazt a továbbítási logikát biztosítja, és b) a lehető legkevesebb bejegyzést tartalmazza (lásd 5. ábra). Az ORTC könnyedén és hatékonyan implementálható, és egy gerinchálózati routerben tárolt prefixek számát körülbelül 40%-kal képes csökkenteni. [9]



5. ábra:Az optimális forgalomtovábbítási tábla tömörítő algoritmus által tömörített fa

Az algoritmus négy fa bejárás alatt optimalizálja a routing táblát. (lásd 2. pszeudokód) A működéséhez szükséges a ’#’ művelet definiálása:

$$A \# B = A \cap B, \text{ ha } A \cap B \mathrel{!}= \emptyset,$$

$$A \# B = A \cup B, \text{ ha } A \cap B = \emptyset.$$

Ezen halmazművelet eredménye a két bemenő halmaz metszete, amennyiben az nem üres, és uniója, amennyiben a metszet az üres halmaz. Az első két bejárás során elkészítjük az eredeti bináris prefix fához tartozó komplett bináris prefix fát (3. sor). Ezután történik meg két további bejárás során a next hop címkék újraosztása. A harmadik bejárás során, amelyet postorder módon hajtunk végre, a fa minden belső eleméhez hozzárendelünk egy next hop halmazt a ’#’ művelet segítségével, ahol a művelethez szükséges bemenő halmazok az adott pont bal-, illetve jobboldali részfáinak next hop halmaza. Az utolsó (preorder módon végrehajtott) bejárás során a gyökérelem kivételével minden elemnél megvizsgáljuk, hogy a hozzárendelt next hop halmaz tartalmazza-e a pont által a legközelebbi next hop-pal rendelkező ősétől örökölt next hop értéket. Amennyiben igen, úgy az adott pontba nem szükséges next hop címkét eltárolni, ha viszont nem, akkor a ponthoz tartozó next hop halmazból tetszőlegesen kiválasztunk egy értéket.

2. pszeudokód: Optimális forgalomtovábbítási tábla létrehozása bináris prefix fából

```
1: for each node N (postorder traverse)
2:       if N is a parent node then
3:             nexthops(N) = nexthops(left(N))#nexthops(right(N))
4: for each node N (preorder traverse)
5:       if N is not root and inherited(N)∩nexthops(N) !=0 then
6:             nexthop(N) = NULL
7:       else
8:             nexthop(N) = choose(nexthops(N))
```

A módszer előnye, hogy a bináris fában bizonyítottan a lehető legkevesebb szükséges címkét használja fel, ezáltal jelentős memória-megtakarítást lehet elérni. Hátránya azonban, hogy nem képes kilépni a bináris fa által szolgáltatott keretből. Erre jelent megoldást a következő pontban ismertetett prefix fa hajtogatás módszere.

2.5 Prefix fa hajtogatás

A prefix fa hajtogatás lényege, hogy megkeressük a kiinduló bináris prefix fában az ugyanolyan felépítésű részfákat, és azokat csak egyszer tároljuk el, majd az eredeti, teljesen kiírt fa helyett ezekre a megosztott részfákra hivatkozunk. Ezzel a módszerrel rendkívül jelentős tömörítést lehet elérni, hiszen a gyakorlatban előforduló forgalomtovábbítási táblákban csak viszonylag kevés féle next hop szerepel, ezért főleg a fa alsóbb szintjein ezeknek csak kevés kombinációja állhat elő. Az eredményül kapott adatszerkezet már nem egy bináris prefix fa, még csak nem is fa lesz, hanem egy irányított körmentes gráf, azaz DAG (Directed Acylic Graph), amelyben többek között például a bejárás már egy nemtriviális feladat. Ezen kívül az azonos részfák azonosításához szükség lesz egy segéd-adatszerkezetre is, ennek felépítése, tárolása és kezelése a prefix fa hajtogatás algoritmus egyik központi kérdése.

3. pszeudokód: Prefix fa hajtogatás módszere

```
1: create leaf pushed tree
2: for each node N (postorder traverse)
3:       if N is leaf then
4:             set N.ID in LeafMap
5:       else
6:             set N.ID to INVALID_ID
7: for each node N (postorder traverse)
8:       if N not leaf then
9:             N2 = SubtreeIndex.lookup(left(N).ID,right(N).ID)
10:                  if N2 == INVALID_ID then
11:                        N.ID = new ID
12:                        SubtreeIndex.insert(leftID,rightID,N)
13:                  else
14:                        repointer parent of N2 to N
15:                        delete subtree of N2
```

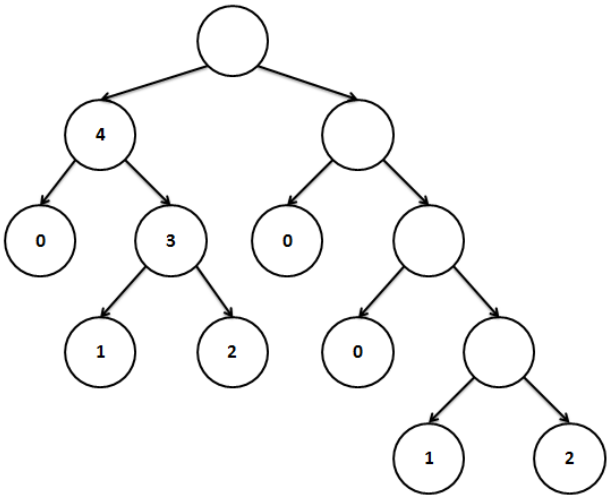
Az algoritmus (lásd 3. pszeudokód) kiindulási alapja tetszőleges bináris prefix fa lehet, de hatékonysági szempontok miatt érdemes először egy bejárással a már bemutatott módon komplett bináris prefix fává alakítani a bemenetet, ezután ugyanis már könnyebb megvalósítani a tömörítő algoritmust, hiszen a fa belső pontjaiban nem kell next hop egyezőséget vizsgálni, hiszen next hop értékek csak a levelekben lesznek. Ezt a konverziót végzi el a pszeudokód első sora.

Ezután egy postorder bejárás során minden levél elemhez hozzárendelünk egy azonosítót, méghozzá a next hop értéke alapján. Vagyis minden olyan levél, amiben ugyanolyan értékű next hop található, ugyanazt az azonosítót fogja megkapni. Ennek meghatározásához be kell vezetni egy segéd-adatstruktúrát, (a pszeudokódban `LeafIDMap`) amelyben eltároljuk a már megtalált next hop értékeket, és az azokhoz tartozó azonosítót. Mivel ebben a struktúrában csak nagyon kevés bejegyzés szerepelhet, ezért a kezelésével kapcsolatos további részletekre nem térek ki. Az első bejárás során csak a levél elemek kapnak azonosítót, a fa belső elemeiben ezt az értéket érvénytelenre állítjuk.

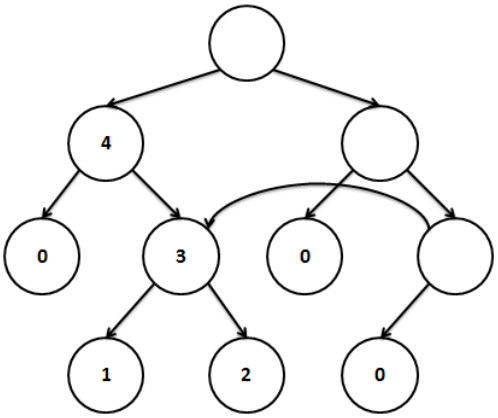
A második, postorder módon végrehajtott bejárás során építjük fel a részfák azonosítására szolgáló segéd-adatszerkezetet. Ezt a bejárást csak a fa belső pontjaira hajtjuk végre, hiszen a levél elemek már az előző bejárás során kaptak azonosítót, és ez adja a kiindulási alapot a második bejáráshoz. Ehhez a beszúrás (`insert`) és keresés (`lookup`) függvényeket használjuk fel. A beszúrás során megvizsgáljuk, hogy az aktuális pont bal- és jobboldali részfa azonosítójával történt-e már bejegyzés a segéd-adatszerkezetbe (3. pszeudokód, 8. sor). Amennyiben nem (3. pszeudokód, 9.-11. sor) akkor az adott elemhez új azonosítót generálunk, és ezzel együtt eltároljuk a segéd-adatszerkezetben. Ha viszont korábban már találtunk azonos alakú részfát (3. pszeudokód, 12.-14. sor), akkor a megtalált részfa szülőjének mutatóját átállítjuk az aktuális pontra, és felszabadítjuk az ily módon feleslegessé vált részfát.

Az algoritmus működését az alábbi példán mutatom be: kiindulási alapnak vegyük az 1.b ábrán látható komplett bináris prefix fát! Az első bejárás során az összes levél elemhez azonosítót rendelünk a next hop címkéik alapján: az ’*1*’ next hop értékhez a ’*0*’-ás, a ’*2*’-es next hophoz az ’*1*’-es, a ’*3*’-as next hophoz a ’*2*’-es azonosítót rendeljük (itt azért nem ugyanazokat a számokat használtam fel, hogy ne lehessen összetéveszteni a next hop-ot a részfa azonosítóval).

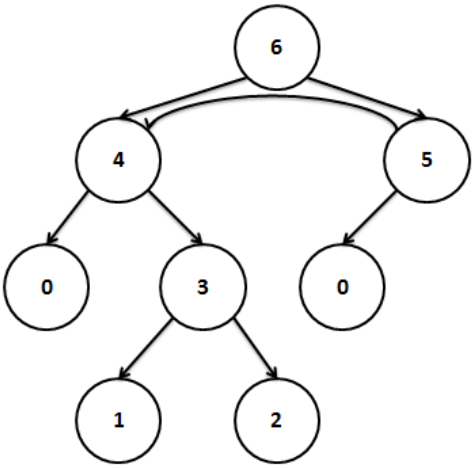
A második bejárás műveleteit (mivel postorder módon hajtjuk végre) először azon a ponton fogjuk végrehajtani, amelynek mindkét gyermekének van már kitöltött azonosítója (’*1*’-es, illetve ’*2*’-es). Ekkor a segéd-adatszerkezetünk még üres, így a `lookup` is az `INVALID_ID`-t fogja visszaadni, aminek hatására új azonosítót generálunk a pont számára (ez lesz a ’*3*’-as). Ezen elem szülőjének így már mindkét gyermeke rendelkezik azonosítóval, itt szintén végrehajtunk egy keresést a részfákat azonosító adatszerkezetben, és újabb azonosítót generálunk. Az algoritmus ezen állapotában a pontokhoz rendelt azonosítók a 6. ábrán látható:



6. ábra: A prefix fa hajtogatás algoritmus 1. fázisa



7. ábra: A prefix fa hajtogatás algoritmus 2. fázisa



8. ábra: A prefix fa hajtogatás algoritmus 3. fázisa

Folytatva a bejárást a fa pontjai sorban azonosítókat kapnak, majd amikor a legjobboldalibb belső

ponthoz érkezünk, megtaláljuk az első közös részfát. Itt végrehajtódik az első mutató-csere, ennek eredményét a 7. ábra mutatja. Egy szinttel feljebb újabb közös részfát találunk, ezt szintén kitöröljük a megfelelő módon, végül az két hátralévő pontok is azonosítót kapnak, de mivel itt már nem találunk újabb közös részfát, így a tömörítéssel is végeztünk. Ezt az állapotot mutatja a 8. ábra.

Az algoritmus előnye, hogy az ily módon betömörített forgalomtovábbítási tábla tárolási igénye sokkal kisebb, mint az előző pontokban bemutatott módszerek esetén. Ezen felül a legspecifikusabb bejegyzés keresése az eredményül kapott irányított körmentes gráfon pontosan ugyanolyan módon implementálható, mint a prefix fák esetében, így a hatékonyságon ezzel sem vesztünk semmit.

Ugyanakkor nagyon fontos megjegyezni, hogy a prefix fa hajtogatás módszer hatékonyságáról tudomásom szerint korábban nem készült átfogó teljesítményvizsgálat, vagyis nem lehetett tudni, hogy valós forgalomtovábbítási táblákon mennyire képes hatékony működésre. Éppen ezért munkám egyik kiemelt célja ennek a teljesítményvizsgálatnak az elvégzése, különös tekintettel a DAG felépítésének idejére és memóriaigényére.

A fejezet zárásaként megjegyezném, hogy az eddig bemutatott forgalomtovábbítási tábla tömörítési módszereken kívül még számtalan egyéb algoritmust is kidolgoztak már, ilyenek többek között a multibit prefix fák [10][11], kontrollált prefix kiterjesztést [12], a forgalomtovábbítási tábla tördelését kihasználó módszerek [13], a fa szintjeinek számát minimalizó algoritmusok [14], dinamikus fa bit térképen alapuló megoldások [15], stb. Ezen módszerek ismeretése és teljesítményanalízise meghaladja a dolgozat kereteit, ezért ezekkel a továbbiakban részletesen nem foglalkozom.

3. A prefix fa hajtogatás módszeréhez szükséges részfaindexek

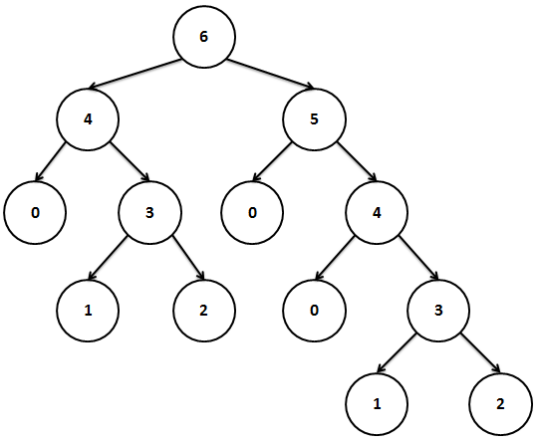
A dolgozat eddigi fejezeteiben áttekintettem az IP forgalomtovábbítás alapvető működését, és a forgalomtovábbítási táblák szervezésének különböző módszereit. A következő fejezetekben rátérek a dolgozat alapját képező munka ismertetésére. Először áttekintem a prefix fa hajtogatás algoritmus működésében kulcsfontosságú szerepet játszó részfaindex adatstruktúrák szervezésének legfontosabb kérdéseit, bemutatok négy lehetséges alternatívát, melyek mind különböző tárolási igényűek, és a keresést is különböző időben támogatják. Az ezt követő fejezetekben bemutatom a forgalomtovábbítási táblák tömörítési algoritmusainak, illetve a prefix fa hajtogatás részfaindexeinek teljesítményelemzéséhez készített szimulációs környezetet, majd ennek segítségével – tudomásom szerint az irodalomban elsőként – elvégzem a különböző módszerek teljesítményvizsgálatát, és levonom a konklúziókat.

A részfaindexek a prefix fa hajtogatás algoritmus során arra hivatottak, hogy segítségükkel fel lehessen ismerni a forgalomtovábbítási táblában szereplő azonos felépítésű részfákat. Közös tulajdonságuk, hogy az eredeti bináris fa harmadik bejárása alatt építjük fel őket. Az első bejárás a már említett komplett bináris prefix fát adja eredményül. A második bejárás során egy külön segéd-adatstruktúrában, a levélindexben minden levél elemhez azonosítót rendelünk, méghozzá a next hop értékeik alapján. Ez a levélindex lesz a kiindulási alapja a harmadik bejárásnak. Ezt a kezdeti struktúrát az összes bemutatandó adatstruktúra felhasználja. Megjegyzem, hogy további optimalizáció céljából ez a lépés összevonható a részfaindexet felépítő bejárással. További megfontolás tárgya, hogy akár a komplett bináris prefix fa bejárásával is összevonható lenne-e a prefix fa hajtogatás bejárása. A 3. pszeudokódban jól látható, hogy a részfaindexek kezeléséhez mindössze két függvényre van szükség: ezek a beszúrás és a keresés műveletei. Mivel ezen két műveleten kívül a pszeudokódban nincs más megkötés a részfaindexekkel szemb en, ezért lehetővé válik, hogy azt a prefix fa hajtogatás algoritmustól függetlenül, absztrakt szinten tárgyaljuk, és a feladat céljára legmegfelelőbb adatstruktúrát válasszuk ki. Ebben a fejezetben négy különböző megvalósításra teszek javaslatot, ezek vektoron, piros-fekete fán, hash táblán, és fix méretű hash táblán alapuló megoldások.

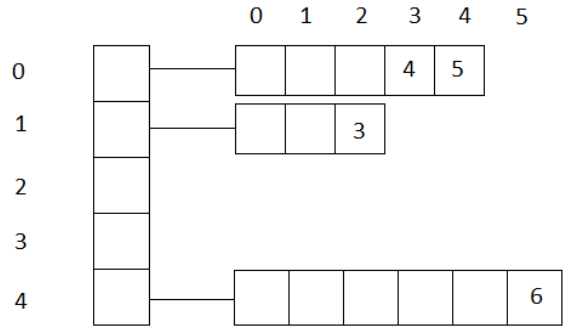
3.1 Vektor alapú részfaindex

Az első bemutatásra kerülő megoldás egy kétdimenziós vektor, amelyben az első dimenzió azonosítója az adott fa-elem baloldali részfájának azonosítója, a második dimenziója pedig a jobboldali részfájának azonosítója. Az eltárolt elem az így azonosított memóriaterületen egy faelem-mutató, amely alapján azonosításra kerülnek az azonos alakú részfák. Ezzel a megoldással egy ritkán

kitöltött két dimenziós vektort kapunk, hiszen csak azokban a pozíciókban tárolunk értékeket, amihez létezik olyan részfa a bináris fában, melynek bal gyermeke a bal, jobb gyermeke a jobb részfa azonosítójával rendelkezik. Ha a 2. fejezetben bemutatott FIB-et tömöríteni szeretnénk a prefix fa hajtogatás algoritmussal, akkor a részfáit a 9. ábra szerint kell megszámoznunk.



9. ábra: Részfák azonosítása



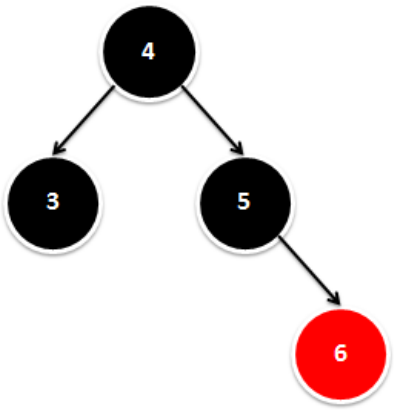
10. ábra: Kitöltött vektor alapú részfaindex

Ezekhez a részfákhoz a 10. ábrán látható vektor alapú részfaindex fog tartozni: (itt a bal oldalon függőlegesen látható vektor tartozik a bal részfa azonosítóhoz, míg a jobb oldalon, vízszintesen ábrázolt vektorok a jobb részfa azonosítót jelölik).

Úgy tűnik tehát, hogy ez az adatszerkezet jelentős, $O(N^2)$ nagyságrendű memóriaszükséglettel rendelkezik (ahol N a fában lévő elemek száma), ami ellentmondani látszik az eredeti célkitűzésnek, miszerint szeretnénk csökkenteni a felhasznált memória mennyiségét. Ugyanakkor elméletileg rendkívül gyors működést tesz lehetővé a tömörítés során, hiszen $O(1)$ lépésben meg lehet találni a „táblázat” megfelelő elemét (vagyis a részfaindexhez tartozó beszúrás és keresés utasítások gyorsan végrehajthatóak).

3.2 Piros-fekete fa alapú részfaindex

Az egyik legegyszerűbb, mégis hatékony, intuitíven adódó megoldás a részfák azonosítására a keresési fa alapú adatstruktúra. [16, 251-273] Ennek rengeteg különböző változata ismert, ezek közül a jelenlegi feladathoz az önkiegyenlítő tulajdonsága, és könnyű megvalósíthatósága miatt a piros-fekete fát választottam. [16, 273-301] [17] Ebben az egyedi kulcsot a bal-, illetve jobboldali részfa azonosítókból képzett pár lexikografikus rendezése adja, a piros-fekete fában eltárolt érték pedig egy fa-elem mutató. A 9. ábrán bemutatott részfa azonosítókhoz tartozó piros-fekete fát a 11. ábra mutatja:

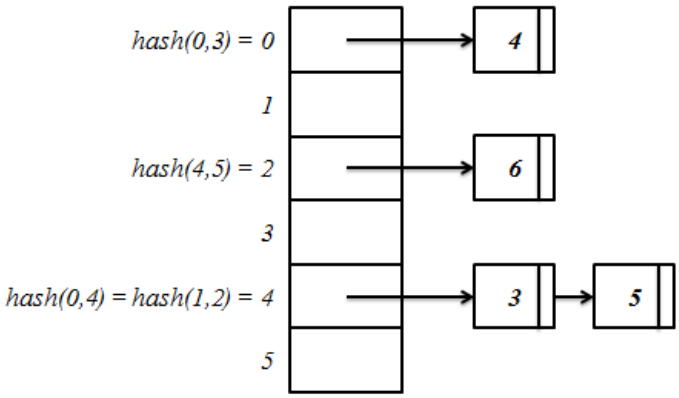


11. ábra: Kitöltött piros-fekete fa alapú részfaindex

A kétdimentiós vektor alapú megoldással szemben a kezeléséhez (vagyis a beszúrás és keresés függvényekhez) *O(log N)* nagyságrendű időre van szükség, (ahol *N* a fában lévő elemek száma), míg a memóriaigénye *O(N)*, vagyis a memóriaigénye alacsonyabb, mint a vektor alapú részfaindex esetén, míg időigénye valamivel magasabb nagyságrendű.

3.3 Hash tábla alapú részfaindex

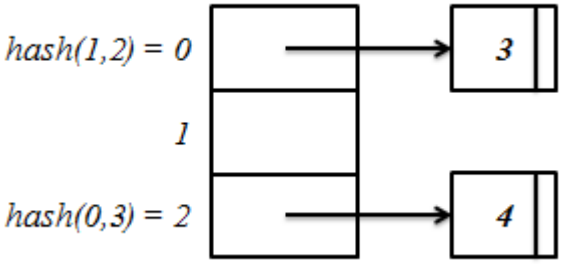
További optimalizációt lehet elérni, ha a piros-fekete fa alapú megoldás helyett hash tábla alapú részfaindexet használunk. [16 224-237] Ebben a bal- és jobboldali részfa azonosítóiból képzett pár hash értéke adja a kulcsot, az eltárolt érték ugyanaz, mint a piros-fekete fa alapú részfaindex esetében (a 9. ábra alapján kitöltött hash tábla a 12. ábrán látható). Ezáltal további gyorsulást tudunk elérni, hiszen a hash számításhoz szükséges idő ütközésmentes esetben *O(1)* nagyságrendű. Természetesen hash-ütközés esetén legrosszabb esetben ez akár *O(K)* is lehet (ahol *K* a hash tábla mérete, de a gyakorlati tapasztalat valóban gyorsulást mutat a piros-fekete fán alapuló megoldáshoz képest. A hash tábla tárolásához szükséges memória *O(N)* nagyságrendű, ahol *N* a fa elemeinek száma.



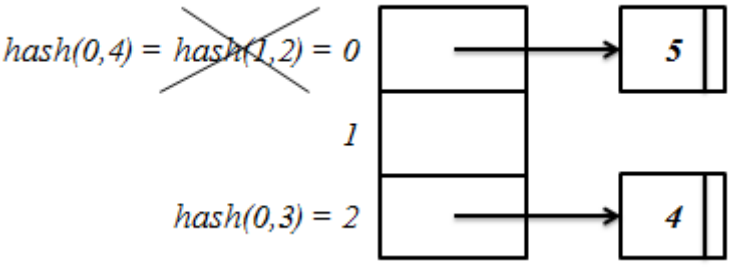
12. ábra: Kitöltött hash tábla

3.4 Fix méretű hash tábla alapú részfaindex

Mint láttuk, az előző pontban bemutatott hash alapú részfaindex tárolási igénye *O(N)* nagyságrendű, vagyis az eredeti tömörítendő fa méretével arányos segéd-adatszerkezetet kell eltárolni. Bár ezt a részfaindexet nem kell gyors elérésű memóriába tenni, hiszen a csomagtovábbítás során nem kell lekérdezést végrehajtani rajta, de ez a memóriaigény még így is jelentős mértékű. Ezen probléma megoldására találtam ki a fix méretű hash táblán alapuló részfaindexet. Ennek alapötlete, hogy mivel a tömörítendő forgalomtovábbítási táblában az egyes next hop címkék gyakorisága rendkívül eltérő, ezért bizonyos részfaindex bejegyzések a többinél sokkal gyakrabban fordulnak elő, illetve rengeteg olyan részfaindex bejegyzés is lesz, amelyek összesen egyszer fordulnak elő.[1] Ezeket felesleges eltárolni, és ezért a részfaindex memóriaigénye ily módon csökkenthető egy veszteséges adatstruktúrával. A fix méretű hash táblán alapuló részfaindex pontosan ezt az ötletet valósítja meg. A 9. ábrán bemutatott részfaindexek tárolásának folyamata a fix méretű hash táblában a 13.-15. ábrákon található. Hogy a módszer lényegét szemléltetni tudjam, ezért a négy beszúrandó elemhez 3 méretű hash táblát használok.

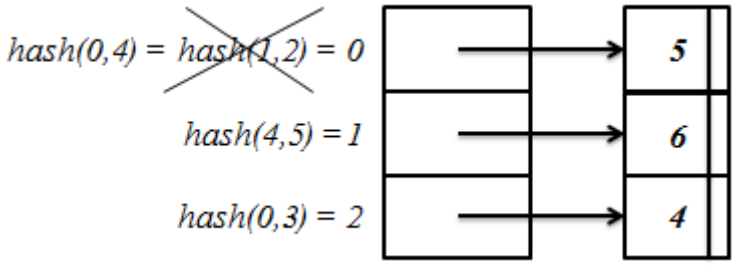


13. ábra: A fix méretű hash tábla két beszúrás után



14. ábra: A fix méretű hash tábla az ütközés után

1 A next hop értékek és részfaindex bejegyzések eloszlásának meghatározására az 5. fejezetben valódi környezetből származó forgalomtovábbítási táblán méréseket végzek.



15. ábra: A fix méretű hash tábla a négy beszúrás után

Alapötlete, hogy nem tároljuk el az összes létező részfát, hanem egy fix méretű hash táblába szúrjuk be a legutoljára megtalált részfákat. A fix méret egy bemenő paraméter, ezáltal explicit módon kézben tartható a felhasznált memória mennyisége. Ennek ára természetesen az, hogy mivel nem az összes részfát ismerjük, a tömörítés hatékonysága romlik. Ugyanakkor egy jól eltalált paraméter értékkel megfelelő kompromisszumot találhatunk a forgalomirányítási tábla tömörsége és az ahhoz felhasznált további memória mennyisége között.

Ahhoz, hogy ez a módszer működhessen, több mindenre kell figyelni, mint az eddigi részfaindex verziók esetén. Először is szükséges egy hash függvény, amely mindig az adott méretű hash táblába címez bele, ezen kívül a keresés metódusa is átalakul. Az eddigi megoldásoknál csak azt kellett nézni a keresés során, hogy a bal- és jobboldali részfák azonosítói által kijelölt helyen van-e már eltárolva azonosító, illetve ha igen, akkor mi az. Mivel azonban ennél a megoldásnál több különböző részfa hash értéke is mutathat ugyanarra a címre, ezért ezt az esetet kezelni kell. Természetesen az előző pontban bemutatott hash tábla alapú részfaindexben is lehetnek ütközések, de ettől függetlenül minden elem eltárolásra kerül, míg most ütközés esetén felülírás történik, és ez okozza, hogy nem feltétlenül ismerjük az összes létező részfát. A felülíráshoz először meg kell vizsgálni, hogy a keresés által visszaadott elem és a bejárás során éppen feldolgozásra kerülő elem bal-, illetve jobboldali részfáinak azonosítói megegyeznek-e. Amennyiben igen, akkor ugyanazt a részfát találtuk meg, ha pedig nem, akkor felül kell írni a hash táblában az aktuális értéket. Így mindig a legutoljára megtalált *M* részfát tudjuk azonosítani, ebből fakad a tömörítés mértékének romlása (ahol *M* a hash tábla maximális mérete). A módszer időigénye a hash függvényből adódóan *O(1)*, de mivel az előző módszerrel ellentétben itt az ütközés esetén nem fűzünk hozzá a hash tábla adott bejegyzéséhez még egy elemet, hanem felülírás történik, ezért az időigény mindig *O(1)* nagyságrendű marad. Ezen kívül memóriaigénye is rendkívül alacsony, hiszen *O(1)* nagyságrendű, vagyis a tömörítendő bináris prefix fa méretétől teljesen független.

3.5 Részfaindexek összefoglalása

Ebben a fejezetben bemutattam négy különböző megvalósítását a részfaindex adatstruktúrának. Ezek mindegyike különböző nagyságrendű tárhelyigénnyel és kiszolgálási idővel rendelkezik, ezt szemlélteti a 16. ábra. Pusztán ezen elméleti tulajdonságok ismeretében azonban nem lehet egyértelműen megállapítani, hogy melyik részfaindex pontosan milyen hatásfokot ér el. Ennek meghatározása érdekében elkészítettem a következő fejezetben ismeretetésre kerülő szimulációs környezetet, majd ennek segítségével teljesítményvizsgálatokat végeztem a részfaindex verziókon.

	Memóriaigény	Időigény
Vektor alapú részfaindex	$O(N^2)$	$O(1)$
Piros-fekete fa alapú részfaindex	$O(N)$	$O(\log N)$
Hash tábla alapú részfaindex	$O(N)$	$O(1)$
Fix méretű hash tábla alapú részfaindex	$O(1)$	$O(1)$

16. ábra: A különböző részfaindex verziók memória-, és időigényének nagyságrendjei

4. A szimulációs környezet ismertetése

Az eddigi fejezetekben bemutatott tömörítési algoritmusok, valamint a prefix fa hajtogatás módszerhez tartozó részfaindex verziók teljesítményanalízise céljából szimulációs környezetet készítettem C++ nyelven, ebben a fejezetben ennek részleteit, többek közt az osztályhierarchiát, az adatmodell főbb tulajdonságait, metódusait, valamint a tömörítési algoritmusok implementációs részleteit ismeretem.

4.1 A szimulációs környezet specifikációja

A fent leírt algoritmusok, adatszerkezetek gyakorlatban való kipróbálásához egy szimulátort készítettem, amely megvalósítja a routing táblák beolvasását, tömörítését, és segítségével méréseket lehet végezni a különböző módszerek hatékonyságáról. A felhasznált konfiguráció részletes listája a következő:

- Oracle VirtualBox 4.1.18 verziójú virtuális gépen futtatott
- Ubuntu 12.04 LTS
- Processzor: Intel Core i5-2410M, 2.3 GHz
- Memória: 1 GB
- kernel verzió: 3.2.0-31-generic-pae
- g++ (verziószám: 4.6)
- Standard Template Library
- C++11 szabvány
- lemon C++ könyvtár (verziószám: 1.2.3) [18]
- boost C++ könyvtár (verziószám: 1.50.0) [19]

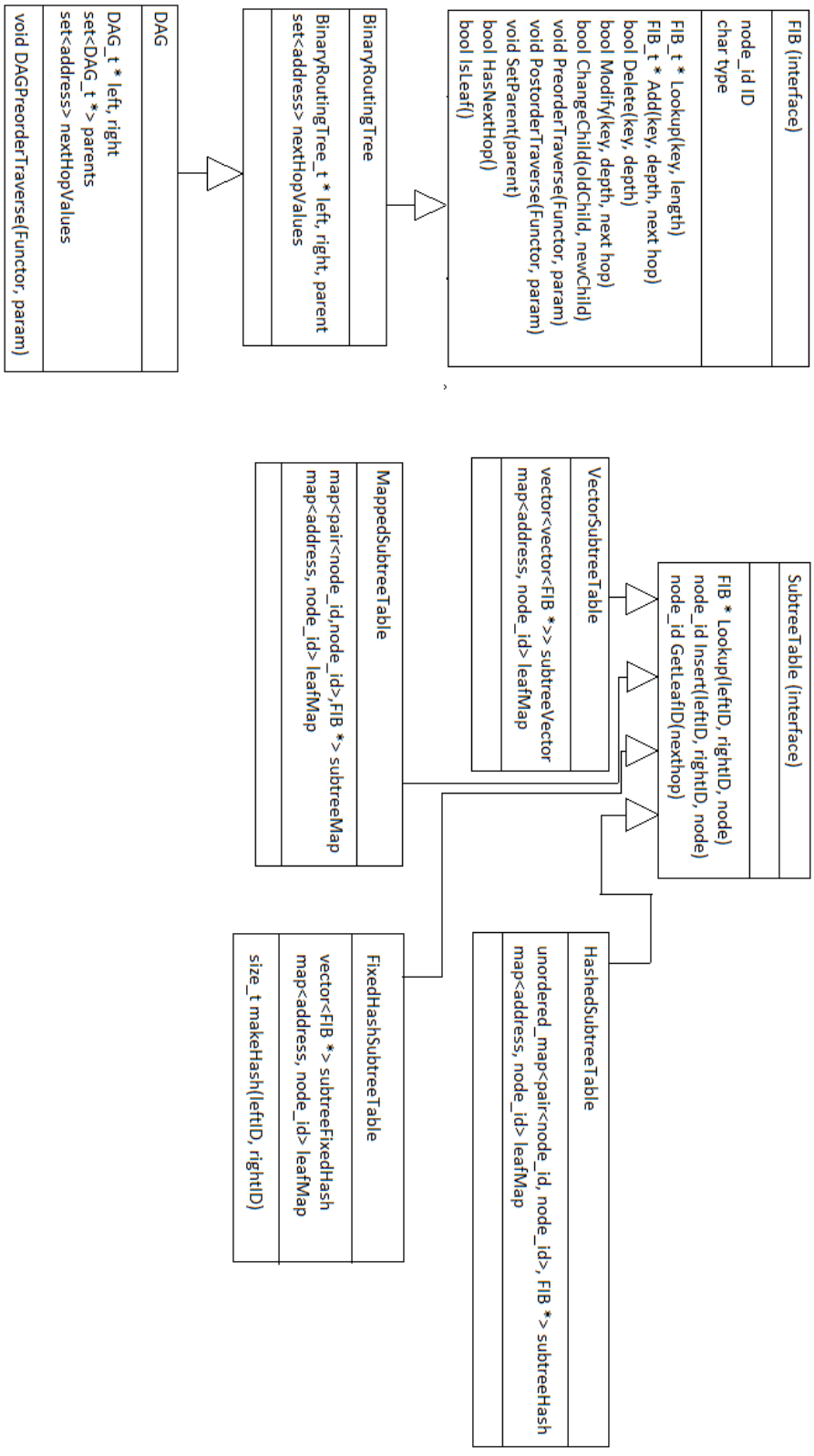
A kész program parancssorról vezérelhető, ahol a forgalomtovábbítási táblát tartalmazó fájlon kívül egyéb bemeneti paraméterek is megadhatóak, többek között a tömörítés módja, a kiírandó adatok mennyisége, a statisztikai adatok láthatósága, az eredményt rögzítő fájl formátuma, illetve az egyes tömörítési módokhoz tartozó további paraméterek is beállíthatóak, például a prefix fa hajtogatás algoritmusnál a részfaindex fajtája, vagy ezen belül a fix méretű hash tábla esetén a tábla mérete.

A program lefutása a paraméterek megadása után minden módszernél hasonló módon történik: első lépésben felépítem a forgalomtovábbítási tábla bejegyzései alapján a tömörítendő bináris prefix fát, majd erről kezdeti statisztikákat gyűjtök. Ezután lefut az adott tömörítő algoritmus, majd az eredményül kapott prefix fáról (vagy DAG-ról) újabb statisztikát készítek. Ez a statisztika a következőkből áll: a fa maximális mélysége, összes pontjainak száma (külön meg van jelenítve a belső pontjainak és levél elemeinek száma, ezek összege persze a fa összes pontjainak számát adja eredményül), a felhasznált mutatók száma, a next hop információt tartalmazó elemek száma, felhasznált memória, időzítési adatok stb.

Fontos megemlíteni, hogy az egész programkódot sablonozott (template) módon valósítottam meg, amelynek hála nem kell előre megadni a bemenő routing tábla formátumát, így az IP cím helyén lehet például IPv4-es vagy IPv6-os cím, az alhálózati maszk értéke lehet egész szám formátumban, vagy a gyakorlatban elterjedt 4*8 bit-es formátumban is, illetve a next hop érték is tetszőleges formátumú lehet (jelenleg a program egész számot és IPv4-es címet tud feldolgozni).

4.2 Osztálydiagram, fontosabb osztályok bemutatása

A 17. ábrán látható osztálydiagramon feltüntettem a legfontosabb osztályokat, kiemelve néhány metódusukat és adattagjukat, illetve az osztályok között lévő öröklési viszonyokat, de az ábra közel sem teljes, csupán a könnyebb átláthatóságot hivatott biztosítani.



17. ábra: A szimulációs környezet osztálydiagramja

4.3 FIB és leszármazott osztályai

A program lelke a `FIB` nevű absztrakt osztály, amely feladata, hogy keretbe foglalja a különböző módon implementált adatstruktúrákon végzett műveleteket. Legfontosabb műveletei a `Lookup`, `Add`, `Delete`, `Modify`. A `Lookup` függvény segítségével lehet egy már létező táblában megkeresni egy adott IP címhez és hálózati maszkhoz tartozó next hop értéket. Az `Add` a tábla felépítéséhez szükséges, míg a `Modify` és a `Delete` a tábla dinamikus frissítésére valók. Ezen kívül olyan további függvényeket tartalmaz, amely egy adott elem gyermekeinek, szüleinek, és a next hop tartlamának kezeléséhez szükségesek. (például `SetChild`, `GetChild`, `ChangeChild`, `SetParent`, `UnsetParent`, `GetParent`, `HasNextHop`, `AddNextHop`, `SetNextHops`, `ClearNextHops`, `GetNextHops`, `IsLeaf`). Továbbá definiálja a preorder és postorder bejárást is, és néhány függvényt a `FIB` struktúrában eltárolt adatok kiíratására. Jelenleg ezt az absztrakt osztályt két leszármazottja implementálja. Ezek közül a fontosabb a bináris prefix fa (`BinaryRoutingTree`), hiszen már a beolvasás eredményeként felépülő struktúra is ezt a formátumot használja. Ezen kívül a `FIB` további megvalósítása a már említett prefix `DAG` is. A `DAG` nem csak a `FIB`, de a `BinaryRoutingTable` leszármazottja is, hiszen minden fa egyben egy DAG is. Így csak azokkal a metódusokkal kellett kibővíteni ezt az osztályt, amik több szülő kezelését teszik lehetővé (`GetParents`, `SetParents`, `UnsetAllParents`), illetve szükség volt egy DAG bejárást megvalósító függvényre is ahhoz, hogy a tömörítés után létrejövő új adatszerkezet elemeit is pontosan meg lehessen számolni. Ez jelenleg nem egy teljes értékű preorder vagy postorder bejárás, mert ahhoz meg kell valósítani a DAG topológikus rendezését. Mivel azonban egyelőre csak a tömörített tábla bejegyzéseinek száma volt a számottevő adat, ezért a bejárást megvalósító függvényben egy egyszerűsítéssel éltem: minden ponthoz eltároltam egy `visited` nevű boolean változót, amelyet igaz értékűre állítottam, ha a bejárás során az adott ponthoz értem, és csak olyan pontokon folytattam a bejárást, melynek `visited` értéke hamis.

Fontos megjegyezni, hogy a `DAG`, és ezáltal a prefix fa hajtogatás tömörítés jelenleg még nem támogatja a `FIB`-hez tartozó dinamikus függvényeket, vagyis az `Add`-ot, `Modify`-t és a `Delete`-t. Ez azt jelenti, hogy a tömörítés során először egy bináris fát építünk, majd ezt tömörítjük, de ezután hozzáadni, módosítani, és törölni nem lehet a `FIB`-ből, ezen függvények implementálása lesz a munkám további részének egyik legfőbb feladata.

4.4 Tömörítési algoritmusok

A második pontban pszeudokód szintjén bemutatott tömörítési módszerek bejárásonkénti szabályait az egyes osztályokon belül az úgynevezett funktorok írják le. A komplett bináris prefix fa két bejárásához tartozó funktorok: `LeafPushFunctor1`, `LeafPushFunctor2`. Az optimális forgalomtovábbítási tábla tömörítésében három bejárás található, így ezen módszer funktorai:

`ORTCFunctor1, ORTCFunctor2, ORTCFunctor3`. Végül a prefix fa hajtogatás módszerében ismét két bejárásra van szükség: `TrieFoldingFunctor1, TrieFoldingFunctor2`.

Ezek azokat a szabályokat tartalmazzák, amelyek alapján eldönthető, hogy egy adott bejárás során a fa egy pontjában milyen műveleteket kell végrehajtani, vagyis mindegyik funktor megfeleltethető az első fejezetben található 1-3. pszeudókódokban az egyes bejárásoknak. Ezeket a funktorokat a preorder és posztorder bejárás paramétereként használom fel. Mivel mindegyik tömörítési módszer több bejárást is alkalmaz, ezért ezeket egy `Compression` nevű metódus fogja egy csokorba, amiben sorban meghívjuk a megfelelő irányú bejárásokat a hozzá tartozó funktorral a tömörítendő bináris prefix fa gyökérelemére. A pre- és postorder bejárás másik paramétere szabadon felhasználható – a komplett prefix fa és az optimális forgalomtovábbítási tábla tömörítési módszerek például nem használja semmire, azonban a prefix fa hajtogatás algoritmus esetében ezen paraméteren keresztül lehet megadni, hogy milyen részfaindex verziót szeretnénk használni.

4.5 Részfaindexek

A `SubtreeTable` nevű absztrakt osztály leszármazottai. Ez az osztály definiálja a kezelésükhöz szükséges függvényeket, (`Lookup, Insert, GetLeafID`) azonban azt, hogy ezek a függvények milyen adatszerkezetet használnak, már megoldásonként változó. Ezen szerkezetekhez a Standard Template Library C++ könyvtár megoldásait használtam fel.

4.5.1 Vektor alapú részfaindex

Ezen részfaindex verzió megvalósításához az `std::vector` konténert használtam fel. A megvalósítása során azonban előkerült egy probléma, amely jelentősen befolyásolja mind a tömörítéshez szükséges időt, mind a felhasznált memória mennyiségét. Mivel előre nem lehet tudni, hogy mennyi bal-, illetve jobboldali részfa azonosító lesz kiosztva, ezért dinamikusan kell növelni a vektorok méretét. Ehhez az `std::vector reserve()`, illetve `resize()` függvényeit használtam, de mint kiderült, ezek rendkívül időigényes műveletek. Ezek megkerülésére egy másik lehetőség, hogy megbecsülöm a lehetséges maximális méretét, és ezt előre lefoglalom. Itt viszont a memóriaszükséglet okoz problémát, ugyanis egy átlagos routing tábla bejegyzéseinek száma, illetve a tömörítendő bináris prefix fa pontjainak száma százezres nagyságrendű lehet, és ennek a négyzetével arányos méretű memóriára lenne szükség, ami pedig már terabájtokban lenne mérhető. Emiatt a hátránya miatt a kétdimenziós vektorokon alapuló részfaindex pusztán elméleti síkon lehet érdekes, a méréseim során a többi megoldáshoz képest több százszor, nagy routing táblák esetén pedig akár már több ezerszer lassabbnak bizonyult, miközben a felhasznált memóriája is nagyságrendekkel nagyobb.

4.5.2 Piros-fekete fa alapú részfaindex

A megvalósításához az `std::map` konténert használtam fel. Ez egy nagyon egyszerűen kezelhető adatszerkezet, amelyben kulcs-érték párokat lehet eltárolni. A kulcs ebben az esetben a bal- illetve jobboldali részfák azonosítóiból képzett pár, az eltárolt érték pedig egy fa-elem mutató. Az STL könyvtár szerinti implementációjának hátterében ez az adatszerkezet egy piros-fekete fát épít fel, vagyis valóban egy piros-fekete fát kapunk eredményül.

4.5.3 Hash tábla alapú részfaindex

Ezt a verziójú részfaindexet az `std::unordered_map` konténer segítségével építettem fel. Ez az előző pontban bemutatott std::map-hez hasonlóan kulcs-érték párok tárolására alkalmas, de a kulcshoz a bal- illetve jobboldali részfa azonosítókból képzett pár hash értékét használja, így a a konténer sem az azonosítók, sem az értékek alapján nem rendezett.

4.5.4 Fix méretű hash tábla alapú részfaindek

Ezt a részfaindex verziót az `std::vector` konténer segítségével alkottam meg. Ahhoz, hogy ezt fix méretű hash táblaként tudjam használni, elkészítettem a `makeHash()` nevű függvényét, amely az (1)-es képlet alapján számolja ki a bal- illetve jobboldali részfa mutatókból képzett hash értéket, és ezt használom fel a vektor elemeinek megcímzésére. A képletben a konstans értéke a jelenlegi implementáció szerint $2^{11} - 1$. Ez az egyszerű számítási módszer megfelel a hash-elés szokásos feltételeinek, de korántsem optimális, ennek pontosabb kidolgozása jövőbeli feladataim egyike.

$$kulcs=(konstans*bal\ részfaindex+jobb\ részfaindex)\%hash\ méret \quad (1)$$

4.6 Egyéb megfontolások

A különböző megvalósítású FIB-eken, a tömörítési algoritmusokon, és a hozzájuk szükséges adatszerkezeteken kívül a program még tartalmazza a `Utilities` nevű header fájlban ezek kezeléséhez szükséges segédfüggvényeket. Ezek között vannak a beolvasáshoz, és kiíratáshoz szükséges függvények (például a bemeneti fájlban található IP címet a program számára is értelmezhető értékké alakító segédfüggvény), matematikai, logikai műveletek függvényei, összehasonlító operátorok deklarációi, illetve itt kaptak helyet a tömörítési algoritmusok definíciói is.

Ezen kívül fontos szerepet tölt be a `Stats` nevű header fájl is, amely tartalmazza mindazokat az osztályokat, amelyek szükségesek a statisztikák készítéséhez. Ilyen például a `TreeStats`, `TimerStats`, és az ezek eredményeit XML fájlba feldolgozó `XMLResult`.

5. Teljesítményvizsgálat

Az eddig bemutatott tömörítési algoritmusokat, és a prefix fa hajtogatás módszerhez tartozó részfaindex verziókat az előző fejezetben bemutatott szimulációs környezettel numerikus értékelésnek vetettem alá, jelen fejezetben a különböző módszerek értékelésére térek rá.

5.1 A kiértékelés alapjai

A 2. fejezetben bemutatásra került tömörítési algoritmusokon, illetve a 3. fejezetben ismertetett prefix fa hajtogatás módszerhez tartozó részfaindexeken az elkészült szimulációs környezettel számos mérést végeztem, ehhez az interneten nyilvánosan elérhető valódi, illetve generált forgalomtovábbítási táblákat használtam fel. [20]

A tömörítési módszerek mérése során az alábbi paraméterek értékeinek alakulását vizsgáltam:

- a tömörítés hatékonysága, vagyis az eredeti bináris prefix fa next hop címkét tartalmazó pontjainak száma, és a tömörített adatszerkezet ugyanezen paraméterének aránya.
- a tömörített adatszerkezet építési ideje (ennek méréséhez a lemon C++ könyvtár által implementált `timer` struktúrát használtam fel.)

A prefix fa hajtogatás algoritmus különböző verziójú részfaindexeinek mérései során az alábbi paramétereket vizsgáltam:

- a részfaindex felépítéséhez szükséges idő (ehhez szintén a lemon C++ könyvtár `timer` struktúráját használtam.) Az időmérést két különböző időpontról indítva is elvégeztem mindegyik verziójú részfaindex esetén: a `main` függvényben a komplett bináris prefix fa módszerének lefutása előtt, illetve után, ezzel megvizsgálva a tömörítés teljes idejét, illetve a részfaindexek kiszolgálási idejét is.
- a részfaindex által lefoglalt memória nagysága. Ennek meghatározására becsléseket készítettem. A vektor alapú részfaindex esetén a felhasznált memória becslését a (2)-es képlet szerint határoztam meg: (a *méret()* függvény az adott vektor nagyságát adja meg, a *mutató_méret* pedig a felhasznált mutatók nagyságát jelzi byte-ban.)

  $$Memóriaméret = mutató\_méret * (méret(vektor) + méret(vektor[i])), \qquad (2)$$

  ahol $i = 0,1, \ldots, méret(vektor)$

  A piros-fekete fa alapú részfaindex által felhasznált memória becslését a (3)-as képlet adja meg. Itt az első, $2*k$ tag a piros-fekete fában az adott elemhez tartozó két gyermek mutató, míg a második tag pedig az eltárolt értékre irányított mutató.

  $$Memóriaméret = mutató\_méret * (2*k + k), \qquad (3)$$

  ahol k a fában tárolt elemek száma.

A hash tábla alapú részfaindex memóriabecslését a (4)-es képlettel határoztam meg. Itt minden hash tábla elemhez tartozik egy láncolt lista, aminek kezeléséhez szükséges egy mutató, és az eltárolt értékre irányított mutató pedig a második tag-ot adja.

$$Memóriaméret=mutató\_méret*(k + k), \qquad (4)$$

ahol k a hash táblában eltárolt elemek száma.

Végül pedig a fix méretű hash tábla alapú részfaindex memóriájának becslését az (5)-ös képlet alapján határoztam meg.

$$Memóriaméret=mutató\_méret*M, \qquad (5)$$

ahol M a hash tábla mérete.

A felhasznált forgalomtovábbítási táblákat lekorlátoztam maximum 24-es mélységű fákra, de mivel ez az IPv4-es címeknél a gyakorlatban azt jelenti, hogy csak az utolsó, legspeciálisabb 8 bithez tartozó információkat vesztettem el, a forgalomtovábbítási táblákban pedig tipikusan ezeknél általánosabb bejegyzések vannak, ezért ezzel az egyszerűsítéssel maximum néhány ezres nagyságrendű bejegyzést töröltem a fából, de az eredményül kapott forgalomtovábbítási táblák már könnyebben kezelhetővé váltak a szimulációs környezetben.

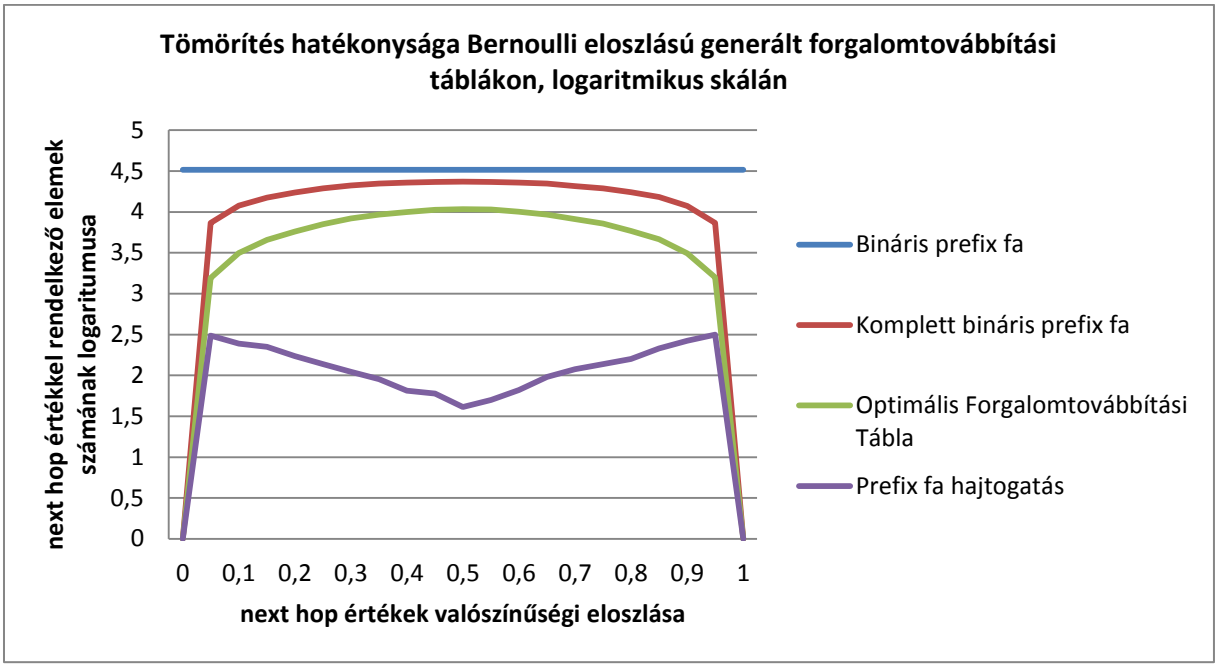
5.2 Tömörítési algoritmusok teljesítménye

Ebben a fejezetben a különböző tömörítési módszerek (komplett bináris prefix fa, optimális forgalomtovábbítási tábla tömörítés, prefix fa hajtogatás algoritmus) által elért tömörítési arányokról, illetve az ehhez felhasznált időről szóló mérési eredményeket mutatom be.

5.2.1 A tömörítés hatékonysága

A különböző tömörítési algoritmusok implementálása után az egyik fő célom a tömörítési hatékonyságuk vizsgálata volt. Azt feltételeztem, hogy a forgalomtovábbítási táblák next hop címkéinek eloszlásában bizonyos szintű entrópia figyelhető meg.[21] Ennek bizonyítására az első méréshez 20 db generált forgalomtovábbítási táblát generáltam, méghozzá úgy, hogy bennük csak kétféle next hop érték szerepelt (’*0*’ és ’*1*’), és ezek valószínűségi eloszlása a Bernoulli eloszlást követte. Eszerint az első és utolsó sorszámú táblákban csupa ’*0*’ illetve ’*1*’ next hop érték szerepelt, a tizedik táblázatban pedig mindkét next hop érték valószínűsége *0,5* volt. Amennyiben a hipotézis helytálló, olyan eredményt kellett, hogy kapjak, melyben a legkevésbé hatékony tömörítést az egyenletes eloszlás esetén kapom (vagyis $p('0') = p('1') = 0{,}5$), míg a determinisztikus esetekben ($p('0') = 1$ illetve $p('1') = 1$) mindegyik módszertől azt vártam, hogy egyetlen pontba, a gyökérelembe tömörítsék a teljes forgalomtovábbítási táblát. A mérések eredményét logaritmikus skálán mutatja a 18. ábra. Ezen jól látható, hogy a komplett bináris prefix fa módszerrel, és az optimális forgalomtovábbítási tábla tömörítéssel valóban a várt eredményeket kaptam, és az is leolvasható az

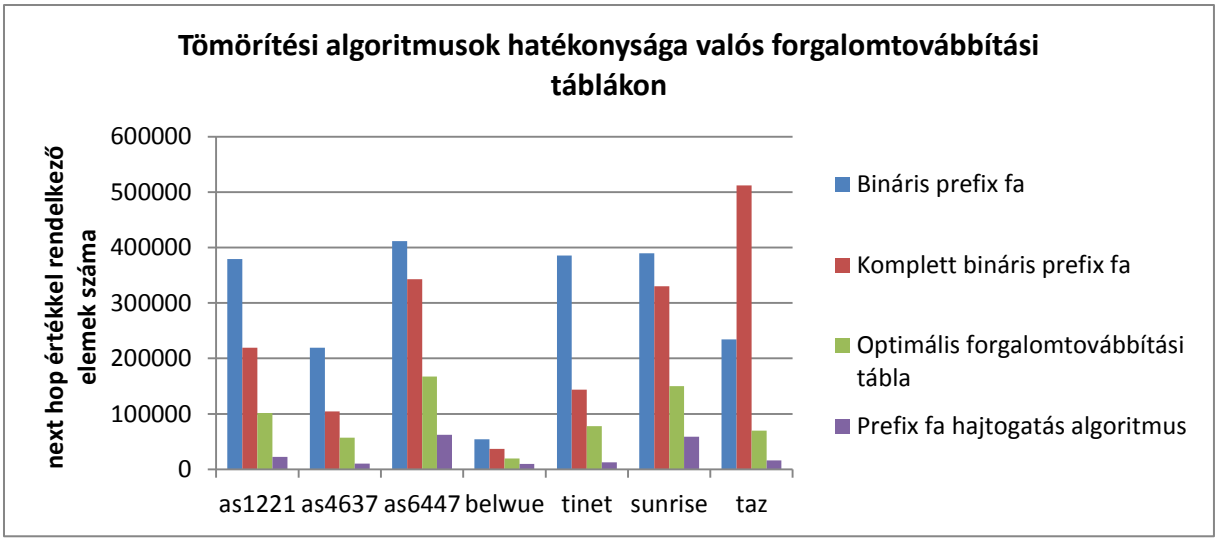
ábráról, hogy a komplex bináris prefix fa módszere érte el a legkisebb tömörítési hatásfokot, utána az optimális forgalomtovábbítási tábla algoritmusa következett, de a prefix fa hajtogatás módszere messze hatékonyabbnak bizonyult mindkettőnél.



18. ábra: A tömörítés hatékonysága Bernoulli eloszlású generált forgalomtovábbítási táblákon, logaritmikus skálán

Érdekes megfigyelés, hogy a prefix fa hajtogatás algoritmusa a tizedik forgalomtovábbítási táblánál, vagyis *p('0') = p('1') = 0,5* esetén érte el a leghatékonyabb tömörítést (leszámítva a két determinisztikus esetet, ahol ez a módszer is egyetlen pontba tömörít). Ennek oka, hogy ha a kétféle next hop érték eloszlása egyenlő, úgy a belőlük képzett különböző típusú részfák is száma egyre kisebb, így egyre hatékonyabb tömörítési eredményt produkál a módszer.

Ezután megvizsgáltam a tömörítési algoritmusok hatékonyságát valódi forgalomtovábbítási táblákból vett adatokon is. Az algoritmusok lefutása után kapott eredmények a 19. ábrán láthatóak. Jól látható, hogy a tömörítés mértékében elért sorrend egyezik az előző pontban ismeretett generált routing táblákon meghatározottakkal, vagyis a prefix fa hajtogatás módszere bizonyult a legsikeresebbnek. Érdekes továbbá megfigyelni, hogy az utolsó, 'taz' nevű forgalomtovábbítási táblán a komplett bináris prefix fa nagyobb elemszámot ért el, mint a bináris prefix fa, ezzel bizonyítva, hogy ezen módszer esetén nem csak tömörítésre, de szerencsétlen eloszlás esetén akár még méretnövekedésre is számítani kell.



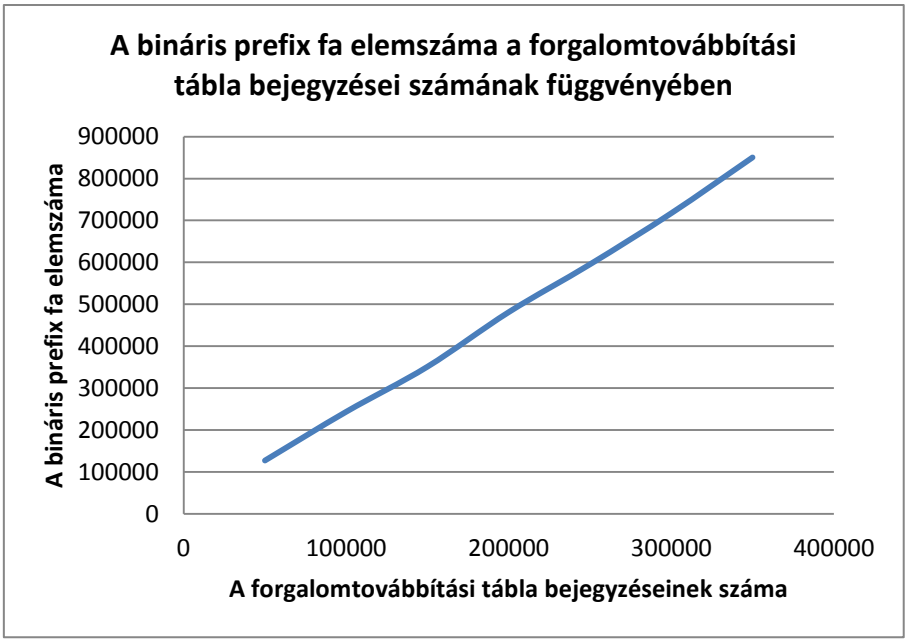
19. ábra: A tömörítési algoritmusok hatékonysága valós forgalomtovábbítási táblákon

A fenti eredményeket minden egyes módszernél átlagolva azt az eredményt kaptam, hogy a komplett bináris prefix fa módszere az eredeti bináris prefix fához képest 85,21%-os, az optimális forgalomtovábbítási tábla módszere 31,12%-os, míg a prefix fa hajtogatás módszer 9,85%-os tömörítési arányt tudott elérni.

Ezen kívül kiszámoltam a valós forgalomtovábbítási táblák átlagos méretét is: ez 7300 kB. Ehhez képest a prefix fa hajtogatás módszerével a táblák átlagos mérete összenyomható 719 kB-ra.

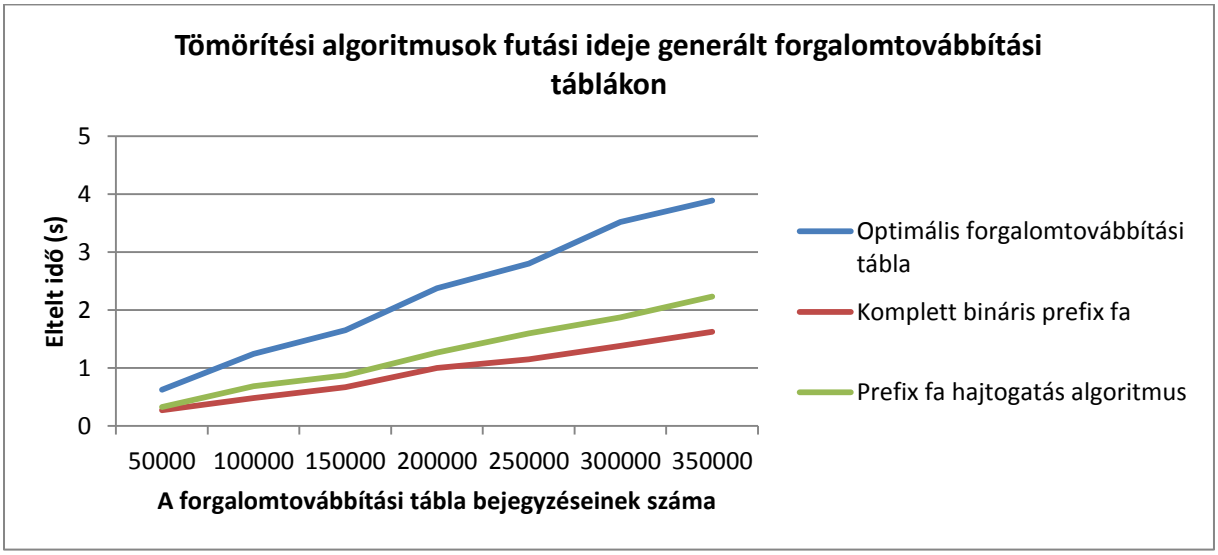
5.2.2 A tömörítési algoritmusok futási ideje

A különböző tömörítési algoritmusok hatékonyságának vizsgálata mellett fontos tényező az is, hogy az adott tömörítés lefutása mennyi időt vesz igénybe. Ehhez újabb forgalomtovábbítási táblákat generáltam, ezúttal egy valós táblát feldarabolva. Ehhez az as1221 nevű valós forgalomtovábbítási táblát vettem alapul (melyben az összes bejegyzések száma 379 489), ezt linux parancssorban 7 darab külön fájlba vágtam az első 50 000, 100 000, stb számú bejegyzést, majd ezeken a táblákon futtattam le a tömörítési algoritmusokat. Azonban ahhoz, hogy biztos lehessek benne, hogy a bejegyzések száma és a tömörítendő bináris prefix fa elemszáma arányos, egy előzetes mérést kellett végeznem, ennek eredményét a 20. ábra mutatja. Eszerint valóban egyenesen arányos a forgalomtovábbítási tábla bejegyzéseinek száma, és a belőlük felépített bináris prefix fa elemszáma.



20. ábra: A bináris prefix fa elemeinek száma a forgalomtovábbítási tábla bejegyzései számának függvényében

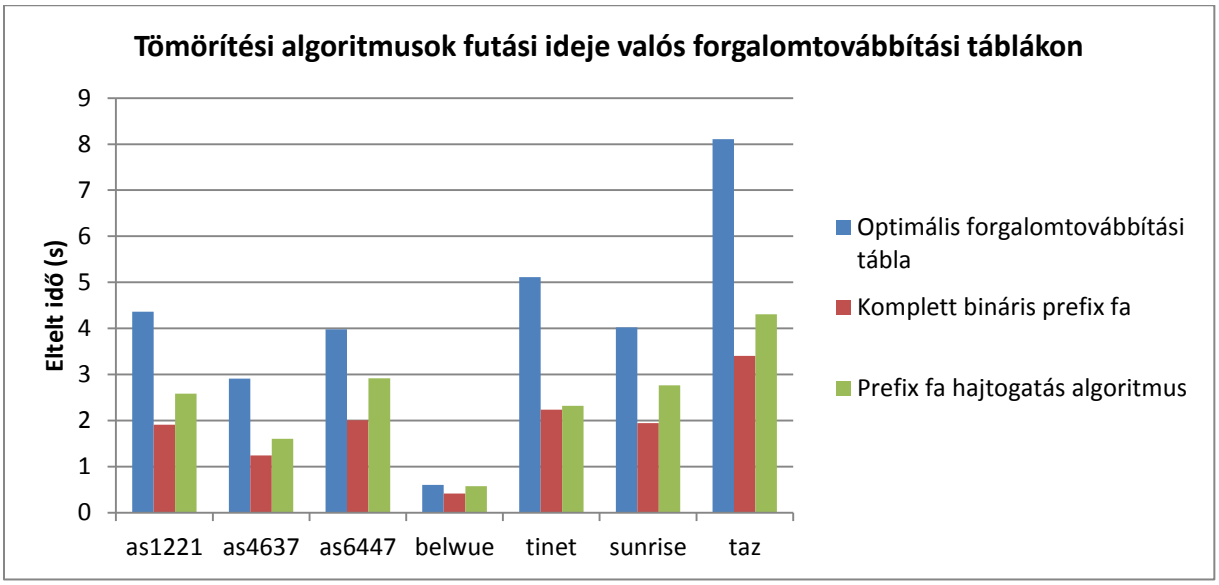
A futási idők mérésének eredményeit a 21. ábra szemlélteti (itt a bináris prefix fa nincs benne az összehasonlításban, hiszen ott nincs effektív tömörítés, egyedül a bináris fa felépítése telik időbe, de mivel ezt mindegyik módszer tartalmazza, nem mértem külön).



21. ábra: A tömörítési algoritmusok futási ideje generált forgalomtovábbítási táblákon

Az eredményekből könnyedén leolvasható, hogy a komplett bináris prefix fán alapuló tömörítés futási ideje a leggyorsabb. Ez elég természetesen adódik, hiszen ez mindkét másik módszer magában foglalja. Az optimális forgalomtovábbítási tábla tömörítési módszerhez volt szükség a legtöbb időre, míg a prefix fa hajtogatás módszere csupán tizedmásodpercekkel lassabb a komplett bináris prefix fa

módszernél. Ezután megmértem a hatékonysági vizsgálatok során már megismert valós forgalomtovábbítási táblákon is az algoritmusok futási idejét, ennek eredményét a 22. ábra mutatja.



22. ábra: A tömörítési algoritmusok futási ideje valós forgalomtovábbítási táblákon

5.3 A részfaindex adatstruktúra hatékonysága

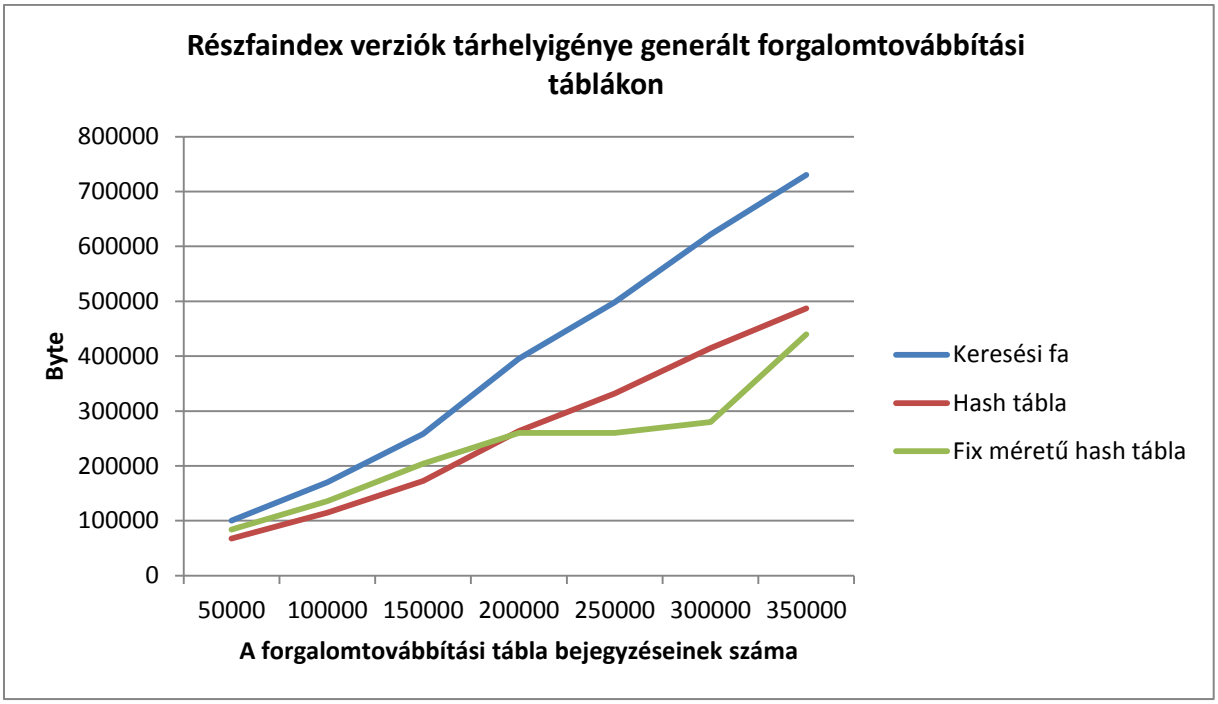
Mivel az nyilvánvalóan látszik az eddigi eredményekből, hogy a prefix fa hajtogatás módszerének hatásfoka messze felülmúlja a bemutatott versenytársakat, ezért a további optimalizációja végett alaposabb szemügyre vettem a különböző részfaindexek tárhelyigényét, és kiszolgálási idejét. Mindegyik mérésre igaz, hogy a vektor alapú részfaindex szerinti megvalósítás hiányzik belőle, mert bár elméletben az *O(1)* nagyságrendű kiszolgálási ideje miatt hatékonynak tűnt, mégis már az implementáció során is bemutatott hátrányai, vagyis az átméretezés időigényes művelete, és az óriási tárhelyigény miatt nyilvánvalóan látszott, hogy a gyakorlatban nem alkalmas a részfaindex adatstruktúra megvalósítására.

5.3.1 A részfaindexek tárhelyigénye

A piros-fekete fa, hash tábla, fix méretű hash tábla alapú megoldások memóriaigényét az előző pontban bemutatott ’feldarabolt’ forgalomtovábbítási táblákon mértem le. Arra voltam kíváncsi, hogy mennyi memóriát használnak fel az egyes részfaindex típusok a forgalomtovábbítási tábla darabok tömörítéséhez.

A fix hash-es megoldásnál abba a problémába ütköztem, hogy mivel explicit módon kell megadni a felhasználható memóriát, nehezen lehet összehasonlítani a másik módszerekkel. Ahhoz, hogy mégis egy diagramon tudjam ábrázolni az összes módszer memóriaigényét, azt a kikötést tettem, hogy a fix hash alapú részfaindexszel való tömörítés eredményéül kapott DAG elemszáma 10%-os hibahatáron belül haladhatja meg a piros-fekete fán illetve a hash táblán alapuló részfaindexek által

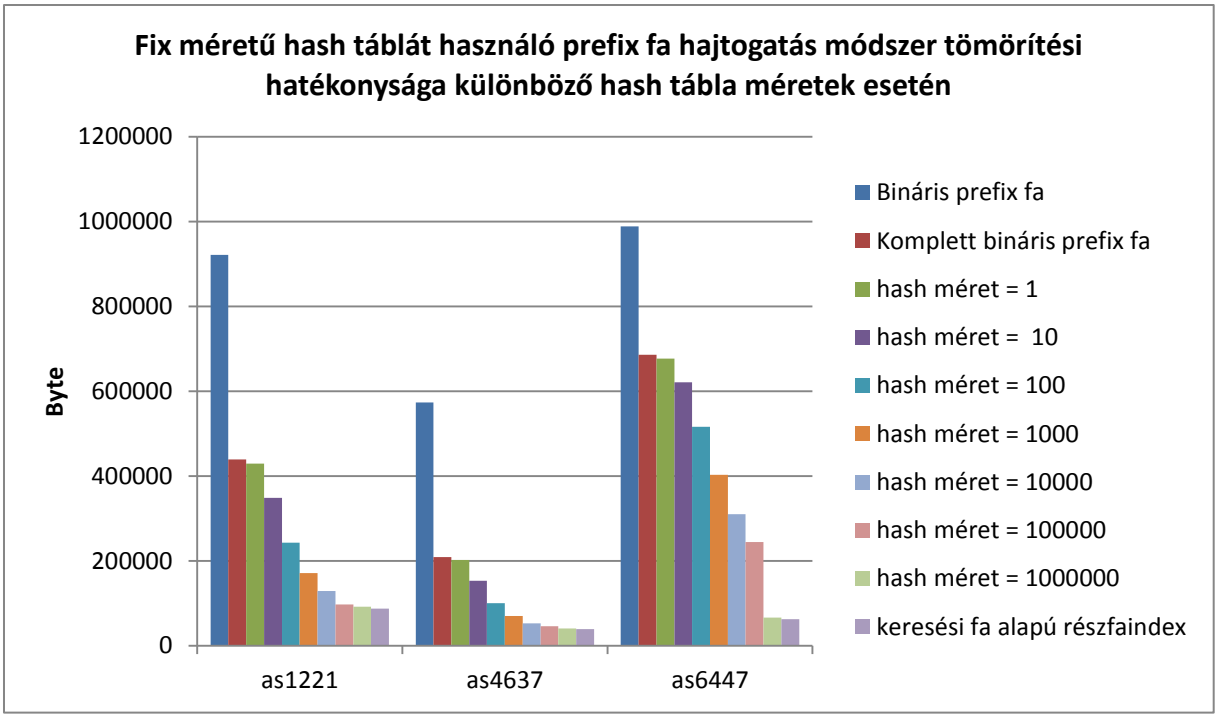
automatikusan teljesített minimális DAG elemszámot. A különböző verziók memóriaigényének mérési eredményei a 23. ábrán láthatók:



23. ábra: A részfaindex verziók tárhelyigénye generált forgalomtovábbítási táblákon

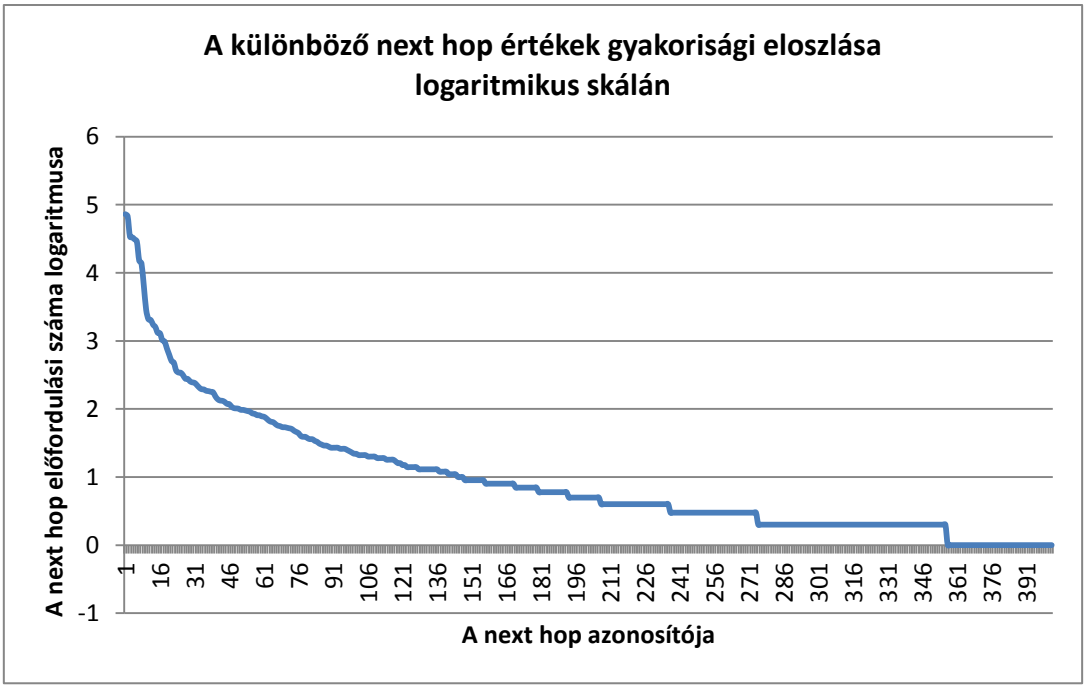
A diagramon jól látható, hogy a piros-fekete fa alapú megvalósítás valamennyivel több memóriát igényelt, mint társai, míg a hash tábla és fix méretű hash tábla alapú részfaindex verziók nagyságrendileg ugyanannyit „fogyasztanak”.

Ugyanakkor a fix méretű hash táblán alapuló részfaindexszel kapcsolatban az is érdekes kérdés lehet, hogy ha a hash tábla méretét, mint paramétert változtatjuk, mekkora tömörítés érhető el vele. A 24. ábrán ezt szemléltetem három valós forgalomtovábbítási táblán. Az ’*x*’ tengelyen van feltüntetve az eredeti, tömörítetlen forgalomtovábbítási táblák pontjainak száma, a komplett bináris prefix fa tömörítés eredménye, ami, mivel a prefix fa hajtogatás algoritmus részét képezi, ezért minimum ennyit a fix méretű hash alapú részfaindex felhasználásával is tömörít az eredeti bináris prefix fán. Ezután 10-es nagyságrendenként növelve a hash tábla méretét feltüntettem a tömörítés nagyságát, végül pedig referenciaként a piros-fekete fa alapú részfaindex eredményét, ami a prefix fa hajtogatás módszerrel elérhető maximális tömörítést nyújtja.

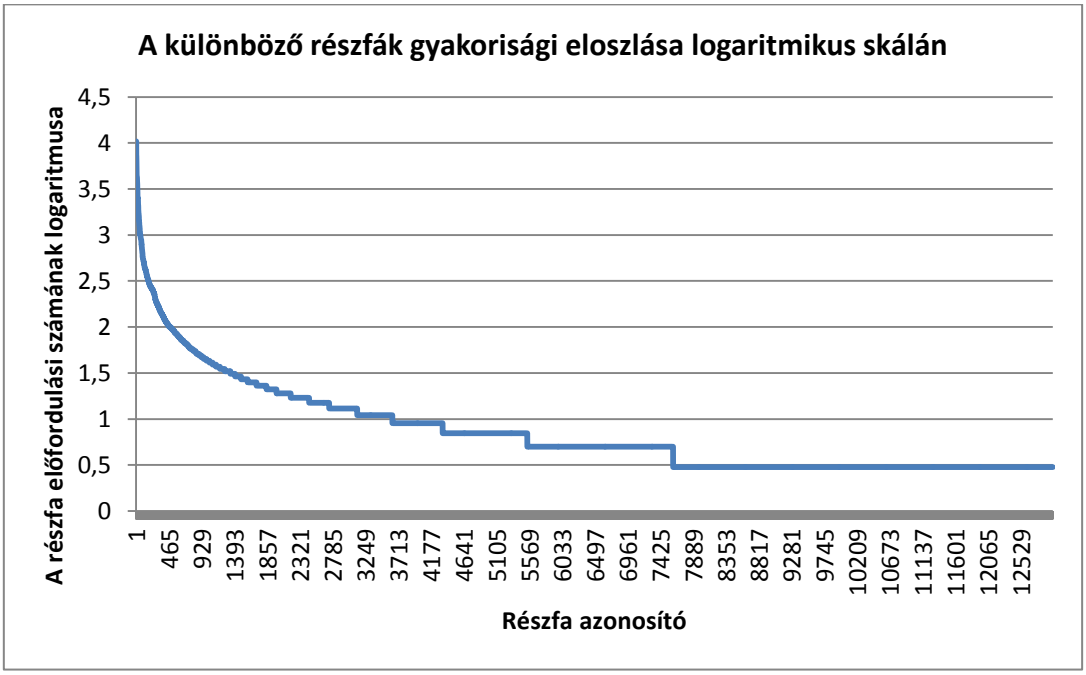


24. ábra: Fix méretű hash táblát használó prefix fa hajtogatás módszer tömörítési hatékonysága különböző hash tábla méretek esetén

Jól látható, hogy már egészen kis hash táblákkal is rendkívül jelentős memória-megtakarítás érhető el. Ez fontos konklúzió, ugyanis azt mutatja, hogy hatékony működésre képes a fix méretű hash táblán alapuló részfaindex-szel megvalósított prefix fa hajtogatás algoritmus. Ennek magyarázata a 25. és 26. ábrákon látható. Ezek közül az elsőn a prefix fa különböző next hop értékeinek eloszlását mutatja, míg a második, a különböző részfák gyakoriságát bemutató diagramról leolvasható, hogy azok eloszlása rendkívül szélsőséges, vagyis néhány részfa a bináris prefix fában nagyon gyakran szerepel, míg a legtöbb részfa csak egyszer fordul elő.



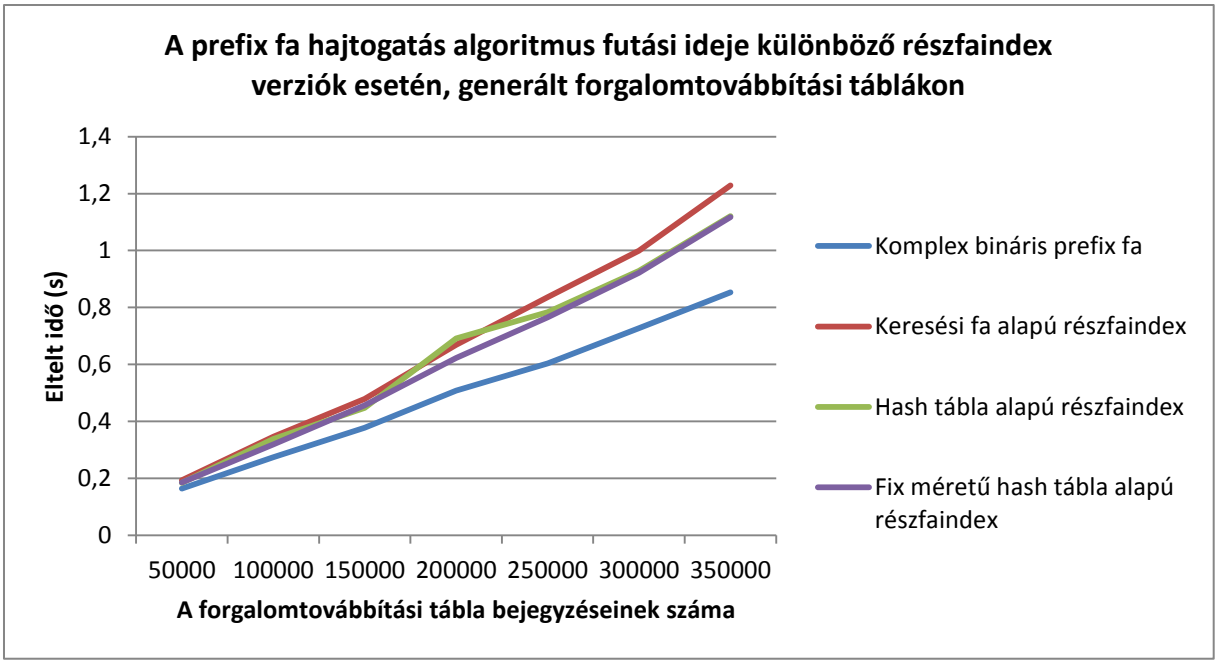
25. ábra: A különböző next hop értékek gyakorisági eloszlása logaritmikus skálán



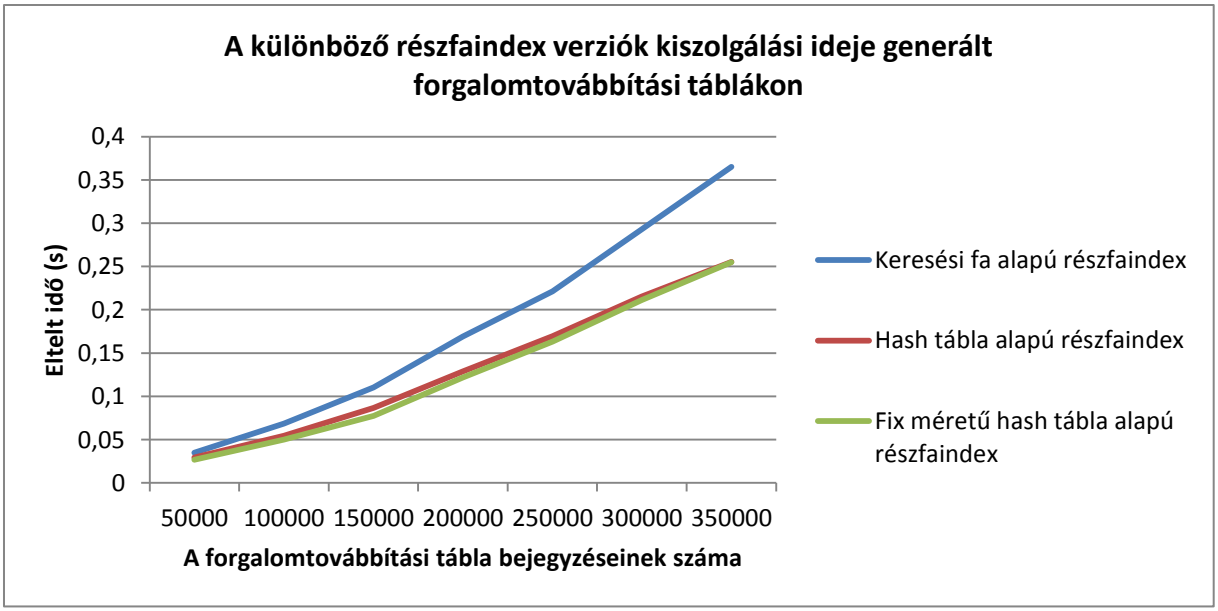
26. ábra: a különböző részfák gyakorisági eloszlása logaritmikus skálán

5.3.2 A részfaindexek kiszolgálási ideje

A prefix fa hajtogatás tömörítéséhez szükséges időket különböző részfaindex verziók esetén, valamint az egyes részfaindex verziók kiszolgálási idejét szintén a fent említett valós forgalomtovábbítási táblákon, illetve azok feldarabolt változatain mértem le. A 27. ábra mutatja a prefix fa hajtogatás módszer kiszolgálási idejeit, a 28. ábrán pedig a részfaindex verziók kiszolgálási idejei láthatók.



27. ábra: A prefix fa hajtogatás algoritmus futási ideje különböző részfaindex verziók esetén, generált forgalomtovábbítási táblákon

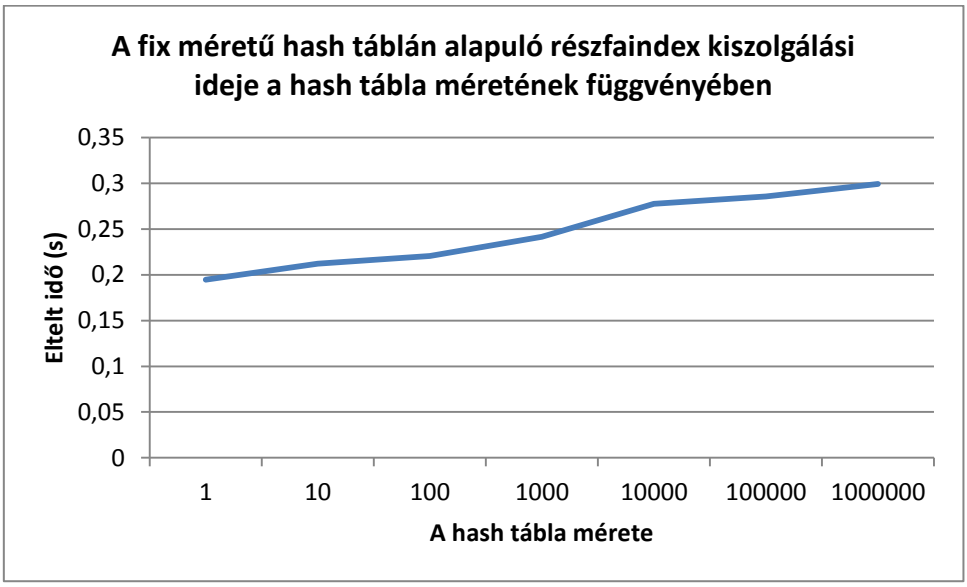


28. ábra: A különböző részfaindex verziók kiszolgálási ideje generált forgalomtovábbítási táblákon

Jól látható, hogy a piros-fekete fán alapuló megoldás a társainál több időt vesz igénybe, míg a hash tábla és fix méretű hash tábla alapú részfaindexeknek szinte pontosan ugyanannyi időre van szükségük a tömörítéshez.

Fontos megjegyezni, hogy az eddigi méréseknél a fix méretű hash tábla alapú részfaindexnél a szabad paramétert, vagyis a hash tábla méretét a jelenlegi alapbeállításon, 10 000-en hagytam. Ezzel nem feltétlenül ér el ugyanolyan mértékű tömörítést, mint társai. Épp ezért azt is megvizsgáltam, hogy hogyan változik a módszer kiszolgálási ideje a hash tábla méretének függvényében.

A 29. ábrán jól látszik, hogy bár volt némi növekedés a felhasznált időt tekintve, mégis azt lehet mondani, hogy a hash tábla mérete csak kis mértékben befolyásolja a szükséges idő mennyiségét, hiszen miközben a tábla méretét 1-ről 1 000 000-ra növeltem, a felhasznált idő csak kicsit több mint egy tizedmásodperccel változott. Ez alapján elmondható, hogy pusztán a tömörítéshez szükséges időt tekintve a három részfaindex megvalósítás közül a piros-fekete fa alapú szerepelt a legrosszabbul, míg a hash tábla és fix méretű hash tábla alapú módszerek kiszolgálási ideje között nincs számottevő különbség.



29. ábra: A fix méretű hash táblán alapuló részfaindex kiszolgálási ideje a hash tábla méretének függvényében

5.4 Az eredmények értékelése

A fejezetben bemutatott mérések eredményeiből az alábbi tanulságokat vonhatjuk le:

- A prefix fa hajtogatás módszer nagyon hatékony tömörítést tesz lehetővé. Az eredeti bináris prefix fa elemeinek számával lineárisan arányos idő alatt lefut, és a forgalomtovábbítási táblákat átlagosan 9,85%-ukra képes betömöríteni. Ennek hozományaként a hálózati csatolókártyákban használt gyorsítótárban (cache) a forgalomirányítási tábla jóval nagyobb hányada fér el. Az elterjedt útvonalválasztókban ehhez tipikusan 256 vagy 512 kB méretű modulokat használnak. Ezekbe a tömörítetlen bináris prefix fa 3,5 illetve 7,0%-a fér el, míg a prefix fa hajtogatás módszer használata esetén ez az arány 35,6 illetve 71,2%-ra ugrik, ezzel jelentősen gyorsítva a forgalomtovábbítás feladatát.
- Az is kiderült, hogy nem mindegy, a prefix fa hajtogatás algoritmushoz milyen szervezésű részfaindex struktúrát használok. A kiszolgálási idejüket tekintve a legeredményesebbek a hash tábla alapú, és a fix hash tábla alapú részfaindexek. De míg a hash tábla alapú megoldásnál *O(N)* nagyságrendű tárhelyre van szükség, addig a fix méretű hash tábla alapú részfaindexnél ez egy veszteséges tömörítéssel, a részfák gyakorisági eloszlásának köszönhetően ez a nagyságrend *O(1)*-re csökkenthető.
- Hatékony módszert adtam arra az esetre, amikor a prefix fa hajtogatás algoritmust a tömörítés hatásfoka szerint szeretnénk optimalizálni: ebben az esetben a hash tábla alapú részfaindex adja a leghatékonyabb működést, és arra az esetre is, amikor a részfaindex struktúra méretét szeretnénk minimalizálni, de eközben elfogadható szinten tartva a tömörítés mértékét.
- A forgalomtovábbítási táblákban létezhet egyfajta entrópia fogalom. A fenti eredmények felhasználásával komoly kutatás kezdődött a forgalomtovábbítási táblák entrópiájának vizsgálatára, mely kutatás kezdeti eredményei nemrégiben egy konferenciacikkben jelentek meg. [21]

6. Konklúziók, jövőbeli tervek

Dolgozatom fő célkitűzése a prefix fa hajtogatás algoritmushoz tartozó részfa adatstruktúrák teljesítményanalízise, és ez alapján egy hatékony forgalomtovábbítási tábla tömörítő algoritmus kidolgozása volt. Ennek érdekében a 2. fejezetben áttekintettem a lehetséges tömörítési algoritmusok közül néhányat, a 3. fejezetben bemutattam az általam kifejlesztett részfaindex adatstruktúrákat, a 4. fejezetben pedig a szimulációs környezet részleteire tértem ki, végül az 5. fejezetben ismertettem a mérési eredményeimet.

Ezen eredmények első feléből, vagyis a különböző tömörítési algoritmusok vizsgálatából tisztán látható, hogy a prefix fa hajtogatás algoritmusa sokkal hatékonyabb tömörítésre képes társainál, és futási ideje is alacsonynak mondható. A módszer segítségével a valós forgalomtovábbítási táblák átlagosan 9,85%-ukra tömöríthetőek.

A mérések második felében a prefix fa hajtogatás módszerhez tartozó részfaindexek teljesítményelemzését végeztem el. Ebből legelőször az derült ki, hogy a vektor alapú részfaindex a gyakorlatban egyáltalán nem hasznosítható. Ezzel szemben a másik három módszer nagyon jól szerepelt a méréseken. E három közül a piros-fekete fán alapuló részfaindex mondható a legkevésbé optimálisnak, hiszen mind a felhasznált memória mennyiségében, mind pedig a kiszolgálási idejében is enyhén rosszabbul teljesített, mint a hash tábla illetve a fix méretű hash tábla alapú részfaindexek. E kettő közül a hash tábla alapú részfaindex a tömörítés hatékonyságában, míg a fix méretű hash táblán alapuló részfaindex a tárolásra felhasznált memória mennyiségében tudta a legjobb eredményeket produkálni.

Láttuk, hogy egy jó tömörítő algoritmussal, és az azt támogató segéd-adatszerkezettel a forgalomtovábbítási táblák mérete jelentősen csökkenthető. Ez alapján úgy gondolom, hogy nagyon is van létjogosultsága a tömörítő algoritmusok alkalmazásának, és a jövőben esetleg már a standard IP útválasztó protokollok (BGP, OSPF stb.) részét is képezheti.

Ahhoz azonban, hogy ez létrejöhessen, még további vizsgálatokat kell végezni. Az eddigi méréseket ugyanis statikus környezetben végeztem el, a jelenlegi implementáció még nem teszi lehetővé, hogy egy már betömörített forgalomtovábbítási táblába újabb elemet szúrjunk be, módosítsunk vagy töröljünk belőle. Ennek megvalósítása és hatékonyságának mérése jövőbeli feladataim közül az egyik legfontosabb. Amennyiben a dinamikus környezetben való alkalmazás is hatékonynak bizonyul, akkor már valóban közel kerülhetünk ahhoz, hogy egy gyakorlatban is jól működő tömörítő algoritmushoz jussunk, amelynek felhasználása az internetszolgáltatók, és az útvonalválasztók gyártói számára is számos előnnyel járna. Ezek az előnyök többek között gazdasági természetűek, hiszen az útvonalválasztók gyártóinak a rendkívül költséges nagy sebességű memóriamodulokból kevesebbet kellene beszerezniük, és az útvonalválasztók hálózati csatolókártyáira

építeni, így ezen eszközök ára is csökkenhet. Ezen felül az internetszolgáltatók újra hadrendbe állíthatnák régi, leselejtezett útvonalválasztóikat, illetve a most használatban lévők is hosszabb élettartammal üzemelhetnének. Ez pedig nem csak gazdasági, de környezetvédelmi előnyökkel is jár, hiszen kevesebb eszköz gyártása kisebb mértékű ipari szennyezéssel jár. További előnye a forgalomtovábbítási táblák tömörítésének az útvonalválasztás sebességének növekedése, az említett gyors elérésű memória kihasználtságának javulása révén. Ennek számszerű meghatározása pontos méréseket igényel, ez a dinamikus eset mellett szintén további terveim részét képezi.

7. Irodalomjegyzék

[1] Zhao, Xiaoliang, Dante J. Pacella, and Jason Schiller. "Routing scalability: an operator's view." Selected Areas in Communications, IEEE Journal on 28.8 (2010): 1262-1270.

[2] Khare, Varun, et al. "Evolution towards global routing scalability." Selected Areas in Communications, IEEE Journal on 28.8 (2010): 1363-1375.

[3] Meyer, David, Lixia Zhang, and Kevin Fall. "Report from the IAB Workshop on Routing and Addressing." RFC2439, September (2007).

[4] Zhao, Xin, et al. "On the aggregatability of router forwarding tables." INFOCOM, 2010 Proceedings IEEE. IEEE, 2010.

[5] Song, Haoyu, et al. "Scalable IP lookups using shape graphs." Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on. IEEE, 2009.

[6] Katajainen, Jyrki, and Erkki Mäkinen. "Tree compression and optimization with applications." International Journal of Foundations of Computer Science 1.4 (1990): 425-448.

[7] Péter Cserkúti, Tihamér Levendovszky, Hassan Charaf: "Survey on Subtree Matching" INES, 2006: 216-220

[8] Ruiz-Sánchez, Miguel Á., Ernst W. Biersack, and Walid Dabbous. "Survey and taxonomy of IP address lookup algorithms." Network, IEEE 15.2 (2001): 8-23.

[9] Draves, Richard P., et al. "Constructing optimal IP routing tables."INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. Vol. 1. IEEE, 1999.

[10] Sahni, Sartaj, and Kun Suk Kim. "Efficient construction of multibit tries for IP lookup." IEEE/ACM Transactions on Networking (TON) 11.4 (2003): 650-662.

[11] Hsieh, Sun-Yuan, Yi-Ling Huang, and Ying-Chi Yang. "Multiprefix trie: A new data structure for designing dynamic router-tables." Computers, IEEE Transactions on 60.5 (2011): 693-706.

[12] Srinivasan, Venkatachary, and George Varghese. "Fast address lookups using controlled prefix expansion." ACM Transactions on Computer Systems (TOCS)17.1 (1999): 1-40.

[13] Tzeng, M-F. "Routing table partitioning for speedy packet lookups in scalable routers." Parallel and Distributed Systems, IEEE Transactions on 17.5 (2006): 481-494.

[14] Li, Zhenqiang, Dongqu Zheng, and Yan Ma. "Using bit selection to do routing table lookup." Frontiers in Algorithmics (2007): 204-215.

[15] Sahni, Sartaj, and Haibin Lu. "Dynamic tree bitmap for IP lookup and update."Networking, 2007. ICN'07. Sixth International Conference on. IEEE, 2007.

[16] Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2001

[17] Sleator, Daniel Dominic, and Robert Endre Tarjan. "Self-adjusting binary search trees." Journal of the ACM (JACM) 32.3 (1985): 652-686.

[18] Library of Efficient Models and Optimization in Networks: http://lemon.cs.elte.hu/trac/lemon

[19] Boost C++ Libraries: http://boost.org

[20] Internet Looking Glass szerverek: http://www.bgp4.as/looking-glasses/ http://bgp.potaroo.net/

[21] Gábor Rétvári, Zoltán Csernátony, Attila Kőrösi, János Tapolcai, András Császár, Gábor Enyedi, and Gergely Pongrácz: "Compressing IP forwarding tables for fun and profit. " Eleventh ACM Workshop on Hot Topics in Networks (HotNets-XI). ACM, 2012.