DIETI UNINA UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Corso di Ingegneria del Software

Test Strutturale

Sommario

- Control Flow Graph (CFG) e Call Graph (CG)
- Copertura degli Statement
- Copertura delle decisioni
- Copertura delle condizioni
- Copertura MC/DC
- Copertura dei cammini base
- Test dei cicli

Riferimenti

M.Pezzè, M. Young; *Software Testing and Analysis. Process, principles, and techniques*.

Cap. 12

Test Strutturale

- Valutare la completezza della test suite in base alla **struttura** del programma stesso
  - Noto anche come “White-box”, “Glass-box”, “code-based”
  - Si distingue dal functional (“black-box”) testing:
    - *La **sorgente di informazione** per derivare i casi di test è la struttura del codice anziché le specifiche*
    - *La misura di **completezza della test suite** cambia (basata sulla copertura di elementi della struttura anziché delle specifiche)*
    - Ciò implica l’utilizzo di tecniche diverse per derivare i casi di test

Perchè il test strutturale?

- Cosa manca ad una test suite derivata dal test funzionale?
  - Se parte di un programma non è eseguita da alcun caso di test, i difetti in quella parte non possono essere rilevati
  - Ma quale parte? Tipicamente, un elemento del *control flow*, o una combinazione di elementi
- Complementa il test funzionale
  - Caso tipico: implementazione di un elemento delle specifiche da più parti del programma
  - Esempio: collisioni nelle tabelle Hash (invisible nelle spec.)

Perché non basta solo il test funzionale?

- Per la tipologia degli errori, alcuni dei quali non possono essere rilevati dal testing black-box
  - Gli errori logici e le assunzioni scorrette sono inversamente proporzionali alla probabilità che un *path* sia eseguito. C'è maggior probabilità di commettere errori nell'elaborazione dei "casi speciali"
  - Spesso riteniamo che un *path* non abbia molte probabilità di essere eseguito, ma in realtà esso è eseguito regolarmente. Ciò può comportare errori di progettazione rilevati solo quando si collauda quel *path*
  - Errori banali (es. di digitazione) che non sono rilevati in compilazione possono essere in un flusso secondario

Test Strutturale

- Sono esaminati i dettagli procedurali
- Viene svolto un test dei cammini logici interni
  - *Test-case* che esercitano insiemi specifici di condizioni e/o cicli
- Difficoltà di eseguire un testing esaustivo di tutti i cammini, che può essere enormemente elevato anche per piccoli programmi

Esempio E1

```
Repeat
  B0
  if R1 then
   if R2 then
      if R3 then
            B1
       else
            B2
      endif
    else
      if R4 then
           B3
      else
           B4
      endif
   else B5
   endif
  endif
until R6
```

Se il ciclo viene eseguito max 20 volte si possono avere oltre 100.000 miliardi di cammini possibili

3.170 anni nell'ipotesi che ciascun test case sia elaborato in 1ms

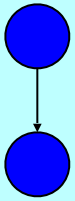
Rappresentazioni grafiche del codice

Control Flow Graph (CFG)

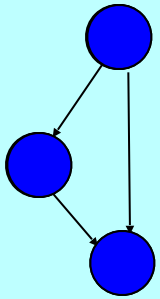
- Grafo orientato che rappresenta il trasferimento del flusso di controllo tra blocchi di istruzioni
- Un **nodo** rappresenta un blocco di 1 o più istruzioni, tutte eseguite se il flusso di controllo raggiunge la prima di esse
- Un **arco** rappresenta il trasferimento del flusso di controllo tra la coppia di nodi che esso collega
- Ogni nodo rappresenta una componente 1-in/1-out, in cui è possibile uno e solamente un cammino di controllo aciclico: nessuna decisione, né congiunzione di flussi, può interrompere la sequenza delle istruzioni incluse nel nodo
  - *Solo alla fine può esservi una ramificazione*
  - *Solo all'inizio può esservi una congiunzione*

CFG per le istruzioni di controllo di flusso

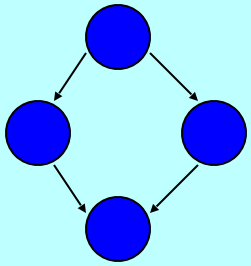
Sequenza



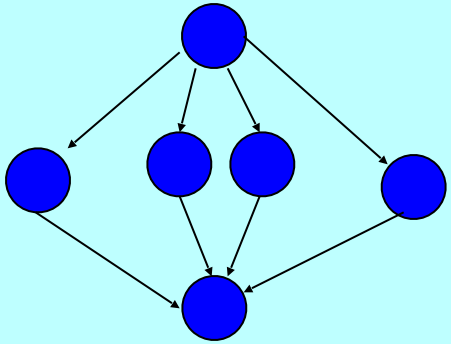
If..then



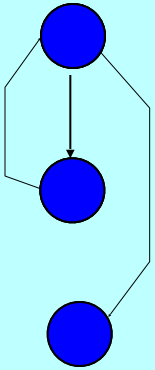
If..then..else



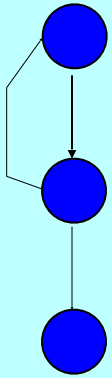
Case



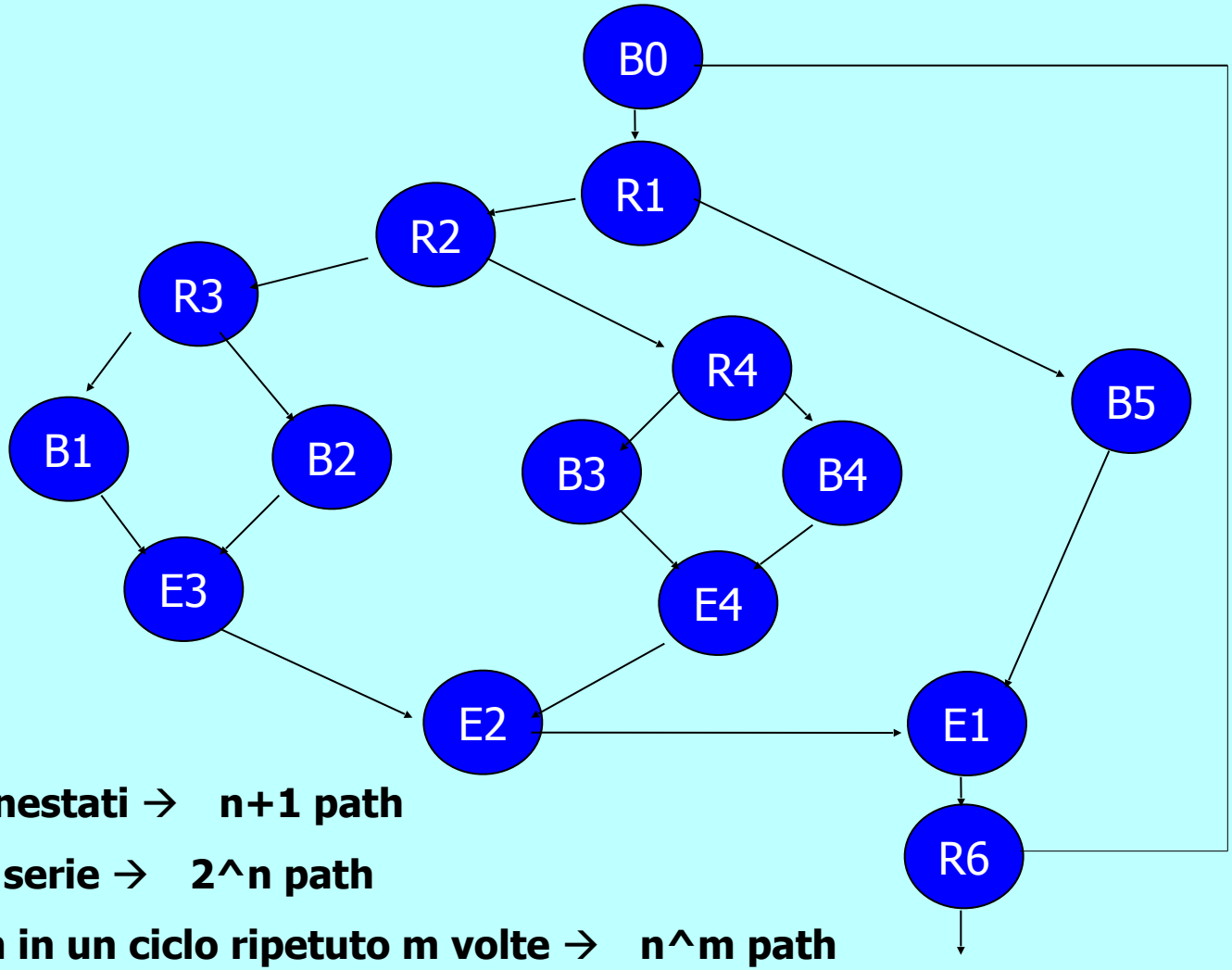
While..do



Repeat..Until



CFG per l'esempio E1



n if innestati → n+1 path

n if in serie → 2^n path

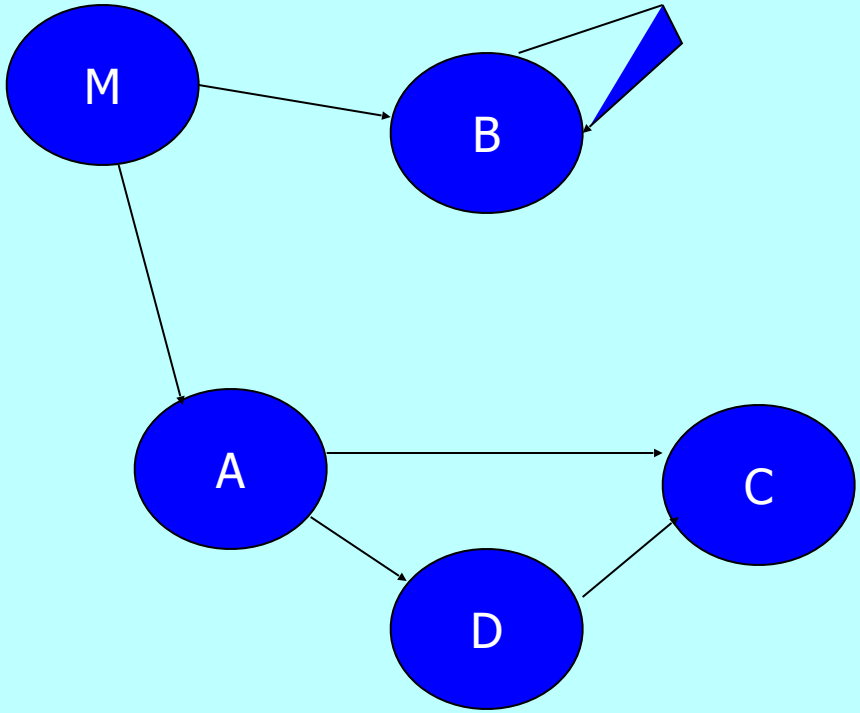
n path in un ciclo ripetuto m volte → n^m path

Call-Graph

Call Graph (CG)

- Grafo orientato che rappresenta le relazioni chiamante-chiamato tra procedure
- I **nodi** rappresentano le procedure e gli archi le relazioni di chiamata tra esse
- Un **arco** è rappresentativo di tutte le chiamate che possono sussistere tra 2 nodi (nel caso si usi un arco per ogni chiamata si ha un *multi-call graph*)
- Vi sarà un nodo che non ha archi entranti, che corrisponde il programma principale

Call-Graph



Test Strutturale nella pratica

- I casi di test sono **derivati a partire dagli elementi strutturali** (per es. del CFG), non dalle specifiche
- La creazione è guidata da una misura di ***coverage*** che identifica quanti elementi sono stati eseguiti
- Gli elementi non eseguiti possono essere dovuti a:
  - Naturali differenze tra specifica ed implementazione
  - Codifica che diverge radicalmente dalla specifica
  - Test suite inadeguata
- Vantaggiosi perché automatizzati
  - La *coverage* è un conveniente indicatore di come procede il test
  - A volte usata come criterio di completamento
    - Da usare con cautela: non assicura una test suite efficiente

Progettazione dei test-case

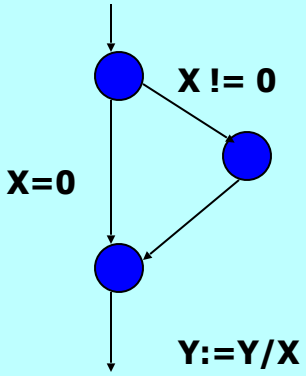
- Progettare test-case in modo che:
  - Tutti i cammini indipendenti all'interno di un modulo siano stati esercitati almeno una volta
  - Siano esercitate tutte le decisioni logiche nei casi vero e falso
  - Eseguano tutti i cicli ai loro valori limite ed all'interno dei valori ammessi
  - Siano esercitate tutte le strutture dati per assicurare la validità

Copertura degli statement

- ***Adequacy Criterion*: Richiede che ogni istruzione sia eseguita almeno una volta**

Esempio:

```
if (x!=0)
 then S1
y=y/x
```



Attenzione:

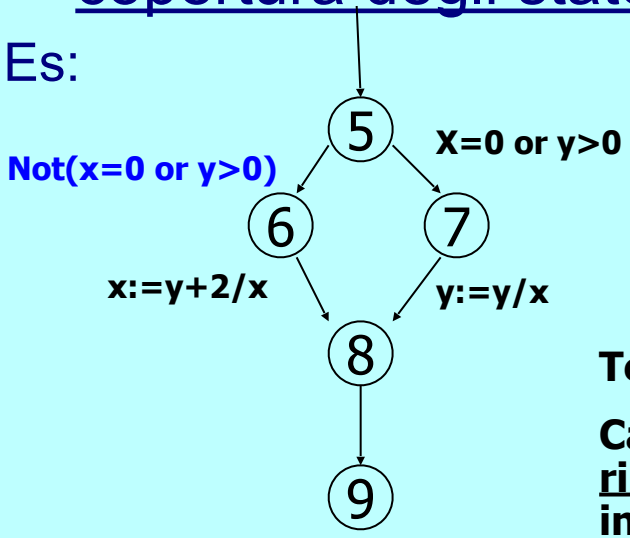
Un test-case che assicuri la copertura degli statement non sempre comporta l'esecuzione del programma per tutti i valori delle decisioni (sia il ramo *true* sia quello false)

Misura di Coverage: **(#Statement Eseguiti) / (#Statement)**

Copertura delle decisioni

- *Coprire tutti gli statement non implica eseguire tutti i rami.*
- ***Adequacy Criterion:*** **Richiede che ogni arco del control-flow graph sia percorso ameno una volta**
- Un test dovrà contenere almeno 2 dati di test per ciascuna decisione, uno per ogni ramo uscente da un nodo decisione
- Il criterio di copertura delle decisioni include quello di copertura degli statement

Es:




```
if (x=0 or y>0)
    then y:=y/x
     else x:=y+2/x
```

Misura di Coverage:
(#Archi Eseguiti) / (#Archi)

Test-Case B= {(x=5,y=5),(x=-5,y=-5)}

Causa l'esecuzione di entrambi gli archi. Però può non rilevare il malfunzionamento dovuto a divisione per 0, infatti B non copre entrambi valori delle 2 condizioni in OR

Copertura delle condizioni

- Criterio di copertura delle condizioni:
- ***Adequacy Criterion*: Richiede che ogni singola condizione che compare nelle decisioni del programma valga sia vero che falso per diversi dati di test**

Es: if (x=0 or y>0)

Un possibile test-case è:

C={(x=5,y=5), (x=0,y= -5)}

Infatti:

(x=5,y= 5)→ la cond x=0 è False e y>0 è True

(x=0,y=-5)→ la cond x=0 è True e y>0 è False

Misura di Coverage: **(#Valori di Verità presi da tutte le condizioni singole)**

(2*#Condizioni singole)

Attenzione:

Un test che soddisfa il criterio di copertura delle condizioni non è detto che soddisfi quello delle decisioni.

Nell'esempio, per C la condizione è sempre e solo vera

→ **Criterio di copertura delle decisioni e delle condizioni**

Copertura delle decisioni e delle condizioni

- Copertuare delle decisioni e delle condizioni
- ***Adequacy Criterion*: Richiede che ogni decisione valga sia vero sia falso ed ogni singola condizione che compare nelle decisioni del programma valga sia vero sia falso per diversi dati di test**

Es: if (x=0 or y>0)
Un possibile test-case è:
C={(x=0,y=5), (x=5,y= -5)}
Infatti:
(x=0,y= 5)→ la cond x=0 è True e y>0 è True
(x=5,y=-5)→ la cond x=0 è False e y>0 è False

Il criterio di copertura delle decisioni e delle condizioni contiene sia il criterio di copertura delle condizione che il criterio di copertura delle decisioni

Copertura delle Condizioni e delle Decisioni

- Decisioni e Condizioni
  - Coprire tutte le condizioni e tutti gli archi
- Copertura delle **condizioni Composte**
  - Copre tutte le possibili condizioni composte (2^N)
  - Copre tutti gli archi (le decisioni)
  - Estremamente costoso

Costi del test

- Copertura degli statement :
  - ***Numero di casi di test richiesti =1***
- Copertura delle decisioni
  - ***Numero di casi di test richiesti= 2***
- Copertura delle condizioni
  - ***Numero di casi di test richiesti= 2***
- Copertura delle condizioni e decisioni
  - ***Numero di casi di test richiesti= 2***
- Copertura delle condizioni composte
  - ***Numero di casi di test richiesti= 4***

Modified Condition/Decision (MC/DC)

- Il criterio delle condizioni composto causa una crescita esponenziale del numero di casi di test
- MC/DC: Testare combinazioni rilevanti di condizioni, evitando una crescita esponenziale.
  - Rilevanti significa: ogni condizione singola che influenza indipendentemente l'esito di una decisione
- Richiede per ogni condizione C due casi di test:
  - I valori di tutte le condizioni valutate tranne C sono gli stessi
  - La condizione composta vale *true* per un caso di test e *false* per l'altro

Modified Condition/Decision (MC/DC)

- Complessità Lineare: N+1 casi di test per N condizioni singole

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

Pezzè & Young, Software Testing And Analysis: Process, Principles And Techniques, 2008

- a decisione
- Questo criterio è richiesto dallo standard avionico DO-178B

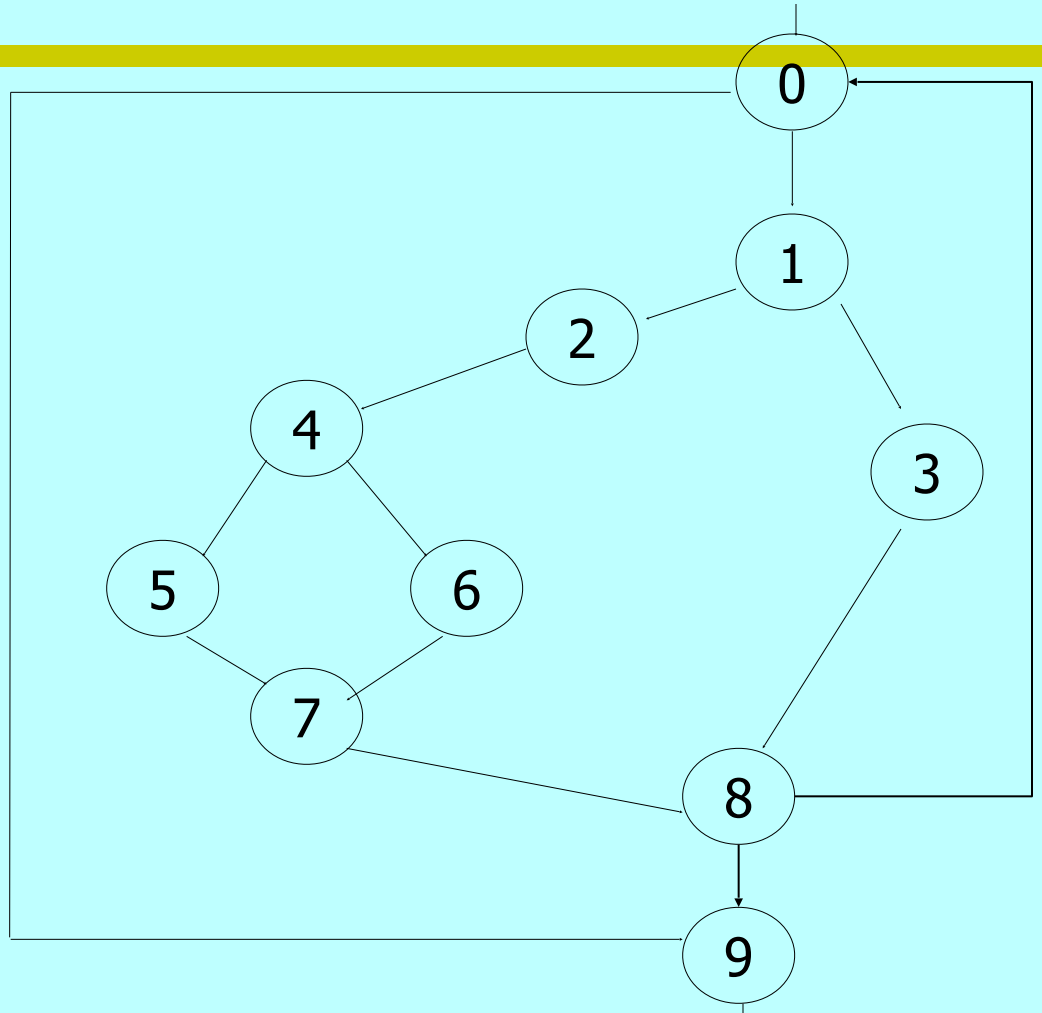
Modified Condition/Decision (MC/DC)

- MC/DC è:
  - Coverage delle condizioni singole (C)
  - Coverage delle decisioni (DC)
  - Più una condizione (M)
    - *Ogni condizione deve influenzare indipendentemente l'output della decisione*
  - La copertura del criterio MC/DC è "implicata" da quella del criterio delle condizioni composte e implica tutti gli altri
    - Più forte del criterio delle decisioni e degli statement
  - Dunque è un buon compromesso tra completezza e numero di casi di test (e per questo ampiamente usato)

Copertura dei cammini di base

- Una misura della complessità logica è usata come guida per definire un insieme di base per l'esecuzione dei cammini
- I test case effettuati sull'insieme di base garantiscono l'esecuzione di ciascuna istruzione almeno una volta
- La **complessita ciclomatica di Mc Cabe**:
  - *definisce il numero dei cammini indipendenti nell'insieme di base di un programma; rappresenta il limite superiore del numero di test da effettuare*
- Cammino indipendente:
  - Qualunque cammino in un programma che introduce almeno un nuovo gruppo di istruzioni o una nuova condizione (cioè deve, in un CFG, percorrere un arco non considerato in precedenza)
- *Adequacy Criterion* dei cammini di base:
  - Richiede che ciascun cammino di un insieme di cammini di base sia eseguito almeno una volta

Esempio E2



Cammino1: 0-9

Cammino2: 0-1-2-4-5-7-8-9

Cammino3: 0-1-2-4-6-7-8-9

Cammino4: 0-1-3-8-9

Cammino5: 0-1-3-8-0-1-3-8-9

= Insieme di Base

Test che esercitano i cinque cammini indicati eserciteranno ciascuna istruzione almeno una volta sia per un valore "vero" che "falso"

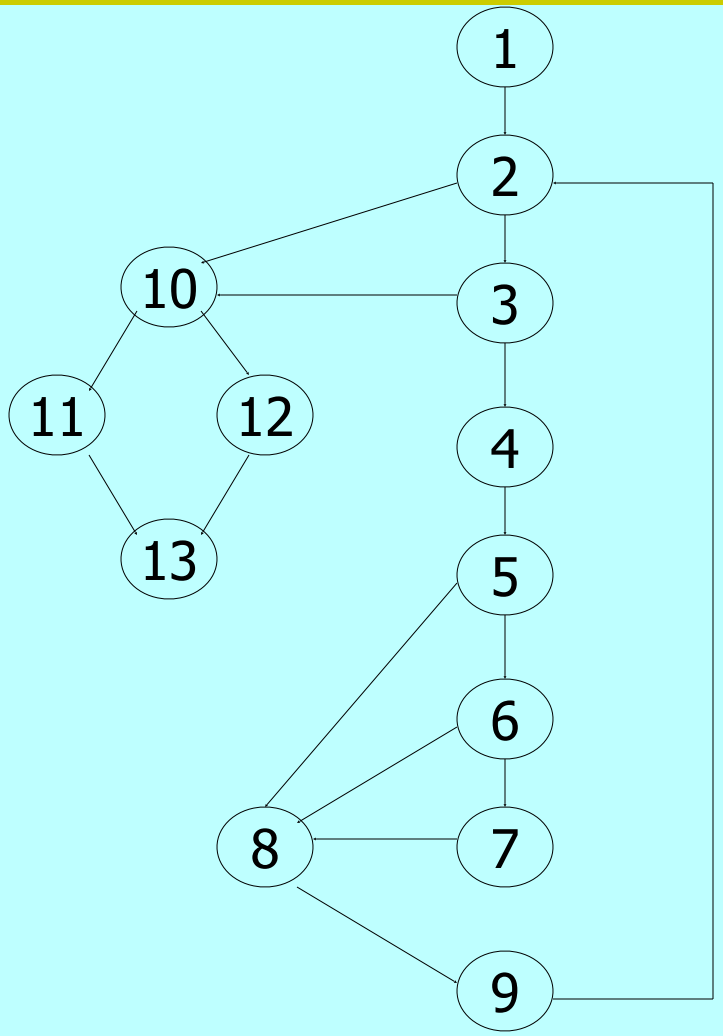
Numero ciclomatico di Mc Cabe

- L'insieme di base non è unico: data una progettazione procedurale, possono essere derivati diversi insiemi di base
- La complessità ciclomatica indica quanti cammini devono essere cercati
- Il **numero ciclomatico di Mc Cabe** è dato da una delle seguenti espressioni:
  1) V(g)= Numero di regioni del grafo
  2) V(g)= F-N+2*C dove F=archi e N=nodi
  3) V(g)= P+1 dove P=nodi predicati

Esempio E3 – 1/4

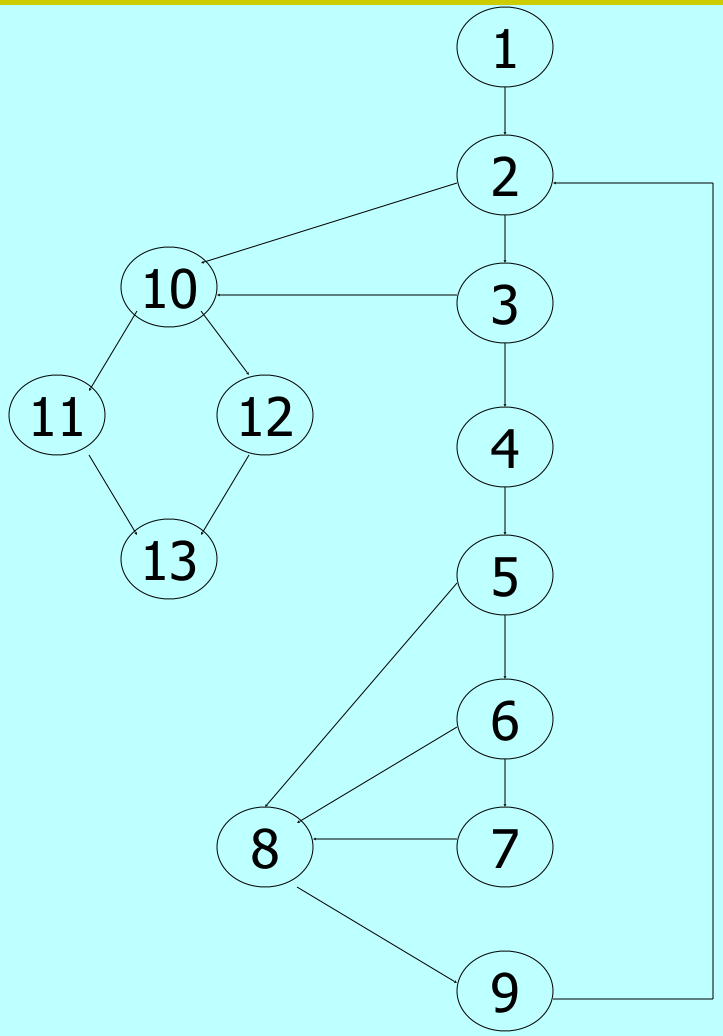
```
Procedura media (valore: vet; minimo, massimo: integer; var,….,Totale.input, Totale.valido: integer; var media:
      real);
var i :integer, somma :integer
   begin
       i:=1
       Totale.input:=0
       Totale.valido:=0;
       somma:=0
       while (valore[i]<>-999 and Totale.input<100) do
                    begin
                        Totale.input:=Totale.input+1;
                            if(valore[i]>=minimo and valore[i]<=massimo)
                                then  begin
                                          Totale.valido:=Totale.valido+1;
                                          somma:=somma+Valore[i];
                                        end;
                            i:=i+1;
                   end;
        if Totale.valido>0
        then media:= somma/Totale.valido
        else media:=-999;
 end;
```

Esempio E3 – 2/4



N.B. Questo CFG considera le singole condizioni in nodi separati

Esempio E3 – 2/4



V(G)=6

C1: 1-2-10-11-13

C2: 1-2-10-12-13

C3: 1-2-3-10-11-13

C4: 1-2-3-4-5-8-9-2……

C5: 1-2-3-4-5-6-8-9-2…..

C6: 1-2-3-4-5-6-7-8-9-2….

Esempio E3 – 3/4

- Test-Case per C1

  Valore(K)= dato ingresso valido, se k<i

  Valore(i)=-999, 2<=i<=100

Risultati attesi: media corretta sulla base di n valori e totali appropriati

- Test-case per C2

  valore(i)= -999

  Risultati attesi : Media= -999

- Test-Case per C3

Tentativo di esecuzione per più di 100 valori, di cui i primi 100 validi

Risultati attesi: come per c1

Esempio E3 – 4/4

- Test-Case per C4

  Valore(i) = valido ; i<100

  Valore(K) < minimo ; K<i

Risultati attesi: media corretta su i-k valori e Totali appropriati

- Test-Case per C5

Valore(i) = valido ; i<100

Valore(K) > massimo ; K<i

Risultati attesi: media corretta su i-k valori e totali appropriati

- Test-Case per C6

  Valore (i)=Valido ; i<100

  Risultati attesi: Media corretta sulla base di n valori

Ancora testing

- Test di validità dei costrutti dei cicli:
  - Semplici
  - Concatenati
  - Nidificati
  - Non strutturati
- Oltre a quelli dei cammini di base, effettua altri test al fine di rilevare errori di:
  - Inizializzazione
  - Indicizzazione, o incremento
  - Per valori limite

Testing dei cicli

- Cicli semplici
  1) Saltare completamente
  2) Eseguire 1 sola volta
  3) Eseguire 2 volte
  4) Eseguire m volte, m<n (max iterazioni)
  5) Eseguire n-1, n, n+1 volte

Testing dei cicli

Cicli nidificati

Con l'approccio precedente il numero di test aumenterebbe in progressione geometrica

Altro approccio:

1) *Si inizia dal ciclo più interno, fissando gli altri ai valori minimi*
2) *Per il ciclo più interno eseguire i test per i cicli semplici, tenendo gli altri al parametro minimo di iterazione. Si aggiungono altri test con valori esterni all'intervallo di validità per i parametri dei cicli esterni*
3) *Si passa al ciclo di livello immediatamente più esterno, mantenendo i cicli più esterni ai valori minimi e quelli più interni ai valori tipici*
4) *Continuare finché non siano testati tutti i cicli*

Testing dei cicli

- Cicli concatenati
  - Come per i semplici se indipendenti
  - Come per i nidificati se dipendenti (es. il contatore di ciclo del primo è il valore di partenza per il successivo)

- Cicli non strutturati

  Riprogettarli strutturati

Conclusioni

- Sono stati presentati i criteri di adeguatezza (*adequacy criteria*)
  - Ma non si è visto come derivare i casi di test
  - Esempi di tecniche sono quelle basate su esecuzione simbolica o su algoritmi "*evolutionary*" (*search-based software testing*)
- Criteri diversi rilevano diverse classi di fault
- La ***coverage*** piena difficilmente è raggiungibile …
- … e quando lo è:
  - Come trovo gli input del programma che permettono di esercitare *codice morto*? Sono necessari strumenti di supporto automatico
- Piuttosto che richiedere l'adeguatezza piena, è stimato il "grado di adeguatezza" di una test suite dalle misure di *coverage*
  - Può guidare miglioramenti del test