Programiranje ograničenja

Milica Simić i Dubravka Kutlešić
milica7amg@gmail.com i dubravka97@gmail.com

3. novembar 2018.

1 Uvod

Programiranje ograničenja (eng. *constraint programming*) se predstavlja sistemom ograničenja nad nepoznatim promenljivim. Zadatak je naći rešenje. Tačnije, potrebno je odrediti vrednosti promenljivih koje zadovoljavaju postavljena ograničenja, uključujući slučaj kad ne postoje takve vrednosti.

Primer 1.1 *Korisnik želi da odredi vrednosti x, y, z takve da zadovljavaju:*
$x + y + z > 15$
$2x - 5y < 10$
$2y + 7z > 22$

Jedna moguće rešenje je:
$x = 0$
$y = 0$
$z = 16$

Ograničenja mogu da budu različitog tipa. Na primer, postoje ograničenja iskazne logike ($a \wedge b$), linearna ograničenja ($a > 1$), ograničenja nad konačnim domenom ($a \in \{3, 5, 8\}$)

2 Kako rešiti pitanje ograničenja?

Tradicionalni imperativni pristup koji ulaz obrađuje na unapred definisan način i izračunava izlaz ne mora da dovede do efikasnog rešenja zbog broja mogućnosti za koje treba proveriti da li su rešenja. Tačnije, korisnik može zadati ograničenja u kojima proveravajući sve mogućnosti dolazi do kombinatorne eksplozije. Programska paradigma uz pomoć koje opisujemo koja ograničenja rešenje treba da ispoštuje, a ne kako dolazimo do njega, naziva se deklarativna programska paradigma (eng. *rule-based language*) i ključna je kod programiranja ograničenja. Logički jezici su primer deklarativne paradigme. U njima se proizvoljnim redom uspostavljaju relacije i ograničenja nad promenjivim.[4]

U slučaju imperativne paradigme $x < y$ se procenjuje u tačno ili netačno, dok u paradigmi ograničenja zadaje relaciju između objekata x i y koja mora da važi.

3 Podrška za programiranje ograničenja

Podrška za ograničenja može biti ugrađena u programski jezik (npr. Oz, Kaleidescope). Za neke programske jezike podrška je data preko biblioteka.

Postoje različite bibloteke za pogramiranje ograničenja za jezike C, C++, JAVA, Phyton. Neke od biblioteka su IBM ILOG CPLEX, Microsoft 23...

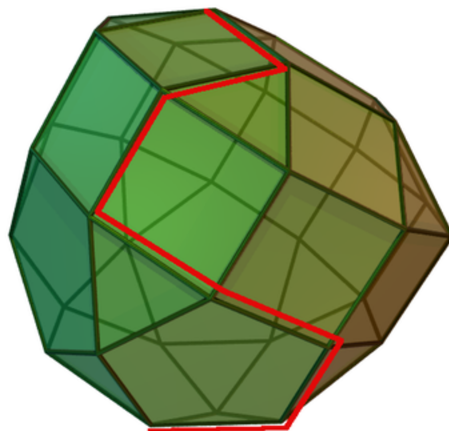
Većina Prolog implementacija uključuje jednu ili više biblioteka za programiranje ograničenja (B-Prolog,CHIP V5, Ciao, ECLiPSe, SICStus, GNU Prolog, Picat,SWI Prolog).

4 Linearno programiranje

Pretpostavimo da je cilj maksimizovati funkciju oblika $c^T x$ gde su c i x realni vektori u $\mathbb{R}^n$ tj. za zadati vektor $c = (c_1, c_2...c_n)$ pronaći $x = (x_1, x_2...x_n)$, ali tako da vektor x zadovoljava ograničenja oblika $Ax \leq b$ gde je A realna matrica dimenzija mxn i b realan vektor $b = (b_1, b_2..., b_m)$. Napomenimo da se problem u kom su ograničenja zadata u jednakosnom, a ne u nejednakosnom obliku može svesti na gore navedeni ubacivanjem dva nejednakosna ograničenja umesto jednakosnog ($Ax = b$ se menja sa $Ax \leq b$ i $Ax \geq b$). Matematički gledano, ograničenja oblika $Ax \leq b$ zadaju konveksni politop u $\mathbb{R}^n$ čije su odgovarajuće pljosni n hiperravni

$$\sum_{k=1}^{n} a_{ik} \dot{x}_k = b_i$$

za $i \in 1, 2, ..., m$. Vrednosti za x koje zadovoljavaju gore navedena ograničenja nalaze se unutar tog konveksnog politopa (Politop je dat na slici 1)



Slika 1: politop simplex algoritma u 3D

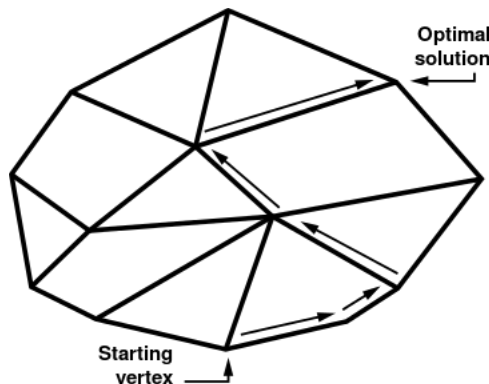
Može se pokazati da ako data funkcija dostiže najveću vrednost na granici politopa, onda tu vrednost dostiže u najmanje jednom temenu. S obzirom na to

da postoji konačan broj temena politopa, u ovom slučaju vreme izračunavanja se redukuje. [3]

Takođe, moguće je pokazati da ako se ekstremna vrednost ne dostiže u nekom temenu, postoji ivica, koja ga sadrži, duž koje vrednost funkcije strogo raste idući od tog temena duž pomenute ivice. Ako je ta ivica konačna, na drugom kraju se dostiže veća vrednost. Ukoliko je ta ivica beskonačna, uz pomoć vrednosti iz skupa ograničenja možemo dostići proizvoljno veliku vrednost funkcije i smatramo da je maksimalna vrednost ∞. [3]

Ukoliko ne dolazi do slučaja beskonačnih ivica, uz pomoć simplex algoritma i BFS-a je moguće odrediti maksimum funkcije. Dole navedeni primer detaljnije pojašnjava algoritam.

Primer 4.1 *Dat je graf $G = (V, E)$ koji odgovara politopa. Temena politopa su čvorovi grafa, a ivice politopa su ivice grafa. Za svaki čvor su poznate vrednosti funkcije koja se maksimizuje. Zadržavaju se ograničenja sa početka. Takođe, poznato je da maksimum funkcije nije ∞.*



Slika 2: obilazak grafa

Simplex algoritam opisuje kako se dolazi do temena sa najvećom vrednosti. Na početku se bira proizvoljno teme za početno i obilazi se graf. U poseban niz poseta za svako teme se čuva da li je posećen ili ne (1 ukoliko je posećen, 0 ukoliko nije). Proveravaju se vrednosti funkcije njegovih prvih suseda i prelazi se u najveći od njih, izmenivši niz poseta. Ovaj postupak se ponavlja dok se ne posete svi čvorove (dok niz poseta ne bude sadržao sve jedinice). Ovakav obilazak grafa je poznat kao BFS (eng. *Breadth first search*) ili obilazak grafa u širinu. Obilazak grafa je dat na slici 2.

Interesantno je da je problem celobrojnog linearnog programiranja (zadržava se postavka problema za obično linearno programiranje uz ograničenje da su elementi vektora x celi brojevi) značajno teži od običnog linearnog programiranja. Može se pokazati da je to NP-kompletan problem.

“ Klasa problema za koje postoji nedeterministički algoritam polinomijalne vremenske složenosti zove se NP. Problem X je NP-težak problem ako je svaki

problem iz klase NP polinomijalno svodljiv na X. Problem X je NP-kompletan problem ako pripada klasi NP, i X je NP-težak.” [6]

Za dati vektor x u polinomijalnom (ovde kvadratnom) vremenu je moguće proveriti da li ono zadovoljava ograničenja, dakle problem celobrojnog linearnog programiranja jeste NP.

Preostalo je neki poznat NP-kompletan problem svesti na ovaj. U primeru 4.2 to će se uraditi uz pomoć problema klika.

“Problem klika je NP-kompletan problem” [6]

Klika je podgraf grafa u kome su svi čvorovi iz klike povezani sa svim ostalim čvorovima iz klike tj. potpun podgraf. Svodeći klike na problem celobrojnog programiranja, dobijamo da on nije lakši od njega, tj. jeste NP-kompletan problem.

Primer 4.2 *Dat je graf* $G = (V, E)$, $|V| = n$, x_1, x_2, ... x_n *su čvorovi koje modelujemo tako da je* $x_i = 1$ *ako ulazi u kliku ili* $x_i = 0$ *ako ne ulazi u kliku. Dodatno ograničenje je da svaka dva čvora između kojih ne postoji grana važi da je* $x_i + x_j \leq 1$ *(jer ne mogu oba pripadati kliki). Potrebno je naći maksimum sume:*

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

.

Ovim je problem proizvoljne klike u grafu sveden na specijalni slučaj celobrojnog linearnog programiranja. Dakle, to nije lakši problem od problema klika. Iz definicije NP-kompletnosti i primera dobija se da je problem celobrojnog linearnog programiranja NP-kompletan.

5 Primene i primeri

Osnovna primena programiranja ograničenja su najčešće u operacionim istraživanjima kod rešavanja kombinatornih i optimizacionih problema. Navešćemo još neke interesantne primere gde je moguće iskoristiti pristup programiranja ograničenja. Ovi primeri NISU osnovna primena, ali pospešuju razumevanje:

- Kriptoaritmetike (matematičke igre u kojima se rešavaju jednačine kod kojih su cifre brojeva zamenjene određenim slovima), npr. $WRONG + WRONG = RIGHT$ [2]

- Rasporediti kraljice na šahovskoj tabli tako da se međusobno ne napadaju. [2]

- Rasporediti topove na šahovskoj tabli tako da se međusobno ne napadaju. [2]

- Rešiti sistem nejednakosti, npr:
  $x \in \{1, 2, ..., 100\}$
  $y \in \{1, 2, ..., 100\}$
  $x + y < 100$, $x > 10$, $y < 50$

- Odrediti položaje za najmanji broj predajnika tako da pokriva određeni prostor. [2]

- Definisati ponašanje semafora tako da protok saobraćaja bude najbolji. [2]

6 Programiranje ograničenja u Python-u

Modul python-constraint podržava programiranje ograničenja na konačnom domenu. [1]

Programiranje ograničenja nad konačnim domenom sastoji se od tri dela:

1. Generisanje promenljivih i njihovih domena
2. Generisanje ograničenja nad promenljivama
3. Obeležavanje (eng. *labeling*) - instanciranje promenljivih

U okviru Pythona rešenja se daju za sve promenljive tako da je instanciranje podrazumevano. [2]

Primer 6.1 (kriptoaritmetika) *Svakom karatkteru potrebno je dodeliti odgovarajuću cifru tako da je zadvoljena jednačina* $WRONG + WRONG = RIGHT$. *Različitim karakterima odgovaraju različite cifre. (Rešenje je dato na slici 3.)*

```
1  import constraint
2
3  problem = constraint.Problem()
4
5  # Definisemo promenljive i njihove vrednosti
6  problem.addVariables('WR',range(1,10))
7  problem.addVariables('ONGIHT',range(10))
8
9  # Definisemo ogranicenje za cifre
10 def o(w, r, o, n, g, i, h, t):
11   if(w*10000 + r*1000 + o*100 + n*10 + g + w*10000 + r*1000 + o*100 + n*10 + g)
12   == (10000*r + 1000*i + 100*g + 10*h + t):
13   return True
14
15 # Dodajemo ogranicenja za cifre na svim pozicijama
16 problem.addConstraint(o,"WRONGIHT")
17
18 # Dodajemo ogranicenje da su sve cifre razlicite
19 problem.addConstraint(constraint.AllDifferentConstraint())
20
21 resenja = problem.getSolutions()
```

Slika 3: rešenje primera 6.1

Primer 6.2 *Pekara proizvodi hleb i kifle. Za mešenje hleba potrebno je 10 minuta, dok je za kiflu potrebno 12 minuta. Vreme potrebno za pečenje se zanemaruje. Testo za hleb sadrži 300g brašna, a testo za kiflu sadrži 120g brašna. Zarada koja se ostvari prilikom prodaje jednog hleba je 7 dinara, a prilikom prodaje jedne kifle je 9 dinara. Ukoliko pekara ima 20 radnih sati za mešenje peciva i 20kg brašna, koliko komada hleba i kifli treba da se umesi kako bi se ostvarila maksimalna zarada, pod pretpostavkom da će pekara sve prodati? (Rešenje je dato na slici 4.)*

```
1  import constraint
2
3  problem = constraint.Problem()
4  # Dodajemo promenljivu H i definisemo njen domen
5  problem.addVariable('H', range(0,121))
6  # Dodajemo promenljivu K i definisemo njen domen
7  problem.addVariable('K', range(0,68))
8
9  def ogranicenje_vremena(h,k):
10     if 10*h + 12*k <= 1200:
11         return True
12
13 def ogranicenje_materijala(h,k):
14     if 300*h + 120*k <= 20000:
15         return True;
16
17 # Dodajemo ogranicenja vremena i matrijala
18 problem.addConstraint(ogranicenje_vremena, 'HK')
19 problem.addConstraint(ogranicenje_materijala, 'HK')
20 resenja = problem.getSolutions()
21
22 # Pronalazimo maksimalnu vrednost funkcije cilja
23 max_H = 0
24 max_K = 0
25
26 for r in resenja:
27     if 7*r['H'] + 9*r['K'] > 7*max_H + 9*max_K:
28         max_H = r['H']
29         max_K = r['K']
30 print "Maksimalna zarada je {0:d}, komada hleba je {1:d}, a komada kifli {2:d}"
31 .format(7*max_H + 9*max_K, max_H, max_K)
```

Slika 4: rešenje primera 6.2

Primer 6.3 (Ajnštajnova zagonetka) *U ulici stanuje 5 ljudi različitih nacionalnosti tako da svaki od njih živi sam i svaka kuća je različite boje. Kuće su poređane redom sleva nadesno. Svaki čovek pije određenu vrstu pića, puši određenu vrstu cigareta i ima kućnog ljubimca. Sve navedene karakteristike su različite za svakog od njih. Poznato je da:*

1. *Britanac živi u crvenoj kući*
2. *Šveđanin ima psa*
3. *Danac pije čaj*
4. *Zelena kuća je levo od plave kuće*

5. *Osoba koja puši Pall Mall poseduje pticu*
6. *Vlasnik žute kuće puši Dunhill*
7. *Čovek koji živi u centralnoj kući pije mleko*
8. *Norvežanin živi u prvoj kući*
9. *Čovek koji puši Blend živi pored osobe koja ima mačku*
10. *Čovek koji ima konja živi pored osobe koja puši Dunhill*
11. *Čovek koji puši BlueMaster pije pivo*
12. *Nemac puši Prince*
13. *Norvežanin živi pored plave kuće*
14. *Čovek koji puši Blend ima za suseda osobu koja pije vodu*

Čiji ljubimac je ribica? [5]
(Rešenje otkucano u Prolog-u je dato na slici 5.)

```
right(X, Y) :- X is Y+1.
left(X, Y) :- right(Y, X).
next(X, Y) :- right(X, Y).
next(X, Y) :- left(X, Y).
solution(Street, FishOwner) :-
    Street = [
           house(1, Nationality1,  Color1,     Pet1,   Drinks1,    Smokes1),
           house(2, Nationality2,  Color2,     Pet2,   Drinks2,    Smokes2),
           house(3, Nationality3,  Color3,     Pet3,   Drinks3,    Smokes3),
           house(4, Nationality4,  Color4,     Pet4,   Drinks4,    Smokes4),
           house(5, Nationality5,  Color5,     Pet5,   Drinks5,    Smokes5)],
    member(house(_, brit, red, _,_, _), Street),
    member(house(_, swede, _, dog, _, _), Street),
    member(house(_, dane, _, _, tea, _), Street),
    member(house(A, _, green, _, _, _), Street),
    member(house(B, _, white, _, _, _), Street),
    left(A, B),
    member(house(_, _, green, _, coffee, _), Street),
    member(house(_, _, _, birds,  _, pall_mall), Street),
    member(house(_, _, yellow, _, _, dunhill), Street),
    member(house(3, _, _, _, milk, _), Street),
    member(house(1, norweigan, _, _, _, _), Street),
    member(house(C, _, _, _, _, blend), Street),
    member(house(D, _, _, cats, _, _), Street),
    next(C, D),
    member(house(E, _, _, horse, _, _), Street),
    member(house(F, _, _, _, _, dunhill), Street),
    next(E, F),
    member(house(_, _, _, _, beer, bluemaster), Street),
    member(house(_, german, _, _, _, prince), Street),
    member(house(G, norwegian, _, _, _, _), Street),
    member(house(H, _, blue, _, _, _), Street),
    next(G, H),
    member(house(I, _, _, _, _, blend), Street),
    member(house(J, _, _, _, water, _), Street),
    next(I, J),
    member(house(_, FishOwner, _, fish, _, _), Street).
```

Slika 5: rešenje primera 6.3

Literatura

[1] Python Constraint library documentation. link: https://labix.org/python-constraint.

[2] Slajdovi profesorke Milene Vujošević Janičić. Kurs dizajn programskih jezika, link: http://www.programskijezici.matf.bg.ac.rs/dpj/2017/predavanja/02/skript_ogranicenja.pdf.

[3] George B. Dantzig. Origins of the simplex method. 1987.

[4] Sebesta Robert. *Concepts of programming language*. Pearson, 2009.

[5] Mattie Williams. Einstein's riddle. link: https://udel.edu/~os/riddle.html.

[6] Miodrag Živković. *Algoritmi*.