TDK-dolgozat

Kolozsi István Zoltán

Új, FFD-alapú algoritmusok tapasztalati elemzése többdimenziós esetekre

Kolozsi István Zoltán

Programtervező informatikus Msc

3. évfolyam

dr. Balogh János

Egyetemi docens

Szegedi Tudományegyetem

Természettudományi és Informatikai Kar

Számítógépes Optimalizálási Tanszék

Tartalomjegyzék

Absztrakt

A modern számítástechnikában arra törekednek, hogy minden feladat minél gyorsabban és hatékonyabban készüljön el. A felhasználók türelmetlenek, minden kérésükre azonnali választ várnak. Ezért a szolgáltatásokat biztosító nagy vállalatok számára a hatékony erőforrás-felhasználás elengedhetetlen a versenyben maradásuk szempontjából. Példának okáért vegyünk egy céget, amely Virtuális Gépek erőforrásait kínálja a felhasználóik számára kibérlésre, ahol a felhasználók által kapott feladatok kiosztása egy alapvető és hihetetlenül fontos kérdés. Ez a feladat megfeleltethető a vektorpakolási (*Vector Bin-Packing,* **VBP$_d$**) problémának.

A vektorpakolási feladat egyfajta többdimenziós általánosítása a jól ismert ládapakolási problémának, amely egy klasszikus optimalizálási feladat, amiről tudjuk, hogy már az egydimenziós változata is NP-nehéz. Ezért számos közelítő algoritmust, heurisztikát fejlesztettek ki rá.

A tanulmányomban megvizsgálom a már jól ismert és hatékonynak bizonyuló *First Fit Decreasing* (FFD) algoritmus variánsokat, és a szakirodalom szerint különösen hatékonynak bizonyuló *Geometric Heuristics* (Dot-Product és L2) algoritmusokat és az általam készített új, FFD-n alapuló, de egyedi szabályrendszerrel működő algoritmusokat (*FFDRev, FFDRevAdv, FFDVal, FFDRatio, FFDGroups, FFDBox, FFDBG*). Az összes algoritmust az általam készített különféle bemeneti (benchmark) példákon tesztelem 1, 2, 3, 4 és 6 dimenzióban.

Azt tapasztaltam, hogy a GH-típusú algoritmusok valóban jobban teljesítenek a szakirodalomban korábban vizsgált FFD-variánsoknál. A GH algoritmusok működéseinek alapja, hogy minden egyes lépésben újraszámolják a súlyokat, ami alapján az elpakolandó tárgyat kiválasztják, de ez költségesnek bizonyult nagyobb tárgyszám esetén. Ezért olyan új algoritmusokat definiáltam, amik FFD-n alapulnak, de képesek legyenek felvenni a versenyt a GH algoritmusokkal, azáltal, hogy az FFD szabály szerinti eredeti sorrenden változtatnak valamilyen szabályrendszer alapján. Ez egy kihívást jelentő feladatnak bizonyult, de az FFDBox, FFDGroups és FFDBG nem determinisztikus módon működő algoritmusok ígéretes eredményeket értek el. A kapott futási eredményeket táblázatokon mutatom be. A háromdimenziós bemeneti példákra a vizsgálataim során azt tapasztaltam, hogy az esetek 67%-ban megjavítjuk a szakirodalmakban ismertetett FFD-variánsok eredményeit és az esetek 26%-ban a GH algoritmusokét is. Hasonló jó eredményeket hoztak a hatdimenziós vizsgálataim is. Az összes vizsgált dimenzióban, mindössze az esetek 6%-ában maradnak alul az új algoritmusok az FFD és GH algoritmusok eredményeihez képest, és legrosszabb esetben is maximum 4%-kal használnak több ládát az addig ismert legjobbnál.

1. Bevezetés

Dolgozatomban bemutatom a vektorpakolási feladat fontosságát, nehézségeit és lehetséges felhasználási módjait. Tapasztalati úton fogom elemezni az „*Heuristics for Vector Bin Packing*” című [1] cikkben szereplő és az általam kifejlesztett új, FFD-alapú algoritmusokat. Az elemzéshez definiáltam és generáltam különféle bemeneti adatokat is, amelyeken a teszteléseket elvégeztem. Ezek részben az irodalomban előírt módon generált (benchmark) példákból és az FFD-jellegű algoritmusokra ellenpéldaként szolgáló feladatokból állnak.

Egy rövid áttekintést ismertetek néhány, a szakirodalomban bemutatott algoritmusokat tárgyaló cikkről. A következő részben bemutatok néhányat a virtuális gépek olyan szolgáltatásai közül, melyekhez a vektorpakolási probléma hatékony megoldása segítséget nyújthat. Ezt követően elméleti szinten ismertetem a VBP_d problémát, az FFD algoritmusok családját, a Geometrikus Heurisztikákat és új, általam definiált FFD-alapú algoritmusokat.

Az elméleti áttekintést követően betekintést nyújtok az implementált programok szerkezetébe, részletezem az algoritmusok megvalósításait, és definiálom az általam készített benchmark példákat. A dolgozat záró része a futási eredmények ismertetése, amelyben táblázatokon és diagramokon hasonlítom össze a tárgyalt algoritmusokat és elemzem átlagos, tapasztalati viselkedésüket.

1.1. Szakirodalmi áttekintés

Panigrahy és szerzőtársai [1] cikke (2011) képezi a tanulmányom fő inspirációját és alapját. Bemutatták a Geometrikus Heurisztikák (Dot-Product és L2 algoritmusok) és a Grasp[K] használatát. Tapasztalati elemzéseket végeztek különböző dimenziószámú feladatokon és összehasonlították az általuk készített algoritmusok eredményeit az ismert FFD-variánsokkal. Összességében arra a konklúzióra jutottak a szerzők, hogy olyan heurisztikákat sikerült megalkotniuk, amelyek a gyakorlatban is jól alkalmazhatóak, pl. a Microsoft's Virtual Machine Manager [2] a Dot-Product és a Norm-based Greedy heurisztikákat használja. Ezeket, a fent említett módszerekkel együtt lentebb részletesen tárgyalom majd.

Filipe Brandao és Joao Pedro Pedroso cikkükben [3] ismertetnek egy hatékony, de számításigényes megoldást a **VBP_d**-re és a „*cutting stock*” problémára. Felírtak az adott problémára egy megoldási gráfot, majd ezen a gráfon iteratívan futtatják az általuk definiált gráfredukciós módszert, amely egy fajta él-folyam problémának is tekinthető. Viszont a módszerük csak egészszám-vektorokra alkalmazható, és egzakt megoldást ad meg, így

nagyméretű feladatok megoldására az nem alkalmas. (A feladat NP-nehéz, mint később tárgyaljuk, így jelen tudásunkkal ez nem is várható egzakt megoldó módszerektől.)

A következő tanulmány azért érdemes említésre, mert részletesen kitér arra, hogy mennyire fontos feladat jelenleg a Virtuális Gépek ütemezési problémája [4]. A cikket Saikishor Jangiti és Shankar Sriram jegyzik, dolgozatukban kihangsúlyozzák, hogy a cégeknek gazdaságilag is fontos, hogy minél hatékonyabban használják fel a rendelkezésre álló forrásaikat.

A „*The Three-Dimensional Bin Packing Problem*” című [5] dolgozat olyan háromdimenziós példákat tartalmaz, amiket a következőkben ismertetni és használni is fogok kisebb változtatással. Mivel én vektorpakolási problémákkal foglalkozom, így az ebben a cikkben szereplő bemeneti példák nem mindegyike hasznosítható a számomra. Az alapvető eltérés, hogy ők megengedik a dimenziók cseréjét (a tárgyakat lehet forgatni), amíg ez a vektoros esetben tiltott.

Lars Nagel és szerzőtársai nemrég megjelent, 2023-as publikációjukban [6] a vektorütemezési és a vektorpakolási problémákat vizsgálták részletesebben. Ők is megemlítik konklúzióként, hogy az egzakt algoritmusok és a közelítő sémák a várakozásoknak megfelelően nem alkalmasak gyakorlati célokra a hosszú futási idejük miatt. (Ez is illusztrálja az általam kutatott és definiált algoritmusok jelentőségét.) Több új algoritmust definiáltak és vizsgáltak, zömében a vektorütemezési feladatra koncentrálva. A vektorpakolási feladat kapcsán kevesebb algoritmust vizsgáltak, legfőképpen egy nagy futási idejű CNS (consistent neighborhood search) algoritmust, és egy LS lokális algoritmust, ami más algoritmusok outputján végez utófeldolgozást lokális keresés segítségével (ami extra időt vesz igénybe), és még foglalkoztak a láda- és tárgyközpontú szemléletek vegyítésével. Kísérleteik és eredményeik arra a következtetésre adnak okot, hogy általános kompromisszumot kell kötni a futási idő és a pakolás minősége között.

A fentebb említett tanulmányok közül számos utalt és használta fel Alberto Caprara és Paolo Toth munkáját [7] és az általuk definiált bemeneti osztályokat a teszteléshez inputként. Ezért én is ihletet merítettem belőle és reprodukáltam az őáltaluk bemutatott bemeneti példák osztályait is, más tesztpéldákkal együtt. Cikkükben a 2-dimenziós vektorpakolási feladatra koncentráltak, és alapvetően kétféle megközelítést jártak körül. Az elsőnek az alapja többféle alsó korlát kiszámítása, annak ellenére, hogy ezek gyorsan számíthatóak, sajnos alkalmazásuk nem ad elég hatékony eredményeket. A másik megközelítésben felírnak egy egész értékű programozási formulát nagy változó számmal az adott problémára. A relaxált lineáris

programozási feladat megoldására az oszlopgenerálás módszerét alkalmazzák, de ez nagyon sok időt vesz igénybe, és kis méretű feladatokra alkalmazható, amint az egzakt algoritmusok kapcsán megemlítettük, és ez is azok közé tartozik.

A Geometrikus Heurisztikák [1] eredményeinek javításával a [8] cikkben is foglalkoztak. Ők olyan iteratív algoritmusokat próbáltak futtatni a GH futási eredményein, amelyek megpróbálják átrendezni a már elpakolt tárgyakat, ezáltal egy új jobb érvényes pakolást létrehozni. Ebből én is ihletet merítettem, hasonló kiinduló szándékkal vizsgálom majd az FFD-alapú vektorpakolási heurisztikákat.

1.2. Motiváció a feladatra: Virtuális Gépek (VM) rövid bemutatása

Az Infrastruktúra, mint Szolgáltatás (Infrastructure as a Service, **IaaS**), a felhőalapú szolgáltatásnyújtási modell igény szerinti hozzáférést biztosít a különböző felhőalapú adatközpontokban (Cloud Data Center, **CDC**) rendelkezésre álló távoli számítástechnikai erőforrásokhoz. A Virtuális Gépek (Virtual Machine, **VM**) a Felhő-alapú technológiák alapjai, amelyek felváltják a hagyományos számítástechnikai infrastruktúrákat. A felhasználók a saját alkalmazásaik futtattatásának érdekében kibérelnek **VM**-eket az ismert Felhőszolgáltatóktól (Cloud Service Provider, **CSP**), olyanoktól, mint például az Amazon EC2, Google Compute Engine, vagy a Microsoft Azure, csak néhány népszerűt megemlítve [4].

Az **Amazon Elastic Compute Cloud** (Amazon EC2**)** [9] egy rugalmas és skálázható felhőalapú számítási szolgáltatás, amely lehetővé teszi a felhasználók számára virtuális szerverek létrehozását és kezelését az **Amazon Web Services**-en. Az Amazon EC2 rendszerén keresztül létrehozhatja a felhasználó a számára szükséges hálózattokat, virtuális szervereket vagy biztonsági beállításokat. Ha valós idejű adatokra van szükségünk az Amazon EC2-vel kapcsolatban, akkor ezek általában olyan adatokat jelentenek, amelyek az aktuális állapotot, teljesítményt vagy használatot mutatják az EC2 példányokkal kapcsolatban.

A **Microsoft Azure** [10] egy felhőalapú számítási platform és több, mint 200 szolgáltatás gyűjteménye. Az Azure lehetővé teszi a fejlesztőknek és az IT-szakembereknek alkalmazások futtatását, tesztelését és kezelését a Microsoft globális adatközpontjainak hálózatában.

A **Google Compute Engine (GCE)** [11] egy felhőalapú infrastruktúra-szolgáltatás, amely annyi VM-et biztosít a felhasználó számra, amennyire szüksége van. A Google Compute Engine lehetővé teszi VM-ek létrehozását és futtatását a Google adatközpontjaiban. Ezek a virtuális gépek lehetnek Windows vagy Linux alapúak, és különböző számítási erőforrásokkal rendelkeznek, például *CPU, RAM* és *tárhely*.

Az energiahatékony irányítás a CDC-k aktív kutatási területe, amelynek célja a működési költségek csökkentése. A CDC energiafogyasztásának jelentős részét a fizikai gépek (**PM**-ek) teszik ki, és a PM-ek még üresjáratban is a teljes terheléshez képesti fogyasztás 60%-át folyamatosan használják [4]. Ha csökken az energiafogyasztás, az hozzájárul a szén-dioxid-kibocsátás és annak káros környezeti hatásainak minimalizálásához. Az autonóm felhőkezelők a VM-kérelmeket megpróbálják minél kevesebb PM-hez rendelni a működési költségek minimalizálása érdekében, ami tekinthető *Vector Bin-Packing* (**VBP_d**) problémának. A *d* itt az alsó indexben arra utal, hogy *d*-dimenziós, azaz ennek során *d* számú különböző erőforrást is figyelembe vehetünk.

Tehát a **VM elhelyezési probléma**, és ez által a VBP_d nem csak matematikai és informatikai problémának tekinthető, hanem gazdasági és környezetvédelmi problémának is. Ráadásul a vektorok ütemezése és a vektorok ládába pakolása alapvető problémája az operációkutatásnak [6]. Az előbb említett problémák fontossága vitathatatlan, épp ezért kutatják mai napig is a VBP_d problémát.

2. $\mathbf{VBP_d}$

A d-dimenziós Vektorpakolási Probléma (**$\mathbf{VBP_d}$**) esetén adott egy I halmaz, amely n elemből áll, ezek $I^1, I^2, \ldots, I^n$, ahol mindegyik $I^l \epsilon\ \mathbb{R}^d$. Egy érvényes pakolásnak nevezzük azt, amikor az I-t felosztjuk k halmazra (vagy ládára), amelyek $B_1, \ldots, B_k$ lesznek, ahol minden $1 \leq j \leq k$ láda és minden $1 \leq i \leq d$ dimenzió esetén teljesül az alábbi feltétel:

$$\sum_{I^l \epsilon B_j} I_i^l \leq 1,$$

azaz az egy ládabeli elemek (vektorok) összege egyetlen dimenzióban sem haladhatja meg a láda kapacitását (ami itt 1).

A VBP_d probléma célja, hogy találjunk egy érvényes pakolást, miközben minimalizáljuk a felhasznált ládák k számát. A VBP_d a *Bin-Packing Problem* (**BPP**) egy olyan általánosítása, ahol a tárgyak és a ládák d-dimenziós vektorok.

Megjegyezzük, hogy a VBP_d NP-nehéz minden d-re (az erős értelemben) [12], tehát nem ismert rá (nagy méretű feladatokra is) hatékony egzakt algoritmus. Egydimenziós esetben létezik aszimptotikus PTAS (polinomiális idejű approximációs séma) [13], viszont a túlságosan nagy futási ideje miatt ez a gyakorlatban teljesen használhatatlan. Viszont, ha $d \geq 2$, akkor a probléma már APX-nehéz [1], nem adható rá aszimptotikus PTAS, ha $P \neq NP$ [14]. Ha lenne, (akár csak $d = 2$-re), az bizonyítaná, hogy P=NP.

Ezáltal ez a feladat jól reprezentálja a *Virtuális Gépek* (VM) *elhelyezési problémáját*, több erőforrás-kapacitás megszorítással. A gyakorlatban ez a probléma modellezi például a *statikus erőforrás kiosztást (allokációt)*, ahol minimális mennyiségű, ismert kapacitású szerverrel kell kielégíteni a szolgáltatások igényeit.

3. Az FFD algoritmusok családja

A szakirodalmakban ismertetett megoldási megközelítései a VBP_d problémának a *First Fit Decreasing* (**FFD**) algoritmus család tagjai. Ezek a klasszikus egydimenziós FFD algoritmus [15] általánosításai. Az FFD algoritmus az egydimenziós feladat esetén először nem növekvő sorrendbe rendezi a tárgyakat a méretük alapján, majd a *First Fit* (**FF**) algoritmust használja, amely minden elemet az első (legkorábban létrehozott), olyan ládába pakolja, ahová befér az adott elem, azaz nem haladja meg a láda maximális kapacitását a berakása után sem. Ha nincs ilyen láda, akkor nyit egy újat és abba helyezi el a tárgyat. Az FFD algoritmus rendkívül népszerű a gyakorlatban [15]. Dósa György és szerzőtársai [16] megmutatták, hogy az FFD egydimenziós esetben egy olyan pakolást talál, amely legfeljebb $\mathrm{FFD(I)} \leq \frac{11}{9}\ \mathrm{OPT(I)} + \frac{6}{9}$ számú ládát használ minden I inputra, ahol $\mathrm{FFD(I)}$ az FFD algoritmus által felhasznált ládák száma és $\mathrm{OPT(I)}$ az optimális megoldás ládaszáma. Tehát, ha $d = 1$, akkor ez a megközelítés nagyon hatékonynak bizonyul, mind formálisan, mind pedig a gyakorlatban is. A [17]-es cikkben megtalálható az is, hogy a többdimenziós feladatra az (online algoritmusnak is tekinthető, azaz az elemek rendezése nélküli) FF algoritmus legrosszabb eset aszimptotikus versenyképességi hányadosa függ a dimenziószámtól, mégpedig $d + 0{,}7$.

A többdimenziós vektorpakolási feladatnál az alkalmazott FFD algoritmusban a pakolásra szolgáló FF algoritmus a fentihez hasonlóan definiálható. Viszont az első fázis, azaz az elemek rendezése többféleképpen definiálható. Ezt az alapján hajtjuk végre, hogy a heurisztikákban milyen súlyfüggvényt alkalmazunk a tárgyakon. Az alkalmazott különböző súlyfüggvények alapján egy-egy eltérő algoritmus-variáns azonosítható, például a következőek gyakran használtak [1,7]:

$$w(I) = \prod_{i \leq d} I_i \qquad \text{(FFDProd)},$$

$$w(I) = \sum_{i \leq d} I_i \qquad \text{(FFDSum)},$$

$$w(I) = \frac{\sum_{i \leq d} I_i}{d} \qquad \text{(FFDAvg)}.$$

Megjegyezzük, hogy ez valójában csak két variánst eredményez, hiszen az FFDSum és FFDAvg esetén az elemek súlyai mindig egy d hányadosban térnek el egymástól, így ugyanazt a sorrendbe rendezést eredményezik, ha nincs „döntetlen” az elemek súlya között, azaz nincsenek azonos súlyú elemek.

Az előkészítő lépésben a tárgyakat a legnagyobbtól a legkisebbig rendezzük az adott súlyfüggvény alapján, de a döntetlen tetszőlegesen feloldható [1]. A következőkben a döntetlent, vagy holtversenyt minden esetben úgy oldom fel, hogy mindig az a tárgy kerül előrébb a rendezésben, amelyik hamarabb szerepel az adott bemeneti példában, azaz a kisebb indexű elem kerül hamarabb sorra az elpakolásban.

Az FFD-nek nincs egyértelmű általánosítása több dimenziós esetekre, hiszen a többdimenziós méretek alapján egydimenziós súlyok definiálása és a kettő közötti kapcsolat felállítása nem egyértelmű, pontosabban többféle módon is megadható. Először is el kell dönteni, hogyan rendeljük hozzá a súlyokat a d-dimenziós vektorokhoz, amely alapján a csökkenő (nem növekvő) rendezést elvégezzük. Ennek több lehetséges módja ismeretes a fent említetteken kívül is, például ahol egyes dimenziókat fontosságuk miatt egy nagyobb szorzóval vesszük figyelembe, lásd pl. [1].

3.1. Az FFD futási ideje

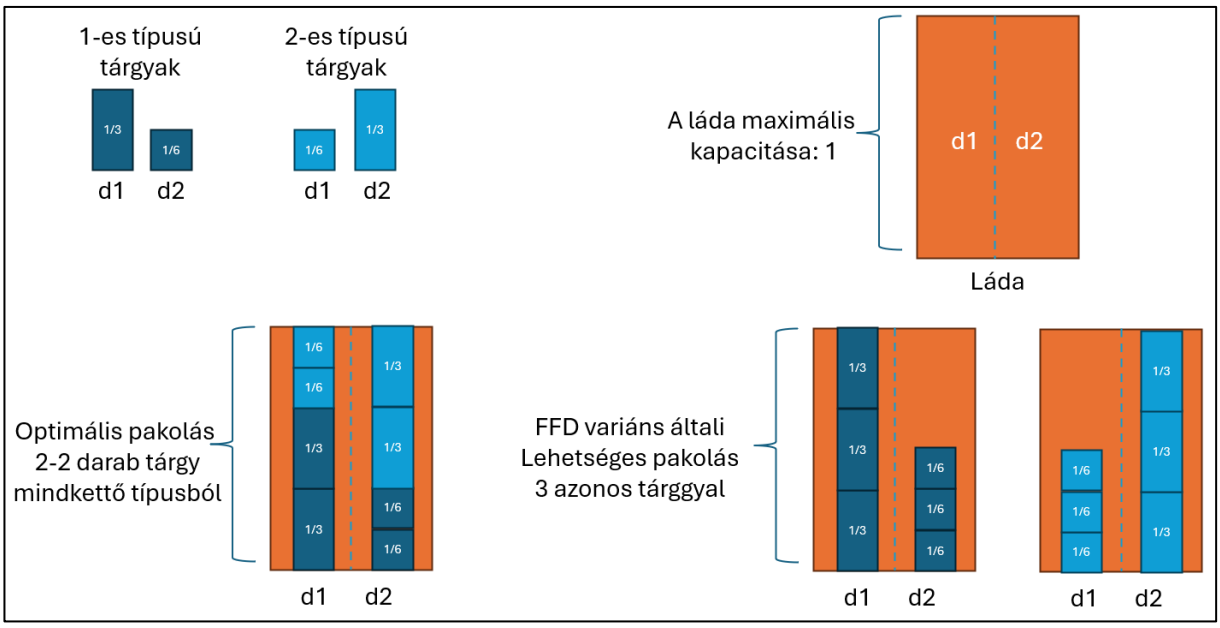
Az egydimenziós FFD algoritmus könnyen implementálható $O(n^2)$ idő alatt, de készíthető $O(n \log n)$ [12] futás idejű implementációja is ennek. A többdimenziós esetben a futási idő az algoritmus két részéből adódik össze. Egyrészt a tárgyak rendezéséből, ami $O(nd + n\,log\,n)$ időt vesz igénybe. Másrészt a tárgyak elpakolásából. Egy olyan algoritmus, amelyik végigellenőrzi az összes ládát, amikor egy elemet el szeretne helyezni, annak a futási ideje $\Omega(n \log n + nk)$, ahol k a ládák száma a megoldásban.

3.2. Ellenpéldák az FFD-re

Az algoritmusok elemzése során (a fent említett módon) feltételezzük azt, hogy ha kettő vagy több tárgy azonos súllyal rendelkezik, akkor mindig a kisebb indexű tárgyat pakoljuk el először. Ennek a következőkben tárgyalt ellenpéldákban lesz nagy szerepe, mivel az azonos súlyú, viszont azonos dimenziókban különböző igényű (méretű) tárgyakat használ fel.

1. példa. Gondoljunk arra a kétdimenziós példára, ahol 2n számú elemünk van, az elemek felének, azaz n elemnek a mérete $(\frac{1}{3}, \frac{1}{6})$, és a másik felének $(\frac{1}{6}, \frac{1}{3})$. Ebben az esetben, az optimális megoldás 4 tárgyat rak ládánként, mindkét fajtából kettőt, míg bármelyik FFD változat három elemet rak ládánként, tehát az optimum 3/2-szeresének megfelelő ládaszámot produkál.

A fentiekben leírtakat az 1. ábrán mutatom be részletesen. Látható a két típusú tárgy, amiket az előbb definiáltam. Illetve bemutatom az optimális megoldást, és azt, hogy az FFD variánsok hogyan fogják pakolni a tárgyakat.



1. ábra. *Két-dimenziós ellenpélda az FFD-re.*

2. példa. Három-dimenziós esetben háromféle különböző tárgyat hozzunk létre, amelyek erőforrás-igényei rendre: $(\frac{1}{3}, \frac{1}{3}-\varepsilon, \frac{1}{3}+\varepsilon)$, $(\frac{1}{3}-\varepsilon, \frac{1}{3}+\varepsilon, \frac{1}{3})$ és $(\frac{1}{3}+\varepsilon, \frac{1}{3}, \frac{1}{3}-\varepsilon)$. Érkezzen mindhárom típusból egyaránt $3n$ számú elem, ahol n tetszőleges pozitív egész, és ε kis rögzített pozitív érték. Erre az inputra az optimális megoldás az lesz, hogyha ebből a háromféle típusú tárgyból egyet-egyet-egyet rakunk egy ládába. Viszont az említett FFD-variánsok erre nem lesznek képesek, mert az azonos típusúakat fogják egy-egy ládába kettesével pakolni. (Erre szintén az optimum $3/2$-szeresét produkálja.)

A fentebb említett ellenpéldáknál rosszabb ellenpéldákat is kaphatunk, amint az [1]-es cikk 5. oldalán található, következő tétel állítja.

Tétel [1]. *Bármilyen egész k-ra és elég nagy n-re van olyan ellenpélda, amelyre az FFD az n tárgyat* $\frac{n}{k}$ *ládába pakolja el, amíg van olyan felosztás, amivel a tárgyakat* $\frac{n}{(k-1)d}$ *ládába elpakolhatjuk. Tehát az approximációs hányadosa az FFD-nek legalább* $\left(1-\frac{1}{k}\right)d$.

A bizonyítás alapötlete érdekes és fontos, mivel hasonló ötletekre épülő inputokat, tesztpéldákat fogok használni. Az [1]-beli bizonyítás azon alapul, hogy megadható egy olyan input, ahol feltételezzük, hogy a különböző T_i tárgytípusok száma d (azaz megegyezik a dimenziószámmal). Minden T_i $(1 \leq \mathrm{i} \leq \mathrm{d})$ típusú I tárgy azonos, és legyen a közös méretük

$$I_j = \begin{cases} \frac{1}{k}, & \text{ha } j = i, \\ \frac{1}{(d-1)(k-1)k}, & \text{ha } j \neq i. \end{cases}$$

Illetve azt feltételezzük, hogy az input pontosan n/d számú tárgyat tartalmaz mindegyik típusból (és azt is, hogy n osztható d-vel). Ekkor azt vehetjük észre, hogy ha egy algoritmus pontosan $k-1$ számú tárgyat pakol mindegyik ládába mindegyik típusból, akkor a töltöttség mindegyik i dimenzióban $\frac{k-1}{k} + \frac{(d-1)(k-1)}{(d-1)(k-1)k} = 1$ (hiszen $k-1$ számú elemet tartalmaz az i-edik, T_i típusból, és $k-1$ számú elemet is egyenként a többi $d-1$ számú típus mindegyikéből), tehát ebből következően ez egy optimális pakolás OPT ládaszámára és az $\text{OPT} \leq n/d(k-1)$ teljesül. Továbbá a fentebb említett FFD algoritmus variánsok függetlenek attól, hogy pontosan hogyan voltak a súlyok kiszámolva, hiszen ugyanazon típusú tárgyakhoz egy ilyen algoritmus ugyanazokat a súlyokat rendeli hozzá és így az összes ugyanolyan típusú tárgyat egymás után rakja az előkészítő részben. Viszont, amikor k számú azonos (T_i) típusú tárgyat elhelyez ugyanabba a ládába, akkor az i-edik dimenzió kapacitása megtelik vagyis eléri a kapacitás konstansát és több tárgyat az inputból nem tud elhelyezni az adott ládádba (mivel nincs olyan tárgyunk, amelynek i-edik dimenziója zérus lenne). Levonható az a következtetés, hogy az FFD legalább n/k ládába pakolná el a tárgyakat. Vagyis az FFD-alapú heurisztikák ezekben az esetekben messze lehetnek az optimumtól.

Az ehhez hasonló példák valószínűleg a virtuális gépek elhelyezésénél is előfordulhatnak. Ez volt a munkám fő motivációja. A fenti ellenpéldák azt sugallják, hogy ha sok, közel azonos súlyú, de mégis az összes dimenziót tekintve megkülönböztethető méretű elemek érkeznek, akkor ezek elronthatják az FFD-alapú algoritmusok viselkedését. Például a tudományos számítások során esetenként magas lehet a CPU igény, de alacsony az I/O igény, miközben a webszerverek esetében ez éppen pont a fordítottja lehet. Hasonlóan, ha a dimenziók időegységeket jelölnek, amikor egy munkaegység magas igényekkel rendelkezik napközben, valószínűleg éjszaka alacsony igényei lesznek.

4. A vizsgált algoritmusok elméleti ismertetése

A következőkben néhány, a szakirodalomban korábban definiált algoritmust adok meg [1,7]. Ezek között vannak nem FFD-alapúak is. Először az FFD két, technikai megközelítésben különböző változatát ismertetem, majd az [1]-es cikkben ismertetett Geometrikus Heurisztikákat és végül az általam definiált új, FFD-alapú algoritmusokat.

4.1. Az FFD Elem- és Ládacentrikus megközelítése

Az FFD megvalósításának **Elemközpontú** szemlélete szerint, vesszük a rendezés szerinti első elemet és azt elrakjuk az első ládába, majd minden elemet az első olyan ládába, amibe belefér, és új ládát nyitunk, ha nincs ilyen. Ezt az eljárást ismételjük a hátralévő tárgyakra, amíg el nem fogy az input összes eleme. Míg a **Ládaközpontú** szemlélet alapján, az FFD-nek csak egy nyitott ládája van egyszerre és minden lépésben belehelyezi a legnagyobb még pakolatlan tárgyat, ami még belefér a jelenlegi nyitott ládába. Ha nincs ilyen tárgy, akkor a láda lezártnak tekinthető, és az algoritmus nyit egy új, üres ládát, és ismétli a fentieket az el nem pakolt tárgyakra.

Több-dimenzióban a tárgyak nem növekvő sorrendjének meghatározása, illetve a következő pakolandó elem kiválasztása vezet el bennünket az FFD stratégia módosításához. E gondolatmenet alapján juthatunk el az FFD olyan módosításaihoz, amelyek versenyképesen működhetnek a szakirodalom szerinti legjobb és a gyakorlatban is alkalmazott módszerekkel szemben. A következőkben először ezen algoritmusok alapelveit ismertetem, majd az általam tervezett, heurisztikus stratégiákét, mely utóbbiakat az FFD módosításával alkottam meg.

4.2. Geometrikus Heurisztikák

4.2.1. Dot-Product (röviden: DotP)

Ez a heurisztika úgy definiálja a „legnagyobb” tárgyat (azaz a következő, pakolásra kiválasztott tárgyat), hogy maximalizálja az úgynevezett pontonkénti szorzatot a szabad kapacitások vektora és a tárgyak igényvektora (mérete) között. Formálisan, ha egy t időpillanatban $\mathrm{r(t)}$ jelöli a jelenleg nyitott ládák maradék kapacitás vektorát, amit úgy kapunk, hogy kivonjuk a tárolók, ládák kapacitásából a már elhelyezett tárgyak igényeinek összegét. Azt az I^l tárgyat tegyük bele abba a ládába, ami maximalizálja a $\sum_i I_i^l r(t)_i$ pontonkénti szorzatot a fennmaradó kapacitások vektorának felhasználásával úgy, hogy nem sértjük meg eközben a kapacitás-előírást. (Azaz nyilvánvalóan ennek során csak olyan ládákat veszünk figyelembe a lehetséges pakolására, amelyekbe befér az adott elem.) Ezáltal mindig egy olyan elem-láda párost fogunk választani pakolásra, amely a lehetőségek közül a legjobban illeszkedő párt alkotják.

4.2.2. Norm-based Greedy (L2)

Ez a heurisztika a különbséget veszi az I^l vektor és az *r(t)* maradék kapacitások között egy bizonyos norma szerint, a pontszorzat helyett. Ez a szakirodalomban *L2*-nek nevezett algoritmus az összes el nem helyezett I^l tárgyat elhelyezi úgy, hogy minimalizálja a $\sum_i \left(I_i^l - r(t)_i\right)^2$ mennyiséget, és a hozzárendelés természetesen itt sem sértheti meg a kapacitásokra vonatkozó előírást. Megemlítjük, hogy az L2 algoritmushoz hasonlóképpen, a $|\cdot|_1$ és a $|\cdot|_\infty$ normák használata adja meg az *L1*-nek és *LInf*-nek nevezett algoritmusokat. Az [1,6] tanulmányok vizsgálatai szerint ezek nem működnek jobban, mint az L2, ahogy az általuk vizsgált példákon keresztül ezt megmutatták.

4.2.3. Grasp[k]

A fenti geometrikus heurisztikák *Grasp[k]*-val való kibővítését is leírták az [1]-es cikkben. A Grasp[k] egy nagyon egyszerű koncepció arra a kérdésre, hogy biztosan mindig az-e a legjobb választás, ha az éppen optimálisnak tűnő lépés mellett dönt az algoritmus (a súlyszámítások alapján legjobbnak ítélt tárgy elhelyezése). Vagyis ez a módszer nem a választható legjobb, hanem a *k-adik* legjobb megoldást választja. Ha a *k* értékét 1-nek választjuk az megegyezik azzal, mintha az alap algoritmus futna. Azonban, ha 1-nél nagyobb a *k* értéke, akkor az előbb leírtak teljesülnek, viszont a tapasztalataim alapján arra figyelni kell, hogy ha a *k* értékét túl nagyra választjuk, akkor a kívánt javulás helyett már romlást lehet tapasztalni.

4.3. Új, FFD alapú algoritmusok

Eddig ismertettem az FFD algoritmusok közül a klasszikusnak mondhatókat, és a Geometrikus Heurisztikákat (továbbiakban: **GH**), amelyekkel a [1] cikk foglalkozott. Viszont szerettem volna az FFD-k és a GH-k eredményein javítani, így megpróbálkoztam, olyan FFD alapú algoritmusok létrehozásával, amelyek kicsit módosítanak, „csavarnak” a pakolási sorrenden.

A választás indoklása és célja. Úgy gondolom, hogy a GH-k azért hoznak jobb tapasztalati eredményeket, mert okosabban választják meg az elpakolás sorendjét azáltal, hogy lényegében mindig újra kiszámolják a súlyokat, ami felelős a tárgyak elpakolási sorrendjéért (nem ragaszkodva egy előre meghatározott pakolási sorrendhez). Az [1] tapasztalati eredményei alapján a GH-típusú algoritmusok jobban viselkednek az általuk megadott és tesztelt bemeneti példákon. A célom az volt, hogy olyan heurisztikákat adjak meg, amelyek a szakirodalom szerint legjobb GH-típusú heurisztikákkal a gyakorlatban **versenyképesek**, de ugyanakkor mégis alapvetően az FFD-alapú heurisztikák módosításával keletkeznek.

Módszer. Ezért definiáltam és teszteltem az FFD-szerű algoritmusok olyanfajta módosításait, amelyek bizonyos gyakorisággal módosítanak az előre felállított sorrendhez képest azáltal, hogy „hátranyúlnak” és nem feltétlenül a soron következő tárgyat választják ki el pakolásra. Vagyis az alapötlet az volt, hogy az ellenpéldákban rejlő nehézséget elkerülendő, olykor szakítunk az előre felállított pakolási sorrenddel (azaz egy FFD-variáns előkészítő lépésében, valamely súlyfüggvény által megszabott sorrenden). Így az általam definiált algoritmusok annyiban hasonlítanak az FFD-szerű algoritmusokhoz, hogy:

- az elején, az előkészítő lépésben rendezik az inputot valamely súlyfüggvény használatával,
- pakolásra a *First Fit* (FF) szabályt használják.

Azonban a pakolás sorendjén változtatni fognak egy egyedi szabályrendszer alapján.

A következőkben a fenti ötletek alapján megtervezett algoritmusokat definiálom. Fontos megjegyezni, hogy ezek eddig a szakirodalomban **nem publikált variánsok**, azaz saját magam által fejlesztett algoritmusok. Mindegyik algoritmus saját szabályrendszer alapján működik és rendelkeznek különféle paraméterekkel, amelyek működését és használatát is ismertetni fogom. Az algoritmusok (és a paraméterbeállítások) hatékonyságát pedig benchmark példákon való kísérletezéseken mutatom majd meg.

Amint azt látni fogjuk a tapasztalati elemzéseken keresztül, az újonnan definiált algoritmusok között vannak olyanok, amelyek versenyképesek a gyakorlatban (a szakirodalom szerint [1] a cloud szerverek kapcsán) is alkalmazott DP és L2 algoritmusokkal.

A következőekben felsorolt algoritmusok az így kapott variánsok, amelyek neveit én adtam (FFDRev, FFDRevAdv, FFDRatio, FFDVal, FFDGroups, FFDBox, FFDBG), próbálva utalni az algoritmus jellegére.

Először ezek általános ötletét és elméleti hátterét írom le. Majd a későbbiekben bemutatom a pontos működésüket és a megvalósításhoz használt programozási szerkezeteket. Nem mindegyik algoritmus ért el figyelemre méltó eredményeket, de ennek ellenére, hogy jobban látszódjon a gondolatmenet, meghagytam azokat az elképzeléseket, és a kísérletek egy részét is a definiált algoritmusok kapcsán, amelyek végül nem bizonyultak versenyképesnek.

4.3.1. FFDRev & FFDRevAdv algoritmusok

Az **FFDRev** alapötlete az, hogy minden második elpakolandó tárgyat a sor legvégéről fogja venni. Vagyis az elpakolási sorrend úgy fog kinézni a rendezett I tárgy halmazra, hogy: $I^1, I^n, I^2, I^{n-1}, I^3, I^{n-2}, \ldots$ Így gyakorlatilag alternálva próbál olyan párosításokat keresni, hogy a legnagyobbat a legkisebbel, majd a második legnagyobbat a második legkisebbel, és így tovább.

Az alap FFDRev algoritmus azon hibáját, hogy ha túl nagy tárgyak vannak elől, amik mellé nem férnek be a kicsi, sor végi tárgyak, és ezáltal elkezdi a kicsiket egy ládába csoportosítani az algoritmus, szerettem volna kiküszöbölni az **FFDRevAdv** algoritmussal. Ezért arra gondoltam, hogy figyelni fogom a tárgyak méretét úgy, hogy addig nem kezdek bele a felváltva történő pakolásba amíg nem rakunk el egy olyan nagy (a sor elején lévő) tárgyat, ami mellé a legkisebb tárgy befér.

4.3.2. FFDRatio

Az **FFDRatio** algoritmusnál a felhasználó megadhat egy tetszőleges egész számot, X-et. Ezt a számot felhasználva a pakolás során $1/\mathrm{X}$ eséllyel fog hátranyúlni a pakolási sorban és elpakolni az aktuálisan legkisebb, még el nem pakolt tárgyat. Vagyis ez az algoritmus már nem lesz determinisztikus, tehát előre nem lehet tudni mikor fog hátrnyúlni, így nem adható meg előre a pakolás sorrendje. A tapasztalati kísérleteim alapján arra jutottam, hogy az $\mathrm{X} = 15$ egy jó választás (azaz $1/15$ eséllyel választani kis tárgyat).

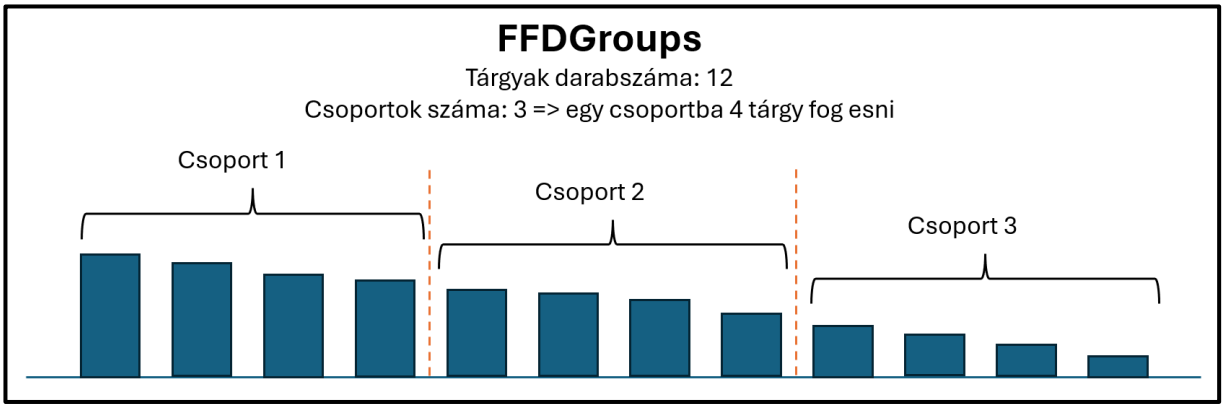
4.3.3. FFDVal

Ennél az algoritmusnál 50% az esély arra, hogy a még el nem pakolt tárgyak közül a sor elején állót pakoljuk el, illetve 25-25% annak az esélye, hogy a sor közepéről vagy a végéről. Ennek az algoritmusnak természetesen akkor van nagyobb esélye a javításra, ha nagyon sok a tárgyunk és releváns különbség van a sorrend elején lévő *„nagy”*, a sorban középen lévő *„közepes”* és a sor végén szereplő *„kicsi”* tárgyak között. (Csökkenő sorba rendezést követően a tárgyak első 1/3-a a nagyok, 1/3-a és 2/3-a között a közepesek, míg a kicsit az utolsó harmadban lévők.)

4.3.4. FFDGroups

Az **FFDGroups** algoritmus alapötlete abból jött, hogy szerettem volna a tárgyakat a méretük alapján csoportosítani (más szóval klaszterezni). Miután a tárgyakat az igényeik alapján nem növekvő sorrendbe rendeztem, a csoportok kialakításához veszek egy küszöbszámot és ezzel egyező darabszámú, nagyjából egyforma méretű csoportot hozok létre. Amint azt a 2. ábra

illusztrálja, az 1-es számú csoportba az elpakolási sor elején álló tárgyak tartoznak, míg az utolsó csoportba (a küszöbszám megegyezik a csoport sorszámával) kerülnek a sor végén álló tárgyak. A kialakított csoportokat egyesével veszem (elsőnek az 1-es számú csoportot) és elpakolom az adott csoport összes tárgyát véletlen sorrendben a First Fit (*FF)* szabálya alapján. Ha elpakoltam az aktuális csoport összes elemét, akkor lépek tovább a következő csoportra és elpakolom az abban szereplő tárgyakat is véletlenszerűen. Az algoritmus futása akkor ér véget, amikor az utolsó csoport utolsó elemét is elhelyeztük egy ládába.

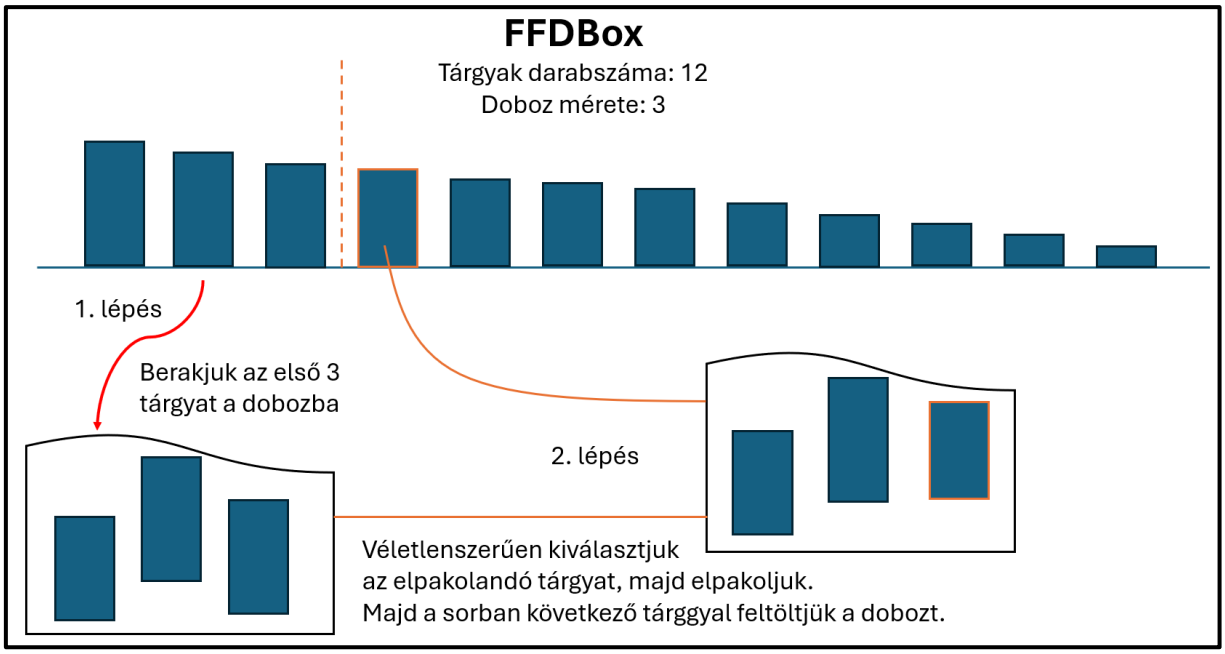


2. ábra. *Az FFDGroups algoritmus csoportjainak vizualizációja.*

4.3.5. FFDBox

Az algoritmust az FFD-hez hasonlóan azzal kezdjük, hogy nem növekvő sorrendbe rendezzük a tárgyakat, ezt a sorrendet nevezzük a tárgyak elpakolási sorának vagy sornak. A következő részt úgy lehetne elképzelni, hogy van egy dobozunk, amibe valahány darab tárgyat rakunk ideiglenesen (a dobozba annyi tárgy fér, amennyit a felhasználó megad majd a későbbiekben egy paraméterben, de a doboz mérete nem egyenlő egy ládával).

Első lépésként feltöltjük a dobozunkat a paraméter szerinti tárgy mennyiséggel az elpakolási sor elejéről (ezeket a tárgyakat eltávolítjuk az elpakolási sorból). Majd a dobozból véletlenszerűen kiválasztunk egy tárgyat, és azt elpakoljuk egy ládába a végleges helyére. Így lett egy üres hely a dobozban, ezt az üresedést feltöltjük a pakolási sor következő elemével. Ezt addig folytatjuk, amíg el nem fogynak a tárgyak (természetesen az algoritmus futásának végéhez közeledve, amikor már nincsen a sorban tárgy akkor még a dobozban lévő tárgyakat el kell pakoljuk). (Lásd a 3. ábrán.)



3. ábra. *Az FFDBox algoritmus stratégiájának vizualizációja.*

4.3.6. FFDBG

Ez az algoritmus az FFDBox és az FFDGroups algoritmusok egyesítésével hoztam létre. A méretük szerint klaszterezett tárgyakat az FFDGroups alapján osztom fel csoportokra. Viszont a csoportokon belül a tárgyakat nem teljesen véletlenszerűen, hanem az FFDBox dobozos szemlélete alapján pakolom el.

Ennek pontos működését, valamint az összes algoritmus paraméterlistáját és azok pontos leírását a későbbiekben ismertetem. A paramétereket tapasztalati úton választottam ki. Hiszen a paraméterek megfelelő beállítása segíthet célunkban, a jobb eredmények elérésének érdekében.

5. A program ismertetése

Ebben a fejezetben az algoritmusok megvalósítása során szemlőtt tartott gondolatmeneteket és az algoritmusok részleteit fogom ismertetni. A fejlesztés során **PyCharm** [18] fejlesztő környezett használtam és az algoritmusok **Python** nyelven készültek el. Minden elkészült kód, eredmény táblázat és dokumentum megtekinthető az alábbi GitHub felületen: https://github.com/Pityundra/Pakolas_tdk [19].

Az elkészített kódok mappaszerkezetének bemutatása:

- Algorithms (Elkészített algoritmusok egy, kettő és három dimenzióra egyaránt)
  - FFD: Elem- és Ládaközpontú szemlélettel
  - FFDDet: FFDRev, FFDRevAdv
  - FFDNotDet: FFDBox, FFDGroups, FFDBG, FFDRatio, FFDVal
  - GH: DotP és L2
- BadExamples (Ellenpéldák)
  - epsilon.txt (ellenpélda háromdimenziós esetekre)
- Data
  - Az osztályokkal és az optimumokkal rendelkező benchmark példák dimenziókként mappába rendezve. Az osztályokkal rendelkező példák alatt itt és a következőkben azok a példák értendőek, amelyek a szakirodalmak alapján generált tesztpéldák, de nem ismert az optimumuk. Ezek egyenletes eloszlással generálódnak, a szabályokat a későbbiekben részletezem.
  - GenerateFiles
    - benchmark példákat generáló fájlok
- Resources (segéd fájlok)
  - Bin (bin1D.py, bin2D.py, bin3D.py)
  - Item (item1D.py, item2D.py, item3D.py)
  - PlacItem (dataLoad.py, simpleLowerBound.py, weightInform.py)
- Results (Eredmények)
  - 1D_Results
  - 2D_Result
  - 3D_Results
  - Egyesített eredmények.xlsx
- main.py (Az elkészült algoritmusok meghívása különböző bemenetekre)

A fejlesztés során azt a szemléletet követtem, hogy minden egyes algoritmus külön fájlban készült el és azokon belül algoritmusonként kezelem a különböző dimenziójú eseteket. Igyekeztem könnyen értelmezhető, átlátható rendszerű programot készíteni, amely képes a beolvasott adatokon egyszerre több különböző algoritmust is futtatni függvényhívások segítségével.

5.1. Az algoritmusokhoz szükséges segédfájlok megvalósítása

Amikor tervezni kezdtem, még elméleti szinten az algoritmusok megvalósítását, igen hamar szembesültem vele, hogy szükségem lesz olyan adatreprezentációs eszközre, amiknek a segítségével gyorsan le tudom kérni, mondjuk a tárgy egyes dimenzióiban az igényét vagy a ládák töltöttségi szintjét. Ezért hoztam létre az **Item** és a **Bin** osztályokat, amelyek egy-egy tárgy vagy láda tulajdonságait tárolják és a **WeightInform** osztályt, ami egy tárgyat, egy ládát és a kettőjük kapcsolatából számított súlyt tartalmazza.

5.2. Item és Bin osztály

Az Item osztály egy tárgyat fog reprezentálni, konstruktorának szüksége van egy sorszámra és a tárgy dimenziók szerinti igényeire. Egy tárgy létrehozása után kiszámolom a tárgy néhány fontosabb tulajdonságát is (úgy, mint az igények összege, átlaga) és a kezdeti súlyát a tárgynak 0-ra állítom (a súly nem mindegyik algoritmusnál használatos). Annak érdekében, hogy minél olvashatóbban tudjak dolgozni a tárgyakkal, felüldefiniáltam az osztály kiíratását.

A Bin osztályban egy láda létrehozásához szükség van egy indexre, ami a feladatban a láda sorszámát fogja tárolni, illetve a láda dimenziónkénti maximális kapacitására. Ezek mellett egy ládáról eltárolom, hogy dimenziónként mennyi az aktuális töltöttsége és mennyi szabad hely van benne, valamint az összes dimenziónkénti összegét, illetve azt is, hogy hány darab tárgy van az adott ládában és a benne lévő tárgyakat egy listába mentem el. Létrehoztam egy függvényt *addItem(item)* névvel, ami csak egy tárgyat vár paraméterül (a ládakapacitás ellenőrzést az algoritmusokban végzem, ezért itt a tárgy elhelyezése után nem sértjük meg azt).

5.3. PlaceItem metódus

Az új FFD-n alapuló algoritmusok megvalósítása során a hangsúly az éppen elpakolandó tárgy kiválasztásán van. Miután kiválasztottam a tárgyat, azután fogom megkeresni a tárgy helyét, vagyis egy olyan ládát, amibe a tárgyat elhelyezve nem lépi túl a láda maximális kapacitását. Ha nincs ilyen láda, akkor új üres ládát kell nyitni. Ezt a folyamatot tartalmazza a *placeItem* metódus, amelyből készítettem mindegyik vizsgált dimenzióra egy sajátot. Ezeknek a metódusoknak a paraméterei rendre: *item* – elpakolandó tárgy, *bins* – eddig felhasznált ládák, *binsIndex* – ládák indexelése (a nullával való indexelés elkerülése végett), *binSize* – a használatos ládák dimenzióinak kapacitásvektora. A visszatérési értékei pedig az eddig felhasznált ládák és az utolsó láda indexe. A metódus működése röviden annyi, hogy megkeresi az első olyan ládát, ahova a kapott tárgy belefér (nem sérti a láda maximális kapacitását) és elpakolja oda.

5.4. Az algoritmusok megvalósítása

Az Algorithms mappán belül található az összes általam készített és tesztelt algoritmus. A következő részben bemutatom a függvényként meghívható algoritmusokat és a paramétereiket. Az FFD- és a GH-típusú algoritmusok esetén létrehoztam egy külön fájlt arra, hogy kiválassza a felhasználó által adott a paraméterek alapján azt, hogy melyik algoritmust futtassa.

5.4.1. Az FFD megvalósítása

FFD Item-centric ***(Elemközpontú FFD)***

Rendezzük a tárgyakat csökkenő sorrendben a súlyuk szerint.
FOR *l*=1 **to** *n* **do**
Helyezzük el az *l* tárgyat az első ládába, amibe belefér.
endfor

4. ábra. *Elemközpontú FFD pszeudókódja.*

Az *FFD(SAP, centric, items, binSize, dataName)* függvény az Algorithms és azon belül az FFD mappában található ffd.py fájlban. Az FFD() függvény előkészíti a terepet az algoritmusok számára, úgy, hogy megnézi a SAP (sum, avg vagy prod) paramétert és a megfelelő súlyérték alapján rakja sorrendbe a bejött tárgyakat. Majd megvizsgálja a „centric" és a „binSize" paramétereket, az első megmondja, hogy elem- vagy ládaközpontú szemléletet alkalmazunk, a második pedig, hogy mekkora dimenzióban dolgozunk (a ládák dimenziónkénti maximális kapacitása egy vektorban van tárolva és ennek a hossza megmondja a dimenziók számát).

FFD Bin-centric ***(Ládaközpontú FFD)***

Rendezzük a tárgyakat csökkenő sorrendben a súlyuk szerint.
While (Amíg) van elpakolandó tárgy **do**
Nyissunk egy új ládát
While (Amíg) van tárgy, ami belefér a ládába **do**
Helyezzük el a „legnagyobb" (súlyú) hátralévő tárgyat, ami belefér a ládába
endwhile
endwhile

5. ábra. *Ládaközpontú FFD pszeudókódja.*

Ezek után már a program tudja, hogy pontosan melyik függvényt kell majd meghívnia az alábbiak közül: *FFDIC1(items, binSize), FFDIC2(items, binSize), FFDIC3(items, binSize), FFDBC1(items, binSize), FFDBC2(items, binSize), FFDBC3(items, binSize).* Ezek nem mások, mint az elem- és ládaközpontú szemléletek (4. és 5. ábra) megvalósításai különböző dimenziószámok esetén. A fejlesztés során arra a döntésre jutottam, hogy az átláthatóság

céljából nem az algoritmus törzsét készítem fel a különböző dimenziókra, hanem különválasztom azokat, a nagyobb kódismétlést vállalva.

5.4.2. A GH algoritmusok megvalósítása

A fentebb ismertetett FFD variánsoknál az [1] cikkben próbáltak létrehozni pontosabb eredményeket hozó algoritmusokat. Ahogy azt már korábban említettem, a GH algoritmusok működéseinek alapja, hogy minden egyes lépésben újraszámolják a súlyokat, ami alapján az elpakolandó tárgyat kiválasztják. A Dot-Product és az L2 algoritmusok ennek a súlyértéknek a kiszámítási matematikai alapjában különböznek.

A GH() függvény feladata az, hogy eldöntse a paraméterek alapján, hogy a DotP vagy az L2 algoritmust használjuk, és hogy mekkora dimenzióban vagyunk. Ezek alapján meghívja a megfelelő függvényt.

A DotP.py fájlban találhatóak a DotP algoritmus megvalósításai 1, 2, 3, 4 és 6 dimenzióban. A DotP algoritmus megvalósítása vázlatosan úgy néz ki, hogy van egy külső *while-ciklusom,* ami addig megy, amíg vannak tárgyak az *itemCopy* listában. Ezen belül van egy dupla *for-ciklusom* a külső ciklus a tárgyakat veszi egyesével végig, míg a belső ciklus a ládákat. Ezáltal mindegyik tárgy mindegyik ládába bele lesz próbálva. Viszont csak azokra az esetekre számolok súlyokat, ahol a tárgyak el is férnek az adott ládába. Az ötlet alapja ezen a súlyszámolásban teljesedik ki.

A DotP algoritmus esetén úgy számolja ezt a súlyt, hogy dimenziókként összeszorozza a tárgy méretét és a láda szabad kapacitását, majd a szorzatokat összegzi, ezt az értéket eltárolja a tárgy súlyaként. Ezek után, veszi a legnagyobb súlyú tárgyat és elpakolja a megfelelő ládába. Ha nem tudott egyetlen elemre sem súlyt kiszámítani az algoritmus, akkor új ládát fog nyitni. Azt érdemes megjegyezni, hogy minden egyes pakolás előtt újra kell számolni az össze lehetséges súlyt, ezáltal az algoritmus összes számítási igénye magasabb, mint az FFD-alapú algoritmus variánsoké.

Az L2 algoritmus elve hasonlít az előbb ismertettet Dot-Productéhoz. Az L2 dimenziónként veszi a tárgy mérete és a láda szabad kapacitásának különbségét és ezen értékek összege négyzetre emelve fogja megadni a súlyt. Mivel az L2 algoritmus különbséget számít, ezért a GH-tól eltérően itt a legkisebb súlyú tárgyat fogjuk elpakolni.

A GH-típusú algoritmusoknál használom a Grasp[k] technikát [1], ami annyit tesz, hogy nem a legjobb, hanem a k-adik legjobb súlyértékű tárgyat pakolják el az algoritmusok. A k értékét a

paraméterekben adhatja meg a felhasználó az algoritmus meghívásakor (ha 1-est ad át, akkor az megfelel annak, mintha az alap algoritmust futtatná).

5.4.3. FFDRev megvalósítása

```
FFDRev

    Rendezzük a tárgyakat csökkenő sorrendben a súlyuk szerint.
    While (Amíg) van elpakolandó tárgyunk do
        Helyezzük el az l-edik tárgyat az első ládába, amibe belefér.
        Helyezzük el az (n+1)-l-edik tárgyat az első ládába, amibe belefér.
        if Csak egy tárgy van még, amit nem pakoltunk el
            Helyezzük el az utolsó tárgyat az első ládába, amibe belefér.
      endWhile
```

6. ábra. *FFDRev pszeudókódja.*

Elindítok egy *while-ciklust*, ami addig fog futni, amíg vannak elpakolandó tárgyak és kimentem a legnagyobb tárgyat (*itemF = items[0]*) és a legkisebbet (*itemL = items[len(items)-1]*). Majd először a legnagyobb tárgyra, majd a legkisebbre meghívom a *placeItem()* metódust, ezáltal garantálom a váltakozást, illetve következő lépésként pedig törlöm a két elpakolt tárgyat a listából. (Lásd 6. ábrán a részleteket.) A folyamatos eltávolítással azt érem el, hogy a statikus kiválasztások mindig jó tárgyat fognak adni.

5.4.4. FFDRevAdv megvalósítása

```
FFDReVAdv

    Rendezzük a tárgyakat nem növekvő sorrendben a súlyuk szerint.
     While (Amíg) van elpakolandó tárgy do
        Pakoljuk el súly szerinti legnagyobb tárgyat.
        FOR i in ládák
           if i-edik ládába belefér az utolsó tárgy
                Pakoljuk el az utolsó tárgyat az adott i-edik ládába.
                FOR l=1 to n/2 do
                    Helyezzük el az l-edik tárgyat az első ládába, amibe belefér.
                    Helyezzük el az (n+1)-l-edik tárgyat az első ládába, amibe belefér.
                if Csak egy tárgy van még, amit nem pakoltunk el
                    Helyezzük el az utolsó tárgyat az első ládába, amibe belefér.
                endfor
        endfor
    endwhile
```

7. ábra *FFDRevAdv pszeudókódja.*

Kezdetben indítok egy *while-ciklust*, ami addig megy, amíg van még tárgy a listában és ezen belül első lépésként megfogom a legnagyobb tárgyat és elpakolom, majd kiveszem a listából. Ezután indítok egy belső *for-ciklust*, amely megnézi az összes eddigi ládára, hogy valahova belefér-e a legkisebb tárgy, amit egy *if* szerkezettel ellenőrzők. Ha nem, akkor szimplán folytatódik tovább a *while-ciklus,* és elpakoljuk a következő sor elején lévő tárgyat. Viszont abban a pillanatban, hogy a legnagyobb tárgy mellé el tudjuk rakni a legkisebb tárgyat, akkor megtesszük ezt a lépést és ezek után már az **FFDRev** futását fogja követni az algoritmus.

5.4.5. FFDVal és FFDRatio megvalósítása

FFDVal

Rendezzük a tárgyakat nem növekvő sorrendben a súlyuk szerint.
FOR *l*=1 **to** *n* **do**
véletlen_szám = Kérünk egy véletlen egész számot 1 és 4 között
if véletlen_szám == 4
Helyezzük el a legkisebb aktuálisan pakolatlan tárgyat az első ládába, amibe belefér.
if véletlen_szám == 3
Helyezzük el az $\left\lceil \frac{n}{2} \right\rceil$el nem pakolt tárgyat az első ládába, amibe belefér.
else
Helyezzük el az *l* tárgyat az első ládába, amibe belefér.
endfor

8. ábra. *Az FFDVal algoritmus pszeudókódja.*

Az FFDVal alapját egyetlen *for-ciklus* fogja adni, ami a tárgyakat tartalmazó lista hosszáig fut. Még a fájl elején beimportáltam a *numpy* könyvtárat *np* névvel, azért hogy itt használni tudjam az *np.random.random_integers(start, stop)* függvényt, amely visszaad számunkra egy véletlen egész számot a [start, stop] zárt intervallumból. A programban én 1 és 4 közötti egész számot kérek, így 4 lehetséges értékem van [1,2,3,4] vagyis bármelyik érték $1/4$ eséllyel adja vissza a véletlen szám generátor. Készítettem egy *if* szerkezetet, ami megnézi a véletlenszerűen választott számom értékét, és ha az 4, akkor a rendezett lista végéről választja ki az elpakolandó számot, ha a kapott szám a 3-as, akkor a lista középső elemét ($\left\lceil \frac{n}{2} \right\rceil$) választja. Egyéb esetben (1,2 értékeknél) pedig a még elpakolásra váró tárgyak közül a legnagyobb súllyal rendelkezőt választjuk. Ez fog megfelelni annak, hogy ez utóbbi választás 1/2 eséllyel valósul meg (Lásd 8. ábra.)

FFDRatio

Rendezzük a tárgyakat nem növekvő sorrendben a súlyuk szerint.
A felhasználótól kapott pozitív egész érték legyen *X*
FOR *l*=1 **to** *n* **do**
véletlen_szám = Kérünk egy véletlen egész számot 1 és *X* között
if véletlen_szám == *X*
Helyezzük el a legkisebb tárgyat az első ládába, amibe belefér.
else
Helyezzük el az *l* tárgyat az első ládába, amibe belefér.
endfor

9. ábra. *Az FFDRatio pszeudókódja.*

Az FFDRatio megvalósításánál (9. ábra) $1/X$ (X felhasználó által adott pozitív egész szám) eséllyel szeretnénk a legkisebb tárgyat elrakni. a véletlen számot így generálom: *np.random.random_integers(1, X)* és ha a X -et kapunk, akkor elrakjuk a még el nem pakolt tárgyak közül a legkisebb súlyút, és minden más esetben pedig az elpakolási sor legelejéről választjuk a tárgyat.

5.4.6. FFDGroups megvalósítása

FFDGroups

Rendezzük a tárgyakat nem növekvő sorrendben a súlyuk szerint.
Az elpakolandó tárgyak számát elosztva a kért csoportok darabszámával (*CsoportokSzáma*) megadja a számunkra, hogy hány tárgy van egy csoportban (*CsoportTárgySzám*)

FOR *j*=1 **to** *CsoportokSzáma* **do**
If (Ha) nem az utolsó csoportban vagyunk
For *i*=1 **to** *CsoportTárgySzám*
Pakoljuk el a tárgyakat véletlen sorrendben
else (Az utolsó csoportban vagyunk)
For *i*=1 **to** Maradék tárgyak számossága
Pakoljuk el a tárgyakat véletlen sorrendben
endfor
endfor

10. ábra. *FFDgroups pszeudókódja.*

Az **FFDGroups** ötletének lényege, hogy felosztja a bejövő tárgyakat nagyjából egyenlő méretű csoportokra. Ehhez létrehozok egy *groupNumber* nevezetű változót, amiben az van letárolva, hogy hány darab (nagyjából egyenlő méretű) csoportot szeretnénk létrehozni. Illetve ennek

segítségével, valamint a tárgylistám hossza alapján kiszámolom, hogy hány darab tárgy esik egy csoportba, az alábbi módon: *splitNumber = int(ceil(len(itemsCopy) / groupNumber))*-t ($\left\lfloor \frac{Tárgyak\ darabszáma}{Létrehozandó\ csoportok\ szám} \right\rfloor$). Abban az esetben, ha a tárgyak száma nem pontosan osztható a készítendő csoportok számával, akkor az utolsó csoportban lesz több tárgyunk (az egyszerűség kedvéért lefelé kerekítünk, így ott mindig több tárgyunk lesz, vagy legalábbis sosem lesz kevesebb, mint a többi csoportnál). Ez a *for-cikluson* belül van kezelve, mégpedig úgy, hogy a ciklusom a csoportok darabszámáig megy, és a cikluson belül van egy feltételrendszerem. Ha még nem az utolsó csoport van soron, akkor indítok egy *for-ciklust*, ami annyit számol, amekkora a *splitNumber* értéke (amennyi tárgyat kell elrakni). A belső cikluson belül generálok egy véletlen számot 0 és *splitNumber-1-i* (hány darab tárgyat kell elpakolnunk, -1 a nullával kezdő indexelés végett és mínusz az *i* ciklusváltozó, ami számolja, hogy eddig hány tárgyat pakoltunk már el). Ha megkaptuk a véletlen számot, akkor elpakolja, az azzal a sorszám indexszel rendelkező tárgyat, ezáltal frissül a ládák adatai, majd töröljük a listából a tárgyat. Abban az esetben, ha az utolsó csoporthoz ér az algoritmus, akkor csak a véletlen szám felső korlátját kell máshogy megválasszuk, mégpedig *len(itemsCopy)-1*-nek, ami a még elpakolandó tárgyainkat tartalmazó lista hossza -1 (nullával kezdődő indexelés miatt).

5.4.7. FFDBox algoritmus megvalósítása

FFDBox

Rendezzük az elemeket nem növekvő sorrendben a súlyuk szerint.
FOR *i*=1 **to** *TárgyakDarabSzáma* **do**
If (Ha) a *DobozMérete* nagyobb, mint az *ElpakolandóTárgyakSzáma*
Generálunk egy véletlen egész számot 0 és az *ElpakolandóTárgyakSzáma*-1 között és elrakjuk az ezzel a sorindexszel rendelkező tárgyat.
else (Több elpakolandó tárgyunk van, mint a *DobozMéret*-e)
Generálunk egy véletlen egész számot 0 és az *DobozMéret* -1 között és elrakjuk az ezzel a sorindexszel rendelkező tárgyat.
endfor

11. ábra. *Az FFDBox pszeudókódja.*

Készítek egy *boxSize* nevezetű segédváltozót, amit a felhasználó adhat meg és meghatározza hány darab tárgyunk lehet egyszerre a dobozban. Az előzőktől eltérően itt csak egy *for-ciklusra* lesz szükségem, mégpedig egy olyanra, ami a tárgyakat tartalmazó lista hosszáig megy. Itt is van egy belső feltétel rendszerem, ha már nincs annyi tárgy, mint amekkora a doboz mérete, akkor a véletlenszám generálása így történik:

np.random.random_integers(0, len(itemsCopy)-1) (0 és az elpakolandó tárgyak hossza-1). De ha még van hátra elegendően sok tárgyunk, akkor a felső korlátunk értéke a *boxSize-1* lesz (A 11. ábra (is) mutatja mindezt).

5.4.8. FFDBG megvalósítása

FFDBG

Rendezzük az elemeket nem növekvő sorrendben a súlyuk szerint.
Számoljuk ki, hány darab csoportot hozzunk létre (*CsoportokSzáma*) és, hogy hány tárgy van egy csoportban (*CsoportTárgySzám*)

FOR *j*=1 **to** *CsoportokSzáma* **do**
If (Ha) nem az utolsó csoportban vagyunk
FOR *i*=1 **to** *CsoportTárgySzám* **do**
If (Ha) a *DobozMérete* nagyobb, mint az *CsoportTárgySzám*
Kérünk egy véletlen egész számot 0 és az *CsoportTárgySzám* -1 között és elrakjuk az ezzel a sorindexszel rendelkező tárgyat.
else
Kérünk egy véletlen egész számot 0 és az *DobozMéret* -1 között és elrakjuk az ezzel a sorindexszel rendelkező tárgyat.
endfor
else (Ha az utolsó csoportban vagyunk)
FOR *i*=1 **to** *TárgyakDarabSzáma* **do**
If (Ha) a *DobozMérete* nagyobb, mint az *ElpakolandóTárgyakSzáma*
Kérünk egy véletlen egész számot 0 és az *ElpakolandóTárgyakSzáma*-1 között és elrakjuk az ezzel a sorindexszel rendelkező tárgyat.
else
Kérünk egy véletlen egész számot 0 és az *DobozMéret* -1 között és elrakjuk az ezzel a sorindexszel rendelkező tárgyat.
endfor
endfor

12. ábra. *FFDBG pszeudókódja.*

Az FFDGroups és az FFDBox megvalósításának összefésüléséből egyszerűen megvalósítható ez az algoritmus. Futtatok egy for-ciklust a paraméterben megkapott groupNumber-ig és ezen belül megkülönböztetem azt az esetet, ha az utolsó csoportban vagyunk. Ha még nem vagyunk az utolsó csoportban, akkor indítok egy új for-ciklust a csoport darab számáig és azon belül használom az FFDBox-os elpakolást. Ha az utolsó csoportban vagyunk, akkor olyan ciklust indítok, ami végig megy az összes tárgyon és a dobozos szemlélet alapján pakolja el azokat (A 12. ábra illusztrálja mindezt.)

5.5. Az implementált algoritmusok és paraméterlistáik

FFD algoritmusok: *FFD(SAP, centric, items, binSize, dataName)*

1. SAP – Súly függvény típusa, ami lehet „*sum”*, „*avg*” vagy „*prod*” Ezt egyébként nem teljesen értem.
2. centric – Elem vagy ládaközpontú szemlélet: „*item”* vagy „*bin”*
3. items – Tárgyak listája
4. binSize – Ládák maximális kapacitása
5. dataName – A benchmark példa neve

GH típusú algoritmusok: *GH(alg, item, binSize, grasp, dataName)*

1. alg – Melyik GH algoritmus fusson, *„DotP”* vagy *„L2”*
2. items – Tárgyak listája
3. binSize – Ládák maximális kapacitása
4. grasp – Grasp[k] mérete
5. dataName – A benchmark példa neve

FFDRev(items, binSize, dataName)

1. items – Tárgyak listája
2. binSize – Ládák maximális kapacitása
3. dataName – A benchmark példa neve

FFDRevAdv(items, binSize, dataName)

1. items – Tárgyak listája
2. binSize – Ládák maximális kapacitása
3. dataName – A benchmark példa neve

FFDBG(items, binSize, GroupNumber, boxSize, runTime, dataName)

1. items – Tárgyak listája
2. binSize – Ládák maximális kapacitása
3. GroupNumber – Hány csoportra osszuk fel a bemeneti adatokat
4. boxSize – Az első hány tárgyból választhatunk véletlenszerűen
5. runTime – Hányszor futtassuk az algoritmust az adott bemeneten
6. dataName – A benchmark példa neve

FFDBox(items, binSize, boxSize, runTime, dataName)

1. items – Tárgyak listája
2. binSize – Ládák maximális kapacitása
3. boxSize – Az első hány tárgyból választhatunk véletlenszerűen
4. runTime – Hányszor futtassuk az algoritmust az adott bemeneten
5. dataName – A benchmark példa neve

FFDGroups(items, binSize, GroupNumber, runTime, dataName)

1. items – Tárgyak listája

2. binSize – Ládák maximális kapacitása
3. GroupNumber – Hány csoportra osszuk fel a bemeneti adatokat
4. runTime – Hányszor futtassuk az algoritmust az adott bemeneten
5. dataName – A benchmark példa neve

FFDRatio(items, binSize, ratio, runTime, dataName)

1. items – Tárgyak listája
2. binSize – Ládák maximális kapacitása
3. ratio – 1/ratio eséllyel pakolunk el egy tárgyat a végéről
4. runTime – Hányszor futtassuk az algoritmust az adott bemeneten
5. dataName – A benchmark példa neve

FFDVal(items, binSize, runTime, dataName)

1. items – Tárgyak listája
2. binSize – Ládák maximális kapacitása
3. runTime – Hányszor futtassuk az algoritmust az adott bemeneten
4. dataName – A benchmark példa neve

6. Benchmark példák (leírásuk, generálásuk és alsó korlátok)

Ahhoz, hogy majd a megvalósított algoritmusok helyességét és képességeit ellenőrizni tudjam, fontos a megfelelő teszt adathalmazoknak a létrehozása. A szakirodalmakban sokszor A. Caprara és P. Toth által írt tanulmányban [7] használt adathalmaz szerkezeteket vették alapul, pl. [1]-ben is. Négyféle bemeneti tesztpélda-halmazt készítettem és használtam és a következő a fejezetekben ismertetni fogom ezeket.

6.1. Bemeneti példák osztályokkal

A tesztpéldák osztályainak megtervezésében alapvetően a feldolgozott szakirodalmakban javasolt irányelveket követtem. Az első 5 osztály paramétereit egy az egyben az [1]-ben leírtak alapján valósítottam meg. A 6-os, 7-es és 8-as osztályokat a [5]-es cikkből választottam ki.

	Láda maximális kapacitása	**Tárgy minimális igénye**	**Tárgy maximális igénye**
class1	1000	100	400
class2	1000	1	1000
class3	1000	200	800
class4	1000	50	200
class5	1000	25	100
class6	100	1	50
class7	10	1	10
class8	40	1	35

1. táblázat. *Bemeneti példák osztályainak leírása.*

A fenti, 1. táblázat mutatja az osztállyal rendelkező bemeneti példák ládáinak maximális kapacitását és a tárgyak minimális és maximális kapacitását. A tárgyakat egyenletes eloszlás szerint, véletlenszám generátorral hoztam létre a táblázatban látható minimális és maximális igény közötti zárt intervallumból. A fenti osztályok elkészültek 1, 2, 3, 4 és 6 dimenziós példákra, a táblázatban szereplő korlátok egy dimenzióra értendőek, a nagyobb dimenzió számú esetekre mindegyik dimenzióra ugyan ezeket a megkötéseket használtam. Mindegyik osztályból 400, 800 és 1000 tárgyat tartalmazó példák készültek. Mindegyik dimenziószámra készítettem 3 bemeneti példát osztályonként.

6.2. Korreláló bemeneti példák

2, 4 és 6 dimenzióra készítettem, olyan bemeneti példákat, ahol a tárgyak mérete összefüggésben van egymással 400, 800 és 1000 darab tárggyal. Ezeket class9-nek és class10-nek neveztem el, mindkettő osztályban a ládák maximális kapacitása 150. Kétdimenziós esetben a class9 úgy néz ki, hogy a tárgy első dimenzió igényeként (méreteként) generálok egy véletlen számot a [20, 100] zárt intervallumból, nevezzük ezt az értéket *d1*-nek. A tárgy

második dimenzió igényéhez pedig felhasználom az előbb generált *d1* értéket, azáltal, hogy az intervallumot [*d1*-10, *d1*+10]-nek határozom meg. Ezzel a class9-ben a dimenziók között pozitív korreláció van. Class10 esetén pedig negatív korrelációt fogok létre hozni. Ez kétdimenziós esetben úgy néz ki, hogy a tárgy első dimenzió igényét szintén a [20, 100]-as zárt intervallumból generálom, de a második dimenziós igényét most [110-*d1*, 130-*d1*] zárt intervallumból generálom. A 4 és 6 dimenziós esetekben egyszerűen megismétlem a kétdimenziós esetre ismertetett lépést, így kapom azt, hogy négydimenziós esetben az első dimenzió a másodikkal és a harmadik dimenzió a negyedikkel fog kapcsolatban állni, korrelálni. Hatdimenziós esetben még az ötödik dimenzió fog korrelálni a hatodik dimenzióval. (Ezek megegyeznek az [1]-beli korrelált adatok halmazának a generálási módjával.)

6.3. Bemeneti példák ismert optimummal

tesztesetek (db)	Optimum	Láda maximális kapacitása
10	10	100
5	20	100
5	100	100
5	500	1000
5	1000	1000

2. táblázat *Bemeneti példák ismert optimummal.*

Az itt látható táblázatban (2. táblázat) azokat a bemeneti osztályokat írtam le, amelyeknek előre tudjuk az optimális megoldását. A feldolgozott szakirodalomban nem találtam ilyenfajta bemeneteken való kísérletezést, de ezeknél a példáknál pontosan fogjuk tudni az optimumtól való távolságot, és ha az új algoritmusok tudnak javítani az eddig használatban lévő algoritmusokhoz képest, jobban meg fogjuk tudni ítélni a javulás eredményességét.

A 2. táblázatban pl. az első sor azt jelenti, hogy 10 olyan fájlt készítek, amikben a ládák maximális kapacitása 100 mindegyik dimenzióban és a tárgyakat optimálisan 10 ládában lehet elpakolni. Itt a tárgyak mennyiségét nem tudjuk előre, és az igényeik 1 és a ládák max kapacitása között vannak.

6.4. Ellenpélda FFD-re

Ahogy a 3.2-es fejezetben említettem, egyes esetekben messze lehet az alap FFD algoritmusok eredménye az optimumtól, ezen elméletek alapján létrehoztam néhány bemeneti példát. Háromdimenziós esetben a $(\frac{1}{3}, \frac{1}{3} - \varepsilon, \frac{1}{3} + \varepsilon)$, $(\frac{1}{3} - \varepsilon, \frac{1}{3} + \varepsilon, \frac{1}{3})$, $(\frac{1}{3} + \varepsilon, \frac{1}{3}, \frac{1}{3} - \varepsilon)$ és szabálynak megfelelően hoztam létre egy ellenpéldát. A ládák maximális kapacitását 999-nek választottam az ε-t 1-nek és n-t 30-nak. Így kialakult 10-10-10 darab (334,333,332), (332, 334,333) és (333,332,344) méretű tárgy. Erre az optimum 10 felhasznált láda lenne úgy, hogy ládánként mindegyik fajta tárgyból 1 darab van.

6.5. Alsó korlátok számítása

Készítettem egy egyszerű alsó korlát számító algoritmust is, ami egyszerűen összeadja mindegyik dimenzióban a tárgyak igényeit és elosztja a ládák maximális kapacitásával, majd ezt az értéket felfelé kerekítve megkapjuk, hogy minimálisan hány ládára van szükségünk, az adott dimenzióban, hogy el tudjuk pakolni a tárgyakat. (Ha több dimenziós példáról beszélünk, akkor a legnagyobb ládaszámot adó, vagyis alsó korlátot adó dimenzió ládaszámát választjuk [8]). Azt még fontos megjegyezni, hogy ez az alsó korlát legtöbb esetben nem egyenlő az optimummal, akár messze is lehet attól, mert a tárgyak mérete miatt rések (szabad helyek) lehetnek az optimális pakolás ládáiban, amit ez a számítás nem vesz igénybe. Viszont, ha valamelyik algoritmus eléri az alsó korlátot, akkor kijelenthetjük azt, hogy optimális megoldást találtunk.

7. A tesztelés során használt paraméterek ismertetése

A benchmark példák futtatása során több paramétert is kipróbáltam az algoritmusokra, hiszen eltérő típusú bemeneti adatokon eltérő paraméterlista lehet hatásosabb. Az eredmények ismertetésekor mindig kiválasztottam az adott bemeneti példára a legjobb eredményeket adó paramétereket, és azokat az eredményeket tüntetem fel, helyezem el a táblázatokban.

A GH algoritmusokra (DotP és L2) alkalmaztam a Grasp[k]-t, ahol a k-nak 2-es, 3-as és 4-es értéket adtam. A tapasztalataim alapján nem érdemes túl nagyra választani a k értékét, mert javulás helyett romlást fog okozni.

Az FFDRatio esetén paraméterként megadhatunk egy számértéket, amely megadja, hogy 1/X eséllyel pakolja el az algoritmus legkisebb eddig még el nem pakolt tárgyat. A tesztesetek futása során kipróbáltam az algoritmust 1/2, 1/5, 1/10 és 1/15 valószínűségekkel. Ennél az algoritmusnál, minél kisebb esélyt adunk a „hátranyúlásra”, annál inkább fogjuk az alap FFD algoritmus eredményéhez közeli eredményt kapni.

Az FFDBox algoritmus esetén a harmadik, boxSize nevű paramétere határozza meg, hogy hány tárgy közül választunk véletlenszerűen. Kipróbáltam a doboz méretét több esetre is, konkrétan 3, 4, 5 és 6 méretű dobozzal. Az eredmények alapján egy- és kettő dimenziószám esetén a legjobb dobozméretnek a 3-as bizonyult, míg három dimenzió esetén a 4-es doboz méret hozta a legjobb eredményeket.

Az FFDGroups algoritmus esetén megadhatjuk, hogy hány darab csoportra oszthatjuk fel a bemeneti tárgyakat. Ezt is több lehetséges paraméterrel teszteltem, pontosan 4, 6, 10 és 20 darab csoportra. Itt a különböző dimenziókban eltérő paraméterek adtak jobb eredményeket. Egydimenzióban a 10 csoportos változat, kettő dimenzióban a 4 csoportos és három dimenzióban pedig a 20 csoportos változat bizonyult a legjobbnak.

Az FFDBG-t 3-féle paraméterrel futtattam, emlékeztetőül itt a doboz méretét és a csoportok számát is be lehet állítani. A 3-féle bemenetnél a doboz mérete és a csoportok mérete páronként 4-4, 6-4, és 5-3 voltak.

8. A kapott eredmények elemzése

8.1. Eredmények benchmark osztályokra

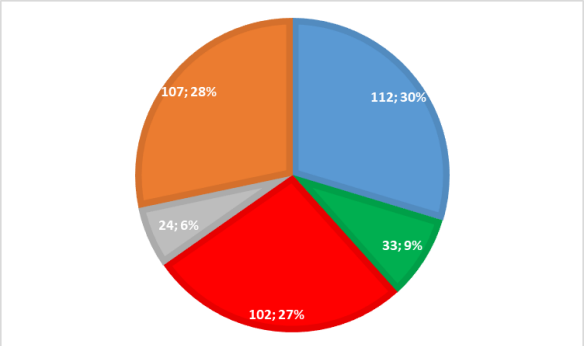
Az Egyesített eredmények dokumentumban készítettem táblázatokat az algoritmusok futási eredményeinek elemzésének a céljából. Mindegyik osztályból a 400, 800 és 1000 tárgyat tartalmazó példák eredményeit dolgoztam fel, mindegyik dimenziószám esetén.

Az alsó korlát alatt láthatóak a szakirodalmakban is ismertetett FFD Tárgy- és Ládaközpontú szemléletek, a DotP és az L2 algoritmusok (gp – Grasp[K] értéket használ) eredményei. Ezek után látható két új algoritmus, amelyek determinisztikusak (FFDRev, FFDRevAdv) és azok eredményei. Az utolsó 5 sorban találhatóak az új nem-determinisztikus algoritmusok. A nem-determinisztikus algoritmusokat (FFDBox, FFDGroups, FFDBG, FFDRatio, FFDVal) 100-szor futtatam le és a táblázatban a 100 futás átlagosan felhasznált ládaszám mellett feltüntettem a legkisebb és legnagyobb felhasznált ládaszámot, amit az algoritmusok szolgáltattak outputként. Az összegző táblázatokban kiválasztottam a legjobb eredményeket az adott algaritmus különböző paraméterű eredményei közűl.

A fő elemzési szempontom a már szakirodalomban [1] ismertetett algoritmusok eredményeinek az összehasonlítása az új, FFD alapú algoritmusokéval. A későbbi táblázatok könnyebb értelmezhetőségének érdekében színekkel jelöltem azokban az egyes értékeket. Ezek a későbbi táblázatokban karakterszínek lesznek. Itt a következő táblázatban (3. táblázat) adom meg azok jelentését, illetve összegzem a kapott eredményeket.

Összes tesztesetek száma		72	78	72	78	78	378
Dimenzió szám		**1D**	**2D**	**3D**	**4D**	**6D**	**Össz.**
	Megismételtük az FFD és GH algoritmusok eredményét	43	39	19	9	2	112
	Elértük az Optimumot	25	8	-	-	-	33
	Új alg. megjavítja az FFD és GH eredményeit is	4	10	21	31	36	102
	Nem sikerült megközelíteni	-	8	5	8	3	24
	FFD-n javít, de GH-n nem	-	13	27	30	37	107

3. táblázat. *A tesztek összesített eredményei és színkód magyarázat.*



1. diagram. *Az összesített eredmények megoszlása, százalékos formátumban.*

Ahogy az összegző táblázatban (3. táblázat) és diagramon (1. diagram) is látható, kékkel jelöltem a legjobb „megdöntendő” eredményeket (amiknél, ha jobbat érünk el, akkor biztosan javítunk), vagyis az FFD vagy GH algoritmusok közül a legjobbakat, de ha valamelyik algoritmusom szintén eléri az adott értéket, akkor az is kék színt kapott. Zöld színt azok az értékek kaptak, amelyek elérték az alsó korlátott, vagyis elmondhatjuk, hogy optimumban vagyunk, tehát, ha akarnánk se lehetne rajta javítani. Narancssárga színjelzést kaptak azok az eredmények, amelyek képesek voltak javulást elérni az FFD algoritmusokhoz képest, viszont a GH algoritmusok eredményeit nem sikerült megdönteni. Ezt azért tartottam fontosnak jelölni a táblázatokban, mivel az összes újonnan bemutatott algoritmus alapja az FFD, és ha már ezekhez képest sikerül javítani, az önmagában is egy jó teljesítmény. A piros szín jelenti a legjobbat a számunkra, mert az azt jelzi, hogy valamelyik új algoritmus sikeresen megjavítja mind az FFD és a GH algoritmusok eredményét is. Ezek a színkódolások érvényesek a tanulmányban található összes táblázatra.

Megjegyzem, hogy itt is és a későbbiekben is a futtatások során, mivel az általam definiált algoritmusok véletlen lépést is tartalmaznak, így, ellentétben a determinisztikus algoritmusokkal, többszöri futtatás esetén különböző eredményeket adhatnak. Emiatt a táblázatokban megtalálható az algoritmus futásainak az átlaga és mellette a legjobb és a legrosszabb elért futási eredmény is.

	1D					
	class4_800					
	példa_1		példa_2		példa_3	
Alsó Korlát	102		102		102	
FFD-bin	106		106		107	
FFD-item	102		103		103	
dotP	102		103		103	
dotP-gp	103		103		103	
L2	102		103		103	
L2-gp	103		103		103	
FFDRev	108		108		109	
FFDRevAdv	108		108		109	
FFDBox	102.42	102-103	103.0	103	103.0	103
FFDGroups	102.82	102-103	102.92	102-103	103.12	103-104
FFDBG	102.36	102-103	102.99	102-103	103.0	103
FFDRatio	103.41	103-104	103.6	103-104	104.31	104-105
FFDVal	104.81	104-105	104.74	104-105	105.49	105-106

4. táblázat. *1-dimenziós eredmények,*
class4-típusú bemenetre 800 db tárgyara, 3 különböző bemeneti példára.

Egydimenziós esetben (amint azt a 4. táblázat mutatja) összeségében elmondható, hogy az eddig használt algoritmusok nagyon jól teljesítenek, számos alkalommal elérték az optimumot vagy nagyon megközelítik az alsó korlátot, ezért nehéz ezeken javulást elérni, de a 72 tesztesetből 4-szer sikerült. A többi esetben pedig elértük ugyanazt az eredményt, mint az FFD- vagy GH-típusú algoritmusok.

	2D					
	class8_800					
	példa_1		példa_2		példa_3	
Alsó Korlát	368		363		368	
FFD-bin	377		375		379	
FFD-item	376		375		378	
dotP	377		374		379	
dotP-gp	379		378		381	
L2	376		375		379	
L2-gp	378		379		382	
FFDRev	397		398		396	
FFDRevAdv	397		399		395	
FFDBox	375.41	375-376	375.0	375	378.19	377-379
FFDGroups	376.88	376-379	376.19	375-377	379.38	378-380
FFDBG	375.46	375-376	375.0	375	378.2	377-379
FFDRatio	379.44	378-381	379.14	377-381	382.6	382-384
FFDVal	397.61	396-405	406.68	398-418	409.43	405-419

5. táblázat. *2-dimenziós eredmények,*
class8-típusú bemenetre 800 db tárgyara,3 különböző bemeneti példára.

Kétdimenziós esetekben (5. táblázat), sokkal nehezebb elérni az optimumot és az új algoritmusok vegyesebb eredményeket hoztak. De az elmondható, hogy az új FFD-n alapú algoritmusok igen szépen teljesítenek. Nagyon kicsi annak az esélye, hogy az ismert algoritmusokhoz képest rosszabb eredményt produkáljanak (ha ez az eset áll fent, akkor is nagyon minimális a „lemaradás”). Azt sem szabad elfelejtenünk, hogy azokban az esetekben, amikor elérjük a többi algoritmus által elért eredményeket, és nem sikerült rajtuk javítani, az értékek lehetnek akár optimumok is, amelyekről nem tudunk. Például három-, négy- és hatdimenziós esetekben már egyszer sem sikerül elérni az alsó korlátot. De ez könnyen lehet amiatt is, hogy az alsó korlátok értéke valójában jóval az optimum értéke alatt lehet.

	3D					
	class6_1000		class7_400		class8_1000	
Alsó Korlát	259		220		452	
FFD-bin	273		269		477	
FFD-item	271		269		476	
dotP	268		269		477	
dotP-gp	269		272		482	
L2	270		274		476	
L2-gp	272		272		485	
FFDRev	303		315		510	
FFDRevAdv	301		314		510	
FFDBox	272.34	270-274	269.42	268-271	475.98	475-477
FFDGroups	272.81	270-275	269.02	268-271	477.49	476-479
FFDBG	272.36	270-274	269.34	268-271	475.9	475-477
FFDRatio	276.83	275-280	270.86	269-273	484.78	482-488
FFDVal	288.93	286-290	295.24	293-305	501.05	498-506

6. táblázat. *3-dimenziós eredmények, 3 különböző bemeneti példa: class6 (1000 tárggyal), class7 (400 tárggyal) és class8 (1000 tárggyal).*

A háromdimenziós bemeneti példákra a vizsgálataim során azt tapasztaltam, hogy az esetek 67%-ban megjavítjuk a szakirodalmakban ismertetett FFD-variánsok eredményeit és az esetek 26%-ban a GH algoritmusokét is.

Négy- és hatdimenziós esetekben szintén kedvező, néhol kiugró eredményeket is láthatunk. Az esetek legnagyobb részében sikerült a szakirodalmakban ismertetett FFD-variánsok eredményét megjavítani, valamint számos esetben érünk el javulást a GH algoritmusokhoz képest is.

Összességében elmondható, hogy nagyobb dimenziószám esetén nagyobb eséllyel fognak az új algoritmusok jobb eredményt hozni, mint az eddig ismert GH vagy FFD algoritmusok. Sajnos az FFDRev, FFDRevAdv és az FFDVal eredményei általában nem versenyképesek, az FFDRatio algoritmusnak pedig néhány esetben van jobb eredménye, de nagy szórás van a kapott eredményei között. Főleg az FFDBox, az FFDGroups, és a kettő egyesítéséből létrejött FFDBG hoznak igazán jó eredményeket.

Arra még szeretném felhívni a figyelmet, hogy ha valaki hasonló körülmények között lefuttatja az algoritmusokat, a nem-determinisztikus mivoltukból adódóan nem pontosan ugyanezeket az eredményeket fogja kapni, de vélhetően az átlagokat tekintve hasonló eredményeket tapasztalhat.

8.2. Eredmények korreláló bemeneti példákon

Ahogy már korábban említettem, a class9 és class10 olyan bemeneteket tartalmaz, ahol egy tárgy dimenzió igényei között összefüggések vannak. A példákat kettő-, négy- és hatdimenzióban 400, 800 és 1000 tárgyra készítettem el, ami összesen 18 példát jelent.

Kétdimenziós esetben a 6 futási esetből ötször elértük ugyan azt az eredményt, mint az FFD és a GH algoritmusok és egy esetben sikerült javítani mindkettő eredményén. Hatdimenziós esetben azt lehet megfigyelni, hogy pozitív korreláció esetén (class9) sikerült megjavítani az FFD variánsok futási eredményeit. Negatív korreláció estén (class10) pedig mindegyik esetben sikerült nem csak az FFD algoritmusok, hanem a GH algoritmusok futási eredményeit is megjavítanunk.

6D						
Tárgyak darabszáma	400		800		1000	
Alsó Korlát	165		323		406	
FFD	201		404		501	
dotP	203		404		501	
L2	208		406		509	
FFDBox	201.81	201-203	403.41	402-406	501.46	500-503
FFDGroups	201.86	200-204	402.8	400-406	501.93	500-505
FFDBG	201.82	201-203	403.58	402-405	501.48	500-503
FFDRatio	202.82	200-204	402.42	400-404	501.53	500-503

7. táblázat. *6-dimenziós eredmények negatív korrelációval rendelkező bemeneti példákon (class10) 400, 800 és 1000 darab tárgy esetén.*

8.3. Eredmények ismert optimummal rendelkező bemenetekre

Egydimenziós esetben elmondható, hogy az FFD- és a GH-alapú algoritmusok egyaránt sikerrel veszik az akadályokat és van olyan konfigurációjuk, amelyikkel sikeresen elérik az optimumot mindegyik tesztesetre. De az is kijelenthető, hogy az általam készített algoritmusok sem maradnak alul. Mindegyik példára sikerült megtalálni az optimális megoldást.

Kétdimenziós esetben a GH-típusú algoritmusok közelítik meg legjobban az optimális eredményt, de nem túl sokszor sikerül elérni azt, nagyon sokszor opt+1 ládát használnak fel (opt itt az optimális ládaszámot jelöli). A nagyobb optimumok estén (minél több tárgyat kell elpakolni) az FFD algoritmusok egyre jobban elmaradnak a GH algoritmusokhoz képest. Az Új FFD alapú algoritmusaim eredményei a GH és FFD közé esnek. Többször sikerül elérni a legjobb eredményeket hozó L2 algoritmus eredményeit, de az is kijelenthető, hogy az L2

algoritmus mindegyik esetben jobb eredményeket produkál, mint az FFD. Kiemelnék egy kirívó esetet, amikor a legjobb eredményt az új algoritmusaim hoztak.

Algoritmus	Futási eredmények	
FFD-bin-prod	12	
FFD-item-sum	11	
DotP-gp1	11	
L2-gp1	11	
FFDBox-bx4-rt100	10.93	10-11
FFDGroups-gn4-rt100	10.98	10-11
FFDBG-gn5-bx3-rt100	10.96	10-11

8. táblázat. *Ismert optimummal rendelkező bemeneten, az új algoritmusok érik csak el az optimumot. Az új algoritmusok mellett szerepel az átlagos futási eredménye (100 futásra) és a legjobb és legrosszabb elért eredmény is.*

A háromdimenziós esetekben az figyelhető meg, hogy azokban az esetekben, ahol 10 láda az optimum (kb. 40-50 tárgy van egy példában), ott az új FFD-alapú algoritmusok nem csak az FFD- hanem a GH-jellegű algoritmusok eredményein is rendszeresen javítanak, sőt többször is elérik az optimumot. Viszont a nagyobb tárgyszámú esetekben ugyanúgy, mint a kétdimenziós esetekben az L2 algoritmus hozza a legtöbbször a legjobb eredményt, amit néhány esetben elérnek az új FFD-alapú algoritmusok.

8.4. Az ellenpéldákra kapott eredmények

Az 3.2. fejezetben leírt módon hoztam létre ellenpéldát a háromdimenziós esetekre, ami 30 tárgyat tartalmaz (10-10-10 a fentebb említett típusú tárgyakból). Az FFD algoritmusok a várt eredményt hozták, 15 ládára van szükségük, hogy elpakolják a 30 tárgyat. A DotP és L2 algoritmusokon nem fogott ki ez az ellenpélda és megtalálták az optimális megoldást (10 ládába pakolták el a tárgyakat).

Az FFDRev szabályából adódóan sikeresen megtalálta az optimumot, viszont az elmondható, hogy más típusú bemeneteken nem hozott túl jó eredményeket. A nem-determinisztikus algoritmusok közül szintén azok az algoritmusok (FFDRatio, FFDVal) hoztak optimumhoz közeli eredményt, amelyek a többi benchmark példán alul maradtak a többi algoritmushoz képest.

Az eddigi jó eredményeket hozó algoritmusok közül a legjobban az FFDBox teljesített, mivel 12 ládába sikerült elpakolnia a tárgyakat. Az kijelenthető, hogy ez az ellenpélda nagyon megnehezíti az FFD-típusú algoritmusok dolgát, de az új algoritmusoknak sikerült javulást elérniük. A lenti táblázatban (9. táblázat) mutatom be az algoritmusok futási eredményeit, a nem-determinisztikus algoritmusokat 100-szor futtattam az ellenpéldán és feltüntettem a futási eredmények átlagát és a legkevesebb és a legtöbb felhasznált ládát az adott algoritmustól.

Algoritmus	Futási eredmények	
FFD	15	
DotP	10	
L2	10	
FFDRev	10	
FFDRevAdv	14	
FFDBox	13.56	12-15
FFDGroups	14.11	14-15
FFDBG	14.14	14-15
FFDRatio	12.82	11-15
FFDVal	12.63	11-14

9. táblázat. *Futási eredmények a háromdimenziós ellenpéldákra. Az új algoritmusok mellett szerepel az átlagos futási eredménye (100 futásra) és a legjobb és legrosszabb elért eredmény.*

9. Konklúzió

Dolgozatomban egy manapság (az adatközpontok, cloud technológia és cloud számítások területén) nagyon fontos gyakorlati alkalmazással bíró feladatot, a többdimenziós vektorpakolási feladatot vizsgáltam.

Áttekintettem a szakirodalmak által legjobbnak talált algoritmusokat, és saját algoritmusokat is definiáltam, amelyek az FFD-alapú algoritmusok hatékonyságával rendelkeznek, ugyanakkor arra törekednek, hogy kiküszöböljük azoknak az ellenpéldáik által igazolt nehézségeit. Implementáltam az általam definiált algoritmusok mellet a szakirodalom legjobb algoritmusait.

Az algoritmusok hatékonyságainak az összehasonlítására a szakirodalmakból vett egy- és többdimenziós tesztpéldák mellett néhány FFD-ellenpélda generálását is elvégeztem. Az eredmények összehasonlításához számos, ezek outputjainak eredményeit összegző táblázatot készítettem.

Kiemelném, hogy az általam definiált algoritmusok közül az FFDBox, az FFDGroups és az FFDBG hozták a legjobb eredményeket. A vizsgálataim során, az FFD variánsok futási eredményeit 55%-ban javította meg valamelyik újonnan definiált algoritmus, ráadásul az esetek 27%-ában a legjobb eredményt hozták az osztállyal rendelkező bemeneti példákon. Csak az esetek 6%-ában maradtak alul az új algoritmusok a szakirodalomban ismertetettekhez képest, de ezekben az esetekben is a lemaradás az ismert legjobb eredményhez képest is maximum 4%-os a felhasznált ládák számában. Legjobban a 6-dimenziós példákon szerepeltek az új algoritmusok, mivel az FFD variánsokat az esetek 93% sikerült megjavítani és az esetek 46%-ban érték el a legjobb eredményt.

Összeségében elmondható, hogy sikerült ígéretes eredményeket adó algoritmusokat definiálni. Különböző dimenziószámok és feladatméretek esetén is több esetben versenyképesnek bizonyultak a szakirodalom legjobb algoritmusaival való összehasonlításban.

Hivatkozások

1. Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, Udi Wieder, Heuristics for Vector Bin Packing Microsoft Research Silicon Valley, Microsoft's VMM product group, (2011). https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/VBPackingESA11.pdf Utolsó elérés: 2024. április 27.
2. Microsoft Systems Center Virtual Machine Manager, http://www.microsoft.com/systemcenter/virtualmachinemanager. Utolsó elérés: 2024. április 27.
3. Filipe Brandao, João Pedro Pedroso, Bin packing and related problems: General arc-flow formulation with graph compression, Computers & Operations Research, Volume 69, pages 56-67, 2016. https://www.sciencedirect.com/science/article/abs/pii/S0305054815002762
4. Saikishor Jangiti, Shankar Sriram, Scalable and direct vector bin-packing heuristic based on residual resource ratios for virtual machine placement in cloud data centers, Computers & Electrical Engineering, Volume 68, 44-61, 2018. https://www.sciencedirect.com/science/article/abs/pii/S0045790617312168
5. Silvano Martello, David Pisinger, Daniele Vigo, The Three-Dimensional Bin Packing Problem, Operations Research, Vol. 48, No. 2., 256–267, 2000. https://doi.org/10.1287/opre.48.2.256.12386
6. Lars Nagel, Nikolay Popov, Tim Süß, Ze Wang, Analysis of Heuristics for Vector Scheduling and Vector Bin Packing. In: Learning and Intelligent Optimization: 17th International Conference, LION 17, Nice, France, June 4–8, 2023, 583–598, 2023 https://www.researchgate.net/publication/374960821_Analysis_of_Heuristics_for_Vector_Scheduling_and_Vector_Bin_Packing
7. A. Caprara, P. Toth ,Lower bounds and algorithms for the 2-dimensional vector packing problem (2001), Discrete Applied Mathematics Volume 111, Issue 3, 231-262, 2001 .https://www.sciencedirect.com/science/article/pii/S0166218X00002675
8. Nadia Dahmani, François Clautiaux, Saoussen Krichen, El-Ghazali Talbi, Iterative approaches for solving a multi-objective 2-dimensional vector packing problem, Computers & Industrial Engineering, Volume 66, Issue 1, 158-170, 2013. https://www.sciencedirect.com/science/article/abs/pii/S0360835213001745
9. What is Amazon EC2? - https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html Utolsó elérés: 2024. április 27.
10. What is Azure? - https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure Utolsó elérés: 2024. április 27.
11. Google Compute Engine - https://www.techtarget.com/searchaws/definition/Google-Compute-Engine Utolsó elérés: 2024. április 27.
12. Michael T. Goodrich, Bin Packing https://ics.uci.edu/~goodrich/teach/cs165/notes/BinPacking.pdf Utolsó elérés: 2024. április 27.
13. V. V. Vazirani. Approximation algorithms, Springer, 2003. ISBN 978-3-642-08469-0 ISBN 978-3-662-04565-7 (eBook) DOI 10.1007/978-3-662-04565-7
14. Shuchi Chawla előadásjegyzete (Approximations Algorithms), elérhető: https://pages.cs.wisc.edu/~shuchi/courses/880-S07/scribe-notes/lecture05.pdf Utolsó elérés: 2024. április 27.

15. E. G. Coffman, Jr. M. R. Garey, D. S. Johnson, Approximation algorithms for bin packing: a survey elérhető: https://www.labri.fr/perso/eyraud/pmwiki/uploads/Main/BinPackingSurvey.pdf
16. György Dósa, Rongheng Li, Xin Han, Zsolt Tuza, Tight absolute bound for First Fit Decreasing bin-packing (2013), Theoretical Computer Science, Volume 510, 13-61, 2013. .https://www.sciencedirect.com/science/article/pii/S0304397513006774
17. M. R. Garey, R. L. Graham, and D. S. Johnson. Resource constrained scheduling as generalized bin packing. Journal of Combinatorial Theory Series A, 21(3):257–298, 1976.
18. Pycharm weboldala - https://www.jetbrains.com/pycharm/?var=1 Utolsó elérés: 2024. április 27.
19. Kolozsi István Zoltán saját GitHub repository-ja - https://github.com/Pityundra/Pakolas_tdk Utolsó elérés: 2024. április 28.