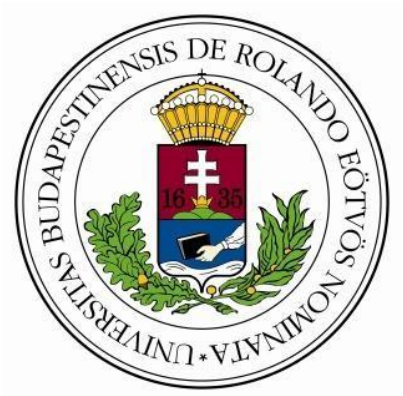


Eötvös Loránd Tudományegyetem

Informatikai Kar

Algoritmusok és Alkalmazásaik

Tanszék

A hátizsák probléma általános és speciális eseteinek a vizsgálata

Témavezető:

Dr. Szabó László Ferenc

tanszékvezető egyetemi docens

Szerző:

Novák Márk Attila

Programtervező informatikus MSc.

Budapest, 2023

Tartalomjegyzék

1. Bevezetés

A hátizsák probléma (angol irodalomban: Knapsack Problem, rövidítés: KP) egy nagyon híres klasszikus optimalizációs probléma a számítástudományban a kombinatorikus optimalizáció területén. Nevét arról a problémáról kapta, hogy adott egy hátizsák bizonyos súlykapacitással, amibe tárgyakat szeretnénk belehelyezni. Minden tárgynak van egy súlya és egy értéke, aminek figyelembevételével egy olyan kombinációs pakolást akarunk megadni, ahol a szerezhető összérték maximális anélkül, hogy túllépnék a súly korlátot.

A hátizsák feladata számos ipari ágazatban is megjelenik, mint például a logisztikában, szállításban, telekommunikációban. Széleskörű alkalmazása miatt rengeteg fajtája és speciális esete létezik. A legismertebbek közül megemlíthető a 0/1, töredékes, valamint 2-, 3- vagy több dimenziós hátizsák probléma. A feladat bizonyítottan NP-nehéz, tehát nem létezik olyan ismert algoritmus, amely minden lehetséges inputra hatékony gyors megoldást adna. Léteznek azonban olyan feltételek, amelyek mellett a probléma polinomiális időn belül megoldható. A feladat optimális megoldásának elérésére számos algoritmus létezik, amelyek különböző stratégiákkal és feltételek mellett meghozzák a kívánt eredményt. Azonban vannak helyzetek, amikor ezek nem alkalmazhatóak például futás idő vagy memória korlátok miatt. Ilyenkor egy minél jobban közelítő érték megtalálása rövidebb időn belül célravezetőbbnek tűnhet.

A diplomamunkám célja a hátizsák probléma speciális eseteinek vizsgálata. Ezen bizonyos esetek megoldására született módszerek áttekintése és gyakorlatban való implementációja, valamint összehasonlítása megadott szempontok alapján. Implementálásra az egyszerűség és könnyen olvashatóság miatt a választás a Python nyelv 3.11.2 verziójára esett. További cél lehetőség alapján performancia javítások alkalmazása az input adatok megvizsgálásával, heurisztikus módszerek, sajátos esetek, korlátok és feltételek melletti kísérletezés által. A 2D esetben a guillotine vágásos hátizsák probléma kerülne tanulmányozásra, ennek összevetése a nem guillotine vágásos megoldással. Végül pedig egy 2D guillotine vágásos algoritmust szeretnénk úgy általánosítani, hogy az a 3D guillotine vágás problémáját is megoldja.

2. 1D hátizsák probléma

A hátizsák probléma ahogy a bevezetőben is említésre került rengeteg sajátos esettel rendelkezik. Alap esetnek a legismertebb bináris 0/1 hátizsák problémát fogjuk tekinteni. Ez ugyanis más optimalizációs problémák megoldásának alapjaként is szolgál. A probléma nevét arról a feladatról kapta, ahol minden tárgyra el kell döntenünk, hogy az optimális megoldásban szerepel-e, ami binárisan is ábrázolható.

A problémát széles körű ismerete és használata miatt sokféleképpen is meg lehet fogalmazni, de a legáltalánosabb talán így néz ki: adott egy *n* elemű halmaz, ami tárgyakat tartalmaz. A hátizsákunk egy fix kapacitással (*W*) rendelkezik, a tárgyak pedig adott súllyal (*w*) és értékekkel (*v*). A feladatunk, hogy kiválasszunk a tárgyakból egy olyan részhalmazt, amelyeket súlyuktól fogva összegezve az így kapott összsúly nem fogja átlépni *W*-t, és teljesül az a feltétel is, hogy a tárgyak összértéke a legnagyobb. Tehát az optimális megoldást szeretnénk megkapni, amiről elmondható, hogy a tárgyak értéke maximális és nem lépjük át a súlyküszöböt. Ezáltal a feladatot formálisan a következőképpen is definiálhatjuk: $\max(\sum_{i=1}^{n} v_i x)$ – azaz maximalizálni szeretnénk az értékeket úgy, hogy teljesüljön még $(\sum_{i=1}^{n} w_i x) \leq W$. A formulában szereplő w_i és v_i rendre az adott i-edik tárgy súlya és értéke. Az *x* bináris együttható (ahol $x \in \{0,1\}$) fejezi ki, hogy a tárgy kiválasztásra került: ha a tárgyat kiválasztjuk 1, különben 0 értékkel. A továbbiakban alfejezetekre bontva vizsgálnánk meg a megoldására szolgáló kivitelezéseket és egyéb megközelítéseket. [1]

2.1. Nyers erő (Brute force)

A brute force algoritmusokat egy probléma megoldása során a legelső próbálkozások alkalmával használjuk, ugyanis alkalmazásukhoz nem szükséges nagy intuíció, az úgynevezett „favágó” módszernek felel meg. Általánosan a feladatok megoldásmenete során az összes lehetséges eset bejárása által kapjuk meg a megoldást. Habár garantálja az optimális érték megtalálását, ez a megközelítés leginkább kisebb problémák esetén valósítható meg. Nagy méretű bemeneti adatok esetén a futás idő szempontjából az összes eset végig próbálása rendkívül idő igényes. Esetünkben az összes lehetséges eset vizsgálata azt jelenti, hogy az algoritmus meghatározza a tárgyak összes lehetséges kombinációját ahogy kiválaszthatóak a

hátizsákba. Ahogy az alap esetnél megbeszéltük a bináris név eredetét, a megoldásunk során is ez a módszer kerül alkalmazásra. A tárgyak mindegyikére el kell döntenünk, hogy kiválasztjuk-e, ami 2^n lehetséges kombinációt jelent. Az összes lehetséges részhalmazt egy bináris számláló segítségével célnak megfelelően számon tudjuk tartani. Egy megoldás során az adott tárgyra a neki megfelelő pozíción levő bitet kell nézni, hogy a tárgy kiválasztásra került-e. [2]

Az implementálás során a fentebb megbeszéltek alapján járunk el, ahogy az 1. ábrán is látható. A külső ciklusunk i-vel iterál, ez fogja a bináris számot jelképezni, ami az elemek részhalmazát reprezentálja, a belső ciklus pedig az adott kombinációt járja be. Ahogy említettük, a bináris szám minden bitje egy tárgynak felel meg. A (i >> j) & 1 feltételünk ellenőrzi, hogy a j-edik elem benne van-e az adott i bináris számmal reprezentált részhalmazban. A „>>” operátor a bináris szám jobbra történő biteltolását jelenti. A j értéke fogja eldönteni, hogy melyik bitjét kell nézni i-nek. A „&” operátor a bináris „ÉS” műveletet jelöli, ami meghatározza, hogy az adott helyiértéken megjelenő 1-es bit az eredményben is így szerepeljen, más különben 0 értékkel. Ezáltal, ha a feltételben szereplő kifejezés értéke 1, akkor az i-edik bináris számban a j-edik bit a tárgy kiválasztását jelzi. A belső iterációval így meghatároztuk a bit számláló alapján az kiválasztott tárgyak részhalmazát. Ehhez kiszámítottuk ezen tárgyak által szerezhető összértéket és -súlyt. Ezt követően az így kapott megoldásokról kell eldönteni, hogy súly alapján belefférnek-e a hátizsák kapacitásába. Végezetül az érvényes megoldások közül a maximális összértéket adót választjuk ki.

Az algoritmus abból a szempontból megfelelt a kezdeti elvárásainknak, hogy megtalálja az optimális megoldást. Az implementáció során minden kombinációra végig kellett iterálni a bitszámlálón a tárgyak súly és érték összegzése miatt. Költség szempontjából ez minden lehetséges megoldásra *O(n)* műveletet jelent. A keresési terünk miatt $O(2^n)$ egyértelmű, mivel *n* tárgy esetén 2^n a lehetőségek száma arra vonatkozóan, hogy bevesszük-e az adott tárgyat. Ezáltal $O(n2^n)$ komplexitású futási időt kaptunk. Memória szemszögéből nézve két *n* méretű listát tartunk számon, ami *O(n)* költséget eredményez. [2], [3]

```python
def knapsackBruteForce(capacity, weights, values, n):
    maxValue, maxWeight, selectedItems = 0, 0, []
    for i in range(0, 2**n):
        actualValue, actualWeight, actualSelectedItems = 0, 0, []
        for j in range(0, n):
            if (i >> j) & 1:
                actualSelectedItems.append(j+1)
                actualWeight += weights[j]
                actualValue  += values[j]
        if actualWeight <= capacity and actualValue > maxValue:
            maxValue, maxWeight, selectedItems = actualValue, actualWeight, actualSelectedItems
    return maxValue, maxWeight, selectedItems
```

1. ábra: Brute force algoritmus implementálása

Összességében elmondható, hogy a brute force algoritmus jó kiindulási pontként szolgál a probléma megértésének elindulásában. Habár megtalálja az optimális megoldást, azt jelentős számítási költséggel éri el. Kisebb inputok esetében használható, több tárgy esetében viszont már nem érdemes az elvet alkalmazni a hosszú számítási idő miatt. Azonban olyan esetekben is képes a megoldás megtalálására, amikor a tárgyak értékei vagy súlyai egész számok. Természeténél fogva minden lehetőséget végig vizsgál, így nincs értelme optimalizálásnak. Implementáció szempontjából a legegyértelműbb megoldás menetet választottuk. Megemlítendő viszont, hogy rekurzív módszerrel $O(2^n)$ futás időt is elérhetünk, azonban a dolgozat következő fejezetében egy kifejezetten erre specializálódó elvet vizsgálunk meg.

2.2. Oszd meg és uralkodj (Divide and conquer)

Az „oszd meg és uralkodj” jellegű algoritmusok általánosan az olyan problémák megoldásában hatékonyak, amelyek felbonthatóak kisebb, egymástól független részproblémákra. A kapott részfeladatokat később önállóan külön-külön megoldjuk, majd a részproblémák eredményeit kombináljuk az eredeti probléma megoldásához. Habár a részproblémák általában egymástól függetlenek, esetleges ismétlődések előfordulása esetén többször is megoldhatja ugyanazt az alproblémát. Megjegyzendő, hogy rekurzív algoritmusokról beszélünk, amelyek hatékonysága nagyban függ az adott problémától.

Egy feladat megoldás során három főbb lépést alkalmazunk:

1. lépés: osszuk fel az adott kezdeti feladatot kisebb részfeladatokra. Például összefésüléses rendezés során a bemeneti tömböt két egyenlő részre bontjuk.
2. lépés: a kapott kisebb feladatokat megoldjuk rekurzívan. Összefésüléses

rendezésnél így a részfeladatok tömbjét újra kisebb részre bontjuk. Ezt a folyamatot mindaddig folytatjuk, amíg el nem érünk egy viszonylag egyszerű ponthoz, például amikor a tömbnek már csak két eleme van.

3. lépés: a részfeladatok megoldásait kombináljuk és így kapjuk meg az eredeti feladat megoldását. Ez az összefésülés esetén azt jelenti, hogy először rendezzük a bal oldalt, utána a jobb oldalt, végezetül a két oldalt összefésüljük. [4]

Az oszd meg és uralkodj módszer során tehát a megoldás a kisebb részfeladatok eredményének összekombinálásából áll elő, a rész eredményeket pedig a rekurzív hívások által kapjuk meg. Ez a megközelítés, ahogy említettük akkor a leghatékonyabb, amikor a részproblémák egymástól külön állóak. A hátizsák probléma esetében elmondható, hogy egyes tárgyak beválasztása vagy kihagyása kihatással van a soron következő elemre. Ez befolyásolja az így elérhető súlyt és összértéket, emiatt az alproblémákról elmondható, hogy összefüggőek. Mivel a tárgyak kiválasztása befolyással van egymásra, a független részfeladatokra bontás elve nem feltétlenül egyezik problémánkkal. Ezen oknál fogva az irodalom kevésbé foglalkozik a hátizsák probléma optimális eredményének megtalálásával az oszd meg és uralkodj módszerrel. Persze vannak próbálkozások, amik az elvet pontosan használják és a tárgyakat két kisebb halmazra osztják[1]. Fele tárgy esetében kevesebb a keresési tér, így könnyebben megoldható a probléma. Azonban, ha nem vesszük figyelembe az összes tárgyat, akkor a kapott megoldások nem biztos, hogy közel fognak kerülni az optimális megoldáshoz. Vannak további alkalmazások is, mint például az együtthatók megbecslése majd azok mentén való optimalizálás[2].

A stratégiát mi a hagyományostól eltérő módon próbáljuk meg alkalmazni: tárgyanként két részfeladatra bontjuk fel az eredeti feladatot. Ekkor minden tárgyra azt kell megvizsgálni, hogy az adott tárgy beválasztása (első alprobléma) vagy kihagyása (második alprobléma) közül melyik hoz jobb eredményt. [5]

[1] lásd Fernando A Morales, Jairo A Martinez „On the Implementation and Assessment of several Divide & Conquer Matheuristic Strategies for the solution of the Knapsack Problem” c. kutatását

[2] lásd Ali Ugur Guler, Emir Demirovic, Jeffrey Chan, James Bailey, Christopher Leckie, Peter J. Stuckey: „A Divide and Conquer Algorithm for Predict+Optimize with Non-convex Problems” c. kutatását

```python
def knapsackDivideAndConquer(capacity, weights, values, n):
    if n == 0 or capacity == 0:
        return 0, 0, []
    if weights[n-1] > capacity:
        return knapsackDivideAndConquer(capacity, weights, values, n-1)

    includeValue, includeWeight, includeItems = knapsackDivideAndConquer(capacity - weights[n-1], weights, values, n-1)
    includeValue  += values[n-1]
    includeWeight += weights[n-1]
    includeItems.append(n)

    excludeValue, excludeWeight, excludeItems = knapsackDivideAndConquer(capacity, weights, values, n-1)

    if includeValue > excludeValue:
        return includeValue, includeWeight, includeItems
    else:
        return excludeValue, excludeWeight, excludeItems
```

2. ábra: Oszd meg és uralkodj algoritmus implementálása

Az elv implementációja a 2. ábrán látható. Először is az alapfeltételeket ellenőrizzük a rekurziók miatt, hogy a kapacitásunk és a tárgyaink száma nagyobb-e mint 0. Ha az n-edik tárgy súlya nagyobb, mint a kapacitásunk, akkor alapértelmezetten nem tesszük bele. [6], [7]

A két alprobléma megoldása a következők szerint alakul:

1. alprobléma: az n-edik elemet bele tesszük a hátizsákba. Ez akkor történik meg, ha a súlya kisebb vagy egyenlő a maradék kapacitással. A függvény rekurzívan meghívja magát a hátralevő elemekkel és csökkenti a maradék kapacitást az n-edik elem súlyával. A rekurzív hívások során mindig visszatérünk a szerezhető összsúly és -értékkel, a kiválasztott elemek listája az utolsó n-edik értékével növekszik.
2. alprobléma: az n-edik elem nem kerül bele a hátizsákba, a függvény rekurzívan meghívja magát a maradék elemekkel és ugyanazzal a kapacitással.

A két alprobléma végeztével összehasonlítjuk a rekurzív hívásokkal visszaadott értékeket. A két eredmény közül kiválasztjuk a legnagyobb összértéket eredményezőt, amit még bele tudunk tenni a hátizsákba. Az algoritmus maximális futás ideje ezáltal exponenciális lesz: $O(2^n)$. Ez habár jobb, mint a fentebb bemutatott brute force megközelítés, nagy inputok esetén ugyan úgy nem hatékony.

2.3. Feljegyzéses módszer (Memoization)

Az előző két módszerünk majdnem hasonló, exponenciális futás időt hozott nekünk. Az oszd meg és uralkodj stratégia elve az volt, hogy az eredeti problémát kisebb részproblémákra bontsuk. Azonban nem vettük figyelembe azt, hogy ugyanazt a részproblémát többször is megkaphatjuk. Esetünkben az átfedő részproblémák kialakulhatnak, ha az elemek különböző kombinációja ugyan azt a kapacitást eredményezik. Emiatt előfordulhat, hogy ugyanazt a problémát többször is meg kell oldani. Amennyiben fel tudnánk használni az azonos részmegoldások eredményeit, jelentősen gyorsíthatnánk a futási időn.

Az ötlet tehát, hogy a megoldás javítható lenne, ha a már kiszámított részeredményeket valahogy el tudnánk tárolni. Így amennyiben egy olyan részproblémához jutunk, amit már megoldottunk, újra fel tudjuk azt használni fölösleges számolás nélkül. Lényegében ez lesz az alapelve a feljegyzéses módszernek. Az oszd meg és uralkodj módszerből kiindulva, hasonló módon fentről-lefele fogunk építkezni, így a részproblémák megoldásait a rekurzívan hívásokkal fogjuk kiszámolni. Ezeket a későbbi felhasználás céljából egy táblázatban tároljuk el. Ezzel az egymást átfedő részproblémák megoldása esetén jelentősen csökkenhet a futásidő. Általánosan ez a fajta megközelítés olyan problémáknál alkalmasabb, ahol a részproblémákat nem lehet könnyen azonosítani vagy rendezni, azonban közöttük jelentős átfedés van. [8], [9]

```python
def knapsackMemoizationHelper(c, capacity, weights, values, n):
    if n == 0 or capacity == 0:
        return 0
    if c[n][capacity] != 0:
        return c[n][capacity]
    if weights[n-1] <= capacity:
        c[n][capacity] = max(values[n-1] + knapsackMemoizationHelper(c, capacity - weights[n-1], weights, values, n-1),
                             knapsackMemoizationHelper(c, capacity              , weights, values, n-1))
    else:
        c[n][capacity] = knapsackMemoizationHelper(c, capacity, weights, values, n-1)
    return c[n][capacity]

def knapsackMemoization(capacity, weights, values, n):
    c = [[0 for i in range(0, capacity + 1)] for j in range(0, n + 1)]
    maxValue = knapsackMemoizationHelper(c, capacity, weights, values, n)
    maxWeight, selectedItems, i, j = 0, [], n, capacity
    while i > 0 and j > 0:
        if c[i][j] != c[i-1][j]:
            selectedItems.append(i)
            maxWeight += weights[i-1]
            j -= weights[i-1]
        i -= 1
    return maxValue, maxWeight, selectedItems
```

3. ábra: A feljegyzéses módszer algoritmusának implementálása

A 3. ábrán látható a módszer algoritmusának implementálása, ami egy fő és egy segédfüggvényből tevődik össze. A megoldás 3 főbb lépésből tevődik össze: [6], [9]

1. lépés: inicializáljuk a táblát, amiben a részeredményeket fogjuk tárolni.
2. lépés: kitöltsük a táblázatot a rekurzív hívások segítségével, ami által megkapjuk a lehetségesen elérhető legnagyobb összértéket.
3. lépés: eredmény megadása a táblázat alapján való visszakövetés által

A táblázat kitöltésére egy segédfüggvényt használunk, ez fogja elvégezni a rekurzív hívásokat. Először is biztosítjuk a kezdeti alap esetet: ha nincs tárgyunk vagy a hátizsák kapacitása 0, akkor nem tudunk elemeket választani. Ezt követően megnézzük, hogy a jelenlegi részproblémának az eredményét kiszámoltuk-e már az adott n-edik elem és az addig fennmaradó kapacitással bezárólag. Ha már kiszámoltuk, akkor csak szimplán visszaadjuk annak értékét. Amennyiben nem számoltuk ki még az aktuális részprobléma megoldásának az értékét, akkor megtesszük azt a rekurzív hívásokkal. Erre két lehetőséget kell figyelembe venni:

1. lehetőség: Az aktuális tárgy súlya kisebb vagy egyenlő, mint a hátizsák maradék kapacitása. Ekkor meg kell néznünk mi a megszerezhető maximum érték az alapján, hogy az adott tárgyat kiválasztjuk-e vagy nem. Két eset közül kell a kedvezőbbet választani:
   - a tárgy kiválasztásra kerül: csökkenteni kell a hátizsák kapacitását a súlyával és hozzá kell venni az értéket az addig elért összértékhez.
   - a tárgy nem kerül kiválasztásra: folytatjuk a rekurzív hívást a soron következő tárgyra.
2. lehetőség: A tárgy súlya nagyobb a kapacitásnál. Ekkor nincs választásunk, a következő tárgyra kell menni.

A táblázat kitöltésének a végeztével megkapjuk a megszerezhető legnagyobb értéket. Ezek után a kiválasztott tárgyak megadása következik, ami könnyen visszakövethető a táblázat alapján. Ehhez ellenőriznünk kell, hogy az aktuális részprobléma megoldásának értéke különbözik-e az előző tárgy során már kiszámított értéktől. Ha különböznek, akkor az aktuális elemet kiválasztottuk, így kivonjuk a súlyát j-ből, hogy megkapjuk a hátizsák fennmaradó kapacitását. Ezután csökkentjük az i-t, majd ismételve leellenőrizzük az előtte való tárgyra ugyanezt.

Az algoritmusunk a maximum érték meghatározásához *O(nW)* futás időt produkál (ahol *n* a tárgyak száma, *W* pedig a hátizsák kapacitása). A kiválasztott tárgyak visszafejésére használt ciklus, habár maximum *O(n+W)* futás időt tesz hozzá, ez az algoritmus futási idején nem változtat. A memória használatról is hasonlót mondhatunk el: a felhasznált táblában a részproblémák eredményei összesen *O(nW)* méretű helyet foglalnak. A kiválasztott tárgyakat *O(n)* méretű listában tároljuk, továbbá *O(n)* helyet még bele kell számítani a call stack-ben a rekurzív hívások miatt. A végeredmény memória komplexitás terén így marad *O(nW).* Az oszd meg és uralkodj módszerből kiindulva a javítás által eleget tettünk a kezdeti elvárásnak: lényegében jobb futásidőt kaptunk a nagyobb tárhelyért cserébe. [6]

2.4. Dinamikus programozás (Dynamic programming)

A dinamikus programozás stratégiáját leggyakrabban optimalizálási feladatok megoldására használják. Az algoritmus egy táblázatot épít fel, amiben minden sor és oszlop egy adott részproblémáját szimbolizálja az eredeti főproblémának. A cellákban a tárolt érték az adott részprobléma megoldását tartalmazza. A módszer a táblázat sor és oszlopain lentről-felfele való végig iterálásával halad, mely során az adott részfeladat kiszámolásával éri el a kívánt eredményt. Ez a hozzáállás azért válik hatékonnyá, mert a táblázatban eltároljuk a részfeladatok eredményeit, amit a későbbiekben felhasználunk a felesleges számítások elkerülése véget. Megfigyelhettük, hogy az előző feljegyzéses módszer is hasonló megoldást alkalmazott. Ott azonban csak azt számítottuk ki, ami szükséges volt, a rekurzió használata bonyolultabbá tette a megoldás menetét. Ezzel szemben a dinamikus programozással a részproblémák egymásra épülnek így iteratívan kell őket kiszámolni. [10]

A dinamikus programozás elve hasonlít az oszd meg és uralkodj módszerhez, hiszen a feladatot részproblémákra bontjuk, és a részfeladatok megoldásaiból kombináljuk össze az eredeti feladat megoldását. A különbség a két elv között, hogy míg az előző fentről-lefele, a dinamikus programozás alulról-felfele építkezik. Így a megközelítés során a részproblémák megoldásait a már említett iteratív módon számoljuk ki, amit majd egy táblázatban tárolunk a későbbi felhasználás céljából. A stratégia alkalmasabb az olyan problémák megoldásánál amikor a részproblémák könnyen meghatározhatóak, átfedés van közöttük és természetes módon rendezhetők. [10]

Általános esetekben egy feladat optimális eredményének a megtalálása a dinamikus programozás által a következő megoldás menetet követi:

1. lépés: részfeladatok meghatározása
2. lépés: az optimális részstruktúra tulajdonság fennállásának belátása
3. lepés: felírjuk az összefüggést a részfeladatok es az eredeti feladat megoldása között.
4. lépés: visszakövetés – ha a feladat kéri, akkor meg kell adni, hogy milyen úton kaptuk a megoldást.

Az említett optimális részstruktúra tulajdonság fennállásának belátása azt jelenti, hogy ami a rész problémában optimális, az optimális lesz a végeredményre nézve is. Vegyük példának a legrövidebb út keresés esetét. Amennyiben találunk egy utat két pont közt és az a legrövidebb, akkor bármely ezen az úton lévő két pont közti távolság is a legrövidebb lesz. Tehát alulról-felfele építkezve felírtuk az összefüggést a részfeladatok és az eredeti feladat megoldása között. A már kiszámolt részfeladatok megoldásait egy táblázatban kapjuk meg, ami így az eredeti feladat megoldását is tartalmazza.

A bináris hátizsák probléma esetében is a fentebb említett eljárásmódot követjük:

1. lépés: meg kell határoznunk a részfeladatot: legyen ez R[i, j]. Ezzel szeretnénk reprezentálni, hogy az első i tárgy közül választhatunk, és a kapacitásunk j.
2. lépés: az optimális részstruktúra tulajdonság belátása: Tegyük fel, hogy *o* egy optimális megoldása R[i, j]-nek. Ekkor két esetünk lehet:
   - ha $t_i \notin o$, akkor *o* optimális megoldása R[i - 1, j]-nek is, hiszen nem vettük bele a hátizsákba
   - ha $t_i \in o$, akkor $o \setminus \{ t_i \}$ optimális megoldása R[i - 1, j - w_i]-nek, ahol w_i az i-edik tárgy súlyát jelképezi. Ez azt jelenti, hogy ha belevesszük a tárgyat, akkor az egyel korábbi lépésben egyel kevesebb tárgyal kell foglalkoznunk, és a megengedett súlyunk is csökkent a tárgy súlyával. Ez igaz, hiszen, ha lenne $o \setminus \{ t_i \}$ -nél egy nagyobb értéket adó megoldása R[i - 1, j - w_i]-nek, akkor ehhez hozzávéve t_i-t, *o*-nál jobb megoldást kapnánk. *o*-ról azonban azt mondtuk, hogy optimális, ez így ellentmondást jelentene.
3. lépés: meg kell határoznunk az összefüggést a részfeladatok megoldása és az eredeti feladat megoldása között. Ehhez vegyünk fel egy táblázatot, melynek

legyen a neve „c”, ez fogja tartalmazni az összértékeket. A sorok a felhasznált tárgyaink, az oszlop pedig a súly, amit elbír a hátizsák. Ha nincs tárgyunk, amit bele tudnánk rakni a zsákba, vagy 0 az az érték, amit elbír a zsák, akkor nyilván ott a táblázat értéke 0 lesz. Ha sem az i sem a j nem 0, akkor két lehetséges eset fordulhat elő: a tárgy súlya kisebb vagy nagyobb annál, mint ami belefér a zsákba. Utóbbi esetben nyilván nem tudjuk belerakni, így c[i, j] = c[i - 1, j] lesz az érték, azaz egyel feletti sor értékét másoljuk le. Ha viszont belefér a zsákba, akkor további két eset lehetséges: meg kell vizsgálnunk, hogy azáltal járunk jobban, ha belevesszük, vagy azáltal, ha nem. Amennyiben nem vesszük bele, akkor az érték c[i - 1, j] mint korábban. Viszont ha belevesszük, akkor a tárgy értékéhez hozzá kell adni a súlyával korábban levő oszlopból az értéket ami v_i + c[i - 1, j - w_i]-t adja. Ezek közül a nagyobb fogja megadni a c[i, j] értékét. [11]

A feladat megoldása ezután úgy adódik, hogy a táblázat sorain haladunk végig balról-jobbra majd fentről-lefele amíg a végeredmény a jobb alsó sarokban nem lesz. Végül akár egy külön mátrixba feljegyezhetjük azt is, hogy a tárgy beválasztása vagy kihagyása vezetett az optimális megoldáshoz. Ez alapján lekövethető, hogy mikor melyik tárgyat választottuk.

A dinamikus programozás időigénye így pszeudo-polinomiális. Ez azt jelenti, hogy a futás idő és a memória hatékonysága nem csak a bemenet számától, hanem azok értékétől is függ, mint például a hátizsák kapacitása. Esetünkben a cellák kiszámolásának költség *O(1)*, és az algoritmus futási idejének költsége a táblázat bejárása, ami *O(nW)* – ahol *n* a tárgyak számát, *W* pedig a hátizsák kapacitását jelöli. [6]

Most nézzük meg egy példán keresztül, hogyan is történik a táblázat kitöltésével a megoldás menete: a hátizsákunk kapacitása 10, a tárgyak pedig az 1. táblázatban szereplő érték és súly párral rendelkeznek. [12]

tárgyak	1.	2.	3.	4.
v - érték	10	40	30	50
w - súly	5	4	6	3

1. táblázat: A tárgyak értéke és súlya a bemutatandó példához

Első sorban elkészítjük a táblázatunkat: 0-tól számozzuk jobbról-balra a hátizsák kapacitásáig a számokat (j index), fentről-lefele 0-tól addig ahány tárgyunk van (i index). Kezdetből kinullázzuk a 0. sort és oszlopot, ugyanis nincs olyan tárgy, ami beleférne 0 kapacitásba. Az oszlopokban fogjuk figyelni, hogy az adott részprobléma esetében mekkora a megengedett kapacitás. A táblázat (továbbiakban c) kitöltését a c[1,1] cellától kezdjük. Ekkor az 1. sorban az 1. tárgyat nézzük, aminek 5 a súlya. Amikor az adott tárgyunk súlya kisebb, mint a hátizsák kapacitása, csak másoljuk a felette levő oszlopot: 1. tárgy esetében ameddig j=4 (4<5). Ha a tárgyunk súlya <= mint a kapacitás (5<=5) akkor maximum értéket választunk: c[1,5] cella esetén c[0,5](aktuális cella felett levő cella) vagy az első tárgy értékét(10) hozzáadjuk a felette levő sor hátizsák aktuális kapacitás és tárgy súlya különbsége oszlopban levő (jelenlegi esetben c[0,0]) cellához. Amennyiben optimális megoldást találtunk az adott helyzetben(oszlopban), a könnyebb felismerés véget a cellát egy T betűvel is ellátjuk. A második sorban a 1. és 2. tárgy közül választhatunk. Hasonló módon amíg el nem érjük iterálásban a j=3 értéket addig 0-kat írunk be a c[2,0] és c[2,3] közötti cellákba. A c[2,4] és c[2,8] között mivel a felsőbb sor adott celláiban az érték mindig 0, így összegként a tárgy értékével töltjük fel a cellákat, mivel az igy kapott összeg nagyobb mint a fentebb levő cella összege. A c[2,9] és c[2,10] esetén jön el, hogy eltolt cella értéke és a tárgy összege (10+40) nagyobb mint a felette levő cella értéke (10). Mivel a 0-án kívüli értékek esetén mindegyik oszlopban optimális értékhez jutottunk, odatesszük a T jelet, amit majd a visszakövetés során fogunk használni. Az eddigieket folytatva kitöltjük a táblázatot, és a jobb sarokban megkapjuk, hogy az adott esetben 90 a legnagyobb érték, amit el tudtunk érni.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10 T	10 T	10 T	10 T	10 T	10 T
2	0	0	0	0	40 T	40 T	40 T	40 T	40 T	50 T	50 T
3	0	0	0	0	40	40	40	40	40	50	70 T
4	0	0	0	50 T	50 T	50 T	50 T	90 T	90 T	90 T	90 T

2. táblázat: A bemutatott példán végigvezetett táblázat eredménye

A példában említett táblázat végeredménye a 2. táblázaton látható. A visszafejtés során a jobb sarokból indulunk, az útvonalat a piros négyzet jelöli, és amennyiben kiválasztottunk egy tárgyat azt a sárga színezett rész mutatja. A jobb sarokban (c[4,10]) mivel szerepel a T a cellában, a 4-es tárgyat kiválasztottuk. Megnézzük mennyi a súlya, és azt kivonjuk a hátizsák súlyából, majd a fentebb levő sorban az így kapott cellához navigálunk (10-3 => c[3,7] cella). Abban az esetben, ha az így elért cellában nem látunk T betűt, addig megyünk fentebb az oszlopban, amíg elérjük a 0. sort, vagy nem találunk egy olyan oszlopot, amiben van. Mivel a 2. sorban ezt megtaláltuk, így ezt a tárgyat is kiválasztottuk, majd súlyt kivonva a (7-4) c[1,3] cellához jutunk, de mivel ez nem volt optimális így fentebb lépünk. Azonban elértünk a 0. sorhoz így a visszafejtés véget ért. A fenti példában tehát a 2. és 4. tárgyakat választottuk ki, a maximum érték ezáltal 90 lett, a 10 súllyal rendelkező hátizsákba pedig így 7 súly került.

A megoldási eljárás implementációját a 4. ábrán is láthatjuk. A megoldás menete a következőképpen alakult:

1. lépés: a táblának megfeleltetünk egy 2 dimenziós mátrixot, amit 0 értékekkel töltünk fel. A mátrix sorai fogják reprezentálni az adott i-edik tárgy értékét, oszlopai pedig a hátizsák maximális súlyát.
2. lépés: elkezdjük a tábla kitöltését lentről-felfele, balról-jobbra típusú építkezéssel. Minden i-edik tárgyra és j-edik maximális kapacitásra kiszámoljuk a legnagyobb megszerezhető értéket azáltal, hogy beletesszük a tárgyat vagy nem. Mivel az első sorban és oszlopban nincsen olyan tárgy amire 0 lenne a kapacitás emiatt, ha itt vagyunk marad 0 a cella értéke. Ha a tárgyunk súlya kisebb vagy egyenlő az addig megszerezhető maximum súlynál, két érték közül vesszük a jobbat:
   - az előző tárgynál kapott maximum érték es elérhető súly.
   - a jelenlegi tárgy értéke + az előző maximális értéke és a megmaradó súly miután hozzáadtuk az aktuálisat. Ha ennek súlya nagyobb, mint a maximális kapacitása a hátizsáknak, akkor a maximum súlya a tárgynak és az eddig megszerezhető érték meg fog egyezni az előző tárgynál is megszerzettel.
3. lépés: a feladat optimális összértéke a táblázat jobb sarkából, a c[n][W] cellából szerezhető meg, ahol *n* a tárgyak száma *W* pedig a maximum súlya a hátizsáknak.

4. lépés: a visszakövetést, hogy megtudjuk melyik tárgyak kerültek bele az optimális megoldásban a c mátrix jobb sarkából kezdjük. Amennyiben az i-edik tárgy j-edik súlynál vett értéke különbözik az előző tárgyétól akkor a tárgy hozzá lett adva, igy hozzávesszük a kiválasztott elemek listájához. [6], [13]

```
def knapsackDP(capacity, w, v, n):
    c = [[0 for i in range(0, capacity + 1)] for j in range(0, n + 1)]
    for i in range(0, n + 1):
        for j in range(0, capacity + 1):
            if i == 0 or j == 0:
                c[i][j] = 0
            elif w[i-1] <= j:
                c[i][j] = max(c[i-1][j], v[i-1] + c[i-1][j - w[i-1]])
            else:
                c[i][j] = c[i-1][j]
    i, j, maxWeight, selectedItems = n, capacity, 0, []
    while i > 0 and j > 0:
        if c[i][j] != c[i-1][j]:
            selectedItems.append(i)
            maxWeight += w[i-1]
            j -= w[i-1]
        i -= 1
    return c[n][capacity], maxWeight, selectedItems
```

4. ábra: A dinamikus programozás algoritmusának implementálása

Összegzésképpen a kapott dinamikus programozáson alapuló megoldásunkkal a korábban kiszámított értékeket egy táblában jegyeztük fel. A mátrixunkat alulról-felfelé haladva töltöttük ki, ahol a cellák csak az előző sor és/vagy oszlop értékeitől függtek. A cellák mindig azt a maximális értéket tartalmazták, amely az adott elemek egy részhalmazával volt elérhető a megadott súlyhatárig. A végeredmény kiszámításánál ez lehetővé tette, hogy elkerüljük ugyanazon részfeladatok többszörös kiszámolását, ezáltal hatékonyabb, mint a szimpla rekurzív megközelítés. Amennyiben csak az elérhető maximális értékre lenne szükségünk, az algoritmust tovább lehetne optimalizálni helyfoglalás szempontjából. Ez elérhető azáltal, ha figyelembe vesszük, hogy a számítások során mindig csak az előző sorra volt szükségünk. Tehát a 2D tömb helyett elegendő 1D tömbben eltárolni az előző sor eredményeit, amivel elérhető a *O(W)* helyfoglalás is. [6], [15]

2.5. Mohó algoritmusok (Greedy)

A mohó stratégia egy olyan heurisztika, amelyet optimalizációs problémák megoldására használnak. A megoldás menete során mindig a lokális legjobb döntés kiválasztásával törekszik elérni a globális maximumot, azaz a lehető legjobb megoldást. Az alap mohó algoritmus a hátizsák probléma esetében sajnos nem garantálja az optimális megoldás megtalálását. Habár bizonyos esetekben elérhető vele, amennyiben nem is azt adná akkor is egy optimálishoz közelit fogunk kapni. Esetünkben ez a fajta megközelítés talán a legegyszerűbb, ugyanis a probléma visszavezethető egy rendezésre majd összegzésre. Az elv az, hogy határozzunk meg egy sorrendet, ami lehet akár súly alapján növekvő, érték vagy az érték és súly arányának alapján csökkenően. Pozitív érv a mohó stratégia használata mellett, hogy a futásideje nagyon jó, azonban leginkább akkor lehet hasznos amikor nagy mértékben különböznek a kiszámított arányok. [14]

Az implementáció során az 5. ábrán az érték és súly arányának alapján határoztuk meg az adott sorrendet, mivel általános esetben ez tűnt a legjobb megoldásnak. Ezt úgy kaptuk meg, hogy elosztottuk az adott tárgy értékét a súlyával. Súly alapján akkor érdemes sorba rendezni, ha a célunk az, hogy minél több tárgyat bepakoljunk. Ilyenkor általában a tárgyak értéke relatív közeli skálán mozog, a súlyok viszont nagy mértékben különböznek. Érték alapján akkor érdemes sorba rendezni amikor több nagy értékű tárgyaink vannak, amelyeknek viszonylag alacsonyabb a súlyuk. Ezek elsőbbségével jobb eredmény érhető el azokban a helyzetekben amikor arányok alapján például hátrább állna a sorban az adott tárgy.

A sorrendben helyezést követően végig iterálunk a kapott a tárgyakon és ha az adott tárgy belefér a hátizsákba akkor beletesszük. Egy tárgy behelyezése során frissítjük az összegyűjtött súly és értéket, valamint a már bepakolt tárgyak sorszámát. Az így kapott algoritmus mohó, mivel mindig a soron következő legjobb aránnyal rendelkező tárgyat választja, a választás során pedig nem vesszük figyelembe az adott döntés hatását a jövőre kihatólat. Futás idő szempontjából a rendezés miatt *O(n * log n)* komplexitású költséget értünk el, ahol *n* a tárgyak száma. [2]

```python
def knapsackGreedy(capacity, weights, values, n):
    maxValue, maxWeight, selectedItems = 0, 0, []
    p = [(values[i] / weights[i], i) for i in range(n)]
    p.sort(reverse=True)
    for _, i in p:
        if maxWeight + weights[i] <= capacity:
            selectedItems.append(i+1)
            maxValue  += values[i]
            maxWeight += weights[i]
    return maxValue, maxWeight, selectedItems
```

5. ábra: A mohó módszer algoritmusának implementálása

A bemutatott algoritmusokhoz képest az eddigi összes megkapta az optimális megoldást. Azokat az eseteket leszámítjuk, amikor például brute force esetében olyan méretű inputot adunk, hogy a megoldást nem vagyunk képesek kivárni. A robusztussági tényezőtől eltekintve tehát normális keretek között célt értek. Ilyen szempontból ez az első, amely nem garantálja az optimális megoldást. A biztosság helyett azonban a memória igény és a futás idő az eddigi legkedvezőbb. A gyakorlatban rengeteg olyan helyzet adódhat, amikor ez a két faktor előnyt jelenthet az optimális megoldással szemben. Élő példa, amikor a hátizsákunk kapacitása relatív nagy, a tárgyak súlyai között pedig akár 10000-es nagyságrendű különbségek vannak. Ilyen esetben lehet, hogy a gyorsaság fontosabb szempont, mint kivárni a dinamikus programozás által megkapható optimális megoldását.

Ahogy korábbiakban már említettük, az algoritmus nem garantálja az optimális eredményt, de képes elérni azt. Fontos azt is megjegyezni, hogy kevesebb tárgy esetében az algoritmus nagyobb valószínűséggel kapja meg az optimális eredményt, mivel kevesebb lehetősége van a hibára. Egy másik dolog, hogy lehetséges, hogy az adott problémára az adott optimális megoldás (maximális összérték) több kombináció által is kijöhet. A következőkben tesztelés során olyan speciális eseteket sorolnánk fel, ahol az algoritmus a dinamikus programozással azonos összértéket ért el, azonban jóval kedvezőbb idő alatt:

- Magasabb értékű tárgyaknak alacsonyabb a súlya, kevesebb értékűeknek nagyobb például {(w:1, v:10), (w:2, v:9), …}
- A tárgyak súlya konstans és megegyeznek, nincs két ugyan olyan tárgy.
- A tárgyak értéke konstans és megegyeznek, nincs két ugyan olyan tárgy. Ebben a specifikus példában az elért összértékek megegyeznek, azonban a mohó

algoritmus jobb megoldást fog adni a súlyra. Azért van ez mert a dinamikus programozás nem veszi figyelembe a súlyt, a mohó pedig a kisebb súlyúakat veszi előbbre. Ezért az olyan esetekben, ahol a kisebb összsúly értéket preferáljuk rövidebb idő alatt a mohó algoritmus célra vezetőbb lehet.

- Egyértelmű eset: a megoldás tényleg a legjobb érték/súly alapján dől el.

A hátizsák probléma esetében a mohó heurisztikát alkalmazó algoritmust számos módon fel lehet használni egyéb más stratégiák javítására. Egy módszer, ha a mohó algoritmussal egy gyors megoldást kapunk, amit kezdő pontként fel lehetne használni. Ezt különböző metaheurisztikus algoritmusok, mint a genetikus-evolúciós algoritmusok megpróbálhatják tovább fejleszteni. További érv mellette, hogy a keresési tér jelentősen csökkenhető általa, mivel már egy jó megoldás körül keresgélünk. Erre példa lehet a lokális kereső algoritmusok alkalmazása, mint a tabu keresés[3] vagy szimulált hűtés. Keresési tér csökkentésénél megemlíthető az inputok előfeldolgozása is. Így a jelentéktelen értékkel vagy túl nagy súllyal rendelkező tárgyakat ki lehetne szűrni. Ehhez hasonló módszerekkel egyensúlyozhatunk a futás idő és a megoldás minősége között. [15]

2.6. Visszalépés (Backtracking)

A visszalépés módszere egy mélységi keresésen alapuló kimerítő keresés. Hasonló kimerítő kereséseket láthattunk már a bemutatott brute force és oszd meg és uralkodj algoritmusoknál. Nevét arról kapta, ahogy a döntések során lépésenként haladunk előre, amikor is egy adott ponton nyilvánvalóvá válhat, hogy az adott döntés rossz irányba halad. Ekkor visszalépünk egy megadott számú lépést és onnan más irányba folytatjuk tovább. Ez a hozzáállás a mélységi keresésen alapszik, ugyanis a keresési térben egy után olyan mélyre megyünk amennyire csak lehet és csak utána fedezzük fel a többi lehetséges esetet. A lépések során meghozandó döntéssel már találkoztunk: behelyezzük az adott tárgyat a hátizsákba vagy nem. Az oszd meg és uralkodj módszerhez hasonlóan rekurzív hívásokkal fogjuk a tárgyak ellenőrzését elvégezni. Az

[3] Az ötlet más dimenzió mentén is alkalmazható, lásd például C. García-Martínez, J. Rodriguez, M. Lozano: „Tabu-enhanced iterated greedy algorithm: A case study in the quadratic multiple knapsack problem” c. kutatását

elemeket addig próbáljuk meg elhelyezni amíg lehetséges. Sikertelenség esetén pedig visszalépdelünk és eltávolítjuk az adott tárgyat és leellenőrizzük a többi lehetőséget. [16], [17]

Első sorban a mohó stratégiából kiindulva és a visszalépés elvét hozzáadva egy újfajta megoldás menetet kaphatunk (6. ábra). A mohó módszer algoritmusát használjuk fel, azonban, ha egy tárgy már nem fér bele a hátizsákba, 1 lépéssel visszalépünk és eltávolítjuk az előzőleg hozzávett elemet. Ezután megpróbáljuk a soron következő tárgyat bele pakolni. Ez a megközelítés néhány speciális esetben hasznos lehet. Ilyen például amikor egy tárgynak jobb az érték/súly aránya, azonba a következő kettő együttesen kevesebb súllyal sokkal jobb értéket hoznak az elérhető összeghez. Habár csak 1-1 lépéssel lépegetünk visszafele még nem garantálja az optimális megoldást, némely esetben azonban jobb eredményt hozhat, mint a szimpla mohó algoritmus.

```python
def knapsackBacktrackGreedy(capacity, weights, values, n):
    maxValue, maxWeight, selectedItems = 0, 0, []
    p = [(values[i] / weights[i], i) for i in range(0, n)]
    p.sort(reverse=True)
    for _, i in p:
        if maxWeight + weights[i] <= capacity:
            selectedItems.append(i + 1)
            maxValue  += values[i]
            maxWeight += weights[i]
        elif len(selectedItems) > 0:
            index = selectedItems.pop()
            maxValue  -= values[index - 1]
            maxWeight -= weights[index - 1]
    return maxValue, maxWeight, selectedItems
```

6. ábra: A mohó módszer visszalépéses módszerrel való bővítése

Most nézzük meg az eredeti optimális megoldással szolgáló visszalépéses algoritmust (7. ábra). Kezdetben inicializáljuk az értékeket, amikben az optimális megoldást szeretnénk megtartani. A backtrackHelper belső függvény által végezzük a rekurziót és az összes kombináció felfedezését. Úgynevezett *nonlocal* változókkal érjük el, hogy ez a beágyazott függvény elérje a külső függvényben használt változókat értékfrissítés szempontjából. Ha az adott tárgy behelyezésével túllépjük a kapacitást akkor visszalépünk mert nincs miért tovább keresgélni azon az úton. Amennyiben eddiginél jobb összeget kaptunk a jelenlegi tárgy behelyezésével frissítjük a külső

változók értékét. Az eddig bepakolt elemek listáját azért szükséges másolnunk, mivel a listák változhatnak, hozzáadunk és eltávolítunk belőlük elemeket. Amennyiben nem másolnánk az utolsó elem eltávolításakor referencia miatt eltávolítanánk a benne levő utolsó tárgyat, amit nem szeretnénk. A rekurziónk megállási feltétele amikor elértük az utolsó tárgyunkat, ezzel minden elemet számításba vettünk. [18], [19]

Oszd meg és uralkodj módszerhez hasonlóan ellenőrizzük egy tárgy behelyezése során kapott két lehetőséget. Hasonlóan először azt az esetet vizsgáljuk, amikor hozzávesszük az aktuális tárgyat. Ezt követően a rekurzív hívással megnézzük a hátralevő tárgyakra vett esetet. Amennyiben túl lépjük a súlykorlátot, eltávolítjuk az utolsó elemet és a visszalépést alkalmazzuk. Ezáltal amikor hozzáadunk egy tárgyat a fában a gyerek csomópontra lépünk, eltávolításkor vissza a szülőre. Miután minden lehetőséget felfedeztünk az aktuális tárgyból a rekurzív hívás után a tárgy eltávolítása következik. Így biztosítjuk az összes lehetséges kombináció felfedezését, amiben a tárgy nem szerepel. Másodjára leellenőrizzük azokat az eseteket, amikor a tárgyat nem vettük hozzá a beválasztott elemek közé. Az algoritmus a 0-adik tárgytól, vagyis a gyökértől indul, a mélységi keresés logikáját követve.

```python
def knapsackBacktrack(capacity, weights, values, n):
    maxValue, maxWeight, selectedItems = -1, -1, []

    def backtrackHelper(i, actualWeight, actualValue, actualSelectedItems):
        nonlocal maxValue, maxWeight, selectedItems
        if actualWeight > capacity:
            return
        if actualValue > maxValue:
            maxValue, maxWeight, selectedItems = actualValue, actualWeight, actualSelectedItems.copy()
        if i == n:
            return

        actualSelectedItems.append(i + 1)
        backtrackHelper(i + 1, actualWeight + weights[i], actualValue + values[i], actualSelectedItems)
        actualSelectedItems.pop()

        backtrackHelper(i + 1, actualWeight            , actualValue            , actualSelectedItems)

    backtrackHelper(0, 0, 0, [])
    return maxValue, maxWeight, selectedItems
```

7. ábra: A visszalépéses módszer algoritmusának implementálása

Ez által egy újfajta megoldás menetet láthattunk a hátizsák problémára. Hatékonyság szempontjából a keresési tér maradt az összes kombináció, csak a bejárás sorrendje változott. Futás ideje és memória igénye így megegyező az oszd meg és uralkodj módszerből már ismertekkel. Ebből is következik, hogy az optimális megoldás megtalálása is garantált, ha az idő nem kényszeríti a korai leállást. Javítási

lehetőségként alkalmazhatóak heurisztikák, amik által egy keresési határt vagy korlátot tudunk szabni. Így csökkenthető a keresési tér és korábban eldönthető lenne mikor kell visszalépni, amivel nem tennénk fölösleges lépéseket. A következőkben egy ilyen módszert vizsgálunk meg, ahol ezt a keresési teret megpróbáljuk korlátozni bizonyos határok által.

2.7. Elágazás és korlátozás (Branch and bound)

A branch and bound megközelítés egy széleskörűen ismert eljárás az egész értékű feladatok megoldására. Az egész értékű programozási feladatok azon optimalizálási problémák kategóriájába tartoznak, ahol elvárjuk, hogy a változok részben vagy teljesen egész értéket vegyenek fel. A hátizsák probléma esetében ez egyértelmű, hiszen a döntéseket a már megbeszéltek alapján binárisan is ábrázolni tudjuk. A branch and bound stratégia széles körben képes kezelni ezen típusú feladatokat és optimális megoldást ad rájuk. Számításügyileg itt is exponenciális futás időt érünk majd el, így nagy méretű problémák során legrosszabb esetben az eredmény várattathat magára. [2], [15]

Az optimális megoldás megtalálására törekszik azáltal, hogy egy keresőfát épít fel, ahol a gyökér fogja reprezentálni az elindulási kezdetleges pontot. A fa csomópontjai az addigi részmegoldásokat fogják tartalmazni. Az ágak azon döntéseket mutatják, hogy beválasztjuk a következő tárgyat vagy nem, ami által eljutunk a következő részmegoldásig. A keresés során mindig számontartjuk az elért legjobb megoldást, ami alapján majd metsszük a fát. Ehhez egy felső korlátot határozunk meg majd az összes csomópontot eliminájuk, ami nem tud jobb megoldást hozni az eddig elértnél. Ezt a korlátot általában az eredeti probléma lazításával határozzuk meg azáltal, hogy néhány együtthatót lazítunk vagy eltávolítunk. A lehetséges megoldások megszerzése érdekében így a probléma egy könnyített verzióját kapjuk meg. A branch and bound módszer esetében az együtthatók lazítása általában a felső határ kiszámítására utalnak, ami segít elvetni azokat az ágakat, amik nem vezetnek az elvárt eredményhez. [20]

A megközelítés az oszd meg és uralkodj illet mohó stratégiákat részben magába foglalja. Mohó mivel mindig az aktuális legjobb döntést vesszük figyelembe, illetve a kiindulási pontot és bejárási sorrendet is az érték és súly aránya alapján határozzuk

meg. Oszd meg és uralkodjnál használt megközelítés a probléma két részproblémára bontásában látszik meg, ugyanis ezek lesznek az ágaink. Mindegyik csomópontra minden lépésben az algoritmus kiterjeszt 2 lehetséges megoldást: kiválasztjuk-e a tárgyat vagy nem. Lépésenként egyszerre egy döntést választunk, majd ez alapján valamilyen módon meghatározzuk az adott döntés lehetséges kimenetelét. Ezeket az eredményeket használjuk fel majd az újabb részproblémák létrehozására, amelyeket hasonló módon oldunk meg.

A keresési folyamat során a csomópontokban tartottuk számon az addig megtalált legjobb megoldást és a döntés által hozott hasznot, amit felhasználunk a fa metszéséhez. Ezt azáltal tesszük, hogy minden csomópontból megbecsüljük az elérhető legnagyobb megszerezhető értéket. Megpróbáljuk addig hozzáadni a tárgyak súlyát és értékét a már kiválasztottakhoz a számontartott szinttől amíg a soron következő tárgy súlya még kisebb, mint a fentmaradó kapacitás. Miután már nem fér bele teljesen, az értékének egy részét fogjuk hozzávenni az alapján, hogy mennyi tárgyat sikerült hozzávennünk. Ez számszerűen úgy néz ki, hogy a haszon-korláthoz hozzávesszük a fennmaradó kapacitás és az adott tárgy érték-súly arányának szorzatát. Így egy elég jó becslést kapunk arra, hogy a megmaradó csomópontokkal mekkora maximális érték szerezhető meg. Ez a fajta korlát meghatározás a részleges hátizsák problémára alapszik, ahol megengedett, hogy súlyok egy részét is beletegyünk a hátizsákba. [2]

A megoldás teret tehát ezáltal próbáljuk csökkenteni, hogyha az adott haszon, amit a becslés eredményezett kisebb, mint a jelenlegi legjobb elérhető korlát, akkor elmetsszük a fa adott ágát, amit már nem terjesztünk ki. Ezzel lényegében minden olyan csomópontot levágunk vagy korlátozunk, amely nem tud jobb megoldással szolgálni, mint az addigi legjobb. A kiterjesztési, illetve döntési sorrendet mindig prioritás alapon választjuk, ahol a kedvezőbb eredménnyel kecsegtető kerül előtérbe. Minél hatékonyabb módon tudjuk megbecsülni az ágak felső határait, azáltal az algoritmus hatékonyságát is vele együtt tudjuk növelni. [21]

```python
class Node:
    def __init__(self, capacity, weight, value, level, selectedItems):
        self.capacity      = capacity
        self.weight        = weight
        self.value         = value
        self.level         = level
        self.selectedItems = selectedItems
        self.bound         = 0

    def __lt__(self, other):
        return self.bound > other.bound

    def include(self, items):
        weight        = self.weight        + items[1]
        value         = self.value         + items[2]
        level         = self.level         + 1
        selectedItems = self.selectedItems + [items[3] + 1]
        return Node(self.capacity, weight, value, level, selectedItems)

    def exclude(self):
        return Node(self.capacity, self.weight, self.value, self.level + 1, self.selectedItems)
```

8. ábra: A csomópontok reprezentálására szánt Node osztály

Az ötlet implementáláshoz szükségünk van a csomópontok reprezentálása érdekében egy osztályra, ami a 8. ábrán is látható, ez által képesek leszünk az addigi megoldásokat számontartani. Minden csomópontnál feljegyezzük a már kiválasztott tárgyakat, az általuk megszerzett értéket és súlyt, illetve a fentmaradó kapacitást. A szint fogja mutatni, hogy melyik a soron következő tárgy, amit ki fogunk terjeszteni, illetve milyen mélyen járunk a fában. Mivel mindig a legígéretesebb korláttal rendelkezőt szeretnénk kiterjeszteni, egy prioritásos sort használunk. Ehhez képesnek kell lennünk eldönteni, hogy az adott csomópontok mi alapján legyenek sorba rendezve. Erre fogjuk használni a már meghatározott bound-ot, amiben mindig az adott csomópontokból számított elérhető érték felső becslését fogja jelezni. [22], [23], [24]

Amikor egy tárgyat kiválasztunk akkor mindig növelni kell a megszerzett összegeket a tárgy által nyújtott értékkel és súllyal. Ehhez hozzá kell még vennünk a már kiválasztottak listájához és ezzel együtt növelni az adott döntési szintet a fában, majd visszaadjuk az így kapott állapotot. Amikor a tárgyat nem választjuk ki elég csak az szintet növelni, ugyanis ezzel szeretnénk elérni, hogy egy gyerek csomópontot hozzunk létre.

A megszerezhető érték becslése fogja nekünk jelezni a potenciálisan elérhető felső határt, amit a fában az adott út mentén szerezhetünk. Ehhez ahogy a 9. ábrán is látható, először ellenőrizzük, hogy addig a csomópontig bezárólag nem léptük-e túl a

kapacitást. Ebben az esetben ugyanis eldobjuk a jelenlegi csomópontot és egy másik pontról folytatjuk a keresést. A felső becsülendő határ kiindulási pontja az addig megszerzett érték, súly hasonlóan. Mindig az adott szinttől kezdjük az ellenőrzést a még hátramaradó utolsó tárgyig. Ha a tárgy súlya teljes egészében belefér a kapacitásba akkor beletesszük ezáltal növelve a súlyt és a becsült értéket a tárgy értékével. Amennyiben egy tárgy már nem fér bele a fentebb megbeszélt módon az érték-súly arány és a fentmaradó kapacitás szorzata alapján növeljük a becsült értéket.

```
def calculateBound(self, items):
    if self.weight >= self.capacity:
        return 0
    bound, totalWeight, i = self.value, self.weight, self.level
    while i < len(items) and (totalWeight + items[i][1]) <= self.capacity:
        totalWeight += items[i][1]
        bound       += items[i][2]
        i += 1
    if i < len(items):
        bound += (self.capacity - totalWeight) * (items[i][2] / items[i][1])
    return bound
```

9. ábra: A legnagyobb megszerezhető érték megbecslése adott csomópontból

A fő algoritmust a 10. ábrán láthatjuk. A mohó módszerből indulunk ki, sorrendbe helyezzük a tárgyakat az érték-súly arányuk alapján, megjegyezve a kezdeti pozíciókat. Létrehozzuk a prioritásos sort kezdetben a legjobb arányú tárgyal inicializálva, aminek a felső szerezhető határát is kiszámoljuk. A tárgyakat alap esetben a már mohó módszer által meghatározott sorrendben fogjuk megvizsgálni. Az adott vizsgálandó csomópontra megnézzük a becsült értéket. Ha ez a jelenlegi legjobb értéknél jobbnak ígérkezik akkor hozzáadjuk kiterjesztésre. Az algoritmus addig tart, amíg az összes kiterjesztett csomópontot meg nem vizsgáltuk. A sorban mindig a legígéretesebbnek tűnőt nézzük meg először. A már ismert módon először is azt nézzük meg, az aktuális tárgy kiválasztása mit eredményez. Ha ez jobb, mint az eddigi legjobb akkor azt választjuk ki az új megoldásnak. Ezt követően megnézzük azt az esetet is, hogy nézne ki a megoldás, ha a tárgyat nem választanánk ki. Mindkét esetben megbecsüljük a maximálisan nyerhető értéket és ha azok jobbak, mint a jelenlegi akkor betesszük őket a sorba és majd következőkben amint rájuk kerül a sor kiterjesztjük azokat is. [22], [23], [24]

```python
def knapsackBranchAndBound(capacity, weights, values, n):
    items = sorted([(values[i] / weights[i], weights[i], values[i], i) for i in range(0, n)], reverse=True)
    queue = PriorityQueue()
    root = Node(capacity, 0, 0, 0, [])
    root.bound = root.calculateBound(items)
    queue.put(root)
    maxValue, maxWeight, selectedItems = 0, 0, []

    while not queue.empty():
        node = queue.get()
        if node.bound > maxValue and node.level < n:
            level = node.level

            includeNode = node.include(items[level])
            if includeNode.weight <= capacity:
                if includeNode.value > maxValue:
                    maxValue, maxWeight, selectedItems = includeNode.value, includeNode.weight, includeNode.selectedItems
                includeNode.bound = includeNode.calculateBound(items)
                if includeNode.bound > maxValue:
                    queue.put(includeNode)

            excludeNode = node.exclude()
            excludeNode.bound = excludeNode.calculateBound(items)
            if excludeNode.bound > maxValue:
                queue.put(excludeNode)
    return maxValue, maxWeight, selectedItems
```

10. ábra: A branch and bound fő algoritmusának implementálása

A branch and bound algoritmus során elérhető legrosszabb futás idő megegyezik az oszd meg és uralkodj algoritmussal, ami $O(2^n)$. A teljesítménye gyakorlatban ennél sokkal jobb, ugyanis nagy mértékben függ attól, hogy az adott csomópont választási stratégiánk mennyire jó a becslés alapján. A memória használat szempontjából mivel prioritásos sort használunk legrosszabb esetben ez is $O(2^n)$. Ezen értékek javíthatóak, ha a csomópontokat hatékonyan tudjuk szelektálni, esetleg a sor maximális méretére is adhatunk korlátokat. [21]

```python
def knapsackBranchAndBound(capacity, weights, values, n, nodePercentage=100):
# ...
    processedNodes, maxNodes = 0, (2**n) * nodePercentage // 100

    while not queue.empty() and processedNodes < maxNodes:
        node = queue.get()
        processedNodes += 1
# ...

def knapsackBranchAndBound(capacity, weights, values, n):
# ...
    while not queue.empty():
        node = queue.get()
        if node.bound < maxValue:
            break
# ...
```

11. ábra: A branch and bound algoritmusának optimalizálási kísérletei

Egyéb ötletként megpróbálhatunk a futás idő és az elért eredmény jósága között játszani. Ahogy 11. ábrán is látható, egy százalékos határt szabhatunk hány csomópontot szeretnénk kiterjesztésre megengedni, amit majd maximálisan megvizsgálhatunk (1. módszer). Ez alapértelmezetten (100%) megegyezik az eredeti

módszerrel, ahol a legrosszabb bekövetkezhető eset 2^n. Más ötletként, ha azt vesszük észre, hogy a sor tetején felső becslés már kisebb, mint az eddig megszerzett legjobb összérték, akár idő elött is terminálhatunk (2. módszer). A két módszert külön-külön vagy akár egyszerre alkalmazva (3. módszer) a 3. táblázaton szereplő eredményeket érhetjük el. [25], [26]

nodePercentage	70	60	40	Opt. %	30	Opt. %	10	Opt. %
Alap módszer	1.516	1.230	1.519		1.478		1.679	
1. módszer	1.309	0.859	1.039	99.83%	1.333	99.65%	1.199	90.35%
2. módszer	1.228	0.770	0.964		0.899		0.939	
3. módszer	0.829	0.579	0.959	99.83%	0.910	99.65%	0.954	90.35%

3. táblázat: Az optimalizálási módszerek által elért eredmények a különböző paraméterek hatására

A táblázat kitöltését 100 tesztesettel végeztük el, amit majd a későbbi részekben is felhasználunk és bővebben kifejtünk. A táblázattal az algoritmus alap 100 teszteset átlagos futásidejének javítását szerettük volna szemléltetni milliszekundumban az optimális eredménytől való eltávolodás mellett. A sárga háttérrel jelzett soroknál (1. és 3. módszer) használtuk a csomópontok kiterjesztésének határát. Az első sorban szereplő számok ennél a két módszernél a számban kifejezett értéket jelölik. Némely ilyen sor mellett a szaggatott vonal jobb oldalán egy % is szerepel. Ezekben az esetekben fordult elő, hogy átlagosan ennyire közelítettük meg az optimális eredményt. Ahol nem szerepel százalék abban az esetekben elértük az optimális megoldást. A sorok 5 futtatás utáni átlagos időt szemléltetik. A táblázatból megállapítható, hogy a módszerek alkalmazásával javítani tudtunk az alap módszer által kapott átlagos futási időn.

2.8. Evolúciós-genetikus algoritmus (Genetic algorithm)

A genetikus algoritmusok olyan optimalizációs algoritmusok amelyeket Darwin természetes kiválasztódás elmélete inspirálta. Az elmélet szerint a környezeti hatásokat a legrátermettebb egyedek élhetik túl, majd azok utódaiknak továbbítják a génkészletüket, a gyenge tulajdonságokkal rendelkező egyedek pedig kihalnak. A genetikus algoritmusnak tehát képesnek kell lennie lemodellezni az adott evolúciós folyamatokat, innen ered a név is. Az ilyen jellegű algoritmusok, habár nem garantálják

az optimális megoldás megtalálását, képesek megtalálni azt, vagy legalább is egy ahhoz közeli jó megoldást. [27]

A genetikus algoritmusok általános megoldási elve, hogy kezdetben inicializáljuk az első generációt vagy populációt, ami egy lehetséges megoldásokat tartalmazó halmaz. A halmazban levő egyedeket kromoszómáknak nevezzük el, amik kezdetben véletlenszerűen kerülnek inicializálásra. Az evolúciós folyamat során ahhoz, hogy ki tudjuk választani az egyedeket minden kromoszómának egy rátermettségi értéket határozunk meg. Ez fogja mutatni, hogy az adott egyed mennyire képes versenybe szállni a többiekkel, azaz minél nagyobb az érték annál jobb a megoldás is. [28]

Ahogy a biológiában is, az erős tulajdonságokkal rendelkező kiválasztásra kerül, míg a gyenge kimarad. A kiválasztási folyamatra számos stratégia létezik melyek választása a problémától és az elvárt céltól függ. Egyes stratégiák jobbak a globális maximum megtalálásában, mint például versenytorna típusú, rang szerinti. Mások lehet, hogy a lokálisat találhatják meg gyorsabban például a rulettkerék. Egyéb helyzetekben például, ha a rátermettségi értékek között nagy eltérések vannak akkor a versenytorna és a rang alapú kiváltképp jól működhetnek. [30]

A kiválasztás után a keresztezés fázisa következik. Miután sikeresen kiválasztásra került a két szülő, génjeiket tovább kell adniuk az utódok létrehozása céljából. Az, hogy a keresztezés folyamata miként zajlik le, szintén számos stratégia létezik. Megemlíthető a tervezett gén eloszlás vagy véletlenszerűen jelöljük ki a géneket a szülőktől. Valamely megegyező pozíción/pozíciókon való elmetszés majd a részeik felcserélése is egy járható út. Az utódok létrejöttével a következő fázisba érve elérkeztünk a mutációs szakaszhoz. [27], [28]

A mutációs folyamat során az egyedeket valamilyen szempont szerint névből adódóan mutációnak tesszük ki, ami bizonyos eséllyel következhet be. A mutációt amiatt használjuk, hogy elkerüljük a korai konvergenciát és a populáció egyhangúságát, fenntartva ezzel a változatosságot. A mutációra is különböző stratégiák léteznek, például megcseréljük a biteket bizonyos véletlenszerű pozíciókon. Egyéb ötlet, hogy véletlenszerű szakaszokat fordított sorrendbe helyezünk, véletlenszerű pozíciókon átbillentjük a bitet 0-ról 1-re és fordítva. [29]

Végső lépés, hogy a mutációs fázis végén kapott egyedeket visszaengedjük a populációba annak érdekében, hogy frissítjük a génállományt és előállítsuk a következő

generációt. Amiatt, hogy a populáció száma változatlan maradjon a régiből ki kell dobnunk bizonyos egyedeket, amelyek meghatározásához a kiválasztás során használt stratégiákat is fel tudjuk használni. Ezt követően az újabb generáció kiértékelésre kerül, így a folyamatot addig ismételjük elölről, amíg az optimális vagy egy jó eredményt nem kapunk. [27], [28]

```python
def generatePopulation(size, capacity, weights, values, n):
    population, fitnessValues = [], []
    for _ in range(0, size):
        acceptable = False
        while not acceptable:
            chromosome = BitArray(length=n)
            for i in range(0, n):
                chromosome[i] = random.choice([0, 1])
            fitness = calculateFitness(capacity, weights, values, chromosome)
            if fitness != 0:
                acceptable = True
                fitnessValues.append(fitness)
        population.append(chromosome)
    return population, fitnessValues
```

12. ábra: A véletlenszerű populáció generálás implementációja

Az implementációt a kezdetleges populáció generálásával kezdjük (12. ábra), ahol mindig a megadott mennyiségű egyedet fogjuk legyártani. A kezdettben 0 rátermettségi értékkel generált érvénytelen kromoszómák a későbbi szakaszokban megzavarhatják az algoritmust. Ezért miután véletlenszerűen generáltunk egy egyedet, kiértékeljük a rátermettségi értékét. Így addig fogjuk ismételni ezt az újra generálási folyamatot, amíg egy 0-tól különböző rátermettséget nem kapunk. A genetikus algoritmusoknak általában a rátermettségi függvénnyel való számolása a legköltségesebb művelete. Hatékonyság szempontjából emiatt eltároljuk a már kiszámolt értékeket. A kromoszómáink hossza a tárgyaink számával fognak megegyezni, így a bitsorozat minden egyes bitje megfeleltethető lesz az adott tárgynak. A pozíciókon levő 1 fogja reprezentálni, hogy a tárgy kiválasztásra került, 0 ellenkezőjét, ezáltal csak a 0 és 1 számok közül kell véletlenül választanunk egyet. [27], [28], [29]

```python
def calculateFitness(capacity, weights, values, chromosome):
    maxValue, maxWeight = 0, 0
    for i in range(0, len(chromosome)):
        if chromosome[i] == 1:
            maxWeight += weights[i]
            maxValue  += values[i]
    if maxWeight > capacity:
        return 0
    else:
        return maxValue
```

13. ábra: A rátermettségi függvény implemetálása

Az adott kromoszóma rátermettségi értékének meghatározására használt függvényünk (13. ábra) végig iterál az adott egyeden. Ha 1-es szerepel a bizonyos helyen levő pozíción akkor azon az indexen levő tárgy súlya és értéke beszámításra kerül. Ha az így szerzett súly nagyobb, mint a hátizsák kapacitása, akkor az nem lehet megoldás, emiatt 0 rátermettségi értéknek vesszük. Amennyiben a súly kisebb és megfelelő, a kapott érték összeget feleltetjük meg neki. [28]

```python
def selectChromosomesRankbased(population, fitnessValues):
    fitnessTuples = list(enumerate(fitnessValues))
    fitnessTuples.sort(key=lambda value: value[1])
    ranks = [i+1 for i in range(0, len(fitnessTuples))]
    sumRanks = sum(ranks)
    probabilities = [rank / sumRanks for rank in ranks]
    oldIndexes = [i for i, _ in fitnessTuples]
    parent1 = random.choices(oldIndexes, weights=probabilities, k=1)[0]
    parent2 = random.choices(oldIndexes, weights=probabilities, k=1)[0]
    return population[parent1], population[parent2]

def selectChromosomesRouletteWheel(population, fitnessValues):
    probabilities = [f / sum(fitnessValues) for f in fitnessValues]
    parent1 = random.choices(population, weights=probabilities, k=1)[0]
    parent2 = random.choices(population, weights=probabilities, k=1)[0]
    return parent1, parent2

def selectChromosomesTournament(population, fitnessValues, tournamentSize=4):
    parent1 = max(random.sample(list(zip(population, fitnessValues)), tournamentSize), key=lambda x: x[1])[0]
    parent2 = max(random.sample(list(zip(population, fitnessValues)), tournamentSize), key=lambda x: x[1])[0]
    return parent1, parent2
```

14. ábra: Kromoszóma kiválasztási stratégiák implementációja

A kromoszómák kiválasztására számos stratégia létezik ahogy már fentebb kifejtettük. Három fajta stratégiát nézünk meg: rang szerinti, rulettkerék és versenytorna (14. ábra). A rulettkerék elvét a [27]-es cikk tökéletesen leírja: „egyedek kiválasztása arányos a rátermettségi értékükkel, azaz minél jobb a rátermettségi értéke az adott egyednek annál nagyobb esélye van a kiválasztódásra, és minél alacsonyabb a rátermettségi érték az esély annál kevesebb.” Nevét is a rulettkerék mechanizmusa alapján kapta: a rátermettebb egyedek nagyobb szeletet kapnak, kevésbé

rátermettebbek ezzel arányosat, de pörgetés során a kerék mégis bármely helyen megállhat. Így tehát összegezzük az rátermettségi értékeket majd mindenki annyi százalék esélyt kap a kiválasztódásra amennyit ér.

A rangszerinti kiválasztás lényege, hogy a nagyobb rátermettségi értékkel rendelkező egyedek magasabb rangot kapjanak, míg alacsonyabbak kevesebbet. Így egy sorrendbe rendezzük az indexekkel a rátermettségi értékek alapján és a legjobb a legmagasabb rangot kapja, a második a második legnagyobbat és így tovább. A tárgyak kiválasztásának esélyét így az elért rang és az összes kiosztott rangpont aránya fogja adni. Végezetül pedig az így kapott indexen levő kromoszómákat választjuk ki. A másik kiválasztási taktika, amit még megvizsgálunk a verseny típusú kiválasztás, nevéből adódóan egy versenyt rendezünk előre megadott fővel. A szereplőket véletlenszerűen választjuk ki a populációból. A verseny nyertese pedig a legjobb rátermettségi értékkel rendelkező egyed lesz. [29], [30]

```python
def crossoverOnePoint(parent1, parent2, crossoverProbability):
    child1, child2 = parent1.copy(), parent2.copy()
    if random.random() < crossoverProbability:
        point1 = random.randint(1, len(parent1) - 2)
        child1 = parent1[:point1] + parent2[point1:]
        child2 = parent2[:point1] + parent1[point1:]
    return child1, child2

def crossoverTwoPoint(parent1, parent2, crossoverProbability):
    child1, child2 = parent1.copy(), parent2.copy()
    if random.random() < crossoverProbability:
        point1 = random.randint(1, len(parent1) - 2)
        point2 = random.randint(1, len(parent1) - 2)
        if point1 > point2:
            point1, point2 = point2, point1
        child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
        child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]
    return child1, child2
```

15. ábra: Keresztezési stratégiák implementációja

A kiválasztás során két szülőt választottunk ki annak érdekében, hogy keresztezni tudjuk őket. A folyamat fontos szerepet játszik a keresési tér felfedezésében, ahol jobb megoldásokat szeretnénk találni, valamint a jó megoldások alapján szeretnénk újakat létrehozni. A keresztezés bekövetkezésének értékével jelentősen befolyásolni tudjuk az eredmény megtalálásának folyamatát. Ha 1, vagyis mindig bekövetkezik akkor magas gén eloszláshoz vezethet, de így akár gyakran megszakíthatja az ígéretes

megoldásokat. Amennyiben az érték alacsony akkor meg ellenkezője történik, nem kapunk sok új megoldást, a keresési tér nem elégséges felfedezésével pedig optimálistól eltérő eredményeket kaphatunk. Maga a keresztezés szempontjából az egy és két pont keresztezést vizsgáljuk meg (implementációt lásd 15. ábra). Ha a folyamat a véletlenszerűség alapján bekövetkezhet, rendre egy vagy két random számot választunk pozícióként, ahol majd el fogjuk metszeni őket. Amennyiben a pozíciót a [0, n-1] tartományból választanánk véletlenszerűen megtörténhet, hogy az első és utolsó pozícióknál kell metszeni. Ezzel lényegében visszakapnánk a szülőket, így maga a keresztezés célja, a gén csere megvalósulása nem történne meg. Ezért a [1, n-2] tartományból választunk értéket. Fontos megjegyezni, hogy ez indexelési hibát okozhat azokban az esetekben, ha kevesebb mint 2 tárgyunk van. [28]

```python
def mutateBitFlip(chromosome, mutationProbability):
    if random.random() < mutationProbability:
        mutationPoint = random.randint(0, len(chromosome) - 1)
        chromosome[mutationPoint] = 1 - chromosome[mutationPoint]
    return chromosome

def mutateBitSwap(chromosome, mutationProbability):
    if random.random() < mutationProbability:
        point1, point2  = random.sample(range(0, len(chromosome)), 2)
        chromosome[point1], chromosome[point2] = chromosome[point2], chromosome[point1]
    return chromosome

def mutateBitAll(chromosome, mutationProbability):
    for i in range(0, len(chromosome)):
        if random.random() < mutationProbability:
            chromosome[i] = 1 - chromosome[i]
    return chromosome
```

16. ábra: Mutációs stratégiák implementálása

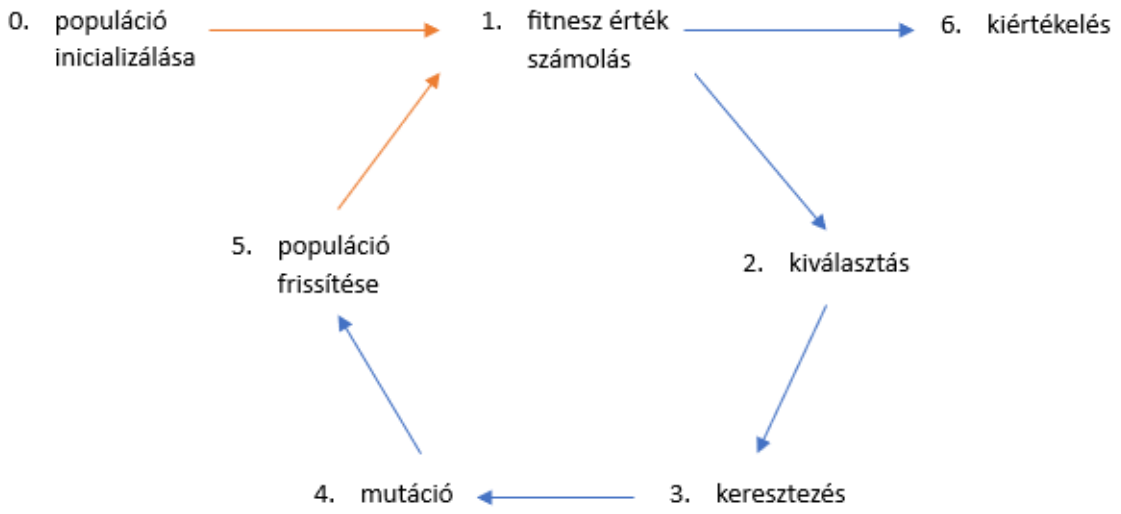
Mutációnak a keresztezés fázis után kapott gyerekeket tesszük alá a populáció változatossága érdekében. A futtatás elején statikusan lefixáljuk az adott mutációs folyamat bekövetkezésének esélyét. Ezt a valószínűséget általában alacsonyan tartjuk mert nem szeretnénk túlzottan megzavarni a genetikai állományokat. Több alkalmazható stratégia van itt is, most hármat nézünk meg (implementációt lásd 16. ábra). Egyik lehetőség, ha véletlenszerűen választunk egy pozíciót, ahol átbillentjük a bit értéket: ha 0 érték szerepel akkor 1 lesz, és fordítva. Két véletlenszerű pozíción levő bit felcserélése is egy lehetőség. Azt az esetet is megnézhetjük, amikor minden bitet kitesszünk a mutáció lehetőségének, és átfordítjuk őket az adott esély függvényében. [27]

```python
def refreshPopulationElitist(child1, child2, population, fitnessValues, capacity, weights, values):
    fitnessChild1 = calculateFitness(capacity, weights, values, child1)
    fitnessChild2 = calculateFitness(capacity, weights, values, child2)
    fitnessTuples = list(enumerate(fitnessValues))
    fitnessTuples.sort(reverse=True, key=lambda value: value[1])
    indexChild1, indexChild2 = fitnessTuples[-1][0], fitnessTuples[-2][0]
    population[indexChild1]   , population[indexChild2]    = child1, child2
    fitnessValues[indexChild1], fitnessValues[indexChild2] = fitnessChild1, fitnessChild2
    return population, fitnessValues

def refreshPopulationRandomly(child1, child2, population, fitnessValues, capacity, weights, values):
    fitnessChild1 = calculateFitness(capacity, weights, values, child1)
    fitnessChild2 = calculateFitness(capacity, weights, values, child2)
    indexChild1, indexChild2 = random.sample(range(0, len(population)), 2)
    population[indexChild1]   , population[indexChild2]    = child1, child2
    fitnessValues[indexChild1], fitnessValues[indexChild2] = fitnessChild1, fitnessChild2
    return population, fitnessValues
```

17. ábra: A populáció frissítésére használt stratégiák implementálása

A populáció frissítése során is számos stratégiát használhatunk, egyik például az elitista módszer (17. ábra). A stratégia a legrátermettebb egyedek kiválasztását támogatja, amit többféleképpen is lehet használni. Esetünkben ez úgy fog kinézni, hogy kidobjuk a populációban szereplő utolsó kettő legrosszabb rátermettségi értékkel rendelkezőt minden generációban. Ezáltal mindig megőrizzük az eddigi legjobb értékeket. Mivel mindig két gyerek születik a mutációs fázis végeztével, ezeknek kiszámoljuk a rátermettségi értékét majd az adott indexen levő kidobott egyedek helyére tesszük a két új kromoszómát, frissítve a rátermettségi listában vett értéküket is. Más megközelítés alapján véletlenszerűen (lásd 17. ábra) is kiválaszthatunk két egyedet, amit kidobunk a populációból, az új gyerekeket pedig betesszük ezek helyére.



18. ábra: Az evóluciós-genetikus algoritmus szakaszainak lépései

Az fő algoritmus implementációja során (19. ábra) az általunk meghatározott és a 18. ábrán is látható 7 lépést fogjuk követni:

0. lépés: legeneráltuk a populációnkat, majd a számításokat annak érdekében, hogy ne kelljen minden egyes alkalommal újra elvégezni egy listában eltároljuk.
1. lépés: populáció inicializálása után együttesen kiszámoljuk vagy frissítjük a rátermettségi értékeket. Emiatt a 0. és 1. illetve 5. és 1. lépéseket egyszerre végezzük el (ezt a 18. ábrán narancssárga vonalakkal jelöltük).
2. lépés: a szülő kromoszómák kiválasztása történik meg az adott stratégia mentén.
3. lépés: a két szülő keresztezése a kiválasztott stratégiával.
4. lépés: a keresztezés útján kapott gyerekeket tesszük ki a mutáció alá az adott stratégia által. Azonban ahogy a biológiában is nem mindig következik be mutáció, hasonlóan a stratégiától függetlenül sincs garancia a bekövetkezésére.
5. lépés: frissítjük a populációs állományunkat azáltal, hogy a mutációnak kitett egyedeket – attól függetlenül, hogy az valóban bekövetkezett – visszatesszük a populációba. Esetünkben a folyamatot, vagyis az 1.-5. lépéseket mindaddig folytatjuk ameddig el nem értük a megadott cél generációt.
6. lépés: eredmény megadása

Megállási feltételnek egyéb szempontok mentén is lehet határt szabni, nem kell feltétlenül az adott generációig iterálni. Például egy jó kilépési feltétel lehet az is, hogy figyeljük a rátermettségi függvény által határozott értékeket, és ha észrevesszük, hogy az utóbbi X lépés során az értékek nem változnak mert konvergálni kezdtek akkor leállítjuk az evolúciós folyamatot. [28]

```python
def knapsackGenetic(capacity, weights, values, n, generations, populationSize, crossoverProbability, mutationProbability):
    # 0. initialize population & 1. fitness evaluation
    population, fitnessValues = generatePopulation(populationSize, capacity, weights, values, n)
    for _ in range(generations):
        # 2. select chromosomes
        parent1, parent2 = selectChromosomesRankbased(population, fitnessValues)
        # 3. crossover
        child1, child2 = crossoverOnePoint(parent1, parent2, crossoverProbability)
        # 4. mutation
        child1 = mutateBitFlip(child1, mutationProbability)
        child2 = mutateBitFlip(child2, mutationProbability)
        # 5. refresh population & 1. update fitness values
        population, fitnessValues = refreshPopulationElitist(child1, child2, population, fitnessValues, capacity, weights, values)
    # 6. final evaluation
    return getBestChromosome(population, fitnessValues, weights, values)
```

19. ábra: A genetikus algoritmus fő függvényének implementálása

A kiértékelésnél (20. ábra) az utolsó generációban megkeressük a legjobb rátermettségi értékkel rendelkező kromoszómát. Ezek után már csak vissza kell fejteni az adott súlyt, értéket és a tárgyakat. A bináris reprezentálás miatt tudjuk, hogy akkor vesszük számításba a tárgyat, ha az adott pozición 1-es érték szerepel. [28]

```
def getBestChromosome(population, fitnessValues, weights, values):
    maxFitness = max(fitnessValues)
    maxIndex = fitnessValues.index(maxFitness)
    bestChromosome = population[maxIndex]
    maxValue, maxWeight, selectedItems = 0, 0, []
    for i in range(0, len(bestChromosome)):
        if bestChromosome[i] == 1:
            maxWeight += weights[i]
            maxValue  += values[i]
            selectedItems.append(i+1)
    return maxValue, maxWeight, selectedItems
```

20. ábra: A genetikus algoritmus megoldásának kiértékelése

Összességében tehát elmondhatjuk, hogy a genetikus algoritmusunk a hátizsák probléma esetében elég komplex, mivel sok tényező befolyásolja a működését. A futás függ a populáció méretétől, a generáció méretétől, a tárgyak számától, rátermettségi függvény hatékonyságától. A futásidőt és a minél közelebbi eredmény megtalálását pedig tovább befolyásolják a kromoszóma választási, keresztezési, mutációs és populáció frissítési stratégiák. Ezen paraméterek finomhangolásával javíthatunk mindkét előbb említett szemponton.

A futásidőt pontosan nehéz megbecsülni, ha a legrosszabb esetet nézzük, akkor is polinomiálisat érünk el. A becslés során számításba kell venni a rátermettségi értékek kiszámolását, ami viszonylag a legköltségesebb művelet. A rátermettségi függvény a hátizsák probléma esetében a tárgyakon való iterálást jelenti lényegében, ezáltal mondhatjuk, hogy ez közel arányos a tárgyak számával. Egyéb műveletek, amit megemlíthetünk a kiválasztás, keresztezés, mutáció, populáció frissítése. Általánosan viszont elmondhatjuk, hogy ezek a műveletek a kromoszómákon és a populáción hajtódnak végre, így azok méretével arányosak. Ezáltal futás idő becsléseként mondhatnánk, hogy *O(PGn)*, ami megegyezik a populáció méretének *(P)* és a generációk számának (*G*) és a tárgyak számának (*n*) szorzatával. Memória szempontjából a populáció méretének megfelelő listában tárolunk rátermettségi értékeket, illetve a tárgyaink számának megfelelő hosszú kromoszómát bit tömbként.

Tehát ezáltal a memória igény lineáris, *O(Pn)* ahol *P* a populáció mérete és *n* a kromoszómánk hossza, ami a tárgyak számával megegyező.

Optimalizáció szempontjából rengeteg dologgal kísérletezhetünk. Kezdeti szakaszban például a random generált kromoszómák mellett néhány jó értéket manuálisan is megadhatunk. Amennyiben ezt nem szeretnénk, esetleg a már említett mohó algoritmust is lefuttathatjuk, ami gyorsan egy jó közelítő értéket is adhat. Ezáltal javítani tudunk az algoritmus eredményén és a megoldás gyorsabb elérésén is. Továbbá különböző kombinációkban próbálhatjuk ki a kiválasztás, keresztezés, mutáció és populáció frissítési stratégiákat. A paraméterek finomhangolása, mint a populáció mérete, generáció szám, esély a kereszteződésre és mutációra egyéb olyan tényezők, amely hosszas kísérletezésekkel járhatnak.

2.9. Randomizált algoritmusok (Randomized algorithm)

A randomizált algoritmusok ahogy a nevéből is következik, olyan algoritmusok, amelyek futás közben nem determinisztikus sémát követnek. A szükséges döntések meghozatala során valamely ponton vagy teljes egészben véletlenül fogunk választani az adott lépésekben. Mivel az algoritmus végrehajtása közben a véletlen faktor szerepet játszik, ugyan azon inputra többszörös futtatás után is eltérő eredményeket kaphatunk. Emiatt az optimális eredmény elérésére nincsen semmi garancia, azonban az iterációk számának növelése és más pontokból való indítás által lehetséges az eredmény javítása. Ezeket az algoritmusokat legyakrabban akkor használjak a determinisztikusokkal szemben amikor azok nagy futási vagy számítási költséggel járnak. Továbbá alkalmazhatóak akkor is mikor túl nagy a keresési tér vagy az adott input a lehető legrosszabb vagy egy ahhoz közeli végkimenetelt eredményez. [31]

A randomizált algoritmusokat két főbb kategóriába oszthatjuk: Monte Carlo és Las Vegas típusú algoritmusok. A Monte Carlo olyan randomizált algoritmusok, amelyek valószínűség alapján adnak eredményt, például 80% valószínűséggel azt állítja, hogy az eredmény helyes. Ezáltal lehetőség van arra, hogy az elért eredmény helytelen, azonban általában kevés százalékkal. Futásidő szempontjából garantált, hogy belátható időn belül terminálni fog, de a helyességre ez nem mondható el. A Las Vegas fajta algoritmusok ezzel ellentétben biztosan helyes megoldást nyújtanak, még ha nem is az optimálisat. Futásidő szempontjából azonban nehezebb ezt belátni, mivel a

véletlenszerű döntések az eredmény eléréséhez szükséges időt befolyásolják. A neve is a szerencsejátékról híres Las Vegasról ered, ahol valamikor biztos célt érünk, viszont megkockáztatjuk, hogy mennyi ideig is fog ez tartani. Tehát összeségében a Monte Carlo megközelítésben az eredmény helyességét, Las Vegas esetben a futás időt bízzuk a véletlenre. A hátizsák probléma esetében a cél, hogy a tárgyak beférjenek a megadott kapacitás figyelembevételével, emiatt a hiba nem megengedett, ezáltal a Las Vegas megközelítést próbáljuk meg alkalmazni. [32]

```python
def knapsackRandomized(capacity, weights, values, n, maxIteration=50):
    maxValue, maxWeight, selectedItems = 0, 0, []
    for _ in range(0, maxIteration):
        acceptable = False
        while not acceptable:
            tempMaxValue, tempMaxWeight, tempSelectedItems = 0, 0, []
            for i in range(0, n):
                item = random.choice([0, 1])
                tempSelectedItems.append(item)
                if item == 1:
                    tempMaxWeight += weights[i]
                    tempMaxValue  += values[i]
                if tempMaxWeight > capacity:
                    break
            if tempMaxWeight <= capacity:
                acceptable = True
                if tempMaxValue > maxValue:
                    maxValue, maxWeight, selectedItems = tempMaxValue, tempMaxWeight, tempSelectedItems
    selectedItems = [i+1 for i, item in enumerate(selectedItems) if item == 1]
    return maxValue, maxWeight, selectedItems
```

21. ábra: A randomizált algoritmus implementálása

Az implementációhoz (21. ábra) segítségünkre lehet a genetikus algoritmusoknál is már felhasznált véletlenszerű populáció generálás, valamint rátermettségi érték számítás. A Las Vegas elvet alkalmazva tehát érvényes megoldásokat szeretnénk generálni. Egy megoldás az eddig már bemutatott megközelítések által reprezentálható *n* darab bittel. Ha 1-es érték választottunk az i-edik tárgynak, akkor beszámítjuk a súlyát és értékét. A megoldást akkor tartjuk érvényesnek, amennyiben nem lépjük át a súly küszöböt, ezért addig fogjuk generálni a megoldásokat, amíg egy kapacitás alatt levő összértékűt nem kapunk. Az elért eredmény, habár nem garantált, hogy optimális lesz, az iterációk számával a nagy számok törvénye alapján minél jobban közelítheti. Így az algoritmus eredménye a megadott iteráció szám alatt elért legjobb összértékűvel lesz egyenlő.

A futás időre azt mondhatjuk, hogy megközelítőleg *O(nI)* lesz a komplexitása, ahol *n* a tárgyak számát jelöli, *I* az iterációk számát. Általános esetben azt várjuk el, hogy egy elfogadható megoldás generálása nem fog túl sok iterációt elvenni. Ezáltal, ha egy konstans tényezőként tekintünk erre a számra, a futásidő komplexitása nem fog

változni. Ha a legrosszabb esetet nézzük, amikor nagyon balszerencsések vagyunk a helyes megoldások gyártásába, *k* darab iterációként is tekinthetünk rá. Ezáltal a legrosszabb eshetőségbe vett idő komplexitás *O(n * I * k)* lenne. Memória szempontjából két darab *n* hosszú listában tároljuk a megoldásokat, tehát a komplexitása *O(n)* lesz.

2.10. Összehasonlítás és kiértékelés

A módszerek összehasonlításának könnyebb átláthatósága érdekében azt egy táblázat segítségével szeretnénk reprezentálni (4. táblázat). A kitöltés és az algoritmusok összehasonlítás során a hátizsák probléma esetében eddig elért eredmények volt a főbb meghatározó mérce. Az egyes számértékek kiosztásánál, valamint az így kialakított skála kialakításánál az eddig bemutatott algoritmusok egymáshoz vett viszonyítás alapja lett kifejezve. A táblázatban a következő attribútumok alapján lett az összehasonlítás elvégezve:

Futás idő - Az algoritmus futtatásához szükséges időt komplexitását mutatja, hogy mennyi idő alatt számolja ki a megoldást a problémára a legrosszabb esetben.

Pontosság - Az adott algoritmus eredményének pontosságát mérjük, hogy az általa kiszámolt megoldás mennyire közelít az optimális megoldáshoz. A %-hoz relatív hibát számolunk (relatív hiba = |optimális érték – kapott érték| / optimális érték * 100). 100% jelenti, hogy az algoritmus megkapja az optimális megoldást, az idő függvényét azonban ehhez nem számítjuk hozzá. Azokban az esetekben, ahol ettől különböző érték szerepel, általános összefüggés nélküli esetekben vizsgáljuk meg őket. Véletlenszerűen generáltunk 100 teszt esetet, ahol a tárgyak száma az [1, 25], súlyok és értékek az [1, 100] intervallumból került ki. A hátizsák méretét a kapott súlyok összege alapján határoztuk meg százalékos alapon, például 60%-al. Ezzel az értékkel szerettük volna kifejezni, hogy nagy valószínűséggel a tárgyak 40%-át nem tudjuk beletenni, tehát döntéseket kell hozni a mérlegelésre. A 100 teszteset mindegyikére kaptunk így egy relatív hibát, ami jelzi, hogy mennyire tér el az optimális eredménytől. Az átlagos teljesítmény megkapásához ezeket összegeztük, majd elosztottuk a tesztesetek számával. Az így kapott számot 100-ból levonva megkaptuk azt a pontossági százalékot, amennyire az algoritmus 100 véletlenszerű tesztesetben átlagosan megközelítette az optimális eredményt. Persze ezzel csak egy átfogó képet kaphatunk,

a valós teljesítmény meghatározása jelentős mértékben függ az adott problémától, speciális esetektől és fennálló összefüggésektől.

Memória - Az algoritmus futása során feltehetőleg lefoglalt memória komplexitását mérjük az adott stratégia által elért legrosszabb esetében.

Robusztusság - Az algoritmusokat különböző típusú bemeneti adatokon teszteljük annak érdekében, hogy teljesítenek különböző körülmények között például növekvő számú tárgyak mellett. A robusztusság és a futás idő között valamilyen szinten egy párhuzam húzható. A cél a tűrő képesség ellenőrzése, meddig mehetünk el a bementi adatok valamely értékeinek növelésével úgy, hogy az eredmény kivárható legyen. A skála 1-5 értékek között fog mozogni, ahol az 1-es értékkel rendelkező a legkevésbé tűrőképes, az 5-ös érték pedig a legjobban teljesít az input adatok növelése által.

Implementáció - Az algoritmus milyen könnyen implementálható, milyen fokú elmélet szükséges a kivitelezéshez, az így kapott megoldás összetettsége. Az adott kód mennyire érthető és olvasható. Az alábbi skála alapján fogjuk mérni:

1. Nagyon nehéz: A megvalósítása rendkívül összetett, és nagyon sokáig tarthat. Megköveteli a probléma szakértői szintű megértését vagy fejlett programozói ismereteket, továbbá jelentős mennyiségű hibakeresést, tesztelést, optimalizálást és finomhangolást igényelhet.
2. Nehéz: A megvalósítása összetett és hosszabb időt is igénybe vehet. Megköveteli a probléma mély megértését vagy fejlett programozói ismereteket, továbbá jelentős hibakeresést és tesztelést is igényelhet.
3. Közepes: A megvalósítása mérsékelt erőfeszítéssel megtehető, de megköveteli a probléma jó megértését és bizonyos programozói ismereteket, továbbá megkívánhatja a tesztelés és hibakeresés szükségességét is.
4. Egyszerű: A megvalósítása könnyen, ésszerű időn belül megtehető, de megköveteli a probléma megértését és igényelhet némi programozói ismeretet és erőfeszítést.
5. Nagyon egyszerű: A megvalósítása könnyen érthető, egyszerű, gyorsan megtehető és minimális erőfeszítést igényel.

Sorszám	Algoritmus neve	Futás idő	Pontosság	Memória	Robusztusság	Implementálás
1	Brute force	$O(n2^n)$	100%	O(n)	1	4
2	Oszd meg és uralkodj	$O(2^n)$	100%	O(n)	2	3
3	Feljegyzéses módszer	O(nW)	100%	O(nW)	3	3
4	Dinamikus programozás	O(nW)	100%	O(nW)	3	4
5	Mohó	O(n*log n)	99.17%	O(n)	5	5
6	Mohó visszalépéses	O(n*log n)	83.23%	O(n)	5	5
7	Visszalépéses	$O(2^n)$	100%	O(n)	2	3
8	Branch and bound	$O(2^n)$	100%	$O(2^n)$	4	2
9	Evolúciós-genetikus módszer	O(PGn)	95.18%	O(PK)	3	1
10	Randomizált	O(n*I*k)	89.67%	O(n)	4	5

4. táblázat: A különböző módszerek összehasonlítása

Brute force esetén $O(n2^n)$ a legrosszabb futás idő amit elértünk a bemutatott algoritmusokkal a hátizsák probléma kapcsán. Minden lehetőséget meg kell néznünk, majd minden értékre ki kell számolni a súlyt és értéket. Habár mindig optimális megoldást fog adni, a robusztussága a skálán nem hiába a legalacsonyabb. Hajlamos a stack-overflow jellegű hibákra nagy bemeneti értékek esetén. Memória szempontjából mindig csak a jelenlegi tárgy kombinációt és annak az összértékét tartjuk számon. Implementálása könnyű mivel egyszerű algoritmus, csak végig kell iterálni az összes lehetséges kombináción, habár a bitmask indexelést meg kell érteni.

Oszd meg és uralkodj algoritmus esetében mivel minden tárgyra megnézzük, hogy elhelyezzük-e, így kapjuk a $O(2^n)$ futás időt. Rekurziót használ, aminél nagy méretű stack-re lehet szükség nagyobb mértékű inputok esetében. Memória szempontjából itt is csak a kiválasztott tárgyakat tároljuk el egy listában. A rekurzió megértése és a probléma megoldás menete elvár némi ismeretet, azt leszámítva számítva könnyű kivitelezni.

A feljegyzéses módszer során lényegében kompromisszumot kötöttünk. Elkerüljük a redundáns számolásokat, amiket az előző két algoritmusnál végrehajtottunk. A futás időn javítottunk, de eltároljuk az adatokat és memória igényt ezzel együtt növeltük. A táblázat alapján $O(nW)$ a legjobb futás idő, ami elméletben elérhető a hátizsák optimális megoldásának megtalálásánál érdekében. Memória lefoglalása a táblázat miatt ugyan erre nőte ki magát. Robusztussága jobb, mint az előző kettőnek, implementálásnál viszont itt is a rekurzió használata és az érték eltárolása és eredmény visszafejtés miatt egy szinten van az oszd meg és uralkodj algoritmussal.

A dinamikus programozás módszer memória és futásidő költségben majdnem megegyezik a feljegyzéses módszerrel. Ezen esetben lentről-felfele építkezünk a számolásnál meg mindig figyelembe vesszük az előző sor eredményét. Robusztus, implementálásánál csak a 2D tömb feltöltéséhez való számítási logikát, és visszakövetést kell megérteni, azután elég egyértelmű. Általában a hátizsák problémától eltekintve a feljegyzéses módszernél akkor jobb ez a fajta stratégia, amikor a részproblémák sorba rendezhetőek és közöttük nagyobb átfedés van. Vagyis a feljegyzéses módszer képes arra, hogy jobb memória költséget érjen el mivel a táblázatban nem kell minden cellát eltárolni csak, ami szükséges.

A mohó algoritmusunk futás ideje a bemutatott algoritmusok között legjobb, O(n * log n) ugyanis egy rendezéstől és az n elem végig iterálásától függ. A memóriában is csak a kiszámított értéket tároljuk el, illetve a már kiválasztott tárgyakat. Robusztus, eddig legkönnyebben érthető és implementálható algoritmus, azonban az optimális megoldást nem garantálja. Az adott iterálási elvet számítva más-más esetekben különböző eredményeket érhetünk el. A táblázatban az arányos sorba rendezés alapján 100 véletlenszerű tesztesetben 99.17%-ban közelítettük meg az optimális eredményt. Azonban meg kell jegyezni, hogy ugyan ezen a teszthalmazon 98.51% értünk el az érték alapján vett sorba rendezéssel, ami azt jelenti, hogy 19/100 esetben jobb eredményt kaptunk az arányos rendezéssel szemben. A súly alapján növekvő sorrendbe rendezéssel 87.97%-ra közelítettük meg átlagosan az optimális értéket, azonban nem volt olyan eset, ahol jobb eredményt kaptunk volna a másik kettőnél.

A mohó visszalépéses módszer során lényegében szimpla mohó módszerrel egyenlő futás időt kaptunk, azonban az optimális eredménytől sokkal jobban eltávolodtunk. A 100 véletlenszerű teszteset egyikében sem szolgált jobb megoldással ez a módszer. A további tesztelés során összesen 3 alkalom volt, amikor jobban teljesített mint az alap mohó megoldás, olyan helyzetekben amikor a tárgyak súlyai és értékei párhuzamosan növekedtek. Ahogy említettük, a módszer speciális esetekben jó lehet, például amikor kisebb arányú tárgyak jobb kombinációt alkotnak. Olyan esetekkel is lehetett találkozni azonban, ahol a hátizsákba, ami befér jó aránnyal rendelkezik így elől helyezkedik el. A tárgyak pedig amik nem férnek be mert alacsony a hátizsák kapacitása a végén. Egy ilyen esetben az algoritmus ahogy nem tud elhelyezni egy tárgyat eltávolítja a legutolsót. Amennyiben több ilyen tárgy van, lehet, hogy az

összes behelyezett tárgyat végül eltávolítjuk és 0 összértékű eredményt adunk vissza. Ilyen értelemben ezzel a módszerrel inkább rontottuk a mohó stratégián, mintsem, hogy valami javítást elértünk volna.

A branch and bound módszerünk a mohó módszerből kiindulva legrosszabb esetben exponenciális $O(2^n)$ futás időt érünk el, de jó korlát becsléssel és csomópont elvágási technikával polinomiális futás időt is produkálhat. Hasonlóan memória igény is a futás idővel megegyező, ami ugyan olyan módon redukálható. Robusztus, és garantáltan megtalálja az optimális megoldást. Implementálása valamely szinten komplex, ugyanis egy jó becslő érték és kellő út szelektálás néha kihívást jelenthet. A becslés, csomópontok és prioritásos sorral való további potenciális optimalizálása azonban lehetséges. Fontos még megemlíteni, hogy annak ellenére, hogy a futás idő és memória $O(2^n)$ értéke a legrosszabb esetben. Gyakorlatban az esetek nagyobb részében ezek az értékek jelentősen alacsonyabbak, és a dinamikus programozásnál is jobb eredményeket produkálhatnak.

Az evolúciós-genetikus módszer esetében egy fokkal nehezebb meghatározni az optimális eredmény megközelítését, ugyanis rengeteg befolyásoló tényezővel rendelkezik. Az optimálist közelítő érték tesztje során a kromoszóma kiválasztásnál a rulett kerék stratégiát, keresztezésnél az egy pont keresztezést, mutációnál az egy bit átfordító módszert, populáció frissítésénél az elitista elvet alkalmaztuk. Az 22. ábra az algoritmus során használt paraméterekkel való kísérletezést szeretné ábrázolni. A 100 véletlenszerű inputra azt a következést vonhatjuk le, hogy első sorban a generáció száma befolyásolja az eredmény jóságát (a többi értékeket ilyenkor lefixáltuk). A keresztezés és mutáció értékeivel való játszadozás viszonylag 95% körüli optimális eredményhez közeli pontossággal szolgált. Ez jobb, mint a populáció méretének növelésével kapott eredmények. Megjegyzendő, hogy ezeket az értékeket egyszeres futtatás által határoztuk meg. A folyamat során rengeteg tényező véletlenszerű esélyen alapul, így többszörös futtatás esetén ezek az értékek változhatnak. A táblázatban szereplő végeredménynek az 22. ábra alján szereplő paraméterek által kapott 95.18% értéket vesszük.

konstans értékek:	
populationSize	100
crossoverProbability	0.7
mutationProbability	0.3

generations	Pontosság
10	89.79%
200	98.27%
250	98.11%
500	99.06%

konstans értékek:	
generations	100
crossoverProbability	0.7
mutationProbability	0.3

populationSize	Pontosság
10	91.40%
200	94.07%
250	93.85%
500	94.66%

konstans értékek:	
generations	100
populationSize	100

crossoverProbability	mutationProbability	Pontosság
0.5	0.5	94.75%
0.6	0.4	94.73%
0.8	0.2	95.36%
0.9	0.1	95.04%

generations	populationSize	crossoverProbability	mutationProbability	Pontosság
100	100	0.7	0.3	95.18%

22. ábra: Az generikus algoritmus paramétereivel való kísérletezés

A finomhangolás során első sorban azt teszteltük, mely módszerek kombinációjával érhető el általánosan a legközelebbi optimális eredmény. A 100 véletlenszerű teszteseten ötször futtattuk le a bemutatott módszerek különböző kombinációját, ami 36 lehetőséget jelentett (kiválasztás – 3, keresztezés – 2, mutáció – 3, frissítés – 2). Minden tesztesetre más, azonban teszteseten belül minden módszerre ugyan azt a kezdeti generációt használtuk fel. Ötszörös tesztelés után is a top5 kombináció sorrendjeiben véve váltakozott, azonban összességében maradt ugyan az, ezeket átlagolva az legjobb optimális eredményt nyújtók az 5. táblázaton látható.

Rang	Kiválasztás	Keresztezés	Mutáció	Frissítés	Opt. %	Idő
1	selectChromosomesTournament	crossoverTwoPoint	mutateBitFlip	refreshPopulationElitist	96.382	9.259
2	selectChromosomesTournament	crossoverOnePoint	mutateBitSwap	refreshPopulationElitist	96.193	9.960
3	selectChromosomesTournament	crossoverOnePoint	mutateBitFlip	refreshPopulationElitist	96.188	10.74
4	selectChromosomesTournament	crossoverTwoPoint	mutateBitSwap	refreshPopulationElitist	96.024	10.13
5	selectChromosomesRankbased	crossoverOnePoint	mutateBitFlip	refreshPopulationElitist	95.270	22.77

5. táblázat: A genetikus algoritmus finomhangolása során az optimális eredményt legjobban megközelítő stratégiák kombinációja

A táblázatban a fázisok során használt nevek a függvény nevére utalnak. Az elért eredményeket 100 generációval, 100 populációval, 0.7 keresztezés és 0.3 mutációs aránnyal értük el. A táblázat első oszlopa a módszerek által elért rangsort mutatja. A kiválasztás stratégia alapján úgy tűnt, hogy a verseny-torna típusú volt a legjobb választás, ahol a torna számát minden esetben 3-ra állítottuk be. Keresztezés és Mutáció során váltakozó, azonban frissítésnél határozottan az elitista módszer volt a hatásos. A 6. oszlopban látható, mennyire közelítettük meg átlagba véve az optimális eredményt. Az elért idő a 7. oszlopban milliszekundumban és hasonló átlagban értendő.

Az így kapott legjobb kombinációs módszereket megpróbáltuk a keresztezés és mutációs értékek [0, 0.1, … 0.9, 1] halmazból vett értékek lehetséges kombinációinak alkalmazásával javítani. Az első módszerrel átlagosan 97.188%-kal tudtuk megközelíteni az optimális eredményt 0.9 keresztezési és 0.2 mutációs arány mellett. A 2. rangú módszerrel ez az érték 96.71%-ra javult 0.9 keresztezési és 0.3 mutációs esély mellett. A 3. számú megközelítéssel 96.749% kaptunk (0.9, 0.7) értékekre. Látható tehát, hogy különböző módszerekkel a keresztezési és mutációs értékekkel változásával is tudtunk átlagosan javítani az általuk kapott eredményen.

A mohó módszernél említettük, hogy az eredménye felhasználható a kezdeti populáció generálásakor. Ezt az ötletet bevetve a 100 generáció, 100 populáció, 0.9 keresztezés és 0.3 mutációs arány mellett, a 3. rangú megoldás 99.711%-ra, az első 99.674%, a 2. rangú pedig 99.646%-ra közelítette meg átlagosan az elért eredményt. Ezáltal látható volt, hogy a genetikus algoritmusok tesztelés, finomhangolás és egyéb heurisztikák felhasználásával átlagosan 99%-os pontossággal meg tudja közelíteni az optimális eredményt egy véletlenszerűen generált problémán. Fontos kihangsúlyozni, hogy ezt az eredményt a populáció méretének és generációk számának konstans 100 értéken tartásával értük el.

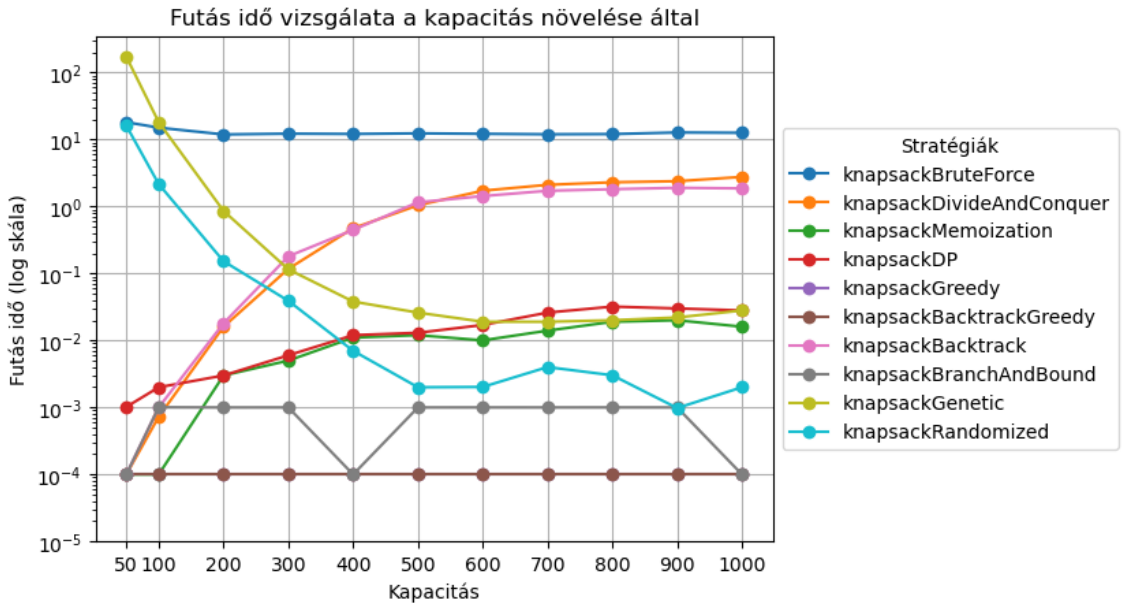
Gyorsaság szempontjából is ellenőrizhetjük ugyan ezen körülmények között (100 generációval, 100 populációval, 0.7 keresztezés és 0.3 mutációs arány) az elért eredményeket. Így egy másfajta sorrendet kapunk ahogy az alsó táblázaton is látható. A főbb befolyásoló tényező ahogy látható a 6. táblázaton leginkább a verseny-torna és random populáció frissítés volt.

Rang	Kiválasztás	Keresztezés	Mutáció	Frissítés	Opt %	Idő
1	selectChromosomesTournament	crossoverTwoPoint	mutateBitFlip	refreshPopulationRandomly	93.97	5.67
2	selectChromosomesTournament	crossoverOnePoint	mutateBitFlip	refreshPopulationRandomly	93.31	5.78
3	selectChromosomesTournament	crossoverTwoPoint	mutateBitSwap	refreshPopulationRandomly	93.37	6.15
4	selectChromosomesTournament	crossoverOnePoint	mutateBitSwap	refreshPopulationRandomly	93.25	6.21
5	selectChromosomesTournament	crossoverOnePoint	mutateBitAll	refreshPopulationRandomly	90.04	6.31

6. táblázat: A genetikus algoritmus finomhangolása során a leggyorsabb eredményt adó stratégiák kombinációja

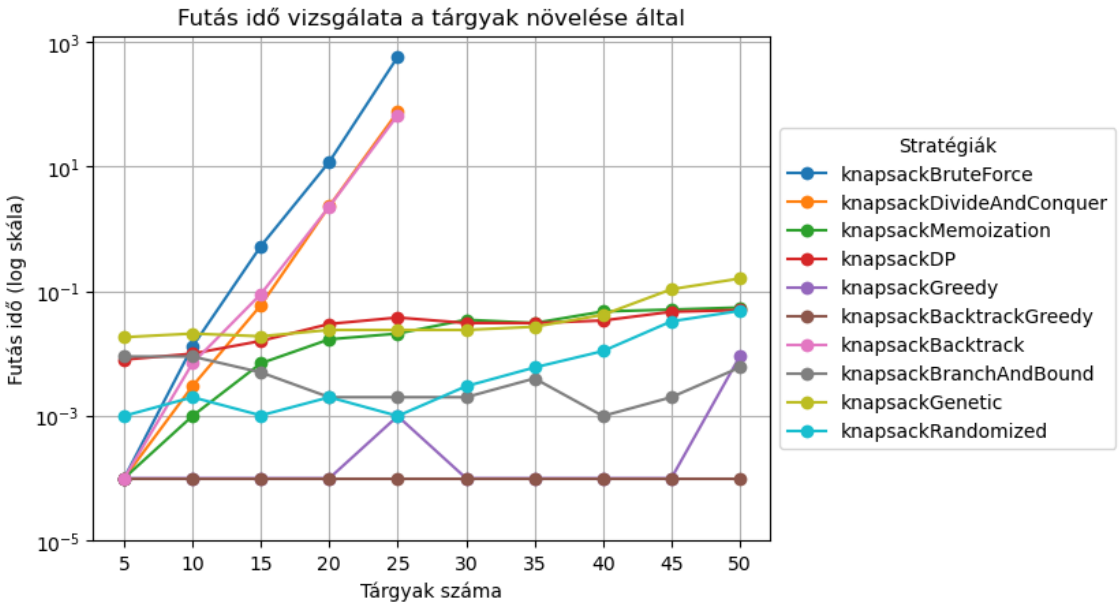
A randomizált módszer során kezdetben 150-re állítottuk be a maximális iteráció számot, ami 91.03% pontosságot eredményezett. A következő eredmény 89.67% ami a 100 maximális iteráció mellett jött ki. Tovább csökkentve ezt az értéket, 50 maximális

iteráció mellett 86.55% lett az átlag a 100 teszt esetre nézve. Tehát a véletlen tényezőt is beleszámítva láthatjuk, hogy a maximális iteráció szám növelésével növekedik az optimális eredmény, persze ennek is van egy határa.



23. ábra: Az algoritmusok összehasonlítása futás idő szempontjából a kapacitás növelése által

Ahogy a 23. ábrán is latható, a robusztusság tesztelése során összevetettük az algoritmusok által elért teljesítményt a kapacitás növelésével. A tesztek során fix 20 tárgyat próbáltunk bepakolni, amelyek súlyai és értékei az [1,100] intervallumból kerültek ki véletlenszerűen. Következtetésként egyértelműen látható, hogy a brute force teljesít a legrosszabbul. Az oszd meg és uralkodj módszer és a visszakövetés lényegében hasonló mértékben növekedik, és látható, hogy a kapacitás növekedésével skálázódik. A genetikus és randomizált algoritmusoknál láthatjuk, hogy minél kisebb a hátizsák kapacitása annál több időre van szükségük egy valós megoldás generálásában. Ennek oka, hogy sokkal nagyobb az esély, hogy a kapott eredmény túllépje a határt. A dinamikus programozás és feljegyzéses módszer is egymás mellett skálázódnak, látottak alapján utóbbi egy kicsivel még jobb is volt. A branch and bound módszerrel láthatóan jól sikerült megbecsülnünk a felső határokat, amivel jól tudtuk szelektálni a fában levő utakat. A véletlenszerű tesztesetek alapján így ezt a módszert nevezhetnénk a legjobbnak, ami a legrövidebb idő alatt eredményezi az optimális eredményt. A mohó és visszalépéses mohó is lathatóan az elvártnak megfelelően ugyan olyan futás időt értek el.



24. ábra: Az algoritmusok összehasonlítása futás idő szempontjából a tárgyak növelése által

Másik módszer, amivel a robusztusságat teszteltük az a tárgyak növelését jelentette ahogy a 24. ábrán is látható. A brute force a logaritmikus skálán látszólag hasonlóan teljesített, mint az oszd meg és uralkodj és visszalépés módszere, a valóságban azonban legalább 5x rosszabb. 25 tárgynál meg kellett állítanunk a futást, mivel a brute force körülbelül 500 másodperc, a másik kettő körülbelül 70 másodperc alatt teljesítette ezt a szintet. A dinamikus programozás, feljegyzéses módszer, valamint a genetikus látszólag hasonló eredménnyel növekedik. A randomizált és branch and bound viszonylag jól bírta a tárgyak növelését. A mohó módszerektől az eredmény pedig elvárható volt.

3. 2D hátizsák probléma

A 2D (két-dimenziós) hátizsák probléma (angol irodalomban használt rövidítés: 2KP) egy olyan kombinatorikus optimalizálási probléma, amelynek az iparban széles körben alkalmazott a felhasználása. Példaként megemlítve számos esetben szolgál alapul a vágási és csomagolási feladatok megoldásában. A problémának sok fajta esete van, amit megkülönböztethetünk egymástól a problémák adott tényezőitől függően. Ilyenek lehetnek például amikor elérhetőség alapján limitált vagy korlátlan számú tárgy áll a rendelkezésünkre; egy vagy több tárolóba szeretnénk pakolni; a tárgyak alakja négyszögű vagy tetszőleges. Ezen variációk miatt a név sokértelmű és megtévesztő lehet, ugyanis az alap probléma értelmezése felfogás, valamint kontextus kérdése.

Az eredeti (KP) 1 dimenziós esethez hasonlítva, a 2D-nek nevezett eset megkapható, ha 1 dimenzió mentén bővítjük az eredeti állapotteret, ezáltal a tárgyainknak 2 súlya, és 1 értéke lenne. A valóságban nehéz elképzelni, hogy egy tárgynak két súlya van ezért a problémát próbáljuk meg általánosítani. Az eredeti koncepciónál maradva így azt kapjuk, hogy 2 attribútum alapján szeretnénk maximalizálni az adott értéket. Ahhoz, hogy a 2D hátizsák probléma feladatát meg tudjuk fogalmazni, az attribútumok értelmezését vesszük figyelembe a kívánt cél érdekében. A következőkben két példa ismertetésével mutatjuk be a való életben történő alkalmazását.

1. példa:

A 2D hátizsák problémát úgy fogalmazzuk meg, mint projekt kiválasztási folyamatot. A projektek olyan tevékenységek, amelyeket egy szervezetnek el kell végeznie a profit megvalósítása érdekében. A költségkapacitás a projektekre fordítható maximális pénzügyi erőforrást jelenti, míg az emberi erőforrás a projektekben részt vevő személyek számát jelöli. Az 1D esetben a hátizsáknak súly kapacitás korlátja volt, 2D esetben azonban egy költség, illetve emberi erőforrás kapacitást definiálunk. A tárgyaink tehát maguk a projektek lesznek, amelyeknek van költsége, ember igénye és profitja. Ha egy projektet elvállalunk és a hátizsákba tesszük, akkor szimplán levonjuk a költséget és ember igényt a meglévő erőforrásainkból. A feladat ezen értelemben úgy szól, hogy válasszunk ki a projektek egy részhalmazát, amelyek költsége nem fogja átlépni a költség és ember igény küszöböt. Ezen feltételek mellett szeretnénk, hogy a

projektek profitjainak összege maximális legyen. Tehát az optimális megoldást szeretnénk megkapni, ahol a projektek profitja maximális, nem lépi át a megadott költséget és emberi küszöböt. Ezáltal az 1D problémából kiindulva a kívánt felhasználás által meg tudtuk fogalmazni a feladatot az adott kontextussal, amivel a 2D esetet a döntések során felhasznált attribútumok száma alapján definiáljuk. [33]

2. példa:

Egy másik területről származó felhasználás a pakolási feladatok megoldására szolgál, amikor a 2D hátizsák problémát geometriai szempontból értelmezzük. Nevéből adódóan a 2D térben az XY koordináta rendszerben úgy fogjuk elképzelni, mint egy téglalapot, ami egy konténer szerepét akarja betölteni. A tárgyaink szélességgel és hosszúsággal rendelkező téglalapok, melyeknek egy meghatározott értékük van. Ezeket szeretnénk a konténerben valamilyen módon elhelyezni úgy, hogy a tárgyaink által meghatározott összérték a lehető legnagyobb legyen. A probléma ezen értelmezése miatt több feltételt is ki kell kötnünk. Első sorban a tárgyak nem fedhetik egymást, valamint a konténer szélességén és hosszúságán belül kell elhelyezkedniük, nem lóghatnak ki abból. A probléma igazi kombinatorikus jellege akkor látszik meg, amikor a tárgyak forgatását is megengedjük. Ekkor az összes tárgy minden potenciális elrendezését és forgatását figyelembe kell venni az optimális megoldás megtalálásához. Ezzel a keresési tér hatalmas lehet, mivel jelentősen megugrik a probléma komplexitása, továbbá ezen indokból kifolyólag az alap esetben nem engedélyezzük a tárgyak elforgatását. Ahogy a [34]-es kutatásban olvashatjuk, ez a felfogás „Talán ez a legtermészetesebb két-dimenziós általánosítása az alap egy-dimenziós hátizsák problémának”. A továbbiakban így ezt az esetet fogjuk a 2D hátizsák alap esetének nevezni. Ezen probléma nehézségéről az előbb említett dolgozat ezt mondja: „Habár az 1D hátizsák problémára a dinamikus programozással pszeudo-polinomiális időben tudtunk egy optimális megoldást adni, a 2D hátizsák probléma erősen NP-nehéz, ugyanis az egy-dimenziós tárolóba helyezés problémát (1BP) is legenerálja” [34]. Amennyiben módosítjuk a feladatot, és nem törődünk a tárgyak értékével, csak a minél hatékonyabb elrendezéssel, akkor megkapjuk a két-dimenziós tárolóba helyezés problémát (2BP), ahol a feladat a fel nem használt hely minimalizálása. [34]

A második eset alapján így már formálisan is megfogalmazhatjuk az alap feladatot: adott egy téglalap alakú konténer, ahol előre definiált annak szélessége (*W*) és hosszúsága (*H*). Továbbiakban adott egy *n* elemű halmaz, amely tárgyakat (*t*) tartalmaz. Minden i. ($i \in \{1, ..., n\}$) tárgy: rendelkezik szélességgel (w_i), hosszúsággal (h_i) és értékkel (v_i). A feladat célja megtalálni a tárgyaknak egy olyan részhalmazát, amelyre a tárgyak összértéke maximális. A formula megegyező az 1D esetben már bemutatottal: $\max(\sum_{i=1}^{n} v_i\, b)$. A formulában szereplő v_i az i. tárgy értéke, amire a b ($b \in \{0,1\}$) bináris együttható fejezi ki, hogy a tárgy kiválasztásra került-e vagy sem (0-nem, 1-igen). Miután egy tárgyat beletettünk a konténerbe számolni kell az adott i. tárgy XY-koordináta rendszerben vett kiinduló pozíciójával, ami (x_i, y_i). Egy tárgyat ezáltal a kiinduló pontból x tengely mentén levő szélesség, illetve y tengely mentén húzott hosszúságával való vízszintes és merőleges párhuzamos vonalak által közre zárt terület fog reprezentálni. Az így kapott tárgy a konténeren belül kell maradjon, ezért teljesülnie kell minden i. tárgyra a következőnek: $\forall i: (0 \le x_i + w_i \le W \text{ és } 0 \le y_i + h_i \le H)$. Ezen kívül a tárgyak a 2D térben nem fedhetik egymást, azaz metszetük üres, ami kifejezhető a következőképpen: $\forall i, j: (x_i + w_i \le x_j \vee x_j + w_j \le x_i \vee y_i + h_i \le y_j \vee y_j + h_j \le y_i)$ ahol $i \ne j$. Ezáltal matematikailag is ki tudtuk fejezni azt a 4 lehetőség, ami fennállhat, ha az i és j tárgy kiválasztásra került:

- $x_i + w_i \le x_j$ – i tárgy van bal oldalon, j tárgy jobb oldalon
- $x_j + w_j \le x_i$ – j tárgy van bal oldalon, i tárgy jobb oldalon
- $y_i + h_i \le y_j$ – i tárgy j alatt lehet
- $y_j + h_j \le y_i$ – j tárgy i alatt lehet

[35], [36], [37]

A 2D hátizsák probléma a gyakorlatban való sok színű felhasználási lehetősége miatt jelentős méretű figyelmet kapott a kutatóktól. Variánsaira számos megoldás született már az idők során beleértve a pontos optimális megoldásokat, illetve heurisztika felhasználásával jó becslést adókat, valamint a hibrid algoritmusokat. A területen elért sikerekhez jelentős hozzájárulás köszönhető a guillotine vágás fogalmának a bevezetésével.

3.1. Guillotine vágás

A guillotine vágás egy speciális fajta vágási technika, amit gyakran használnak a 2D vágási és pakolási feladatok megoldása során. A hátizsák probléma esetében ez bónusz kikötéseket és korlátokat jelent melyek bizonyos mértékben lecsökkenthetik a probléma nehézségi fokát egy újfajta variánsát adva a feladatnak. A problémával számos tanulmányban foglalkoztak, valamint mutattak be jobbnál jobb közelítő algoritmusokat. Az optimális megoldás eléréséhez jelentősebb hozzájárulás köszönhető például Mhand Hifi branch and bound módszerén[4] valamint Nicos Christofides dinamikus programozáson[5] alapuló megközelítésének.

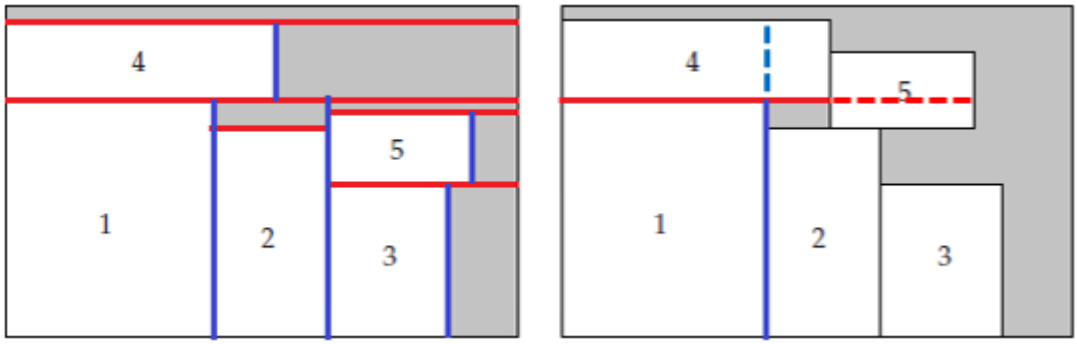
A guillotine vágás egy folytonos vonal mentén vett széltől szélig húzó vágási minta. Elsődlegesen téglalap alakú alakzatokon alkalmazzuk vágások sorozatát, azonban más alakzatok esetében is adaptálható az ötlet. Alapértelmezetten ortogonális metszéseket hajtunk végre, vagyis a vágások a konténer vízszintes és függőleges tengelyei mentén kell megtörténnie, továbbá ezek a vágások nem keresztezhetik egymást. Alap esetnél maradva egy vágás után két téglalap alakú területet kapunk anélkül, hogy a már eddig belehelyezett elemeket elmetszettük volna az élüktől különböző pozíció mentén. Miután a vágás megtörtént nem lehet már visszahelyezni az elvágott felületet, valamint a további vágások is csak az előző által alkotott kisebb részeken hajthatóak végre. Ezáltal minden egyes rákövetkező vágás az előző vágás által alkotott teret fogja további kisebb részekre osztani. Ezt a fajta vágást leginkább úgy könnyű elképzelni, hogy az adott tengely mentén, ha a vágó fej elkezdi a vágást, akkor végig halad miközben a fej nem emelhető fel. A 25. ábrán látható, hogyan is néz ki a guillotine vágással, illetve anélkül vett szabási minta két-dimenziós esetben. [35], [38]

Amennyiben az elvet pakolás során alkalmazzuk, akkor a kapott pakolásnak elérhetőnek kell lennie valamilyen sorrendben vett guillotine vágások sorozataként. Guillotine vágással a tárgyak kisebb darabokra szabhatóak a körvonaluk mentén és így is kell elhelyezni őket. Ezzel szemben a nem guillotine vágásnál, ahol az előző elvet próbálva a tárgyainkat nem tudjuk ily módon feldarabolni emiatt úgy kell elhelyezni

[4] lásd Mhand Hifi: „Exact algorithms for the guillotine strip cutting/packing problem” c. kutatását

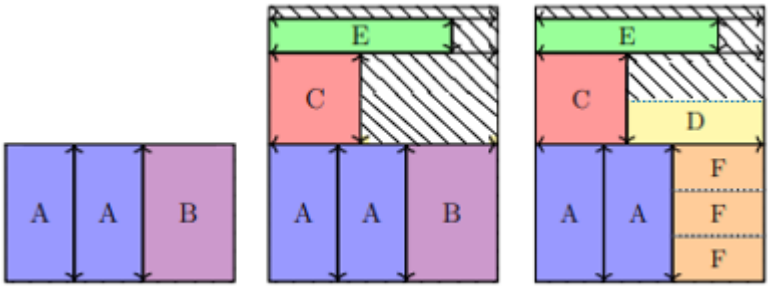
[5] lásd Nicos Christofides, Eleni Hadjiconstantinou: „An exact algorithm for orthogonal 2-D cutting problems using guillotine cuts” c. kutatását

őket ahogy vannak. Általánosan guillotine vágásos módszerek könnyebben és gyorsabban implementálhatóak, mint az egyéb nem guillotine vágással vett megoldások. Ezen esetekben azonban az optimális megoldás, avagy a helyzetben levő leghelytakarékosabb pakolás előállítása nem garantált. [38], [39], [40]



25. ábra: guillotine vágás (bal) és nem guillotine vágás (jobb) [35]

A guillotine vágás, mint probléma számos kategóriára osztható az alkalmazott dimenziók vagy pedig a megszabott kikötések mentén. Ezért pont, mint a hátizsák esetében itt is beszélhetünk az egy-, kettő- vagy három-dimenziós guillotine vágásról. Egy dimenziós esetben csak 1 tengely mentén, azaz csak vízszintesen, vagy csak függőlegesen vághatunk. Speciális esetek között megemlíthetjük a nem ortogonális esetet, amikor az alap ortogonálistól eltérően nem csak 90°-ban végezhetjük a metszéseket. [40]

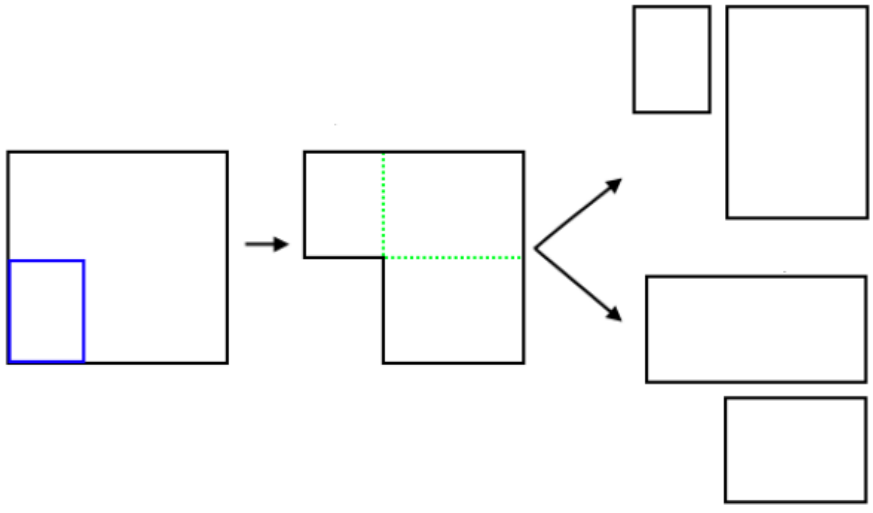


26. ábra: 1-, 2-, 3-szakasos guillotine vágás [40]

A vágásokra vonatkozóan speciális esetekként is beszélhetünk korlátozás nélküli és szakaszos vágásról. Az első a nevéből is adódik, szabadabbak vagyunk, ugyanis a vágások itt bármilyen sorrendben követhetik egymást a szakaszostól eltérően. Utóbbi esetben ugyanis minden vágás során egy függőleges vágást egy vízszintes követ és fordítva. Ezáltal a korlátozás nélküli megközelítés sokkal rugalmasabb, amivel jobb szabásmintát lehet alkotni, azonban mivel több lehetséges esetet vizsgál meg ezzel párhuzamosan nő a megoldás komplexitása és erőforrásigénye. Visszatérve az

általános vágáshoz, a név ennél is árulkodó, ugyanis a vágásokat szakaszokon át végezzük, amelyekre akár egy határt is adhatunk. Minden egyes szakaszban egyszerre csak egy tengely mentén végzünk el egy vagy a szükséges számú vágást. A szakasz végeztével következik a következő szakasz, ami az előzőtől eltérő, egy másik tengely mentén végzi el a vágásokat. A folyamat, avagy a szakaszok váltakozása mindaddig folytatódik, amig a megadott szakasz határt el nem érjük vagy az összes kívánt négyszöget meg nem szereztük. [40]

A szakaszok száma alapján megkülönböztetjük a 1-, 2-, 3-, k-szakaszos guillotine vágást, amit az 26. ábra is szemléltetni kíván. A 2-szakaszos esetben 2 sorozatban, sorozatonként egy adott irányban, 3-szakaszos hasonlóan. Utóbbira például elvégezzük a kívánt függőleges vágásokat, a felosztott partíciókon ezt követően vízszintes vágásokat hajtunk végre, amit még egy adag függőleges vágások sorozata követ. Minél magasabb az adott szakaszos vágást jelző szám, annál hatékonyabb és rugalmasabb pakolás érthető el. Egyes esetekben a k-szakaszos vágással megtalálhatjuk az optimális megoldást, míg ezzel szemben a többiek maximum egy jó közelítéssel tudnak szolgálni. Az, hogy hány szakaszban szeretnénk végrehajtani a vágásokat általában az adott problémától, megkötésektől és vágási stratégiától függ. [40]



27. ábra: A guillotine vágás után kapható szabad területek [41]

A hátizsák probléma esetében az eddigiek alapján a vágások a konténerünk tengelyeihez igazodnak és egy adott irányban, egyetlen függőleges vagy vízszintes vonallal vett vágással elvághatóak anélkül, hogy az valamely más tárgyat metszene. A guillotine vágás alkalmazásával a tárgyaink konténerbe történő bepakolása az adott

szélességgel és hosszúsággal vett párhuzamos vonalak mentén történhet. A belehelyezés után így valamelyik tengelyen végre kell hajtani a vágást, ami az eddig létező szabad helyet két részre osztja. Egyértelműen, attól függően, mely tengely mentén végezzük el a vágást, más fajta rész partíciók keletkeznek a rendelkezésre álló helyből (27. ábra). Ahogy az 1-dimenziós esetben is, hasonló módon alkalmazhatjuk a különböző stratégiákat az implementáció során. A legegyszerűbb próba, amely során guillotine vágással megoldható a probléma, a mohó módszeren fog alapulni.

3.1.1. Mohó algoritmus

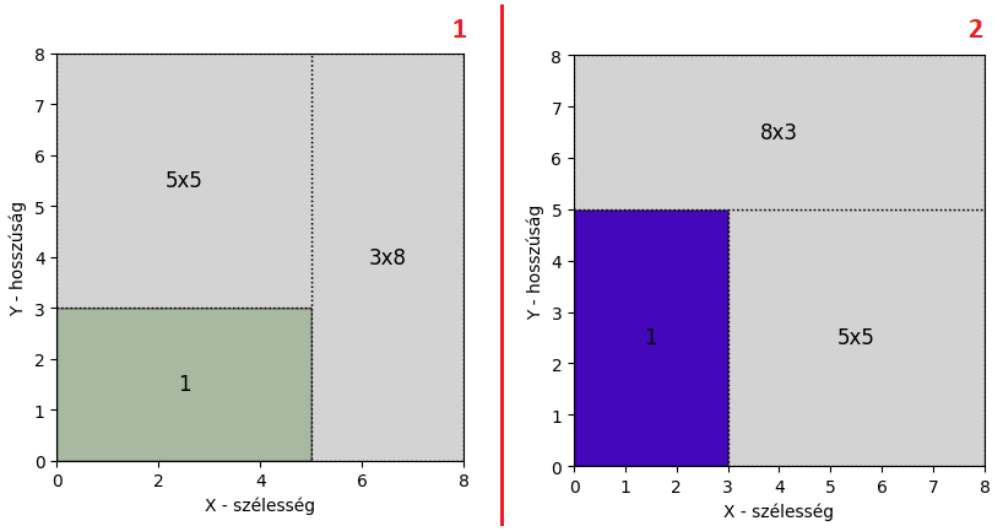
A mohó módszernél, hasonlóan az 1D esethez viszonylag egyszerű és gyors megoldást kapunk, azonban nincs garancia arra, hogy a talált megoldás a legjobb lesz, mivel mindig a lokálisan legjobb értéket választjuk. A megoldás során egy maximum téglalapokat nyújtó algoritmusból fogunk kiindulni (MRA), amely a tárolóba pakolás hatékonyságára célzott megoldást adni. Az algoritmus alap ötlete, hogy mohó módszertan alkalmazásánál a már használt indítással, rendezzük sorba a tárgyainkat. Egy listában számontartjuk a már behelyezett téglalapokat és azok pozícióját, valamint a meglevő szabad helyeket, ami kezdetben a konténer méretét kapja. Ezt követően végig megyünk a már rendezett téglalapokon, és megpróbáljuk őket guillotine vágással belehelyezni a konténerbe levő szabad helyekre. Ha belehelyezhető, akkor beletesszük és frissítjük a tárolót azáltal, hogy egy vágással felosszuk a meglévő szabad helyeket és kivesszük a tárgy beletevése előttit. Amennyiben nem helyezhető bele a tárgy, megpróbáljuk elfordítani ortogonálisan és megnézzük úgy bele tudjuk-e helyezni. A folyamat addig tart, amíg minden egyes téglalapra meg nem néztük, hogy belehelyezhető-e. [41]

```python
class Guillotine:
    def __init__(self, width, height):
        self.width            = width
        self.height           = height
        self.freeSpaces       = [(0, 0, width, height)]
        self.placedRectangles = []
        self.maxValue         = 0

    def placeRectangle(self, index, width, height, value):
        self.freeSpaces.sort(key=lambda space: space[2] * space[3])
        for i, (x, y, w, h) in enumerate(self.freeSpaces):
            if width <= w and height <= h:
                self.placedRectangles.append((index + 1, x, y, width, height))
                self.freeSpaces.pop(i)
                self.maxValue += value
                if w - width > h - height:
                    if w - width > 0 and height > 0:
                        self.freeSpaces.append((x + width, y         , w - width,     height))
                    if w > 0 and h - height > 0:
                        self.freeSpaces.append((x        , y + height, w        , h - height))
                else:
                    if w - width > 0 and h > 0:
                        self.freeSpaces.append((x + width, y         , w - width, h         ))
                    if width > 0 and h - height > 0:
                        self.freeSpaces.append((x        , y + height,     width, h - height))
                return True
        return False
```

28. ábra: A tároló osztály létrehozása valamint egy tárgy elhelyezésének menete

Az implementáció tehát a tároló létrehozásával kezdődik (28. ábra), aminek előre megadott szélessége és hosszúsága van. Továbbiakban eltároljuk még az eddig bepakolt tárgyak értékét (ami kezdetben 0), valamint a fentebb megbeszélt módon a már behelyezett tárgyakat (ami kezdetben üres), és elérhető szabad helyeket. Mivel a tárgyakat a XY koordináta rendszeren szeretnénk elhelyezni, ezért kezdetben az üres hely maga a konténer. Tehát ez a (0,0) pontból X tengely mentén húzott szélesség és Y tengely mentén húzott hosszúság által bezárt téglalapot jelenti.



29. ábra: Guillotine vágás ábrázolása egy 8x8-as konténer esetében

Minden tárgy behelyezése során első lépésünk, hogy növekvő sorrendbe rendezzük a szabad helyeket, ugyanis ezzel szeretnénk elérni, hogy a tárgyak a lehető legjobban méreteikhez illeszkedő szabad helyre kerüljenek be (Best Area Fit). A behelyezés során így csak azt fogjuk ellenőrizni, hogy melyik az az első szabad partíció a listában, ahova a tárgyunk már behelyezhető, vagyis rendelkezik a kellő szélességgel és hosszúsággal. A behelyezés során a szabad hely bal alsó sarkába próbáljuk meg elhelyezni a tárgyat. A már belerakott téglalapokról számontartjuk, hogy melyik tárgyról is van szó, az akkori szabad hely mely pontjából fog kiindulni a méreteivel rendelkező két húzott vonal. Ezt követően eltávolítjuk a használatban levő szabad területet, ugyanis felosztás következik, ahol növeljük a gyűjthető összeget a tárgy értékével. [41]

A szabad helyek újra osztása során a behelyezés után fentmaradó szélesség és hosszúság alapján hozzuk meg a döntésünk:

- fentmaradó szélesség kisebb, mint a hosszúság: függőleges vágás. A tárgy hosszúsága mentén levő teljes jobb oldal, majd a tárgy felett megmaradt szabad hely kerül hozzá (29. ábra 1. eset).
- fentmaradó szélesség nagyobb, mint a hosszúság: vízszintes vágás. A jobb oldal, majd a tárgy fölött levő szélessége mentén vett teljes szabad helyet adjuk hozzá (29. ábra 2. eset).

A szabad helyek hozzáadása során a kiinduló ponthoz hozzá vesszük a tárgy X vagy Y tengelyen levő méreteit a hely pozíciójától függően. Az elérhető új helyek méretét pedig a régi és behelyezett tárgyak méretének különbsége fogja adni hasonlóan X vagy Y tengely mentén. Végezetül visszatérünk az értékkel, hogy a behelyezés sikeres vagy sikertelen volt, ugyanis ezáltal lehetőséget adunk a forgatásra (amennyiben megengedjük).

```
def guillotineGreedy(rectangles, containerWidth, containerHeight):
    guillotine = Guillotine(containerWidth, containerHeight)
    sortedRectangles = sorted(enumerate(rectangles), key=lambda value: value[1][2], reverse=True)
    for index, (width, height, value) in sortedRectangles:
        guillotine.placeRectangle(index, width, height, value)
        #if not guillotine.placeRectangle(index, width, height, value):
        #    guillotine.placeRectangle(index, height, width, value)
    return guillotine.placedRectangles, guillotine.maxValue
```

30. ábra: A mohó guillotine vágás fő algoritmusa

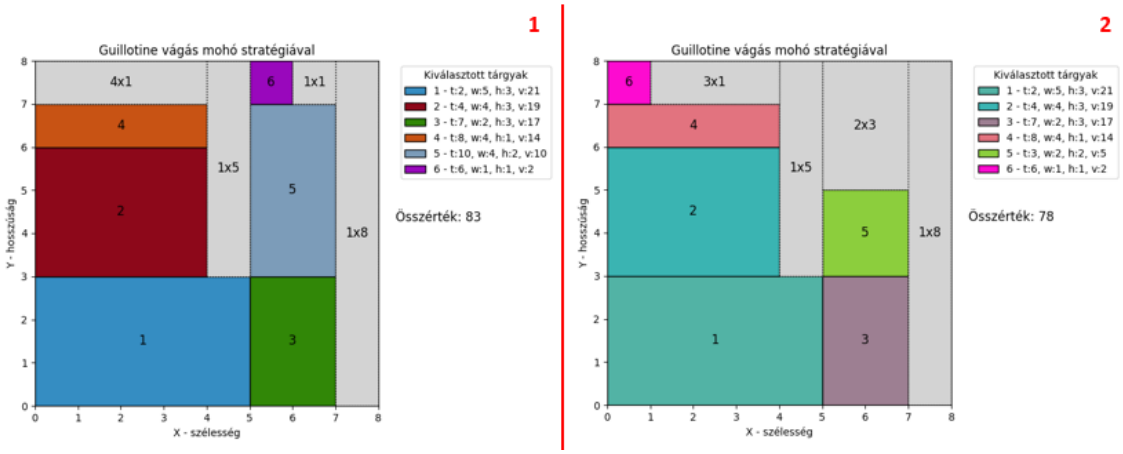
A fő algoritmus ahogy az 30. ábrán is látható, elég egyszerű. Inicializáljuk a tárolónkat a fentebb leírt módon, majd a téglalapokat ahogy a mohó módszernél megszokott, érték alapján csökkenő sorrendbe rendezzük. Az iteráció így már a sorba rendezett téglalapokon megy végbe, amiket majd egyesével megpróbálunk bele pakolni. Az esetünkben a jobb megoldás megtalálása érdekében megengedjük a forgatást. Így, ha az adott tárgy érték szerint a következő a sorban, akkor ne dobjuk el, próbáljuk meg ortogonálisan elforgatva ismételten belehelyezni. Amennyiben kizárjuk a forgatás lehetőségét, akkor ahogy a kikommentált sor mutatja, csak szimplán meghívjuk az elhelyező függvényt, annak eredményét pedig már nem ellenőrizzük le. Végezetül megkapjuk az összegyűjtött értéket, valamint a bele helyezett tárgyak listáját, amely során számontartottuk, hogy melyik tárgyat pakoltuk bele, illetve azt, hogy hol helyezkedik el.

A gyakorlatban az algoritmus működése 2D-ben már vizuálisan is reprezentálható, amit a következő példán keresztül nézünk meg: a tárgyakat a konténerbe próbáljuk tenni, úgy, hogy az általuk adott összérték a lehető legnagyobb. A 7. táblázat tartalmazza minden egyes tárgyra (*t*) a megadott szélességet (*w*), hosszúságot (*h*) és értéket (*v*). A tárgyakat egy 8 hosszú és széles konténerbe szeretnénk belepakolni.

t	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
w	3	5	2	4	3	1	2	4	4	4	4
h	2	3	2	3	3	1	3	1	5	2	4
v	9	21	5	19	12	2	17	14	3	10	8

7. táblázat: A tárgyak szélessége, magassága és értéke a bemutatandó példához

Az algoritmus futásának vizuális eredménye a 31. ábra 1. esetében látható. A 11 tárgyból 6 került kiválasztásra, amivel 83 maximális összértéket tudtunk forgatással elérni. Az ábrán az XY koordináta rendszerben kirajzolt téglalapok jelzik az adott tárgyat, a rajtuk szereplő szám az adott tárgy konténerbe behelyezésének sorrendjét reprezentálja. A kiválasztott tárgyak megjegyzés alatt láthatjuk, melyik tárgyról is volt szó és milyen adatokkal rendelkezett eredetileg (táblázat), ugyanis a lehetséges forgatások miatt gondot okozhat a beazonosítás. A szürke mezők a guillotine vágások által megmaradt elérhető szabad helyek szélességét és hosszúságát jelölik.

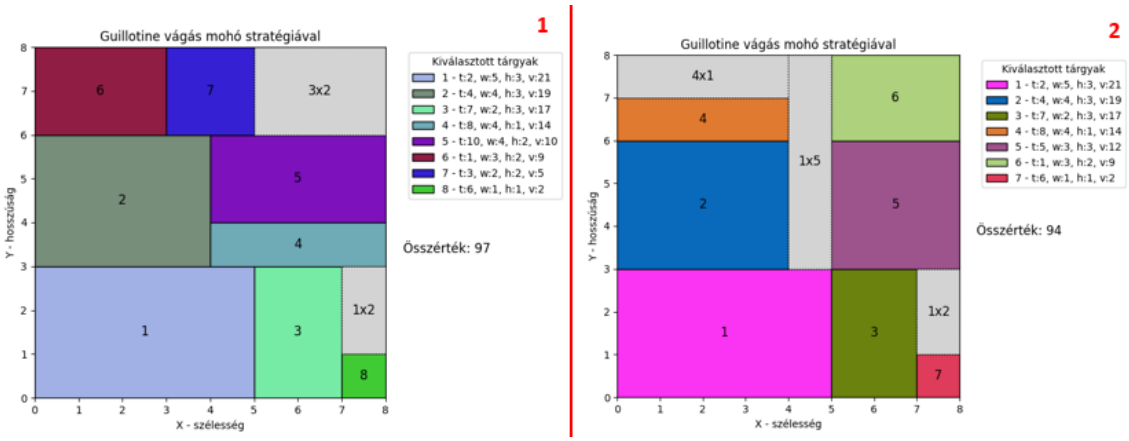


31. ábra: Guillotine vágás forgatás megengedéssel (1) és kizárással (2) a fentmaradó hely alapján

Amennyiben megtiltjuk a tárgyak forgatását megengedő feltételt, a 31. ábra 2. esetében szereplő elrendezést kapnánk. Látható, hogy hasonlóan 6 tárgy került kiválasztásra, de csak 78 összértéket sikerült megszerezni, illetve a szabadon maradó felület is sokkal nagyobb. Ezáltal is láthatjuk, ahogy a probléma komplexitásának növelésével (forgatások) egyaránt közelíthetünk az optimális megoldás fele is.

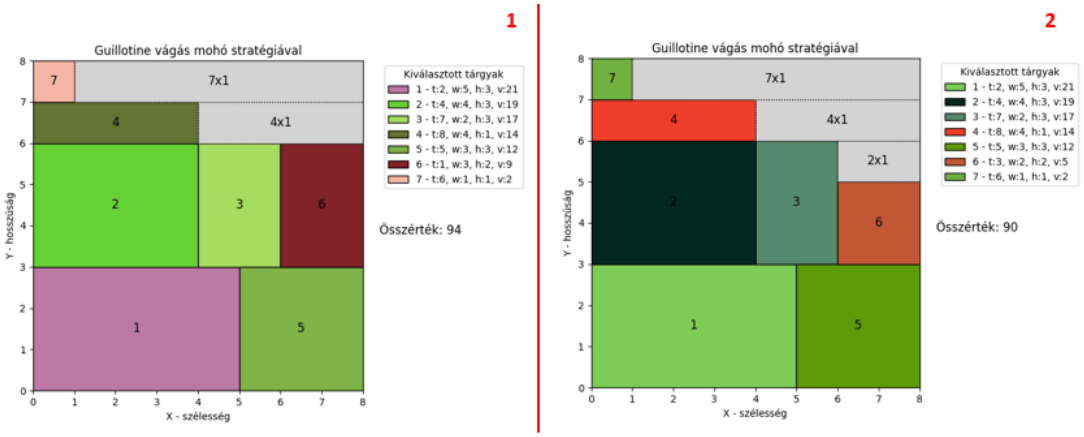
Az algoritmus teljesítménye teszteléssel és különböző heurisztikák kombinációjával potenciálisan javítható. A téglalapokat a guillotine vágás esetében a legkönnyebben és gyakrabban használt elhelyezés a már kiválasztott szabadhelyen a bal alsó sarokba helyezés. A heurisztikák alkalmazhatóak például a vágás során vagy akár a szabad helyek kiválasztáskor. A fentebb leírt implementációban a vágás során megmaradó hely alapján döntöttük el, mely tengely mentén tegyük. Egyéb heurisztikák lehetnek például a rövidebb/hosszabb tengelyekkel párhuzamos/merőleges oldal metszése, csak vízszintes, vagy csak függőleges vágások alkalmazása.

Az, hogy javítani tudjunk az elért eredményen néhány befolyásoló tényezőtől is függhet, mint például megengedjük-e a forgatást, vagy maga az input milyen sajátossággal rendelkezik. A fenti két heurisztikát alkalmazva a már bevezetett példán keresztül, amennyiben csak vízszintes vágásokat alkalmaztunk, illetve a hosszabb oldallal párhuzamos tengely menti vágással 97 összértéket tudtunk szerezni (32. ábra 1. eset). Amikor a rövidebb tengely mentére korlátozódott a vágás (32. ábra 2. eset) 94 összérték volt elérhető. A csak függőleges vágások esetén ugyan azt az összértéket kaptuk, mint a kezdeti forgatással megengedett esetben. Összességében 3 esetben is javítani tudtunk az elért elsődleges eredményen, a forgatás megengedése vagy kizárása nem mutatkozott nagy különbséget okozó befolyásoló tényezőként.



32. ábra: A tárgy hosszabb (1) és rövidebb (2) oldala mentén való vágás

A már bemutatott implementációban növekvő sorrendbe helyeztük a szabad helyeket a terület alapján. Ezzel elértük, hogy a behelyezendő tárgy és a szabad terület különbsége a lehető legkevesebb legyen. A heurisztika fordítva is alkalmazható: ekkor az elérhető legnagyobb helyre fogjuk mindig tenni a tárgyunkat (Worst Area Fit). A forgatás megengedése (33. ábra 1. eset) és kizárása (33. ábra 2. eset) a csak vízszintes vágásokkal adott minimálisan különböző eredményt.



33. ábra: Szabad területek növekvő sorrendbe helyezése forgatás megengedéssel (1) és kizárással (2) csak vizszintes vágások mentén

Eredményül elmondható, hogy a kezdetleges algoritmussal forgatás nélkül az adott példán 78, forgatással 83 volt a szerezhető érték, heurisztikák alkalmazásával és forgatás kizárásával a legnagyobb megszerezhető érték 97 lett. Ez leginkább a vágási technika módosítása által volt elérhető. A heurisztikák alkalmazása és az általuk elért eredményeket azonban nagyban befolyásolja a meglévő input adat eloszlása. Művelet igények szempontjából a futás idő *O(nh)*, ahol *n* a tárgyak száma *h* pedig a szabad helyek száma. A rendezés költsége *O(n*log n),* azonban az input növekedésével az *nh*

gyorsabban fog nőni így emiatt nem számoljuk bele. Memória szempontjából *O(n)* a kapott komplexitás, ugyanis a szabad helyeket és tárgyakat tartjuk számon a listában. Ez legrosszabb esetben n, azaz ha minden tárgyat belepakolunk az eredményt tartalmazó listába.

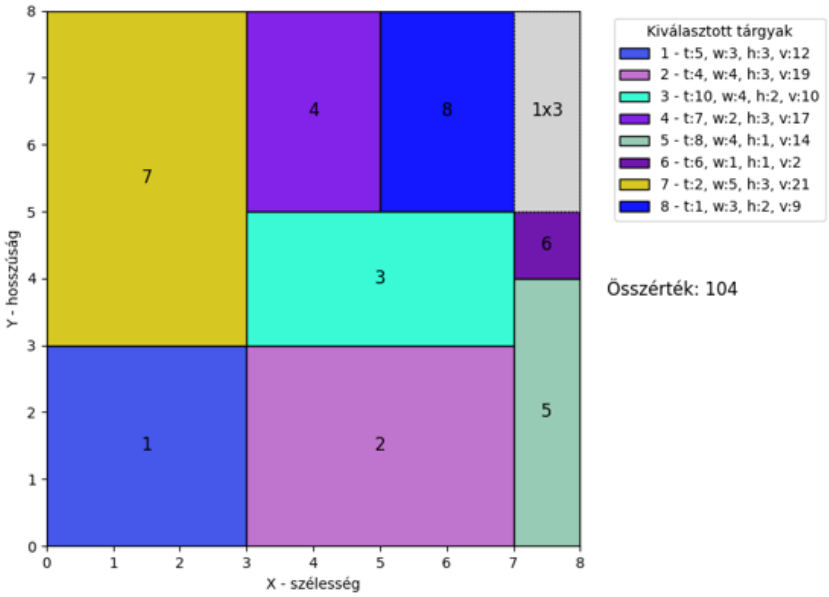
3.1.2. Randomizált algoritmus

A Las Vegas fajta randomizált algoritmusokat ahogy az 1D esetben, a probléma erősen NP-nehézsége miatt a bővített 2D hátizsák esetében is ajánlottan felhasználhatjuk. A mohó algoritmusból kiindulva minimális módosítással meg is kaphatjuk a randomizált algoritmusunkat. A szabad helyek felosztási elve megmarad, a kezdetleges, nagyobb különbséget adó tengely vonala határozza meg a vágás menetét. Az adott tárgyat a méreteihez legközelebb álló üres helyre próbáljuk elhelyezni a kiindulópontként szolgáló bal alsó sarkokban.

```
def guillotineRandom(rectangles, containerWidth, containerHeight, maxIteration=100):
    placedRectangles, maxValue = [], -1
    rectangles = [(i, rectangle) for i, rectangle in enumerate(rectangles)]
    for _ in range(0, maxIteration):
        guillotine = Guillotine(containerWidth, containerHeight)
        random.shuffle(rectangles)
        for index, (width, height, value) in rectangles:
            #guillotine.placeRectangle(index, width, height, value)
            if not guillotine.placeRectangle(index, width, height, value):
                guillotine.placeRectangle(index, height, width, value)
        if guillotine.maxValue > maxValue:
            placedRectangles, maxValue = guillotine.placedRectangles, guillotine.maxValue
    return placedRectangles, maxValue
```

34. ábra: Randomizált guillotine vágásos algoritmus implementálása

A fő algoritmus (34. ábra) elve lényegében az 1D-esetben használtan alapszik, azaz érvényes megoldásokat szeretnénk adni, amiket a megadott ismétlés számig generálunk. Azért, hogy mindig más eredményt kapjunk a mohó stratégián alapuló sorba rendezést lecseréltük véletlenszerű sorba rendezésre. Így mindig más sorrendben fogjuk lehelyezni a tárgyakat, ami által más elrendezést kapunk. A jobb és egyedibb elrendezések érdekében megengedjük a tárgyak elforgatását is. Végeredményül az iterációs lépések végeztével a futás alatti legjobb összértéket nyújtót fogadjuk el eredményül.



35. ábra: A randomizált algoritmus által kapott eredmény forgatás megengedésével

Az algoritmust tesztelve (35. ábra) 3 futtatás után, alkalmanként 100 iterációval a már bemutatott példán keresztül sikerült elérni az eddigi legnagyobb összértéket, vagyis 104-et. Persze számítani kell vele, hogy az optimális megoldás így sem garantált, mivel véletlenszerűen választjuk a tárgyak sorrendjét más-más eredményeket kaphatunk a futtatások során. Ennek ellenére, amennyiben az erőforrások megengedik a többszörös iterációt, belátható mennyiségű tárgyal eredményes módszernek ígérkezik ez a fajta megközelítés. Futás idő szempontjából *O(Inh)*, ahol *I* a maximális iterációk száma, *n* a tárgyaink száma és *h* pedig a szabad helyek száma. Memória szempontjából hasonlóan a tárgyakat és szabad helyeket tároljuk, ami *O(n)* lesz.

3.2. Nem guillotine vágás

A nem guillotine vágást használó vágás technikák alkalmazásával, szemben a guillotine vágással, a metszéseknek több lehetséges formációt adunk. Így nem kell ahhoz a megkötéshez ragaszkodni, hogy a szabad helyeket mindig kisebb téglalap alakú felületekre osszuk a minta alapján. Az elhelyezett tárgyak tetszőlegesen bárhol elhelyezkedhetnek, mivel a vágásoknak nem kell a hátizsák tengelyeihez igazodniuk. Ebből kifolyólag a nem guillotine vágásos technikákkal sokkal rugalmasabbak tudunk lenni. A tárgyakat jobban el lehet helyezni, habár ez a problémát sokkal komplexebbé is alakíthatja. Így szemben a guillotine vágással jobb eredményt adhat, általánosan pedig sokkal nehezebb implementálni a heurisztikákra hagyatkozva. A számítások több

időt és erőfeszítést is igénybe vehetnek, emiatt az irodalomban is viszonylag kevesen foglalkoztak a nem guillotine vágásos pontos eredményt adó algoritmusokkal. Erre a típusú megközelítésre főként heurisztikus módszerek használatát javasolják. A legegyszerűbb eset szemléltetésére kiindulhatunk a mohó guillotine vágásos algoritmusunkból is a megkötések eltávolításával és annál nem használt heurisztikák bevezetésével. [42]

3.2.1. Első szabad helyre való elhelyezés (First-fit)

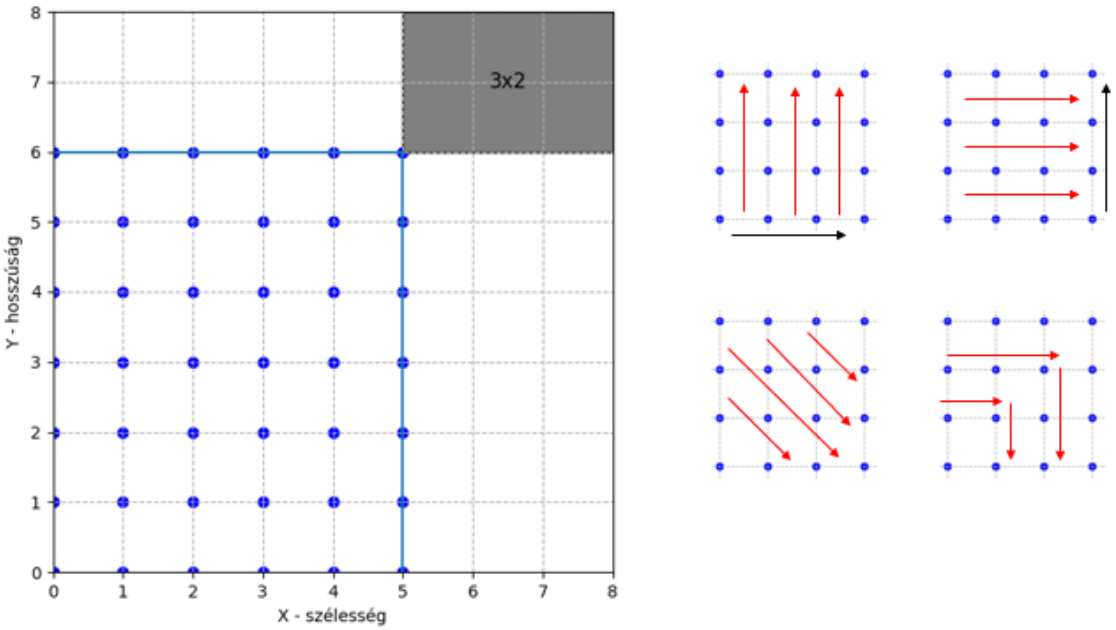
```
class NonGuillotine:
    def __init__(self, width, height):
        self.width           = width
        self.height          = height
        self.placedRectangles = []
        self.maxValue        = 0

    def placeRectangle(self, index, width, height, value):
        for (i, j) in bottomToTopThenLeftToRight(self.width, self.height, width, height):
            canFit = True
            for (_, x, y, w, h) in self.placedRectangles:
                if not(i + width <= x or i >= x + w or j + height <= y or j >= y + h):
                    canFit = False
                    break
            if canFit:
                self.placedRectangles.append((index + 1, i, j, width, height))
                self.maxValue += value
                return True
        return False
```

36. ábra: A guillotine vágástól eltérő vágás implementálása

Az implementálás során (36. ábra) a guillotine vágásra adott mohó algoritmusunkat módosítottuk ott, ahol szükséges volt. Az osztályból eltávolítottuk a szabad helyek feljegyzését, ugyanis a guillotine vágás megkötés eltávolításával csak bonyolultabb lenne kiszámolni és eltárolni őket. A tárgyak elhelyezésére és kiválasztására különféle heurisztikákat alkalmazhatunk, ami lehet akár az első szabad helyre való elhelyezés, amit a példában is felhasználtunk. [41], [43]

Attól függően, hogy melyik pontból kiindulva helyezzük el a tárgyunkat, az elérhető pontok iterálási sorrendjére használt stratégia fogja eldönteni. Az iterálás a bal alsó sarokból, vagyis a (0,0) pontból indul meg és a tárgyat a konténer jobb felső sarkába helyezvén annak a bal alsó koordinátáig tart. Ezáltal a már említett fordított L alakot kapjuk, aminek a szélei mentén levő pontok fogják mutatni azokat a koordinátákat, ahol a tárgy még úgy elhelyezhető, hogy nem lóg ki a konténerből (37. ábra).



37. ábra: A tárgyak lehelyezésének potenciális pontjai és példák a ezek iterációs sorrendjére

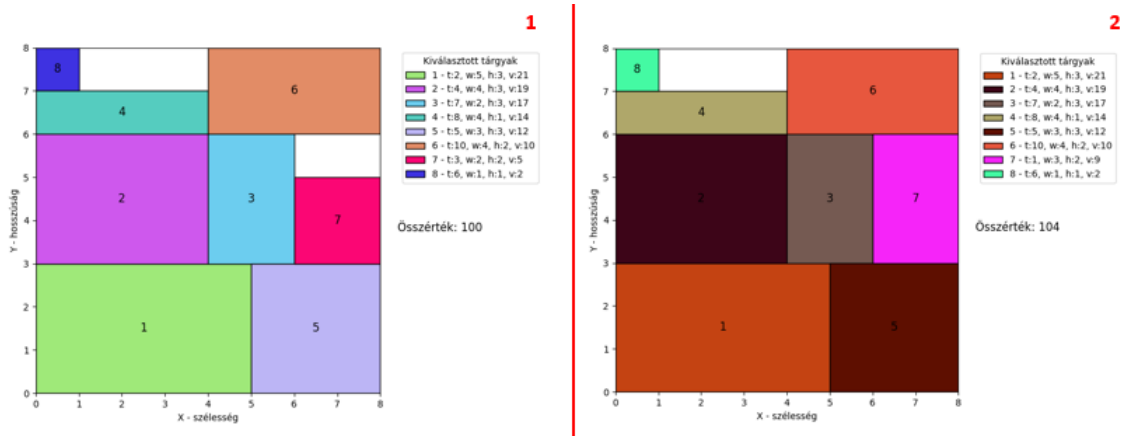
Az iteráció irányára, avagy mely sorrendben ellenőrizzük le a pontokat több fajta lehetőség is van (37. ábra). Maga ez a sorrend jelentős befolyásoló tényező lehet az első szabad helyre történő elhelyezésű heurisztikánál. Az adott pontban lévő tárgy érvényességét ellenőrizni kell, hogy lehelyezhető-e. Ez azt jelenti, hogy a választandó pontra el kell dönteni, hogy az összes behelyezett téglalapoktól rendre balra, jobbra, alatta vagy felette helyezkedik el. A négy pozicionálási lehetőség ellenőrzésével meggyőződhetünk, hogy egyik téglalapban sincs benne, ezáltal a tárgy a konténerbe helyezhető. Végezetül a megoldáshoz használt fő függvény megmaradt, az osztály és egyéb névváltoztatásokat leszámítva teljes mértékben megegyezik a guillotine vágásnál használttal.

```
    def placeRectangle(self, index, width, height, value):
        for (i, j) in bottomToTopThenLeftToRight(self.width, self.height, width, height):
            # ...
def bottomToTopThenLeftToRight(width, height, itemWidth, itemHeight):
    for i in range(0, width - itemWidth + 1):
        for j in range(0, height - itemHeight + 1):
            yield i, j
```

38. ábra: A lentről fel majd balról jobbra való iterációs sorrend implementációja

A tesztelés során első sorban a guillotine vágásnál már bemutatott példán keresztül fogjuk megnézni a heurisztikák által alkotott szabásmintákat és az általuk kapott eredményt. A legjobb eredmény egyikét a kezdetben is bemutatott, azonban nem kifejtett lentről-fel majd balról-jobbra (38. ábra) történő iterációs sorrend adta, aminek

az eredménye a 39. ábrán is látható.



39. ábra: Az első szabad helyre való elhelyezés forgatás nélkül (1) és forgatás megengedésével (2) a lentről fel majd balról jobbra való iterációs sorrend során

A következő iterációs mód a balról-jobbra majd lentről-fel bejárás (40. ábra) ami az előzőnél is használt két ciklus sorrendjének megváltoztatásával megkapható. A fenti példánkban ezáltal 95 összértéket tudtunk elérni, ami nem változott a forgatás megengedésél vagy kizárásával se.

```
    def placeRectangle(self, index, width, height, value):
        for (i, j) in leftToRightThenBottomToTop(self.width, self.height, width, height):
            # ...
def leftToRightThenBottomToTop(width, height, itemWidth, itemHeight):
    for j in range(0, height - itemHeight + 1):
        for i in range(0, width - itemWidth + 1):
            yield i, j
```

40. ábra: A balról-jobbra majd lentről-fel való iterációs sorrend implementációja

A diagonális esetben (41. ábra) előző iterációval hasonlóan forgatás megengedésével és anélkül 95 összértéket értünk el. A kapott iterációs sorrend megkapásához az elérhető (x,y) koordinátákat összegként fogjuk fel. A belső ciklussal az összes érvényes és lehetséges x pontkoordinátán megyünk végig, az y-t pedig az összegből kivonva kapjuk meg. Amennyiben a tárgy elhelyezhető az így kapott ponton úgy, hogy nem lépünk ki a konténerből, akkor visszaadjuk generátorként a koordinátákat. Ez azért jó, mert a generátor számontartja a belső állapotát, így csak egyszer számolja ki az adott értéket és tudja hol kell folytatnia következőleg. Ezáltal a behelyezés ellenőrzésénél az így kapott generátoron kell majd végig iterálnunk és ennek függvényében kell a módosítást elvégezni.

```
def placeRectangle(self, index, width, height, value):
    for (i, j) in diagonalCoordinates(self.width, self.height, width, height):
        # ...
def diagonalCoordinates(width, height, itemWidth, itemHeight):
    for sum in range(0, width + height - itemWidth - itemHeight + 1):
        for x in range(max(0, sum - height + itemHeight), min(sum, width - itemWidth) + 1):
            y = sum - x
            if x + itemWidth <= width and y + itemHeight <= height:
                yield x, y
```

41. ábra: Diagonális iterációs sorrend implementációja

A +/+ koordináták mentén értelmezett spirális esetben ugyan azt az összértéket értük el, valamint ugyan azokat a tárgyakat választottuk, mint a lentről-fel és balról-jobbra követő sorrend esetében. Persze a tárgyak pozicionálása eltérő, azonban a forgatással és anélkül vett esetek is megegyeznek. Az implementációhoz (42. ábra) az Y tengely mentén szintenként tartunk felfele. Minden ilyen során a (0,y) pontból addig megyünk jobbra X tengely mentén, amíg el nem érjük azt a pontot amire x=y, ekkor lefele kell haladnunk (x,0) pontig. Előző módszerhez hasonlóan egy generátort adunk vissza, ha az adott pontra leellenőriztük, hogy a tárgy elhelyezhető-e oda.

```
def placeRectangle(self, index, width, height, value):
    for (i, j) in spiralCoordinates(self.width, self.height, width, height):
        # ...
def spiralCoordinates(width, height, itemWidth, itemHeight):
    level = 0
    while level <= max(width - itemWidth, height - itemHeight):
        x, y = 0, level
        for i in range(0, y):
            if i + itemWidth <= width and y + itemHeight <= height:
                yield i, y
        x = y
        for j in range(y, -1, -1):
            if x + itemWidth <= width and j + itemHeight <= height:
                yield x, j
        level += 1
```

42. ábra: Spirális iterációs sorrend implementációja

Kiértékelés szempontjából ez a módszer optimális értékhez közelit hozott. Futás idő szempontjából *O(nwh),* ahol *n* a tárgyak száma és *w* a konténer szélessége és h a magassága. Ezt úgy kapjuk, ha feltesszük, hogy *wh* pontot kell leellenőrizni minden *n* tárgyra. Memória szempontjából *O(n)* a legrosszabb eset, amennyiben minden tárgyat belehelyeznénk a hátizsákba.

3.2.2. A soron következő szabad pozícióra helyezés (Next-Fit)

Egy másik használható heurisztika a tárgyak elhelyezésére a magasság szerint csökkenő sorrendben, a következő szabad pozícióra helyezés. Hasonlóan ez is a bal alsó sarokból indul, a meglévő konténert pedig a behelyezett tárgyak függvényében szintekre osztja. A tárgyakat megpróbáljuk jobbra egymás mellé berakni addig, amíg oldalt még befférnek a konténerbe. Amennyiben már nem helyezhető több tárgy a sorban, szintet lépünk és ismételjük a folyamatot a sor elejétől. A következő szint pedig onnan fog kezdődni, ahol az előző sorban levő legmagasabb tárgy helyezkedett el.

```
class NonGuillotine:
    def __init__(self, width, height):
        # ...
        self.lastX            = 0
        self.lastY            = 0
        self.rowHeight        = 0

def canFitNew(self, width, height):
    if self.lastX + width <= self.width and self.lastY + height <= self.height:
        return (True, self.lastX, self.lastY)
    elif self.lastY + self.rowHeight + height <= self.height:
        self.lastX = 0
        self.lastY += self.rowHeight
        self.rowHeight = height
        return (True, self.lastX, self.lastY)
    return (False, -1, -1)

def placeRectangle(self, index, width, height, value):
    canFit, x, y = self.canFitNew(width, height)
    if canFit:
        self.placedRectangles.append((index + 1, x, y, width, height))
        self.maxValue += value
        self.lastX    += width
        self.rowHeight = max(self.rowHeight, height)
    return canFit
```

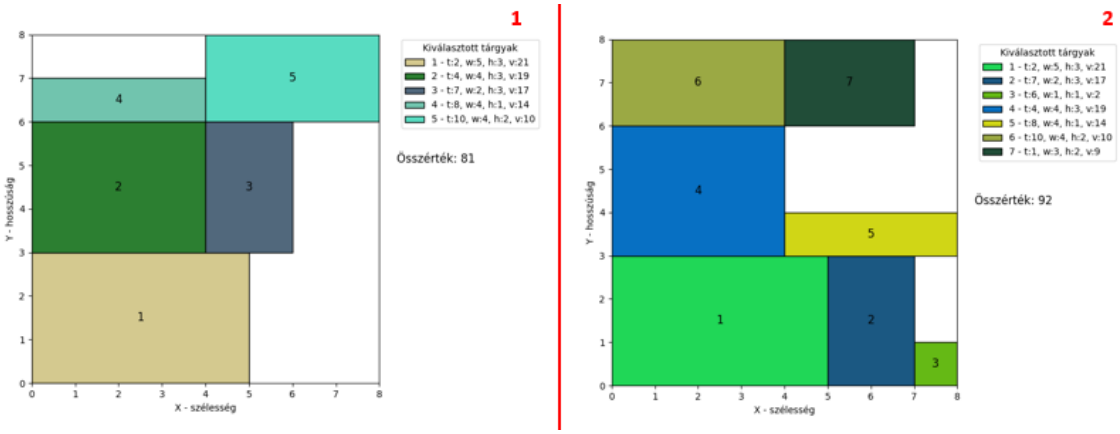
43. ábra: A soron következő szabad pozicióra helyezés heurisztikájának implementálása

Az implementációhoz (43. ábra) 3 új értékre lesz szükségünk, amivel a konstruktort kell bővíteni, ez fogja számontartani a tárgyak letevéséhez szükséges utolsó X és Y pontot, valamint a sor magasságát az új szint meghatározása miatt. Ahhoz, hogy a tárgy beférjen a konténerbe a méreteit, továbbá az adott tengelyen levő utolsó pontot kell ellenőrizni. Amennyiben nem fér be, megnézzük van-e lehetőség új sor kezdésére, ekkor balról, (0, y) pozícióból indulva az előző sor legmagasabb elemével egy vonalban fogjuk pakolni a tárgyakat. Miután egy tárgyat bepakoltunk, frissíteni kell az a következő tárgy lerakásához az x pontot, valamint ellenőrizni kell a tárgy magasabb-e az eddigi szintnél.

Ahogy látható volt a heurisztika könnyen implementálható, viszonylag gyorsan, olcsón és kevés számítással fog eredményt adni. Az az elért eredmény (44. ábra)

azonban lehet távolabb lesz az eddig elért optimális értékhez közelítő megoldásoktól. Eddig csökkenő sorrendben rendeztük érték szerint a tárgyakat, a pakolás során pedig egy bele nemférő nagyobb tárgy esetén új sort kezdtünk. Ezzel azonban sok elérhető szabad helyet veszíthetünk. Az módszer során egy tárgyat, ha nem tudunk lerakni következőre ugrunk. Emiatt a tárgyak rendezés a legköltségesebb művelet, ami által *O(n*log n)* futás időt, az eredmény eltárolására pedig maximum *O(n)* költséget kapunk.

Az elért eredményen javíthatunk, ha minden pontban megpróbálunk mindenképp elhelyezni egy tárgyat. Ha az adott tárgy nem fér bele a helyre, mielőtt új sort kezdenénk megnézzük, hogy a soron következő tárgy befér-e és így tovább. Ezáltal új szintre csak akkor próbálunk meg pakolni, ha az adott sorban már végképp egy tárgy se pakolható le az utolsó pontra. Habár ezzel a hozzáállással növelni tudtunk az elérhető értéken, legrosszabb esetben minden vizsgálandó pontra minden tárgy ellenőrzésre kerül. Pár lépéssel közelebb kerültünk az optimális értékhez, de a futásidő költsége is vele együtt emelkedett, ami legrosszabb esetben $O(n^2)$, ahol n a tárgyak száma. Ez azért, mert minden tárgyra megnézzük, hogy lehelyezhető e a jelenlegi sorba, ha nem, megnézzük a következőre sorra is amig lehetséges.



44. ábra: A soron következő szabad pozicióra helyezés alapeset (1) és javított verzió (2)

3.2.3. Randomizált algoritmus

A randomizált algoritmusának elve a nem guillotine vágás esetében is alkalmazható, azonban a futási idő és komplexitás jóval felülmúlja a guillotine vágását. Az alkalmazandó elv (implementációt lásd 45. ábra) tehát megegyezik: egy helyes megoldást szeretnénk előállítani. Ehhez mohótól eltérő módon, a tárgyakat véletlenszerűen összekeverjük majd az így kapott sorrendben iteráljuk be őket. A guillotine vágásnál a tárgyakat a területileg legkevesebb különbséget adó szabad helyre próbáltuk elhelyezni. Amennyiben nem volt ilyen hely és a behelyezés sikertelen volt, a tárgy forgatásával esetenként megismételtük.

```python
def randomPlacedRectangles(rectangles, containerWidth, containerHeight, maxAttempts):
    placedRectangles, maxValue = [], 0
    random.shuffle(rectangles)
    for (index, (width, height, value)) in rectangles:
        for _ in range(0, maxAttempts):
            i = random.randint(0, containerWidth  - width)
            j = random.randint(0, containerHeight - height)
            canFit = True
            for (_, x, y, w, h) in placedRectangles:
                if not(i + width <= x or i >= x + w or j + height <= y or j >= y + h):
                    canFit = False
                    break
            if canFit:
                placedRectangles.append((index + 1, i, j, width, height))
                maxValue += value
                break
    return placedRectangles, maxValue
```

45. ábra: A randomizált során tárgy elhelyezés a nem guillotine vágás esetében

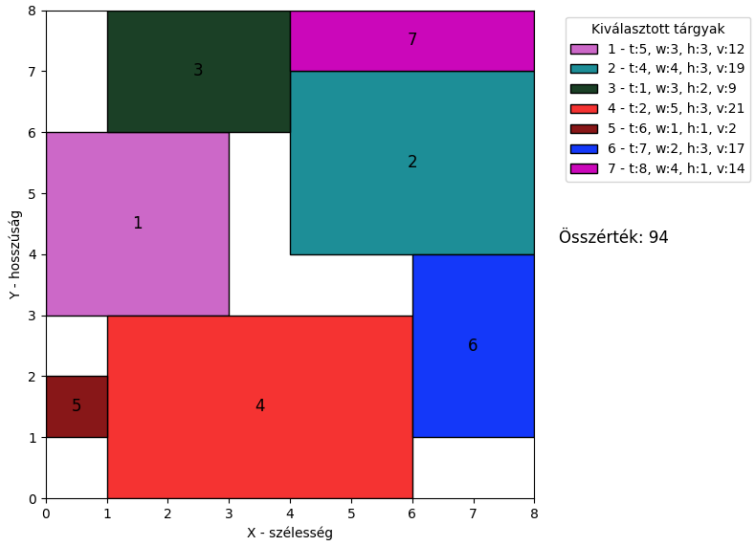
Nem guillotine vágásnál nem tartottuk számon a szabad helyeket. A tárgy elhelyezésére kiválasztott pozíció az összes érvényes pont koordináta közül került ki, amelyeket már láttunk a korábbi példán keresztül. Ezáltal véletlenszerűen próbálunk ezen pontok közül választani, amire majd ellenőrizni kell, hogy szabad-e és nem szerepel egy már lehelyezett téglalapon belül. Továbbá azt is ellenőrizni kell, hogy a pontból lerakva a tárgy nem fog metszeni egy már bepakoltat. Ezeket a már eddig is bemutatott módon fogjuk megvizsgálni. Ha a pont kiválasztás sikeres volt és a tárgy elhelyezhető, akkor beszámítjuk a megoldásba. Egyértelműen, minél későbbi fázisában járunk a pakolásban, annál nehezebb lesz egy olyan pontot találni, ahova az adott tárgy gond nélkül elhelyezhető lenne. A próbálgatások jelentős terhelést jelenthetnek, ami folyamatosan csak növekszik. Ebből kifolyólag szükséges egy felső határt szabni arra, hogy hányszor próbáljuk meg a tárgyat elhelyezni.

A fő függvény (46. ábra) is szinte megegyező a guillotine vágással, a megadott iteráció számig gyártjuk a helyes megoldásokat. Az eltelt idő alatt talált legjobb összértékkel rendelkező elrendezést pedig megtartjuk.

```
def nonGuillotineRandom(rectangles, containerWidth, containerHeight, maxIteration=100, maxAttempts=50):
    placedRectangles, maxValue = [], -1
    rectangles = [(i, rectangle) for i, rectangle in enumerate(rectangles)]
    for _ in range(0, maxIteration):
        tempPlacedRectangles, tempMaxValue = randomPlacedRectangles(rectangles, containerWidth, containerHeight, maxAttempts)
        if tempMaxValue > maxValue:
            placedRectangles, maxValue = tempPlacedRectangles, tempMaxValue
    return placedRectangles, maxValue
```

46. ábra: A randomizált algoritmus fő függvényének implementálása

Kiértékelés szempontjából (47. ábra) alapértelmezett maximum 100 iterációból, tárgyak bepakolása során legfeljebb 50 próbálkozás értékekkel futtattuk a már bemutatott inputon. Harmadik futtatásra 94 összértéket kaptunk. Ez az érték hasonlóan a maximális iteráció és próbálkozás szám növelésével lehetségesen javítható. Futás idő szempontjából *O(Inhm)*, ahol *I* a maximális iterációk száma, *n* a tárgyaink száma és *h* a már lehelyezett tárgyak száma *p* pedig a próbálkozások száma. Memória szempontjából ugyancsak *O(n)* lesz, ugyanis csak az eredményt tároljuk el.



47. ábra: A randomizált algoritmus által kapott elrendezés

3.3. Összehasonlítás és kiértékelés

A mohó és randomizált algoritmusok egyszerűségének köszönhetően könnyen át lehet tekinteni a két vágási módszer közötti különbségeket. Láthattuk, hogy implementálás szempontjából a guillotine vágás megkötés jelentősen csökkentette a költségeket a szabad helyek felosztását tekintve. Továbbá a szabad helyek bal alsó sarkába való elhelyezéssel nagy mértékben sikerült redukálni a problémát. Nem guillotine vágás esetében minden lehetséges pont leellenőrzése sokkal költségesebbé tette a feladatot.

Összehasonlításképpen nézzük meg újra a bemutatott módszerek által elért futásidőt és memória komplexitást (8. táblázat). Mivel mohó és randomizált algoritmusokról beszélünk a memória igényünk minden esetben meg fog egyezni. Guillotine vágás esetében a szabad helyeket, valamint a már elhelyezett tárgyakat tároltuk. Nem guillotine esetében pontjaink vannak, amiket nem tárolunk így itt csak az elhelyezett tárgyakat tároljuk, ami által a két módszer komplexitása megegyező.

Sorszám	Módszer	Futás idő	Memória
1	Guillotine mohó	*O(nh)*	*O(n)*
2	Guillotine randomizált	*O(Inh)*	*O(n)*
3	Nem guillotine E.SZ.H.V.E.	*O(nwh)*	*O(n)*
4	Nem guillotine S.K.SZ.P.H.	*O(n*log n)*	*O(n)*
5	Nem guillotine S.K.SZ.P.H. javított	$O(n^2)$	*O(n)*
6	Nem guillotine randomizált	*O(Inhm)*	*O(n)*

8. táblázat: A bemutatott guillotine és nem guillotine vágásos módszerek futás idő és memória komplexitása

A mohó guillotine vágás során elhelyezés tekintetéből a bal alsó sarkot választottuk. A szabad hely kiválasztására megnéztük a Best-Fit és Worst-Fit stratégiákat. A vágás irányán alapuló döntésünket 5 hozzáállás alapján hoztuk meg:

- a behelyezés után a fentmaradó hely kisebb különbséget adó oldallal párhuzamos vágás
- csak vízszintes vágások
- csak függőleges vágások
- a tárgy nagyobb oldalával (mérettel) párhuzamos vágás

- a tárgy nagyobb oldalával (mérettel) merőleges vágás

A nem guillotine vágás során csakis az elhelyezésre nézhettünk heurisztikákat, tekintve, hogy nincs vágás vagy szabad hely kiválasztás. Az elhelyezés megtalálására való heurisztikák alkalmazása során a First-Fit által tanulmányoztuk az iterációs sorrend 4 fajta bejárásával az eredmények változását. Továbbá a Next-Fit herusztika és annak a javított verziójára is láthattunk példát.

A tesztek alapján elmondható, hogy egy bizonyos eredmény több módszer mentén is elérhető. Általánosan mondhatnánk, hogy a csak vízszintes vágások alkalmazása jó ötlet akkor, amikor a halmazban levő tárgyak többsége nagyobb szélességgel rendelkezik. Ugyan ez az eredmény elérhető általában akkor is, ha csak akkor vágunk szélesség mentén, ha az nagyobb, mint a magasság. A két vágási feltételt (vízszintes vágás kisebb szélesség esetén) megcserélve hasonló mondható el eredményeiben összemérve a csak függőleges vágásokról.

t	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
w	2	2	5	2	3	3	2	4	2	3	3	4	2	5	5
h	4	2	3	3	4	5	2	2	4	5	4	1	5	1	1
v	10	23	13	16	6	17	25	11	7	13	1	9	1	12	4

9. táblázat: A tárgyak szélessége, magassága és értéke a bemutatandó példához

Egy véletlenszerű példán keresztül megnézzük az összes eddigi lehetséges eljárás kombinációját. A példában (9. táblázat) 15 tárgyat próbálunk meg elhelyezni egy 10x10-es konténerben. A tárgyak szélessége és hossza [1,5] értéke [1,25] intervallumból került ki. Az eredmény a következő oldalon szereplő 10. táblázaton is látható, az algoritmus nevekben használt rövidítés G – guillotine, N. G. – nem guillotine -ra utal. A módszernevek a már bemutatott, illetve a tesztelés során használt függvény nevekre utal.

Sorszám	Algoritmus	Módszer	Forgatás	Összérték
1	N. G. First-Fit	bottomToTopThenLeftToRight	Igen	160
2	G. Best-Fit	nagyobbFentmaradoHely	Igen	156
3	G. Best-Fit	merettelParhuzamos	Igen	156
4	G. Best-Fit	csakFuggoleges	Igen	156
5	N. G. First-Fit	diagonalCoordinates	Igen	156
6	G. Best-Fit	merettelMeroleges	Igen	153
7	G. Best-Fit	csakVizszintes	Igen	153
8	N. G. First-Fit	leftToRightThenBottomToTop	Nem	153
9	N. G. First-Fit	leftToRightThenBottomToTop	Igen	153
10	N. G. First-Fit	spiralCoordinates	Nem	153
11	N. G. First-Fit	spiralCoordinates	Igen	153
12	G. Random	nagyobbFentmaradoHely	Igen	150
13	G. Best-Fit	nagyobbFentmaradoHely	Nem	147
14	G. Worst-Fit	nagyobbFentmaradoHely	Igen	147
15	G. Best-Fit	merettelParhuzamos	Nem	147
16	G. Worst-Fit	merettelParhuzamos	Igen	147
17	G. Worst-Fit	csakVizszintes	Igen	147
18	G. Best-Fit	csakFuggoleges	Nem	147
19	G. Worst-Fit	csakFuggoleges	Igen	147
20	N. G. First-Fit	diagonalCoordinates	Nem	147
21	G. Worst-Fit	merettelMeroleges	Nem	143
22	G. Worst-Fit	merettelMeroleges	Igen	143
23	G. Random	nagyobbFentmaradoHely	Nem	143
24	N. G. Random	-	Nem	143
25	G. Best-Fit	merettelMeroleges	Nem	139
26	G. Best-Fit	csakVizszintes	Nem	139
27	G. Worst-Fit	csakVizszintes	Nem	130
28	N. G. Next-Fit 1	-	Nem	117
29	N. G. Next-Fit 2	-	Nem	117
30	G. Worst-Fit	nagyobbFentmaradoHely	Nem	112
31	G. Worst-Fit	merettelParhuzamos	Nem	112
32	G. Worst-Fit	csakFuggoleges	Nem	112

10. A különböző kombinációk által kapott eredmények a példa esetében

A két vágás randomizált algoritmusát egymás mellé téve látható igazán a különbség. Ugyan azokat a lépéseket követjük mindkét esetben, a tárgyak egyedi elhelyezési logikáját kivéve, amiből is látszik a guillotine korlátok:

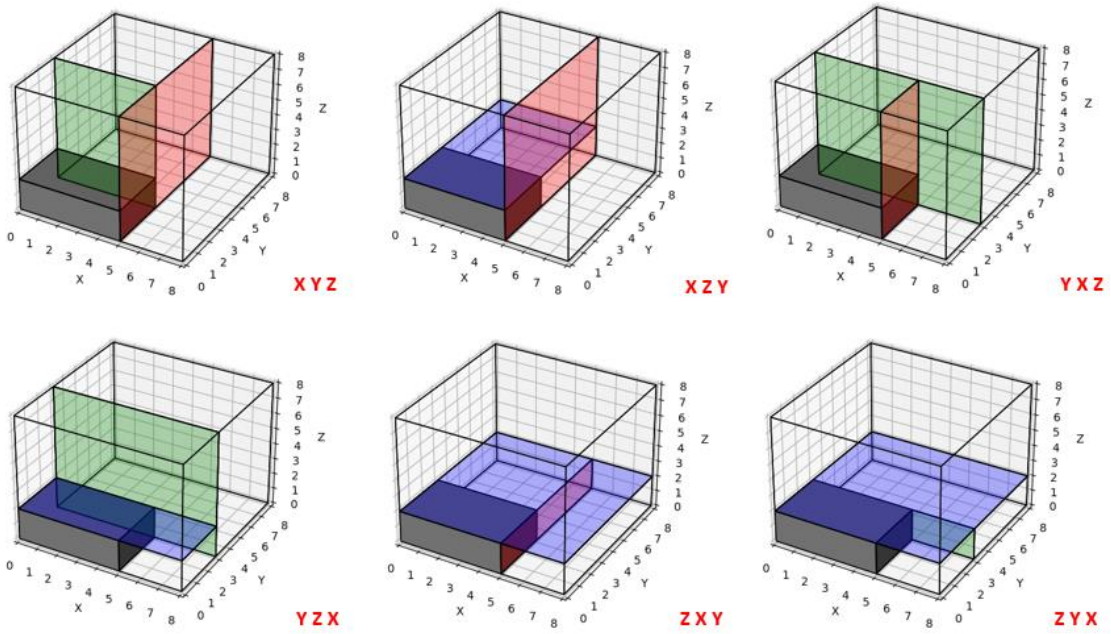
1. tárgyak felsorolása, hogy a keverés után is megmaradjon az eredeti index
2. a megadott korlátig való iterálás
3. tárgyak megkeverése
4. egyedi: tárgy elhelyezése
5. maximum érték kiválasztása

Guillotine vágás során ugyanis a szabad területekből maximum n+1 lehet abban az esetben, ha minden tárgyat beleteszünk a hátizsákba és a konténerben van még hely. Ilyenkor az adott üresen álló partíciót eltávolítjuk és a tárgy bal alsó sarokba való elhelyezése után két részre osztjuk. Ezáltal maximum n+1 esetre kell megnézni, hogy egy pontból a kiinduló két vonal (tárgy méretei) nem lépik-e át az elhelyezendő hely határait. Nem guillotine vágásnál ahogy láthattuk, ha a tárgyaink természetes számú értéket vesznek fel akkor a konténer méreteinek és a tárgy méreteinek különbségének a szorzata adja meg a 37. ábrán látott L alak alatti pontokat. Ezen pontok közül a véletlen generált pontot minden lehelyezett téglalapra ellenőrizni kell, hogy nem fog-e ütközni a 4 oldal mentén. A már mennyiségük miatt is és minél később járunk a tárgyak elhelyezésében annál nehezebb véletlenszerűen egy pontot úgy lehelyezni, hogy az ne ütközőn semmivel. Habár korlátot adtunk erre, érzékelhető, hogy a guillotine vágásnak sokkal könnyebb dolga van a randomizált módszer keretein belül. Emiatt kijelenthető, hogy ugyan azon iterációs szám mellett és a véletlenszerű esélyt is beszámítva, jelentősen több esélyünk van az optimális eredmény megközelítésére. Annak ellenére, hogy bizonyos esetekben csak a nem guillotine vágás által érhetjük el az optimális megoldást, az járhat olyan erőforrás igénnyel, hogy a módszer már nem éri meg.

4. 3D hátizsák probléma

A 3-dimenziós hátizsák probléma a 2-dimenziós eset kiterjesztése egy plusz tengely mentén, mely a problémát szabásminták kivágásáról sokkal inkább eltereli a már mindennapi életben használt pakolás fele. A hátizsákot egy téglalap helyett most egy téglatestként kell értelmezni, amit elképzelhetünk úgy, mint egy dobozt. Célunk, hogy csomagokat pakoljunk bele ebbe a dobozba, úgy, hogy a tárgyak egy olyan elrendezését kapjuk, amelyre az összegyűjthető érték maximális.

A guillotine vágás elve ugyan úgy alkalmazható ebben a speciális esetben is, mivel jelentősen leegyszerűsítheti a problémát a lehetséges vágások korlátozásával. A megkötés hasonlóan megmarad, tehát széltől szélig terjedő vágásokat hajtunk végre egy bizonyos sorrendet követve. Míg a két-dimenziós esetben a két tengely mentén 2 lehetséges vágás volt alkalmazható, csak aközött kellett mérlegelni, hogy függőleges vagy vízszintessel párhuzamosat alkalmazzunk. Először is a szabad helyet széltől-szélig ketté metszettük a tárgy valamely oldalából kiindulva, majd ugyanezt a vágást ismételve merőlegesen a tárgy vonalai mentén. Ez két szabad területet eredményezett, azonba három-dimenziós esetben bonyolódnak a lehetőségek: a vágás itt két tengely által közre zárt téglalapnak fog megfelelni. A három-dimenziós eset miatt a tárgyunk körbe vágásához a 3 tengely mentén 3 vágásra lesz szükségünk, ami által 6 lehetséges esetünk lesz a lehetséges vágások tekintetében (48. ábra).



48. ábra: A lehetséges guillotine vágások a 3-dimenziós esetben

Egy tengely mentén való vágást most úgy fogunk értelmezni, hogy rögzítjük az adott tengely pont koordinátájának értékét, majd a másik két tengely mentén növeljük az értékeket. Például ha a P(x,y,z) pont koordinátájában az x értéke 5, akkor az így keletkezett téglalapot az egyszerűség kedvéért X vágásnak fogjuk nevezni. A szabad helyek felosztása során az alkalmazandó vágások sorrendjét az adott tengelyt jelölő vágás betűinek a felsorolásával fogjuk megkapni (például 48. ábra).

```
class Guillotine3D:
    def __init__(self, width, height, depth):
        self.width         = width
        self.height        = height
        self.depth         = depth
        self.freeSpaces    = [(0, 0, 0, width, height, depth)]
        self.placedCuboids = []
        self.maxValue      = 0
```

49. ábra: A 2D osztály bővítése a 3D feladat megoldásához

A három-dimenziós feladat megoldására a két-dimenziós feladat mohó algoritmusának általánosítását szeretnénk alkalmazni. Ehhez az alap osztályt megtartjuk, csak bővítjük egy mélység dimenzióval (49. ábra), amit majd a szabad helyek inicializálásánál is számításba veszünk.

```
def chooseCuttingOrder(self, width, height, depth):
    dimensions = [('X', width), ('Y', height), ('Z', depth)]
    dimensions.sort(key=lambda x: x[1], reverse=True)
    cuttingOrder = dimensions[0][0] + dimensions[1][0] + dimensions[2][0]
    return cuttingOrder

def placeCuboid(self, index, width, height, depth, value):
    self.freeSpaces.sort(key=lambda space: space[3] * space[4] * space[5])
    for i, (x, y, z, w, h, d) in enumerate(self.freeSpaces):
        if width <= w and height <= h and depth <= d:
            self.placedCuboids.append((index+1, x, y, z, width, height, depth))
            self.freeSpaces.pop(i)
            self.maxValue += value

            cuttingOrder = self.chooseCuttingOrder(width, height, depth)
            self.addFreeSpaces(cuttingOrder, x, y, z, w, h, d, width, height, depth)
            return True
    return False
```

50. ábra: A vágási technika kiválasztása valamint egy téglatest elhelyezése a térben

A téglatest lehelyezése során (50. ábra) hasonlóan először sorba rendezzük a szabad helyeket a bővített térfogat alapján. Mielőtt bele helyeznénk a tárgyat, le kell ellenőriznünk mindhárom tengely mentén, hogy belefér-e. Ami eltér a két-dimenziós

esettől, hogy itt a vágások sorrendjének kialakítása a hosszabb tengelyek mentén való metszésekre alapul. Minden tárgy esetében csökkenő sorendbe helyezzük az adott tárgy méreteit, majd a vágásokat az így kapott sorrendben, ami az adott tengelynek megfelel, azonos vágási sorrendben kerül alkalmazásra. A 51. ábrán pedig a szabad helyek meghatározása következik az adott vágási sorozat figyelembevétele után.

```
def addFreeSpaces(self, cuttingOrder, x, y, z, w, h, d, width, height, depth):
    if cuttingOrder == "XYZ":
        self.freeSpaces.append((x + width, y         , z         , w - width, h         , d))
        self.freeSpaces.append((x        , y + height, z         , width    , h - height, d))
        self.freeSpaces.append((x        , y         , z + depth, width    , height    , d - depth))
    elif cuttingOrder == "XZY":
        self.freeSpaces.append((x + width, y         , z         , w - width, h         , d))
        self.freeSpaces.append((x        , y         , z + depth, width    , h         , d - depth))
        self.freeSpaces.append((x        , y + height, z         , width    , h - height, depth))
    elif cuttingOrder == "YXZ":
        self.freeSpaces.append((x + width, y         , z         , w - width, height    , d))
        self.freeSpaces.append((x        , y + height, z         , w        , h - height, d))
        self.freeSpaces.append((x        , y         , z + depth, width    , height    , d - depth))
    elif cuttingOrder == "YZX":
        self.freeSpaces.append((x + width, y         , z         , w - width, height    , depth))
        self.freeSpaces.append((x        , y + height, z         , w        , h - height, d))
        self.freeSpaces.append((x        , y         , z + depth, w        , height    , d - depth))
    elif cuttingOrder == "ZXY":
        self.freeSpaces.append((x + width, y         , z         , w - width, h         , depth))
        self.freeSpaces.append((x        , y + height, z         , width    , h - height, depth))
        self.freeSpaces.append((x        , y         , z + depth, w        , h         , d - depth))
    elif cuttingOrder == "ZYX":
        self.freeSpaces.append((x + width, y         , z         , w - width, height    , depth))
        self.freeSpaces.append((x        , y + height, z         , w        , h - height, depth))
        self.freeSpaces.append((x        , y         , z + depth, w        , h         , d - depth))
```

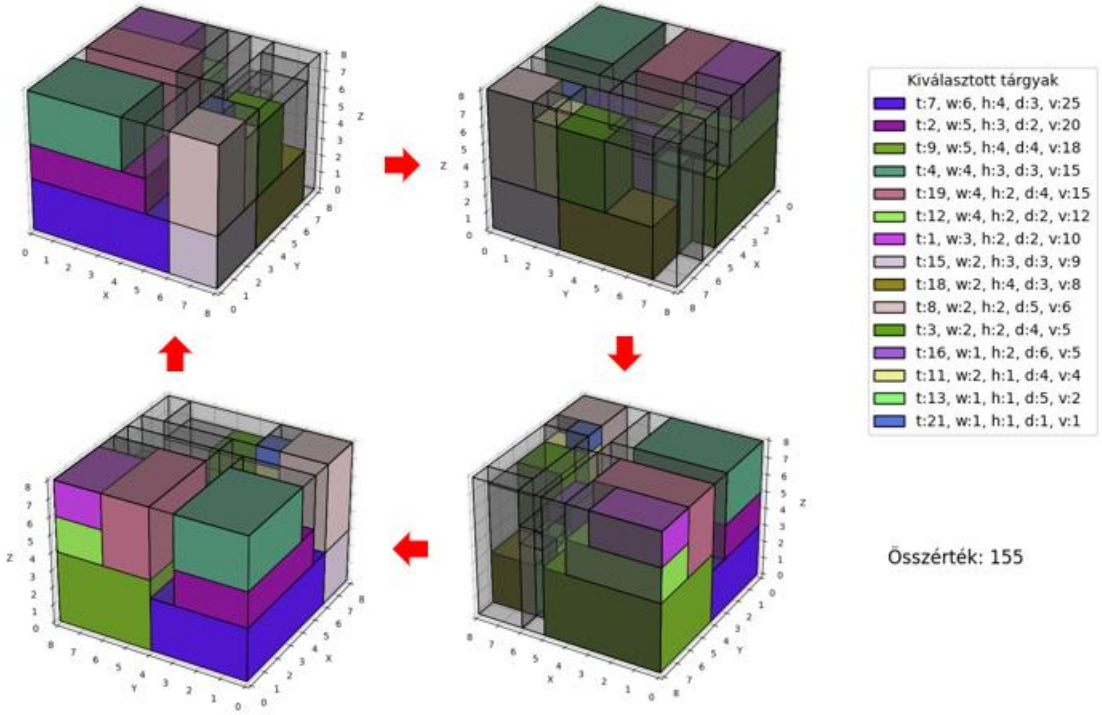
51. ábra: A lehetséges guillotine vágások sorrend szerinti felsorolása

Az megoldásra meghívandó függvény (52. ábra) is a két-dimenziós megoldáson alapul: érték szerint sorba rendezzük a téglatesteket, majd ebben a sorrendben szeretnénk elhelyezni őket. A tárgyak forgatását ebben az esetben kizártuk, mivel az jelentős mértékű költség növekedéssel járna, és még komplexebbé tenné az egy dimenzióval bővített alapból is komplex problémát. Végeredményül elmondható, hogy a vágások és szabad helyek felosztásának az újra gondolásán kívül, az algoritmus többi része egy kis bővítéssel lényegesen alapozható volt a két-dimenziós megoldásra.

```
def guillotineCut3D(cuboids, containerWidth, containerHeight, containerDepth):
    guillotine = Guillotine3D(containerWidth, containerHeight, containerDepth)
    sortedCuboids = sorted(enumerate(cuboids), key=lambda value: value[1][3], reverse=True)
    for index, (width, height, depth, value) in sortedCuboids:
        guillotine.placeCuboid(index, width, height, depth, value)
    return guillotine.placedCuboids, guillotine.maxValue
```

52. ábra: A fő algoritmus implementálása a 3D esetben

Az elért eredményünket grafikusan is ábrázolhatjuk a három-dimenziós térben, azonban a két-dimenziós esettől eltérően sokkal nehezebb átlátható diagramot kapunk. Ahogy az 53. ábrán is látható, a teszt halmazban 21 tárgyból 15 tárgyat sikerült behelyezni forgatás nélkül. Az áttetsző részek a vágások során keletkezett szabad helyeket reprezentáljak, amelyek a vágásoknak megfelelnek. Elmondhatjuk tehát, hogy sikerült egy 2D hátizsák problémát megoldó mohó algoritmust úgy általánosítani, hogy az egy megoldással tudjon szolgálni a 3D esetre is. Azonban mindenképp megjegyzendő, hogy mohó algoritmus lévén az optimális eredmény megtalálása nem garantált.



53. ábra: A 3D hátizsák guillotine vágásokkal, 4 nézetből 90° -os oldal forgatásokkal

5. Összegzés

A diplomamunka során áttekintettük a hátizsák probléma 1-,2- valamint 3-dimenziós eseteit.

1-dimenziós esetben 9 módszert sikerült megvizsgálni, amelyeket implementálva 10 algoritmust kaptunk, amit teljesítmény ügyileg össze tudtunk hasonlítani. A tesztesetek alapján összeségében elmondható, hogy a hátizsák probléma esetében a brute force módszer teljesített a legrosszabbul, használata csak végső esetben javasolt. Az oszd meg és uralkodj módszer sem volt az igazi a probléma természetéből eredendően, ugyanis a hátizsák esetében összefüggő részproblémáink vannak. Szimplán két esetre való mérlegelés és szétválasztás nem hozza meg a kellő nyereséget, ehhez hasonlóan teljesített a visszalépéses módszer is. A mohó módszer elég jó eredményeket mutatott, néhány speciális esetben felül múlva az összes többit is. A továbbfejlesztése a visszalépéses keresés 1 lépéssel való visszalépése által sajnos nem bizonyult helyzet javítónak. A feljegyzéses és dinamikus programozás módszer elméleti futás időben a legjobbnak mutatkozó módszerek a hátizsák probléma optimális megoldásának megtalálására. Gyakorlatban azonban a kiválasztott stratégiák függvényében a branch and bound módszer jobbnak ígérkezett. A randomizált módszer és genetikus módszerek esetében pedig láthattuk, hogy finomhangolással képesek elég jó közelítő értékekkel szolgálni. A [2] számú kutatásban a dinamikus és genetikus módszerek voltak feltüntetve, mint ajánlott stratégia a hátizsák probléma megoldására. A kapacitás és tárgyak növelése, a véletlenszerű esetek által az elvégzett tesztek és összehasonlítás alapján más következtetésre juthatunk. Az eredmények szerint gyakorlatban általános esetekben a branch and bound és mohó módszerek használata javasolt.

A 2-dimenziós esetben láthattuk a guillotine és nem guillotine vágásos technikák közötti lényeges főbb különbséget. Egy tárolóba pakolás algoritmusból kiindulva készíteni tudtunk a vágás technikák elveinek megfelelő mohó és randomizált algoritmusokat. Ezek a módszerek, habár nem garantálják az optimális megoldás megtalálását, azonban egyszerűek, átláthatóak és relatív gyorsak. Az input függvényében, valamint a heurisztikák alkalmazása által ebben az esetben is bemutattuk a lehetőséget egy megoldás végig vezetése során a javításra.

A 3-dimenziós esetre, habár nem került nagy hangsúly, egy minimális betekintést kaphattunk róla. A guillotine vágások beazonosítása után láthattuk, hogy mennyivel is komplexebbé válik a helyzet a dimenziók bővítése során már a vágási technika alkalmazásával is. A 2-dimenziós esetnél is felhasznált algoritmust sikeresen tudtuk úgy általánosítani, hogy az egy megoldással tudjon szolgálni a probléma bővített változatára is.

6. Irodalomjegyzék

[1] Knapsack problem [Online] (elérés dátuma: 2023.03.01) https://en.wikipedia.org/wiki/Knapsack_problem

[2] Maya Hristakeva, Dipti Shrestha: Different Approaches to Solve the 0/1 Knapsack Problem, Corpus ID: 8985789, 2005, [15]

[3] Knapsack - brute force algorithm [Online] (elérés dátuma: 2023.03.28) https://stackoverflow.com/questions/29669259/knapsack-brute-force-algorithm

[4] Algoritmusok és adatszerkezetek gyakorlat - Oszd meg és uralkodj [Online] (elérés dátuma: 2023.03.29) https://docplayer.hu/160397378-Algoritmusok-es-adatszerkezetek-gyakorlat-03-oszd-meg-es-uralkodj-nagy.html

[5] Optimalizációs stratégiák 1. [Online] (elérés dátuma: 2023.03.29) https://users.nik.uni-obuda.hu/sztf2/Optimalizacio1.pdf

[6] 0/1 Knapsack Problem [Online] (elérés dátuma: 2023.03.13) https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

[7] 0/1 Knapsack Problem [Online] (elérés dátuma: 2023.03.17) https://www.scaler.com/topics/knapsack-problem/

[8] 01 Knapsack using Memoization | Concept of Memoization [Online] (elérés dátuma: 2023.03.18) https://www.youtube.com/watch?v=dT6dvdbpChA&ab_channel=Techdose

[9] How to 0/1 Knapsack Problem using Memoization in Python and C++ [Online] (elérés dátuma: 2023.03.18) https://www.educative.io/answers/how-to-0-1-knapsack-problem-using-memoization-in-python-and-cpp

[10] Difference Between Divide and Conquer and Dynamic Programming [Online] (elérés dátuma: 2023.03.19) https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

[11] Válogatás az ELTE IK programtervező informatikus MSc szak Algoritmusok tervezése és elemzése és Speciális algoritmusok tantárgyainak anyagából [Online] (elérés dátuma: 2023.03.20)

https://people.inf.elte.hu/szabolaszlo/HaladoAlgoritmusok.pdf

[12] 0/1 Knapsack problem [Online] (elérés dátuma: 2023.03.21) https://www.javatpoint.com/0-1-knapsack-problem

[13] 0/1 Knapsack Problem Fix using Dynamic Programming Example [Online] (elérés dátuma: 2023.03.13) https://www.guru99.com/knapsack-problem-dynamic-programming.html

[14] Binary Knapsack Problem using Greedy Algorithm [Online] (elérés dátuma: 2023.03.25) https://codecrucks.com/binary-knapsack-problem-using-greedy-algorithm/

[15] Ősz Edina: Egészértékű programozási feladatok megoldásának gyorsítása heurisztikus módszerek segítségével, 2013, [55]

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms Third Edition, The MIT Press, 2009, [1313], ISBN-978-0-262-03384-8

[17] Steven S. Skiena: The Algorithm Design Manual Second Edition, Springer, 2008, [739], ISBN: 978-1-84800-069-8

[18] Implement 0/1 knapsack problem in python using backtracking [Online] (elérés dátuma: 2023.03.27) https://stackoverflow.com/questions/69878657/implement-0-1-knapsack-problem-in-python-using-backtracking

[19] 01 Knapsack Algorithm – Recursion – Backtrack – Algo [Online] (elérés dátuma: 2023.03.27) https://inlovewithcode.wordpress.com/2015/08/04/knapsnack/

[20] Elágazás és korlátozás [Online] (elérés dátuma: 2023.04.05) https://hu.wikipedia.org/wiki/El%C3%A1gaz%C3%A1s_%C3%A9s_korl%C3%A1toz%C3%A1sFgh

[21] Branch and bound: Method, knapsack problem [Online] (elérés dátuma: 2023.04.07) https://ocw.mit.edu/courses/1-204-computer-algorithms-in-systems-engineering-spring-2010/df7362cc2e2d15ddb9eff4a0e37ef88b_MIT1_204S10_lec16.pdf

[22] 0/1 Knapsack using Least Cost Branch and Bound [Online] (elérés dátuma: 2023.04.09)

https://www.geeksforgeeks.org/0-1-knapsack-using-least-count-branch-and-bound/

[23] 0/1 Knapsack using Branch and Bound [Online] (elérés dátuma: 2023.04.09) https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/

[24] Implementation of 0/1 Knapsack using Branch and Bound [Online] (elérés dátuma: 2023.04.09) https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/

[25] Christos H. Papadimitriou, Kenneth Steiglitz: Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, 1982, [487], ISBN: 0-13-152462-3

[26] S. Martello, P. Toth: Knapsack Problems: Algorithms and Computer Implementation, John Willey & Sons Ltd., 1990. [304], ISBN: 0-471-92420-2

[27] How to Solve the Knapsack Problem Using Genetic Algorithm in Python [Online] (elérés dátuma: 2023.04.19) https://plainenglish.io/blog/genetic-algorithm-in-python-101-da1687d3339b

[28] Genetic Programming in Python: The Knapsack Problem [Online] (elérés dátuma: 2023.04.17) https://www.kdnuggets.com/2023/01/knapsack-problem-genetic-programming-python.html

[29] Genetic Algorithm to solve the Knapsack Problem [Online] (elérés dátuma: 2023.04.22) https://arpitbhayani.me/blogs/genetic-knapsack

[30] Genetikus algoritmusok [Online] (elérés dátuma: 2023.04.21) https://www.ms.sapientia.ro/~kasa/prog12ab.pdf

[31] Randomized algorithm [Online] (elérés dátuma: 2023.04.23) https://en.wikipedia.org/wiki/Randomized_algorithm

[32] Randomized Algorithms | Set 2 (Classification and Applications) [Online] (elérés dátuma: 2023.04.23) https://www.geeksforgeeks.org/randomized-algorithms-set-2-classification-and-applications/

[33] Ali Asghar Tofighian, B. Naderi: Modeling and solving the project selection and scheduling, Computers & Industrial Engineering, Volume 83, May 2015, [30-38]

[34] Mohammad Dolatabadi, Andrea Lodi, Michele Monaci: Exact algorithms for the two-dimensional guillotine knapsack, Computers & Operations Research, Volume 39, Issue 1, January 2012, [48-53]

[35] Abdelghani Bekrar, Imed Kacem: An Exact Method for the 2D Guillotine Strip Packing Problem, Advances in Operations Research, Volume 2009, January 2009, Article ID 732010, [20]

[36] Alberto Caprara, Michele Monaci: On the two-dimensional Knapsack Problem, Operations Research Letters, Volume 32, Issue 1, January 2004, [5-14]

[37] Arindam Khan, Aditya Lonkar, Arnab Maiti, Amatya Sharma, Andreas Wiese: Tight Approximation Algorithms for Two-Dimensional Guillotine Strip Packing, International Colloquium on Automata, Languages and Programming, Corpus ID: 246822551, 12 February 2022, [32]

[38] Christofides N., Whitlock, C.: An algorithm for two-dimensional cutting problems. Operations Research, Volume 25, Issue 1, February 1977, [30-44]

[39] Silvano Martello, Michele Monaci, Daniele Vigo: An Exact Approach to the Strip-Packing Problem, Informs Journal on Computing, Volume 15, Issue 3, August 2003, [12]

[40] Mateus Martin, Horacio Hideki Yanasse, Maristela O. Santos, Reinaldo Morabito: Models for two- and three-stage two-dimensional cutting stock problems with a limited number of open stacks, International Journal of Production Research, Volume 61, Issue 9, April 2022, [31]

[41] Jukka Jylänki: A Thousand Ways to Pack the Bin – A Practical Approach to Two-Dimensional Rectangle Bin Packing, 27 February 2010, [50]

[42] Eleni Hadjiconstantinou, Nicos Christofides: An exact algorithm for general, orthogonal, two-dimensional knapsack problems, European Journal of Operational Research, Volume 83, 1995, [39-56]

[43] Andrea Lodi, Silvano Martello, Michele Monaci: Two-dimensional packing problems: A survey, European Journal of Operational Research, Volume 141, Issue 2, 1 September 2002, Pages [241-252]