# Efektiven algoritem dinaminega programiranja pri problemu nahrbtnika z minimalnimi stroški in maksimalno zapolnitvijo

*Luka Vidic, Veronika Nabergoj*

*november 2017*

## Uvod

Predstavila bova problem nahrbtnika z danim naborom predmetov, ki imajo predpisano vrednost in težo. Skušala bova poiskati maksimalno napolnitev nahrbtnika z najmanjšo skupno vrednostjo vsebovanih predmetov. Napisala bova efektiven dinamičen algoritem, ki ima psevdo-polinomsko časovno zahtevnost.

---

## Kratek opis problema

S tem delom bova poskusila predstaviti poglobljeno študijo točnih metod reševanja MCMKP do dokazljive optimalnosti. Najprej, bova podala učinkovit dinamični algoritem za reševanje MCMKP, katerega časovna zahtevnost je $O(nC)$ in prostorska zahtevnost $O(n + C)$. Nato bova predstavila nove teoretične rezultate o stabilnosti enega izmed najboljših MPI (»mixed-integer programming« ) modelov iz literature. Kasneje bova v obsežni računski študiji na raznolikih primerih pokazala hitrost dinamičnega algoritma v primerjavi z vrhunskim komercialnim mešano celoštevilskim programskim (MIP) reševanjem.

### Klasični problem nahrbtnika

Najbolj znan problem v kombinatorični optimizaciji je problem nahrbtnika (PN), ki je definiran takole: Nahrbtnik s kapaciteto $C > 0$ in naborom predmetov $I = 1, \ldots, n$, z vrednostjo $p_i \geq 0$ in težo $w_i \geq 0$. Iščemo maksimalen profit, pri katerem teža nahrbtnika ne presega kapacitete nahrbtnika. Ta problem rešimo s programi mešanega celoštevilskega linearnega programiranja:

$$\max \left\{ \sum_{i \in I} p_i x_i : \sum_{i \in I} w_i x_i \leq C, x_i \in \{0, 1\}, i \in I \right\}, \tag{1}$$

kjer vsaka spremenljivka $x_i$ zavzame vrednost 1, če je i-ti predmet v nahrbtniku. Vsi parametri so cela števila. Optimizacijski problem, ki je podoben temu problemu je tudi najin problem nahrbtnika.

**Definicija 1:** ***Minimalni stroški, maksimalno število predmetov v nahrbtniku - MCMKP***

Z začetnimi podatki $(I, p, w, C)$, je cilj najti maksimalno napolnitev nahrbtnika $S^* \subset I$, ki minimizira vrednosti izbranih stvari.

$$S^* = \arg\max_{S \subset I} \left\{ \sum_{i \in S} p_i \middle| \sum_{i \in S} w_i \geq C \text{ and } \sum_{i \in S \setminus j} w_i < C, \forall j \in S \right\}$$

Pri problemu MCMKP privzamemo, da je problem $(I, p, w, C)$ netrivialen in je pripadajoča optimalna rešitev \$S\^* \$ lastna podmnožica množice $I$. Obratno kot pri klasičnem problemu nahrbtnika, pri MCMKP iščemo

lastno podmnožico predmetov, ki jih damo v nahrbtnik, pri čemer minimiziramo njihovo vrednost. Reševanje klasičnega PN, pri katerem maksimiziramo vrednost je pri MCMKP nadomeščena z minimiziranjem vrednosti ter s tem dobimo ničelno zapolnjenost. Zato mora biti pogoj maksimalne zapolnitve nahrbtnika izražen eksplicitno, ko iščemo optimalno rešitev. MCMKP lahko apliciramo na problem razporejanje nalog na enem stroju z istim časom dospetja in istim rokom. Ta je torej sestavljen iz izbiranja podmnožice nalog, katere moramo razvrstiti (pred iztekom roka) pri čemer vrstni red postane nepomemben. $I$ je množica vseh nalog, trajanje nalog je podano z $w_i$, strošek izvajanje naloge je $p_i$. Skupen rok je $C$, ki ustreza kapaciteti nahrbtnika v MCMKP. Cilj je minimiziranje stroška pri izvajanju izbranih nalog in hkrati maksimizirati podmnožico izbranih nalog. Tako se problem iskanja najboljšega razporeda prevede na problem nahrbtnika (MCMKP). Podobno, pri primeru MCMKP, je cilj maksimizirati »dobiček« razvrščenih nalog pri čemer želimo minimalizirati podmnožico nalog, ki presegajo rok (to je, ohraniti najmanjše pokritje).

---

## Lastnosti rešitve MCMKP-ja

V tem poglavju so opisane lastnosti, ki so privzete v dinamičnem algoritmu. Prva lastnost je, da so vsi predmeti razvrščeni v nepadajočem zaporedju glede na njihove teže, torej $w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_n$. Za vsak $i \in I$ definiramo $C_i := C - w_i$, tako dobimo $C_1 \geq C_2 \geq \cdots \geq C_3$. Prav tako pa lahko $w_n$ označimo tudi z $w_{max} := \max_{i \in I} w_i \ \ (= w_n)$.

Definiramo še $W := \sum_{i \in I} w_i$ ter $P := \sum_{i \in I} p_i$. Za vsako podmnožico $S \subset I$ je $w(s) = \sum_{i \in S} w_i$ in $W_i = \sum_{j < i} w_j$.

**Lastnost 1:** ***(Furini et al. (2015))***

V nekem naključnem, dosegljivem MCMKP z rešitvijo $S \subset I$, je kapaciteta navzdol omejena tako:

$$w(S) \geq C - w_{max} + 1.$$

Definiramo še *kritičen predmet*, to je predmet, katerega teža da natančno zgornjo mejo prvemu predmetu, ki ga izpustimo iz katerekoli dosegljive rešitve.

**Definicija 2:** ***MCMKP-jeva kritična teža in MCMKP-jev kritičen predmet***

Naj sedaj z

$$i_c := arg\min \left\{ i \in I \middle| \sum_{j \leq i} w_j > C \right\},$$

označimo indeks kritičnega predmeta. To je indeks prvega predmeta, ki preseže kapaciteto. Privzeto je, da bodo vsi predmeti z indeksi $i \leq i_c$ prav tako vsebovani. Kritična teža, označena z $w_c$ je teža kritičnega predmeta: $w_c := w_{i_c}$

**Trditev 1:** ***(Furini et al. (2015))***

Teža prvega izpuščenega predmeta katerekoli dosegljive MCMKP rešitve $S$ je navzgor omejena s kritično težo $w_c$ oziroma:

$$S \text{ je dosegljiva } \Rightarrow \min_{i \notin S} w_i \leq w_c$$

Posledično lahko rečemo, da je kapaciteta nahrbtnika omejena navzdol z

$$w(S) \geq C - w_c + 1.$$

**Dokaz:**

To lahko vidimo zelo preprosto: Predvidevamo, da obstaja taka dosegljiva zapolnjenost nahrbtnika $S \subset I$, da je teža najmanjšega predmeta, ki smo ga izpustili iz $S$, $> w_c$. V tem primeru mora S vsebovati vse predmete $i$, za katere velja $w_i \leq w_c$. Po definiciji $w_c$, je kapaciteta nahrbtnika, ki vsebuje seznam predmetov $S$, strogo večja od $C$, kar je protislovje.

**Trditev 2:**

Naj bo $S \subset I$ dosegljiva MCMKP rešitev. Če je predmet $i \in I$ i-ti prvi, katerega izpustimo iz $S$, imamo $S = \{1, \ldots, i-1\} \cup S^{'}$, kjer je $S^{'} \subseteq \{i+1, \ldots, n\}$, teža množice $S^{'}$ je prav tako spodaj omejena z:

$$w(S^{'}) \geq C - \sum_{j<i} w_j - w_i + 1.$$

**Dokaz:**

Predpostavimo, da velja $w(S^{'}) \leq C - \sum_{j<i} w_j - w_i + 1$. Če to drži, potem bi lahko $i$-ti predmet dodali v $S$, ne da bi prekoračili kapaciteto nahrbtnika (ker je $w(S^{'}) \geq 0$ in vsota na desni vsebuje $w_i$), po drugi strani pa trdimo, da je S maksimalno pakiranje nahrbtnika, kar je seveda protislovje.

**Lastnost 2:**

Kljub temu da imamo problem minimizacije, ni odveč definirati zgornjo mejo kapacitete nahrbtnika.

Spodnja trditev dobro opiše primer zakaj se poslužujemo zgornje meje

**Trditev 3:**

Če imamo $w_i = p_i$ za $\forall i \in I$, potem kapaciteta katerekoli optimalne rešitve $S$ ne presega $C$, tudi v primeru, ko tega pogoja ne določimo eksplicitno.

**Dokaz:**

Označimo z MSMKP relaksiran problem brez omejitve kapacitete nahrbtnika. Naj velja $w_i = p_i$, za $\forall i \in I$ in naj bo $S$ optimalna rešitev (minimalne moči) relaksiranega problema, tako da, $w(S) > C$. Naj bo $k$ indeks najlažjega predmeta iz $S$, to je, $k = \arg\min_{i \in S} w_i$. Sedaj sestavimo novo rešitev $S^{'}$ kot $S^{'} = S \backslash \{k\}$. Tako imamo $w(S^{'}) + w_i \leq C$ za nek predmet $i \in I \backslash S^{'}$, kajti v nasprotnem primeru bi bila $S^{'}$ maksimalna zapolnitev nahrbtnika z višjo končno vrednostjo kot $S$, kar pa je protislovje.

Naj bo sedaj $\delta = C - w(S^{'})$, kar je v bistvu preostanek celotne kapacitete nahrbtnika potem, ko odstranimo $k$ iz $S$. Sedaj bomo poskušali to luknjo zapolniti z reševanjem naslednjega podproblema:

$$\max \sum_{i \in I \backslash S^{'}} w_i x_i \qquad \text{tako da} \qquad \sum_{i \in I \backslash S^{'}} w_i x_i \leq \delta, \qquad x_i \in \{0, 1\}$$

Označimo s $S^{"}$ optimalno rešitev tega problema. Očitno velja $S^{''} \neq \emptyset$. Nova rešitev $\tilde{S} = S^{'} \cup S^{''}$ tako zadostuje naslednji lastnosti: $w(\tilde{S}) \leq C$ in (po definiciji) $w(\tilde{S}) + w_i > C$ za $\forall i \notin \tilde{S}$, to je, $\tilde{S}$ je še ena dosegljiva rešitev relaksiranega problema z višjo močjo kot $S$, kar pa je protislovje.

---

## Dinamičen algoritem za reševanje MCMKP

S podanimi trditvami in definicijami lahko sedaj oblikujemo dinamičen algoritem s katerim lahko rešujemo MCMKP. Ta algoritem je zasnovan na dejstvu, da ko enkrat poznamo najmanjši izvzeti predmet, se reševanje MCMKP reducira na problem nahrbtnika s spodnjo in zgornjo mejo kapacitete (v nadaljevanju označimo z LU-KP). To lastnost lahko izrazimo takole:

**Trditev 4:**

Optimalno rešitev MCMKP lahko izračunamo na sledeč način:

$$OPT = \min_{i \in I: i \leq i_c} \left\{ KP(i, \underline{C}, \bar{C}) + \sum_{j<i} p_j \right\}, \tag{2}$$

kjer je $\underline{C} = C - W_i - w_i + 1$ in $\bar{C} = C - W_i$, in

$$KP(i, \underline{C}, \bar{C}) = \min \left\{ \sum_{j>i} p_j y_j : \underline{C} \leq \sum_{j>i} w_j y_j \leq \bar{C}, y \text{ je binaren} \right\}. \tag{3}$$

**Dokaz:**

Naj bo $\tilde{X}$ optimalna rešitev MCMKP. Naj bo $i$ najmanjši izvzet predmet iz $\tilde{X}$. Zaradi tega vsi elementi $\{1, \ldots, i-1\}$ pripadajo $\tilde{X}$. Te predme lahko izvzamemo, če maksimalno kapaciteto nahrbtnika $C$ reduciramo na $\bar{C} = C - w_i$. Potem lahko vrednost $\tilde{X}$ pridobimo kot $KP(i, \underline{C}, \bar{C}) + \sum_{j<i} p_j$, kjer je LU-KP, definiran kot zgoraj, je obravnavan na preostalih predmetih iz niza $\{i+1, \ldots, n\}$. Ker mora biti rešitev $\tilde{X}$ največja zapolnitev, je minimalna kapaciteta definirana kot $\underline{C} = C - W_i - w_i + 1$ (glej Lastnost 2).

Vrednost optimalne rešitve MCMKP je pridobljena s poskušanjem vključitve vseh možnih predmerov $i \leq i_c$, in izberemo najboljšo izmed dobljenih rešitev.

### Opis programiranja

Na tem mestu predstaviva dinamični algoritem, ki teče v $O(nC)$ časovni zahtevnosti ter zavzame le $O(n + C)$ prostora. Spodaj je definiran algoritem `DinamicniAlgo(I, C)`.

Kot vhodne podatke bo ta algoritem sprejel `I <- data.frame(w = c(...), p = c(...))` teže in cene/profitov predmetov urejenih po naraščajoči teži, torej $w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_n$. Vektor $M$ velikosti $C + 1$ bo hranil sprotne rešitve problema $KP(i, 0, C)$. Za posodabljanje vrednosti vektorja $M$, algoritem sledi postopku reševanja problema nahrbtnika, kjer iščemo podmnožico predmetov, katerih skupna teža je natančno $k$ ($k \in \{0, 1, \ldots, C\}$), tako da je vsota cen/dobičkov minimizirana. V vsakem koraku $i$ ($i = n, \ldots, 1$), vrednost $M[k+1]$ hrani minimalen dobiček podmnožice predmetov, katerih skupna teža je natančno $k$, pri čemer upošteva le predmete iz množice $\{i+1, \ldots, n\}$. Ko enkrat dosežemo korak $i$, tako da velja $i \leq i_c$ (to je, vsi predmeti, manjši od $i$ so lahko v nahrbtniku), bo algoritem posodobil najboljšo vrednost rešitve po formuli (2).

```
source("./lib/libraries.r", encoding = "UTF-8")
DinamicniAlgo <- function(I,C,tabe){
  # tabe samo določa ali izpiše tabelo ali pa je ne
  # določim vrednost i_c
  indeks <- 0
  sum <- 0
  while(sum <= C){
    indeks <- indeks + 1
```

```
    sum <- sum + I$w[indeks]
  }
  i_c <- indeks
  tabela <- data.frame()
  # definiram glavni del algoritma
  M <- c()
  OPT <- c()
  M[1] <- 0
  OPT = Inf
  M[2:(C+1)] <- c(Inf)
  spodnja <- NA
  zgornja <- NA
  w_i <- c(0,cumsum(I$w))
  p_i <- c(0,cumsum(I$p))
  le <- c(length(I$w):1)
  for(i in le){
    if(i <= i_c){
      zgornja <- max(0, (C - w_i[(i)]))
      spodnja <- max(0, (C - w_i[(i)]- I$w[i] + 1))
      tmp <- min(M[(zgornja + 1):(1 + spodnja)] + p_i[(i)])   #dodam + 1, ker se M začne pri 1 namesto
      if(tmp < OPT){
        OPT <- tmp
        i_st <- i
        C_st <- which.min((M[(1+spodnja):(1+zgornja)] + p_i[(i_st)])) + spodnja - 1 + w_i[(i_st)]
      }
    }
    if(C>=I$w[i]){
      for(k in c(C:I$w[i])){
        M[k+1] <- min(M[k+1], M[k+1-I$w[i]] + I$p[i])
      }
    }
    if(tabe){
     tabela <- bind_rows(tabela, data.frame(i = i, p_i = I$p[i], w_i = I$w[i], t(M), C_l = spodnja, C_u
    }
  }
  if(tabe){
    names(tabela) <- c("i", "p_i", "w_i", t(paste0("C = ", seq(0:(length(M)-1))-1)), "C_l", "C_u", "P_i
  result <- list(OPT = OPT, C = C_st, i = i_st, tabela = tabela)
  }
  else{
    result <- list(OPT = OPT, C = C_st, i = i_st)
  }
  return(result)
}
```

Med izvajanjem algoritma lahko konstruiramo tabelo velikosti $O(nC)$, kjer vrednosti $M_i[k+1]$ predstavljajo najboljšo LU-KP vrednost rešitve, kjer upoštevamo le predmete iz $\{i+1,\ldots,n\}$ ter skupno kapaciteto $k$, $k \in \{0,\ldots,C\}$. V spodnji tabeli so ponazorjeni rezultati manjšega problema.

```
test_1 <- DinamicniAlgo(data.frame(w = c(1,1,2,3,4), p = c(3,4,6,2,1)),5,TRUE)
kable(test_1$tabela, caption = "Tabela 1", format = "latex") %>%
  kable_styling("striped") %>%
  add_header_above(c("Predmet" = 3, "Kapaciteta" = ncol(test_1$tabela)-8, " " = 5))
```

Table 1: Tabela 1

| Predmet | | | Kapaciteta | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | p_i | w_i | C = 0 | C = 1 | C = 2 | C = 3 | C = 4 | C = 5 | C_l | C_u | P_i | W_i | OPT |
| 5 | 1 | 4 | 0 | Inf | Inf | Inf | 1 | Inf | NA | NA | 15 | 7 | Inf |
| 4 | 2 | 3 | 0 | Inf | Inf | 2 | 1 | Inf | 0 | 1 | 13 | 4 | 13 |
| 3 | 6 | 2 | 0 | Inf | 6 | 2 | 1 | 8 | 2 | 3 | 7 | 2 | 9 |
| 2 | 4 | 1 | 0 | 4 | 6 | 2 | 1 | 5 | 4 | 4 | 3 | 1 | 4 |
| 1 | 3 | 1 | 0 | 3 | 6 | 2 | 1 | 4 | 5 | 5 | 0 | 0 | 4 |

**Trditev 5:**

Optimalno vrednost rešitve MCMKP lahko z `DinamicniAlgo` izračunamo v $O(nC)$ času.

**Dokaz:**

Ključen del novega dinamičnega algoritma je dejstvo, da dinamičen program za klasičen problem nahrbtnika poda vse potrebne informacije za računanje optimalne vrednosti rešitve MCMKP. Med procesiranjem $i$-tega predmeta (opomba: predmeti so izbrani po padajoči teži), lahko poiščemo optimalno vrednost LU-KP, kjer upoštevamo primer podmnožice predmetov $\{i+1,\dots,n\}$ za vse $k \in \{0,\dots,C\}$. To storimo z implementacijo sledeče rekurzije v algoritem `DinamicniAlgo()`:

$$M_i[k] := \begin{cases} \min\{M_i[k], M_{i+1}[k-w_i]+p_i\}, & \text{if } k \geq w_i \\ M_{i+1}[k], & \text{sicer} \end{cases} \forall i = n,\dots,1; k = 0,\dots C$$

kjer so začetne vrednosti $M_{n+1}[k] := \infty$ za vse $k = 1,\dots,C$ in $M_{n+1}[0] := 0$.

Izkoriščenje podobnosti med vrednostim rešitev MCMKP in LU-KP (*Trditev 5*), v vsakem koraku $i$ (začnemo s takim $i$, da velja $i \leq i_c$), lahko pridobimo optimalno vrednost rešitve MCMKP, pri čemer v rešitvi upoštevamo vse še ne koriščene predmete (tiste iz množice $\{1,\dots,i-1\}$). Vrednost te rešitve se hrani v spremenljivki *tmp*, usklajeno pa se posodablja tudi globalno optimalna rešitev. Opazimo, da predmeti $\{1,\dots,i-1\}$ določajo preostalo kapaciteto nahrbtnika (katera je dana s $\bar{C}$), ki jo je potrebno zapolniti s predmeti iz $\{i+1,\dots,n\}$, kjer je $\underline{C}$ določen na tak način, da bo zapolnitev maksimalna in da bo $i$-ti predmet najlažji izpuščen predmet. S tem zaključimo dokaz.

Če dobro pogledamo, opazimo, da nam algoritem `DinamicniAlgo(I,C)` vrne le optimalno vrednost $OPT$. V kolikor pa bi želeli imeti optimalno rešitev $S^* $, pa imamo na voljo dva načina. V škodo prostorske zahtevnosti (iz $O(n+C)$ na $O(nC)$), lahko optimalno rešitev skonstruiramo v času $O(n)$. Za konstrukcijo rešitve je dovolj hraniti informacijo o tem, ali je bil predmet $i$ dodan v rešitev LU-KP v trenutnem koraku ali ne, za vsako težo $k \in \{0,\dots,C\}$ in korak $i = n,\dots,1$. Hranimo sled kazalnikov $A_i[k+1] \in \{0,1\}$ definiranimi kot:

$$A_i[k+1] := \begin{cases} 1, & \text{če } M_i[k+1] = M_{i+1}[k+1-w_i]+p_i \\ 0, & \text{sicer (tj., če} M_i[k+1] = M_{i+1}[k+1]) \end{cases} \forall i = n,\dots,1, \qquad k = 0,\dots,C$$

Potem ko pridobimo optimalno vrednost rešitve $OPT$, lahko zračunamo tudi optimalno rešitev $S^* $, tako da uporabimo rahlo modificiran proces vzvratnega reševanja normalnega KP (*backtracking procedure*). Spomnimo se, da nam algoritem `DinamicniAlgo(I,C)` vrne kapaciteto $C^* $ zahtevano s strani optimalne rešitve. Najprej določimo $S^* = \{1,\dots,i^*-1\}$, nato, začnemo pri $A_{i^*}[C^* - W_{i^*}]$, nato nadaljujemo s procesom vzvratnega reševanja za KP.

## Dinamičen MIP algoritem za reševanje MCMKP

Ta algoritem sva zapisala v Sage-u MCMKP_MIP, ker je boljša izbira kot R studio. Podatke bova uvozila iz R v csv obliki ter z uporabo funkcije branja csv datotek uvozila podatke v Sage. V funkciji $MCMKP_MIP$

je definiran algoritem, ki nam za vhodne podatke vzame $C$-kapaciteto, $P$-vrednosti predmetov in $W$-težo predmetov, vrne pa optimalno vrednost nahrbtnika, $OPT$, seznam *vzamemo* na katerem so ničle in enke, ki povedo katere predmete smo vzeli v nahrbtnik, ter čas izvajanja algoritma za podane vhodne podatke. Čas izvajanja izmerimo s pomočjo vgrajene funkcije *time*. Čas odčitamo na začetku izvajanja algoritma s $s$ ter po izvedenem algoritmu ponovno preberemo čas $t$, nato pa odštejemo in dobimo čas trajanja.

### Opomba

V tem algoritmu je $C_i$, ki je bil definiran že prej, definiran kot $Csez$. Enako je $i_c$ v tem algoritmu definiran kot *kriticen* ter $x_i$ je definiran kot *vzamemo*.

```
def MCMKP_MIP(C,P,W):
    from time import time
    s = time()
    program = MixedIntegerLinearProgram(maximization=False,solver="GLPK")
    vzamemo = program.new_variable(binary=True)
    I=range(len(W))
    program.set_objective(sum(P[i] * vzamemo[i] for i in I))

    Csez = [C-x for x in W]

    #Določimo indeks kritičnega predmeta
    indeks= 0
    vsota= 0
    while(indeks< len(W) and vsota <=C):
        vsota=vsota+ W[indeks]
        indeks=indeks + 1

    kriticen=indeks - 1

    #Dodamo omejitve algoritma
    program.add_constraint(sum(W[i] * vzamemo[i] for i in I) <= C)
    for i in range(kriticen + 1):
        program.add_constraint(sum(min(W[j],Csez[i]+1)*vzamemo[j] for j in I if i!=j) + min(W[kriticen]

    #solve nam vrne optimalno rešitev OPT, get_values nam vrne seznam 1 n 0, ki povedo katere predmete
    program.solve()          #time nam meri čas, ki ga algoritem porabi za delovanje
    #program.show()                  #zapiše celoten program na bolj pregleden način

    resitev = program.get_values(vzamemo)
    koncniC = sum(resitev[i] * w for i, w in enumerate(W))
    t = time()
    trajanje = t-s # trajanje v sekundah
    #če bi želeli dodati še seznam predmetov, ki smo jih vzeli damo v return še program.get_values(vzam
    return (program.solve(),trajanje,koncniC)
```

### Prva omejitev

$$\sum_{i \in I} W(i) * vzamemo(i) \leq C$$

Ta omejitev nam pove, da teža vseh izbranih predmetov ne sme presegati kapacitete $C$. #### Druga omejitev

$$\sum_{j\in I, j\neq i} \min\{W(j), Csez(i)+1\}x_j + \min\{W(c), Csez(i)+1\}x_i \geq Csez(i)+1 \text{ za } \forall i \in I : i \leq i_c$$

Te neenakosti dobimo iz:

$$\sum_{j\in I, j\neq i} W(j)x_j + (Csez(i)+1)x_i \geq Csez(i)+1 \text{ za } \forall i \in I : i \leq i_c$$

To pa smo definirali že v Trditvi 1 in Lastnosti 2. Te nam zagotovijo, da če dodamo dodaten predmet $i$, za $i \leq i_c$, bo kapaciteta presežena.

Podatki, ki sva jih vnesla v funkcijo algoritma so bili uvoženi iz $R - studia$ in zapisani v $csv$ obliki. Prebrani so bili s sledečo kodo:

```
import csv
import sys

seznamresitev = []
for i in imena:
    f = open(i, 'rt')
    reader = csv.reader(f,delimiter = ';')
    M = matrix([[RR(a),RR(b),RR(c)] for a,b,c in reader])
    W, P = M.column(1), M.column(2)
    f.close()
    C=round((sum(W)*75)/100,0) #vzamemo 75% teže vseh predmetov
    seznamresitev.append(MCMKP_MIP(C,P,W))
seznamresitev
```

---

## Eksperimenti

Najin algoritem bova preizkusila na različnih podatkih, ter primerjala rezultate. Prva skupina podatkov je sestavljena tako, da so teže $w_i$ in cene/profit $p_i$ med sabo korelirani, v drugi skupini pa nekorelirani. Izbrala sva vzorce velikosti $\bar{N} \in \{10, 50, 100\}$. Teže predmetov bova izbrala naključno porazdeljene enakomerno na $w_i \in [1, R]$, kjer bova vzela $R \in \{10, 100, 1000\}$. Velikosti nahrbtnika pa definirava kot delež skupne teže predmetov, torej: $C \in \{\lfloor 25\%W \rfloor, \lfloor 50\%W \rfloor, \lfloor 75\%W \rfloor\}$. Podatki so bili generirani v R-u s sledečo kodo:

```
korelirani <- function(R = integer(), size = integer()){
  w_1 <- sample(x = 1:R, size = size, replace = TRUE)
  p_1 <- sapply(w_1, function(x){
    a <- sample((x-100):(x+100),size = 1, replace = TRUE)
    while(a<1){
      a <- sample((a):(a+100),size = 1, replace = TRUE)
      }
    a
    })
  I <- data.frame(w = w_1, p = p_1)
  I <- I[order(I$w, decreasing = FALSE),]
  return(I)
}
```

```
nekorelirani <- function(R = integer(), size = integer()){
  w_1 <- sample(x = 1:R, size = size, replace = TRUE)
  p_1 <- sample(x = 1:R, size = size, replace = TRUE)
  I <- data.frame(w = w_1, p = p_1)
  I <- I[order(I$w, decreasing = FALSE),]
  return(I)
}

# Podatke shranim v ./Podatki/...
R <- c(10, 100, 1000)
N <- c(10, 50, 100)
for(r in R){
  for(n in N){
    kor <- korelirani(r, n)
    write.csv2(x = kor, file = paste0("./Podatki/Korelirani_R", r, "_N", n, ".csv"), fileEncoding = "UTF
    nekor <- nekorelirani(r, n)
    write.csv2(x = nekor, file = paste0("./Podatki/Nekorelirani_R", r, "_N", n, ".csv"), fileEncoding =
  }
}
# uzela bova še večjo količino podatkov
R <- c(100, 1000)
N <- c(1000, 5000)
for(r in R){
  for(n in N){
    kor <- korelirani(r, n)
    write.csv2(x = kor, file = paste0("./Podatki/Korelirani_R", r, "_N", n, ".csv"), fileEncoding = "UTF
    nekor <- nekorelirani(r, n)
    write.csv2(x = nekor, file = paste0("./Podatki/Nekorelirani_R", r, "_N", n, ".csv"), fileEncoding =
  }
}
```

Preverimo ali so generirani podatki res korelirani oziroma nekorelirani:

```
R <- c(10, 100, 1000)
N <- c(10, 50, 100)
Kovariance <- data.frame()
korelirane <- data.frame()
nekorelirane <- data.frame()
for(r in R){
  vrsta1 <- c()
  vrsta2 <- c()
  for(n in N){
    assign(paste0("Korelirani_R", r, "_N", n), read.csv2(file = paste0("./Podatki/Korelirani_R", r, "_N
    assign(paste0("Nekorelirani_R", r, "_N",n), read.csv2(file = paste0("./Podatki/Nekorelirani_R", r,
    vrsta1 <- c(vrsta1, cor(get(paste0("Korelirani_R", r, "_N", n)))[2])
    vrsta2 <- c(vrsta2, cor(get(paste0("Nekorelirani_R", r, "_N", n)))[2])
  }
  vrsta1 <- t(vrsta1)
  vrsta2 <- t(vrsta2)
  names(vrsta1) <- N
  names(vrsta2) <- N
  korelirane <- bind_rows(korelirane, r = vrsta1)
  nekorelirane <- bind_rows(nekorelirane, r = vrsta2)
}
```

```
Kovariance <- bind_cols(korelirane, nekorelirane)
row.names(Kovariance) <- R
kable(Kovariance, format = "latex") %>%
  kable_styling("striped") %>%
  add_header_above(c("R"=1,"Korelirani" = 3, "Nekorelirani" = 3))
```

| R | Korelirani | | | Nekorelirani | | |
|---|---|---|---|---|---|---|
| | 10 | 50 | 100 | 101 | 501 | 1001 |
| 10 | -0.0785938 | 0.1870644 | 0.2176946 | -0.1684360 | 0.1942795 | -0.0114211 |
| 100 | 0.2887955 | 0.3423425 | 0.3213019 | -0.2985040 | -0.0509144 | 0.0723526 |
| 1000 | 0.9776895 | 0.9838431 | 0.9795622 | -0.1918498 | 0.0971616 | -0.0619735 |

Zanima naju, kateri od algoritmov je hitrejši, zato tukaj nastavimo parameter `tabe` v funkciji `DinamicniAlgo(I,C,tabe)` na `FALSE`, tako da se tokrat ne oblikuje tabela, da na ta način prihranimo nekaj prostora. Merjen bo tudi čas, potreben za izračun optimalne vrednosti pri maksimalni zapolnitvi.

```
Podatki <- list.files(path = "./Podatki/")[c(-1)]
Podatki_1 <- gsub(".csv", "", Podatki)
if(file.exists("./Rezultati/Results.csv")){
  listi <- read.csv2("./Rezultati/Results.csv", fileEncoding = "UTF-8", row.names = "X")
  Podatki_1 <- setdiff(Podatki_1, listi$name)
}
else{
  listi <- data.frame()
}
for(x in Podatki_1){
  cat(x, "\n")
  data <- read.csv2(paste0("./Podatki/", x, ".csv"), fileEncoding = "UTF-8", row.names = "X")
  C <- trunc(sum(data$w)*0.75)
  time <- system.time(res <- DinamicniAlgo(I = data, C, FALSE))
  listi <- bind_rows(listi,data.frame(name = c(x), Time=time[3], c(res), stringsAsFactors = FALSE))
}
write.csv2(x = listi, file = "./Podatki/Results.csv", fileEncoding = "UTF-8")
```

```
sage <- read.csv2(file = "./Rezultati/sa.csv", fileEncoding = "UTF-8", row.names = "X")
listi <- read.csv2("./Rezultati/Results.csv", fileEncoding = "UTF-8", row.names = "X")
primerjava <- left_join(listi, sage, by = "name") %>% mutate(res = Time.x - Time.y) %>% select("DP" = T
```

```
## Warning: Column `name` joining factors with different levels, coercing to
## character vector
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.2
```

```
kable(x = primerjava, caption = "Primerjava", format = "latex") %>%
  kable_styling("striped") %>%
  add_header_above(c("Čas merjen v sekundah" = 3))
```

Opazila sva, da se je v najinem primeru zgodilo, da je bilo računanje z `MIP` metodo hitrejše kakor z metodo `DP`, kar nasprotuje trditvam članka. Poglaviten razlog za to, bi lahko bil, da se računanje v programu Sage izvaja na oblaku z večjo kapaciteto in boljšimi procesorji. V ta namen sva naredila preprost test. V vsakem programu sva zagnala zanko for na seznamu dolžine 10 milijonov ter merila čas:

```
k <- 0
tim <- system.time(for(i in c(1:10000000)){
  k <- k+i
```

Table 2: Primerjava

| Čas merjen v sekundah | | |
|---|---|---|
| DP | MIP | Razlika |
| 0.02 | 0.0255091 | -0.0055091 |
| 0.93 | 0.1609769 | 0.7690231 |
| 0.19 | 0.0330830 | 0.1569170 |
| 0.04 | 0.0050721 | 0.0349279 |
| 9.31 | 0.3149869 | 8.9950131 |
| 797.27 | 397.0535870 | 400.2164130 |
| 1.89 | 0.0921609 | 1.7978391 |
| 20020.74 | NA | NA |
| 0.76 | 0.0043778 | 0.7556222 |
| 78.16 | 0.5202682 | 77.6397318 |

Table 3: Primerjava OPT in C

| Razlika v OPT | | | Zapolnjena kapaciteta | |
|---|---|---|---|---|
| DP | MIP | Razlika OPT | C DP | C MIP |
| 241 | 245 | -4 | 40 | 42 |
| 2139 | 2147 | -8 | 435 | 436 |
| 851 | 851 | 0 | 192 | 192 |
| 220 | 220 | 0 | 321 | 321 |
| 3801 | 3807 | -6 | 4291 | 4292 |
| 33556 | 33556 | 0 | 38087 | 38087 |
| 1872 | 1872 | 0 | 1874 | 1874 |
| 169576 | NA | NA | 191766 | NA |
| 3690 | 3690 | 0 | 3727 | 3727 |
| 34945 | 34945 | 0 | 37193 | 37193 |

```
})
sage_time <- 4.56
tabela2 <- data.frame("time_r"=tim[3],time_sage=sage_time)
print(tabela2)
```

```
##         time_r time_sage
## elapsed   0.92      4.56
```

Izkazalo se je, da je `R` hitrejši kot Sage, kar pa ne pomeni, da je dinamični algoritem počasnejši od `MIP`.

```
primerjava <- left_join(listi, sage, by = "name") %>% mutate(res = OPT.x - OPT.y) %>% select("DP" = OPT
```

```
## Warning: Column `name` joining factors with different levels, coercing to
## character vector
```

```
kable(x = primerjava, caption = "Primerjava OPT in C", format = "latex") %>%
  kable_styling("striped") %>%
  add_header_above(c("Razlika v OPT" = 3, "Zapolnjena kapaciteta" = 2))
```

Kot vidimo, se kapaciteta nahrbtnika razlikuje pri 3 eksperimentih. Vzrok temu je zaokroževanje, saj sva v `MCMKP_MIP()` uporabila funkcijo `round()` v `DinamcniAlgo()` pa funkcijo `trunc()`. Posledično se razlika pojavi tudi v `OPT`.

## Zaključek

Za nadaljevanje bi bilo smiselno končati še algoritma v skripti `Solve.r`, saj bi tako dobili še seznam predmetov v nahrbtniku pri optimalni rešitvi. Zanimivo bi bilo videti še, kako se algoritma obnašata na večji količini podatkov. Poskusila sva z večjim številom predmetov (N=5000), kar pa je zahtevalo 20.000 sekund, v Sage-u pa še celo več. Problem se nama je zdel zelo zanimiv, saj se lahko aplicira na življenjske primere.