

# Profajleri i njihova vizualizacija u jeziku Python

Seminarski rad u okviru kursa  
Metodologija stručnog i naučnog rada  
Matematički fakultet

Andželka Milovanović, David Popov

Jelisaveta Smiljanić, Petar Zečević

mi15145@alas.matf.bg.ac.rs, mi16102@alas.matf.bg.ac.rs

mi16138@alas.matf.bg.ac.rs, mi16169@alas.matf.bg.ac.rs

31. mart 2020.

## Sažetak

Tema ovog seminarskog rada je vezana za profajlere u programskom jeziku *Python*. Osnovni cilj rada je da se čitaoci upoznaju sa različitim alatima za profajliranje, njihovim mogućnostima i praktičnom primenom kroz primere u jeziku Python. Takođe, jedan od ciljeva je da se čitalac ubedi zašto je korisno koristiti profajlere i kako oni mogu značajno pomoći u analiziranju napisanog koda i njegovog efikasnijeg izvršavanja.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Šta je profajliranje?</b>	<b>2</b>
<b>3</b>	<b>Načini profajliranja u jeziku Python</b>	<b>3</b>
3.1	Moduli cProfile i pstats . . . . .	4
<b>4</b>	<b>Alati za vizualizaciju profajliranja</b>	<b>5</b>
4.1	Py-Spy . . . . .	5
4.2	SnakeViz . . . . .	6
4.3	Pycallgraph . . . . .	8
4.4	Gprof2dot . . . . .	8
4.5	Vprof . . . . .	10
<b>5</b>	<b>Zaključak</b>	<b>11</b>
	<b>Literatura</b>	<b>12</b>
<b>A</b>	<b>Dodatak</b>	<b>13</b>

# 1 Uvod

Optimizacija koda je značajan deo razvoja svih ozbiljnijih i kompleksnijih softvera [4]. Postavlja se pitanje koje delove koda treba razmatrati za ovakve promene, kada kod treba razmatrati i da li ga uopšte treba razmatrati. Kod softvera složenijeg dizajna i arhitekture, teško je intuitivno zaključiti šta treba optimizovati. Upravo sa ciljem jednostavnijeg, temeljnijeg i bržeg procesa ispitivanja kodova nastali su **profajleri** [16].

U nastavku ovog rada biće reči o profajlerima generalno, kao i o njihovoj podeli na determinističke i statističke. Zatim se prelazi na profajliranje u jeziku Python, gde se govori o profajlerima profile, cProfile i hotshot. Konačno, uvešće se priča o alatima za vizualizaciju, gde će biti opisan i pokazan rad nekih od najpoznatijih alata: Py-Spy, SnakeViz, Pycallgraph, Gprof2dot i Vprof.

Svi profajleri i alati za vizualizaciju su testirani na kodu 3 datom u dodatku rada. Kod se sastoji od funkcija koje na različite načine računaju n-ti Fibonačijev broj [17].

## 2 Šta je profajliranje?

Profajliranje je proces koji značajno pomaže da se detektuju delovi koda koji mogu da učine program optimalnim. Optimizacijom koda možemo smanjiti vreme izvršavanja, memoriju koja se koristi ili detektovati mrežno opterećenje kojim se troši vreme na dohvaćanje informacija [1]. Postoje različiti načini kako možemo profajlirati kod. Za vremensko profajliranje postoji jednostavna komanda u konzoli *time* kojom možemo dobiti kratak izveštaj izvršavanja celog programa (videti sliku 1) ili modul *time* u Python-u kojim možemo meriti vreme unutar koda (videti primer 5). Kako su ovi načini ponekad nedovoljni ili naporni za implementaciju kod programa složenije strukture, programeri su osmislili alate koji služe za automatsko profajliranje koda i detektovanje kritičnih delova [18]. Ti alati se nazivaju profajleri.

Profajliranje se deli na **determinističko** (eng. tracing) i na **statističko** (eng. sampling). **Determinističko** profajliranje se izvršava zajedno sa kodom i profajleri beleže svako pokretanje i svako završavanje funkcije u programu. Loša strana ovakvog načina profajliranja je što je skupo, jer ugrožava performanse programa time što mora da prati stanje svih funkcija. To može dovesti do nerealne predstave o tome koliko je koja funkcija na primer vremenski zaista zahtevna, jer profajliranje takođe oduzima neko vreme [5]. **Statističko** profajliranje (uzoračko) se vrši uzorkovanjem vrednosti u IP (eng. Instruction Pointer) registru dok program radi. Uzorci se povezuju sa konkretnim funkcijama i potprogramima, a zatim se vrši statistička analiza da bi se odredilo koji se delovi koda najduže izvršavaju [8]. Ni ovakvim profajliranjem se ne dobijaju tačni rezultati, jer se program ne posmatra u svakom trenutku izvršavanja. Stanje se posmatra samo u tim trenucima koji se uzorkuju, što je ujedno i najveća mana statističkog profajliranja. Prednost u odnosu na determinističko profajliranje je manja memorijska zahtevnost da bi se podaci skladištili, kao i manje vremena potrebnog da se profajliranje izvrši iz razloga što se program posmatra periodično [9].

Pored podela na determinističko i statističko, profajliranje se može razlikovati i prema tome da li meri vremensko ili memorijsko opterećenje [13]. U daljem tekstu biće dati primeri takvih profajliranja kroz programski jezik Python.

### 3 Načini profajliranja u jeziku Python

Osnovni vid vremenskog profajliranja u jeziku Python se može prikazati korišćenjem modula *time*. Pokretanjem koda 5 koji se nalazi u sekciji A, kao izlaz iz programa dobija se uvek neka vrednost veća od postavljene vrednosti za promenljivu *n*. Na primer, izlaz može biti nešto poput:

*Ukupno je proteklo 5.002248048782349 sekundi.*

Razlog zašto je izmereno vreme veće od očekivanog je to što je ovo vreme celokupnog izvršavanja programa od njegovog pokretanja do kraja. To vreme nije isto kao vreme koje naš program provodi koristeći procesorsku jedinicu, jer u tom trenutku pokretanja našeg programa ima još aktivnih procesa koji se izvršavaju na procesoru. Dakle, postoje tri različite vrste vremena koja se mere: stvarno (eng. real or total), korisničko (eng. user) i sistemsko (eng. system). Korisničko meri kumulativno vreme provedeno na procesoru prilikom izračunavanja, dok sistemsko meri vreme potrošeno na memorijske alokacije [13]. Dodavanjem komande *time* u konzoli UNIX sistema, pre pokretanja neke druge komande, može se dobiti kratak izveštaj za ova tri vremena (za svaki operativni sistem je drugačije, a primer na macOS Catalina, u zsh terminalu nad kodom 5 prikazan je na slici 1). Pokretanjem koda 1 na četiri različita računara svakog od autora (konfiguracije računara videti u tabeli A), dobijena su vremena i predstavljena kroz barplot na slici 2.

```
→ msnr time python timing.py
Ja sada spavam 5 sekundi.
Ukupno je proteklo 5.004364013671875 sekundi.
python timing.py 0.03s user 0.01s system 0% cpu 5.051 total
→ msnr █
```

Slika 1: UNIX komanda *time*.

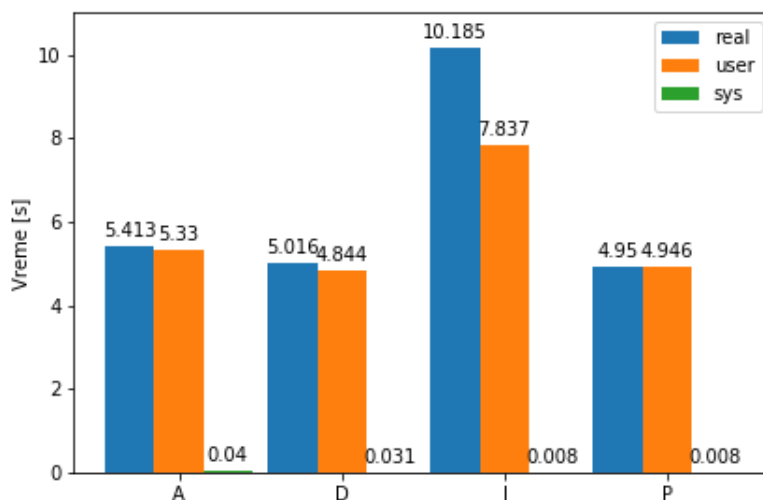
Drugi način da se celine programa u jeziku Python vremenski prorate je korišćenjem modula *timeit*. Modul meri vreme potrebno da se neki deo koda izvrši tako što ga pokreće u petlji *n* puta. Ovakav proces može da se ponavlja *r* puta, nakon čega uzima najbolju dobijenu vrednost kao konačnu. Zbog načina na koji radi, korisno ga je koristiti samo za **precizno** merenje malih delova koda [13], zbog čega se neće detaljnije razmatrati.

Preporuka je da se profajliranje, kao i sve optimizacije koda, rade tek na samom kraju projekta, jer preuranjeno optimizovanje može biti koren svih nevolja [12]. Nakon što se izmeri vreme izvršavanja celog programa, mogu se identifikovati celine koje bi mogle potencijalno da se unaprede i ubrzaju. U jeziku Python za to postoje 3 ugrađena modula iz standardne biblioteke:

- profile
- hotshot
- cProfile

Ovi ugrađeni moduli obezbeđuju determinističko profajliranje. Statističko ugrađeno profajliranje ne postoji, ali postoje biblioteke za rad sa njim.

Modul *profile* je pisan u čistom Python-u i njegovo izvršavanje može da bude skupo za performanse. Modul *hotshot* je znatno brži, pisan je u



Slika 2: Vremena potrebna za izvršavanje istog koda na 4 računara.

jeziku C, ali se ne koristi toliko često i nije podržan u Python 3. Najkorišćeniji modul za profajliranje je *cProfile*, pisan je u C jeziku i njegove funkcionalnosti su slične prvom modulu, ali je znatno brži [7]. U narednoj celini detaljnije će biti obrađen ovaj modul, jer je najkorišćeniji i najviše alata se oslanja na njega.

### 3.1 Moduli *cProfile* i *pstats*

Da bi se dobio izveštaj profajliranja kroz *cProfile* modul, potrebno je pokrenuti komandu [7]:

```
$ python -m cProfile test_kod.py
```

Test kod predstavlja funkcije definisane u primeru 3, sa izmenom:

```
number = 20
2 print('Input number is: ' + str(number))
for i in range(0, 1000):
4     for_fib(number)
     recur_fib(number)
6     tail_recur_fib(number)
     functional_fib(number)
```

Kod 1: Dodatak za testiranje *cProfile*.

Ono što se može videti iz izveštaja 3 je koliko funkcijskih poziva je ispratio *cProfile* i koliko njih su primitivni pozivi. U ovom slučaju samo 4004 poziva su bila primitivna, dok su ostali pozvani rekurzijom. Sve je sortirano prema poslednjoj koloni (što može da se modifikuje), a u tabeli 1 su objašnjeni preostali argumenti [7]. Izveštaj koji napravi *cProfile* može da se napravi unutar programa i sačuva u obliku fajla, a zatim obradi i formatira putem klase *Stats* modula *pstats*. Uključivanje zaglavlja, startovanje funkcije i čuvanje izveštaja se postiže komandama:

```

→ msnr python -m cProfile test_kod.py
Input number is: 20
21933004 function calls (4004 primitive calls) in 5.100 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.001    0.001    5.100    5.100    test_kod.py:1(<module>)
1000   0.002    0.000    0.002    0.000    test_kod.py:1(for_fib)
21891000/1000  5.085    0.000    5.085    0.005    test_kod.py:10(recur_fib)
20000/1000   0.006    0.000    0.006    0.000    test_kod.py:16(tail_recur_fib)
21000/1000   0.006    0.000    0.006    0.000    test_kod.py:23(<lambda>)
1      0.000    0.000    5.100    5.100    {built-in method builtins.exec}
1      0.000    0.000    0.000    0.000    {built-in method builtins.print}
1      0.000    0.000    0.000    0.000    {method 'disable' of '_lsprof.Profiler' objects}

```

Slika 3: Primer izveštaja iz cProfile modula.

```

import cProfile
# ... ovde se nalaze funkcije
cProfile.run('test_function()', './izvestaj')

```

Kod 2: Primer čuvanja izveštaja u fajl.

Modul *pstats* sadrži veliki broj mogućnosti za filtriranje dobijenih informacija iz izveštaja. Na primer, ukoliko je cilj da se razume koji algoritam oduzima najviše vremena, sortiranje izveštaja treba da se vrši po koloni *cumulative*, nakon čega se komandom *print\_stats(5)* može izdvojiti 5 vremenski najzahtevnijih poziva. Takođe, moguće je kombinovati module *cProfile* i *pstats* i jedan takav primer je prikazan u kodu 4. Klasa *Stats* može da napravi instance od izveštaja koji su u fajlovima ili direktno od *Profile* klase. Nad njima se mogu pozivati metode poput: *strip\_dirs*, *add*, *dump\_stats*, *sort\_stats*, *print\_stats*, o kojima se više može videti na [7].

Tabela 1: Argumenti iz cProfile izveštaja i njihovi opisi.

ARGUMENT	OPIS
ncalls	broj poziva
tottime	ukupno vreme funkcije
percall	tottime/ncalls
cumtime	ukupno vreme funkcije i svih podfunkcija
percall	cumtime/primitive calls
filename:lineno(function)	podaci o funkciji

## 4 Alati za vizualizaciju profajliranja

Statistika koja se dobije profajliranjem postaje nečitljivija što je program kompleksniji. Da bi dobijeni rezultat bio što pregledniji, razvijeni su razni alati za vizualizaciju profajliranja. U ovom poglavlju dati su neki alati u jeziku Python (više alata može se naći na [14]). Kao i do sada, svi alati se testiraju na kodu 3 koji se nalazi u dodatku.

### 4.1 Py-Spy

**Py-Spy** je statistički profajler za jezik Python koji omogućava vizualizaciju informacija o izvršavanju programa tokom njegovog rada. Napisan

je u programskom jeziku Rust. Može da se koristi na operativnim sistemima Linux, OSX i Windows, a podržava verzije 2.3-2.7 i 3.3-3.7 Python-a [3]. Instalira se pomoću naredne komande:

```
$ pip install py-spy
```

Py-spy se pokreće preko komandne linije tako što mu se da PID (eng. process identifier) procesa:

```
$ py-spy --pid 12345
```

ili ime programa koji želimo da profajliramo:

```
$ py-spy -- python myprogram.py
```

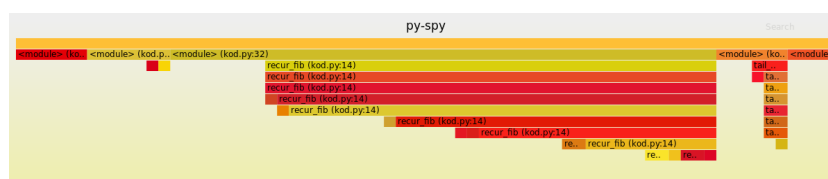
Detaljna instalacija i pokretanje mogu se naći na [3].

#### 4.1.1 Karakteristike alata

Py-Spy je uzorački profajler, što znači da program mora da bude u toku izvršavanja kada se alat pokrene. Dakle, on omogućava profajliranje programa koji moraju neprestano da rade. Većina profajlera zahteva da se kod modifikuje na neki način, dok Py-Spy ne samo da ne zahteva modifikacije koda, već se izvršava u zasebnom programu čime smanjuje vreme izvršavanja i ne ometa rad programa ni na jedan način [2]. Glavna mana Py-Spy profajlera je ujedno i glavna mana statističkog profajliranja: procena umesto tačnog rešenja usled uzorkovanja podataka [9].

#### 4.1.2 Primer primene

Pokretanjem alata Py-Spy dobija se slika programa sa trajanjem svih funkcija u njemu. Pokrenut je referentni program ali u beskonačnoj petlji, da bi Py-Spy mogao da uzorkuje u vreme izvršavanja programa. Rezultat pokretanja prikazan je na slici 4. Što funkcija traje duže, linija kojom je ona predstavljena je duža. Takođe, može se videti redosled poziva funkcija.



Slika 4: Py-Spy vizualizacija.

## 4.2 SnakeViz

**SnakeViz** je alat za vizualizaciju statistike profajliranja koji je generisan od strane *cProfile* modula i predstavlja alternativu *pstats* modulu. Inspirisan je RunSnakeRun alatom. Snakeviz podržava verzije interpretera Python 2.7 i Python 3 [10]. Fajl koji se generiše je u formatu *.profile*. Alat SnakeViz se instalira pomoću naredne komande:

```
$ pip install snakeviz --user
```

Kada se pokreće Python kod, potrebno je navesti ime fajla u formatu .profile:

```
$ python -m cProfile -o test.profile test.py
```

Zatim je potrebno izvršiti sledeću komandu da bi se dobila vizualizacija u obliku .html fajla:

```
$ snakeviz test.profile
```

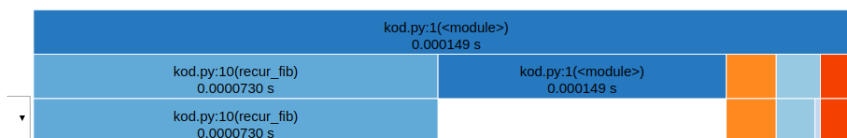
Detaljnije informacije o instalaciji i pokretanju alata mogu se naći na [10].

#### 4.2.1 Karakteristike alata

SnakeViz nije u mogućnosti da podrži velike profile zbog poteškoća reprezentovanja ogromnih stabala pomoću JSON niski. Za sada se smatra da SnakeViz može da podrži stabla sa manje od nekoliko hiljada čvorova. Međutim, iako ne uspe da napravi vizualizaciju i dalje će se dobiti potpuna tabela statistika. To znači da neuspeh vizualizacije SnakeViz alata ne utiče na samo izvršavanje profajliranja [15]. Alat SnakeViz omogućava izbor kriterijuma po kom se podaci u tabeli sortiraju.

#### 4.2.2 Primer primene

Pokretanjem SnakeViz alata otvara se .html fajl u veb pretraživaču. Vizualizacija se može predstaviti u *Icicle* i *Sunburst* formatu. Na slici 5 je dat primer koda nad kojim se testiraju vizualizacije. Pored toga,



Slika 5: SnakeViz vizualizacija.

SnakeViz za svaku funkciju može da prikaže vreme izvršavanja funkcije (u milisekundama i procentima), kao i liniju na kojoj se funkcija nalazi u kodu. Primer ovih statistika za funkciju koja računa rekurentno n-ti Fibonačijev broj dat je na slici 6.

```
Name: _____  
recur_fib  
Cumulative Time:  
0.0000730 s (48.99 %)  
File:  
kod.py  
Line:  
10  
Directory:
```

Slika 6: SnakeViz statistike.

## 4.3 Pycallgraph

**Pycallgraph** je Python modul pomoću kog se na osnovu koda može dobiti graf poziva. Postoji podrška za Python 2.7+ i Python 3.3+ [11]. Moguće ga je pokrenuti iz komandne linije ili uključiti u kod. Pycallgraph se instalira izvršavanjem sledeće komande:

```
$ pip install pycallgraph
```

Osim toga, neophodno je imati instaliran *Graphviz* ili *Gephi* kako bi vizualizacija bila moguća:

```
$ sudo apt-get install graphviz
```

Graf poziva se dobija u direktorijumu iz kog se pokreće naredna komanda:

```
$ pycallgraph graphviz -- ./test.py
```

Format u kom se čuva izlaz se može jednostavno zadati korišćenjem opcije `-f`, a ukoliko se format ne zada podrazumevani će biti `.png` (`pycallgraph.png`). Detaljnije informacije o instalaciji i pokretanju alata mogu se naći na [11].

### 4.3.1 Karakteristike alata

Pri korišćenju ovog alata moguće je čvorove grafa poziva obojiti različitim bojama, u zavisnosti od toga koliko puta je funkcija pozvana, ili koliko je vremena ili memorije potrebno za njeno izvršavanje. Pored toga, različiti moduli programa mogu biti vizuelno grupisani kako bi graf bio lakši za razumevanje. Poslednja verzija ovog alata izašla je 2013. godine, ali i dalje se bez problema može koristiti [11].

### 4.3.2 Primer primene

Pokretanjem naredne komande nad test kodom 3, sa opcijom *max-depth* pomoću koje se zadaje maksimalna dubina, dobijena je slika 7.

```
$ pycallgraph --max-depth=4 graphviz -- ./test.py
```

Na slici 7 bojenje čvorova izvršeno je na osnovu vremena potrebnog za izvršavanje funkcija i broja poziva. To znači da su tamnijim bojama predstavljene funkcije koje su vremenski zahtevnije.

## 4.4 Gprof2dot

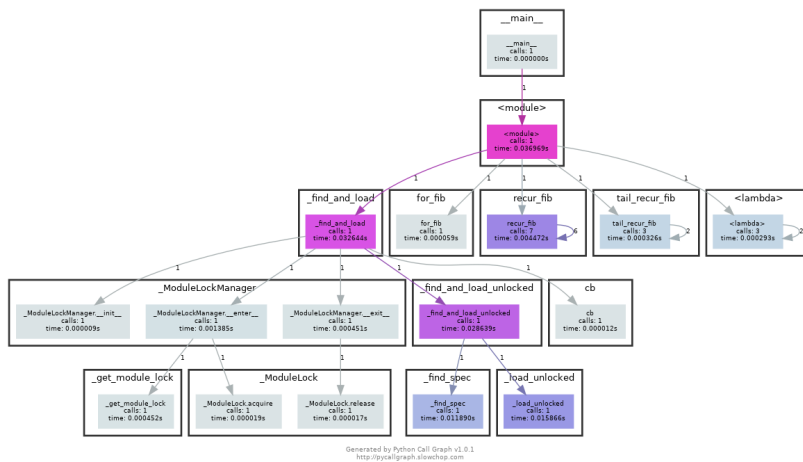
**Gprof2dot** je alat za vizualizaciju izlaza različitih profajlera u obliku grafa sa čvorovima. Napisan je u programskom jeziku Python. Rad je omogućen na bilo kojoj platformi na kojoj su instalirani Python i Graphviz. Komanda za instalaciju `gprof2dot` alata je:

```
$ pip install gprof2dot
```

Da bi se `gprof2dot` koristio, mora se generisati izlaz koji on može da čita. To se postiže narednom komandom:

```
$ python3 -m cProfile -o output.pstats path/to/your/script arg1 arg2
```





Slika 7: Pycallgraph vizualizacija za graf dubine 4.

Nakon toga potrebno je pokrenuti komandu za izvršavanje gprof2dot alata i čuvanje rezultata u obliku slike output.png:

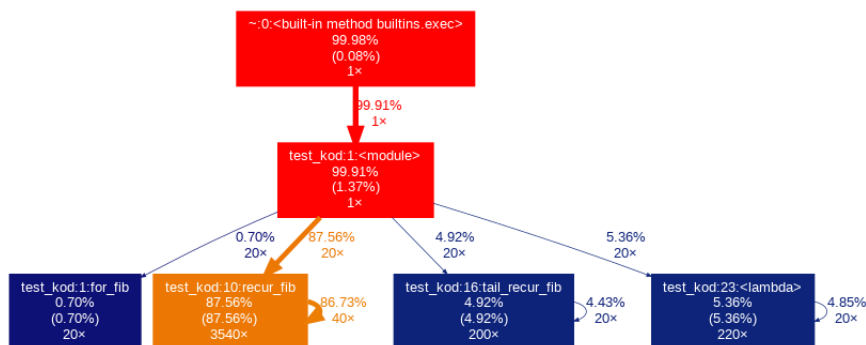
```
$ gprof2dot.py -f pstats output.pstats | dot -Tpng -o output.png
```

#### 4.4.1 Karakteristike alata

Korišćenjem dodatne opcije pri zadavanju komande za izvršavanje gprof2dot alata moguće je dobiti potkresano drvo, do čega se dolazi eliminacijom svih čvorova ispod zadatog limita. Jedna od prednosti ovog alata je to što korišćenjem boja uspeva da skrene pažnju na kritične delove koda. Neki od profajlera čiji izlaz gprof2dot može da čita su: Linux Perf, Valgrind's Callgrind Tool, Python Profilers, gprof, itd. [6]. Pomoću njega funkcijski pozivi se mogu predstaviti u vidu grafa, što je korisno kada su programi veliki. Nekada je teško uočiti grešku gledajući sirove podatke, dok graf može značajno olakšati uočavanje istih.

#### 4.4.2 Primer primene

Komandom iznad (4.4) poziva se gprof2dot, čime se generiše slika grafa i čuva u .png formatu. Primer je pokrenut za 30. Fibonačijev broj.



Slika 8: gprof2dot vizualizacija.

Sadržaj jednog čvora:

ime funkcije
ukupno vreme% ( vreme%)
broj poziva

gde je:

- ime funkcije - funkcija na koju se odnosi sadržaj čvora
- ukupno vreme% - odnosi se na procenat vremena provedenog u funkciji, u odnosu na ukupno vreme rada programa
- vreme% - odnosi se na procenat vremena provedenog u samo ovoj funkciji (bez poziva drugih funkcija)
- broj poziva - koliko puta je funkcija pozvana (uključujući i rekurziju)

Jedna grana grafa predstavlja poziv između funkcija. Ona povezuje roditeljsku i dete funkciju, gde je roditeljska funkcija ona koja poziva, dok je dete funkcija ona koja je pozvana iz roditeljske funkcije. Grana nosi informacije o tome koliko puta je roditelj pozvao dete funkciju, kao i procenat ukupnog vremena koje je potrebno da se izvršavanje prebaci sa jedne funkcije na drugu.

Sa slike 8 može se videti da je najmanje vremena utrošeno na funkciju `for_fib`, koja je pozvana najmanji broj puta (zato je plave boje). Funkcija `recur_fib` oduzela je najviše vremena i pozvana je najviše puta (zato je narandžaste boje). Dva roditeljska čvora su crvene boje jer oni predstavljaju glavnu nit, koja traje koliko i ceo proces (zato i ima najveći procenat utrošenog vremena i samo jedan poziv).

## 4.5 Vprof

**Vprof** radi tako što pokreće program, izvršava merenje i potom startuje lokalni server gde prikazuje rezultate u podrazumevanom veb pretraživaču.

Komanda za instalaciju alata je:

```
$ pip install vprof
```

Pokretanje vprof alata izgleda ovako:

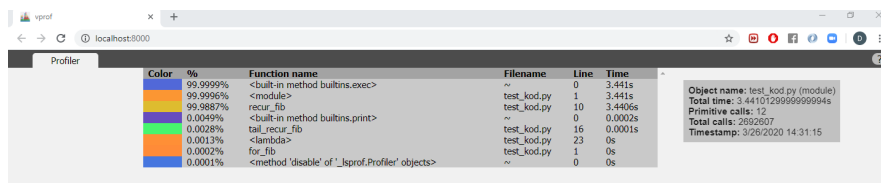
```
$ vprof -c <config> <src>
```

Argument <config> podrazumeva sledeće opcije:

- p - profajler
- m - graf memorije
- h - toplotna mapa koda
- c - CPU flejm graf

Prednost u odnosu na ostale alate ogleda se u tome što se rezultati prikazuju u interaktivnoj veb aplikaciji. Loša strana je to što se nalazi u konstantnom razvoju, pa može doći do bagova. Pokretanje vprof alata, prilikom kog se dobija vizualizacija izvršavanja koda u veb pretraživaču, realizuje se komandom:

```
$ vprof -c p test_kod.py
```



Slika 9: vprof vizualizacija.

Detaljnije informacije o instalaciji i pokretanju alata mogu se naći na [19].

## 5 Zaključak

Kroz ovaj rad predstavljene su osnovne ideje procesa profajliranja. Čitalac je upoznat sa modulima za profajliranje u jeziku Python, kao i sa alatima za vizualizovanje statističkih izveštaja koje profajleri generišu. Korišćenjem alata **Py-Spy** predstavljeno je kako rade statistički profajleri i zašto su oni bitni, dok su kroz alate **SnakeViz**, **Pycallgraph**, **Gprof2dot** i **Vprof** predstavljene neke od mogućnosti determinističkih profajlera. Za sve alate prikazan je i ukratko objašnjen izlaz koji oni generišu kada se pokrenu nad test kodom 3.

Prednost korišćenja profajlera je mogućnost dobijanja bolje slike o tome odakle je koja funkcija pozvana, koliko je njeno izvršavanje trajalo, da li je ona pozivala još neke funkcije, itd. Praćenjem toka izvršavanja programa mogu se uočiti razni propusti u kodu ili delovi koje je moguće optimizovati zbog toga što oduzimaju puno vremena. Dakle, kada razvoj softvera dođe u fazu da je potrebno debugovanje ili njegova optimizacija, profajliranje je nešto što značajno može ubrzati proces detekcije kritičnih delova.

Čitalac se upućuje da nakon upoznavanja sa ovde predstavljenim alatima i njihovim osnovnim funkcionalnostima, samostalno istraži alate koji su mu se posebno dopali, korišćenjem referenci datih u poglavlju alata. Osim toga, dva dodatna modula za Python koja mogu biti interesantna za dalje istraživanje su: **line\_profiler** i **memory\_profiler**. Više o njima

se može pročitati u poglavlju *IPython: Beyond Normal Python - Profiling and Timing Code* knjige [18]. Prvi modul omogućava linijsko profajliranje, dok drugi modul prati memorijsko opterećenje funkcija iz koda, za koje nije bilo prostora u ovom radu.

## Literatura

- [1] J. Athalye, A. Gjengset and J. J. G. Ortiz. The Missing Semester of Your CS Education, 2020. on-line at: <https://missing.csail.mit.edu/2020/debugging-profiling/>.
- [2] Ben Frederickson (benfred). Py-Spy: A sampling profiler for Python programs, 2018. on-line at: <https://docs.rs/crate/py-spy/0.1.4>.
- [3] Ben Frederickson (benfred). PySpy vizualization tool, 2018. on-line at: <https://github.com/benfred/py-spy>.
- [4] Jon Louis Bentley. *Writing Efficient Programs (Prentice-Hall Software Series)*. Prentice Hall Ptr, 1982.
- [5] Julia Evans. How do Ruby & Python profilers work?, 2020. on-line at: <https://jvns.ca/blog/2017/12/17/how-do-ruby---python-profilers-work-/>.
- [6] José Fonseca. gprof2dot, 2019. on-line at: <https://github.com/jrfonseca/gprof2dot>.
- [7] Python Software Foundation. The Python Profilers, 1990-2020. on-line at: <https://docs.python.org/2/library/profile.html>.
- [8] Texas Instruments. Statistical Profiling, 2018. on-line at: [https://processors.wiki.ti.com/index.php/Statistical\\_Profiling](https://processors.wiki.ti.com/index.php/Statistical_Profiling).
- [9] JetBrains. Profiling Types, 2018. on-line at: [https://www.jetbrains.com/help/profiler/Profiling\\_Guidelines\\_Choosing\\_the\\_Right\\_Profiling\\_Mode.html#](https://www.jetbrains.com/help/profiler/Profiling_Guidelines_Choosing_the_Right_Profiling_Mode.html#).
- [10] Matt Davis (jiffyclub). SnakeViz vizualization tool, 2012. on-line at: <https://jiffyclub.github.io/snakeviz/>.
- [11] Gerald Kaszuba. Python Call Graph, 2013. on-line at: <https://pycallgraph.readthedocs.io/en/master/>.
- [12] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, December 1974.
- [13] Gabriele Lanaro. *Python high performance programming*. Packt Publishing Ltd, 2013.
- [14] Antonio Molner. Python Profiling, 2019. on-line at: <https://medium.com/@antoniomdk1/hpc-with-python-part-1-profiling-1dda4d172cdf?fbclid=IwAR3tcsUjhQ4femF9h5JK4GaZEM5ja02rupa0F88-V2csNjY7hHrL2eNAsdA>.
- [15] Pen and Pants. SnakeViz 0.1 – A Python Profile Viewer, 2012. on-line at: <https://penandpants.com/2012/09/21/snakeviz-0-1-a-python-profile-viewer/>.
- [16] Steve Scargall. *Profiling and Performance*, pages 295–312. 01 2020.
- [17] Tony Scott and Pan Marketos. On the origin of the fibonacci sequence. pages 1–46, 03 2014.
- [18] Jake VanderPlas. *Python data science handbook: Essential tools for working with data*. "O'Reilly Media, Inc.", 2016.
- [19] Nick Volynets. vprof, 2020. on-line at: <https://github.com/nvdrv/vprof>.

## A Dodatak

```
# This program is used for testing purposes
2 def for_fib(n):
    a = 0
4     b = 1
    for i in range(0, n):
6         temp = a
            a = b
8             b = temp + b
    return a

10 def recur_fib(n):
12     if n <= 1:
        return n
14     else:
        return(recur_fib(n-1) + recur_fib(n-2))

16 def tail_recur_fib(n, a = 0, b = 1):
18     if n == 0:
        return a
20     if n == 1:
        return b
22     return tail_recur_fib(n - 1, b, a + b)

24 functional_fib = (lambda x, a=1, b=0:
26     b if x == 0
        else functional_fib(x - 1, a + b, a))

28 number = 10

30 print('Input number is: ' + str(number))
    print('For loop Fibonacci ' + str(for_fib(number)))
32 print('Recursion Fibonacci: ' + str(recur_fib(number)))
    print('Tail recursion Fibonacci: ' + str(tail_recur_fib(number)))
34 print('Functional Fibonacci: ' + str(functional_fib(number)))
```

Kod 3: Kod korišćen za testiranje modula i alata.

```
import cProfile
2 import pstats
import io

4 number = 20
    print('Input number is: ' + str(number))

8 def test_function():
    for i in range(0, 1000):
10         for_fib(number)
            recur_fib(number)
12             tail_recur_fib(number)
                functional_fib(number)

14 if __name__ == '__main__':
16     prof = cProfile.Profile()
        prof.enable()
18     # cProfile.run('test_function()', './izvestaj')
        test_function()
20     prof.disable()

22     s = io.StringIO()
        sortby = 'tottime'
24     ps = pstats.Stats(prof, stream=s).sort_stats(sortby)
        ps.print_stats()
26     print(s.getvalue())
```

Kod 4: Kod korišćen za testiranje modula cProfile i pstats.

```

import time
2 # ocitamo vreme na pocetku
tic = time.time()
4 n = 5
print("Ja sada spavam {} sekundi.".format(n))
6 time.sleep(n)
# ocitamo vreme na kraju
8 toc = time.time()
print("Ukupno je proteklo {} sekundi.".format(toc-tic))

```

Kod 5: Primer profajliranja korišćenjem modula *time*.

	Procesor	RAM	OS
A - Anđelka	Intel Core i5-7360U @ 2.3GHz x 2	16GB	macOS Catalina
D - David	Intel Core i5-3210M @ 2.5Ghz x 2	8GB	Windows 10
J - Jelisaveta	Intel Core i5-520M @ 2.4GHz x 4	6GB	Ubuntu 19.04
P - Petar	Intel Core i3-6100U @ 2.3GHz x 4	4GB	Ubuntu 18.04

Tabela 2: Karakteristike računara.