UNIVERZITET U BEOGRADU
Elektrotehnički fakultet
Katedra za računarsku tehniku i informatiku

Testiranje softvera

Vežbe - Strategije bele kutije

Profesor:
dr Dragan Bojić

Asistent:
dipl. ing. Dražen Drašković

Beograd, 2012.

Verzija dokumenta: 2.0

Verzija	Promena
2.0	21.11.2012.

Teorijske osnove

Strategije bele kutije obuhvata testiranje kontrole toka i testiranje putanja.

Grafovi kontrole toka predstavljaju vizuelnu reprezentaciju strukture koda nekog programa. Algoritam za testiranje pomoću grafova kontrole toka prvo pretvara programski kod u graf, a zatim analizira moguće puteve kroz graf. Postoji niz tehnika koje možemo da primenimo u zavisnosti koliko temeljno želimo da testiramo kod. Zatim ćemo na izabranom nivou testiranja stvoriti test primere.

Na najnižem nivou testiranja kontrolom toka imamo pokrivenost iskaza (eng. *Statement Coverage*). Sinonimi za ovo testiranje je pokrivenost instrukcija ili pokrivenost koda, a znači isto: Da li se u nekom trenutku izvršava svaka pojedinačna linija koda aplikacije. Statistika kaže da testiranje strategijama crne kutije bez korišćenja strategija bele kutije pokriva u proseku oko 30% iskaza.

Sledeći korak testiranja kontrolom toka naziva se pokrivenost odluka ili pokrivenost grana (eng. *Decision* or *Branch Coverage*). Svaku odluku u kodu treba da pokrijemo bar jednom.

Treću grupu čine testiranje zasnovano na pokrivenosti uslova (eng. *Condition Coverage*), kada svi uslovi moraju biti zadovoljeni bar jednom i testiranje zasnovano na pokrivenosti višestrukih uslova (eng. *Multiple Condition Coverage*), koji daje sve moguće kombinacije prostih uslova. Kombinacijom prethodnih strategija dobijaju se pokrivenost odluka i uslova (*eng. Decision/Condition Coverage*) i modifikovana pokrivenost odluka i uslova (MC/DC).

POKRIVANJE ISKAZA (eng. Statement Coverage)

Teorijske osnove

Pokrivanje iskaza: svaki iskaz programa mora se bar jednom izvršiti. Da bismo postigli pokrivenost iskaza, moramo izabrati test primere koji izvršavaju svaku liniju koda u programu.

Da bismo izračunali trenutni nivo pokrivenosti iskaza koji smo postigli, delimo broj iskaza u kodu koji smo izvršili, sa brojem ukupnih iskaza u celom kodu. Ako je količnik ta dva broja 1, imamo 100% pokrivenost iskaza.

Pokrivanje iskaza je najmanje efikasna tehnika kontrole toka, jer da bi se postigao ovaj skromni nivo pokrivenosti, tester mora da dovoljnim brojem test primera primora svaku liniju koda da se izvrši bar jednom.

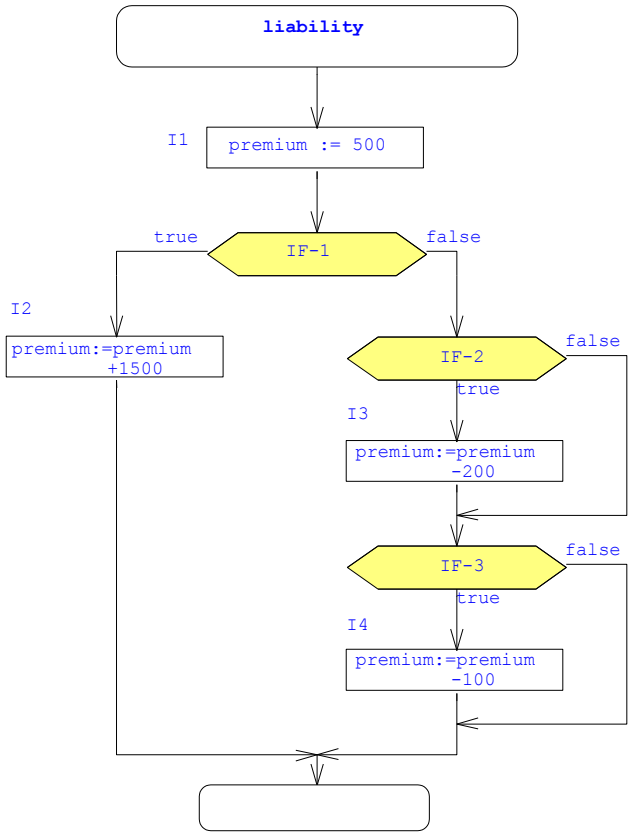
Zadatak 1.

Za datu proceduru na Pascal-u odrediti skupove testova koji pokrivaju iskaze.

```
type mf = (male, female);

procedure liability(age:integer; sex:mf; married:boolean; var
premium:integer);
begin
  premium := 500;
  if ((age < 25) and (sex = male) and (not married)) then
    premium := premium + 1500;
  else
  begin
    if (married or (sex = female)) then
      premium := premium - 200;
    if ((age > 45) and (age < 65)) then
      premium := premium - 100;
  end
end.
```

Rešenje:

Najpre se za proceduru iz postavke zadatka, crta dijagram toka:



Na osnovu prikazanog dijagrama toka kreira se odgovarajuća tabela:

Pokriveni iskazi	Godine	Pol	U braku	Test primer
I1, IF-1, I2	<25	male	False	(1) 15 m F
I1, IF-1, IF-2, I3, IF-3, I4	>45, <65	female	True	(2) 50 f T

POKRIVANJE ODLUKA (eng. Decision/Branch Coverage)

Teorijske osnove

Umesto gledanja iskaza, ovo pokrivanje gleda samo odluke. Pokrivanje odluka znači da svaka izlazna grana uslovnih iskaza mora biti bar jednom izabrana. Svaka odluka ima mogućnost da se izvrši ili kao TRUE ili kao FALSE. Nema drugih mogućnosti. Kako uvek prolazimo i kroz TRUE i kroz FALSE grane, pokrivanje odluka garantuje pokrivanje iskaza, ali obrnuto ne važi.

Postoji niz različitih odluka, koje programski jezici formiraju:

Odluke	Odluke kod petlji

`if (uslov) {`
`  ...`
`} else {`
`  ...`
`}`
	`while (uslov) {`
	`  ...`
	`}`

`switch (uslov) {`
`  case const1: {...} break;`
`  case const2: {...} break;`
`  case const3: {...} break;`
`  default: {...}`
`}`
	`do {`
	`  ...`
	`} while (uslov)`
	`for (inic; uslov; iterator) {`
	`  ...`
	`}`

Za one koji misle da Switch uslovni iskaz daje više od dve odluke, konceptualno jeste tako. Switch iskaz je kompleksan skup odluka, koji je često izgrađen od strane kompajlera kao tabela. Međutim, generisani kod zaista predstavlja samo binarna poređenja, koja se nastavljaju redom sve dok ne pronađe svoju granu („case“) ili dok se ne izvrši „default“ grana. Odluke koje se javljaju kod Switch iskaza vraćaju True ili False.

Petlje će biti kasnije analizirane.

Zadatak 1.

Za datu proceduru na Pascal-u odrediti skupove testova koji pokrivaju odluke.

```
type mf = (male, female);

procedure liability(age:integer; sex:mf; married:boolean; var
premium:integer);
begin
  premium := 500;
  if ((age < 25) and (sex = male) and (not married)) then
    premium := premium + 1500;
  else
  begin
    if (married or (sex = female)) then
      premium := premium - 200;
    if ((age > 45) and (age < 65)) then
      premium := premium - 100;
  end
end.
```

Rešenje:

Rešavanje zadatka ponovo možemo započeti kreiranjem dijagrama toka (pogledati sekciju: Pokrivanje iskaza).

Na osnovu kreiranog grafa, možemo kreirati narednu tabelu:

Pokrivene odluke	Godine	Pol	U braku	Test primer
IF-1	< 25	male	False	(1) 23 m F
IF-1	< 25	female	False	(2) 23 f F
IF-2	*	female	*	(2)
IF-2	>= 25	male	False	(3) 50 m F
IF-3	<= 45	female #	*	(2)
IF-3	>45, <65	*	*	(3)

Napomena: Ovaj podatak nije bitan za IF-3, ali je bitan za IF-1, da bismo bili sigurni da ćemo u IF-1 dobiti FALSE (i tako ući u else granu), na primer u slučaju da je [god < 25 AND brak=false] tada mora biti pol=female.

Napomena: za potpuno definisanje test primera moraju se navesti i očekivani izlazni rezultati (vrednost promenljive premium) radi kasnijeg uporedjivanja sa stvarnim izlaznim rezultatima.

POKRIVANJE USLOVA (eng. Condition Coverage)

Teorijske osnove

Pokrivanje uslova: Svi uslovi (i negacije uslova) u programu moraju biti zadovoljeni barem jednom. Kod složenih uslova posmatraju se samo elementarni uslovi nezavisno jedan od drugoga.

Pokrivanje uslova ne garantuje pokrivanje odluka (pogledati Zadatak 1 iz tekuće sekcije). Ali jedna zanimljivost je da će pokrivenost odluka i pokrivenost uslova biti iste, kada se sve odluke sastoje od jednostavnih atomskih izraza, na primer kod izraza: if (a==1){} pokrivanje uslova biće identično pokrivanju odluka.

Zadatak 1.

Za datu proceduru na Pascal-u odrediti skupove testova koji pokrivaju uslove.

```
type mf = (male, female);

procedure liability(age:integer; sex:mf; married:boolean; var
premium:integer);
begin
  premium := 500;
  if ((age < 25) and (sex = male) and (not married)) then
    premium := premium + 1500;
  else
  begin
    if (married or (sex = female)) then
      premium := premium - 200;
    if ((age > 45) and (age < 65)) then
      premium := premium - 100;
  end
end.
```

Rešenje:

Pokriveni uslovi	Godine	Pol	U braku	Test primer
IF-1	< 25	female	False	(1) 23 f F
IF-1	>= 25	male	True	(2) 30 m T
IF-2	*	male	True	(2)
IF-2	*	female	False	(1)
IF-3	<= 45	*	*	(1)
IF-3	> 45	*	*	(3) 70 f F
IF-3	< 65	*	*	(2)
IF-3	>= 65	*	*	(3)

Napomena: Ovi test primeri ne izvršavaju then klauzule IF-1 i IF-3, kao i (praznu) else klauzulu IF-2.

Zadatak 2.

Neka je dat deo koda u programskom jeziku Java:

```
...
1. z=0;
2. if (a>b)
3.     z=12;
4. rez=72/z;
...
```

Napisati test primere koji vrše:

1) Pokrivanje iskaza
2) Pokrivanje odluka
3) Pokrivanje uslova

Da li izvršavanjem neke tehnike može doći do potencijalnog problema?

Rešenje:

TP1: a=3, b=2 => rez=6
TP2: a=2, b=3 => rez=?

POKRIVANJE ODLUKA I USLOVA (eng. Decision/Condition Coverage)

Teorijske osnove

Pokrivanje uslova i odluka: svaka izlazna grana uslovnih iskaza mora biti bar jednom izabrana. Svaki uslov (i njegova negacija) u programu moraju biti zadovoljeni bar jednom.

Zadatak 1.

Za datu proceduru na Pascal-u odrediti skupove testova koji pokrivaju uslove.

```
type mf = (male, female);

procedure liability(age:integer; sex:mf; married:boolean; var
premium:integer);
begin
  premium := 500;
  if ((age < 25) and (sex = male) and (not married)) then
    premium := premium + 1500;
  else
  begin
    if (married or (sex = female)) then
      premium := premium - 200;
    if ((age > 45) and (age < 65)) then
      premium := premium - 100;
  end
end.
```

Rešenje:

Pokrivene odluke i uslovi	Godine	Pol	U braku	Test primer
IF-1(odluka)	< 25	male	False	(1) 23 m F
IF-1 (odluka)	< 25	female	False	(2) 23 f F
IF-1 (uslov)	< 25	female	False	(2)
IF-1 (uslov)	>= 25	male	True	(3) 70 m T
IF-2 (odluka)	*	female	*	(2)
IF-2 (odluka)	>= 25	male	False	(4) 50 m F
IF-2 (uslov)	*	male	True	(3)
IF-2 (uslov)	*	female	False	(2)
IF-3 (odluka)	<= 45	*	*	(2)
IF-3 (odluka)	>45, <65	*	*	(4)
IF-3 (uslov)	<= 45	*	*	(2)
IF-3 (uslov)	> 45	*	*	(4)
IF-3 (uslov)	< 65	*	*	(4)
IF-3 (uslov)	>= 65	*	*	(3)

Napomena: Ova tabela je nastala prostim spajanjem svih odluka (iz tabele pokrivenosti odluka) i svih uslova (iz tabele pokrivenosti uslova), a zatim je minimizovan broj test primera.

POKRIVANJE VIŠESTRUKIH USLOVA (eng. Multiple Condition Coverage)

Teorijske osnove

Pokrivanje višestrukih (razvijenih, složenih) uslova: sve moguće kombinacije prostih uslova i njihovih negacija u svakom od složenih uslova moraju biti zadovoljeni bar jednom.

Zadatak 1.

Za datu proceduru na Pascal-u odrediti skupove testova koji pokrivaju uslove.

```
type mf = (male, female);

procedure liability(age: integer; sex: mf; married: boolean; var
premium: integer);
begin
  premium :=500;
  if ((age < 25) and (sex = male) and (not married)) then
    premium := premium + 1500
  else
  begin
    if (married or (sex = female)) then
      premium := premium - 200;
    if ((age > 45) and (age < 65)) then
      premium := premium - 100;
  end
end.
```

Rešenje:
Svako grananje treba posmatrati posebno:

Višestruki uslovi	Godine	Pol	U braku	Test primer
IF-1	< 25	male	True	(1) 23 m T
IF-1	< 25	male	False	(2) 23 m F
IF-1	< 25	female	True	(3) 23 f T
IF-1	< 25	female	False	(4) 23 f F
IF-1	>= 25	male	True	(5) 30 m T
IF-1	>= 25	male	False	(6) 70 m F
IF-1	>= 25	female	True	(7) 50 f T
IF-1	>= 25	female	False	(8) 30 f F
IF-2	*	male	True	(5)
IF-2	*	male	False	(6)
IF-2	*	female	True	(7)
IF-2	*	female	False	(8)
IF-3	<=45, >=65	*	*	nemoguće
IF-3	<=45, <65	*	*	(8)
IF-3	>45, >=65	*	*	(6)
IF-3	>45, <65	*	*	(7)

Zadatak 2.

Odrediti sve moguće kombinacije višestrukih (razvijenih) usova i odgovarajući skup test primera za navedeni programski segment.

```
if !C then
  if A or (C and D) then
    X=1
  else X=2
else
  X=3;
```

Rešenje:

Razvijeni uslov	Test primer
(IF1) C==true	(1)
(IF1) C==false	(2)
(IF2) C==true A==true D==true	nije moguć
(IF2) C==true A==true D==false	nije moguć
(IF2) C==true A==false D==true	nije moguć
(IF2) C==true A==false D==false	nije moguć
(IF2) C==false A==true D==true	(2)
(IF2) C==false A==true D==false	(3)
(IF2) C==false A==false D==true	(4)
(IF2) C==false A==false D==false	(5)

TESTIRANJE PETLJI

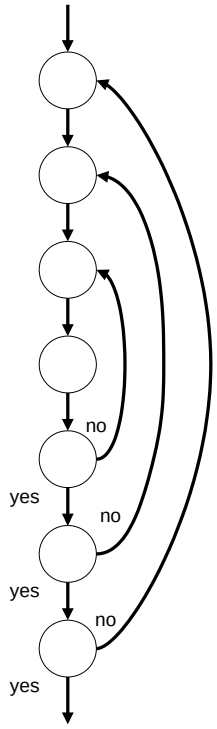
Teorijske osnove

Cilj testiranja petlji jeste testiranje while-do, repeat-until (ili do-while) i bilo koje druge petlje u programu - pokušavajući da svaku izvrši minimalan, tipičan i (ukoliko je definisan) maksimalan broj puta - takođe pokušavajući da "prekine" program, izvršavajući petlju manje puta od minimalnog, kao i veći broj puta od maksimalnog broja iteracija.

Tipovi petlji

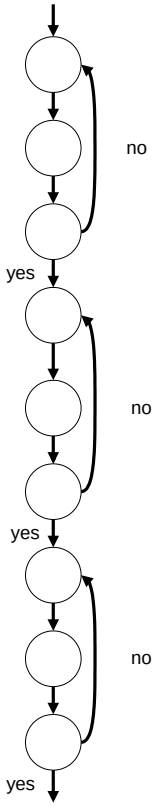
Tri tipa petlje, odnosno kombinacije petlji, biće testirane:

- Jednostavne petlje su petlje čije telo ne sadrži druge petlje



- Ugneždene petlje su kombinacija petlji tako da je svaka sadržana unutar prethodne. Na slici levo prikazan je primer tri ugneždene repeat-until petlje.

- Nadovezane petlje su uzastopne petlje u kodu - naredna petlja počinje nakon završetka bloka prethodne petlje. Slika desno prikazuje tri repeat-until petlje koje se nadovezuju.



Kako testirati petlje?

U zavisnosti od tipa petlje, postoje sledeći načini:

Uputstvo za proste petlje

- Test primer koji ne izvršava telo petlje.
- Test primer koji izvršava telo petlje tačno jedan put.
- Test primer koji izvršava telo petlje tačno dva puta.
- Test primer koji će telo petlje izvršiti tipičan broj puta.

Ukoliko postoji definisana gornja granica, n, za broj puta koliko petlja može biti izvršena, onda treba dodati i sledeće test primere:

- Test primer koji izvršava petlju tačno n-1 put.
- Dizajnirati test primer koji telo petlje izvršava tačno n puta.
- Pokušati dizajn test primera koji bi telo petlje izvršio tačno n+1 puta.

Uputstvo za ugneždene petlje

- Primeniti testiranje proste petlje za najugneždeniju petlju, dok se za ostale petlje uzima minimalni broj (veći od 0) iteracija.
- Zatim krenuti ka spoljnim petljama, primenjujući prethodno opisani test za svaku petlju, broj iteracija okružujućih petlji treba da bude minimalan mogući (ovo podrazumeva da se odabere minimalna vrednost koja će omogućiti izvršavanje unutrašnje petlje željeni broj puta), unutrašnje petlje treba da se izvršavaju "standardni" broj puta.

Uputstvo za nadovezane petlje

Ukoliko su petlje nezavisne, tako da broj iteracija jedne petlje ne zavisi od broja iteracija bilo koje druge petlje, onda je najjednostavnije primeniti uputstvo za proste petlje na svaku petlju redom kojim se pojavljuje u kodu.

Sa druge strane, ukoliko broj iteracija jedne petlje zavisi od broja iteracija druge petlje onda treba primeniti sledeća pravila:

- Primeniti uputstvo za proste petlje na poslednju petlju u kodu, pri čemu broj iteracija prethodnih petlji treba držati na minimalnoj vrednosti.
- Razmatrati sve petlje do prve u kodu. Na svaku primeniti testiranje proste petlje, pri čemu svaka prethodna petlja treba da bude na minimalnoj vrednosti, a svaka petlja koja sledi u standardnim granicama.

Zadatak 1.

U sledećem programu za sortiranje testirati petlje:

```
S1                  i := 2
C1                  while (i is less than or equal to n) do
S2                     j := i - 1
C2                     while ((j is greater than or equal to 1) and
                                 (A[j] is greater than A[j+1])) do
S3                        temp := A[j]
S4                        A[j] := A[j+1]
S5                        A[j+1] := temp
S6                        j := j-1
                       end while
S7                     i := i + 1
                    end while
```

Rešenje:

Navedeni program sadrži dve while-do petlje.

U navedenom primeru, maksimalni broj iteracija unutrašnje petlje je za jedan manji od trenutne vrednosti promenljive *i*, s obzirom da je *int* promenljiva *j* inicijalizovana na vrednost *i-1* pre ulaska u unutrašnju petlju. Promenljiva *j* dekrementira se na kraju jednog ciklusa i petlja se završava kada vrednost promenljive *j* postane jednaka 0.

Minimalni broj iteracija unutrašnje petlje je 0, a pošto se petlja može završiti trenutno (i to se dešava ukoliko element A[i-1] ima vrednost koja je manja ili jednaka vrednosti elementa A[i]).

Takođe, treba primetiti da kada započne izvršavanje unutrašnje petlje, za datu vrednost *i*, element na poziciji A[i] niza je element koji je na toj poziciji bio na početku izvršavanja, dok su elementi na pozicijama A[1], A[2],…, A[i-1] preuređeni, tako da se u nizu pojavljuju u rastućem redosledu.

Ukoliko je *k* izmedju 0 i *i*, petlja će biti izvršena tačno *k* puta u ovom ternutku, ako i samo ako je A[i] manje od tačno *k* elemenata koji se pojavljuju na prvih *i-1* pozicija u ulaznom nizu: kada se unutrašnja petlja izvršava za datu vrednost *i*, ulazi na prvih *i-1* pozicija su sortirani. Stoga, distance koju element na poziciji *i* treba pomeriti unapred, u cilju sortiranja prvih *i* ulaza niza, je ista kao i broj ulaza na prvih *i-1* pozicija koji su manji od tog elementa - i ovo je isto kao broj iteracija unutrašnje petlje korišćene u ovom trenutku.

Određivanje broja izvršavanja za spoljašnju petlju znatno je jednostavnije: ova petlja će uvek biti izvršena tačno *n-1* puta, ukoliko je *n* veličina ulaznog niza.

Pretpostavimo da je ovaj program u mogućnosti za sortiranje nizova od barem 100 elemenata. U tom slučaju i unutrašnja i spoljašnja petlja treba da budu testirane srednjim vrednostima (na primer veličine između 40 i 60), kao i veličinama 1, 2, 99 i 100.

Kako bi broj iteracija unutrašnje petlje bio tipičan, dok se pokušava kontrola broja iteracija spoljašnje petlje, najverovatnije je efikasno započeti sa nizom čiji su elementi odabrani na slučajan način. Kako bi se broj iteracija unutrašnje petlje držao na "minimalnoj" vrednosti, potrebno je upotrebiti niz koji je već sortiran u rastućem redosledu. Na kraju, kako bi smo broj iteracija unutrašnje petlje postavili na maksimalnu vrednost, potrebno je upotrebiti različite elemente, koji su inicijalno sortirani u opadajućem redosledu, umesto u rastućem.

Test koji će obezbediti da se unutrašnja petlja izvrši dva puta je sledeći:

Ulaz: n = 3; A[1] = 3, A[2] = 2, A[3] = 1
Očekivani izlaz: A[1] = 1, A[2] = 2, A[3] = 3

Sledeće, razmotrimo primere koji će unutrašnju petlju izvršiti 50. puta, 98. puta, 99. puta i 100. puta. Kako bi smo ovo izvršili dok broj iteracija spoljašnje petlje čuvamo na najmanjoj mogućoj vrednosti, koristićemo testove gde su dužine nizova 51, 99, 100 i 101 respektivno, tako da je najmanji element na kraju ulaznog niza u svakom slučaju.

Test
Ulaz: n = 51, A[i] i for i between 1 and 50, and A[51] = 0
Očekivani izlaz: A[i] = [i-1] for i between 1 and 51

Test
Ulaz: n = 99

- A[i] = 1 if i is between 1 and 98 and is even
- A[i] = 2 if i is between 1 and 98 and is odd,
- A[99] = 0

Očekivani izlaz:

- A[1] = 0,
- A[i] = 1 if i is between 2 and 50,
- A[i] = 2 if i is between 51 and 99

Test
Ulaz: n = 100, A[i] = 101 - i for i between 1 and 100
Očekivani izlaz: A[i] = i for i between 1 and 100

Test
Ulaz: n = 101,

- A[i] = 2 if i is between 1 and 50,
- A[i] = 1 if i is between 51 and 100,
- A[101] = 0

Očekivani izlaz:

- A[1] = 0,
- A[i] = 1 if i is between 2 and 51,

- A[i] = 2 if i is between 52 and 101

Konačno, potrebno je definisati testove kako bi se istestirala spoljašnja petlja. Treba dizajnirati testove kako bi se petlja izvršila 50. puta, 98. puta, 99. puta i 100. puta, dok se vrednosti unutrašnje petlje čuvaju na tipičnim vrednostima.

U cilju održavanja unutrašnje petlje standardni broj puta, najbolje je izabrati slučajno popunjavanje ulaznog niza.

POKRIVANJE LINEARNO NEZAVISNIH PUTANJA

Teorijske osnove

Pokrivanje putanja (putevi od startnog do završnog čvora u grafu toka kontrole): potrebno je pokriti bazični skup putanja. Bazični skup putanja je skup svih linearno nezavisnih putanja. Nezavisna putanja: razlikuje se od svih drugih bar po jednom svom segmentu.

Za odredjeni graf može se odrediti u opštem slučaju više različitih bazičnih skupova, ali svi imaju istu kardinalnost. McCabe-ov broj ciklomatske kompleksnosti programa V(G) daje broj linearno nezavisnih putanja u grafu.

Računanje: $V(G) = e - n + 2p$,
gde je e - broj grana grafa,
n - broj čvorova grafa,
p - broj povezanih komponenata grafa
(p = 1 za regularne procedure i funkcije, to jest,
ako postoje jedinstveni ulazni i izlazni čvorovi, svi čvorovi dostižni iz startnog).

Alternativno računanje: $V(G) = b + 1$,
gde je b broj odluka u programu.

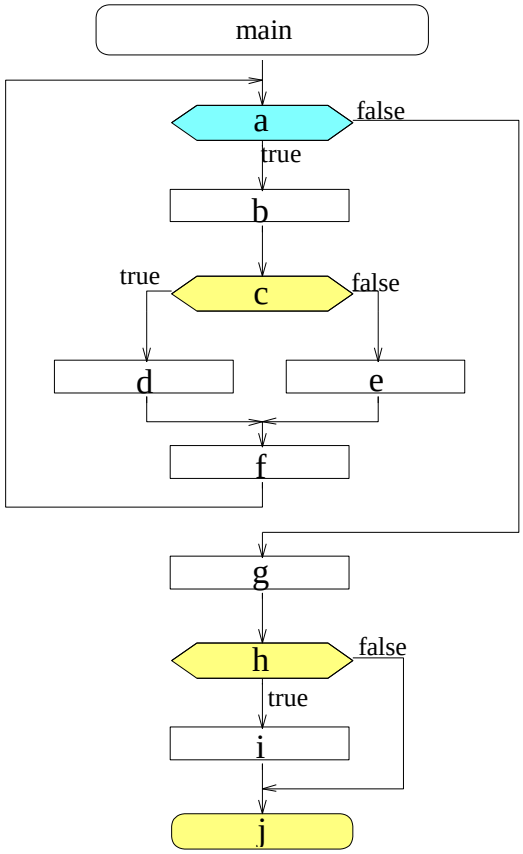
Zadatak 1.

Metodom pokrivanja svih linearno nezavisnih putanja odrediti skup testova za sledeći C++ program koji određuje da li nastavniku treba povećati platu na osnovu pokazanog uspeha njegovih studenata na kvalifikacionom ispitu.

```
#include <iostream.h>
  int main () {
    int passes = 0, failures = 0, studentCounter = 1, result;
1   while (studentCounter <= 10) {
2     cout<<"Enter result (1=pass,2=fail):";
3     cin >> result;
4     if (result == 1)
5       passes++;
6     else
        failures++;
7     studentCounter++;
    }
8   cout << "Polozili " << passes << endl;
9   cout << "Pali " << failures << endl;

10  if (passes > 8)
11    cout << "Podici platu " << endl;
12  return 0;
  }
```

Rešenje:

Prilikom rešavanja najpre crtamo graf toka kontrole:



Sa grafa je moguće pročitati vrednosti za e i n: e = 13, n = 11.

V(G) = e – n + 2 = 4

Potrebno je odrediti 4 nezavisne putanje (izbor nije jednoznačan).

1. a – g – h – j
2. a – b – c – e – f – a – g – h – j
3. a – b – c – d – f – a – g – h – j
4. a – b – c – d – f – a – g – h – i – j

Prilikom rešavanja nije ispoštovana McCabe-ova preporuka za pronalaženje bazičnih putanja (pogledati predavanja). Dalje, treba odrediti testove koji pokrivaju određene putanje. Putanja 1, ne može se pokriti testom, s obzirom da se uvek prvo prolazi kroz while petlju 10 puta da bi while uslov postao false.

Ostale putanje malo modifikujemo. Kod putanja 2 i 4 podvučeni deo se ponavlja još 9 puta zbog while uslova i tada ih je moguće pokriti.

Kod putanje 3, posle a – b – c – d – f dodajemo 9 puta a – b – c – e – f.

Dakle dobijamo 3 test primera:

1. Za putanju 2: ulaz je 2, 2, 2, 2, 2, 2, 2, 2, 2, 2.
   Očekivani izlaz je: Polozili 0 Pali 10
2. Za putanju 3: ulaz je 1, 2, 2, 2, 2, 2, 2, 2, 2, 2.
   Očekivani izlaz je: Polozili 1 Pali 9
3. Za putanju 4: ulaz je 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.
   Očekivani izlaz je: Polozili 10 Pali 0 Podici platu

LCSAJ (eng. LINEAR CODE SEQUENCE AND JUMP)

Teorijske osnove

LCSAJ (*Linear Code Sequence And Jump*) se definiše kao linearna sekvenca koda koja se izvršava ili od početka programa ili od mesta gde tok kontrole može skočiti, pa sve do kraja, ili do narednog skoka.

Zadatak 1.

```
1.    A = 1
2.    IF X1 GOTO 40
3.    B = 1
4.    IF X2 GOTO 40
5.    A = 2
6.    GOTO 50
7.    40 A = A + B
8.    IF X3 GOTO 50
9.    B = B + 1
10.   50 WRITE(A, B)
```

Rešenje:

Počinje se od liste grana u programu (navedeni su i uslovi koji treba da budu zadovoljeni da bi se grana izvršila).

2 -> 3 (X1 false)
2 -> 7 (X1 true) SKOK!
4 -> 5 (X2 false)
4 -> 7 (X2 true) SKOK!
6 -> 10 (true) SKOK!
8 -> 9 (X3 = false)
8 -> 10 (X3 = true) SKOK!

Pomoću prethodne liste nalazimo LCSAJ početne tačke. To je početak programa (linija 1), zatim linije na koje kontrola skače, ukoliko to nije sa prethodne naredbe (linije 7 i 10).

Postoje tri LCSAJ sekvence koji počinju od linije 1 (gledamo gde sve počev od 1 može da se izvrši skok):
(1,2,7), (1,2,3,4,7) i (1,2,3,4,5,6,10)

LCSAJ sekvence sa početkom kod tačke 7:
(7,8,10) i (7,8,9,10)

Zadatak 2.

```
1.    READ(X, Y)
2.    IF X GOTO 10
3.    Z = 1
4.    GOTO 20
5.    10 W = 1
6.    20 IF Y GOTO 30
7.    Z = W + Z
8.    GOTO 40
9.    30 Z = Z - 2
10.   40 END
```

Rešenje:

Postoje sledeće sekvence LCSAJ:

(1, 2, 5)
(1, 2, 3, 4, 6)
(5, 6, 9)
(5, 6, 7, 8, 10)
(6, 9)
(6, 7, 8, 10)
(9, 10)

Zadatak 3.

Neka je dat sledeći program u Javi. Odrediti sve LCSAJ za dati program. Pretpostaviti da su sve funkcije definisane i da vraćaju očekivani rezultat.

```
1.   Scanner sc = new Scanner (System.in);
2.   int broajc, x, y, c;
3.   System.out.println ("br?");
4.   brojac = sc.nextInt();
5.   System.out.println ("br?");
6.   x = sc.nextInt();
7.   y = sc.nextInt();
8.   while (brojac > 0) {
9.    if (x < y)
10.        x = razlika(x, y);
11.     else
12.        y = zbir(x, y);
13.  brojac = brojac - 1;
14.  }
15.  c = stepen(x, y);
16.  System.out.println ("Rezultat: " + c);
```

Rešenje:

Do linije 8, izvršava se uvek ista sekvenca (1, 2, 3, ... , 8). Nakon toga imamo WHILE petlju i IF-ELSE uslov, koji mogu biti mesta gde se linearna sekvenca prekida i vrši skok.

Postoje sledeće sekvence LCSAJ:

(1, 2,..., 8, 15)
(1, 2,..., 8, 9, 10, 13)
(1, 2,..., 8, 9, 11)
(11, 12, 13, 14, 8)
(13, 14, 8)
(8, 9, 10, 13)
(8, 9, 11)
(8, 15)
(15, 16)

Zadatak 4.

Neka je dat sledeći deo programskog koda.

a) Odrediti sve LCSAJ za datu metodu.
b) Odrediti minimalan skup testova koji pokrivaju sve LCSAJ.

```
   public void uredi() throws Greska{
1.      int n = niz.length;
2.      for (int i=1; i<n; i++) {
3.          double p = niz[i];
4.          int j = i - 1;
5.          while (j>=0 && p < niz[j]) {
6.                niz[j+1] = niz[j--];
7.                prikazi(); //funkcija za prikazivanje niza
8.          }//while
9.      niz[j+1] = p;
10.     prikazi();
11.     }//for
12.   }
```

Napomena: pretpostaviti da je kod u metodi `prikazi()` linearna sekvenca i da u njoj ne postoje skokovi.

Rešenje:

U ovom zadatku pokazan je rad sa FOR petljom kod LCSAJ. Ono što je važno znati je da pre prvog ulaska u FOR petlju se takođe ispituje ispunjenost uslova petlje.

Postoje sledeće sekvence LCSAJ:

(1, 2, 12)
(1, 2, 3, 4, 5, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 5)
(5, 6, 7, 8, 5)
(9, 10, 11, 2)
(2, 3, 4, 5, 6, 7, 8, 5)
(2, 3, 4, 5, 9)
(2, 12)

GRANIČNO TESTIRANJE UNUTRAŠNJE PUTANJE

Teorijske osnove

Metod graničnog testiranja unutrašnje putanje (eng. *boundary interior path testing*), u programskim grafovima sa zatvorenim putanjama (petljama) zahteva pokrivanje sledećih putanja:

1. Izvršavanje tela petlje 0. puta
2. Izvršavanje tela petlje 1. put (granična vrednost)
3. Ponavljanje tela petlje bar jednom (unutrašnji test)

Za svaki od prethodnih slučajeva potrebno je potpuno prekriti sve (otvorene) putanje u grafu.

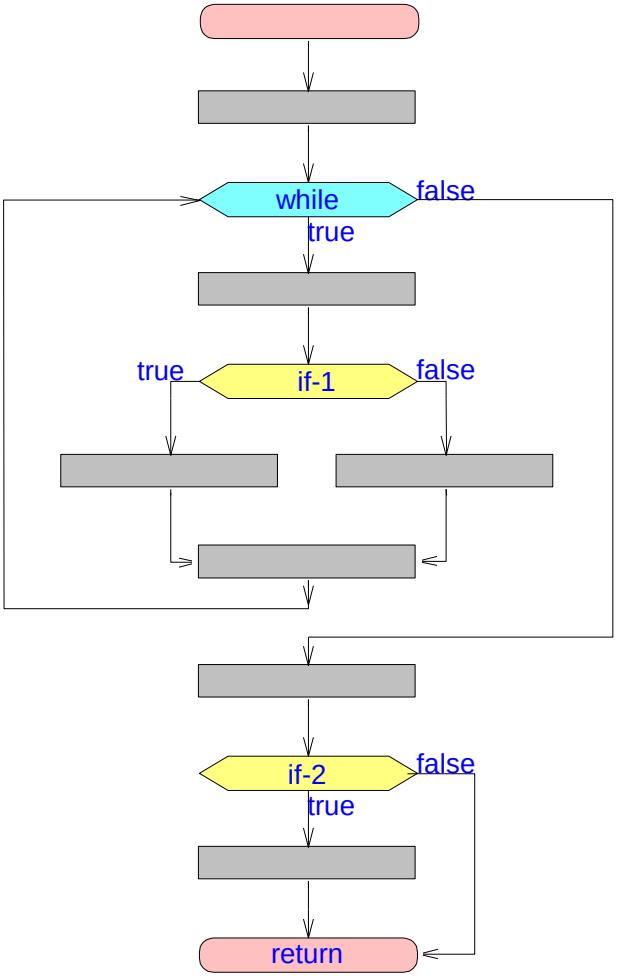
Zadatak 1.

Metodom graničnog testiranja unutrašnje putanje (boundary interior path testing) odrediti test primere za dati C++ program.

```
#include <iostream.h>

int main () {
  int passes = 0, failures = 0;
  int studentCounter, result;

  cout<<"Number of students:";
  cin>>studentCounter;

  while (studentCounter > 0)  {
    cout<<"Enter result (1=pass,2=fail):";
    cin >> result;
    if (result == 1) passes++;
    else failures++;

    studentCounter--;
  }

  cout << "Passed " << passes << endl;
  cout << "Failed " << failures << endl;

  if (passes > 8) cout<<"Raise Tuition";
  return 0;
}
```

Rešenje:

Zadatom programu odgovara sledeći graf:



1. Za slučaj izvršavanja tela petlje 0 puta, postoje 2 putanje u grafu:
1.1 false grana while, false grana if-2. Odgovarajući test primer je: 0 studenata.
1.2 false grana while, true grana if-2. Ovde nije moguće odrediti test primer (ako ima 0 studenata, nemoguće je postići passes>8).

2. Za slučaj izvršavanja tela petlje 1 put, postoje 4 putanje u grafu:
2.1 true grana while, false grana if-1, false grana while, false grana if-2. Odgovarajući test primer je: 1 student koji je pao.
2.2 true grana while, true grana if-1, false grana while, false grana if-2. Odgovarajući test primer je: 1 student koji je prošao.
2.3 true grana while, false grana if-1, false grana while, true grana if-2. Ovde nije moguće odrediti test primer (ako ima 1 student, nemoguće je postići passes>8).
2.4 true grana while, true grana if-1, false grana while, true grana if-2. Ovde nije moguće odrediti test primer (ako ima 1 student, nemoguće je postići passes>8).

3. Za slučaj ponavljanja tela petlje 1 ili više puta, postoje 4 putanje od interesa u grafu:

3.1 true grana while, false grana if-1, true grana while,..., false grana while, false grana if-2. Odgovarajući test primer je: 2 studenta koji su pali.
3.2 true grana while, true grana if-1, true grana while,..., false grana while, false grana if-2. Odgovarajući test primer je: 2 studenta koji su prošli.
3.3 true grana while, false grana if-1, true grana while,..., false grana while, true grana if-2. Odgovarajući test primer je: 10 studenata, prvi pao, ostali prošli.
3.4 true grana while, true grana if-1, true grana while,..., false grana while, true grana if-2. Odgovarajući test primer je: 10 studenata, svi prošli.

Primetiti razliku izmedju 2. i 3. tačke.

TESTIRANJE METODOM TOKA PODATAKA (Data Flow Testing)

Teorijske osnove

Selekcija programskih putanja na osnovu lokacija u programu gde se promenljivama dodeljuje vrednost ili gde se ta vrednost koristi.

Postoji veći broj strategija testiranja na osnovu toka podataka; mi ćemo obraditi samo osnovne.

Def. Za iskaz, označen sa, S, možemo definisati sledeće skupove:

- DEF(S) = {X | iskaz S sadrži dodelu vrednosti promenljivoj X}
- USE(S) = {X | iskaz S sadrži upotrebu promenljive X}
- Primer: 1: a = b; DEF(1) = {a}, USE(1) = {b}.
- Primer: 2: a = a + b; DEF(2) = {a}, USE(2) = {a, b}.

Def. Dodela promenljivoj X u iskazu S je Živa (Live) u iskazu S1, ako postoji putanja izvršavanja programa od S do S1 koja nigde ne sadrži dodelu vrednosti promenljivoj X.

Def. Lanac dodele-upotrebe (definition-use chain, DU chain) promenljive X u oznaci [X, S, S1], gde su:

- S i S1 iskazi
- X € DEF(S)
- X € USE(S1)
- Dodela promenljivoj X u iskazu S je živa u iskazu S1.
- Primer:

```
if (exp)     // 1
  x = 1;     // 2
else         // 3
  x = 2;     // 4

if (case)    // 5
  P1(x);     // 6
else         // 7
  P2(x);     // 8
```

- DEF(2) = DEF(4) = {x}
- USE(6) = USE(8) = {x}
- Ima četiri DU lanca za x: [x, 2, 6] [x, 2, 8] [x, 4, 6] [x, 4, 8]

(Prvo numerišemo svaki iskaz, a zatim uočavamo klase DEF i USE za svaki iskaz. Mogu se posmatrati koje se nalaze u skupu DEF. Na osnovu nadjenih skupova formiramo DU lance)

Def. DU strategija testiranja: svaki DU lanac u programu treba da bude pokriven bar jedanput (najjednostavnija strategija toka podataka).

Primer: test primeri koji pokrivaju DU lance iz prethodnog primera

1. (exp = true, case = true) pokriva DU lanac [x, 2, 6]
2. (exp = true, case = false) pokriva DU lanac [x, 2, 8]
3. (exp = false, case = true) pokriva DU lanac [x, 4, 6]
4. (expr = false, case = false) pokriva DU lanac [x, 4, 8]

Napomena: DU strategija testiranja ne garantuje pokrivenost svih odluka (grana) u programu. if-then, then deo ne sadrži nijednu dodelu vrednosti, bez else grane => false odluka neće biti pokrivena DU testiranjem.

Zadatak 1.

Za sledeći programski fragment, strategijom pokrivanja svih DU lanaca, odrediti test primere.
Napomena: izostaviti iz razmatranja promenljive num_locks, num_stocks i commission.

```
num_locks = 0;
num_stocks = 0;

S1:  read( locks );

S2:  while locks != -1 do
     begin
S3:    read( stocks );
S4:    num_locks := num_locks + locks;
S5:    num_stocks := num_stocks + stocks;
S6:    read( locks );
     end;

S7:  sales := 45 * num_locks + 30 * num_stocks;

S8:  if( sales > 1000.0 ) then
     begin
S9:    commission := 0.1 * 1000;
S10:   commission := commission + 0.2 * (sales - 1000);
     end
     else
     begin
S11:   commission := 0.1 * sales;
     end;
```

Rešenje:

Napomena: Kada imamo petlje može se javiti dodela u nekom broju koji je veći od koraka korišćenja. U našem primeru promenljiva locks.

Za promenljivu *locks*:

DEF: S1, S6; USE: S2, S4;

Dakle postoje četiri DU lanca:

1. c1 = [locks, S1, S2]
2. c2 = [locks, S1, S4]
3. c3 = [locks, S6, S2]
4. c4 = [locks, S6, S4]

Za promenljivu *stocks*:

DEF: S3; USE: S5

Dakle postoji jedan DU lanac

1. c5 = [stocks, S3, S5]

Za promenljivu *sales*:

DEF: S7; USE: S8, S10, S11;

Dakle postoje tri DU lanca:

1. c6 = [sales, S7, S10]
2. c7 = [sales, S7, S11]
3. c8 = [sales, S7, S8]

dva test primera su dovoljna da pokriju sve navedene DU lance:
TP1: locks = 20x, stocks = 20x; (pokriva DU lance: c1, c2, c3, c4, c5, c6, c8)
TP2: locks = 10x, stocks = 10x; (pokriva DU lance: c1, c2, c3, c4, c5, c7, c8);

ISPITNI ZADACI

Zadatak 1.
Ukoliko su u nekom programskom segmentu pokrivene sve odluke, da li su pokriveni:

a) svi elementarni uslovi
b) sve DU putanje
c) svi razvijeni uslovi

Za svaki NE odgovor navesti primer.

Rešenje:

a) NE
Primer: if (a or b) then ...

Ukoliko uzmemo kombinacije

a	b
0	0
0	1

Sa navedenim vrednostima sve odluke će biti pokrivene. Sa druge strane za elementarni uslov a nismo pokrili vrednost 1.

b) NE
Primer:

```
gr = read();     // 1
a = 0;           // 2
i = 0;           // 3

while (i < gr){ // 4
  a = a + 1;     // 5
  i = i + 1;     // 6
}                // 7
```

Za dati primer pronalazimo sledeće DU lance:

[a, 1, 4] [a, 4, 4] [i, 2, 5] [i, 5, 5]

Ukoliko za ulazni podatak gr uzmemo vrednost 1, onda će se izvršiti jedna iteracija petlje, tj biće pokriveni lanci [a, 1, 4] [i, 2, 5], petlja se završava sve odluke su pokrivene ali ne i svi DU lanci.

c) NE
Primer: Kako su elementarni uslovi podkup razvijenih uslova (npr. a or b or (c and d)), to zanači, kao i kod tačke a, nisu pokriveni svi razvijeni uslovi.

a, b, c, d, a or b, c and d, a or b or (c and d)

Zadatak 2.

Naznačiti koliko ukupno definicija, c-upotreba i p-upotreba po svakoj od navedenih celobrojnih promenljivih - *domaci, poeni, ukupno, brojac_studenata,* ima u sledećem Java programu, koji računa ocene na predmetu.

Napomena: pretpostaviti da su sve metode *citajDomaci*, *citajPoeneIspit* i *citajPoeneProjekat* definisane.

```
package ispit;

public class Ocena {
        public static void main(String args[]) {
(1)             int ukupno_studenata = args.length;
(2)             int brojac_studenata = 0;
(3)             while (brojac_studenata < ukupno_studenata) {
(4)                     System.out.println(args[brojac_studenata]);
(5)                     int broj_domacih = 4;
(6)                     int domaci = 0, ukupno = 0;

(7)                     while (domaci < broj_domacih) {
(8)                        int poeni = citajDomaci(args[brojac_studenata],
                           domaci);
(9)                        if (poeni < 0)
(10)                              System.out.println("Nekorektan unos");
(11)                       else {
(12)                              ukupno += poeni;
(13)                              domaci++;
(14)                       }
(15)                    }
(16)                    int poeni = citajPoeneIspit(args[brojac_studenata]);
(17)                    if (poeni < 0)
(18)                          System.out.println("Nekorektan unos");
(19)                    else if (poeni >= 31 && poeni <= 50)
(20)                          ukupno += poeni;

(21)                    poeni = citajPoeneProjekat(args[brojac_studenata]);
(22)                    if (poeni >= 10 && poeni <= 30)
(23)                          ukupno += poeni;

(24)                    System.out.println("Ukupno poena " + ukupno);

(25)                    if (ukupno >= 91)
(26)                          System.out.println("Ocena 10");
(27)                    else if(ukupno >= 81)
(28)                          System.out.println("Ocena 9");
(29)                    else if(ukupno >= 71)
(30)                          System.out.println("Ocena 8");
(31)                    brojac_studenata++;
(32)            }
(33)    }
(34)}
```

Rešenje:

Osnovni pojmovi naučeni na predavanjima:

- Upotreba se naziva **predikatskom** (**p-use**) ako se pojavljuje u predikatskom izrazu naredbi kontrole toka (if, while, switch itd.)
- U suprotnom, upotreba se naziva računskom (**computational use** or **c-use**)

Ovde treba obratiti pažnju kod aritmetičkih operatora sa dodelom vrednosti i inkrementiranja/dekrementiranja. Na primer, kao u sledećim izrazima:

```
(12)  ukupno += poeni;
(13)  domaci++;
```

Ovde imamo:

- u 12.liniji koda: definisanje promenljive ukupno i c-upotrebu promenljive poeni, ali i pormenljive ukupno, jer se ovaj izraz može napisati i kao:

```
(12)  ukupno = ukupno + poeni;
```

- u 13.liniji koda: inkrementiranja promenljive domaci, što znači da je to i definicija i c-upotreba te promenljive, jer se ovaj izraz može napisati i kao:

```
(13)  domaci=domaci+1;
```

Naziv promenljive	*definicija*	*c-upotreba*	*p-upotreba*
domaci	(6), (13)	(8), (13)	(7)
poeni	(8), (16), (21)	(12), (20), (23)	(9), (17), (19), (22)
ukupno	(6), (12), (20), (23)	(12), (20), (23), (24)	(25), (27), (29)
brojac_studenata	(2), (31)	(4), (8), (16), (21), (31)	(3)

Zadatak 3.

Data je C funkcija bintr() za binarno pretraživanje po ključu K sadržaja rastuće sortiranog vektora M od N elemenata. Rezultat je indeks elementa koji sadrži K, ili -1 ako traženje nije uspelo.

```
int bintr(int K, int M[], int N){
   int ID = 0;
   int IG = N - 1;
   while (ID <= IG) {
      IS = (ID + IG) / 2;
      if (K == M[IS]) return IS;
      else if (K < M[IS]) IG = IS;
      else ID = IS;
   }
   return -1;
}
```

a) Odrediti sve DU lance po promenljivama ID i IG. Formirati test primere koji pokrivaju nađene lance.

b) Nacrtati graf toka kontrole i odrediti broj ciklomatske kompleksnosti. Navesti bazični skup putanja i svaku putanju pokriti sa po jednim test primerom.

Rešenje:

Zbog analize rešenja zadatka, numerisaćemo svaku liniju datog koda:

```
0     int bintr(int K, int M[], int N){
1        int ID = 0;
2        int IG = N - 1;
3        while (ID <= IG) {
4           IS = (ID + IG) / 2;
5           if (K == M[IS]) return IS;
6           else if (K < M[IS]) IG = IS;
7           else ID = IS;
8        }
9        return -1;
10    }
```

a) Za tražene promenljive, odredićemo u kojim linijama koda se definišu i koriste:

Promenljive	DEF	USE
ID	1, 7	3, 4
IG	2, 6	3, 4

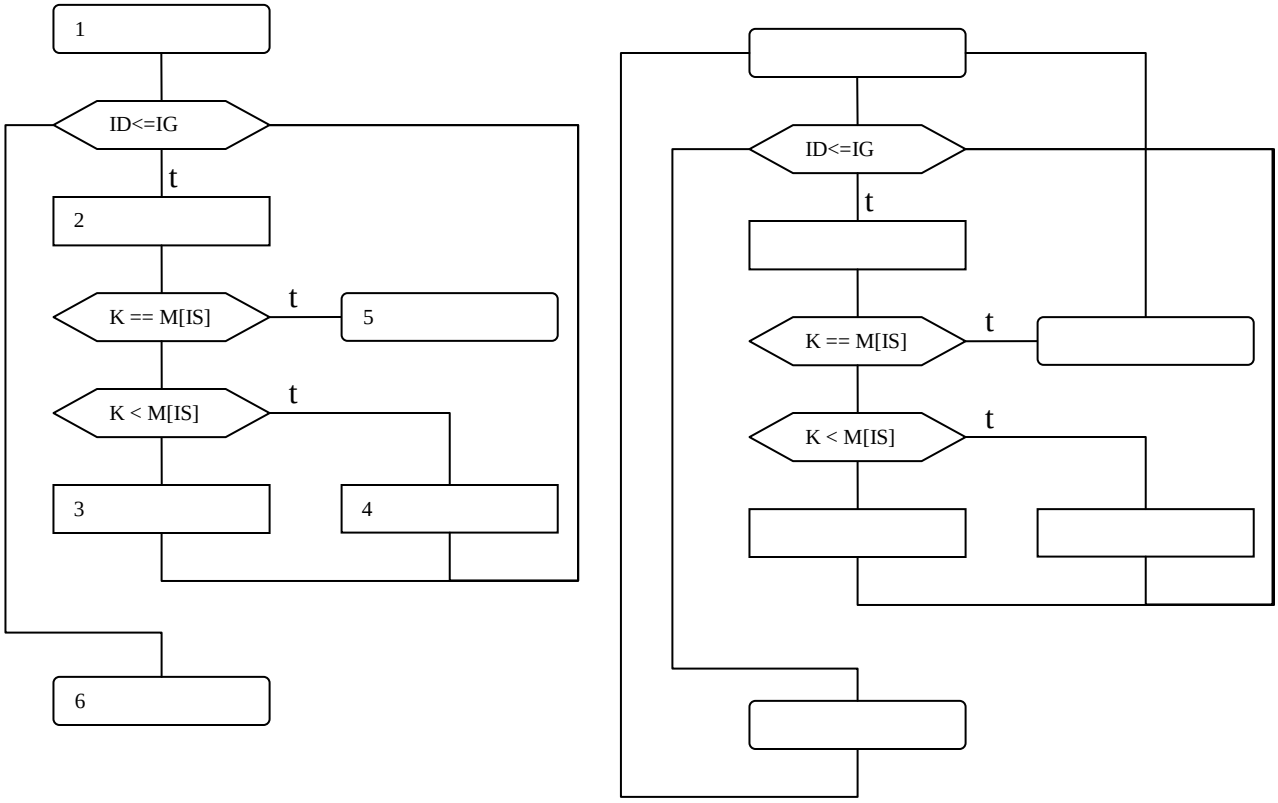
Formiraćemo sledeće DU lance:

[ID,1,3] [ID,1,4] [ID,7,3] [ID,7,4]
[IG,2,3] [IG,2,4] [IG,6,3] [IG,6,4]

Test primer koji bi pokrivao sve DU lance može biti niz M sa 10 elemenata:

M=[1,2,3,4,5,6,7,8,9,10]
N=10
K=4

b) Graf kontrole toka za dati kod izgleda ovako:



Kao što vidimo na levoj slici imamo dva izlazna čvora (čvorovi 5 i 6), pa ne možemo da primenimo formulu V=e-n+2, tako da imamo dve opcije:

- Da primenimo formulu V=b+1, gde je b=3 broj binarnih odluka, pa dobijemo da je V=4;
- Da spojimo sve izlazne čvorove sa ulaznim (kao na slici desno) i da primenimo fomulu V=e-n+1=12-9+1=4.

Bazični skup putanja je:
{ [1, while, 6],
[1, while, 2, if-1, 5],
[1, while, 2, if-1, if-2, 3, while, 6],
[1, while, 2, if-1, if-2, 4, while, 6] }

Prva bazična putanja, pokrivena je ukoliko je ulazni argument N=0, odnosno funkciji se prosleđuje prazan niz. Tada kod While petlje uslov nije ispunjen, pa se odmah ide na krajnju naredbu. Za pokrivanje ostalih bazičnih putanja, koristićemo niz od 3 elementa. Test primeri koji pokrivaju ove bazične putanje:
TP1: N = 0; M = {}; K ne utiče
TP2: N = 3; M = {1, 2, 3}; K=2;
TP3: N = 3; M = {1, 2, 3}; K=3;
TP4: N = 3; M = {1, 2, 3}; K=1;

Napomena:

Sve primedbe, sugestije i eventualne greške u zadacima poslati na: drazen.draskovic@etf.bg.ac.rs