Unija intervala na realnoj pravoj, Can we do better?

Seminarski rad u okviru kursa
Konstrukcija i analiza algoritama 2
Matematički fakultet

Milan Čugurović, 1009/2018
milan_cugurovic@matf.bg.ac.rs

5. februar 2019

Sažetak

Tema ovog seminarskog rada jeste novi algoritam resavanja problema u odnosu na onaj koji je inicijalno predstavljen 1977. godine u radu Viktora Klea 'Can the Measure $\bigcup^n [a_i, b_i]$ be Computed in Less than $O(n \log(n))$ Steps?'
Novi pristup pokazuje se efikasniji ako se meri ukupno vreme izvrsavanja, iako i dalje ostaje u istoj asimptotskoj klasi slozenosti.

Sadržaj

1 Opis problema i istorijat

Problem nalaženja ukupne dužine unije intervala na pravoj pojavio se sedamdesetih godina prošlog veka. U smislu algoritmike jedan je od fundamentalnih problema. Rad na ovu temu objavio je već pomenuti Viktor Kle u radu 'Can the Measure $\bigcup^n [a_i, b_i]$ be Computed in Less than $O(n \log(n))$ Steps?'[2] koji je objavljen aprila 1977. godine u časopisu *The American Mathematical Monthly*, broj 84, na stranama 284. i 285., objavljen od strane Američke matematičke asocijacije.

Problem je inicijalno zamišljen tako što se na ulazu nalazi n intervala $[a_1, b_1]$, $[a_2, b_2]$, ..., $[a_n, b_n]$ na realnoj osi, i potrebno je naci dužinu njihove unije. U uvodnom pasusu pomenutog rada autor postavlja pitanje na koji način je moguće efikasno rešiti ovaj problem. Takodje, na istom mestu on predlaže i rešenje koje kaže: lista tačaka moze biti sortirana u $O(n \log(n))$ koraka, pa rešavanje onda zahteva samo dodatnih $O(n)$ koraka da se prodje kroz ovaj sortirani niz i izračuna dužina unije $\bigcup^n [a_i, b_i]$.

Pitanje koje muči autora rada na ovom mestu je to što ovaj problem, ne zahteva apriori sortiranje koje predstavlja glavnu komponentu složenosti [3] u radu predstavljenog rešenja.

Ovde se razmatra malo modifikovan problem. Naime, ne samo da se zahteva nalaženje dužine unije intervala, vec se zahteva i generisanje sortianih segmenata koji predstavljaju tu uniju. Na osnovu ovog dodatnog zahteva, prosiren je i algoritam predstavljen u radu, tako da ima i ovu dodatnu funkcionalnost.

2 Značaj problema

Pomenuti problem značajan je iz više razloga. Prirodno se pojavljuje u računanju površine datog podskupa euklidske ravni. Naime, pretpostavimo da su $[u_1, v_1]$, $[u_2, v_2]$, ..., $[u_n, v_n]$ dati intervali na realnoj osi, a f_i i g_i neprekidne funkcije na $[u_i, v_i]$, i

$$P_i = \{(x, y) : u_i \leq x \leq v_i, f_i(x) \leq y \leq g_i(x)\} \tag{1}$$

Čak i u situacijama kada površine pojedinačnih P_i možemo lako da izračunamo, računanje površine unije $\bigcup_1^n P_i$ nije tako jednostavno ako se P_i seku na netrivijalan način.

Međutim, ako su funkcije f_i i g_i Lipšicove (što nije redak slučaj), onda je tražena površina približno jednaka:

$$\sum_{j=1}^{n} (measure(L_j \cap P))(s_j - s_{j-1}), \tag{2}$$

gde je $s_0 < s_1 < ... < s_m$ odgovarajuća dovoljno gusta sekvenca iz $\bigcup_{i=1}^n [u_i, v_i]$ a L_j jeste apscisa $(s_{j-1} + s_j)/2$. Odatle

$$L_j \cap P = \bigcup_{i=1}^{n} L_j \cap P_i. \tag{3}$$

Računanje 2 uključuje m problema računanja dužne date unije intervala.

Drugi praktičan primer neophodnosti modifikovane verzije ovog algoritma jeste slučaj kada nam intervali stižu kontinuirano u određenim diskretnim vremenskim trenucima. Tada nam je, pored toga što nam u svakom trenutnu treba dužina unije do sada pristiglih intervala, treba i izlazni niz intervala, da bismo znali šta da činimo sa upravo pristiglim intervalom. U ovome se krije i ideja novog pristupa koji se prezentuje ovim seminarskim radom.

3 Rešenja problema

3.1 Rešenje iz 1977. godine

Rešenje prezentovano u referentnom radu iz 1977. godine nužno zahteva sortiranje datih intervala, što predstavlja njegovu osnovnu slabost. Neka su nam dati intervali $[a_i, b_i]$. Pretpostavimo da je svaka početna tačka intervala a_i označena sa 1, dok su krajnje tačke b_i označene sa -1. Koristeći sortiranje koje je loženosti $O(n\log(n))$, niz od ovih $2n$ tačaka može biti sortiran u rastući redosled $e_1 \leq e_2 \leq ... \leq e_{2n}$ tako da, ako je $t_1, t_2, ..., t_{2n}$ odgovarajuća permutacija oznaka, tada je je $t_i \geq t_{i+1}$ kada god je $e_i = e_{i+1}$. Drugim rečima, ako levi i desni krajevi neka dva intervala predstavljaju istu tačku, sortiranjem nikad desni kraj intervala neće doći pre levog kraja.

U nastavku je dat pseudokod algoritma koji prvo konstruiše uniju $[c_1, d_1], [c_2, d_2], ..., [c_m, d_m]$ prethodnih intervala $[a_i, b_i]$, a zatim računa dužinu njihove unije.

```
begin
  MEASURE ← 0; m ← EXCESS ← 0;
  for i ← 1 until 2n do
    begin
      EXCESS ← EXCESS + t_i;
      if EXCESS = 1 then begin m ← m + 1; c_m ← e_i end;
      if EXCESS = 0 then d_m ← e_i
    end;
  for i ← 1 until m do MEASURE ← MEASURE + (d_i − c_i)
end
```

Slika 1: Original pseudokod algoritma

Vezano za prethodni algoritam dve ključne tačke jestu promenljive EXCESS i MEASURE. Promenljiva MEASURE predstavlja promenljivu u kojoj ćemo čuvati meru preseka, dok promenljiva EXCESS služi kao indikator početka, odnosno kraja preseka.

Na slici 2 data je implementacija prethodnog algoritma, sa malim modifikacijama (kod vraća i uniju koju kreira) u programskom jeziku Python.

Pomenuta implementacija do detalja prati pseudokod 1 dat u originalnom radu, do na par izmena koje su neophodne radi savladavanja date male modifikacije problema koja se ovde razmatra.

```python
def calculate_lenghth_v1(dots):
  # dots - list length n of pairs (start_point, end_point)
  points = []
  new_points = []
  EXCESS = 0
  m = 0

  for dot in dots:
    points += [(dot[0], 1), (dot[1], -1)]  # startpoint: 1, endpoint: -1

  # First endpoint, then startpoint, if first num is the same
  points.sort(key=lambda x: (x[0], int(x[1])))

  for dot in points:
    EXCESS = EXCESS + dot[1]

    # when jump from 2->1 that isn't begining; (1,2) (2,4) don't want
    if EXCESS==1 and dot[1]==1 and (m==0 or dot[0]!=new_points[-1][1]):
      m += 1
      new_points.append([dot[0], 0])

    if EXCESS==0 and m>0:
      new_points[-1][1] = dot[0]

  return ([(dot[0], dot[1]) for dot in new_points], sum([dot[1]-dot[0] for dot in new_points]))
```

Slika 2: Python 3.0 implementacija

3.2 Novi pristup

Novi pristup zasniva se na sledećoj, veoma jednostavoj ideji. Naime, ideja jeste da se obrađuje interval po interval, dok se u svakom koraku tačno zna ne samo dužina do sada obrađenih intervala, već i njihova unija, kao sortiran niz intervala realne prave. U svakom koraku potrebno je dodatno, za novi pristigli interval pokrenuti dve binarne pretrage kojima se pronalaze pozicije levog i desnog kraja novog intervala u vec sortiranoj uniji. Dalje, na osnovu odnosa novih tačaka i dosadašnje unije, preduzimaju se odgovarajuči koraci.

Naime, baza indukcije jeste slučaj kada imamo samo jedan interval. Tada je izlaz algoritma upravo taj interval, dok je dužina unije u ovom slučaju razlika desnog i levog kraja istog.

Indukcijska hipoteza jeste, da za datih n intervala $[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]$ umemo da nadjemo kako dužinu njihove unije S_n tako i sortiranu listu tačaka $[c_1, c_2, ..., c_m]$ gde c_i jeste par $(a_j, True)$ ili $(b_k, False)$ u zavisnosti da li je to kraj ili početak nekog od intervala.

Indukcijski korak predstavlja obradu $n+1$-vog intervala $[a_{n+1}, b_{n+1}]$. Prvo se, pokreću dve binarne pretrage koje rade u složenosti $O(\log m)$ i koje pronalaze odgovarajuće pozicije tačaka $(a_{n+1}, True)$ i $(b_{n+1}, False)$ u nizu $[c_1, c_2, ..., c_m]$. U zavisnosti od medjusobnog polozaja novih tačaka i tačaka iz niza c_i vrši se eventualno dodavanje tačaka nizu i povećavanje dužine unije intervala S_{n+1} prema sledećim pravilima:
[1]

[1] Primera radi, neka je dosadašnja unija predstavljena kao $(0, 5), (7, 9), (11, 13)$ (na odgovarajući način, nizom c), i neka je interval koji se dodaje $(4.5, 8)$.
Tada, s' obzirom da se levi kraj novopristiglog intervala nalazi u opsegu od 0 do 5, a desni kraj istog u opsegu od 7 do 9, niz c se modifikuje, i nakon ovog dodavanja postaje $(0, 9), (11, 13)$.
Slično za ostale situacije.

```
def add(self, dot):
  if self.n==0:
    self.dots.insert(0, (dot[0], True))  # True mean (
    self.dots.insert(1, (dot[1], False)) # False mean )
    self.n += 2
    return

  i = self.__bsearch(dot[0])
  j = self.__bsearch(dot[1])

  if i==0 and j==self.n:
    self.dots.clear()
    self.n = 0
    self.add(dot)
  elif i==0:  # j!=n
    if self.dots[j][1]==True:
      self.dots.insert(0, (dot[0], True))
      self.dots.insert(j+1, (dot[1], False))
      del self.dots[1:j+1]
      self.n = self.n + 2 - j
    else:
      self.dots.insert(0, (dot[0], True))
      del self.dots[1:j+1]
      self.n = self.n +1 - j
  elif j==self.n:  # i!=0
    if self.dots[i-1][1]==False:
      self.dots.insert(i, (dot[0], True))
      self.dots.insert(j+1, (dot[1], False))
      del self.dots[i+1:j+1]
      self.n = self.n + 2 - (j-i)
    else:
      self.dots.insert(j, (dot[1], False))
      del self.dots[i:j]
      self.n = self.n + 1 - (j-i)
  else:  # i, j 'in' list
    if self.dots[i-1][1]==False and self.dots[j][1]==True:
      self.dots.insert(i, (dot[0], True))
      self.dots.insert(j+1, (dot[1], False))
      del self.dots[i+1:j+1]
      self.n = self.n + 2 - (j-i)
    elif self.dots[i-1][1]==True and self.dots[j][1]==True:
      self.dots.insert(j, (dot[1], False))
      del self.dots[i:j]
      self.n = self.n + 1 - (j-i)
    elif self.dots[i-1][1]==False and self.dots[j][1]==False:
      self.dots.insert(i, (dot[0], True))
      del self.dots[i+1:j+1]
      self.n = self.n + 1 - (j-i)
    else:
      del self.dots[i:j]
      self.n = self.n - (j-i)

```

Slika 3: Implementacija koraka obrade novog, $n + 1$-vog intervala

Implementacija nove ideje data je kroz implementaciju klase 'Intervaler' ciji se izvorni kod moze naci u pratećim materijalima ovog dokumenta. Klasa implementira prethodni algoritam dodavanja kroz metodu 'add', ali i ostale dodatne neophodne funkcionalnosti.

4 Implementacija

4.1 Specifikacija

Seminarski rad je implementiran kao Jupyter notebook u programskom jeziku Python, verzija 3.0.
Kod je izvršavan na Google colab[1] serveru, čije su specifikacije:

- 2 core CPU
- each core: Intel(R) Xeon(R) CPU @ 2.30GHz
- 13GB RAM

Detaljne specifikacije sistema mogu se videti u pratećem notebook kodu.

4.2 Merenje vremena izvršavanja

Merenja su vršena nad intervalima koji pripadaju uniformnoj raspodeli na segmentu $[-10, 10]$, pri čemu je i dužina pojedinačnog intervala generisana iz uniformno na itervalu $[0, 5]$.

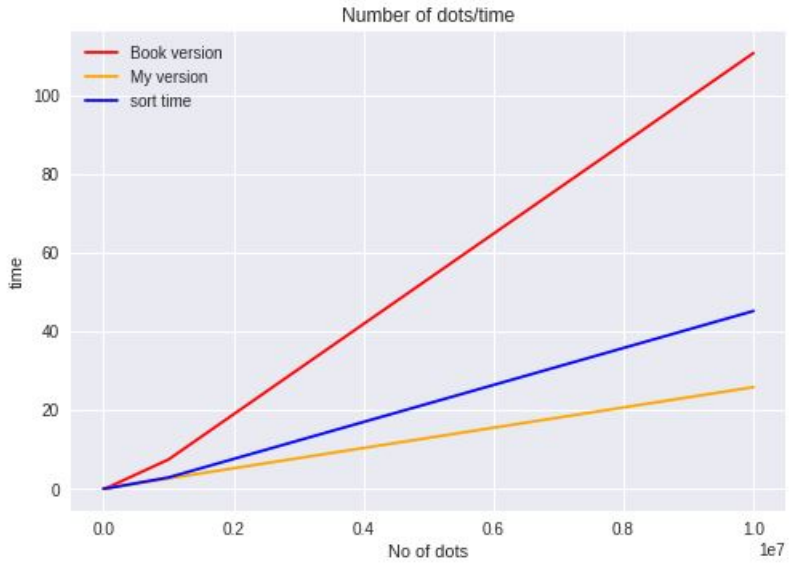
Rezultati merenja dati su sa (n predstavlja broj intervala):

Broj tačaka	Originalna ideja	Nova ideja
10	1.64ms	643ms
100	1.8ms	1.67ms
10 000	37.7ms	31.3ms
1 000 000	51ms	25.9s

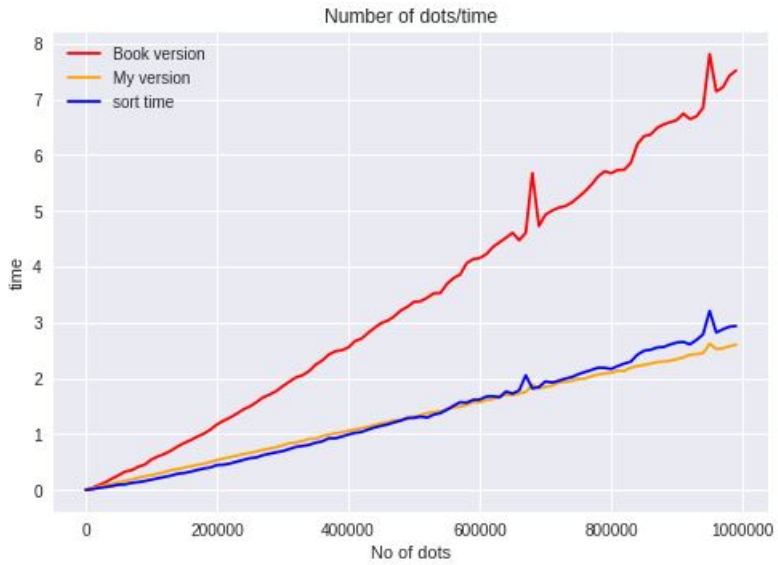
Na osnovu prethodnih rezultata zapažamo veoma veliko ubrzanje novog metoda u slučaju dosta gustog [2] skupa intervala.

Dodatno u nastavku je dat grafički prikaz rezultata nekih merenja vremena izvršavanja algoritama. Više informacija nalazi se u pratećem notebook dokumentu.

[2] U smislu kada je broj intervala koji predstavlja disjunktnu uniju ulaza m, dosta manji od broja ulaznih intervala n



Slika 4: Merenje performansi kada se za broj tačaka uzimaju redom brojevi liste $[1, 10, 100, 1000, 10000, 100000, 1000000, 10000000]$



Slika 5: Merenje performansi kada se za broj tačaka uzimaju brojevi počev od 10 a zaključno sa 1 000 000, sa korakom 10 000

Literatura

[1] Google Colaboratory. https://colab.research.google.com/.

[2] V. Klee. *Can the Measure* $\bigcup^n [a_i, b_i]$ *be Computed in Less than O(n*log(n)) Steps?* Taylor and Francis, Ltd. on behalf of the Mathematical Association of America, 1977.

[3] Vesna Marinkovic. Konstrukcija i analiza algoritama 2 http://poincare.matf.bg.ac.rs/~vesnap/kaa2/kaa2.pdf, 2018.