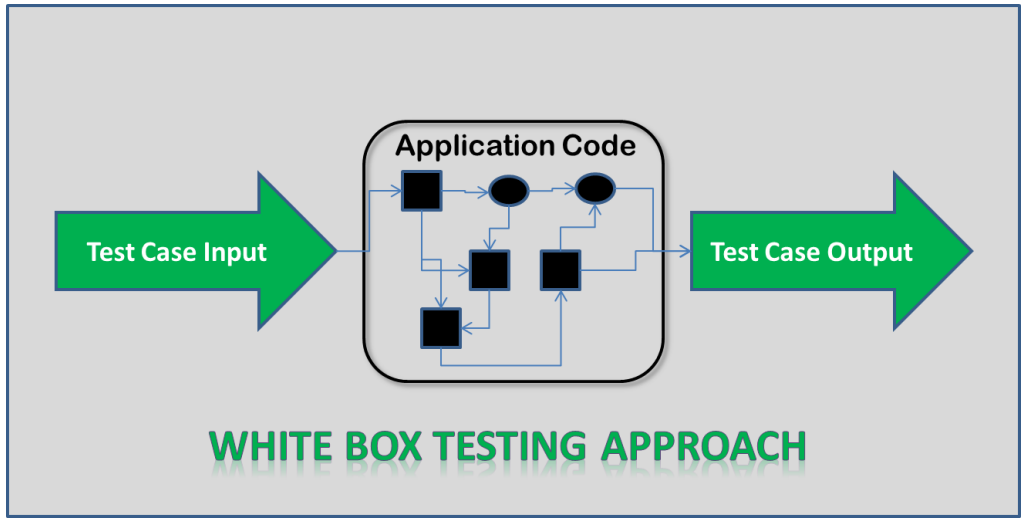


Tehnike bele kutije (white-box)

Bela kutija

- Strukturno testiranje, još poznato i kao testiranje metodama bele ili staklene kutije, je tehnika testiranja u kojoj je stvarna implementacija softvera poznata testerima.
- Dok se u tehnikama crne kutije tester fokusira na to šta sistem radi, u metodama bele kutije focus je na tome kako sistem radi.
- Cilj testiranja je da se izvrše sve programske strukture, kao i strukture podataka u programu. Specifikacija se ne proverava prilikom primene metoda bele kutije, već se posmatra samo kod.

Bela kutija

- Strukturno testiranje se može primeniti na svim nivoima testiranja (jedinično, integraciono, sistemsko), i najčešce se primenjuje nakon metoda crne kutije, koje su bazirane na specifikaciji i verifikuju funkciju sistema.
- Sami programeri često koriste strukturno testiranje na jediničnom i integracionom nivou, pošto se očekuje da su komponente koje su razvili temeljno testirane pre nego što se obavi dalja integracija tih komponenti u sistem i isporuči testerima na testiranje.
- Dva različita testa mogu da aktiviraju identične stavke u programu i da postignu istu pokrivenost, ali na osnovu različitih ulaznih podataka jedan može pronaći grešku, dok drugi ne.

Testiranje tehnikama bele kutije

Najznačajnije metode bele kutije se mogu podeliti na:

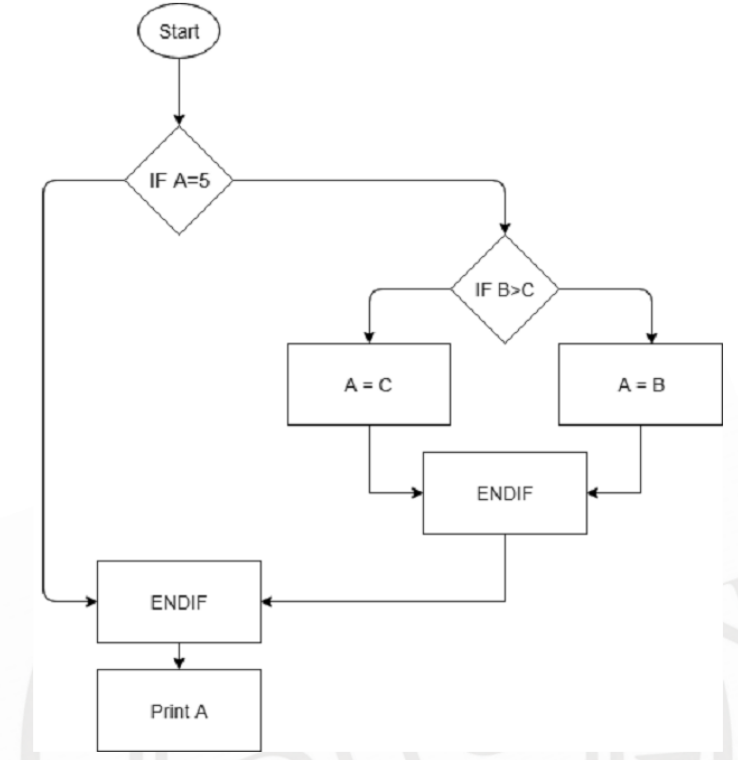
- Tehnike zasnovane na toku kontrole
  1. Pokrivanje iskaza (engl. Statement coverage)
  2. Pokrivanje odluka (engl. Decision/branch coverage)
  3. Pokrivanje putanja (engl. Path coverage)
  4. Pokrivanje uslova (engl. Condition coverage)
- Tehnike zasnovane na toku podataka

Graf toka kontrole

Graf toka kontrole (engl. Control Flow Graph – CFG) predstavlja reprezentaciju programa u obliku grafa, i implicitno prikazuje putanje izvršavanja programa.

Čvorovi ovog grafa su instrukcije (ili bazicni blokovi koda), a grane grafa predstavljaju sekvenciranje instrukcija, na primer ako postoji neki uslov, onda postoji grananje u grafu.

```
IF A=5
    THEN IF B>C
        THEN A=B
        ELSE A=C
    ENDIF
ENDIF
Print A
```



1. Pokrivanje iskaza (Statement Coverage)

Metodologija:

- Test primeri se tako projektuju da se svaki iskaz programa izvrši bar jednom.
- Ne možemo znati da li u nekom iskazu postoji greška ukoliko ga ne izvršimo

Prednosti:

- Verifikacija šta se očekuje da napisani kod radi, a šta ne.
- Može se videti kvalitet koda.
- Proverava da li postoji nedostupan kod (mrtav kod).

Mane:

- Ne mogu se testirati netačni uslovi.
- Ne može se detektovati da li je petlja došla do uslova završetka.
- Ne razume logičke operatore
- Ako se iskaz pravilno izvršava za jednu ulaznu vrednost, ne postoje garancije da će se ispravno izvršavati i za sve ostale moguće ulazne vrednosti.

Primer

```
int f1(int x, int y) {    // Euklidov algoritam
    while (x != y) {     //  Nalaženje najvećeg
      if (x>y) then     //  zajedničkog delioca
            x=x-y;
      else
            y=y-x;
    }
    return x;
}
```

Testiranje euklidovog algoritma

Serija testova:

{(x=3,y=3),(x=4,y=3), (x=3,y=4)}

- Izvršava sve iskaze prethodnog programa bar jednom.

2. Pokrivanje odluka (Branch coverage)

Testovi biraju tako da se svaka od različitih grana uslovnih iskaza izvrši bar jedanput.

Grana (engl. *branch*) predstavlja ishod odluke, pa se pokrivanje odluka svodi na merenje koliko je izlaznih grana uslovnih naredbi pokriveno testovima

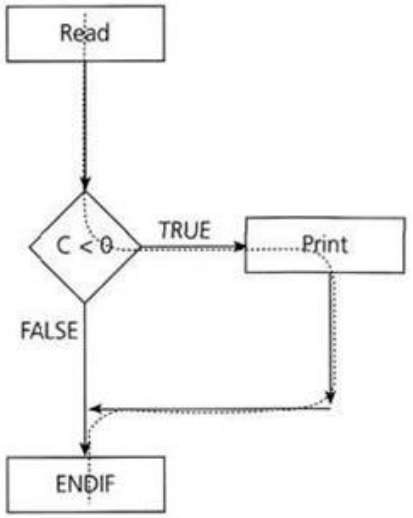
Pod uslovnim iskazom (odnosno odlukom) se podrazumevaju IF naredbe, kontrolne naredbe u petljama (na primer uslov u WHILE petlji) ili SWITCH-CASE naredbe.

Test primeri se projektuju tako da:

- Svaka od različitih grana uslovnih iskaza izvrši bar jednom.

Pokrivanje odluka garantuje pokrivanje iskaza:

- Jači metod od pokrivanja iskaza.



Primer

```
int f1(int x, int y) {    // Euklidov algoritam
    while (x != y) {      //  Nalaženje najvećeg
      if (x>y) then       //  zajedničkog delioca
            x=x-y;
      else
            y=y-x;
    }
    return x;
}
```

- Test primeri za pokrivanje odluka kod Euklidovog algoritma:
  {(x=3,y=3), (x=4,y=3), (x=3,y=4)}

- while (A) {
      B ; break;
  }

Test primer: *A=True* **pokriva sve iskaze** (*while, B, break*), ali **ne pokriva odluku** da se ne uđe u while petlju.

3. Pokrivanje uslova

Pokrivanje uslova (engl. *condition coverage*) je metoda bele kutije koja je bliska pokrivanju odluka, sa većom osetljivošću na tok kontrole.

Međutim, treba biti oprezan, pošto iako je ova metoda kompleksnija od pokrivanja odluka, 100% pokrivenost uslova ne garantuje 100% pokrivenost iskaza.

Test primeri se projektuju tako da:

- Svaka elementarna komponenta složenog uslovnog izraza uzima i tačnu i netačnu vrednost.

Primer:

- Dat je uslovni izraz:
  ((c1 and c2) or c3)
- c1, c2 i c3 ponaosob treba da uzmu obe logičke vrednosti:
- {(c1=T, c2=T, c3=F), (c1=F, c2=F, c3=T)}

Poređenje različitih tehnika

Pokrivanje uslova

- Ne garantuje pokrivanje svih odluka (prethodni primer ne pokriva odluku F)
- Samim tim nije garantovano ni pokrivanje svih iskaza

Neka se posmatra sledeći uslov:

```
if (x > 5 && y == 3) { … }
```

Dva testa, {x = 6, y = 2} i {x = 4, y = 3}, generišu sve moguće vrednosti svih elementarnih uslova (true i false, false i true).

- Oba idu na istu izlaznu granu IF uslova (false grana)

- **100% pokrivenost uslova, nije postignuta ni 100% pokrivenost odluka, ni 100% pokrivenost iskaza.**

3.1 Pokrivanje uslova i odluka

- Pokrivanje uslova i odluka podrazumeva simultano pokrivanje oba kriterijuma. Svaki elementarni uslov mora uzeti i true i false vrednosti, uz dodatak da se svaka odluka kao celina mora evaluirati i sa true i sa false, kako bi se obezbedilo da se sve moguće izlazne grane izvrše.
- Dva testa, {x = 6, y = 3} i {x = 4, y = 2}, generišu sve moguće vrednosti svih elementarnih uslova (true i true, false i false), ali oba sada ne idu na istu granu.
- Prvi test pokriva true izlaznu granu kompletne IF naredbe, dok drugi test pokriva false izlaznu granu.

3.2 Pokrivanje višestrukih uslova (multiple conditions coverage)

Test primeri se projektuju tako da se

- pokriju sve moguće kombinacije vrednosti **elementarnih komponenata** u složenim uslovnim izrazima

Primer

- if (ch == 'x' || ch == 'y')
  ch = 'a';

Test pr. / Elem.komp.	**ch == 'x'**	**ch == 'y'**	**ch == 'a'**	**ch == ?**
ch == 'x' **ch == 'y'**	true false	false true	false false	true true

Pokrivanje višestrukih uslova

Ako uslovni izraz ima n elementarnih komponenata:

- Za pokrivanje razvijenih uslova potrebno nam je u opštem slučaju 2^n test primera.

Praktično jedino ako je n malo.

3.3 Minimalno pokrivanje višestrukih uslova

Test primeri se projektuju tako da

- **Svaka elementarna i neelementarna komponenta** složenog uslovnog izraza uzima i tačnu i netačnu vrednost.

Primer

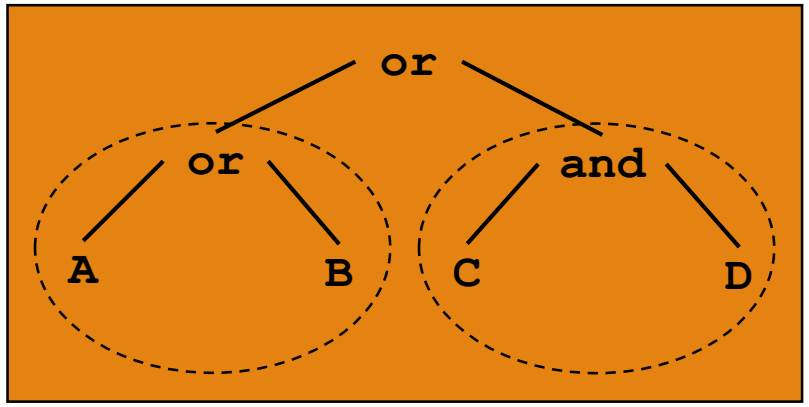
- if (ch == 'x' || ch == 'y')
  ch = 'a';

Test primer / komponente	**ch='x'**	**ch='y'**	**ch='a'**
ch == 'x'	true	false	false
ch == 'y'	false	true	false
(ch=='x'||ch =='y')	true	true	false

Minimalno pokrivanje višestrukih uslova (2)

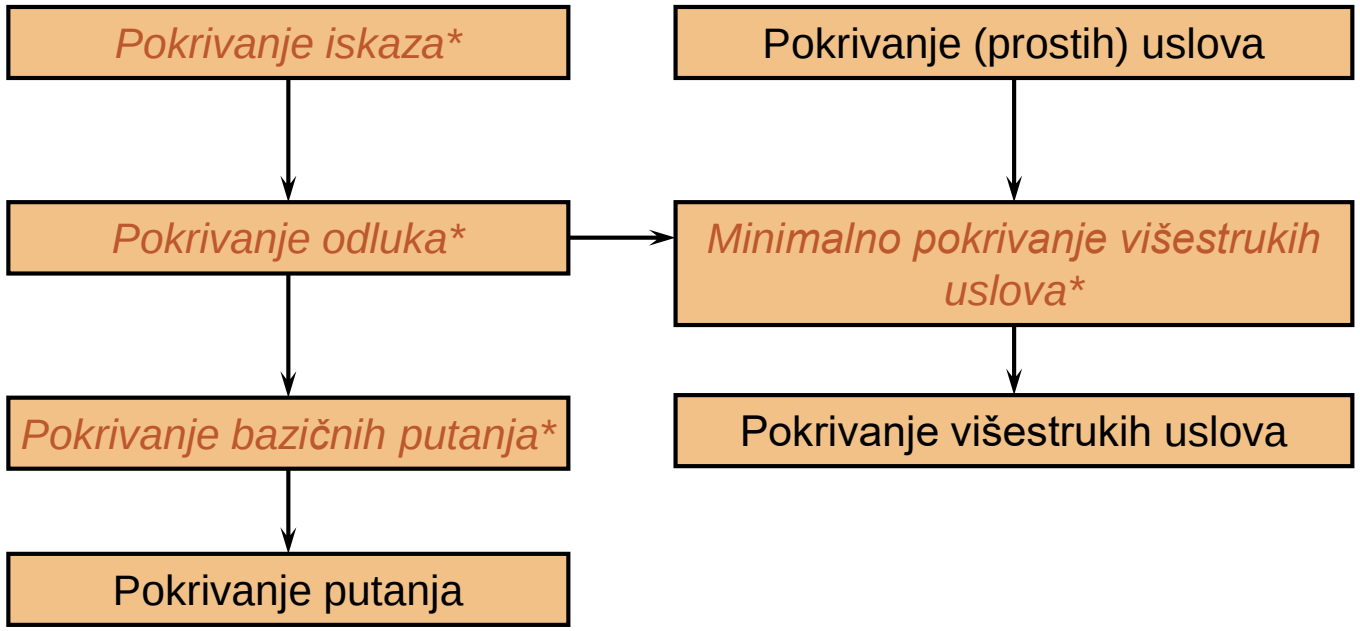
Koje uslove treba razmatrati u sledećem izrazu?

if A or B or (C and D)



- A
- B
- C
- D
- A or B
- C and D
- A or B or (C and D)

Poređenje metoda



T→U = U jače od T

* od praktičnog značaja

4. Pokrivanje putanja

Test primeri se projektuju tako da

- se sve putanje u programu izvrše bar jednom.

Data definicija se zasniva na

- Grafu toka kontrole programa (engl. Control Flow Graph, CFG)

Graf toka kontrole (CFG)

Graf toka kontrole programa opisuje

- Redosled u kome se izvršavaju instrukcije programa.
- Način na koji se kontrola prenosi kroz program.

Kako se iscrtava graf toka kontrole?

Numerisati sve iskaze programa.

Numerisani iskazi:

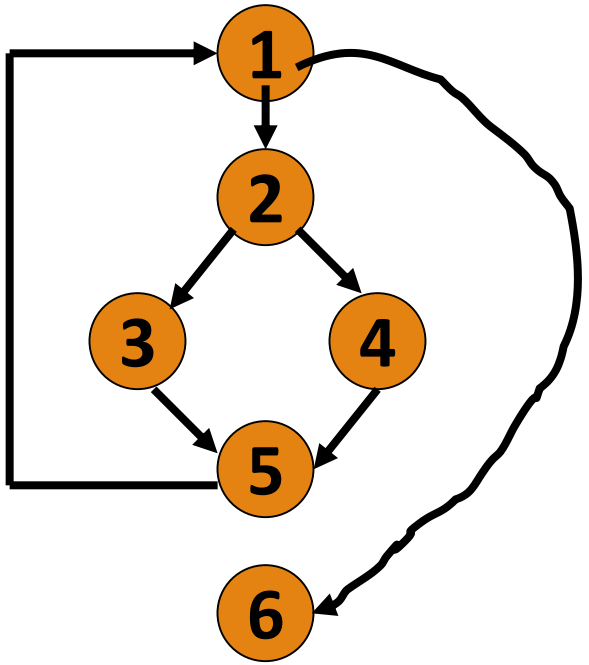
- Predstavljaju se čvorovima u grafu toka kontrole.

Između dva čvora grafa postoji grana:

- Ako po izvršenju iskaza koji je pridružen izvorišnom čvoru kontrola može da se prenese na iskaz koji je pridružen odredišnom čvoru.

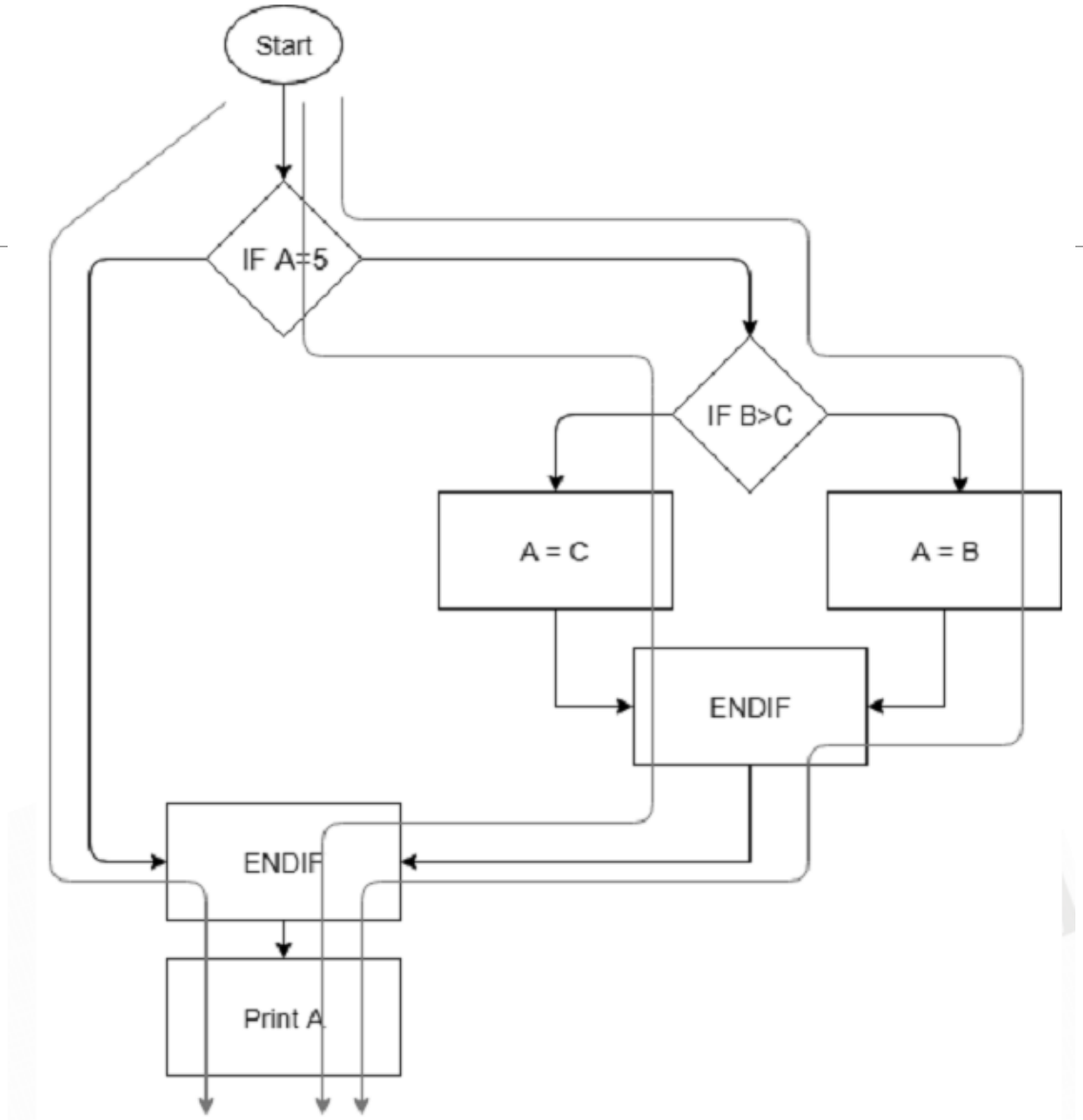
Primer 1

```
int f1(int x, int y){
1.      while (x != y){
2.          if (x>y) then
3.              x=x-y;
4.          else y=y-x;
5.      }
6.      return x;       }
```



Primer 2

```
1 IF A=5
2    THEN IF B>C
3         THEN A=B
4         ELSE A=C
5    ENDIF
6 ENDIF
7 Print A
```

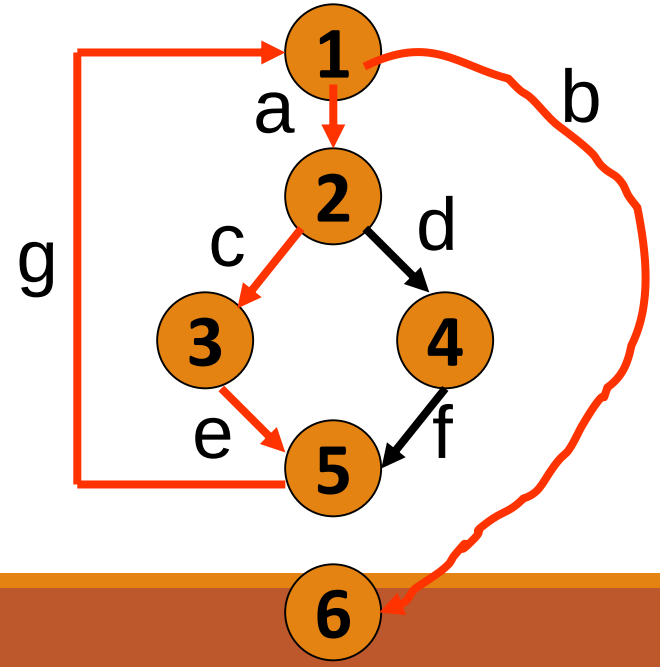


Putanje

Programska putanja je:

- Sekvenca čvorova i grana od početnog do završnog čvora grafa toka kontrole programa.
- Primedba: u opštem slučaju graf kontrole toka može imati više početnih i završnih čvorova.

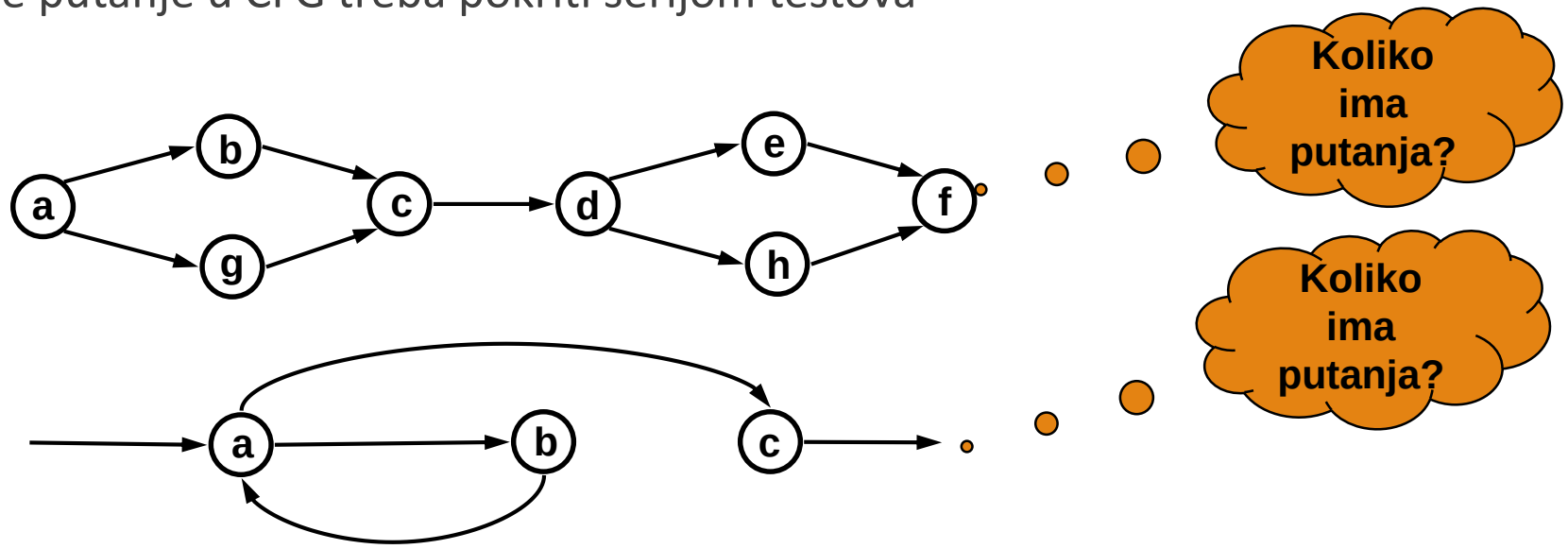
Putanje se predstavljaju sekvencom čvorova (ili alternativno, sekvencom grana):



P1: 1-2-3-5-1-6
P1: a-c-e-g-b

Testiranje potpunim pokrivanjem putanja

Kriterijum: sve putanje u CFG treba pokriti serijom testova



ac, abac, ababac,... i.e.: a{ba}c

- Najbolji pristup – ali nije praktično primenljiv

Pokrivanje bazičnih putanja

Thomas McCabe, sredina ’70tih, primena teorije vektorskih prostora

Projektovati test primere tako da se:

- Sve linearno nezavisne putanje u CFGu izvrše bar jednom!

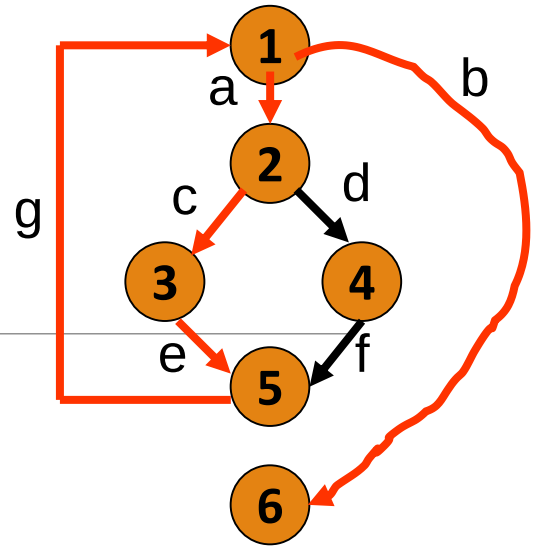
Linearno kombinovanje putanja

Definišemo dve operacije nad putanjama: **sabiranje** i **množenje skalarom**

- Sabiranje dve putanje rezultuje u putanji gde druga putanja sledi prvu
- Množenje odgovara ponavljanju putanje

Skup svih putanja sada predstavlja jedan vektorski prostor

Linearno kombinovanje putanja



Najlakše se obavlja putem matrice incidencije

Grane CFGa / Putanje	a	b	c	d	e	f	g
P1:1-2-3-5-1-6	1	1	1	0	1	0	1
P2:1-2-4-5-1-6	1	1	0	1	0	1	1
P3:1-6	0	1	0	0	0	0	0
P4: P1+P2-P3	2	1	1	1	1	1	2
P5: 2P1-P3	2	1	2	0	2	0	2

Linearno nezavisne putanje

Putanja je **linearno nezavisna** od skupa drugih putanja ako i samo ako:

- Ne može biti predstavljena kao linearna kombinacija putanja iz skupa.

Objašnjenje:

- P1 je nezavisna od {P2, P3} zbog **c**
- P2 je nezavisna od {P1,P3} zbog **d**
- P3 je nezavisna od {P1,P2} jer svaki pokušaj da se izrazi preko njih uvodi neželjene grane

Bazični skup putanja

Skup putanja naziva se **bazičnim** ako i samo ako su

- Putanje u skupu međusobno linearno nezavisne i
- Bilo koja druga putanja u CFGu može biti predstavljena kao linearna kombinacija putanja iz posmatranog skupa

U opštem slučaju može biti više različitih bazičnih skupova za jedan CFG, ali svi oni imaju isti broj elemenata

{P1,P2,P3} je bazični skup u našem primeru.

Broj ciklomatske kompleksnosti V(G)

Daje maksimalni broj linearno nezavisnih putanja u grafu G (karidnalnost bazičnog skupa):

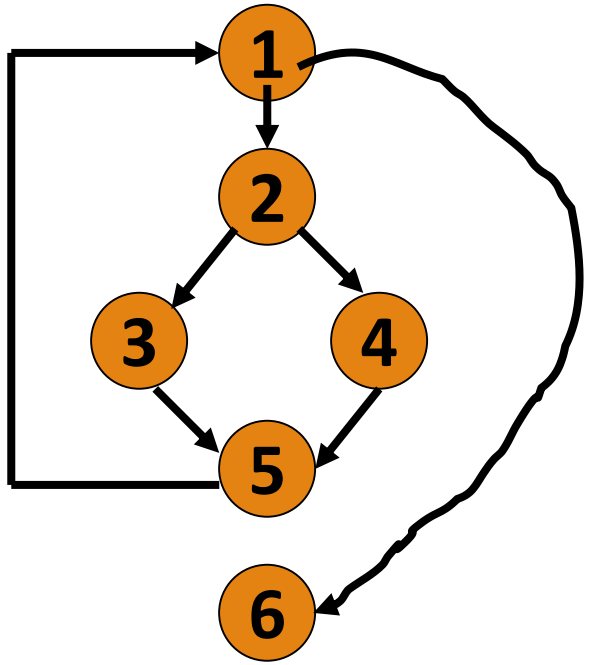
V(G) = e – n + 2 za grafove sa jednim početnim i jednim završnim čvorom

e – broj grana grafa

n – broj čvorova grafa

Primer računanja cikl. kompleksnosti

V(G) = 7 – 6 + 2 = 3.



Računanje ciklomatske kompleksnosti

Drugi način računanja ciklomatske kompleksnosti:

- V(G) = b + 1
- b je broj binarnih odluka u programu (odgovara ranije definisanim uslovima; na primer, if a or b then… ima dve binarne odluke)

Primer – računanje cikl. kompleksnosti

```
int f1(int x,int y){
1.      while (x != y){
2.          if (x>y) then
3.              x=x-y;
4.          else y=y-x;
5.       }
6.       return x;          }
```

- Broj binarnih odluka je 2.
- Ciklomatska kompleksnost = 2+1=3.

Računanje ciklomatske kompleksnosti

Šta kada graf sadrži više od jednog početnog i/ili završnog čvora?

Prvo, transformišemo graf u jako povezani (proizvoljan čvor može se posetiti iz bilo kojeg drugog prateći usmerene grane) povezujući svaki završni čvor sa svakim od početnih.

Potom, upotrebimo formulu V(G)=e-n+2p, gde je broj jako povezanih komponenata grafa p=1

Određivanje test primera

- Nacrtati graf toka kontrole programa
- Izračunati V(G).
- Odrediti bazični skup putanja.
- Pripremiti test primere koji forsiraju izvršavanje programa po svakoj od putanja iz bazičnog skupa.

McCabe-ov Baseline metod

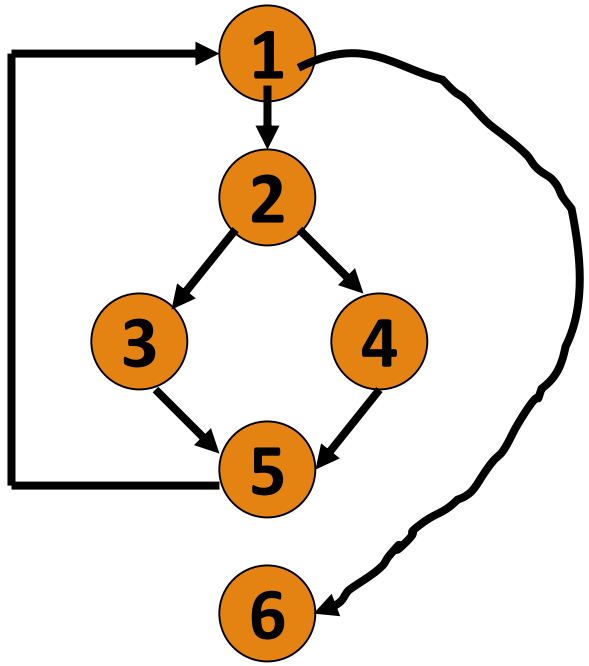
Algoritam za određivanje bazičnog skupa:

1. Odabrati "baseline" putanju, koja treba da odgovara "normalnom slučaju" izvršavanja programa. Izbor putanje je donekle proizvoljan; McCabe savetuje izbor putanje sa što više čvorova odlučivanja (dve i više izlaznih grana).
2. Zatim, prateći prethodno izabranu putanju , pronaći prvi nerazmatran čvor odlučivanja i izabrati izlaznu granu koja nije na putanji, čime se dobija nova putanja.
3. Ponavljati korak 2 dok se ne izabere V(G) putanja.

Određivanje test primera

Broj nezavisnih putanja: 3

- 1, 2, 3, 5, 1, 6 test primer (x=2, y=1)
- 1, 6 test primer (x=1, y=1)
- 1, 2, 4, 5, 1, 6 test primer (x=1, y=2)



Druge primene ciklomatske kompleksnosti

Ciklomatska kompleksnost programa:

- Takođe pokazuje nivo psihološke kompleksnosti programa, odnosno
- Nivo težine razumevanja programa (od strane drugog programera).

Sa stanovišta održavanja softvera,

- Treba ograničiti ciklomatsku kompleksnost svakog modula na neku razumnu vrednost.
- Dobre softverske kompanije:
  - Imaju pravilnike o stilu kodiranja u kome je često ograničena ciklomatska kompleksnost funkcije na maksimalno 10 ili približno tome
  - Postoje alati za proveru poštovanja ovih pravila (npr. Abraxas CodeCheck alat ima bazu pravila koju korisnik može da menja)

Testiranje zasnovano na toku podataka

Tehnike testiranja zasnovane na toku podataka

Metode toka podataka se baziraju na međusobnoj interakciji naredbi kroz zajedničke podatke koje te naredbe koriste.

Motivacija za razvijanje ovih tehnika leži u činjenici da se prostim pokrivanjem iskaza i odluka u kodu mogu prevideti neke greške.

Ukoliko je izračunata vrednost u jednoj naredbi pogrešna, greška se može otkriti tek kada neka druga naredba pokuša da iskoristi tu vrednost.

Prilikom odabira putanja se posmatraju lokacije dodele vrednosti promenljivama i lokacije njihove upotrebe.

Programske putanje se selektuju za testiranje :

- Vodeći računa o lokacijama dodele promenljivama i lokacijama upotrebe istih.

Osnovni pojmovi

Za iskaz S, definišemo 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}.

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**).

Osnovni pojmovi – život promenljive

Za promenljivu X kaže se da je **živa** u iskazu S1, akko

- X je definisana u nekom iskazu S i
- postoji putanja od S do S1 koja ne sadrži novu dodelu promenljivoj X.

Lanac dodele-upotrebe (DU lanac)

Označava se sa [X, S, S1], gde su

- S i S1 iskazi, a X promenljiva tako da važi
- $X \in$ DEF(S) i
- $X \in$ USE(S1) i
- X ima dodelu u iskazu S koja je živa u iskazu S1.

Primer DU lanaca

```
// Is_Complex is true if the roots are not real.
// If the two roots are real, they are produced in R1, R2.
void Solve_Quadratic (float A, float B, float C) {          // 0
        float Discrim = B*B - 4.0*A*C;                      // 1
        bool Is_Complex;                                    // 2
        float R1, R2;                                       // 3
        if (Discrim < 0.0) {                                // 4
            Is_Complex = true;                              // 5
        } else {                                            // 6
            Is_Complex = false;                             // 7
        }                                                   // 8
        if (!Is_Complex) {                                  // 9
            R1 = (-B + sqrt(Discrim)) / (2.0*A);            // 10
            R2 = (-B - sqrt(Discrim)) / (2.0*A);            // 11
        }                                                   // 12
}                                                           // 13
```

Primer: definicije i upotrebe

line	category		
	definition	c-use	p-use
0	A,B,C		
1	Discrim	A,B,C	
2			
3			
4			Discrim
5	Is_Complex		
6			
7	Is_Complex		
8			
9			Is_Complex
10	R1	A,B,Discrim	
11	R2	A,B,Discrim	
12			
13			

```
// Is_Complex is true if the roots are not real.
// If the two roots are real, they are produced in R1, R2.
void Solve_Quadratic (float A, float B, float C) {      // 0
        float Discrim = B*B - 4.0*A*C;                  // 1
        bool Is_Complex;                                // 2
        float R1, R2;                                   // 3
        if (Discrim < 0.0) {                            // 4
            Is_Complex = true;                          // 5
        } else {                                        // 6
            Is_Complex = false;                         // 7
        }                                               // 8
        if (!Is_Complex) {                              // 9
            R1 = (-B + sqrt(Discrim)) / (2.0*A);        // 10
            R2 = (-B - sqrt(Discrim)) / (2.0*A);        // 11
        }                                               // 12
}                                                       // 13
```

Primer: DU lanci

definition-use pair (start line -> end line)	variable(s)	
	c-use	p-use
0 ---> 1	A,B,C	
0 ---> 10	A,B	
0 ---> 11	A,B	
1 ---> 4		Discrim
1 ---> 10	Discrim	
1 ---> 11	Discrim	
5 ---> 9		Is_Complex
7 ---> 9		Is_Complex

```
// Is_Complex is true if the roots are not real.
// If the two roots are real, they are produced in R1, R2.
void Solve_Quadratic (float A, float B, float C) {      // 0
        float Discrim = B*B - 4.0*A*C;                  // 1
        bool Is_Complex;                                // 2
        float R1, R2;                                   // 3
        if (Discrim < 0.0) {                            // 4
            Is_Complex = true;                          // 5
        } else {                                        // 6
            Is_Complex = false;                         // 7
        }                                               // 8
        if (!Is_Complex) {                              // 9
            R1 = (-B + sqrt(Discrim)) / (2.0*A);        // 10
            R2 = (-B - sqrt(Discrim)) / (2.0*A);        // 11
        }                                               // 12
}                                                       // 13
```

Testiranje zasnovano na toku podataka

Različite strategije zahtevaju pokrivanje različitih DU lanaca. Testovi se generišu da postignu 100% pokrivenost za svaki od kriterijuma ako je moguće.

Realna pokrivenost izražava se formulom:

Coverage = (N/T) * 100%

gde je T broj lanaca po nekom kriterijumu (utvrđuje se statičkom analizom programa), a N je broj tih lanaca koje su testovi zaista pokrili.

Strategije testiranja zasnovanog na toku podataka

Prema radu *Rapps-Weyuker* imamo sledeće strategije pokrivanja:

- Sve definicije
- Sve upotrebe
- Sve p-upotrebe
- Sve c-upotrebe
- Sve c-upotrebe, neke p-upotrebe
- Sve p-upotrebe, neke c-upotrebe
- Svi du-lanci

Sve definicije

100% pokrivenost se postiže ako se izvrši bar po jedan du lanac od svake definicije svake promenljive do neke od upotreba (bilo p-use bilo c-use)

Primer

	All Definitions			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
3	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
4	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

Sve c-upotrebe

100% pokrivenost se postiže ako se izvrši bar po jedan du lanac od svake definicije do svih c upotreba te definicije

Primer

	All-c-uses			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	A,B	0 --> 10	0-1-4-7-9-10	1	2	1	F	-1	-1
3	A,B	0 --> 11	0-1-4-7-9-10-11	1	2	1	F	-1	-1
4	Discrim	1 --> 10	1-4-7-9-10	1	2	1	F	-1	-1
5	Discrim	1 --> 11	1-4-7-9-10-11	1	2	1	F	-1	-1

Sve c-upotrebe, neke p-upotrebe

100% pokrivenost se postiže ako se izvrši bar po jedna du lanac od svake definicije do svih c upotreba te definicije; ako definicija nema c-upotreba, mora se pokriti bar jedan du lanac sa p-upotrebom.

Primer

- Pored du lanaca iz svih c upotreba, treba pokriti i jedan du lanac za p-upotrebu promenljive Is_Complex:

				INPUTS			EXPECTED OUTCOME		
test case	variable(s)	d-u pair	subpath	A	B	C	Is_Complex	R1	R2
8	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

Sve p-upotrebe

100% pokrivenost se postiže ako se izvrši bar po jedan du lanac od svake definicije do svih p upotreba te definicije

Primer

	All-p-uses			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
2	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
3	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

Sve p-upotrebe, neke c-upotrebe

100% pokrivenost se postiže ako se izvrši bar po jedan du lanac od svake definicije do svih p upotreba te definicije; ako definicija nema p-upotreba, mora se pokriti bar jedan du lanac sa c-upotrebom.

Primer

- Pored du lanaca iz svih p upotreba, treba pokriti i po jedan du lanac za c-upotrebe promenljivih A,B,C:

				INPUTS			EXPECTED OUTCOME		
test case	variable(s)	du-pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.

Sve upotrebe

100% pokrivenost se postiže ako se izvrši bar po jedan du lanac od svake definicije do svih upotreba (i p-use i c-use) te definicije

Primer

	All-uses / All du-paths			INPUTS			EXPECTED OUTCOME		
test case	variable(s)	d-u pair	subpath	A	B	C	Is_Complex	R1	R2
1	A,B,C	0 --> 1	0-1	1	1	1	T	unass.	unass.
2	A,B	0 --> 10	0-1-4-7-9-10	1	2	1	F	-1	-1
3	A,B	0 --> 11	0-1-4-7-9-10-11	1	2	1	F	-1	-1
4	Discrim	1 --> 4	1-4	1	1	1	T	unass.	unass.
5	Discrim	1 --> 10	1-4-7-9-10	1	2	1	F	-1	-1
6	Discrim	1 --> 11	1-4-7-9-10-11	1	2	1	F	-1	-1
7	Is_Complex	5 --> 9	5- 9	1	1	1	T	unass.	unass.
8	Is_Complex	7 --> 9	7- 9	1	2	1	F	-1	-1

Sve DU putanje

100% pokrivenost se postiže ako se izvrši svaka ‘jednostavna’ putanja od svake definicije do svih upotreba (i p-use i c-use) te definicije

‘jednostavna’ putanja u grafu toka kontrole je putanja kod koje se svaki njen deo posećuje minimalan broj puta (npr. otvorene putanje, jedna iteracija kroz petlju).

Za posmatrani primer, postoje dve jednostavne putanje van onih koje pokrivaju testovi za ’sve upotrebe’. To su:

0-1-4-5-9-10 i 1-4-5-9-10

Medjutim, nije ih moguće pokriti testovima.

Testiranje zasnovano na toku podataka

Strategije zasnovane na toku podataka:

- Korisne za izbor test putanja programa koji poseduju ugneždene if-ove i petlje