Robustni problem nahrbtnika

Eva Babnik in Jan Založnik

December, 2020

Kazalo

Slike

Tabele

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 Γ.

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 Γ, 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 Γ 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 Γ 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 vseh bolj zapletenih funkcijah, tukaj pa bova še podrobneje prikazala funkciji *solve_RKP* in *rekurzija*, ki sta glavni komponenti za izračun RKP. Kot je že bilo omenjeno je zelo pomembno, da so predmeti razvrščeni po padajoči teži $\bar{w}_j$, zato je najlažje če napišemo dodatno funkcijo, ki sama razporedi podatke v pravilni vrstni red, da nam to ni potrebno delati ročno.

6.1 Solve_RKP

```python
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

        najdemo k* ter g1, ki pomeni koliko elementov iz N2(druga polovica predmetov v N) je v optimalni rešitvi:
        g1 = G[c*][število_predmetov_s_povečano_težo]
        k* = K[c*][število_predmetov_s_povečano_težo]
        g* = k* - g1
        return [z*, c*, k*, g*]
```

Slika 1: Skica kode *solve_RKP*

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 Γ, 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 Γ in max_w nam funkcija *solve_RKP* vrne podobne optimalne vrednosti kot prej, le da tokrat reši KP.

6.1.1 Primer Solve_RKP

Recimo, da imamo 10 predmetov, ki jih želimo položiti v nahrbtnik s kapaciteto 20 kilogramov. Recimo, da so podatki za te predmete so sledeči: $N = \{1, \ldots, 10\}$, $c = 20$, $w = [4, 2, 6, 5, 2, 1, 7, 3, 5, 2]$, $p = [8, 5, 17, 10, 14, 4, 6, 8, 9, 25]$, $\Gamma = 4$, $max\_w = [5, 4, 6, 7, 4, 4, 7, 4, 5, 3]$, kjer so podatki w in max_w predstavljeni v kilogramih, podatki p pa v evrih. Če sedaj ta problem rešimo s funkcjo *Solve_RKP* dobimo naslednjo rešitev:

```
>>> solve_RKP({1,2,3,4,5,6,7,8,9,10}, 20, [4,2,6,5,2,1,7,3,5,2], [8,5,17,10,14,4,6,8,9,25], 4, [5,4,6,7,4,4,7,4,5,3])
[66, 20, 4, 3]
>>>
```

Slika 2: Rešitev 1. primera

Funkcija *solve_RKP* nam vrne seznam iz katerega razberemo, da je optimalna rešitev 66 €, bomo v nahrbtnik položili 4 predmete od tega 3 predmete iz prve polovice množice predmetov N ter da bomo nahrbtnik čisto napolnili.

6.2 Rekurzija

Optimalni seznam stvari, ki jih vstavimo v nahrbtnik sva dobila s pomočjo funkcije $rekurzija$, ki sprejme argumente $N, z^*, k^*, c^*, \Gamma, w, \overline{w}_j, p$ in neobvezen argument *rešitev*, pri čemer sva množico stvari N razdelila na dva dela: $N = N_1 \cup N_2$, za katera velja $N_1 = \{1, ..., n/2\}$ in $N_2 = \{n/2 + 1, ..., n\}$. Za primer ko je n liho število, pa sva za elemente množice N_1 vzela prvih $n/2 + 1$ predmetov. Po izračunu optimalne vrednosti s pomočjo funkcije RKP sva optimalni seznam stvari dobila rekurzivno za vsako množico stvari N_i, pri čemer sva si pomagala še z dvema funkcijama, pri katerih gre za različici problema nahrbtnika; s funkcijo *Solve_KP*, ki reši navadni problem

```
def rekurzija(N, z*, k*, c*, gama, w, maks_w, p, vstavljeni_predmeti=None):
    if vstavljeni_predmeti is None:
        vstavljeni_predmeti = []
    if len(N) == 1 and vstavljeni_predmeti == []:
        return [N if gama != 0 and maks_w[0] <= c* or if gama == 0 and w[0] <= c*]
    elif len(N) == 1 and vstavljeni_predmeti != []:
        if gama != 0 and maks_w[0] <= c*:
                vstavljeni_predmeti.append(N[0])
                return(vstavljeni_predmeti)
        elif gama != 0 and maks_w[0] > c*:
                return(vstavljeni_predmeti)
        else:
            if w[0] <= c*:
                vstavljeni_predmeti.append(N[0])
                return(vstavljeni_predmeti)
    else:
        uredi predmete po padajoči vrednosti (maks_w - w)
        polovica =(len(N) / 2 )
        if n % 2 == 0:
           N1, N2 = N[:polovica], N[polovica:]
           w1, w2 = w[:polovica], w[polovica:]
           maks_w1, maks_w2 = maks_w[:polovica], maks_w[polovica:]
           p1, p2 = p[:polovica], p[polovica:]
        else:
           N1, N2 = N[:polovica + 1], w[polovica + 1:]
           w1, w2 = w[:polovica + 1], w[polovica + 1:]
           maks_w1, maks_w2 = maks_w[:polovica + 1], maks_w[polovica + 1:]
           p1, p2 = p[:polovica + 1], p[1+ polovica:]
        if k* >= gama:
            najdi tako kombinacijo c1 + c2 = c*, da bo
            z1(c1) + z2(c2) = z*, pri čemer z1(c1) ter z1(c2)
            dobimo kot:
            z1(c1) = RKP(N1, c1, w1, p1, gama, maks_w1)[0]
            z2(c2) = solve_KP(N2, c2, w2, p2)[1]
            solution_set_kp = solve_KP(N2, c2, w2, p2)[0]
            vstavljeni_predmeti.append(solution_set_kp)
            k1* = RKP(N1, c1, w1, p1, gama, maks_w1)[3]
            return rekurzija(N1, z1(c1) , k1*, c1, gama, w1, maks_w1, p1, vstavljeni_predmeti)
        else:
            najdi tako kombinacijo c1 + c2 = c*, da bo
            z1(c1) + z2(c2) = z*, pri čemer z1(c1) ter z1(c2)
            dobimo kot:
            z1(c1) = solve_eKkP(N1, c1, maks_w1, p1, k*)[1]
            z2(c2) = RKP(N2, c* - c1, w2, p2, gama - k*, maks_w2)[0]
            solution_set_eKkP = solve_eKkP(N1, c1, maks_w1, p1, k*)[0]
            k2* = RKP(N2, c2, w2, p2, gama - k*, maks_w2)[3]
            return rekurzija(N2, z2(c2), k2*, c2,gama - k*, w2, maks_w2, p2, vstavljeni_predmeti)
```

Slika 3: Skica kode *rekurzija*

nahrbtnika, ki je opisan na začetku poročila in s funkcijo *Solve_EkKP*, ki reši problem nahrbtnika z omejenim številom predmetov, ki je podobnen

navadnemu problemu nahrbtnika, le da tu dodamo še omejitev:

$$\sum_{i=1}^{n} x_i = k$$

kjer k pomeni dovoljeno število predmetov v nahrbtniku. Najna delitev problema da dva dela temelji na naslednji lemi:

Lema 1. Če je $k^* \geq \Gamma$, potem je optimalna rešitev sestavljena iz rešitve za robustni problem nahrbtnika s parametrom Γ za predmete iz množice N_1 in iz rešitve za navadni problem nahrbtnika za predmete iz množice N_2, pri čemer upoštevamo njihove nominalne teže. V primeru ko je $k* < \Gamma$, pa je optimalna rešitev sestavljena iz rešitev za *EkKp* s parametrom k^* za predmete iz množice N_1 z njihovimi maksimalnimi težami in iz rešitve za robustni problem nahrbtnika s parametrom $\Gamma - k^*$ za predmete iz množice N_2.

6.2.1 Primer *rekurzije*

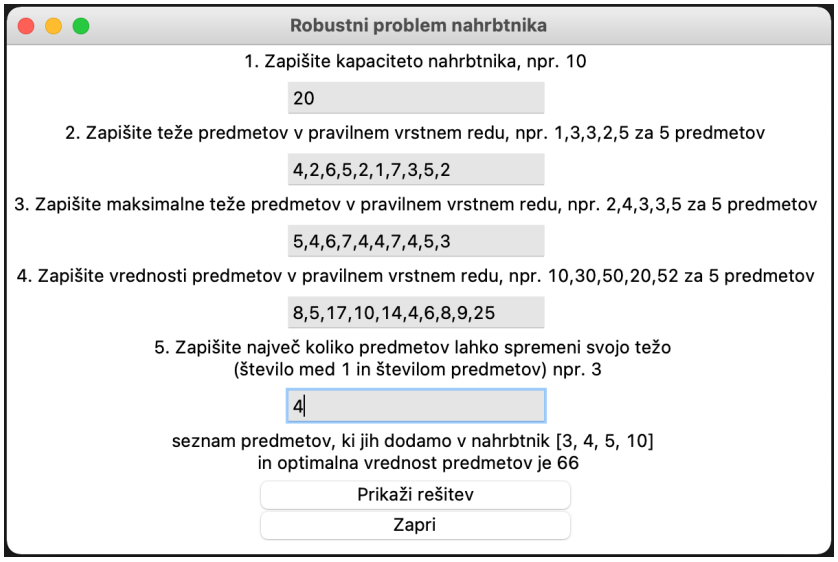
Recimo, da imamo iste podatke kot v primeru 6.1.1 in vrednosti z^*, c^* ter g^*, ki jih dobimo iz rešitve *solve_RKP*. Podatke vstavimo v funkcijo *rekurzija* in dobimo seznam predmetov, ki ji položimo v nahrbtink:

```
>>> rekurzija({1,2,3,4,5,6,7,8,9,10},66,3,20,4,[4,2,6,5,2,1,7,3,5,2],[5,4,6,7,4,4,7,4,5,3], [8,5,17,10,14,4,6,8,9,25])
[10, 4, 5, 3]
```

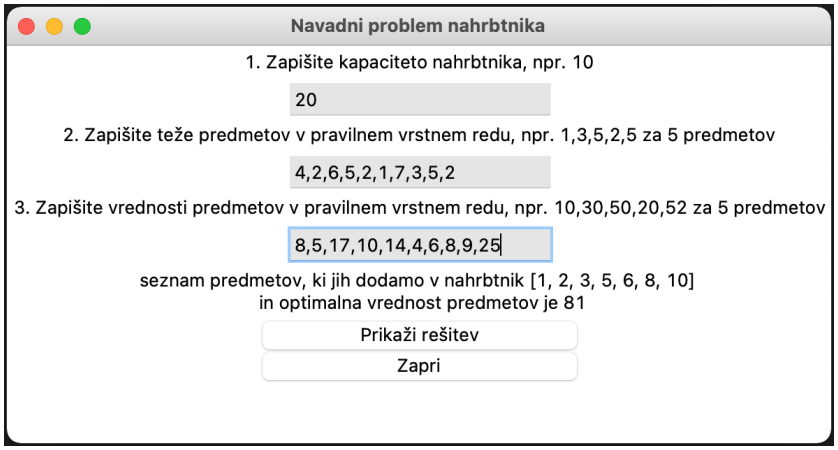
Slika 4: Rešitev rekurzije

7 Grafični vmesnik

V prilogi, natančneje v *koda_RKP* sva s pomočjo *pythonove* knjižnice *tkinter* naredila aplikacijo z grafičnim vmesnikom. Ta vas najprej vpraša, kakšen tip problema imate, torej ali je to KP ali RKP in vam nato izbrani problem na kratko predstavi in vam pove, kako pravilno vnesti podatke na naslednji strani.
Če zopet uporabimo primer iz 6.1.1, nam bo aplikacija izpisala seznam predmetov, ki jih postavimo v nahrbtnik in optimalno vrednost predmetov, kot je razvidno iz slike 5.a). Če vstavimo iste podatke še v okno navadni problem nahrbtnika s to razliko, da je $\Gamma = 0$ dobimo optimalno vrednost 81 in seznam predmetov, ki jih položimo v nahrbtnik.



(a) Robustni problem nahrbtinka



(b) Navadni problem nahrbtinka

Slika 5: Primerjava istega primera v robustni in klasični obliki

8 Časovna zahtevnost

Pogledali si bomo časovni zahtevnosti zgoraj opisanih funkcij, torej *solve_RKP* in *rekurzija*. Pri oceni časovne zahtevnosti funkcije *solve_RKP* si najprej poglejmo matrike z, k, g. Ker se nahajajo v istih zankah, je dovolj da si pogledamo, kaj se med izvajajem algoritma dogaja z matriko z, kjer gremo v zunanji zanki skozi vse elemente $j = 1, ..., n$. Za vsak j upoštevamo vse $s \leq \Gamma$, v notranji zanki pa gremo še skozi vse možne kapacitete nahrbtinka

		$c = 50$	$c = 100$	$c = 200$
$n = 500$	$\Gamma = 1$	0.9400	2.1294	7.4316
	$\Gamma = 10$	2.1714	9.5391	36.5587
	$\Gamma = 50$	7.4621	25.7643	97.5244
$n = 1000$	$\Gamma = 1$	1.4711	7.2800	16.7107
	$\Gamma = 10$	6.2347	23.3414	77.9351
	$\Gamma = 50$	11.1123	59.1479	302.4103
$n = 2000$	$\Gamma = 1$	4.3077	9.7543	37.6998
	$\Gamma = 10$	14.2596	52.3112	195.0519
	$\Gamma = 50$	26.1064	106.8547	700.8570

Tabela 1: Časovna zahtevnost algortima v odvisnosti od n, Γ in c.

$d = 0, ..., c$. Tako dobimo časovno zahtevnost $\mathcal{O}(\Gamma nc)$. Ko pokličemo funckijo *rekurzija* ta v primeru, ko je $k^* \geq \Gamma$ največ časa porabi pri klicanju funkcije *solve_RKP(*c^*, Γ, N_1*)*, ki porabi $\mathcal{O}(\Gamma \frac{n}{2} c^*)$ časa in pri klicanju funkcije *solve_KP* s stvarmi iz množice N_2, ki porabi $\mathcal{O}(\frac{n}{2} c^*)$ časa. V primeru, ko je $k^* < \Gamma$ pa funkcija največ časa porabi ko pokliče funkcijo *solve_EkKP* v času $\mathcal{O}(\Gamma \frac{n}{2} c*)$ in funckijo *solve_RKP* v času $\mathcal{O}(\Gamma \frac{n}{2} c^*)$. Časovna zahtevnost funkcije *rekurzija* znaša torej $\mathcal{O}(\Gamma nc)$.

V tabeli (1) je prikazana časovna zahtevnost algoritma v odvisnosti od Γ, c in n, čas pa je merjen v sekundah. Meritve sva izvedla na naključno generiranih podatkih, dobljenih s funkcijo *naredi_podatke*. Podatki se nahajajo v mapi *podatki_za_merjenje_casa*.

9 Optimizacija portfeljev s pomočjo RKP

Kodo za robustni problem nahrbtinka sva nato uporabila na poenostavljenem finančnem modelu, kjer naju je zanimalo iz katerih delnic sestaviti optimalni portelj, pri čemer so cene delnic negotove.

Recimo, da imamo M delnic, pri čemer imamo za vsako delnico j na voljo ceno P_j, visoko ceno $\overline{P}_j$ in nizko ceno $\underline{P}_j$, ter pričakovani donos delnice R_j. Zanima nas koliko delnic posameznega podjetja j kupiti, da bo pričakovani dobiček čim višji in da ob tem nakupu ne bomo presegli količine denarja B, namenjenega investiranju. Ob tem pa je potrebno tudi upoštevati, da največ Γ delnic spremeni svojo vrednost iz nominalne na maksimalno.Opisani primer lahko formuliramo na naslednji način:

$$max \sum_{j \in M} P_j(1 + R_j)N_j x_j$$

$$\sum_{j\in J} p_j N_j x_j + \sum_{j\in \hat{J}} \overline{P}_j N_j x_j \leq B, \quad \forall \hat{J} \subseteq J \text{ in } |\hat{J}| \leq \Gamma$$

$$p_j \in [P_j - \underline{P}_j, P_j + \overline{P}_j]$$

$$x_j \in \{0,1\}, j \in M$$

Da bi dobila rešitev tega modela sva kodi dodala funkcije *preberi_podatke_za_delnice*, *resitev_za_delnice* in *doloci_gamo*. Prva funkcija prebere datoteko, v kateri so za vsako delnico podatki o imenu, nominalni ceni, maksimalni ceni ter pričakovanem donosu. Funkcija nato za vsako delnico izračnua največ koliko enakih delnic po njihovi maksimalni ceni ob danem proračunu lahko kupimo in temu prilagdi sezname p, $maks_p$ in r. Recimo da je naš proračun enak 10 in maksimalna cena delnice i enaka 5, bomo torej delnico i šteli dvakrat, prav tako pa bomo v zgoraj omenjenih seznamih podvojili tudi njene vrednosti. Funkcija *resitev_za_delnice* pa pokliče funkcijo *rekurzija* in nato poišče katere delnice pripadajo dobljeni optimalni rešitvi.

Γ tu ni podana skupaj s podatki, ampak jo iz podatkov izračunamo s pomočjo funckije *doloci_gamo*. V tem modelu je Γ namreč slučajna spremenljivka in jo lahko ob nekaterih predpostavkah (ki sicer v realnosti ne držijo popolnoma) dobimo verjetnostno, pri čemer sva predpostavila, da je Γ binomsko porazdeljena. Najina ideja je bila torej ugotoviti, kdaj bo trenutna cena delnice P_j narasla na $\overline{P_j}$. Sklepala sva, da večji pričakovani donos R_j pomeni večjo verjetnost, da bo cena delnice v danem trenutku narasla na $\overline{P_j}$, zato sva izračunala povprečen $\overline{R}$. Predpostavila sva, da je $\overline{R}$ kar enak verjetnosti p, ki predstavlja verjetnost, da bo vrednost opazovanih delnic narasla. Nato sva s pomočjo funcije $random.binomial(stevilo_delnic, p)$ simulirala vrednosti slučajne spremenljivke Γ.

9.1 Uporaba RKP na delnicah $S\&P$ 500

Z zgoraj opisanim finančnim RKP modelom sva nato izračunala optimalni portfelj na podlagi o cenah in donosih delnic *S&P 500*, kjer gre za 500 največjih in najbolj likvidnih ameriških podjetij, kjer sva iskala maksimalni donos portfelja glede na različne proračune B. Ob tem je potrebno poudariti, da veva, da dobljene rešitve niso najbolj optimalne, saj ostalih dejavnikov, ki močno vplivajo na sestavo optimalnega portfelja (kot je npr. tveganje), sploh nisva upoštevala. V rezultatu dobimo zelo nerazpršen portfelj, kar ni idealno, saj ravno z diverzifikacijo zmanjšujemo tveganje.

Optimalne portfelje za delnice $S\&P$ 500 sva izračunala za proračune $B = 50$, $B = 100$, $B = 500$, $B = 1000$ in $B = 2000$. V tabeli (2) je prikazana

	B = 50	B = 100	B = 500	B = 1000	B = 2000
Čas	3.9939	8.0711	259.5398	931.2960	5063.1349
Sestava optimalnega portfelja	CARR: 1, AMCR: 1, HBAN: 1	CARR: 2, ZION: 1, HWM: 1, HFC: 1	Carr: 4, Etsy: 2 Nwsa: 1, Intc: 1	Etsy: 5, Carr: 4, Vtrs: 1	Wts: 7, Wrk: 1, Xyl: 1
Dobiček	123	272	1583	3246	3477

Tabela 2: Časovna zahtevnost finančnega RKP modela in dobljeni rezultati

časovna zahtevnost za posamezne primere ter sestava optimalnega portfelja, pri čemer so imena podjetji zapisana z njihovimi kraticami.

10 Viri in literatura

Literatura

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