# Bin Packing

Biști Ștefan and Rădulescu Andrei-Valentin

Universitatea Politehnica, Bucureşti, România

**Abstract.** Problema Bin Packing reprezintă o provocare fundamentală în optimizarea combinatorie, având aplicații vaste în logistică și alocarea resurselor. În această lucrare, analizăm performanța a patru abordări distincte: Brute-force, First Fit Descending (FFD), Modified FFD și Column Generation. Demonstrăm natura NP-Hard a problemei prin reducere de la 3-SAT și evaluăm algoritmii pe seturi de date variate. Rezultatele noastre indică faptul că, deși Column Generation este robust teoretic, euristicile de tip FFD oferă un echilibru superior între precizie și eficiență computațională pentru instanțele de dimensiuni mici și medii.

**Keywords:** Bin Packing · NP-Hard · Column Generation · Euristici · Optimizare.

## 1 Introducere

### 1.1 Descrierea problemei

Problema Bin Packing este o problemă algoritmică clasică, studiată încă din anii 1970[1]. Aceasta presupune "împachetarea" unor obiecte de diferite mărimi în cât mai puține "cutii". Are mai multe dimensiuni precum: Online bin packing unde setul de date nu e cunoscut în avans, Variable-sized bin packing - cutiile au capacități diferite, Multi-dimensional bin packing - obiectele au mai multe dimensiuni(nu doar lungime) și Offline bin packing, problema de interes pentru acest studiu, unde toate obiectele sunt cunoscute în prealabil și au o singură dimensiune, iar cutiile au dimensiune fixă.

Formalizând aceasta definițe: Fie un număr infinit de recipiente de capacitate $C \in Z^+$ și un set de obiecte de diferite dimensiuni $N =_i |i \leq n, w_i \in N\}$.

Care este numărul minim K de submulțimi $S = \{w_j\}; S \subset N$ astfel încât $\sum w_j \leq C$ ?

### 1.2 Istoric

Problema aceasta a fost formulată matematic cu mult înainte de apariția ei în Computer Science. Primele sale rădăcini au apărut în anii 40' prin Cuttin Stock Problem, care presupune minimizarea risipei atunci când vrem sa împărțim o bucată de material în mai multe bucăți de dimensiuni specificate. Aceasta problema a fost concretizata de Leonid Kantorovich[6]. Primele solutii au aparut in anii 70', acestea fiind solutii Greedy de tipul First Fit[7, 8]. O noua tehnica

a fost introdusa in anii 60 de catre Paul Gilmore si Ralph Gomory: Column Generaton[9]. Acestia au aplicat principii de programare liniara in rezolvarea problemei. Este inca cea mai folosita metoda utilizata in industrie si logistica in ziua de astazi.

### 1.3 Aplicatii

Cea mai directa aplicatie a problemei Bin Packing este aranjamentul diferitelor obiecte in pachete/ paleti pentru a fi transportate eficient. De asemenea, problema sa inrudita "Cutting Stock" este prezenta in multe domenii de productie de materii prime(de exemplu sectionarea unor bucati de metal/lemn dupa specificatiile cumparatorului pentru a minimiza materialul irosit). Gasim aplicatii si in domeniul digital, precum cel multimedia unde avem reclame de diverse durate care trebuie aranjate in pauze publicitare de durata fixa. Pe acelasi principiu gasim si aplicatii in zona sistemelor de operare, precum alocarea de memorie si a cuantelor de timp pe procesor.

## 2 Demonstrație NP-Hard

Pentru a demonstra că problema Bin Packing este o problemă NP-HARD, trebuie să construim o reducere polinomială la o altă problema NP-HARD cunoscută. O instanță clasică de problemă NP-HARD este 3-SAT. Astfel vom realiza reducerea $3SAT \leq_p BinPacking$

### 2.1 Enunțarea problemelor

*Decision Bin Packing* Date de intrare: Mulțimea $N = \{w_i \in N | i \leq n\}$, Capacitatea C a recipientelor, Numărul de recipiente K Date de ieșire: Adevărat dacă obiectele pot fi împachetate în cele K recipiente, Fals altfel.

*3-SAT* Date de intrare: O mulțime de variabile $X = \{x_i \in \{True, False\} | i \leq n\}$ și o expresi logică $F = (C_1 \wedge C_2 \wedge C_3 \wedge ... \wedge C_n)$ , Unde $C_n$ este o disjuncție de exact 3 variabile Date de ieșire: Adevărat dacă există un set de variabile $S = \{x_1 ... x_n\}$ pentru care F(S) = True, Fals altfel

### 2.2 Reducrerea problemei [10, 11]

O formula 3-SAT va avea n variabile si m clauze. Fiecărei variabile si clauze îi vom atribui o valoare după cum urmează:

- fiecărei clauze $C_i$ îi atribuim două numere $c_i = 10^i, c_{2*i} = 10^i$
- fiecărei variabile $x_i$ îi atribuim două numere:
  - $a_i = 10^{(m+i)} + \sum_{\{x_i\} \subset C_j} 10^j$
  - $b_i = 10^{(m+i)} + \sum_{\{\neg x_i\} \subset C_j} 10^j$

Capacitatea fiecărui container $C = \sum_{i=0}^{m} 3 * 10^i + \sum_{i=0}^{n} 10^{m+i}$
Vom avea K = 2 containere și vom avea nevoie de un padding de
$2 * W - \sum(a_i + b_i + c_i)$
Pentru exemplificare considerăm următoarea expresie:
$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$, m = 2, n = 3
$x_1$ apare în $C_1$, deci $a_1 = 10^{(2+1)} + 10^1 = 1010$
$\neg x_1$ apare în $C_2$, deci $b_1 = 10^{(2+1)} + 10^2 = 1100$

**Table 1.** Numerical Encoding Table

| **Number** | $\mathbf{10^5}$ | $\mathbf{10^4}$ | $\mathbf{10^3}$ | $\mathbf{10^2}$ | $\mathbf{10^1}$ | **Total** |
|---|---|---|---|---|---|---|
| $a_1$ | 0 | 0 | 1 | 0 | 1 | 1010 |
| $b_1$ | 0 | 0 | 1 | 1 | 0 | 1100 |
| $a_2$ | 0 | 1 | 0 | 0 | 1 | 10010 |
| $b_2$ | 0 | 1 | 0 | 1 | 0 | 10100 |
| $a_3$ | 1 | 0 | 0 | 1 | 1 | 100110 |
| $b_3$ | 1 | 0 | 0 | 0 | 0 | 100000 |
| $c_1$ | 0 | 0 | 0 | 0 | 1 | 10 |
| $c_2$ | 0 | 0 | 0 | 0 | 1 | 10 |
| $c_3$ | 0 | 0 | 0 | 1 | 0 | 100 |
| $c_4$ | 0 | 0 | 0 | 1 | 0 | 100 |
| **Target** | **1** | **1** | **1** | **3** | **3** | **111330** |

Datele de intrare pentru Bin Packing devin:
N = $\{1010, 1100, 10010, 10100, 100110, 100000, 10, 10, 100, 100, 10\}$, padding de 10 pentru 222650 - 111330 * 2
C = 111330
K = 2

### 2.3 Corectitudinea reducerii

*Timp Polinomial*
Analizăm prima oară complexitatea reducerii:

Pentru fiecare clauză creăm un număr, deci m numere. Pentru fiecare variabilă creăm 2 numere, deci 2n. Fiecare număr are maxim $m + n$ cifre. Calculăm suma valorilor create și aflăm padding-ul, deci $O(2n+m)$. Pentru a crea un număr creăm maxim m+n cifre, deci $O(m + n)$ pentru fiecare. Așadar, complexitatea totală este $O(n * (m + n) + m + 2n + m) = O(mn) \in P$

*3SAT -> Bin Packing*
Presupunem că avem o formulă F pentru care 3-SAT(F) = True. Atunci există o configurație $S = \{x_i\}$ pentru care F(S) = True.

Conform capacității calculate, știm că într-un recipient poate încăpea doar $x_i$ sau $\neg x_i$ pentru că altfel cifra corespunzătoare lui ar fi 2, nu 1.

Fiecare variabilă $x_i$ are un număr atribuit conform reducerii. Vom umple primul container cu variabilele din S. Fiecare expresie $C_i$ va conține cu siguranță cel puțin o variabilă $x_i \in S$, așadar cifra corespunzătoare containerului(primele m cifre) va fi fie 1, 2 sau chiar 3. Pentru fiecare container avem câte 2 variabile cu care sa completăm cifra unității pana la 3. Așadar, primul container va ajunge la capacitatea C.

Știm și că suma tuturor elementelor din N este 2*C. Dacă primul container a fost umplut fix cu C, suma elementelor rămase va fi tot C, deci vor începea în cel de-al doilea container.

*Bin Packing -> 3SAT*
Presupunem că avem o configurație validă pentru Bin Packing cu variabilele construite de reducere.

Alegem containerul în care nu a fost introdusă variabila de umputură. Stiind ca suma elementelor din el este C, cu siguranță vom avea câte o variabilă, doar în formă directă sau negată, niciodată ambele. Făcând corespondența conform reducerii, vom avea un set de variabile booleane care satisfac relatia F

## 3 Prezentare Algoritmi

### 3.1 Brute-force

Spre deosebire de restul algoritmilor euristici, metoda brute-force pentru problema Bin Packing este exactă pentru ca incearcă toate soluțiile posibile si o alege pe cea mai optimă. In plus, se folosesc si anumite euristici pentru a eficientiza cautarea.
Idee: pentru fiecare item, algoritmul incearca sa il plaseze in fiecare "container" unde incape. Daca itemul nu mai are loc, backtrack.
Explicatie parametrii:
- items: itemele din care putem alege
- bins: containerele
- m: numarul de containere
- C: capacitatea fiecarui container
- i: indexul itemului curent

```
def bin_packing_BF(items, bins, num_bins, C, i, assignment):
    remaining_items = sum(items[i:])
    remaining_space = num_bins * C - sum(bins)
    if remaining_items > remaining_space:
        return False

    if i == len(items):
        return True  # toate elementele au fost plasate

    used_empty = False
```

```
    for b in range(num_bins):
        if bins[b] == 0:
            if used_empty:
                continue
            used_empty = True

        if bins[b] + items[i] <= C:
            bins[b] += items[i]
            assignment[i] = b
            if bin_packing_BF(items, bins, num_bins, C, i + 1, assignment):
                return True
            bins[b] -= items[i]  # backtrack
            assignment[i] = -1
    return False
```

### 3.2 First Fit Descending

Este unul dintre cei mai populari algoritmi de rezolvare a Bin Packing. Este o optimizare a algorimului First Fit prin sortarea acestuia în ordine descrescătoare(elemenetele mari fiind mai greu de atribuit optim la final), având toate datele prezente de la început.
Idee: Se sortează valorile. Pentru fiecare item se caută primul recipient care îl poate conține. Dacă nu încape în niciunul, se deschide încă un container

```
def bin_packing_FFD(items, bins, n, C):
    sort(items)
    num_bins = 0
    empty_space = []
    bins = []

    for i in range(n):
        j = 0
        while j < num_bins:
            if empty_space[j] >= items[i]:
                empty_space[j] -= items[i]
                bins[j].append(items[i])
                break;
            if j == num_bins:
                bins[num_bins] = [items[i]]
                empty_space[j] = C - items[i]
                num_bins ++
    return (bins, num_bins)
```

### 3.3 Modified First Fit Descending[5]

În 1985, Johnson, David S; Garey, Michael R au înbunatățit acutatețea algorimului FFD prin următoarea strategie:

Grupăm obiectele în mai multe categorii, în funcție de mărimea lor:
$A = \{w_j \in N | w_j \in (C/2, C]\}$
$B = \{w_j \in N | w_j \in (C/3, C/2]\}$
$C = N \setminus A \setminus B$

Vom asigna fiecărui obiect din A un recipient $k_1 ... k_{|A|}$

Trecem din nou prin fiecare recipient și adăugăm cel mai mare obiect B care încape(maxim unul). Pentru cele în care nu am pus nici un obiect din B, dacă cele mai mici două obiecte din C încap, trecem peste, altfel pune cel mai mic obiect din C și cel mai mare obiect din C care încape.

Apoi, trece din nou prin toate recipientele și adaugă cel mai mare element care mai are loc.

Elementele rămase sunt atribuite cu FFD, pornind de la recipientul $k_{|A|+1}$

### 3.4 Column Generation [12]

Introdusă de Gilmore și Gomory în 1961, metoda generării de coloane (Column Generation) reprezintă o abordare bazată pe programare liniară.

Spre deosebire de euristicile de tip greedy, această metodă nu lucrează direct cu obiectele, ci cu *modele de umplere* (patterns). Un model $j$ este definit ca un vector $a_j = (a_{1j}, a_{2j}, \ldots, a_{nj})^T$, unde $a_{ij}$ reprezintă de câte ori obiectul de tip $i$ apare în containerul $j$, respectând condiția: $\sum_{i=1}^{n} w_i a_{ij} \leq C$.

Algoritmul funcționează iterativ între două componente:

1. **Problema Master Restrânsă (Restricted Master Problem - RMP):** Se rezolvă o problemă de programare liniară folosind doar un set mic de modele inițiale (de exemplu, fiecare obiect în propriul container), sub rezerva:

$$\sum_{j \in P} a_{ij} x_j \geq d_i, \quad \forall i \in \{1, \ldots, n\}$$

   unde $x_j$ este numărul de containere care folosesc modelul $j$, iar $d_i$ este numărul de obiecte de dimensiune $i$. Altfel spus, modelele alese trebuie să cuprindă cel puțin toate obiectele. Din această problemă obținem *variabilele duale* $\pi_i$ asociate fiecărei constrângeri. Aceste variabile determină cât de greu este de împachetat acel obiect. Un obiect de dimensiune mare e mai greu de împachetat decât unul de dimensiune mică
2. **Sub-problema (Pricing Problem):** Pentru a îmbunătăți soluția, căutăm un model nou ($a^*$) care are costul redus negativ. Aceasta se reduce la rezolvarea unei **Probleme a Rucsacului (Knapsack Problem)**:

$$\max z = \sum_{i=1}^{n} \pi_i a_i$$

sub rezerva:

$$\sum_{i=1}^{n} w_i a_i \leq C, \quad a_i \in Z^+$$

Dacă valoarea optimă $z > 1$, noul model $a^*$ este adăugat în matricea problemei RMP (ca o coloană nouă) și procesul se repetă. Dacă $z \leq 1$, nu mai pot fi găsite modele care să îmbunătățească soluția curentă, iar algoritmul se oprește.

Deoarece rezultatul $x_j$ poate fi fracționar, soluția finală este obținută de obicei prin aplicarea unui algoritm de tip *Branch and Price* sau prin rotunjirea soluției LP, oferind o precizie mult superioară euristicilor clasice pentru instanțe de mari dimensiuni.

## 4 Evaluare

### 4.1 Acuratețea algoritmilor

*First Fit*

**Theorem 1.** *Dacă algoritmul ajunge la c containere folosite. Atunci, cel putin c-1 containere sunt mai mult de 50*

*Proof.* Presupunem ca ar fi 2 containere care sunt mai mult de jumătate goale. Atunci conținutul celui de al doilea container va fi pus deasupra primului, algoritmul gasind loc liber și nu deschide si al doilea container. Atunci vor fi c-1 containere si c-2 containere pline mai mult de 50

**Theorem 2.** *Fie W suma greutăților și $c^*$ numarul optim de containtere, $W = \sum_{k=0}^{n} w_k$. Atunci $\frac{W}{C} \leq c^*$.*

*Proof.* Pentru a putea împacheta toate obiectele, avem nevoie de W spațiu, fiecare container având C spațiu disponibil. Atunci, în cel mai bun caz, vom avea $\lfloor \frac{W}{C} \rfloor$ containere pline și unul umplut cu $W - \lfloor \frac{W}{C} \rfloor$. Astfel, $\frac{W}{C} \leq c^*$.

**Theorem 3.** *Fie W suma greutăților și c soluția găsită de First Fit. Atunci $c2 * W + 1$*

*Proof.* Știm că cel puțin c-1 containere sunt pline pe jumătate. Atunci $\frac{1}{2} * (c - 1)Wc < 2 * W + 1$

**Theorem 4.** *Dacă c este numarul de containere găsite și $c^*$ numărul optim de containere, atunci $c \leq 2 * c^*$ [2, 3]*

*Proof.* $c < 2 * W + 1; c^* > Wc < 2 * c^*$

**Table 2.** Comparativ: Complexitate și Tip Algoritm

| **Algoritm** | **Tip** | **Complexitate** | **Garantie** |
|---|---|---|---|
| Brute-force | Exact | $O(n^n)$ sau $O(n!)$ | Optim global |
| FFD | Euristic | $O(n \log n)$ | $11/9 \cdot OPT + 6/9$ |
| MFFD | Euristic | $O(n \log n)$ | $71/60 \cdot OPT + c$ |
| Column Generation | LP Relax. | NP-Hard (Knapsack subprob.) | Aproximare LP |

### 4.2 Setul de teste folosite pentru validare

Setul de date de intrare folosite pentru a valida si evalua algoritmii propusi pentru problema Bin Packing este alcatuit din:

- **Instanțe de dimensiuni reduse ($n \leq 30$):** Un set de 100 de teste utilizat pentru a stabili o referință de optimalitate. Acestea permit compararea directă a rezultatelor furnizate de euristicile FFD și MFFD cu soluția optimă generată de algoritmul Brute-Force.
- **Instanțe de dimensiuni mari ($n \leq 1000$):** Un set de 50 de teste destinat analizei performanței computaționale și a scalabilității. Accentul cade pe măsurarea timpului de execuție și pe evaluarea *Optimality Gap*-ului în raport cu limita inferioară teoretică, în absența soluției exacte.

Pentru a simula diverse scenarii din mediul industrial și logistic, am utilizat patru tipuri de distribuții stocastice pentru generarea dimensiunilor obiectelor ($w_i$), raportate la capacitatea containerului ($C$):

1. **Distribuție Uniformă:** Obiectele au dimensiuni extrase aleatoriu din intervalul $[1, C]$, oferind o perspectivă generală asupra comportamentului algoritmilor.
2. **Distribuție de Valori Mici:** Obiecte cu $w_i \in [1, \frac{C}{4}]$. Acest set testează capacitatea algoritmilor de a compacta eficient volume mari de elemente granulare.
3. **Distribuție de Valori Mari:** Obiecte cu $w_i \in [\frac{C}{2}, C]$. Reprezintă un scenariu critic unde majoritatea containerelor pot găzdui un singur obiect, forțând algoritmii să gestioneze spațiul rezidual minim.
4. **Duplicate:** Obiecte cu $w_i \in [\frac{C}{4}, \frac{C}{3}]$. Aceasta este considerată „zona dificilă" pentru euristicile de tip Greedy, și favorează Column Generation pentru ca sunt între 5 și 10 numere diferite.

### 4.3 Specificatiile sistemului de calcul

- Sistem de operare: Windows 11 25H2
- CPU: Intel Core i5-8400 @ 2.8GHz
- RAM: 24GB @ 2400MHz

### 4.4 Rezultatele evaluarii algoritmilor prezentati

**Table 3.** Teste mici.

| Algoritm | Nr. teste | %Succes | Timp executie (ms) |
|---|---|---|---|
| Brute-force | 100 | 100% | 3461 |
| First Fit Descending | 100 | 100% | 2 |
| Modified First Fit Descending | 100 | 99.86% | 3 |
| Column Generation | 100 | 100% | 12449 |

**Table 4.** Teste mari.

| Algoritm | Nr. teste | Procentaj cel mai bun rezultat | Timp executie (ms) |
|---|---|---|---|
| First Fit Descending | 50 | 100% | 14 |
| Modified First Fit Descending | 50 | 100% | 27 |
| Column Generation | 50 | 100% | 55932 |

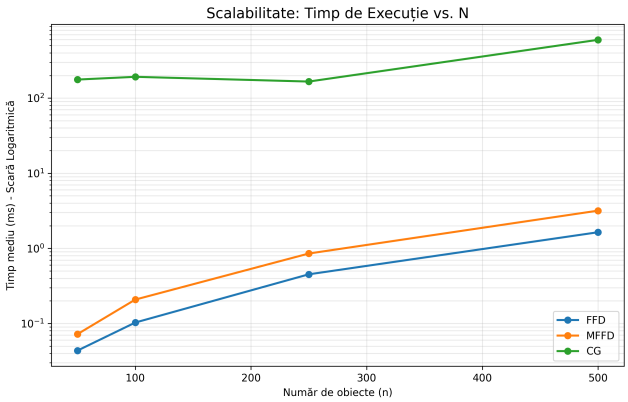


**Fig. 1.** Scalabilitate în timp

Observam un trend crescător pentru FFD si MFFD, iar pentru CG este mai lent, ceea ce ne confirma ipoteza că CG va câștiga într-un set de date suficient de mare

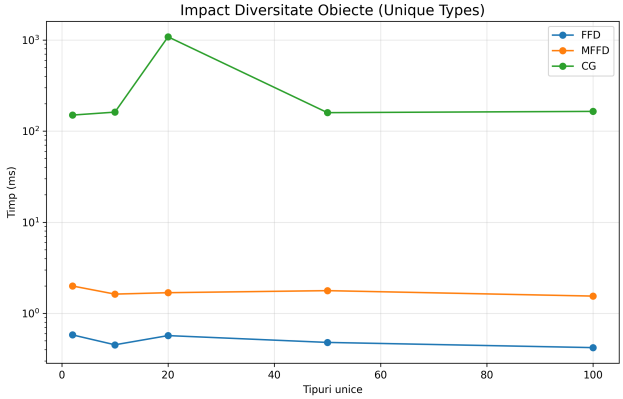


**Fig. 2.** Scalabilitate vs tipuri unice

Din testele noastre nu am găsit o diferență substanțială intre algoritmi când variem numărul de elemente unice. Cel mai evident rezultat este acel spike la $n = 20$ pentru Column Generation. Aceast prag apare deoarece numărul de iterații este maxim atunci când există suficiente exemplare din fiecare tip de obiect pentru a crea conflicte în Master Problem. Peste acest prag, diversitatea ridicată a dimensiunilor facilitează găsirea unor tipare de umplere eficiente în sub-problema Knapsack, ducând la o convergență mai rapidă.

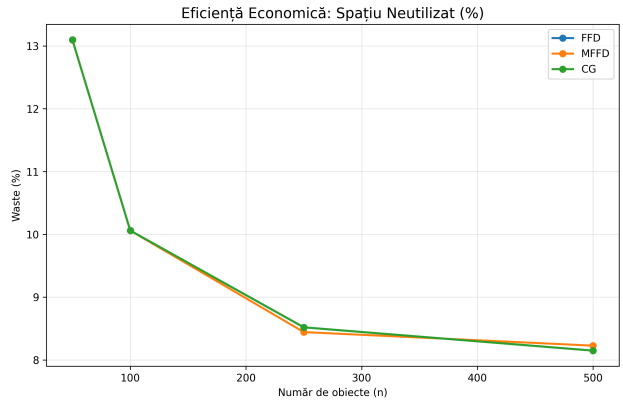


**Fig. 3.** Spațiu irosit

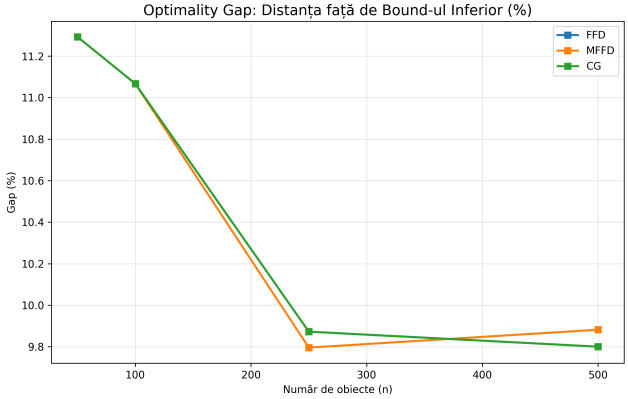


**Fig. 4.** Distanță față de optim

În ambele teste observăm ca algoritmul CG tinde să ajungă "mai optim" pentru instanțe mari, depășind FFD și MFFD

### 4.5 Interpretarea rezultatelor

Rezultatele experimentale sintetizate în Fig. 1 – 4 relevă un compromis clar între complexitatea algoritmică și eficiența computațională. Analiza detaliată a fiecărei abordări indică următoarele concluzii:

- **Brute-Force:** Această metodă rămâne utilă exclusiv pentru validarea rezultatelor pe instanțe de dimensiuni reduse ($n \leq 30$). Deși garantează optimul global, natura sa exponențială o face impracticabilă în scenarii reale. Interesant este faptul că, pe seturi de date foarte mici, BF poate depăși ca viteză Column Generation, deoarece nu implică overhead-ul inițializării unui solver liniar.
- **First Fit Descending (FFD):** S-a dovedit a fi algoritmul cel mai eficient din punct de vedere practic. Deși este o euristică, FFD a obținut soluția optimă în majoritatea testelor (Fig. 4), având un timp de execuție neglijabil. Performanța sa ridicată pe seturile de date testate se datorează faptului că, pentru distribuții relativ uniforme, strategia Greedy de sortare descrescătoare minimizează eficient spațiul rezidual.
- **Modified First Fit Descending (MFFD):** Rezultatele indică faptul că rafinamentul adus de MFFD nu se traduce automat într-o performanță superioară pentru seturile de date arbitrare. Timpul de execuție mai ridicat (de aproximativ 2-3 ori mai mare decât FFD) și rezultatele similare sugerează că această variantă este optimă doar pentru distribuții specifice (ex: obiecte cu dimensiuni între $C/3$ și $C/2$), unde FFD standard întâmpină dificultăți.

– **Column Generation (CG):** Deși fundamentată matematic, această metodă a prezentat cel mai mare consum de resurse în testele noastre. „Spike-ul" de performanță observat la $k = 20$ tipuri unice (Fig. 2) subliniază sensibilitatea algoritmului la diversitatea datelor. Este important de notat că CG rezolvă relaxarea liniară a problemei; prin urmare, pe instanțe mici, soluția poate părea „neoptimă" dacă nu este cuplată cu o tehnică de *branching* pentru a forța integralitatea. Totuși, analiza irosirii (Fig. 3) confirmă că CG tinde spre soluții mai dense pe măsură ce volumul de date crește.

**Sinteză:** Pentru uz industrial general, **FFD** oferă cel mai bun raport între simplitatea implementării și precizie. **Column Generation** rămâne opțiunea superioară doar în medii de producție masivă, unde reducerea risipei de material cu chiar și 1% justifică investiția în resurse computaționale și complexitate logică.

**Table 5.** Comparație între FFD și Column Generation

| **Criteriu** | **First Fit Descending (FFD)** | **Column Generation (CG)** |
|---|---|---|
| Complexitate Implementare | Foarte Mică (Greedy) | Foarte Mare (Simplex + Knapsack) |
| Viteză ($n < 1000$) | Instantaneu | Lent (secunde/minute) |
| Acuratețe | Aproape optim (eroare $<$ 2%) | Optim matematic (teoretic) |
| Utilizare Recomandată | Sisteme real-time, logistică ușoară | Industrie grea, producție de serie |

## 5 Concluzii

Analiza comparativă realizată în acest studiu evidențiază un compromis fundamental între complexitatea implementării și eficiența computațională în rezolvarea problemei Bin Packing. Demonstrația de complexitate prin reducere de la 3-SAT a subliniat natura NP-Hard a problemei, justificând astfel utilizarea metodelor euristice și a programării liniare în detrimentul soluțiilor exacte de tip brute-force pentru instanțe de mari dimensiuni.

Rezultatele experimentale confirmă faptul că euristica *First Fit Descending* (FFD) reprezintă soluția optimă pentru majoritatea aplicațiilor practice de dimensiuni medii ($n < 500$), oferind un echilibru remarcabil între timpul de execuție minim și o eroare de aproximare neglijabilă (sub 1-2% față de optimul teoretic). Deși *Modified FFD* propune o logică mai rafinată, testele noastre indică faptul că sporul de precizie adus nu justifică întotdeauna complexitatea suplimentară a algoritmului în contextul distribuțiilor uniforme.

În ceea ce privește *Column Generation*, deși aceasta este metoda de elecție în industria grea datorită fundamentului său matematic riguros, studiul relevă un

overhead computațional semnificativ pentru instanțe reduse. Cu toate acestea, tendința de convergență spre bound-ul inferior ($L_1$) demonstrează superioritatea metodei în scenarii de producție de serie, unde minimizarea risipei de material (waste analysis) are un impact economic critic. În concluzie, alegerea algoritmului trebuie dictată de granularitatea datelor și de constrângerile de timp ale sistemului de calcul utilizat.

## References


1. Garey, M. R. and Johnson, D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. ACM, 1979. Available at: https://dl.acm.org/doi/epdf/10.1145/3828.3833
2. Piyush, K.: Approximation Algorithms Notes. University of California, Riverside. Available at: https://cs.ucr.edu/ neal/2006/cs260/piyush.pdf
3. MIT OpenCourseWare: Bin Packing Problem (Video Lecture). Available at: *https* : *//www.youtube.com/watch?v* = *R*76*aAh*$_l$*i*50
4. Belov, G. and Scheithauer, G.: A cutting plane algorithm for the one-dimensional cutting stock problem. *European Journal of Operational Research* (2016). Available at: https://www.sciencedirect.com/science/article/abs/pii/S0377221716302491
5. Gilmore, P. C. and Gomory, R. E.: A linear programming approach to the cutting stock problem. *Operations Research* (1961). Available at: https://www.sciencedirect.com/science/article/pii/0885064X85900226
6. Kantorovich, L. V. and Zalgaller, V. A.: Calculation of Rational Cutting of Stock. Lenizdat, Leningrad (1951).
7. Knuth, D. E.: *The Art of Computer Programming*. Addison-Wesley.
8. Ullman, J.: Greedy Algorithms and Performance Bounds. Lecture notes.
9. Gomory, R.: A Linear Programming Approach to the Cutting Stock Problem I
10. Suri, S.: NP-Completeness Notes. University of California, Santa Barbara. Available at: https://sites.cs.ucsb.edu/ suri/cs130b/npc.pdf
11. Schulz, A. S.: Bin Packing Approximation Algorithms. University of Freiburg. Available at: *https* : *//ac.informatik.uni* − *freiburg.de/lak*$_t$*eaching/ws*11$_1$2*/combopt/notes/bin*$_p$*acking.pdf*
12. Jiahui Duan and Xialiang Tong and Fei Ni and Zhenan He and Lei Chen and Mingxuan Yuan A Data-Driven Column Generation Algorithm For Bin Packing Problem in Manufacturing Industry Available at: *https* : *//arxiv.org/abs/*2202.12466