PROJEKTNA DOKUMENTACIJA

TRANSPORTNI PROBLEM

Profesor:

Dr Mirna Kapetina

Anja Buljević

Studenti:

Vanja Babić IN 14/2017

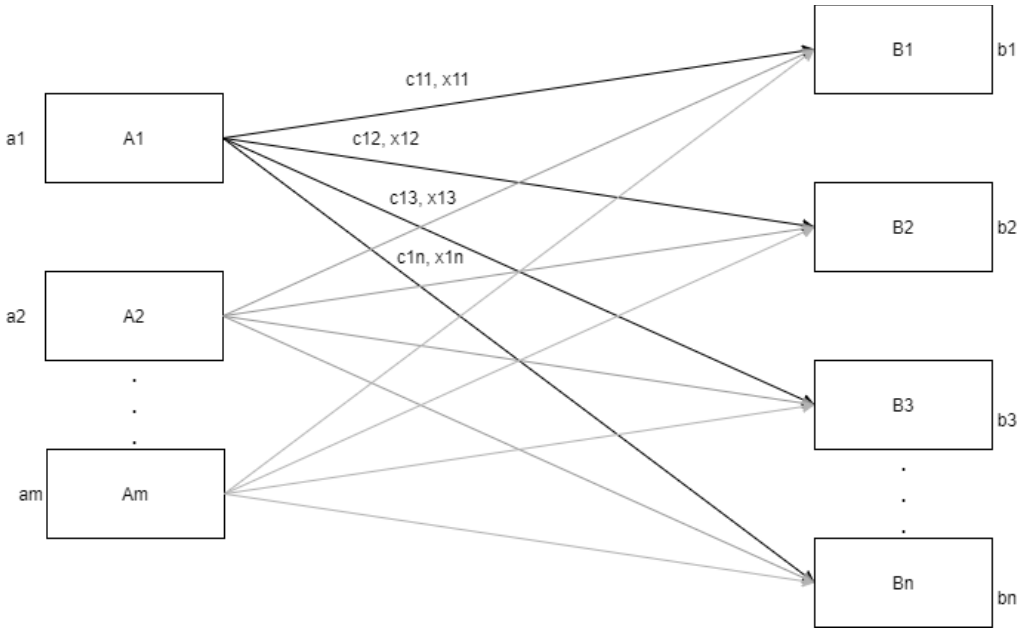
Dejan Dvornić IN 4/2017

Sadržaj

1. Uvod

1.1 Uvod u transportni problem

Transportni problem pripada grupi linearnih problema iz oblasti operacionih istraživanja. Pronalaženje optimalnog plana transporta uglavnom podrazumeva organizovanje transporta od tačke A (izvora) do tačke B (destinacije) takvo da troškovi budu minimalni.



Slika 1. Prikaz transportnog problema

Na slici 1 je moguće videti primer transportnog problema gde su:

Ai (i=1,...,m) izvori proizvoda, a sa ai su označeni njihovi kapaciteti

Bj (j=1,...,n) destinacije proizvoda, a sa bj su označeni njihovi kapaciteti

cij predstavljaju cene transporta od i-tog izvora do j-te destinacije

xij predstavljaju količinu transportovane robe

Dakle, cilj je rasporediti sve količine proizvoda sa izvora $\sum^{m}_{i=1}ai$ na odgovarajuće destinacije $\sum^{n}_{j=1}bj$ tako da funkcija cene bude minimalna.

Ako se sa F označe ukupni troškovi transporta, formulacija trasportnog problema kao linearnog problema glasi:

Min F = $\sum^{m}_{i=1}\sum^{n}_{j=1}$ cij * xij.

Ukupna cena transporta u svakom momentu jednaka je sumi proizvoda vrednosti isporučene robe xij i cene transporta cij.

Ograničenja koja moraju da važe:

x11+…+ xi1+…+ xm1 = b1
x1j+…+ xij+…+ xmj = bj
x1n+…+ xin+…+ xmn = bn

x11+…+ x1j+…+ x1n = a1
xi1+…+ xij+…+ xin = ai
xm1+…+ xmj+…+ xmn = am

xij>=0 za i=1,..,m j=1,..,n

Tačnije, nije moguće na neku destinaciju transportovati više proizvoda od zadatog kapaciteta destinacije, i analogno, iz bilo kog izvora nije moguće transportovati više robe nego što je zadato kapacitetom izvora. Količina transportovane robe od izvora do destinacije mora biti veća ili jednaka nuli.

1.2 Pregled rešavanja transportnog problema

Transportni problem se može rešavati na više načina:

1. simpleks metodom
2. specijalnim metodama za rešavanje transportnih problema linearnog problema.

Kod rešavanje transportnog problema specijalnim metodama razlikujemo više metoda za određivanje početnog rešenja:

1. Metoda severozapadnog ugla
2. Metoda najmanjih cena
3. Vogelova metoda

Da bi se rešio transportni problem, neophodno je napraviti matricu od prethodno datih podataka:

Izvor\Destinacija	B1	B2	…	Bn	Kapacitet izvora
A1	c11 x11	c12 x12	…	c1n x1n	a1
A2	c21 x21	c22 X22	…	c23 x2n	a2
…	…	…	…	…	…
Am	cm1 xm1	cm2 xm2	…	cmn xmn	am
Kapacitet destinacije	b1	b2	…	bn	

Prvi korak u rešavanju transportnog problema jeste određivanje početnog rešenja odnosno toga kolika količina proizvoda iz kog izvora će se transportovati do destinacije. Metodom severozapadnog ugla popunjavamo matricu tako što se krene od gornjeg levog ugla (severozapada) sve dok se kapacitet izvora ne iskoristi u potpunosti ili kapacitet destinacije ne iscrpi. Metodom najmanjih cena se prvo popunjava ono polje tabele sa najnižom cenom, potom se prelazi na sledeće najpovoljnije polje i opet tako sve dok se kapacitet izvora ne iskoristi u potpunosti ili kapacitet destinacije ne iscrpi. Vogelov metod je vrlo efikasan metod pronalaženja početnog rešenja. Često daje ne samo rešenje blizu optimalnog, već i optimalno rešenje. Ovaj algoritam se izvršava tako što se prvo računa za svaki red i svaku kolonu matrice razlika između dve najmanje cene. U narednom koraku izabere se red ili kolona

sa najvećom razlikom i u tom redu ili koloni se traži najniža cena transporta i popunjava polje. Potom se iterativno računaju razlike i popunjava tabela sve dok se svi kapaciteti ne iscrpe.

Nakon određivanja početnog problema vrši se optimizacija:

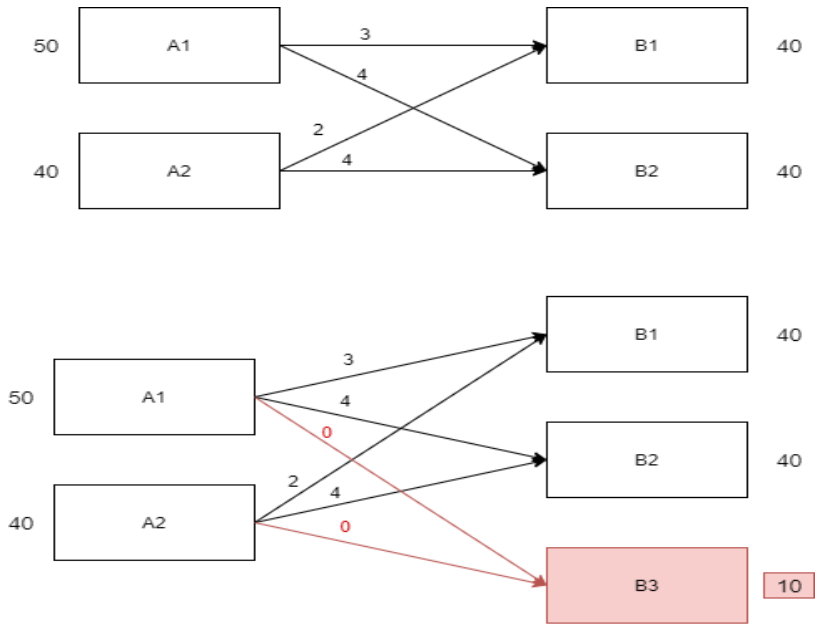
1. Svakom redu i svakoj koloni matrice se dodeljuje po jedan broj – potencijal. Potencijali redova su $u_i (i=1,...,m)$, a potencijali kolona su v_j $(j=1,...,n)$. Potencijali u_i i v_j se biraju tako da je njihova suma jedna odgovarajućoj ceni cij, odnosno: $c_{ij} - u_i - v_j = 0$, pri čemu indeksi (i, j) označavaju indekse opterećenog polja u matrici troškova.
2. Nakon utvrđivanja vrednosti potencijala ui i vj računaju se vrednosti c'_{ij} (težina) za prazna polja u matrici po formuli: $c'_{ij} = c_{ij} – u_i - v_j$.
3. Ukoliko su sve dobijene vrednosti za c'_{ij} pozitivne, onda je dobijeno rešenje optimalno i ne može se dalje poboljšati. Polja kod kojih je c'ij negativno pružaju mogućnost formiranja boljeg rešenja.
4. Kako se ne bi narušila ravnoteža početnog rešenja po redovima i kolonama, uvođenje nove komponente u matrici na mestu sa najnegativnijim c'_{ij} mora da se izvede unutar mnogougla, takvog da se na njegovim ćoškovima nađu samo opterećena polja. Potom se temena mnogougla označe sa + ili – naizmenično, s tim da polje od kog se kreće, tj. ono sa najnegativnijim c'_{ij} uvek bude obeleženo sa +. Među poljima označenim sa – bira se ono koje ima najmanje opterećenje i potom se ta vrednost oduzima od svih polja označenih sa -, a dodaje svim poljima označenim sa +.
5. Postupak se ponavlja sve dok se ne dođe do koraka u kom ni jedno c'_{ij} nije negativno.

2. Balansiranje problema

Sve prethodno navedene metode za rešavanje transportnog problema rade samo u slučajevima kada je transportni problem balansiran, odnosno, kada je suma svih kapaciteta izvora (ponuda) jednaka sumi kapaciteta destinacija (potražnji): $\sum^{m}_{i=1}ai = \sum^{n}_{j=1}bj$. Obzirom da se često dešava da u primeru ovaj slučaj nemamo, pre upotrebe bilo koje od metoda neophodno je izbalansirati problem. Dve opcije koje mogu da se dese jesu da je ukupna poduna veća od ukupne potražnje, i obrnuto, da je ukupna potražnja veća od ukupne ponude.

1.1 Ponuda veća od potražnje

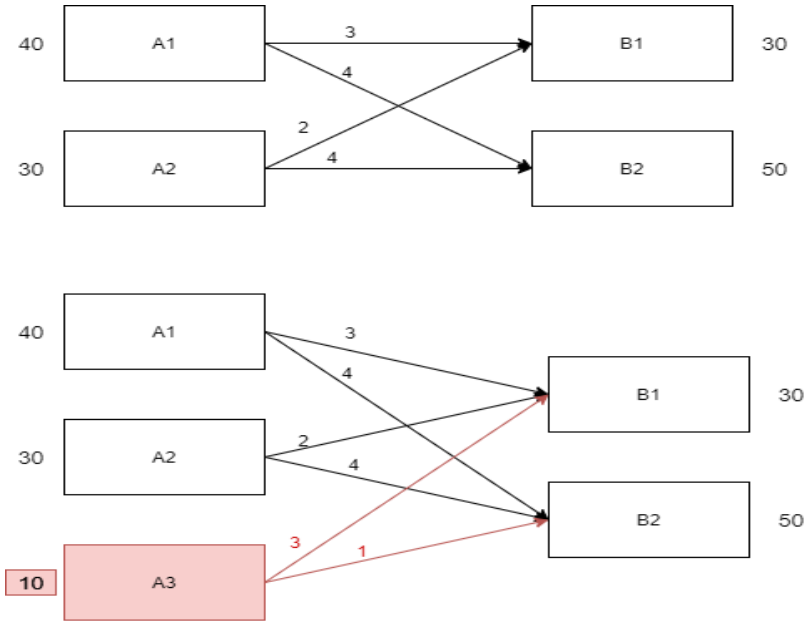
U slučaju kada je ukupna ponuda veća od ukupne potražnje, odnosno $\sum^{m}_{i=1}ai > \sum^{n}_{j=1}bj$, dodajemo jednu *lažnu* destinaciju kojoj dodeljujemo onoliki kapacitet kolika je razlika između ponude i potražnje, kako bismo izjednačili ukupnu ponudu i potražnju. Obzirom da se tu u stvari radi o proizvodima koji nikada neće biti iskorišteni, cena transpotra od izvora do te destinacije je 0.



Slika 2. Primer balansiranja transportnog problema u slučaju kada je ponuda veća od potražnje

1.2 Potražnja veća od ponude

U slučaju kada je ukupna potražnja veća od ukupne ponude, odnosno $\sum^{m}_{i=1}ai < \sum^{n}_{j=1}bj$, dodajemo jednan *lažni* izvor kom dodeljujemo onoliki kapacitet kolika je razlika između ponude i potražnje, kako bismo izjednačili ukupnu ponudu i potražnju. Obzirom da ovde imamo gubitke, odnosno, proizvodi koji su traženi neće biti i dostavljeni, cene transporta iz *lažnog* izvora neće biti jednake nuli, već će predstavljati penale, zbog određenih gubitaka.



Slika 3. Primer balansiranja transportnog problema u slučaju kada je potražnja veća od ponude

1.3 Implementacija balansiranja

Funkciji **balansiranje** prosleđujemo ponudu i potražnju u vektorskom obliku, matricu cena transpotra od izvora do destinacija, i penale za slučaj kada je potražnja veća od ponude. Na početku sumiramo sve vrednosti iz vektora ponude i potražnje kako bismo dobili ukupnu ponudu i potražnju. Potom se vrši provera da li je neka od ove dve vrednosti veća od druge i na osnovu toga kreira se nova ponuda, odnosno potražnja, u zavisnosti od slučaja i nova matrica cena. Ukoliko je primer u samom startu izbalansiran, funkcija će vratiti prosleđene vrednosti.

```
def balansiranje(ponuda, potraznja, cene, penali):
    ukupnaPonuda = sum(ponuda)
    ukupnaPotraznja = sum(potraznja)

    if ukupnaPotraznja > ukupnaPonuda:
        nova_ponuda = ponuda + [ukupnaPotraznja - ukupnaPonuda]
        nove_cene = cene + [penali]
        return nova_ponuda, potraznja, nove_cene
    if ukupnaPonuda > ukupnaPotraznja:
        nova_potraznja = potraznja + [ukupnaPonuda - ukupnaPotraznja]
        nove_cene = cene + [[0]*len(potraznja)]
        return ponuda, nova_potraznja, nove_cene
    return ponuda, potraznja, cene
```

Kada funkciji prosledimo vrednosti iz primera sa Slike 3:

```
ponuda = [40, 30]
potraznja = [30, 50]
cene = [[3, 4],
        [2, 4]]
penali = [3,1]
```

Vrednosti koje funkcija vrati izgledaju:

```
Ponuda: [40, 30, 10]
Potraznja: [30, 50]
Cene: [[3, 4], [2, 4], [3, 1]]
```

3. Izbor početnog rešenja

Kao što je već pomenuto, postoji više metoda pomoću kojih dobijamo početno rešenje transportnog problema, a u nastavku će biti obrađene implementacije:

1. Metode severozapadnog ugla
2. Vogelove metode
3. Metode najmanjih cena

3.1 Metoda severozapadnog ugla

Funkciji **severozapadni_ugao** prosleđujemo vektore ponude i potražnje, sačuvamo dužine početnih vektora kako bismo u nastavku ograničili rad algoritma (broj polja u matrici početnih rešenja ne može da bude veći od m+n-1 gde je m broj redova, a n broj kolona) i kreiramo brojače kojima ćemo prolaziti kroz vektore ponude i potražnje, tj pomoću njih će se pratiti indeksi polja u matrici. Na startu su vrednosti oba ideksa jedanka nuli obzirom da se radi o metodi severozapadnog ugla i da uvek krećemo od gornjeg levog polja u matrici, odnosno polja (0, 0). U svakoj iteraciji za trenutni red i kolonu manja vrednost od ponude i potražnje se oduzima i od trenutne ponude i od trenutne potražnje. Potom se čuva vrednost polja u matrici u kom je upisana vrednost, kao i sama vrednost i vrši se provera da li su svi resursi ponude iskorišćeni i ako jesu prelazi se na sledeći red, i analogno, ukoliko su svi resursi potražnje zadovoljeni, prelazi se u sledeću kolonu.

```
def severozapadni_ugao(ponuda, potraznja):
    i = 0
    j = 0
    pocetnoResenje = []
    ponudaCopy = ponuda.copy()
    potraznjaCopy = potraznja.copy()
    while len(pocetnoResenje) < len(ponuda) + len(potraznja) - 1:
        pon = ponudaCopy[i]
        potr = potraznjaCopy[j]
        v = min(pon, potr)
        ponudaCopy[i] -= v
        potraznjaCopy[j] -= v
        pocetnoResenje.append(((i, j), v))
        if ponudaCopy[i] == 0 and i < len(ponuda) - 1:
            i += 1
        elif potraznjaCopy[j] == 0 and j < len(potraznja) - 1:
            j += 1
    return pocetnoResenje
```

Kada funkciji prosledimo vrednosti ponude i potražnje koje smo u prethodnom koraku izbalansirali, povratna vrednost funkcije izgleda:

```
[((0, 0), 30), ((0, 1), 10), ((1, 1), 30), ((2, 1), 10)]
```

Tj. matrica početnog rešenja bi izgledala: [[30 10] [0 30] [0 10]]

3.2 Vogelova Metoda

Računanje početnog rešenja putem Vogelove metode dobijamo pomoću nekoliko funkcija. Prvi korak jeste računanje penala za redove i kolone tako što ćemo proslediti matricu a zatim je

sortirani ili transponovati pa sortirati u zavisnosti da li nam trebaju penali za kolonu ili red. Nakon toga uzimamo razliku izmedju 2 najmanja broja i računamo penale po kolonama i redovima.

```
def redPenali(cene, potraznja):
    cene1 = np.transpose(cene)
    cene1 = [sorted(i) for i in cene1]
    red_penal=[]
    for i in range(len(cene1)):
            if(potraznja[i] != 0):
                red_penal.append(cene1[i][1] - cene1[i][0])
            else:
                red_penal.append(0)
    return red_penal

def kolonaPenali(cene, ponuda):

    cene1=[sorted(i) for i in cene]
    kolona_penal=[]
    for i in range(len(cene1)):
            if(ponuda[i] != 0):
                kolona_penal.append(cene1[i][1]-cene1[i][0])
            else:
                kolona_penal.append(0)
    return kolona_penal
```

Medju tim penalima tražimo najveću vrednost i u koloni ili redu te najveće vrednosti tražimo najmanju vrednost cene i pamtimo i,j indekse odnosno njenu poziciju u matrici. Nakon toga tu i,j poziciji dodelimo vrednost koja odgovara vrednosti iz potražnje ili ponude. Uzimamo manju vrednost izmedju potražnje i ponude dok veću vrednost umanjujemo. Postupak se zaustavlja kada nam sve vrednosti iz potražnje budu 0.

```
def Uzmi_minimum_u_matrici_red(index_max, ponuda, potraznja, red, cene):
    ponuda_kopija = ponuda.copy()
    potraznja_kopija = potraznja.copy()
    i=0
    minimalni_index = i
    if(red == True):
        l = []
        for ind1 in range(0, len(cene)):
           l.append(cene[ind1][index_max])
        lm = min(l)

        minimalni_index = l.index(min(l))
        return minimalni_index,index_max
    else:
        print(cene[index_max])
        lm=min(cene[index_max])
        l=cene[index_max]
        minimalni_index = l.index(lm)

        return index_max, minimalni_index


def nadjiPocetnoResenje(pocetnoResenje, cene, ponuda, potraznja, i, j, red):

    if(ponuda[i] > potraznja[j]):
        pocetnoResenje[i][j]=potraznja[j]
        ponuda[i]=ponuda[i]-potraznja[j]
        potraznja[j]=0
        for ind in range(0,len(ponuda)):
          cene[ind][j]=99999

    else:
        pocetnoResenje[i][j]=ponuda[i]
        potraznja[j]=potraznja[j]-ponuda[i]
        ponuda[i]=0
        for ind in range(0, len(ponuda)):
            cene[i][ind] = 99999
    return pocetnoResenje,cene
```

3.3 Metod najmanjih cena

Funkciji **mincena** prosleđujemo ponudu, potražnju i cene. U matrici cena pronađemo najmanju vrednost, a potom i indekse polja u kom se ta vrednost nalazi. U svakoj iteraciji, u redu i koloni u kojima se nalazi najmanja vrednost cene, beleži se manja vrednost od ponude i potražnje.

```
def mincena(ponuda, potraznja, cene):
    i = 0
    j = 0
    ponudaCopy = ponuda.copy()
    potraznjaCopy = potraznja.copy()
    pocetnoResenje=[]
    ceneCopy = cene.copy()

    while len(pocetnoResenje) != len(ponuda) * len(potraznja):
        minimum = np.min(ceneCopy, axis=None)
        for ii, red in enumerate(ceneCopy):
            for jj, cena in enumerate(red):
                if cena == minimum:
                    i,j = ii, jj

        minVrednost = min(ponudaCopy[i], potraznjaCopy[j])
        pocetnoResenje.append(((i, j), minVrednost))
        ponudaCopy[i] -= minVrednost
        potraznjaCopy[j] -= minVrednost

        ceneCopy[i][j] = math.inf
```

4. Optimizacija

Nakon odabira početnog rešenja, neophodno je proveriti da li je dobijeno rešenje optimalno ili ga je potrebno optimizovati. Prvi korak u procesu optimizacije jeste određivanje potencijala u i v.

4.1 Određivanje potencijala

Funkciji **potencijali** se prosleđuje početno rešenje i matrica cena transporta. Kreiraju se prazni vektori u i v. Vektor u je iste dužine kao vektor ponude, a duzina vektora v je ista kao dužina vektora potražnje. Vrednost vektora u se za prvi red postavlja na nula. Iz početnog rešenja koristimo dobijene indekse, obzirom da potencijale postavljamo u odnosu na ona polja koja su popunjena u matrici. Za svako popunjeno polje pronađemo vrednost cene, i potom odredimo vrednost za nepoznato u ili v na osnovu formule cij - ui - vj = 0. Ukoliko je nepoznata vrednost i za u_i i za v_j, traži se sledeće popunjeno polje pomoću kog možemo da odredimo neko u_i ili v_j.

```
def potencijali(pocetnoResenje, cene):
    u = [None] * len(cene)
    v = [None] * len(cene[0])
    u[0] = 0
    cena=0
    pocetnoResenjeCopy = pocetnoResenje.copy()
    while len(pocetnoResenjeCopy) > 0:
        for indeksi, resenje in enumerate(pocetnoResenjeCopy):
            i, j = resenje[0]
            if u[i] is None and v[j] is None: continue

            cena = cene[i][j]
            if u[i] is None:
                u[i] = cena - v[j]
            else:
                v[j] = cena - u[i]
            pocetnoResenjeCopy.pop(indeksi)
            break
    return u, v
```

Povratna vrednost funkcije potencijali za početno rešenje dobijeno korišćenjem funkcije severozapadni ugao, za primer:

```
Ponuda: [40, 30, 10]
Potraznja: [30, 50]
Cene: [[3, 4], [2, 4], [3, 1]]
```

Izgleda:

```
u [0, 0, -3]
v [3, 4]
```

4.2 Određivanje c'_{ij} (težina)

Sledeći korak optimizacije jeste određivanje težina za polja koja nisu bazna, tj nisu popunjena u matrici početnog rešenja. Funkciji **tezine** prosleđuje se početno rešenje, cene, i vrednosti potencijala u i v. Pomoću iteratora i i j prolazimo kroz svako polje matrice i za svako polje čuva se vrednost cene. Potom se proverava da li se indeksi trenutnog polja nalaze među indeksima polja koja su u početnom

rešenju. Ukoliko se ne nalaze, znači da za to polje treba da odredimo težine pomoću već pomenute formule $c'_{ij} = c_{ij} – u_i - v_j$.

```
def tezine(pocetnoResenje, cene, u, v):
    tezine = []
    for i, red in enumerate(cene):
        for j, cena in enumerate(red):
            nijeBazno = all([index[0] != i or index[1] != j for index,vrednost in pocetnoResenje])
            if nijeBazno:
                tezine.append(((i, j), cena - u[i] - v[j]))
    return tezine
```

Povratna vrednost funkcije za primer na kom smo do sada radili, tj:

```
Ponuda: [40, 30, 10]
Potraznja: [30, 50]
Cene: [[3, 4], [2, 4], [3, 1]]

u [0, 0, -3]
v [3, 4]
```

i početno rešenje određeno pomoću funkcije severozapadni ugao izgleda:

```
[((1, 0), -1), ((2, 0), 3)]
```

tj, za polje u drugom redu i prvoj koloni vrednost c'_{ij} je -1, a za polje u trećem redu i prvoj koloni vrednost c'_{ij} je 3.

4.3 Provera težina

U sledećem koraku se vrši provera da li su sve vrednosti za c'_{ij} veće od nule, tj. da li je optimizacija uopšte potrebna. Ukoliko su sve vrednosti veće od nule tada smo stigli do optimalnog rešenja, a ukoliko je bar jedna težina manja od nule, rešenje je moguće optimizovati. Funkciji **optimizovati** prosleđujemo težine dobijene u prethodnom koraku, prolazimo kroz sve vrednosti težina i proveravamo vrednost.

```
def optimizovati(tezine):
    for index, vrednost in tezine:
        if vrednost < 0:
            return True
    return False
```

4.4 Početno polje

Kada se utvrdi da je postupak optimizacije neophodan, prvo je potrebno odrediti indekse polja od kog će početi da se kreira mnogougao. Funkciji **pocetnoPolje** prosledimo dobijene težine, pronađemo najmanju vrednost i indekse tog polja vratimo kao povratnu vrednost funkcije.

```
def pocetnoPolje(tezine):
    tezineA = tezine.copy()
    minVrednosti = []
    for index, vrednost in tezine:
        minVrednosti.append(vrednost)
    minVrednost = min(minVrednosti)
    for index, vrednost in tezine:
        if(vrednost == minVrednost):
            return index
```

4.5 Pronalazak petlje

Nakon što odredimo početno polje, tj. polje sa najnegativnijom težinom, sledeći korak jeste pronalazak petlje, mnogougla čije čvorove čine popunjena polja u matrici početnog rešenja. Prvo smo definisali pomoćnu funkciju **potencijalniCvorovi** koja određuje koja to sve polja potencijalno sledeća mogu da se nađu u petlji. Ako znamo koji je poslednji čvor petlje, onda sva polja koja se nalaze u njegovom redu i koloni mogu sledeća da se nađu u petlji. Ukoliko se poslednja dva čvora u petlji nalaze u istom redu (koloni) onda takođe znamo da sledeće polje ne može da bude u istom tom redu (koloni), i tada su potencijalni čvorovi samo čvorovi koji se nalaze u istoj koloni (redu) kao poslednji čvor petlje.

```
def potencijalniCvorovi(petlja, nijePosecen):
    poslednjiCvor = petlja[-1]
    cvoroviURedu = [n for n in nijePosecen if n[0] == poslednjiCvor[0]]
    cvoroviUKoloni = [n for n in nijePosecen if n[1] == poslednjiCvor[1]]
    if len(petlja) < 2:
        return cvoroviURedu + cvoroviUKoloni
    else:
        prepre_cvor = petlja[-2]
        potez_red = prepre_cvor[0] == poslednjiCvor[0]
        if potez_red:
            return cvoroviUKoloni
        return cvoroviURedu
```

Funkciji **pronadjiPetlju** prosleđujemo sve indekse u matrici početnog rešenja koja su popunjena, pocetno polje u petlji, odnosno, polje sa najnegativnijom težinom, i petlju, koja u početku sadrži samo početno polje. Ukoliko je dužina petlje barem 4, proveravamo da li se poslednji čvor u petlji i početno polje nalaze u istom redu ili kolini. Ako je to slučaj, petlju zatvaramo. Dalje, pronađemo sve čvorove koji nisu posećeni, odnosno sva ona polja koja se nalaze u početnom rešenju a još uvek nisu deo petlje. Potom za petlju, odnosno poslednji čvor u petlji odredimo koja to sve polja mogu sledeća da se nađu u petlji. Kada pronađemo ta polja, za njih ponovimo posupak sve dok ne dođe do mogućnosti da se petlja zatvori.

```
def pronadjiPetlju(pocetniIndexi, pocetnoPolje, petlja):
    if len(petlja) > 3:
        kraj = len(potencijalniCvorovi(petlja, [pocetnoPolje])) == 1
        if kraj:
            return petlja

    nisuPoseceni = list(set(pocetniIndexi) - set(petlja))
    moguciCvorovi = potencijalniCvorovi(petlja, nisuPoseceni)
    for sledeci_cvor in moguciCvorovi:
        sledecaPetlja = pronadjiPetlju(pocetniIndexi, pocetnoPolje, petlja + [sledeci_cvor])
        if sledecaPetlja:
            return sledecaPetlja
```

4.6 Novo rešenje

Nakon pronalaska petlje, sledeći korak jeste određivanje novog rešenja. Funkciji **novoResenje** prosledimo početno rešenje i pronađenu petlju. Potom pronađemo indekse onih polja u petlji kojima ćemo vrednosti oduzimati i one kojima ćemo dodavati. Počevši od prvog polja u petlji, koje je i početno

i za koje znamo da će mu se vrednost dodavati, svako drugo od njega takođe će povećati vrednost. Obrnuto, svakom drugom polju počevši od drugog zabeleženog u petlji će se vrednost smanjivati. Potom, među svim vrednostima polja koja se nalaze u grupi *minus* pronađemo ono koje ima najmanju vrednost i tu vrednost zabeležimo. To je vrednost koja će se oduzimati od svih ostalih polja u petlji, obzirom da vrednosti polja ne smeju da budu negativna. Potom svakom polju koje se nalazi u petlji, u zavisnosti od toga da li je u grupi *minus* ili *plus*, oduzimamo ili dodajemo već sačuvanu minimalnu vrednost polja koja se nalaze u petlji.

```
def novoResenje(pocetnoResenje, petlja):
    plus = petlja[0::2]
    minus = petlja[1::2]
    get_bv = lambda pos: next(v for p, v in pocetnoResenje if p == pos)
    minIndeks = sorted(minus, key=get_bv)[0]
    minVrednost = get_bv(minIndeks)

    novoResenje = []
    for p, v in [p for p in pocetnoResenje if p[0] != minIndeks] + [(petlja[0], 0)]:
        if p in plus:
            v += minVrednost
        elif p in minus:
            v -= minVrednost
        novoResenje.append((p, v))

    return novoResenje
```

5. Konačan algoritam

Na posletku, sve prethodno opisane funkcije spajamo u konačan algoritam koji vrši rešavanje transportnog rešenja. Funkciji **transportniProblem** prosleđujemo ponudu, potražnju, cene transporta i penale u slučaju da je potražnja veća od ponude. Potom sve parametre prosleđujemo funkciji **balansiranje**, kako bismo bili sigurni da radimo sa balansiranim primerom. Određujemo početno rešenje pomoću neke od metoda za određivanje početnog problema. Tako dobijeno početno rešenje i dobijene cene prosleđujemo funkciji **optimalno**, koja na početku određuje potencijale i težine. Potom sledi provera da li među težina postoji neka koja ima negativnu vrednosti. Ako ne postoji, prosleđeno rešenje je i optimalno. Ako postoji, pronalazi se petlja i kreira se novo rešenje za koje se ponovo proverava da li je optimalno ili ne.

Kada se dođe do optimalnog rešenja, određujemo i ukupan trošak transporta, odnosno za svako polje množimo cenu transporta i količinu robe koja se transportuje.

```
def transportniProblem(ponuda, potraznja, cene, penali):
    balansiranaPonuda, balansiranaPotraznja, balansiraneCene = (balansiranje(ponuda, potraznja, cene, penali))
    pocetnoResenje = severozapadni_ugao(balansiranaPonuda, balansiranaPotraznja)
    optimalnoResenje = optimalno(pocetnoResenje, balansiraneCene)

    matrica = np.zeros((len(balansiraneCene), len(balansiraneCene[0])))
    for (i, j), v in optimalnoResenje:
        matrica[i][j] = v
    trosakTransporta = odrediTrosak(balansiraneCene, matrica)
    return matrica, trosakTransporta

def optimalno(pocetnoResenje, cene):
    u,v = potencijali(pocetnoResenje, cene)
    cij = tezine(pocetnoResenje, cene, u, v)

    if(optimizovati(cij)):
        najnegativnije = pocetnoPolje(cij)
        petlja = pronadjiPetlju([p for p,v in pocetnoResenje], najnegativnije, [najnegativnije])
        return optimalno(novoResenje(pocetnoResenje, petlja), cene)
    return pocetnoResenje

def odrediTrosak(cene, resenje):
    suma = 0
    for i, red in enumerate(cene):
        for j, cena in enumerate(red):
            suma += cena*resenje[i][j]
    return suma
```

Primer koji smo prosledili ovoj funkciji izgleda:

```
cene = [
    [ 10, 12, 0],
    [8, 4, 3],
    [6, 9, 4],
    [7, 8, 5]]
ponuda = [20, 30, 20, 10]
potraznja = [10, 40, 30]
```

I dobijeno rešenje je:

```
Matrica:
 [[ 0.  0. 20.]
 [ 0. 30.  0.]
 [10.  0. 10.]
 [ 0. 10.  0.]]

Cena transporta: 300.0
```