Alkalmazott Matematikai Lapok **37** (2020), 339–363.
DOI: 10.37070/AML.2020.37.2.15

HIBRID ADATSZERKEZET HALMAZMŰVELETEK HATÉKONY IMPLEMENTÁLÁSÁHOZ

BERTÓK BOTOND

Kombinatorikus vagy egészváltozós optimalizálási algoritmusok leírása gyakran tartalmaz halmazokat és azokon végzett műveleteket. A kombinatorikus algoritmusok között sok az elméletileg is nagy számítási bonyolultságú, ezért a praktikusan megoldható feladatok mérete nagyban függ a megvalósítás minőségétől. Ugyanakkor a halmazműveletek számítógépes implementációja nem kézenfekvő. Ahogy a cikkben bemutatom, nagyságrendi különbségek lehetnek egyes halmazműveletek sebességei között annak függvényében, hogy a halmaz valójában milyen adatszerkezetet takar.

Új eredményként egy olyan hasítótábla-bitvektor hibrid adatszerkezetet javasolok objektumhalmazok számítógépes implementációjára, mely akár több nagyságrenddel gyorsabb futási teljesítményre képes, mint a legújabb fordítóprogramokhoz mellékelt C++ függvénykönyvtári adatszerkezetek.

1. Bevezetés

Munkám során üzleti és műszaki folyamatok optimalizálásával és számítógépes döntéstámogatásával foglalkozom, járműütemezéstől [1] az elektromos hálózatok terheléselosztásáig [2] számos területen, ami magában foglalja a támogatandó feladatok formalizálását [3] és támogató algoritmusaik hatékony implementálását [4]. Ebben a fejezetben azt mutatom be, hogyan fogalmazódott meg a cikkben javasolt módszerrel megoldandó feladat.

1.1. Folyamathálózat-szintézis

A vegyipari folyamatok hatékonyságára akár nagyságrenddel nagyobb hatással lehet annak topológiája és a hálózat elemeinek megfelelő kiválasztása, mint az elemek finomhangolása, ezért Friedler és Fan a korábbiaktól merőben eltérő megközelítést javasolt folyamattervezésre, amit hálózatszintézisnek neveztek el [5]. Szintézisnek azt a kreatív tevékenységet hívták, ahol a lehetséges építőelemek egy halmazából egy működő hálózat előáll. A feladat megfelelő formalizálása és matematikai megalapozása mellett kezdettől fogva fontosnak tartották a strukturális

döntési alternatívák körének kézben tartását [6], mely struktúrák közül – bizonyos kvantitatív paraméterek mellett – bármelyik lehet optimális. Egy-egy struktúra optimális működési pontjának meghatározását másodlagos feladatnak (analízisnek) tekintették, hiszen a legnehezebb kérdések addigra már eldőltek, de a potenciális struktúrák számának csökkentésével az optimális hálózat keresése is jelentősen gyorsítható [7].

1.1.1. A szuperstruktúra megközelítés

A vegyes egész optimalizálási modellek egész része gyakran tartalmaz strukturális döntéseket, például az építőelemek potenciális kapcsolatairól. Az optimalizálás azonban a lehetőségeknek csak a változókkal és korlátokkal definiált körében keres megoldást, azon kívülre nem képes tekinteni. Tehát, ha a matematikai programozási modell felírásakor a változók köréből vagy tartományából kimarad olyan eset, ami a valóságban megvalósítható vagy akár optimális lenne, azt a programozási modell megoldásaként értelmezett optimalizálás nem fedezheti fel. Gyakorlati példákkal és félrevezető modellekkel is szemléltetve a hiba kockázatát, bevezették a szigorú szuperstruktúra fogalmát, ami egy gyakorlati feladatra tekintve bizonyíthatóan tartalmazza annak alternatív struktúrái között legalább egy optimális megoldását [8]. Folyamathálózatok esetén a feladatot a kitűzött célok (termékek) halmazával, az elérhető erőforrások (nyersanyagok) halmazával és a kettőt közbülső célokon átvezető hálózatban összekapcsolni képes egyes megengedett lépések (műveleti egységek) halmazával definiálták [16]. A szuperstruktúra építésére gyors polinomiális algoritmust adtak [10]. Ezután a szuperstruktúra garantáltan részeként tartalmazza a feladat alternatív megoldásainak szerkezetét, és strukturális értelemben a feladat minden lehetséges megoldását definiálja a szuperstruktúra egy részgráfja, ami megadható a benne szereplő lépések, illetve azok előfeltételeinek és következményeinek halmazával.

1.1.2. Problématérkép

Egy gyakorlati optimalizálási feladat esetén a számba veendő lehetséges lépések feltárásában is segít a maximális struktúra szisztematikus felépítése, mely egy technológiai adatbázisban [12] vagy úthálózatban [13, 5. fejezet] lehatárolja a kiindulástól a célig vezető lehetséges lépéseket. Úgy, ahogyan egy gondos mérnök jár el folyamattervezés esetén, amikor figyelembe veszi egy gyártási feladatban felhasználható összes ismert technológiát [11], de figyelembe veszi a cég piaci pozícióját, megadva az általuk reálisan eladható termékeket és reálisan elérhető nyersanyagokat is. Ezáltal megint egy hatalmas technológiai hálóhoz vagy úthálózathoz jutunk, melynek része lesz a választott technológia lépések vagy bejárt útszakaszok sora az optimális hálózatban.

1.1.3. Problémateret leíró gráf algoritmikus feltárása

Vannak olyan folyamatok is, ahol a közbülső lépések lehetséges előzményei és következményei nem adódnak olyan természetesen, mint egy útvonal közbülső városai. Például egy jármű- vagy személyzet-hozzárendelési feladatban a menetrend és térkép alapján eldönthető, hogy egy túra teljesítése után odaér-e még az erőforrás egy másik túra kezdőpontjára, de ez csak algoritmikusan feltárható, a feladatban nem listaszerűen adott [1]. Ugyanígy egy receptben szereplő tevékenységek előfeltételeinek tranzitív láncolatán keresztül visszakövethető, hogy mely feladat után biztosan nem fog egyik berendezés sem olyan feladatot végezni, aminek a recept szerint korábban kell lennie [18]. Hasonlóan a szétválasztási technológiák összes permutációjának okos szisztematikus figyelembevételével generálható a szétválasztási lépések kimenetén szereplő anyagáramok összetételét adó összes elérhető komponens variáció [17, 5. fejezet]. Ezen példák azt mutatják, hogy a problématér gyakran nem explicit felsorolással, hanem logikai szabályokkal adott, melyek alapján mégis bizonyíthatóan felépíthető egy olyan nagy kiterjedésű gráf, mely részeként tartalmazza a feladat minden kombinatorikusan megengedett megoldását. Az állítás még akkor is igaz, ha a kapcsolódó kvantitatív paraméterek sem pontosan, csak bizonyos diszkrét eloszlással ismertek [19].

Fontos megjegyezni, hogy a problématérkép-gráf ábrázolása nemcsak az egzakt modellezést, de egy feladat megértését is segíti. A könnyű érthetőség ellenőrizhető modell-prototípusokhoz, a prototípusok pedig modellezési sémákhoz vezetnek. Mindemellett az érthetőség és a gráf egy-egy kiemelt részének átláthatósága a felhasználói bizalmat is növeli, ha a modellre épített optimalizálás ipari vagy üzleti döntéstámogatás részévé válik.

1.2. Gráfok, halmazok, diszkrét optimalizálás

Ha diszkrét struktúrák, gráfok és hálózatok formális leírásába kezdünk, tanulásunk első lépései között találkozunk a halmazokkal és azokon végzett műveletekkel [9]. Ugyanígy igaz ez a folyamathálózatok leírására [16] és algoritmusaira [5]. Programozás vagy algoritmuselmélet kurzuson pedig megtanuljuk a – Knuth könyvei óta alapismeretként kezelt – alapvető algoritmusokat [14] és adatszerkezeteket [15], de ritkán találkozunk annak tárgyalásával, hogy milyen tulajdonságú halmazokat milyen adatszerkezettel érdemes megvalósítanunk.

1.2.1. Gráfok leírása halmazokkal

Gráfokat gyakran írunk le halmazokkal, például a csúcsok és élek halmazával. Folyamathálózatok esetén az építőelemek kettőssége miatt a folyamat-gráf, processzus-gráf, vagy röviden P-gráf egy páros gráf. A csúcsok két osztályában az aktivitások (hagyományos terminológiával a műveleti egységek) és az előfeltételeiket és következményeiket adó entitások (technológiai hálózatban a műveleti

egységek be- és kimeneti anyagai) vannak. Az aktivitásokhoz egy halmaz- [16] vagy leképezéspárral [4] adják meg az előfeltételeik illetve következményeik (be- és kimeneteik) halmazát. Ezután az aktivitások vagy entitások egy halmazára is leképezések adják meg azok potenciális előzményeit és hatásait.

1.2.2. A folyamathálózat-szintézis algoritmusok leggyakoribb műveletei

Akár a hálózatszintézis algoritmusok alapjait [5], akár azok megvalósítását [4] tekintjük, szinte minden lépésben halmazműveletekkel találkozunk. Annak érdekében, hogy a halmazműveletek közül tudjuk, hogy melyik milyen hangsúllyal szerepel egy feladat megoldásában, nézzünk néhány példát. Ehhez a P-graph Studio szoftverben [3] szereplő megoldóval megszámoltattam – szakirodalomból ismert komplex feladatokra– az alternatív legjobb megoldások generálása során végrehajtott különböző halmazműveleteket. A tesztfeladatok között szerepel egy klasszikus hálózatszintézis feladat (Process Network Synthesis, PNS)[6], egy szétválasztási hálózat szintézise (Separation Network Synthesis, SNS)[8], egy járműhozzárendelési feladat (Vehicle Scheduling Problem, VSP)[22], egy evakuálási útvonal tervezése (Evacuation Rooute Planning, ERP)[21] és egy teljes ellátási lánc környezeti értékelése (Supply Chain Optimization, SCO)[23]. Mindegyik feladat esetén a relaxációvezérelt optimalizáló eljárással a 100 legjobb hálózat leszámlálását kértem, kivéve az evakuációtervezés feladatnál, ahol (a feladat nagy mérete miatt) csak a két legjobb hálózatot generáltattam. A számolások eredményét az 1. táblázat tartalmazza. Megjegyzem, hogy az unió műveletek számába beleszámoltam a halmazok másolását végző értékadás műveleteket is, hiszen a gyakorlatban ugyanarról van szó: a halmaz minden elemét át kell másolni egy másik halmazba.

1. táblázat. Halmazműveletek előfordulása folyamatszintézis során

Művelet	PNS	SNS	VAP	ERP	SCO	Átlag
unió	251 346	72 181 072	6 878 466	36 500 529	76 971	77,41%
metszet	27 385	8 608 074	230 518	2 660 931	6 442	6,32%
részhalmaz	1 892	393 004	13 102	17 105	501	0,34%
új elem beszúrása	100	95	272	1 729	88	0,03%
ismert elem beszúrása	10 142	5 646 549	118 475	2 506 031	4 596	3,94%
elem tartalmazás	29 550	16 593 572	453 379	10 730 444	7 860	11,96%

A tesztfeladatok lépéseinek eloszlását átlagolva, a táblázat utolsó oszlopa alapján megállapítható, hogy a markánsan legnagyobb arányban végrehajtott halmazművelet az unió, melyet nagyságrenddel kisebb számban az elem tartalmazás vizsgálata és a metszet követ. Tehát a megoldás kombinatorikus részének gyorsításához alapvetően az unió és metszet műveletre, valamint az elemtartalmazás-

vizsgálatra kell koncentrálnunk, olyan implementációt kell keresnünk, mely ezen műveleteket nagyon gyorsan el tudja végezni.

1.3. Célkitűzés

Ebben a publikációban a halmazok és halmazműveletek lehetséges implementációit tárgyalom és hasonlítom össze – felépítésük mellett – gyakorlati futási idejüket mérve különböző tesztkészleteken. Eközben javaslatot teszek egy új típusú hibrid adatszerkezetre is, mely általános objektumhalmazok esetén nagyságrendekkel gyorsabban teszi lehetővé halmazműveletek (elsősorban unió és metszetképzés) végrehajtását, mint hagyományos alternatívái.

2. Halmazokat leíró adatszerkezetek

A halmaz adatszerkezeteket két csoportban vizsgálom. Először azokat, melyek csak természetes számok tárolására alkalmasak. Utána pedig azokat, melyekben tetszőleges adat tárolható.

2.1. Számhalmazok

A mai programozási nyelvek alapvetően háromféle halmazt különböztetnek meg. Első a rendezett halmaz, második a rendezetlen halmaz, harmadik pedig a bithalmaz. Példaként mindhárom esetben tekintsük az $\{1; 3; 5\}$ és az $\{1; 3; 8; 9\}$ halmazokat, valamint ezek unióját.

2.1.1. Rendezett halmaz

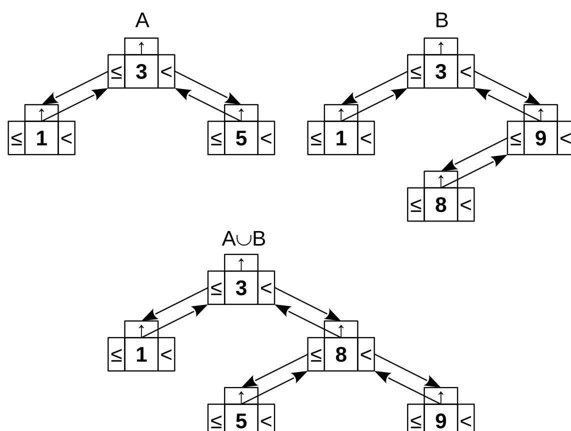
A rendezett halmaz gyakorlatilag egy keresőfa, ahol egy-egy elem várhatóan $log_2(n)$ nagyságrendben beszúrható, törölhető és megtalálható, ha a fa bináris és kiegyensúlyozott, és n a tárolt elemek száma.

Implementációban általában a tárolt elemek értékén kívül mutatók adják meg az adott részfában nála nagyobb, nála kisebb elemek helyét valamely rendezés szerint, továbbá a szülőhelyét, ahogy az 1. ábrán is láthatjuk. Megjegyzem, hogy ugyanazon számhalmazhoz többféle keresőfa is tartozhat, az ábrán egy lehetséges példát, de nem az egyetlen helyes elrendezést láthatjuk. A rendezés számok esetén természetes.

Két halmaz uniójának képzéséhez az egyik keresőfa minden elemét le kell másolnunk, és beletenni mindazon elemeket, melyek ezen túl a másik halmazban szerepelnek. Ez a gyakorlatban számos memóriafoglalási műveletet igényel a fa csúcsainak számára akkor is, ha a halmaz elemeit nem a fában tároljuk, csak hivatkozunk rájuk.

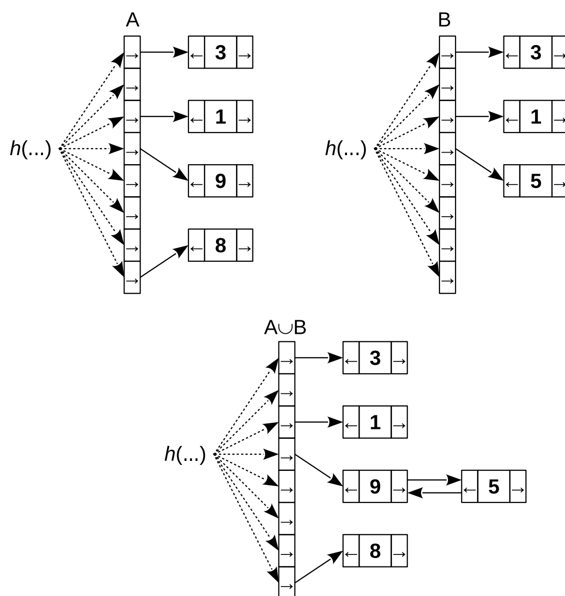
2.1.2. Rendezetlen halmaz

A rendezetlen halmazokat többnyire hasítótáblával implementálják, ahol egy



1. ábra. Számhalmazok és halmazműveletek eredményének leírása keresőfával.

$h()$ hasítófüggvény adja meg, hogy egy adott elem melyik tárolóba kerüljön. Jó esetben minden tárolóban, ha nem is egyetlen, de kevés korlátozott számú elem szerepel. Az egy tárolóban levő elemeket valamilyen dinamikus tárolással felsoroljuk, például láncolt listával, ahogy a 2-es ábrán látható.



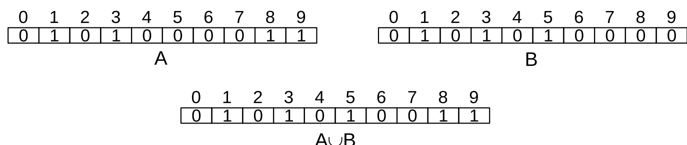
2. ábra. Számhalmazok és halmazműveletek eredményének leírása hasítótáblával.

A hasítótábla előnye, hogy általában adható olyan $h()$ függvény, ami konstans időben kiszámítható, és segítségével közvetlenül a keresett tárolóhoz jutunk, melyben az előre rögzített korlátos elemszám miatt korlátos fix lépésben biztosan megtaláljuk a keresett elemet. Ezért várható értékben egy elem megtalálása gyorsabb, mint a keresőfánál.

Unió képzéséhez itt is minden elemet át kell másolnunk egyik tárolóból a másikba, majd mellé tenni a másik tároló kiegészítő elemeit. Ez is számos memóriafoglalási művelettel jár.

2.1.3. Bithalmaz

A halmazok egy speciális változata, amit az angol irodalomban bithalmazként, a magyar szaknyelvben inkább bitvektorként emlegetünk. Itt egy tömbben az egyes pozícióban szereplő bitek jelzik, hogy a pozíciónak megfelelő szám szerepel-e a halmazban. Tehát, ha például a 2-es sorszámú bit értéke 0, akkor a 2-es szám nem szerepel, ha az 1-es sorszámú bit értéke 1, akkor az egyes szám szerepel a halmazban, lásd 3. ábra.



3. ábra. Számhalmazok és halmazműveletek eredményének leírása bitvektorral.

Ha a 0 értéket hamisnak, az 1 értéket igaznak tekintjük, akkor két bithalmazzal ábrázolt számhalmaz unióját a bitenkénti „vagy”műveletel számolhatjuk. Ennek megvalósítása rendkívül gyors lehet, mert ezt a műveletet minden mai CPU ismeri, és egyszerre végre tudja hajtani annyi bitre, amennyi az adatregiszterek hossza, ami manapság tipikusan 64. Fontos megjegyezni, hogy ebben a leírásban csak természetes számokat tudunk tárolni, vagy olyan adatokat, melyek kölcsönösen egyértelműen leképezhetőek természetes számokra. A tároló mérete pedig nem az elemek számától, hanem a számok értelmezési tartományától függ. Tehát, ha a halmazban csak két szám, a 0 és a 10 000 szerepel, akkor is egy legalább 10 001 bit hosszú tömbre van szükségünk.

2.1.4. Indexelt halmaz

Újdonságként egy olyan halmazleírást javaslok, amely ötvözi a fentiekben bemutatott halmazleírások előnyeit. Ehhez olyan módszerre van szükség, mely tetszőleges objektumokhoz tud természetes számokat rendelni olyan módon, hogy az alábbi kérdések mindegyike gyorsan megválaszolható legyen:

- Egy objektumnak van-e már sorszáma?
- Mi egy már ismert elem sorszáma?
- Egy adott sorszám melyik objektumhoz tartozik?

Tegyük fel, hogy egy keresőfában vagy hasítótáblában nem a halmaz, hanem az értelmezési tartomány elemeit tároljuk, de csak azokat, melyek a halmazműveletek során legalább egyszer előfordultak már. Például ahhoz, hogy egy úthálózatban leképezzük a 73-as és 7302-es út elágazóját, nem kell minden számot kezelnünk 73-tól 7302-ig, melyek olyan utakat jelölnek, amik az ország másik végében vannak.

Ezután az értelmezési tartomány már előfordult elemeihez sorra egy-egy hivatkozást illesztünk egy tömbben, ezáltal egy mesterséges hivatkozási számot vagy indexet rendelve hozzájuk, mely nullától indul és az új elemekkel egyenként nő. Tehát a példánkban a 73-as úthoz hozzárendeljük a 0-s, a 7302-es úthoz pedig az 1-es indexet, így a két elemet a 73-7302 számtartomány helyett a 0-1 index tartománnyal azonosítjuk. Egyúttal az indexet a keresőfában vagy hasítótáblában tárolt elemek mellé is bejegyezzük, ahogy a 4-es és 5-ös ábrán látható. Ezután az elemek halmazait az indexek bithalmazával írjuk le. Ennek eredményeként:

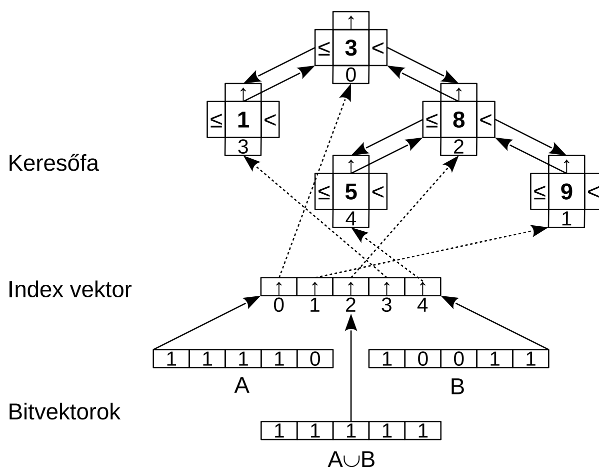
- A keresőfa vagy hasítótábla egyetlen példányban létezik csak, a lehető leggyorsabban megadva egy tetszőleges objektumhoz a mellé bejegyzett indexet.
- A halmazműveletek során a keresőfa vagy hasítótábla nem változik, hiszen már ismert elemekkel dolgozunk, nincs szükség az elemek egyenkénti helyfoglalására és másolására.
- A halmazműveletek bitenkénti logikai műveletekkel elvégezhetők, ugyanúgy és ugyanolyan gyorsan, mint a bithalmazoknál.
- A bithalmazzal ellentétben nem feltétlenül csak számokat tudunk tárolni.
- A bitvektor hossza a műveletek során legalább egyszer előforduló elemek számától és nem az értelmezési tartományától függ.

Ettől a leírástól azt remélem, hogy a korábbiaknál praktikusan sokkal gyorsabban futó halmazműveleteket eredményez. Ezt a későbbi fejezetekben tesztekkel vizsgálom.

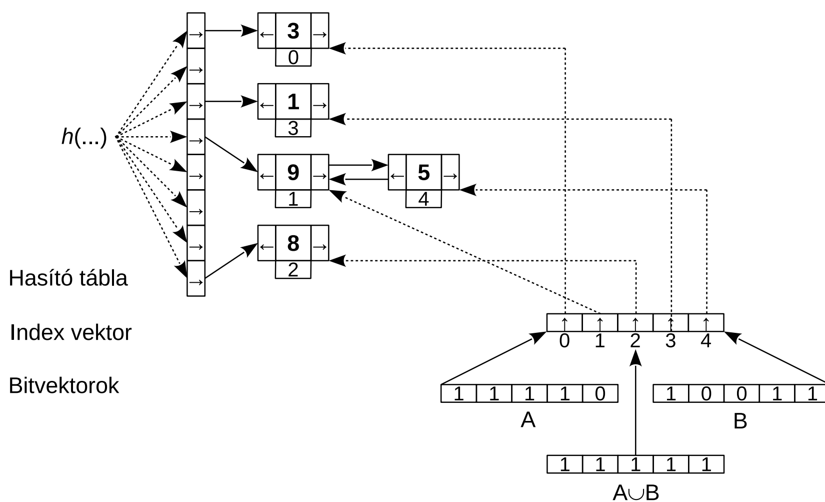
2.2. Univerzális halmazok

Ebben a fejezetben bemutatom, hogy ha nem számokat, hanem tetszőleges objektumot akarunk halmazban tárolni, akkor milyen adatszerkezetet használhatunk. Példaként karakterláncok tárolását tekintjük.

Gondolhatnánk persze, hogy bármit meg tudunk számozni, és onnantól használhatjuk az előző fejezetben javasolt adatszerkezetek bármelyikét. Ugyanakkor



4. ábra. Számhalmazok és halmazműveletek eredményének leírása keresőfával implementált indexelt halmazzal.

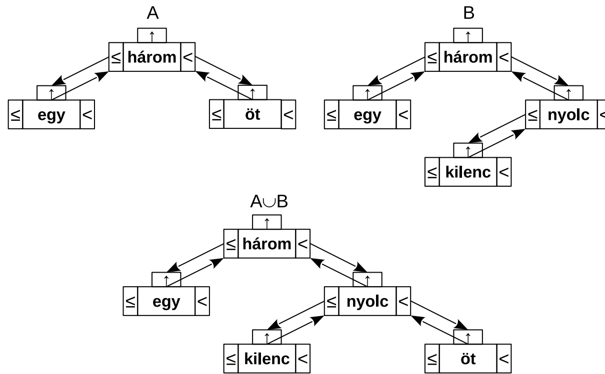


5. ábra. Számhalmazok és halmazműveletek eredményének leírása hasítótáblával implementált indexelt halmazzal.

a számozással pontosan azokba a kérdésekbe ütközünk, melyeket a 2.1.4. fejezetben tárgyaltam, és melyekre adott válaszok a javasolt adatszerkezethez elvezettek. Mindemellett a programozási nyelvek több univerzális beépített adatszerkezetet tartalmaznak objektumok halmazok tárolására.

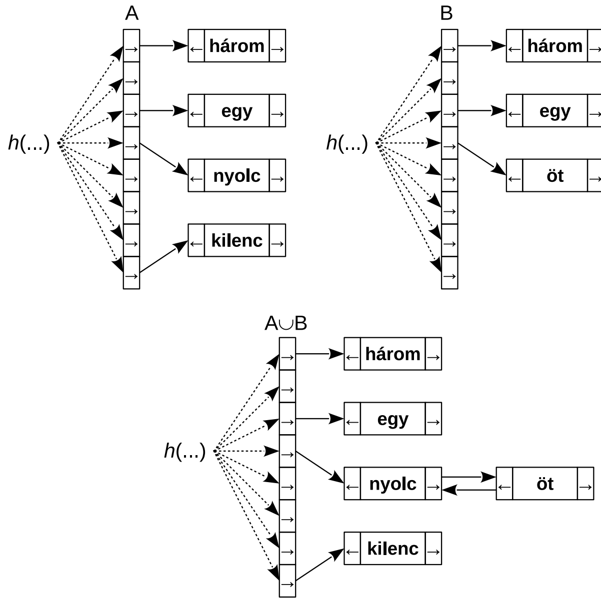
2.2.1. Rendezett halmaz

Ahogy az előző fejezetben láttuk, a rendezett halmaz a gyakorlatban egy keresőfa. Keresőfában minden olyan elem eltárolható, melyen értelmezhető rendezés. Ez karakterláncok esetén lehet azok lexikografikus rendezése, ahogy a 6. ábra mutatja.



6. ábra. Objektumhalmazok és halmazműveletek eredményének leírása keresőfával.

Általános esetben egy objektum rendezésének értelmezéséhez nem kell az összes adatát felhasználnunk, elég ha valamely egyértelmű azonosítóján értelmezett a rendezés, mert akkor az alapján a keresőfában egyértelműen megtalálható lesz. Ha nem az objektumot, hanem rá való hivatkozást tárolunk, akkor az elem keresését – a számokkal összevetésben – csak az hátráltatja, hogy az elemek rendezés szerinti összehasonlítása valójában több összehasonlítás sorozatára bomlik fel, mint a karaktersorozatok lexikografikus rendezésekor: azonos kezdő karakterek esetén a további karaktereket is vizsgálnunk kell.



7. ábra. Objektumhalmazok és halmazműveletek eredményének leírása hasítótáblával.

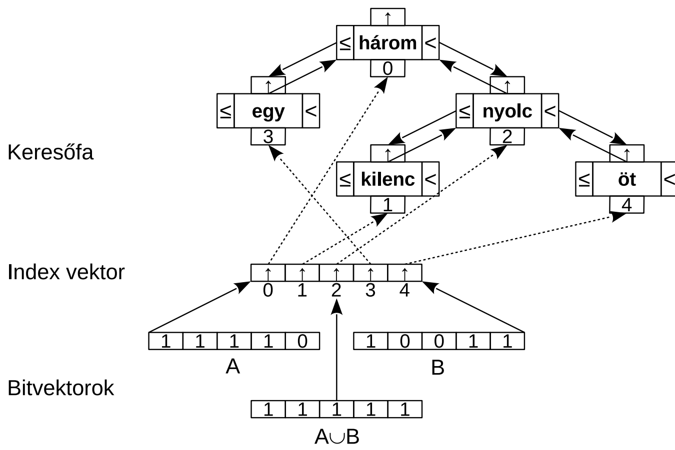
2.2.2. Rendezetlen halmaz

Rendezetlen halmazban való tároláshoz nem szükséges az elemeken rendezésnek léteznie. Ahogy az előző fejezetben is láthattuk, a hasítófüggvény nem feltétlenül rendezett sorrendjét adja az elemeknek. Ugyanakkor tetszőleges objektumra használható hasítófüggvényre van szükségünk, mely informatikai megvalósításban például az objektum bináris ábrázolásának bitmintái alapján hasít olyan módon, hogy továbbra is egy résztárolóba nagy valószínűséggel csak előre tervezett módon korlátos véges számú objektum kerüljön, ahogy azt a 7. ábrán láthatjuk.

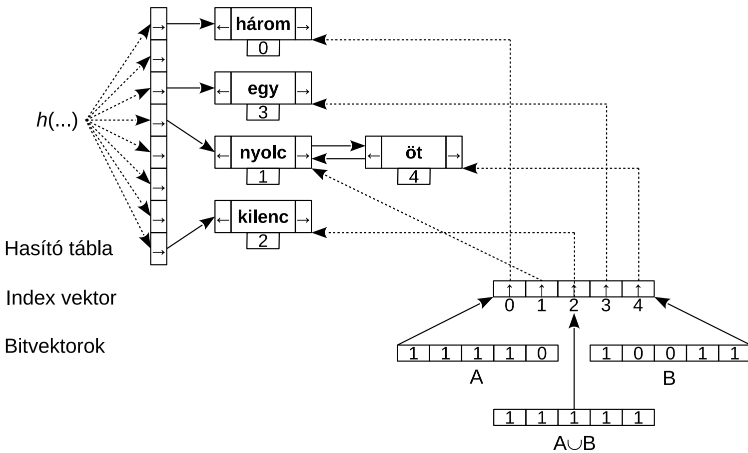
2.2.3. Indexelt halmaz

A 2.1.4. fejezetben bevezetett indexelt halmazok előnye, hogy bár halmazműveleteket ugyanolyan egyszerűen végezhetünk, mint bithalmazokkal, eközben tetszőleges elemet tehetünk bele, ahogy a 8. és 9. ábrán láthatjuk.

Egy ismeretlen elem beszúrása egy ilyen adatszerkezetbe persze még annál is lassabb, mintha csak egy keresőfába vagy hasítótáblába tennénk be, de a megcélzott alkalmazási terület (a folyamathálózat-szintézis eljárások) esetén, új elemek csak addig jelenhetnek meg, míg a szuperstruktúra felépül. Ha minden a szuperstruktúrában szereplő elemnek már van indexe, akkor utána csak a kapcsolódó indexekkel és azok bitvektoraival dolgozunk.



8. ábra. Objektumhalmazok és halmazműveletek eredményének leírása keresőfával implementált indexelt halmazzal.



9. ábra. Objektumhalmazok és halmazműveletek eredményének leírása hasítótáblával implementált indexelt halmazzal.

Például az 1. táblázatban látható PNS feladat maximális struktúrája 35 műveleti egységet és 65 anyagot tartalmaz, tehát összesen 100 gráf csúcsot. A mérési eredmények szerint pontosan 100-szor volt szükség új elem beszúrásra a halmazműveletek során, utána már soha többet, hiába történt több százezer halmazművelet.

3. Adatszerkezetek implementálása

Ebben a fejezetben a halmazok és műveleteik informatikai megvalósítását tekintem át, ami alapján azok gyakorlati sebessége mérhetővé válik.

3.1. Szabványos tárolók

Az adatszerkezetek mindegyikét a lehető legnagyobb sebesség elérése érdekében gépközeli nyelven, C++-ban implementálom, néhány esetben közvetlen gépikódú CPU utasítások beágyazásával.

A C++ fordítók részeként kapunk szabványos, univerzálisan használható adatszerkezeteket is, a szabványos sablon könyvtár azaz Standard Template Library röviden STL elemeként. Az STL elemei a szabványos azaz sztenderd, röviden *std* névtérben találhatóak, ezért az úgynevezett hatókör operátor használatával $std ::$ előtaggal jelöljük őket. Ebben a gyűjteményben megtalálható az $std :: set$, ami egy keresőfa, az $std :: unordered_set$, ami egy hasítótábla. Létezik ugyan egy $std :: bitset$ is, ám annak értelmezési tartománya futási időben nem változtatható, ezért helyette – a fordítóprogram fejlesztéseinek következő lépéseit előre vetítő *boost* könyvtárból– a $boost :: dynamic_bitset$-et használjuk bithalmazként.

3.2. Indexelt halmaz

Az indexelt halmaz megvalósításához a fentiek szerint szükségünk van egy nagyon gyors tároló-kereső adatszerkezetre, ez esetünkben az $std :: unordered_map$ lesz, ami csak annyiban különbözik az $std :: unordered_set$-től, hogy a tárolt objektum mellé az index szerinti sorszámot is be tudjuk tenni másodlagos értékként. Szükségünk van továbbá egy index tömbre, amit az $std :: vector$ biztosít. Az így elkészült univerzális halmazt $objset$-nek neveztem.

3.2.1. Bithalmaz megvalósítása

Az $objset$ belsejében a bithalmazra használhatnánk a $boost :: dynamic_bitset$-et, de egyrészt egy ilyen implementációt már korábban elvégeztem $smallset$ néven, mintsem a $boost :: dynamic_bitset$ létezett volna. Másrészt a $smallset$ megvalósításának részét képezik olyan kódrészek, amelyek a bitenkénti logikai műveleteknél a jelenlegi processzorok multimédiás kiterjesztéseit használva akár 128 bitre is el tudnak végezni egy-egy művelet egyetlen CPU-ban implementált utasítással. Ezen műveletek az SSE (Streaming SIMD Extensions) utasítások között találhatóak, ahol a SIMD a „Single instruction, multiple data”, tehát egyetlen utasítás több adaton rövidítése.

Fontos megjegyezni, hogy a $smallset$ elnevezés arra utal, hogy a halmaz akkor hatékony, ha a tárolt számok értelmezési tartománya kicsi, tehát nem kell sokkal több bitet tárolni, mint ahányféle számmal valójában a műveletek során találkozunk. Ezt majd a mérések alapján is látni fogjuk.

4. Halmazműveletek sebességének tesztelése

Egy teszteléshez 100 véletlen halmazt generálok, köztük számos műveletet végeztetek el mindegyik halmazleírásban. A műveletek mindegyikét 10 000-szer ismételtem, hogy az eltelt idő érdemben mérhető legyen. Végül miden mérést 100 különböző véletlen halmazkészletre ismételtem meg. Tehát összesen minden művelet esetén $100 \times 10\,000 \times 100 = 100\,000\,000$ azaz százmillió művelet idejét mértem meg összesen. A futási idők mindenhol másodpercben értendők. A méréseket egy 3,2 GHz-es Intel i5-8250U processzoron végeztem Linux Mint operációs rendszer alatt.

4.1. Paraméterkészletek

A fentiekben leírtak alapján más-más környezetben az egyes halmazleírások másként viselkedhetnek, ezért négy különböző paraméterkészletet definiáltam:

1. Első esetben 0 és 255 közötti értelmezési tartományt állítok be, és 128 véletlen számot generálok minden halmazba, tehát körülbelül az értelmezési tartomány felében lesznek ténylegesen számok. Valójában valamivel kevesebb szám lesz, mert a generált számok között lehetnek azonosak.

2. Második esetben is 0 és 255 közötti értelmezési tartományt állítok be, de csak 64 véletlen számot generálok minden halmazba, tehát körülbelül az értelmezési tartomány negyedében lesznek ténylegesen számok.

3. Harmadik esetben 0 és 511 közötti értelmezési tartományt állítok be, és megint 64 véletlen számot generálok minden halmazba, tehát körülbelül az értelmezési tartomány nyolcadában lesznek ténylegesen számok.

4. Végül extrém példaként 0 és 99 999 között mindössze 100 számot generáltatok, tehát az értelmezési tartománynak csak az ezrelékét használom valójában.

Az objektumhalmazok teszteléséhez 3 és 8 karakter közötti hosszúságú véletlen karakterláncokat generálok, melyek az angol ábécé betűiből állnak.

4.1.1. További implementációs megfontolások

Az egyes adatszerkezetek összehasonlíthatóságához azonos funkcionalitást kell elérnünk velük akkor is, ha a függvénykönyvtár önmagában nem minden műveletet támogat, ezért az alábbi kiegészítéseket tettem.

A $boost :: dynamic_bitset$ esetén az elemek berakása és törlése a megfelelő bitek $true$-ra illetve $false$-ra állításával történik. A $smallset$ és az $objset$ esetén az elemek berakása a $+=$, kivétele a $-=$ operátorral történik. Az $std :: set$ és $std :: unordered_set$ esetén a beszúrásra az $insert$, törlésre az $erase$ függvényt tudjuk

használni. A generált véletlen számokat minden teszt esetén először egyenként betesszük, majd egyenként kivesszük a halmazból, kivéve az utolsó teszt iterációt, amikor a számok a halmazban maradnak a további halmazműveletek teszteléséhez. Az indexelt halmazok esetén – a fair összehasonlítás érdekében – minden teszt előtt az indexet is töröljük.

A *boots* :: *dynamic_bitset* esetén a tartalmazást az fejezi ki, hogy az adott sorszámú logikai érték igaz-e. A *smallset* és *objset* esetén a tartalmazást a $<$ operátorral tudjuk lekérdezni. Az *std* :: *set* és *std* :: *unordered_set* esetén akkor szerepel egy elem a halmazban, ha rá a *find*() függvény nem *end*() értéket ad.

A *boost* :: *dynamic_bitset* az uniót a $|=$, a metszetet pedig az $\&=$ operátorokkal valósítja meg. A *smallset* és az *objset* szintén támogatják a $|=$ és $\&=$ operátorokat. Az *std* :: *set* esetén a függvénykönyvtár biztosítja a *set_union* és *set_intersection* függvényeket. Sajnos az STL-ben szereplő *set_union* és *set_intersection* műveletek csak rendezett halmazokon értelmezettek, ezért a rendezetlen halmaz esetén csak definíció szerint tudunk uniót és metszetet képezni, tehát unió esetén a másik halmaz elemeit is betesszük az első mellé, míg metszet esetén egy üres halmazba csak azon elemeit tesszük be az egyik halmaznak, ami a másikban is szerepel.

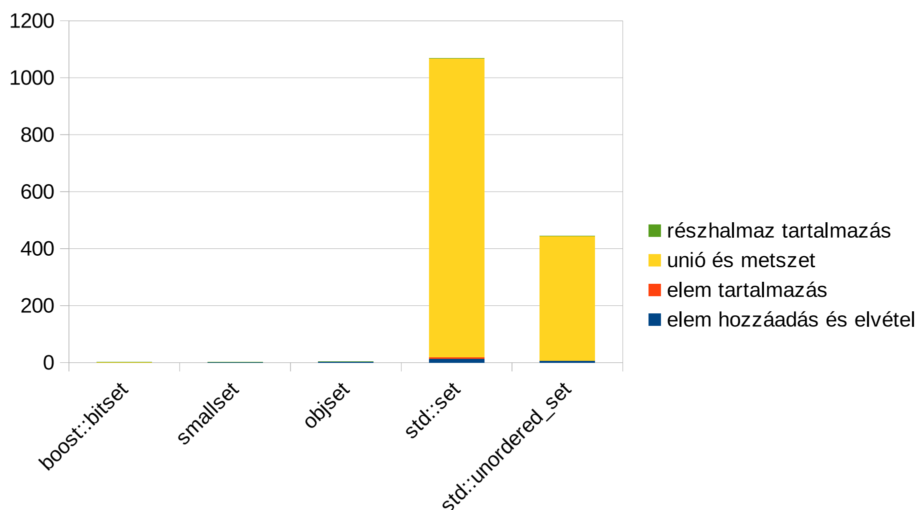
A tesztben a véletlen halmazokon végighaladva minden páros indexűt unió művelettel hozzátettük egy eredményhalmazhoz, minden páratlan indexűvel pedig metszetet képeztünk, és a metszetet tettük vissza az eredményhalmazba. Tehát felváltva bővítjük és szűkítjük az eredményhalmazt.

A részhalmaz tartalmazás vizsgálatára a *boost* :: *dynamic_bitset* tartalmaz egy *is_subset_of* logikai függvényt. A *smallset* és *objset* esetén a részhalmaz tartalmazás a $<$ logikai operátorral vizsgálható. Az *stl* :: *set*-hez a függvénykönyvtár tartalmaz egy *includes* függvényt erre a célra. Sajnos az *includes* függvény csak rendezett halmazra működik, ezért a rendezetlen *std* :: *unordered_set* halmaz esetén csak definíció szerint egyenként vizsgálható, hogy az egyik halmaz minden eleme szerepel-e a másikban.

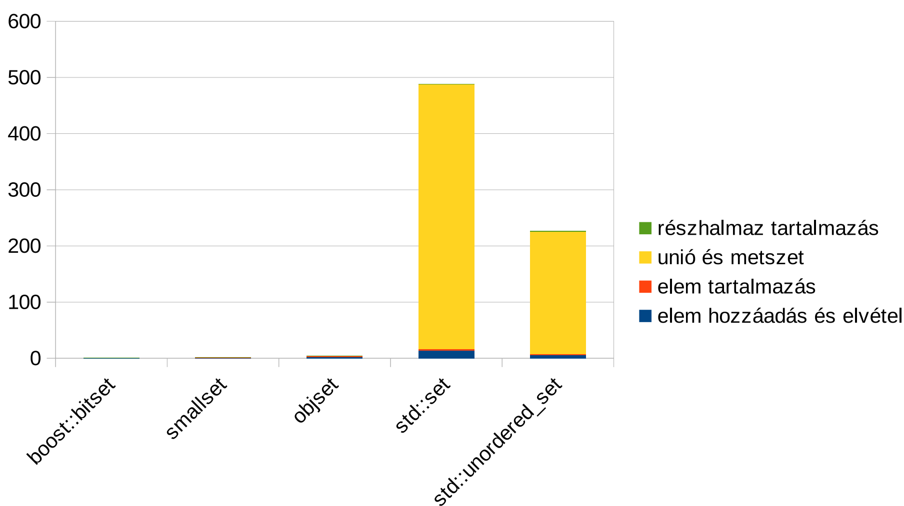
4.2. Futási eredmények számhalmazokra

Ha a 10., 11. és 12. ábrákat végignézzük, akkor azt láthatjuk, hogy az 1-3 paraméterkészletek esetén a *boost* :: *dymamic_bitset*, a *smallset* és az *objset* nagyságrendekkel gyorsabb, mint az *std* :: *set* és *std* :: *unordered_set*. A legnagyobb különbség az unió és metszet műveletekben található. Ami meglepő, hogy az STL tárolók közül a keresőfával megvalósított rendezett halmazhoz képest, a hasítótáblával implementált rendezetlen halmaz gyorsabb körülbelül kétszer, noha az utóbbihoz a függvénykönyvtárban nem is kapunk unió és metszet műveleteket.

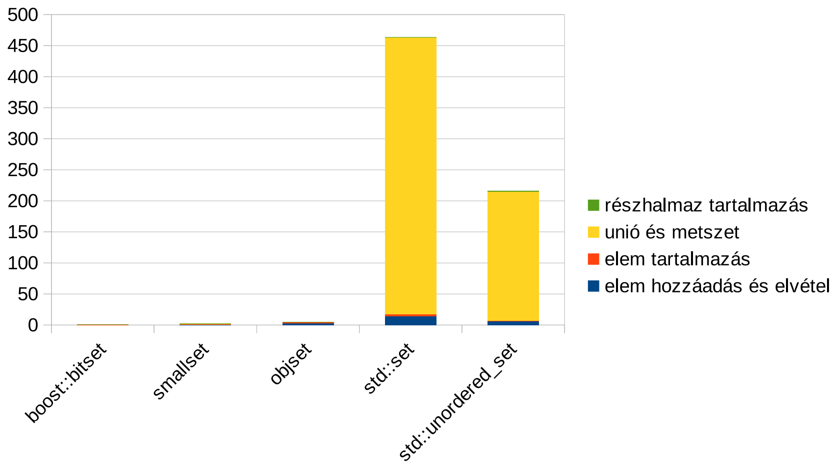
Ha csak külön a *boost* :: *dymamic_bitset*, a *smallset* és az *objset* sebességét kinagyítva vizsgáljuk a 13., 14. és 15. ábrán, akkor azt láthatjuk, hogy ezek közül a *boost* :: *dymamic_bitset* átlagosan gyorsabb. A legnagyobb különbség az elem beszúrásában látható, ahol nyilvánvalóan az *objset* hátrányban van, mert az a



10. ábra. Számhalmazokon végzett műveletek ideje másodpercben az első paraméterbeállítás szerint.

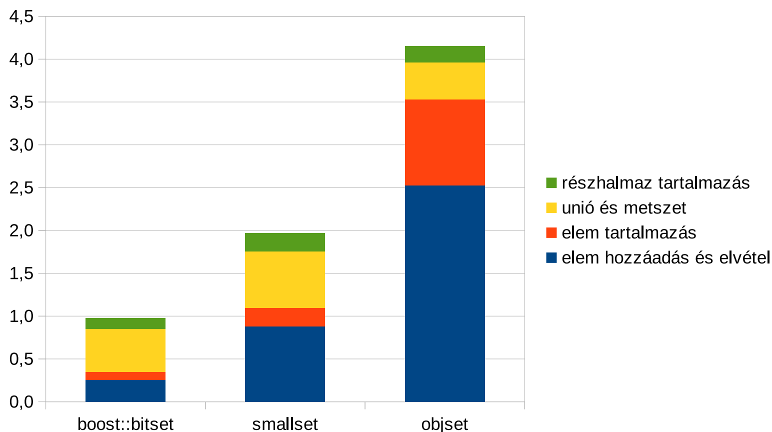


11. ábra. Számhalmazokon végzett műveletek ideje másodpercben a második paraméterbeállítás szerint.

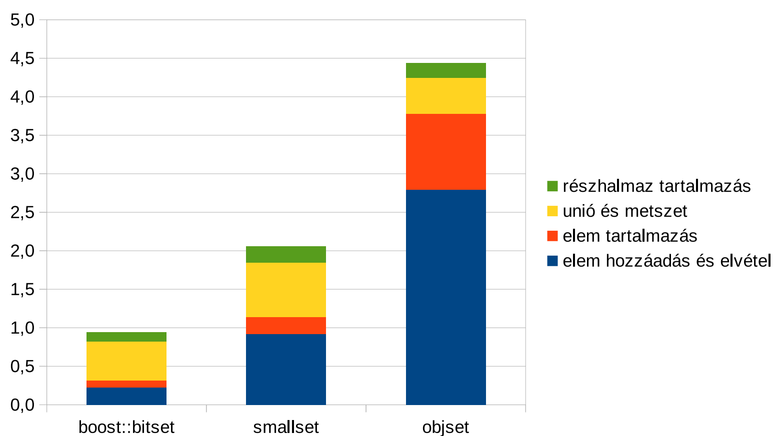


12. ábra. Számhalmazokon végzett műveletek ideje másodpercben a harmadik paraméterbeállítás szerint.

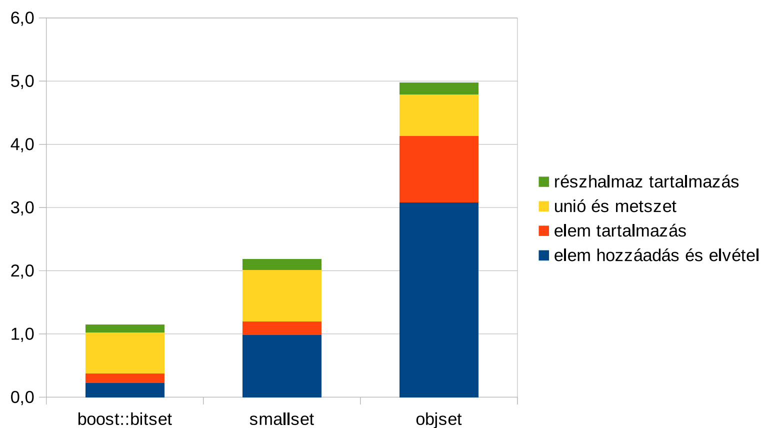
bitvektor mellett még egy dinamikus adatszerkezetet is épít. Ugyanakkor, érdemes megjegyezni, hogy ha az elemek már a halmazban vannak, akkor az unió és metszet műveletek az *objset* eseten a leggyorsabbak, köszönhetően az értelmezési tartomány tömörítésének. Ez a különbség annál szembetűnőbb, minél ritkásabb a halmaz.



13. ábra. Számhalmazokon végzett műveletek ideje másodpercben az első paraméterbeállítás szerint.

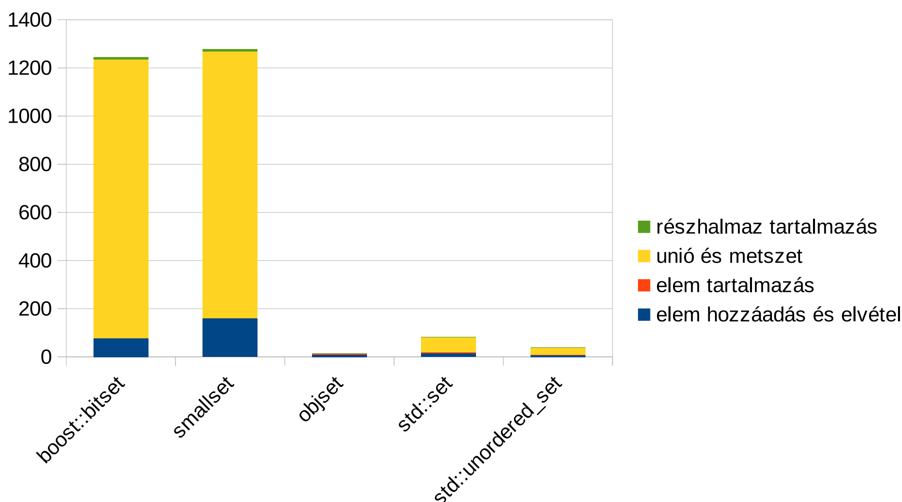


14. ábra. Számhalmazokon végzett műveletek ideje másodpercben a második paraméterbeállítás szerint.



15. ábra. Számhalmazokon végzett műveletek ideje másodpercben a harmadik paraméterbeállítás szerint.

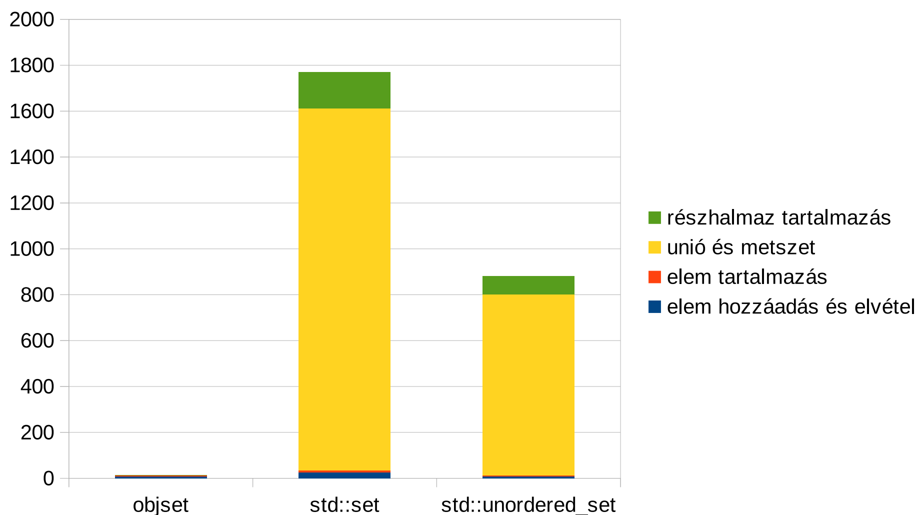
Tekintsük most külön a negyedik paraméterkészletet a 16-os ábrán. Ez az az extrém példa, amikor a tárolandó számok az értelmezési tartományuknak csak az ezredében találhatók. Ilyenkor fordul a kocka, mert kevés elemet tárolunk, de az értelmezési tartomány mérete miatt a lefoglalt bitvektorok hossza óriásira nő a *boost* :: *dynamic_bitset* és a *smallset* esetén. Ebben az esetben a keresőfa és a hasítótábla sebessége jobb, de a legjobb az *objset*, köszönhetően annak, hogy tömöríti az értelmezési tartományt, és ugyanakkor gyors az unió és metszet művelete is, amit bitvektorokon végez.



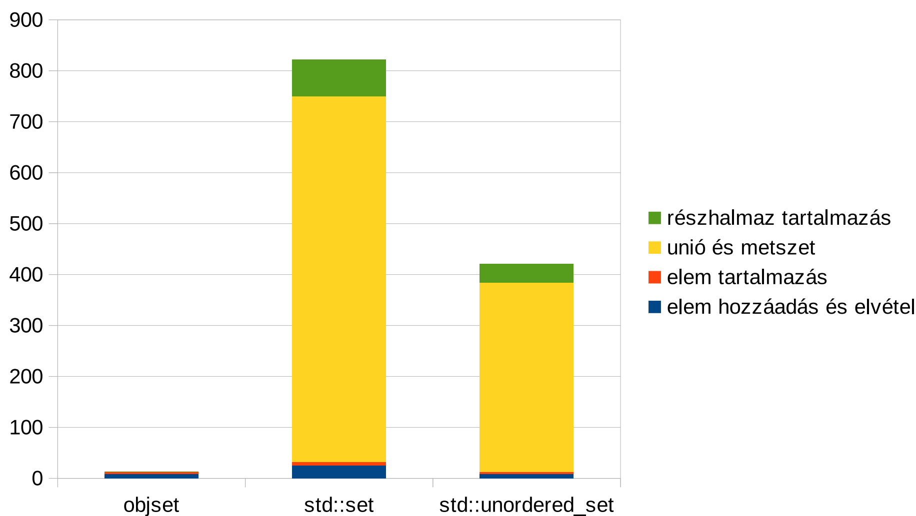
16. ábra. Számhalmazokon végzett műveletek ideje másodpercben a negyedik paraméterbeállítás szerint.

4.3. Futási eredmények univerzális halmazokra

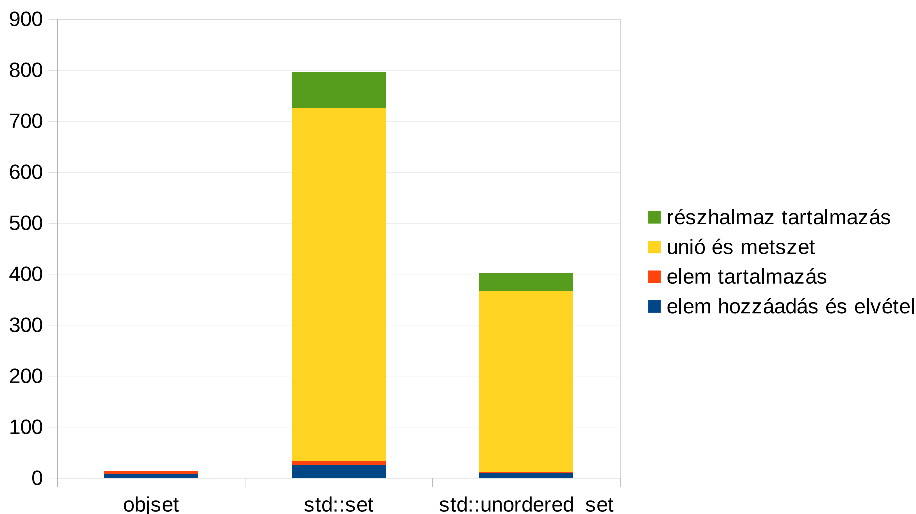
Ha azokat a halmazleírásokat vizsgáljuk, melyek tetszőleges objektum tárolására alkalmasak, akkor egyértelműen az *objset* a leghatékonyabb, a tesztekben körülbelül két nagyságrenddel kisebb időigénnyel, ahogy a 17., 18. és 19. ábrákon a futási eredményeket nézzük. A különbség nagy részét az unió és metszet műveletek adják. Itt is megjegyezzük, hogy az STL tárolók közül az a tároló, a *std* :: *unordered_set* a gyorsabb körülbelül kétszer, amihez a függvénykönyvtárban nem is kapunk unió és metszet műveleteket. Tehát, halmazműveletek implementálására az *std* :: *set* és *std* :: *unordered_set* közül akkor is az utóbbit érdemes használni, ha ahhoz nincsen meg minden művelet a függvénykönyvtárban, hanem egy részét magunknak kell megírnunk.



17. ábra. Objektumhalmazokon végzett műveletek ideje másodpercben az első paraméterbeállítás szerint.



18. ábra. Objektumhalmazokon végzett műveletek ideje másodpercben a második paraméterbeállítás szerint.



19. ábra. Objektumhalmazokon végzett műveletek ideje másodpercben a harmadik paraméterbeállítás szerint.

Amennyiben az egyes halmazműveletek futási idejét súlyozzuk az 1. táblázatban szereplő előfordulási százalékokkal, majd tesztenként normáljuk az *objset* futási idejére, akkor a 2. táblázatban szereplő értékeket kapjuk. Ezek azt mutatják, hogy – a referenciaként használt hálózatszintézis feladatok műveletigénye alapján – hálózatszintézis feladatok esetén az $std :: set$ és $std :: unordered_set$-hez viszonyítva a jelen publikációban javasolt adatszerkezet várhatóan körülbelül 700-szor, illetve 350-szer gyorsabb.

2. táblázat. Halmazműveletek relatív időigénye alternatív megvalósítások esetén

	$objset$	$std :: set$	$std :: unordered_set$
Teszt1	1	1168,3	584,5
Teszt2	1	538,0	277,9
Teszt3	1	447,4	228,5
Átlag	1	717,9	363,7

5. Összefoglalás

A halmazokat megvalósítani képes adatszerkezet implementációk átfogó elemzését és tesztelését mutattam be, azzal a motivációval, hogy hálózatszintézis eljárások megvalósításához a leghatékonyabb halmazimplementációt találjuk meg. A tesztelés során megállapítást nyert, hogy az általam javasolt indexelt halmaz adatszerkezet átlagosan két nagyságrenddel ad hatékonyabb futási eredményt általános objektum halmazon végzett műveletekre, mint a leghatékonyabb ismert függvénykönyvtári megoldások. Ráadásul egyszerű számhalmazok tárolására is átlagosan az általam javasolt módszer a leghatékonyabb, ha a tárolandó számok vagy indexek értelmezési tartománya több nagyságrenddel nagyobb, mint a halmazműveletek során ténylegesen előforduló számok számossága. A javasolt indexelt halmazba új elem beszúrása kicsit lassabb, ellenben ismert elem beszúrása, az unió és metszet műveletek számolása sokkal gyorsabb a szokásos függvénykönyvtári tárolóknál. Ha a futási idők átlagát, a halmazműveleteknek a folyamatszintézis feladatok megoldása során előforduló gyakoriságával súlyozzuk, akkor a sebességkülönbség még nagyobb a javasolt új adatszerkezet javára.

Visszatérve a munkám során felmerült fejlesztésekre, megjegyzem, hogy a [4] publikációban bemutatott folyamatszintézis optimalizáló algoritmus részben azért tud versenyképes lenni nála sokkal komplexebb vegyes-egész matematikai programozási feladat megoldókkal, mert egyáltalán nem kezel egész változókat, hanem helyette a fentiekben bemutatott rendkívül hatékony *objset* halmazokkal megvalósított osztályozásokkal írja le az optimumkeresés minden döntését. A keresés során a részprobléma szétválasztás, a logikai következtetésekre épülő gyorsítások és ellentmondás-vizsgálatok is rendre halmazműveletekkel valósulnak meg, a fentiekben javasolt implementáció segítségével.

Köszönetnyilvánítás

Ajánlom ezt a cikket Steierlein István tanár úrnak, aki befogadott a számítástechnika szakkörére, és először tanított nekem programozást és algoritmizálást a 80-as évek végén, Balatonfüreden a Radnóti Miklós Általános Iskolában, Primo 16 számítógépen.

Hivatkozások

[1] M. Barany, B. Bertok, Z. Kovacs, F. Friedler and L.T. Fan: *Solving vehicle assignment problems by process-network synthesis to minimize cost and environmental impact of transportation*, Clean Technologies and Environmental Policy, Vol. **13**, pp. 637-642 (2011). DOI: 10.1007/s10098-011-0348-2

[2] B. Bertok and A. Bartos: *Renewable energy storage and distribution scheduling for microgrids by exploiting recent developments in process network synthesis*, Journal of Cleaner Production, Vol. **244**, paper: 118520 (2020). DOI: 10.1016/j.jclepro.2019.118520

[3] B. Bertok and A. Bartos: *Algorithmic Process Synthesis and Optimisation for Multiple Time Periods Including Waste Treatment: Latest Developments in P-graph Studio Software*, Chemical Engineering Transactions, Vol. **70**, pp. 97-102 (2018). DOI: 10.3303/CET1870017

[4] A. Bartos and B. Bertok: *Parameter tuning for a cooperative parallel implementation of process-network synthesis algorithms*, Central European Journal of Operations Research, Vol. **27**, pp. 551-572 (2019). DOI: 10.1007/s10100-018-0576-1

[5] F. Friedler, K. Tarjan, Y.W. Huang and L.T. Fan: *Combinatorial Algorithms for Process Synthesis*, Computers & Chemical Engineering, Vol. **16**, pp. S313-S320 (1992). DOI: 10.1016/S0098-1354(09)80037-9

[6] F. Friedler, J.B. Varga and L.T. Fan: *Decision-Mapping: A Tool for Consistent and Complete Decisions in Process Synthesis*, Chemical Engineering Science, Vol. **50**, pp. 1755-1768 (1995). DOI: 10.1016/0009-2509(95)00034-3

[7] F. Friedler, J.B. Varga, E. Feher and L.T. Fan: *Combinatorially Accelerated Branch-and-Bound Method for Solving the MIP Model of Process Network Synthesis*, In State of the Art in Global Optimization, Springer, pp. 609-626 (1996). DOI: 10.1007/978-1-4613-3437-8_35

[8] Z. Kovacs, Z. Ercsey, F. Friedler and L.T. Fan: *Separation-Network Synthesis: Global Optimum through Rigorous Super-Structure*, Computers & Chemical Engineering, Vol. **24**, pp. 1881-1900 (2000). DOI: 10.1016/S0098-1354(00)00568-8

[9] N.L. Biggs, E.K. Lloyd and R.J. Wilson: *Graph Theory*, Oxford University Press, pp. 1736-1936 (1976).

[10] F. Friedler, K. Tarjan, Y.W. Huang and L.T. Fan: *Graph-Theoretic Approach to Process Synthesis: Polynomial Algorithm for Maximal Structure Generation*, Computers & Chemical Engineering, Vol. **17**, pp. 929-942 (1993). DOI: 10.1016/0098-1354(93)80074-W

[11] J. Liu, L.T. Fan, P. Seib, F. Friedler and B. Bertok: *Downstream Process Synthesis for Biochemical Production of Butanol, Ethanol, and Acetone from Grains: Generation of Optimal and Near?Optimal Flowsheets with Conventional Operating Units*, Biotechnology Progress, Vol. **20**, pp. 1518-1527 (2004). DOI: 10.1021/bp049845v

[12] B. Bertok and I. Heckl: *Process Synthesis by the P-Graph Framework Involving Sustainability*, In Sustainability in the Design, Synthesis and Analysis of Chemical Engineering Processes, Butterworth-Heinemann, pp. 203-225 (2016). DOI: 10.1016/B978-0-12-802032-6.00009-8

[13] M. Barany: *Modeling Vehicle Routing Problems as Process-Network Synthesis Problems*, Industrial Applications of the P-Graph Framework, PhD dissertation, University of Pannonia (2015). DOI: 10.18136/PE.2015.595

[14] D.E. Knuth: *The Art of Computer Programming, Fundamental Algorithms*, Addison-Wesley, Vol. **1**, (1968).

[15] D.E. Knuth: *The Art of Computer Programming, Sorting and Searching*, Addison-Wesley, Vol. **3**, (1973).

[16] F. Friedler, K. Tarjan, Y. W. Huang and L.T. Fan: *Graph-Theoretic Approach to Process Synthesis: Axioms and Theorems*, Chemical Engineering Science, Vol. **47**, pp. 1973-1988 (1992). DOI: 10.1016/0009-2509(92)80315-4

[17] Heckl I.: *Az SNS-Multi feladat redukált szuperstruktúrája*, Szétválasztási hálózatok szintézise: Különböző tulajdonságokon alapuló szétválasztó módszerek egyidejű alkalmazása, PhD értekezés, Pannon Egyetem (2007).
https://konyvtar.uni-pannon.hu/doktori/2007/Heckl_Istvan_dissertation.pdf

[18] M. Frits and B. Bertok: *Scheduling Custom Printed Napkin Manufacturing by P-graphs*, Computers & Chemical Engineering, Vol. **141**, 107017 (2020). DOI: 10.1016/j.compchemeng.2020.1070174

[19] E. Konig and B. Bertok: *Process Graph Approach for Two-Stage Decision Making: Transportation Contracts*, Computers & Chemical Engineering, Vol. **121**, pp. 1-11 (2019). DOI: 10.1016/j.compchemeng.2018.07.011

[20] L.T. Fan, B. Bertok and F. Friedler: *A Graph-Theoretic Method to Identify Candidate Mechanisms for Deriving The Rate Law of a Catalytic Reaction*, Computers & Chemistry, Vol. **26**, pp. 265-292 (2002). DOI: 10.1016/S0097-8485(01)00119-X

[21] J.C. Garcia-Ojeda, B. Bertok and F. Friedler: *Planning evacuation routes with the P-graph framework*, Chemical Engineering Transactions, Vol. **29**, pp. 1531-1536 (2012). DOI: 10.3303/CET1229256

[22] M. Barany, B. Bertok, Z. Kovacs, F. Friedler and L.T. Fan, *Solving vehicle assignment problems by process-network synthesis to minimize cost and environmental impact of transportation*, Clean Technologies and Environmental Policy, Vol. **13**, 637-642 (2011). DOI: 10.1007/s10098-011-0348-2

[23] L. Vance, H. Cabezas, I. Heckl, B. Bertok and F. Friedler, *Synthesis of sustainable energy supply chain by the P-graph framework*, Industrial & Engineering Chemistry Research, Vol. **52**, pp. 266-274 (2013). DOI: 10.1021/ie3013264

Bertók Botond 1976-ban született Veszprémben. 1999-ben okleveles mérnök informatikusként végzett a Veszprémi Egyetemen. 2004-ben szerzett PhD fokozatot. Mérai László és Polinszky díjas. OTDT Mestertanár Aranyérmes. Jelenleg a Pannon Egyetem egyetemi docense. Kutatási területe üzleti- és műszaki folyamatok optimalizálása és számítógépes döntéstámogatása. 50 referált nemzetközi folyóirat publikáció és 8 könyvfejezet szerzője. Független hivatkozásainak száma több mint 700. 2000 óta a Magyar Operációkutatási Társaság, 2012-től az MTA Informatikai Tudományos Bizottság tagja.

BERTÓK BOTOND
Pannon Egyetem,
Veszprém, Egyetem utca 10.
bertok@dcs.uni-pannon.hu

HYBRID DATA STRUCTURE FOR EFFICIENT IMPLEMENTATION OF SET OPERATIONS

Botond Bertok

The description of combinatorial or mixed integer optimization algorithms often includes sets and operations on them. Many of the combinatorial algorithms have a theoretically high computational complexity, so the size of the practically solvable problems highly depends on the quality of the implementation. Software implementation of set operations is not trivial. As I present in the article, there may be magnitude differences between the speeds of set operations, depending on what data structure the set actually covers.

As a new result, I propose a hash table – bit set hybrid data structure for software implementation of object sets that can run up to multiple orders of magnitude faster than the C ++ library data structures included in the latest compilers.

Keywords: set operations, C++, combinatorial algorithms.

Mathematics Subject Classification (2000): 03C62 Models of arithmetic and set theory; 68P05 Data structures.