# Robustni problem nahrbtnika

Eva Babnik in Jan Založnik

December, 2020

# Kazalo



# Slike

# 1 Uvod

Namen tega poročila je, da predstaviva robustni problem nahrbtnika in kodo s katero sva ga rešila. Kodo bova sproti ustrezno komentirala ter dodala njeno časovno zahtevnost. Vključila bova tudi nekaj primerov in zraven dodala še časovne vrednosti, ki jih je program potreboval za izračun rešitve. Program bova potem poskušala uporabiti na finančnem modelu. Podatke iz ameriške borze bova uporabila na kodi za robustni problem in tako poskušala sestaviti optimalni portfelj. Prav tako bova tudi tu dodala primere in njihovo časovno zahtevnost.

# 2 Povzetek

Poročilo se prične s kratko predstavitvijo navadnega oziroma klasičnega problema nahrbtnika in nato še z razlago robustnega problema nahrbtnika. V robustnem primer problema nahrbtnika prav tako kot pri klasičnemu problemu iščemo množico predmetov, ki jih bomo položili v nahrbtnik in njihovo optimalno vrednost, vendar v tem primeru nimamo točnih podatkov o vseh težah predmetov. Za njihove teže vemo le, da so element določenega zaprtega intervala, ki torej ima zgornjo in spodnjo mejo. Poleg intervalov tež predmetov pa še vemo maksimalno število elementov, ki lahko spremeni svojo težo, to število označimo z $\lambda$.

Za sledeči problem sva s pomočjo dinamičnega programiranja napisala rekurzivne enačbe, na katerih bazira najina koda v programskem jeziku *python*. Napisala sva program, ki reši tako klasični problem nahrbtnika kot tudi robustni problem nahrbtnika in za rešitev vrne optimalni seznam predmetov, ki jih položimo v nahrbtnik in skupno optimalno vrednost le-teh. Sestavila pa sva tudi enostavnejši grafični program, ki omogoča, da lahko problem reši tudi posameznik, ki nima nikakršnega predznanja v programiranju.

Nato sva kodo malo modificirala in jo uporabila na malo poenostavljenem finančnem modelu. Program prebere datoteko s podatki, ki zajemajo ceno delnice, najvišjo možno ceno delnice in letno stopnjo donosa, vrne pa optimalni portfelj, ki si ga lahko izberemo če imamo na voljo za investicije določeno vsoto denarja.

# 3 Problem nahrbtnika

Klasični problem nahrbtinka (angl. *classical Knapsack Problem*) je računalniški problem, s katerim poskusimo zapolniti nahrbtnik z danimi predmeti, ki ima vsak svojo ceno in težo. Bodisi gre za dejansko polnjenje nahrbtnika, polnjenje nakupovalne vreče ali morda zlaganje predmetov v avto. V nadaljevanju poročila bova za klasični problem uporabljala kar kratico KP. Na voljo imamo množico n-tih predmetov, ki jo označimo z $N = \{1, \ldots, n\}$ in nahrbtnik s kapaciteto $c$. Vsak predmet ima pozitivno vrednost $p_j$ in pozitivno utež $w_j$. Problem nahrbtnika nas sprašuje, katero podmnožico predmetov iz N moramo položiti v nahrbtnik, da bo skupna vrednost le teh čim večja možna in da ne bo presegla nahrbtnikove kapacitete. Torej maksimiziramo skupno vrednost predmetov pri pogoju, da seštevek izbranih uteži ne presega nahrbtinkove zmogljivosti. Problem lahko predstavimo kot celoštevilski linearni program (CLP):

$$max \sum_{j \in N} p_j x_j$$
$$\sum_{j \in N} w_j x_j \leq c$$
$$x_j \in \{0, 1\}, j \in N$$

Kjer $x_j$ zavzame vrednost 1, če j-ti predmet položimo v nahrbtnik, sicer zavzame vrednost 0.

# 4 Robustni problem nahrbtnika

Robustni problem nahrbtnika (angl. *Robust Knapsack Problem*) v nadaljevanju RKP, je nekakšna nadgranja problema nahrbtnika. Dodaten problem se pojavi pri točnosti naših podatkov, in sicer pri utežeh $w_j$. Vsak predmet $j$ ima svojo nominalno težo $w_j$, ki pa je lahko netočna, ampak zanjo vemo, da se nahaja na intervalu $[w_j - \underline{w}_j, w_j + \overline{w}_j]$. Podan imamo tudi celoštevilski parameter $\Gamma$, ki označuje največje možno število predmetov z netočno izmerjeno težo. Pri iskanju rešitve problema moramo torej paziti, da bo seštevek vseh novih uteži še vedno manjši od kapacitete nahrbtnika. Težav z rešitvijo seveda ne bomo imeli, če bodo vse dejanske uteži nižje oziroma lažje od njene nominalne vrednosti, lahko pa se zgodi tudi najslabši možni izid, ko vse uteži zavzamejo zgornjo mejo intervala. Dopustno rešitev, kjer je $J \subseteq N$ lahko

formuliramo na naslednji način:

$$\sum_{j\in J} w_j + \sum_{j\in \hat{J}} \overline{w}_j \leq c, \quad \forall \hat{J} \subseteq J \text{ in } |\hat{J}| \leq \Gamma$$

# 5 Pristop z dinamičnim programiranjem

Naj bo $\bar{z}(d, s, j)$ najvišji dobiček za dopustno rešitev s skupno težo $d$, kjer so upoštevani samo elementi iz množice $\{1, \ldots, j\} \in N$ in samo $s$ izmed njih doseže zgornjo mejo $\hat{w}_j$. Naj bo $z(d, j)$ največji dobiček za dopustno rešitev s skupno težo $d$, kjer so upoštevani samo elementi iz množice $\{1, \ldots, j\} \in N$ in naj jih le $\Gamma$ spremeni težo iz nominalne na zgornjo mejo $\hat{w}_j$. Torej velja $d = 0, 1, \ldots, c$; $s = 0, 1, \ldots, \Gamma$ in $j = 0, 1, \ldots, n$. Ključna lastnost pravilnosti tega pristopa je predpostavka, da so predmeti razvrščeni po padajoči teži $\hat{w}_j$. Problem lahko zapišemo z naslednjimi rekurzivnimi zvezami:

$$\bar{z}(d, s, j) = max\{\bar{z}(d, s, j-1), \bar{z}(d - \hat{w}_j, s-1, j-1) + p_j\}$$
$$\text{za } d = 0, \ldots, c;\ s = 1, \ldots, \Gamma \text{ in } j = 1, \ldots, n.$$

$$z(d, j) = max\{z(d, j-1), z(d - w_j, j-1) + p_j\}$$
$$\text{za } d = 0, \ldots, c \text{ in } j = \Gamma + 1, \ldots, n.$$

Začetna vrednost je $\bar{z}(d, s, 0) = -\infty$ za $d = 0, \ldots, c$ in s $= 0, \ldots, \Gamma$. Nato nastavimo $\bar{z}(0, 0, 0) = 0$. Oba zapisa sta med seboj povezana z enakostjo $z(d, \Gamma) = \bar{z}(d, \Gamma, \Gamma\}$ za vsak $d$. Optimalno vrednost robustnega problema nahrbtnika dobimo kot:

$$z^* = max \begin{cases} max\{z(d, n)|\ d = 1, \ldots, c \\ max\{\bar{z}(d, s, n)|\ d = 1, \ldots, c;\ s = 1, \ldots, \Gamma - 1\} \end{cases}$$

kjer porabimo tudi celotno kapaciteto nahrbtnika $c^* \leq c$.

Algoritem dinamičnega programiranja je sestavljen iz dveh korakov. V prvem koraku dobimo optimalno rešitev, ki vsebuje največ $\Gamma$ elementov s povečano težo. V drugem koraku pa nato dobljeno rešitev lahko razširimo z dodatnimi elementi z nespremenjeno težo. Algoritem lahko razdelimo na dva koraka, ker razvrstitev predmetov po padajoči teži $\bar{w}_j$ zagotavlja, da so v vsaki rešitvi elementi z najmanjšimi indeksi (torej tisti, ki so bili v nahrbtnik položeni prej) tisti, ki bodo dosegli večjo težo.

# 6 Opis kode

V prilogi sva dodala kodo napisano v *pythonu*, ki je ustrezno komentirana pri bolj zapletenih funkcijah, tukaj pa bova še podrobneje prikazala funkciji *solve_RKP* in *rekurzija*, ki sta glavni komponenti za izračun RKP.

```
def solve_RKP(N, c, w, p, gama = None,  max_w = None):
    najprej uredimo podatke po padajoči teži (max_w - w)
    če je max_w = None, uredimo podatke padajoče po teži (w)

    naredimo matrike Z, K, G ter nastavimo začetne pogoje
    element v prvi vrstici in prvem stolpcu matrike nastavimo na 0 vse ostale na minus neskončno:
    for d in range(c + 1):
        for s in range(gama + 1):
            Z[d][s]= float("-inf")
    Z[0][0] = 0
    analogno naredimo še matriki K in G

    izberemo j-ti predmet:
    for j in range(len(N)):
        j-ti predmet poskusimo dodati v nahrbtnik v svoji nominalni teži pri pogoju, da smo že vstavili gama predmetov
        for d in range(c, w[j]-1, -1):
            if Z[d - w[j]][gama] + p[j] > Z[d][gama]:
                Z[d][gama] = Z[d - w[j]][gama] + p[j]
                K[d][gama] = 1 + K[d - w[j]][gama]
                if j  >= ((len(N) / 2)):
                    G[d][gama] = 1 + G[d - w[j]][gama]

        j-ti predmet poskusimo dodati v svoji robustni teži:
        for s in range(gama, 0, -1):
            for d in range(c, maks_w[j] - 1, -1):
                if Z[d - maks_w[j]][s - 1] + p[j] > Z[d][s]:
                    Z[d][s] = Z[d - maks_w[j]][s - 1] + p[j]
                    K[d][s] = 1 + K[d - maks_w[j]][s-1]
                    if j  >= ((len(N) / 2)):
                        G[d][s] = 1 + G[d - maks_w[j]][s - 1]

        največji člen matrike Z je z*:
        z* = max([max(i) for i in Z])

        pogledamo kje v matriki se nahaja z*:
        pozicija = [[index, vrstica.index(z_zvedica)] for index, vrstica in enumerate(Z) if z_zvedica in vrstica]

        vrstica ter stolpec maksimalne vrednosti predstavljata skupno težo vstavljenih predmetov:
        c* = pozicija[0][0]
        število_predmetov_s_povečano_težo = pozicija[0][-1] #to je število predmetov, ki se jim je spremenila teža
```

Slika 1: Prvi del kode

Funkcija *solve_RKP* sprejme štiri obvezne argumente: množico predmetov $N$ oblike $\{1, \ldots, n\}$, kjer je $n$ število predmetov; kapaciteto nahrbtnika $c$; teže predmetov $w$; vrednosti predmetov $p$ in dva neobvezna argumenta $\Gamma$, ki označuje maksimalno število predmetov s spremenjeno težo ter $\bar{w} = max\_w$, ki prikazuje robustne teže predmetov. Vrne nam sledeče podatke: $z^*$ (optimalna vrednost predmetov, ki bodo v nahrbtniku), $c^*$ (optimalno težo predmetov v nahrbtniku), $k^*$ (število predmetov, ki jih damo v nahrbtnik) ter $g^*$ (število predmetov iz prve polovice množice $N$, ki jih dodamo v nahrbtnik).

Vse te optimalne vrednosti potrebujemo kasneje v funkciji *rekurzija*, da lahko dobimo seznam predmetov,ki jih bomo položili v nahrbtnik. V primeru ko ne podamo neobveznih argumentov $\Gamma$ in $max\_w$ nam funkcija $solve\_RKP$ vrne podobne optimalne vrednosti kot prej, le da tokrat reši KP.

# 7 Viri in literatura

## Literatura

[1] Michele Monaci, Ulrich Pferschy, Paolo Serafini
https://www.sciencedirect.com/science/article/pii/S0305054813001342