CUTTING STOCK PROBLEM

CJELOBROJNO PROGRAMIRANJE

Sandro Paradžik

17. juni 2024.

UNIVERZITET U SARAJEVU
PRIRODNO-MATEMATIČKI FAKULTET

Sadržaj

1 Uvod

Mnogo optimizacionih problema se može svesti na probleme cjelobrojnog programiranja (IP), odnosno mixed-integer programa (MIP). Jedan takav problem je i *Cutting stock problem* (CSP), u ovom radu ćemo se najviše fokusirati na njega (tačnije, njegovu 1D varijantu). Problem kod ovih problema je što su često NP-teški (ovo na primjer vrijedi i za CSP [3]). Uporedit ćemo koliko su metode prikladne za rješavanje IP-a efikasnije od brute-force pristupa. Može se reći da ćemo najviše pažnje obratiti na column generation algoritam.

U dijelu 1 ćemo opisati i definisati problem, te dati dvije različite formulacije cjelobrojnih programa. U dijelu 2 ćemo predstaviti algoritme koji su tema ovog rada, te ćemo pokazati njihovu implementaciju. U dijelu 3 ćemo pokazati neke grafke koji opisuju performanse algoritama. Dio 4 govori o interpretaciji rezultata i spominje alternativne pristupe koje nismo uzeli u obzir. Za čitav kod i implementaciju pogledati dodatne materijale. Kako je knapsack problem koji ćemo morati riješiti da bismo pričali o algoritmu generating columns, za detalje o knapsack problemu i implementaciji pogledati dodatak A, ili dodatne materijale.

1.1 Opis problema

Zamislimo da imamo tvornicu papira. Dobijamo velike komade papira dimenzija $1 \times L, L \in \mathbb{R}^+$. Ove velike početne komade možemo rezati samo po širini (svaki papir koji dobijemo rezanjem će idalje imati širinu 1). Zato ćemo za papire dimenzija $1 \times a$ reći da im je dimenzija (odnosno dužina) a. Neka su dimenzije papira za kojim postoji neka potražnja $a_i \in \mathbb{R}^+, i = 1, \ldots, n$. Neka su te potražnje date sa $d_i \in \mathbb{N}, i = 1, \ldots, n$, gdje d_i predstavlja potražnju (količinu) za papirima dimenzije d_i, to jeste koliko papira te dimenzije trebamo napraviti. Ono što želimo minimizirati u rješenju ovog problema je broj velikih komada dimenzije L koje moramo iskoristiti.

1.2 IP formulacija

CSP možemo formulisati u obliku IP problema. Ovo će biti korisno za primjenu algoritama linearnog programiranja. Slijedi IP formulacija za problem (definisan kao u dijelu 1.1) koja je možda na prvi pogled najintuitivnija:

$$
\begin{aligned}
z = \min \sum_{k=1}^{K} y_k & \\
& \sum_{k=1}^{K} x_{ik} \geq d_i \quad i = 1, \ldots, n \\
& \sum_{i=1}^{n} a_i x_{ik} \leq L y_k \quad k = 1, \ldots, K \\
& y_k \in \{0, 1\} \quad k = 1, \ldots, K \\
& x_{ik} \geq 0,\, x_{ik} \in \mathbb{Z} \quad i = 1, \ldots, n;\, k = 1, \ldots, M,
\end{aligned} \tag{1}
$$

gdje $y_k = 1$ ako je komad k iskorišten, inače je 0. Brojevi x_{ik} predstavljaju koliko smo papira dužine a_i dobili iz komada k, i K je gornja granica za broj komada koji su nam potrebni ($z \leq K$).

Važno je naglasiti da je ovaj prvobitni IP model koji smo napravili dosta loš. Razlog tome je što koristimo potencijalno jako velik broj nejednakosti (odnosno varijabli).

Pokažimo sada bolji način da formulišemo problem. Posmatrajmo konfiguracije rezanja. Pod konfiguracijom rezanja smatramo vektor k koji je veličine iste kao i a, te na i-tom mjestu sadrži broj koji govori koliko papira a_i uzimamo u toj konfiguraciji. Da bi konfiguracija bila validna, k sadrži nenegativne cijele brojeve, i vrijedi $a \cdot k \leq L$. Neka je matrica A matrica čije su kolone sve moguće maksimalne konfiguracije rezanja. Konfiguracija je maksimalna ako ne možemo iz nje dobiti više niti jedan predmet. Neka postoji m maksimalnih konfiguracija. Neka je c vektor jedinica veličine iste kao broj konfiguracija rezanja. Sada CSP možemo predstaviti u obliku sljedećeg linearnog (cjelobrojnog) programa:

$$
\begin{aligned}
z = \min c \cdot x & \\
& Ax \geq d \\
& x \geq 0, \quad x \in \mathbb{Z}^m.
\end{aligned} \tag{2}
$$

2 Algoritmi i implementacija

2.1 Branch and bound

Branch and bound je standardan pristup za rješavanje cjelobrojnih programa. Možemo ga smatrati brute-force algoritmom. Slijedi implementacija prilagođena

za CSP, bez implementacije nekih pomoćnih funkcija.

```
import numpy as np
from scipy.optimize import linprog
import math
def branch_and_bound(c, A, b, tx, tfun):
    # Vraca double-ove,
    # inicijalizacija je: tx = [[0]], tfun = [float('inf')]
    '''
        Vraca lp.x, lp.fun
        Podrazumijeva da saljemo linearan program
        koji IMA rjesenja.
        Ovo je okej jer ovu funkciju koristimo za cutting
        stock problem.
        Kao i scipy.optimize.linprog, trazi minimum funkcije.

        tx je lista sa jednim elementom!
        (a taj jedan element je np ndarray)

        tx i tfun su shared state

        n je sirina matrice (len(x), tj len(c))

        Za rjesavanje linearnih relaksacija koristimo
        linprog funkciju iz scipy.optimize.
    '''
    rez = linprog(c, A, b, method='highs')
    # Provjerimo prvo da li ovo ima rjesenja!!!
    if not rez.success:
        return tx[0], tfun[0]

    # Prvo provjeravamo da li je relaksirano rjesenje
    # gore od t (trenutno najbolje)
    if rez.fun > tfun[0]: # rez.fun moze biti None!!??
        return tx[0], tfun[0]

    # Ako smo dobili sve cijele brojeve kao rjesenje, provjeravamo
    # da li je to bolje od tx i tfun, potencijalno updatujemo
```

```
37        # tx, tfun te ih returnamo
38        if niz_cijelih(rez.x):
39            if rez.fun < tfun[0]:
40                tfun[0] = rez.fun
41                tx[0] = rez.x
42            return tx[0], tfun[0]
43
44        # Ako postoji neki xi koji nije cijeli, radimo branching.
45        # (i rj lin relaksacije nije bilo gore od trenutno
46        # najboljeg rjesenja) Dakle, racunamo x1, fun1 i x2, fun2
47        # rekurzivno. Ta rekurzija takodjer koristi isti shared
48        # state kao do sada.
49
50        i = indeks_od_xi(rez.x)
51
52        # >=
53        novi_red2 = np.zeros(len(c))
54        novi_red2[i] = -1
55        A2 = np.append(A, [novi_red2], axis=0)
56        b2 = np.append(b, -math.ceil(rez.x[i]))
57        branch_and_bound(c, A2, b2, tx, tfun)
58
59        # <=
60        novi_red1 = np.zeros(len(c))
61        novi_red1[i] = 1
62        A1 = np.append(A, [novi_red1], axis=0)
63        b1 = np.append(b, math.floor(rez.x[i]))
64        branch_and_bound(c, A1, b1, tx, tfun)
65
66        # Kako koriste isti shared state, dovoljno je:
67        return tx[0], tfun[0]
68
69
70    def csp_branch_and_bound(a, d, L):
71        '''
72            Glavna funkcija koja koristi branch and bound za
73            rjesavanje cutting stock problema.
74        '''
```

```
75
76      trojka = cp_csp(a, d, L)
77      TX = [[0]]
78      TFUN = [float('inf')]
79      rjesenje = branch_and_bound(
80          trojka[0],
81          -trojka[1],
82          -trojka[2],
83          TX,
84          TFUN
85      )
86      return rjesenje[1]
87
88
89  def csp_scipy_integrality(a, d, L):
90      '''
91          Funkcija koja koristi scipy.optimize.linprog funkciju za
92          rjesavanje cutting stock problema koji je formulisan kao
93          cjelobrojni program sa onim cutting konfiguracijama.
94          Mislim da i linprog radi neku vrsu branch and bounda,
95          ali mozemo ocekivati da je to dosta optimizovano, pa bi
96          ova funkcija trebala imati bolje performanse od funkcije
97          CSP_branch_and_bound.
98      '''
99      trojka = cp_csp(a, d, L)
100     rjesenje = linprog(
101         trojka[0],
102         -trojka[1],
103         -trojka[2],
104         integrality=1
105     )
106     return rjesenje.fun
```

2.2 Column generation

Column generation je prvobitno i korišten za CSP, te se kasnije počeo primjenjivati i na neke druge probleme. Prvo ćemo ukratko opisati motivaciju za column generation algoritmom. Posmatrajmo sljedeći primjer.

Primjer 1. Neka je CSP dat sa

$$a = \begin{bmatrix} 110 & 100 & 70 & 66 & 40 & 30 & 30 & 3 & 2 \end{bmatrix}^T$$

$$d = \begin{bmatrix} 41 & 12 & 51 & 20 & 3 & 79 & 32 & 60 & 144 \end{bmatrix}^T$$

$$L = 300$$

.

Konfiguracija rezanja postoji mnogo, i kako se kompleksnost problema povećava, broj konfiguracija raste eksponencijalno. U primjeru 1 postoji 19065 maksimalnih konfiguracija. Čak i generisanje ovih kolona je sporo. Koristeći neki solver, dobijamo rješenje ovog problema. Ispostavlja se da postoji (u tom rješenju) samo 7 iskorištenih kolona.

Ukoliko bismo posmatrali samo tih nekoliko kolona, dobili bismo isto rješenje, i to možemo posmatrati kao prethodnu kolonu A koja ima puno više kolona, ali većina x-ova ima vrijednost 0. Problem je kako odabrati kolone sa kojim tražimo rješenje? Ideja je da počnemo sa nekom početnom matricom A, i u svakoj iteraciji dodajemo kolonu koja najviše smanjuje funkciju cilja. Naravno, da bismo saznali koja je to kolona (i da li postoji), ne smijemo prolaziti kroz sve moguće kolone i gledati koliko su dobre. Razlog je što onda sa ovim pristupom ne bismo dobili nikakvo poboljšanje u odnosu na prethodne pristupe.

Ideja kod generisanja kolona je da počnemo rješavanje problema sa nekim malim podskupom kolona. Zatim rješavamo linearnu relaksaciju problema. Zaokruživanjem na veće cijele brojeve dobijamo neko aproksimativno rješenje koje zasad nije dobro. Sada se pitamo da li je moguće dodati neku kolonu u linearni program tako da ta kolona postane dio baze, odnosno bude dio rješenja (i taj postupak makar malo popravi vrijednost funkcije cilja linearne relaksacije). Da bismo odredili takvu kolonu, u matricu dodajemo kolonu koja ima najmanji *reduced cost* (ako je on negativan, inače stajemo sa algoritmom). Važna ideja je da dualne vrijednosti prethodno izračunate linearne relaksacije predstavljaju gradijent povećavanja odnosno smanjivanja funkcije cilja. Fomulisanjem ovog problem dolazimo do zaključka da da bi odredili koju kolonu da dodamo, dovoljno je riješiti knapsack problem. U ovom radu nećemo obraćati previše pažnje da precizno matematički pokažemo da je ovo slučaj. Razlog je to što je glavni cilj ovog rada implementacija i poređenje performansi ovih algoritama. Važno je za naglasiti da je algoritam generisanja kolona aproksimativni algoritam, ali on daje prilično dobre aproksimacije. Ako bismo hjteli neki algoritam koji sa sigurnošću nalazi optimalno rješenje, mogli bismo koristiti branch and price (kombinacija column

generation algoritma i branch and bound).

Slijedi implementacija generating columns algoritma.

```python
import numpy as np
from scipy.optimize import linprog
from knapsack import knapsack


def csp_generating_columns(a, d, L):
    '''
        Rjesava cutting stock problem metodom generating columns.
        Pocinje sa baznim kolonama takvim da se u svakom cuttingu
        koristi maksimalan broj jedne od duzina a.
        Zatim iterativno dodajemo kolone dok god knapsack daje
        kolone koje ce povecati vrijednost funkcije cilja za csp.
        Da bismo izracunali gradijent za dodavanje neke kolone,
        koristimo skalarni proizvod dualnih vrijednosti i te
        kolone. Ovo nam daje knapsack problem.
    '''
    n = len(a)

    # min reduced cost racunamo kao
    # min(1 - dualne_var * nova_kolona)

    # inicializacija baznih kolona
    kolone = []
    for i in range(n):
        nova_kol = np.zeros(n)
        nova_kol[i] += L // a[i]
        if nova_kol[i] == 0:
            print("Problem nema rjesenja: a[i] > L!")
            return
        kolone.append(nova_kol)
    A = np.column_stack(kolone)

    # inicijalizacija koeficijenata funkcije cilja
    c = np.ones(n)

```

```
    # linprog b = -d

    primal = linprog(c = c, A_ub = -A, b_ub = -d)
    # dualni_problem = linprog(c = d, A_ub = -A.T, b_ub = -c)
    # ne moramo odvojeno rjesavati dualni, mozemo iz linprog za
    # primal izvuci dualne varijable

    x = primal.x
    fun = primal.fun
    dualne = primal.ineqlin.marginals

    while True:
        rez_knapsack = knapsack(l = a, L = L, v = -dualne)
        # -dualne jer knapsack maksimizuje
        # a mi hocemo minimizaciju
        # - rez_knapsack[1] @ dualne == rez_knapsack

        reduced_cost = 1 - rez_knapsack[0]

        if reduced_cost >= 0:
            break

        A = np.column_stack((A, rez_knapsack[1]))
        c = np.append(c, 1)
        primal = linprog(c = c, A_ub = -A, b_ub = -d)

        x = primal.x
        if x[-1] < 0.00000001:
            break # ako smo dodali kolonu koju cemo odma izbacit,
                  # upast cemo u beskonacnu petlju
        fun = primal.fun
        dualne = primal.ineqlin.marginals
        A = A[:, x > 0] # python :) ; ova linija koda uklanja
                # kolone koje ne ucestvuju u rjesenju
                # (tj nisu bazne)
        x = x[x > 0]
        # ova uklanja medju x ovima one koji su bili nula
        c = np.ones(A.shape[1])
```

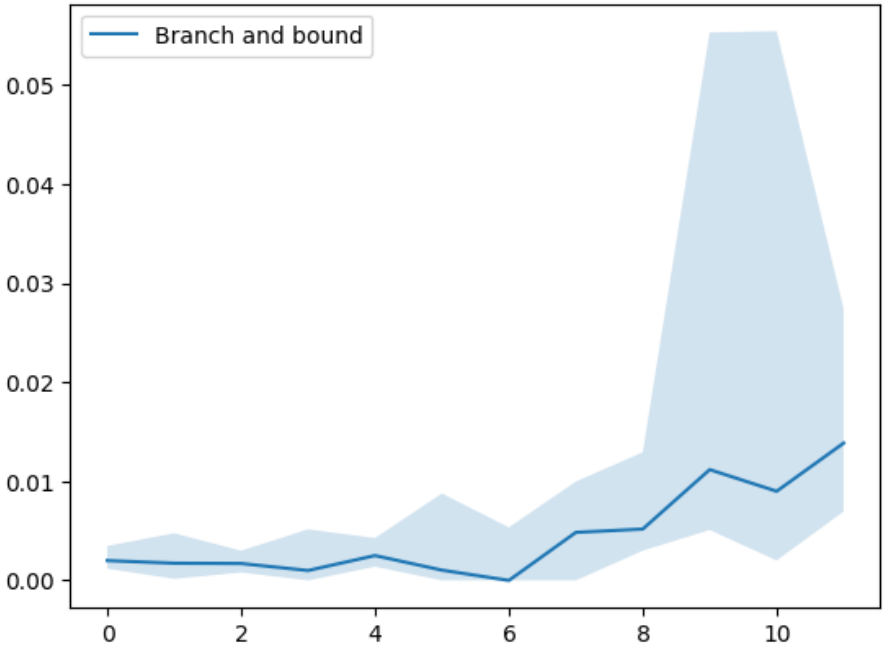
```
            # print(A, x, fun)

        donja_granica = fun
        x = np.ceil(x)
        postignuta_fun = np.sum(x)

        return donja_granica, postignuta_fun, x, A
```

3 Rezultati

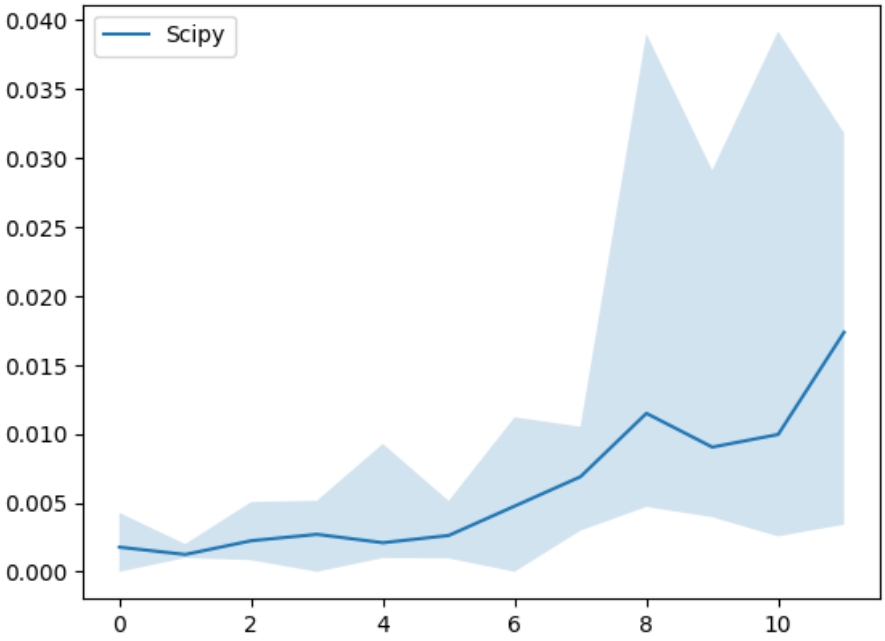
Na slikama 2, 3, 4 su pokazane performanse funkcija koje smo implementirali (za više detalja o implementaciji pogledati dodatne materijale). Slijedi kod koji smo koristili za generisanje je inputa.

```
matrica = eval.evaluiraj_algoritme(
    [algoritam],
    d_max=5,
    L=100,
    n=7,
    broj_iter=12,
    broj_ponavljanja=20,
    n_step=0,
    L_step=10
)
```

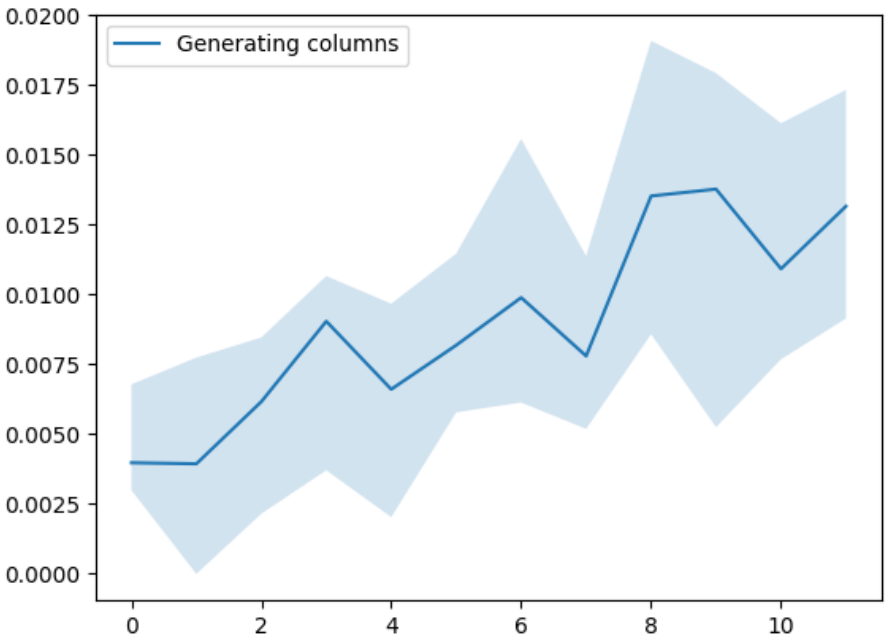
Slika 1: Kod koji generiše input za funkcije i evaluira ih.



Slika 2: Performanse algoritma branch and bound koji smo implementirali na inputima generisanim sa kodom.Na y osi je vrijeme izvršavanja funkcije. Svijetlo plavom bojom su prikazani kvartili, a tamna plava linija predstavlja medijanu.



Slika 3: Performanse algoritma scipy.optimize.linprog sa parametrom integrality = 1 (i koji koristi formulaciju 2) na inputima generisanim sa kodom. Na y osi je vrijeme izvršavanja funkcije. Svijetlo plavom bojom su prikazani kvartili, a tamna plava linija predstavlja medijanu.

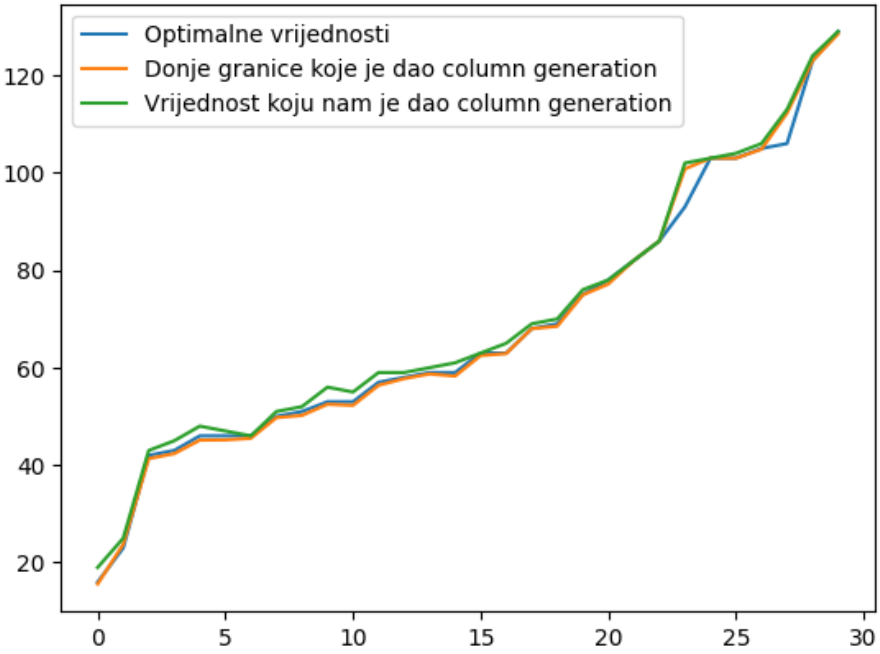


Slika 4: Performanse algoritma generating columns kojeg smo implementirali (pogledati dodatne materijale). Inputi su generisani sa kodom. Na y osi je vrijeme izvršavanja funkcije. Svijetlo plavom bojom su prikazani kvartili, a tamna plava linija predstavlja medijanu.

Možemo vidjeti da generating columns ima dosta bolje performanse. Razlika bi bila još veća na većim primjerima.

Trebamo se zapitati koliko dobre aproksimacije daje column generation algoritam. Koliko su column generation rješenja udaljena od stvarnih možemo vidjeti na slici 5(ovo ne znači da se ne može naći primjer gdje ovo neće vrijediti, ali je ovaj grafik idalje dobar pokazatelj). Slijedi kod za generisanje grafika.

```
1  nacrtaj_cg_fun_razliku(d_max=50, L=150, n=7, broj_ponavljanja=30)
```



Slika 5: Grafik je generisan kodom iznad. Za više detalja pogledati dodatne materijale. Na ovom grafiku vidimo da column generation daje jako dobre rezultate.

4 Diskusija

Iako nismo previše razmatrali formulaciju 1, ona može biti korisna u nekim slučajevima [2]. Na primjer, ako veličine početnih komada papira nisu jednake. Također, ima smisla napraviti eksperimente i uporediti performanse ove formulacije u nekim algoritmima za neke probleme koji imaju određene osobine (na primjer, možda znamo da će konačni broj iskorištenih komada papira biti jako mal).

Dosta važan algoritam za rješavanje CSP problema je *branch and price*. On koristi column generation zajedno sa branch and bound algoritmom. U suštini, kada dobijemo u column generation algoritmu rezultate koji su razlomci, onda radimo branching po toj varijabli. Ovaj algoritam ne mora biti efikasan, ali sa sigurnošću daje tačan rezultat.

U praksi, ako rješavamo neku malu instancu problema, možemo koristiti čak i branch and bound. Za tako male instance to ima puno više smisla nego column generation, jer se može desiti da nam na primjer column generation da rješenje koje koristi 4 komada papira, a optimalno je 3. Ovo može praviti ogromnu razliku u cijeni. Iz ovog razloga je column generation prikladniji za probleme u kojima očekujemo veći broj potrebnih komada papira, i to hoćemo li naći rješenje koje koristi 1000 ili 999 komada nam ne pravi veliku razliku. Ako bi nam to ipak pravilo razliku iz nekog razloga, oslonili bi se na branch and price ili neku drugu metodu.

Važno za column generation algoritam je da se optimalna vrijednost vrlo vjerovatno ne razlikuje mnogo od dobivenog rješenja (razlikuje se za najviše neku malu konstantu kao na primjer 1 ili 2)[1]. Osim ovoga, column generation daje i vrijednost funkcije za linearnu relaksaciju, što nam daje donju granicu za minimum.

Jedan problem u column generation algoritmu sa kojim sam se susreo je da algoritam upadne u beskonačnu petlju. Tačnije, kolona koja se doda u nekoj iteraciji ne bude iskorištena u rješenju, pa onda bude opet dodana i ovo se nastavlja beskonačno mnogo puta. Ovaj problem rješavamo tako što prekinemo algoritam kada se u rješenju ne iskoristi funkcija koju smo tek dodali. Ne tvrdim da je ovo najbolji način rješavnja ovog problema.

Također, za rješavanje CSP problema se koristi i Dantzig-Wolfe dekompozicija, koju ne razmatramo u ovom radu.

5 Dodatni materijali

Kod je dostupan na GitHub-u. Tu je također pokazano kako se funkcije mogu koristiti, te su upoređene performanse algoritama.

Literatura

[1] David P. Williamson, R. A. (2008). Orie 6300 mathematical programming i, lecture 17. Cornell University.

[2] Vance, P. H. (1998). Branch-and-price algorithms for the one-dimensional cutting stock problem. *Computational Optimization and Applications*, 9:211–228.

[3] Wolsey, L. (2020). *Integer Programming*. Wiley.

A Knapsack problem

Pri rješavanju CSP problema metodom column generation, novi problem koji se pojavljuje i trebamo ga riješiti je knapsack problem. Ispod se nalazi implementacija algoritma dinamičkog programiranja koja rješava knapsack problem u vremenu $\mathcal{O}(Ln)$ gdje je L veličina ruksaka, a n je broj vrsta predeta koji su na raspolaganju.

```
import numpy as np
from copy import copy

def knapsack(l, v, L):
    '''
        Rjesava cjelobrojni problem ruksaka. Velicine predmeta su
        l, vrijednosti su v, L je velicina ruksaka. Funkcija vraca
        par: vrijednost funkcije cilja, niz koji predstavlja
        rjesenje. n je broj predmeta.
    '''

    n = len(l)

    DP = [[0, np.zeros(n)] for _ in range(L + 1)]

    for i in range(L + 1):
        for j in range(n):
            if l[j] <= i:
                rez_dp = DP[i - l[j]]
                if rez_dp[0] + v[j]  > DP[i][0]:
                    DP[i][0] = rez_dp[0] + v[j]
                    DP[i][1] = copy(rez_dp[1])
                    DP[i][1][j] += 1
    return DP[L]
```